craftinginterpreters_zh

[在线阅读]

手撸解释器教程《Crafting Interpreters》中文翻译。

这是一个已基本完成的翻译项目，原项目是craftinginterpreters，同时还有配套的英文书，可免费在线阅读。

该书由一门小型的自创语言Lox开始，分别使用Java和C实现了两种类型的解释器，jlox和clox，其中前者是将语法解析成Java中的表示代码，主要依赖Java本身的语法能力实现代码的真正运行；后者则采用了类似编译和虚拟机的机制，实现了一个看上去“更高效”的解释器。

该书中并不是只灌输概念或者只贴出代码，相反，作者经过悉心拆解解释器相关的概念，在每一章节中逐步进行讲解和实现，并且保证每个章节结束之后都有一个可以运行的解释器版本。内容上不会有太过深入的理论，对于普通编程爱好者也可以很容易地上手学习。

如果您的英语阅读能力比较强，建议直接阅读原文，同时也欢迎您参与本项目，分享自己学习的成果，帮助更多的人。

囿于本人能力限制，在译文难免有错漏或者含混之处，请积极留言，我会悉心听取大家的意见，争取可以让这篇译文帮助到更多人。

如果您觉得这篇译文对您的学习有所帮助，可以奖励作者一杯奶茶:)

特别鸣谢

感谢xieyuheng提供的在线阅读支持，为需要在线阅读markdown文档的同学推荐read-only项目，样式简约大方，值得一试。

1. 前言 Introduction

> Fairy tales are more than true: not because they tell us that dragons exist, but because they tell us that dragons can be beaten.

> —— Neil Gaiman, *Coraline*

童话故事是无比真实的：不是因为它告诉我们龙的存在，而是因为它告诉我们龙可以被击败。

> I'm really excited we're going on this journey together. This is a book on implementing interpreters for programming languages. It's also a book on how to design a language worth implementing. It's the book I wish I had when I first started getting into languages, and it's the book I've been writing in my head for nearly a decade.

我真的很兴奋我们能一起踏上这段旅程。这是一本关于为编程语言实现解释器的书。它也是一本关于如何设计一种值得实现的语言的书。我刚开始接触编程语言的时候就希望我可以写出这本书，这本书我在脑子里已经写了将近十年了。

> In these pages, we will walk step by step through two complete interpreters for a full-featured language. I assume this is your first foray into languages, so I'll cover each concept and line of code you need to build a complete, usable, fast language implementation.

在本书中，我们将一步一步地介绍一种功能齐全的语言的两个完整的解释器实现。我假设这是您第一次涉足编程语言，因此我将介绍构建一个完整、可用、快速的语言所需的每个概念和代码。

> In order to cram two full implementations inside one book without it turning into a doorstop, this text is lighter on theory than others. As we build each piece of the system, I will introduce the history and concepts behind it. I'll try to get you familiar with the lingo so that if you ever find yourself in a cocktail party full of PL (programming language) researchers, you'll fit in.

为了在一本书中塞进两个完整的实现，而且避免这变成一个门槛，本文在理论上比其他文章更轻。在构建系统的每个模块时，我将介绍它背后的历史和概念。我会尽力让您熟悉这些行话，即便您在充满PL（编程语言）研究人员的鸡尾酒会中，也能快速融入其中。

> But we're mostly going to spend our brain juice getting the language up and running. This is not to say theory isn't important. Being able to reason precisely and formally about syntax and semantics is a vital skill when working on a language. But, personally, I learn best by doing. It's hard for me to wade through paragraphs full of abstract concepts and really absorb them. But if I've coded something, run it, and debugged it, then I *get* it.

但我们主要还是要花费精力让这门语言运转起来。这并不是说理论不重要。在学习一门语言时，能够对语法和语义进行精确而公式化的推理^1是一项至关重要的技能。但是，就我个人而言，我在实践中学习效果最好。对我来说，要深入阅读那些充满抽象概念的段落并真正理解它们太难了。但是，如果我（根据理论）编写了代码，运行并调试完成，那么我就明白了。

> That's my goal for you. I want you to come away with a solid intuition of how a real language lives and breathes. My hope is that when you read other, more theoretical books later, the concepts there will firmly stick in your mind, adhered to this tangible substrate.

这就是我对您的期望。我想让你们直观地理解一门真正的语言是如何生活和呼吸的。我希望当你以后阅读其他理论性更强的书籍时，这些概念会牢牢地留在你的脑海中，依附于这个有形的基础之上。

> ## 1.1 Why Learn This Stuff？

1.1 为什么要学习这些？

> Every introduction to every compiler book seems to have this section. I don't know what it is about programming languages that causes such existential doubt. I don't think ornithology books worry about justifying their existence. They assume the reader loves birds and start teaching.

每本编译器书籍的导言似乎都有这一部分。我不知道到底是编程语言的哪一点让人产生这样的质疑。我不认为鸟类学书籍会担心如何证明自己的存在。它们假定读者喜欢鸟类，然后开始教学。

> But programming languages are a little different. I suppose it is true that the odds of any of us creating a broadly successful general-purpose programming language are slim. The designers of the world's widely-used languages could fit in a Volkswagen bus, even without putting the pop-top camper up. If joining that elite group was the *only* reason to learn languages, it would be hard to justify. Fortunately, it isn't.

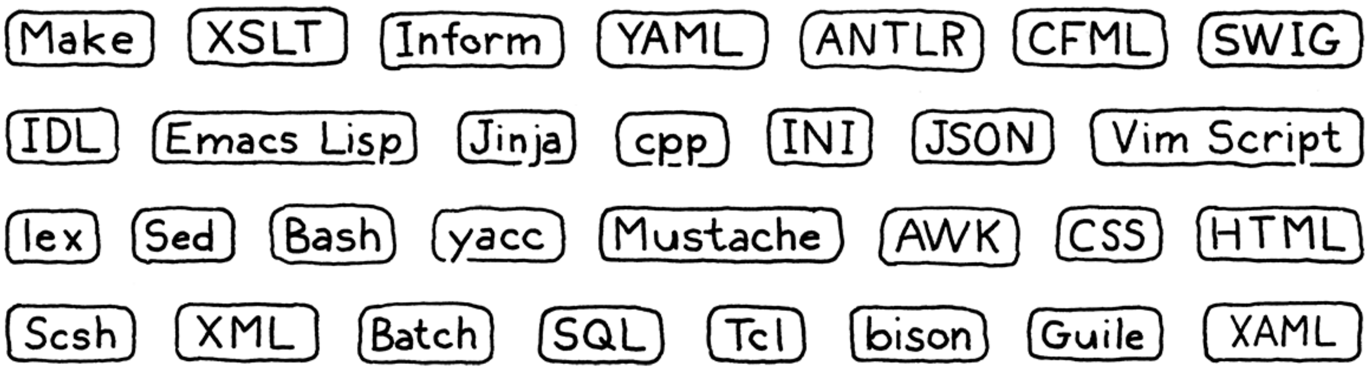
但是编程语言有一点不同。我认为，对我们中的任何一个人来说，能够创建一种广泛成功的通用编程语言的可能性都很小，这是事实。设计这个世界上被广泛使用的语言的设计师们，一辆大众旅游巴士就能装得下，甚至不用把顶上的帐篷加上。如果加入这个精英群体是学习语言的*唯*一原因，那么就很难证明其合理性。幸运的是，事实并非如此。

> ### 1.1.1 Little languages are everywhere

1.1.1 小型语言无处不在

> For every successful general-purpose language, there are a thousand successful niche ones. We used to call them "little languages", but inflation in the jargon economy led today to the name "domain-specific languages". These are pidgins tailor-built to a specific task. Think application scripting languages, template engines, markup formats, and configuration files.

对于每一种成功的通用语言，都有上千种成功的小众语言。我们过去称它们为"小语言"，但术语泛滥的今天它们有了"领域特定语言（即DSL）"的名称。这些是为特定任务量身定做的洋泾浜语言^2，如应用程序脚本语言、模板引擎、标记格式和配置文件。



> Almost every large software project needs a handful of these. When you can, it's good to reuse an existing one instead of rolling your own. Once you factor in documentation, debuggers, editor support, syntax highlighting, and all of the other trappings, doing it yourself becomes a tall order.

几乎每个大型软件项目都需要一些这样的工具。如果可以的话，最好重用现有的工具，而不是自己动手实现。一旦考虑到文档、调试器、编辑器支持、语法高亮显示和所有其他可能的障碍，自己实现就成了一项艰巨的任务。

> But there's still a good chance you'll find yourself needing to whip up a parser or something when there isn't an existing library that fits your needs. Even when you are reusing some existing implementation, you'll inevitably end up needing to debug and maintain it and poke around in its guts.

但是，当现有的库不能满足您的需要时，您仍然很有可能发现自己需要一个解析器或其他东西。即使当您重用一些现有的实现时，您也不可避免地需要调试和维护，并在其内部进行探索。

> ### 1.1.2 Languages are great exercise

1.1.2 语言是很好的锻炼

> Long distance runners sometimes train with weights strapped to their ankles or at high altitudes where the atmosphere is thin. When they later unburden themselves, the new relative ease of light limbs and oxygen-rich air enables them to run farther and faster.

长跑运动员有时会在脚踝上绑上重物，或者在空气稀薄的高海拔地区进行训练。当他们卸下自己的负担以后，轻便的肢体和富氧的空气带来了新的相对舒适度，使它们可以跑得更远，更快。

> Implementing a language is a real test of programming skill. The code is complex and performance critical. You must master recursion, dynamic arrays, trees, graphs, and hash tables. You probably use hash tables at least in your day-to-day programming, but how well do you *really* understand them? Well, after we've crafted our own from scratch, I guarantee you will.

实现一门语言是对编程技能的真正考验。代码很复杂，而且性能很关键。您必须掌握递归、动态数组、树、图和哈希表。您在日常编程中至少使用过哈希表，但*到底*您对它们的理解程度有多高呢？嗯，等我们从头完成我们的作品之后，我相信您会理解的。

> While I intend to show you that an interpreter isn't as daunting as you might believe, implementing one well is still a challenge. Rise to it, and you'll come away a stronger programmer, and smarter about how you use data structures and algorithms in your day job.

虽然我想说明解释器并不像您想的那样令人生畏，但实现一个好的解释器仍然是一个挑战。学会了它，您就会成为一个更强大的程序员，并且在日常工作中也能更加聪明地使用数据结构和算法。

> ### 1.1.3 One more reason

1.1.3 另一个原因

> This last reason is hard for me to admit, because it's so close to my heart. Ever since I learned to program as a kid, I felt there was something magical about languages. When I first tapped out BASIC programs one key at a time I couldn't conceive how BASIC *itself* was made.

这最后一个原因我很难承认，因为它是很私密的理由。自从我小时候学会编程以来，我就觉得语言有种神奇的力量。当我第一次一个键一个键地输入BASIC程序时，我无法想象BASIC语言*本身*是如何制作出来的。

> Later, the mixture of awe and terror on my college friends' faces when talking about their compilers class was enough to convince me language hackers were a different breed of human—some sort of wizards granted privileged access to arcane arts.

后来，当我的大学朋友们谈论他们的编译器课程时，脸上那种既敬畏又恐惧的表情足以让我相信，语言黑客是另一种人，某种可以操控奥术的巫师。

> It's a charming image, but it has a darker side. *I* didn't feel like a wizard, so I was left thinking I lacked some in-born quality necessary to join the cabal. Though I've been fascinated by languages ever since I doodled made up keywords in my school notebook, it took me decades to muster the courage to try to really learn them. That "magical" quality, that sense of exclusivity, excluded *me*.

这是一个迷人的形象，但它也有黑暗的一面。我感觉自己不像个巫师，所以我认为自己缺乏加入秘社所需的先天品质。 尽管自从我在学校笔记本上拼写关键词以来，我一直对语言着迷，但我花了数十年的时间鼓起勇气尝试真正地学习它们。那种"神奇"的品质，那种排他性的感觉，将我挡在门外。

> When I did finally start cobbling together my own little interpreters, I quickly learned that, of course, there is no magic at all. It's just code, and the people who hack on languages are just people.

当我最终开始拼凑我自己的小解释器时，我很快意识到，根本就没有魔法。它只是代码，而那些玩弄语言的人也只是普通人。

> There *are* a few techniques you don't often encounter outside of languages, and some parts are a little difficult. But not more difficult than other obstacles you've overcome. My hope is that if you've felt intimidated by languages, and this book helps you overcome that fear, maybe I'll leave you just a tiny bit braver than you were before.

有一些技巧您在语言之外不会经常遇到，而且有些部分有点难。但不会比您克服的其他障碍更困难。我希望，如果您对语言感到害怕，而这本书能帮助您克服这种恐惧，也许我会让您比以前更勇敢一点。

> And, who knows, maybe you *will* make the next great language. Someone has to.

而且，说不准，你也许会创造出下一个伟大的语言，毕竟总要有人做。

1.2 How the Book is Organized

1.2 本书的组织方式

> This book is broken into three parts. You're reading the first one now. It's a couple of chapters to get you oriented, teach you some of the lingo that language hackers use, and introduce you to Lox, the language we'll be implementing.

这本书分为三个部分。您现在正在读的是第一部分。这部分用了几章来让您进入状态，教您一些语言黑客使用的行话，并向您介绍我们将要实现的语言Lox。

> Each of the other two parts builds one complete Lox interpreter. Within those parts, each chapter is structured the same way. The chapter takes a single language feature, teaches you the concepts behind it, and walks through an implementation.

其他两个部分则分别构建一个完整的Lox解释器。在这些部分中，每个章节的结构都是相同的。 每一章节挑选一个语言功能点，教您背后对应的概念，并逐步介绍实现方法。

> It took a good bit of trial and error on my part, but I managed to carve up the two interpreters into chapter-sized chunks that build on the previous chapters but require nothing from later ones. From the very first chapter, you'll have a working program you can run and play with. With each passing chapter, it grows increasingly full-featured until you eventually have a complete language.

我花了不少时间去试错，但我还是成功地把这两个解释器按照章节分成了一些小块，每一小块的内容都会建立在前面几章的基础上，但不需要后续章节的知识。从第一章开始，你就会有一个可以运行和使用的工作程序。随着章节的推移，它的功能越来越丰富，直到你最终拥有一门完整的语言。

> Aside from copious, scintillating English prose, chapters have a few other delightful facets:

除了大量妙趣横生的英文段落，章节中还会包含一些其它的惊喜：

> 1.2.1 The code

1.2.1 代码

> We're about *crafting* interpreters, so this book contains real code. Every single line of code needed is included, and each snippet tells you where to insert it in your ever-growing implementation.

本书是关于制作解释器的，所以其中会包含真正的代码。所需要的每一行代码都需要包含在内，而且每个代码片段都会告知您需要插入到实现代码中的什么位置。

> Many other language books and language implementations use tools like Lex and Yacc, so-called **compiler-compilers** that automatically generate some of the source files for an implementation from some higher level description. There are pros and cons to tools like those, and strong opinions—some might say religious convictions—on both sides.

许多其他的语言书籍和语言实现都使用Lex和Yacc^3这样的工具，也就是所谓的**编译器-编译器**，可以从一些更高层次的（语法）描述中自动生成一些实现的源文件。这些工具有利有弊，而且双方都有强烈的主张--有些人可能将其说成是信仰。

> We will abstain from using them here. I want to ensure there are no dark corners where magic and confusion can hide, so we'll write everything by hand. As you'll see, it's not as bad as it sounds and it means you really will understand each line of code and how both interpreters work.

我们这里不会使用这些工具。我想确保魔法和困惑不会藏在黑暗的角落，所以我们会选择手写所有代码。正如您将看到的，这并没有听起来那么糟糕，因为这意味着您将真正理解每一行代码以及两种解释器的工作方式。

> A book has different constraints from the "real world" and so the coding style here might not always reflect the best way to write maintainable production software. If I seem a little cavalier about, say, omitting `private` or declaring a global variable, understand I do so to keep the code easier on your eyes. The pages here aren't as wide as your IDE and every character counts.

为了写书，书中代码和"真实世界"的代码是有区别的，因此这里的代码风格可能并不是编写可维护的生产型软件的最佳方式。可能我的某些写法是不太准确的，比如省略*private*或者声明全局变量，请理解我这样做是为了让您更容易看懂代码。书页不像IDE窗口那么宽，所以每一个字符都很珍贵。

> Also, the code doesn't have many comments. That's because each handful of lines is surrounded by several paragraphs of honest-to-God prose explaining it. When you write a book to accompany your program, you are welcome to omit comments too. Otherwise, you should probably use `//` a little more than I do.

另外，代码也不会有太多的注释。这是因为每一部分代码前后，都使用了一些真的很简洁的文字来对其进行解释。当你写一本书来配合你的程序时，欢迎你也省略注释。否则，你可能应该比我使用更多的 `//`。

> While the book contains every line of code and teaches what each means, it does not describe the machinery needed to compile and run the interpreter. I assume you can slap together a makefile or a project in your IDE of choice in order to get the code to run. Those kinds of instructions get out of date quickly, and I want this book to age like XO brandy, not backyard hooch.

虽然这本书包含了每一行代码，并教授了每一行代码的含义，但它没有描述编译和运行解释器所需的机制。我假设你可以简单地拼凑出一个makefile，或者创建一个心仪的IDE中的一个工程，来让代码运行起来。 那种类型的说明很快就会过时，我希望这本书能像XO白兰地一样醇久，而不是像家酿酒（一样易过期）。

> ## 1.2.2 Snippets

1.2.2 片段

> Since the book contains literally every line of code needed for the implementations, the snippets are quite precise. Also, because I try to keep the program in a runnable state even when major features are missing, sometimes we add temporary code that gets replaced in later snippets.

因为这本书包含了实现所需的每一行代码，所以代码片段相当精确。此外，即使是在缺少主要功能的时候，我也尝试将程序保持在可运行状态。因此我们有时会添加临时代码，这些代码将在后面被其他的代码片段替换。

> A snippet with all the bells and whistles looks like this:

一个完整的代码片段可能如下所示：

```
    default:
      if (isDigit(c)) {
        number();
      } else {
        Lox.error(line, "Unexpected character.");
      }
      break;
```

<< lox/Scanner.java
in scanToken()
replace 1 line

> In the center, you have the new code to add. It may have a few faded out lines above or below to show where it goes in the existing surrounding code. There is also a little blurb telling you in which file and where to place the snippet. If that blurb says “replace _ lines”, there is some existing code between the faded lines that you need to remove and replace with the new snippet.

中间是要添加的新代码。这部分代码的上面或下面可能有一些淡出的行，以显示它在周围代码中的位置。还会附有一小段介绍，告诉您在哪个文件中以及在哪里放置代码片段。如果简介说要“replace _ lines”，表明在浅色的行之间有一些现有的代码需要删除，并替换为新的代码片段。

> ## 1.2.3 Asides

1.2.3 题外话

> Asides contain biographical sketches, historical background, references to related topics, and suggestions of other areas to explore. There’s nothing that you *need* to know in them to understand later parts of the book, so you can skip them if you want. I won’t judge you, but I might be a little sad.

题外话中包含传记简介、历史背景、对相关主题的引用以及对其他要探索的领域的建议。 您无需深入了解就可以理解本书的后续部分，因此可以根据需要跳过它们。 我不会批评你，但我可能会有些难过。【注：由于排版

原因，在翻译的时候，将有用的旁白信息作为脚注附在章节之后】

> ### 1.2.4 Challenge

1.2.4 挑战

> Each chapter ends with a few exercises. Unlike textbook problem sets which tend to review material you already covered, these are to help you learn *more* than what's in the chapter. They force you to step off the guided path and explore on your own. They will make you research other languages, figure out how to implement features, or otherwise get you out of your comfort zone.

每章结尾都会有一些练习题。 不像教科书中的习题集那样用于回顾已讲述的内容，这些习题是为了帮助您学习更多的知识，而不仅仅是本章中的内容。 它们会迫使您走出文章指出的路线，自行探索。 它们将要求您研究其他语言，弄清楚如何实现功能，换句话说，就是使您走出舒适区。

> Vanquish the challenges and you'll come away with a broader understanding and possibly a few bumps and scrapes. Or skip them if you want to stay inside the comfy confines of the tour bus. It's your book.

克服挑战，您将获得更广泛的理解，也可能遇到一些挫折。 如果您想留在旅游巴士的舒适区内，也可以跳过它们。 都随你便^4。

> ### 1.2.5 Design notes

1.2.5 设计笔记

> Most "programming language" books are strictly programming language *implementation* books. They rarely discuss how one might happen to *design* the language being implemented. Implementation is fun because it is so precisely defined. We programmers seem to have an affinity for things that are black and white, ones and zeroes.

大多数编程语言书籍都是严格意义上的编程语言*实现*书籍。他们很少讨论如何*设计*正在实现的语言。实现之所以有趣，是因为它的定义是很精确的。我们程序员似乎很喜欢黑白、1和0这样的事物^5。

> Personally, I think the world only needs so many implementations of FORTRAN 77. At some point, you find yourself designing a *new* language. Once you start playing *that* game, then the softer, human side of the equation becomes paramount. Things like what features are easy to learn, how to balance innovation and familiarity, what syntax is more readable and to whom.

就个人而言，我认为世界只需要这么多的FORTRAN 77实现。在某个时候，您会发现自己正在设计一种新的语言。 一旦开始这样做，方程式中较柔和，人性化的一面就变得至关重要。 诸如哪些功能易于学习，如何在创新和熟悉度之间取得平衡，哪种语法更易读以及对谁有帮助^6。

> All of that stuff profoundly affects the success of your new language. I want your language to succeed, so in some chapters I end with a "design note", a little essay on some corner of the human aspect of programming languages. I'm no expert on this—I don't know if anyone really is—so take these with a large pinch of salt. That should make them tastier food for thought, which is my main aim.

所有这些都会对您的新语言的成功产生深远的影响。 我希望您的语言取得成功，因此在某些章节中，我以一篇"设计笔记"结尾，这些是关于编程语言的人文方面的一些文章。我并不是这方面的专家——我不确定是否有

人真的精通这些，因此，请您在阅读这些文字的时候仔细评估。这样的话，这些文字就能成为您思考的食材，这也正是我的目标。

1.3 The First Interpreter

1.3 第一个解释器

> We'll write our first interpreter, jlox, in Java. The focus is on *concepts*. We'll write the simplest, cleanest code we can to correctly implement the semantics of the language. This will get us comfortable with the basic techniques and also hone our understanding of exactly how the language is supposed to behave.

我们将用Java编写第一个解释器jlox。（这里的）主要关注点是概念。 我们将编写最简单，最干净的代码，以正确实现该语言的语义。 这样能够帮助我们熟悉基本技术，并磨练对语言表现形式的确切理解。

> Java is a great language for this. It's high level enough that we don't get overwhelmed by fiddly implementation details, but it's still pretty explicit. Unlike scripting languages, there tends to be less complex machinery hiding under the hood, and you've got static types to see what data structures you're working with.

Java是一门很适合这种场景的语言。它的级别足够高，我们不会被繁琐的实现细节淹没，但代码仍是非常明确的。与脚本语言不同的是，它的底层没有隐藏太过复杂的机制，你可以使用静态类型来查看正在处理的数据结构。

> I also chose Java specifically because it is an object-oriented language. That paradigm swept the programming world in the 90s and is now the dominant way of thinking for millions of programmers. Odds are good you're already used to organizing code into classes and methods, so we'll keep you in that comfort zone.

我选择Java还有特别的原因，就是因为它是一种面向对象的语言。 这种范式在90年代席卷了整个编程世界，如今已成为数百万程序员的主流思维方式。 很有可能您已经习惯了将代码组织到类和方法中，因此我们将让您在舒适的环境中学习。

> While academic language folks sometimes look down on object-oriented languages, the reality is that they are widely used even for language work. GCC and LLVM are written in C++, as are most JavaScript virtual machines. Object-oriented languages are ubiquitous and the tools and compilers *for* a language are often written *in* the same language.

虽然学术语言专家有时瞧不起面向对象语言，但事实上，它们即使在语言工作中也被广泛使用。GCC和LLVM是用C++编写的，大多数JavaScript虚拟机也是这样。 面向对象的语言无处不在，并且针对该语言的工具和编译器通常是用同一种语言编写的^7。

> And, finally, Java is hugely popular. That means there's a good chance you already know it, so there's less for you to learn to get going in the book. If you aren't that familiar with Java, don't freak out. I try to stick to a fairly minimal subset of it. I use the diamond operator from Java 7 to make things a little more terse, but that's about it as far as "advanced" features go. If you know another object-oriented language like C# or C++, you can muddle through.

最后，Java非常流行。 这意味着您很有可能已经了解它了，所以你要学习的东西就更少了。 如果您不太熟悉Java，也请不要担心。 我尽量只使用它的最小子集。我使用Java 7中的菱形运算符使代码看起来更简洁，但

就“高级”功能而言，仅此而已。 如果您了解其它面向对象的语言（例如C＃或C++），就没有问题。

> By the end of part II, we’ll have a simple, readable implementation. What we won’t have is a *fast* one. It also takes advantage of the Java virtual machine’s own runtime facilities. We want to learn how Java *itself* implements those things.

在第二部分结束时，我们将得到一个简单易读的实现。 但是我们得到的不会是一个*执行效率*高的解释器。它还是利用了Java虚拟机自身的运行时工具。我们想要学习Java本身是如何实现这些东西的。

1.4 The Second Interpreter

1.4 第二个解释器

> So in the next part, we start all over again, but this time in C. C is the perfect language for understanding how an implementation *really* works, all the way down to the bytes in memory and the code flowing through the CPU.

所以在下一部分，我们将从头开始，但这一次是用C语言。C语言是理解实现编译器工作方式的完美语言，一直到内存中的字节和流经CPU的代码。

> A big reason that we’re using C is so I can show you things C is particularly good at, but that *does* mean you’ll need to be pretty comfortable with it. You don’t have to be the reincarnation of Dennis Ritchie, but you shouldn’t be spooked by pointers either.

我们使用C语言的一个重要原因是，我可以向您展示C语言特别擅长的东西，但这并不意味着您需要非常熟练地使用它。您不必是丹尼斯·里奇（Dennis Ritchie）的转世，但也不应被指针吓倒。

> If you aren’t there yet, pick up an introductory book on C and chew through it, then come back here when you’re done. In return, you’ll come away from this book an even stronger C programmer. That’s useful given how many language implementations are written in C: Lua, CPython, and Ruby’s MRI, to name a few.

如果你（对C的掌握）还没到那一步，找一本关于C的入门书，仔细阅读，读完后再回来。作为回报，从这本书中你将成为一个更优秀的C程序员。可以想想有多少语言实现是用C完成的: Lua、CPython和Ruby 的 MRI等，这里仅举几例。

> In our C interpreter, clox, we are forced to implement for ourselves all the things Java gave us for free. We’ll write our own dynamic array and hash table. We’ll decide how objects are represented in memory, and build a garbage collector to reclaim it.

在我们的C解释器clox中^8，我们不得不自己实现那些Java免费提供给我们的东西。 我们将编写自己的动态数组和哈希表。 我们将决定对象在内存中的表示方式，并构建一个垃圾回收器来回收它。

> Our Java implementation was focused on being correct. Now that we have that down, we’ll turn to also being *fast*. Our C interpreter will contain a compiler that translates Lox to an efficient bytecode representation (don’t worry, I’ll get into what that means soon) which it then executes. This is the same technique used by implementations of Lua, Python, Ruby, PHP, and many other successful languages.

我们的Java版实现专注于正确性。 既然我们已经完成了，那么我们就要变*快*。我们的C解释器将包含一个编译器^9，该编译器会将Lox转换为有效的字节码形式（不用担心，我很快就会讲解这是什么意思）之后它会执行对应的字节码。 这与Lua， Python， Ruby， PHP和许多其它成功语言的实现所使用的技术相同。

> We'll even try our hand at benchmarking and optimization. By the end, we'll have a robust, accurate, fast interpreter for our language, able to keep up with other professional caliber implementations out there. Not bad for one book and a few thousand lines of code.

我们甚至会尝试进行基准测试和优化。 到最后，我们将为lox语言提供一个强大，准确，快速的解释器，并能够不落后于其他专业水平的实现。对于一本书和几千行代码来说已经不错了。

^1: 静态类型系统尤其需要严格的形式推理。破解类型系统就像证明数学定理一样。事实证明这并非巧合。 上世纪初，Haskell Curry和William Alvin Howard证明了它们是同一枚硬币的两个方面：Curry-Howard同构。^2: pidgins，洋泾浜语言，一种混杂的英语 ^3: Yacc是一个工具，它接收语法文件并生成编译器的源文件，因此它有点像一个输出“编译器”的编译器，在术语中叫作“compiler-compiler”，即编译器的编译器。Yacc并不是同类工具中的第一个，这就是为什么它被命名为“Yacc”—*Yet Another* Compiler-Compiler(另一个Compiler-Compiler)。后来还有一个类似的工具是Bison，它的名字源于Yacc和yak的发音，是一个双关语。 ^4: 警告:挑战题目通常要求您对正在构建的解释器进行更改。您需要在代码副本中实现这些功能。后面的章节都假设你的解释器处于原始(未解决挑战题)状态。 ^5: 我知道很多语言黑客的职业就基于此。您将一份语言规范塞到他们的门下，等上几个月，代码和基准测试结果就出来了。 ^6: 希望您的新语言不会将对打孔卡宽度的假设硬编码到语法中。 ^7: 编译器以一种语言读取文件。 翻译它们，并以另一种语言输出文件。 您可以使用任何一种语言（包括与目标语言相同的语言）来实现编译器，该过程称为“自举”。你现在还不能使用编译器本身来编译你自己的编译器，但是如果你用其它语言为你的语言写了一个编译器，你就可以用那个编译器编译一次你的编译器。现在，您可以使用自己的编译器的已编译版本来编译自身的未来版本，并且可以从另一个编译器中丢弃最初的已编译版本。 这就是所谓的“自举”，通过自己的引导程序将自己拉起来。 ^8: 我把这个名字读作“sea-locks”，但是你也可以读作“clocks”，如果你愿意的话可以像希腊人读“x”那样将其读作“clochs”， ^9: 你以为这只是一本讲解释器的书吗?它也是一本讲编译器的书。买一送一。

> ## CHALLENGES

习题

> 1、There are at least six domain-specific languages used in the little system I cobbled together to write and publish this book. What are they?

在我编写的这个小系统中，至少有六种特定领域语言（DSL），它们是什么？

> 2、Get a “Hello, world!” program written and running in Java. Set up whatever Makefiles or IDE projects you need to get it working. If you have a debugger, get comfortable with it and step through your program as it runs.

使用Java编写并运行一个“Hello, world!”程序，配置一个你需要的makefile或IDE项目让它跑起来。如果您有调试器，请先熟悉一下，并在程序运行时对代码逐步调试。

> 3、Do the same thing for C. To get some practice with pointers, define a doubly-linked list of heap-allocated strings. Write functions to insert, find, and delete items from it. Test them.

对C也进行同样的操作。为了练习使用指针，可以定义一个堆分配字符串的双向链表。编写函数以插入，查找和删除其中的项目。 测试编写的函数。

> ## DESIGN NOTE: WHAT'S IN A NAME?

设计笔记：名称是什么？

> One of the hardest challenges in writing this book was coming up with a name for the language it implements. I went through *pages* of candidates before I found one that worked. As you'll discover on the first day you start building your own language, naming is deviously hard. A good name satisfies a few criteria:
>
> 1. **It isn't in use.** You can run into all sorts of trouble, legal and social, if you inadvertently step on someone else's name.
> 2. **It's easy to pronounce.** If things go well, hordes of people will be saying and writing your language's name. Anything longer than a couple of syllables or a handful of letters will annoy them to no end.
> 3. **It's distinct enough to search for.** People will Google your language's name to learn about it, so you want a word that's rare enough that most results point to your docs. Though, with the amount of AI search engines are packing today, that's less of an issue. Still, you won't be doing your users any favors if you name your language "for".
> 4. **It doesn't have negative connotations across a number of cultures.** This is hard to guard for, but it's worth considering. The designer of Nimrod ended up renaming his language to "Nim" because too many people only remember that Bugs Bunny used "Nimrod" (ironically, actually) as an insult.
>
> If your potential name makes it through that gauntlet, keep it. Don't get hung up on trying to find an appellation that captures the quintessence of your language. If the names of the world's other successful languages teach us anything, it's that the name doesn't matter much. All you need is a reasonably unique token.

写这本书最困难的挑战之一是为它所实现的语言取个名字。我翻了好几页的备选名才找到一个合适的。当你某一天开始构建自己的语言时，你就会发现命名是非常困难的。一个好名字要满足几个标准：

1. **尚未使用**。如果您不小心使用了别人的名字，就可能会遇到各种法律和社会上的麻烦。
2. **容易发音**。如果一切顺利，将会有很多人会说和写您的语言名称。 超过几个音节或几个字母的任何内容都会使他们陷入无休止的烦恼。
3. **足够独特，易于搜索**。人们会Google你的语言的名字来了解它，所以你需要一个足够独特的单词，以便大多数搜索结果都会指向你的文档。不过，随着人工智能搜索引擎数量的增加，这已经不是什么大问题了。但是，如果您将语言命名为" for"，那对用户基本不会有任何帮助。
4. **在多种文化中，都没有负面的含义**。这很难防范，但是值得深思。Nimrod的设计师最终将其语言重命名为" Nim"，因为太多的人只记得Bugs Bunny使用" Nimrod"作为一种侮辱（其实是讽刺）。

如果你潜在的名字通过了考验，就保留它吧。不要纠结于寻找一个能够抓住你语言精髓的名称。如果说世界上其他成功的语言的名字教会了我们什么的话，那就是名字并不重要。您所需要的只是一个相当独特的标记。

2.领土地图 A Map of the Territory

> You must have a map, no matter how rough. Otherwise you wander all over the place. In "The Lord of the Rings" I never made anyone go farther than he could on a given day.
>
> ——J.R.R. Tolkien

你必须要有一张地图，无论它是多么粗糙。否则你就会到处乱逛。在《指环王》中，我从未让任何人在某一天走得超出他力所能及的范围。

> We don't want to wander all over the place, so before we set off, let's scan the territory charted by previous language implementers. It will help us understand where we are going and alternate routes others take.

我们不想到处乱逛，所以在我们开始之前，让我们先浏览一下以前的语言实现者所绘制的领土。它能帮助我们了解我们的目的地和其他人采用的备选路线。

> First, let me establish a shorthand. Much of this book is about a language's *implementation*, which is distinct from the *language itself* in some sort of Platonic ideal form. Things like "stack", "bytecode", and "recursive descent", are nuts and bolts one particular implementation might use. From the user's perspective, as long as the resulting contraption faithfully follows the language's specification, it's all implementation detail.

首先，我先做个简单说明。本书的大部分内容都是关于语言的*实现*，它与*语言本身*这种柏拉图式的理想形式有所不同。诸如"堆栈"，"字节码"和"递归下降"之类的东西是某个特定实现中可能使用的基本要素。从用户的角度来说，只要最终产生的装置能够忠实地遵循语言规范，这些都是东西不过是他们不关心的实现细节罢了。

> We're going to spend a lot of time on those details, so if I have to write "language *implementation*" every single time I mention them, I'll wear my fingers off. Instead, I'll use "language" to refer to either a language or an implementation of it, or both, unless the distinction matters.

我们将会花很多时间在这些细节上，所以如果我每次提及的时候都写"语言实现"，我的手指都会被磨掉。相反，除非有重要的区别，否则我将使用"语言"来指代一种语言或该语言的一种实现，或两者皆有。

> ## 2.1 The Parts of a Language

2.1 语言的各部分

> Engineers have been building programming languages since the Dark Ages of computing. As soon as we could talk to computers, we discovered doing so was too hard, and we enlisted their help. I find it fascinating that even though today's machines are literally a million times faster and have orders of magnitude more storage, the way we build programming languages is virtually unchanged.

自计算机的黑暗时代以来，工程师们就一直在构建编程语言。当我们可以和计算机对话的时候，我们发现这样做太难了，于是我们寻求电脑的帮助。我觉得很有趣的是，即使今天的机器确实快了一百万倍，存储空间也大了几个数量级，但我们构建编程语言的方式几乎没有改变。

> Though the area explored by language designers is vast, the trails they've carved through it are few. Not every language takes the exact same path—some take a shortcut or two—but otherwise they are reassuringly similar from Rear Admiral Grace Hopper's first COBOL compiler all the way to some hot new transpile-to-JavaScript language whose "documentation" consists entirely of a single poorly-edited README in a Git repository somewhere.

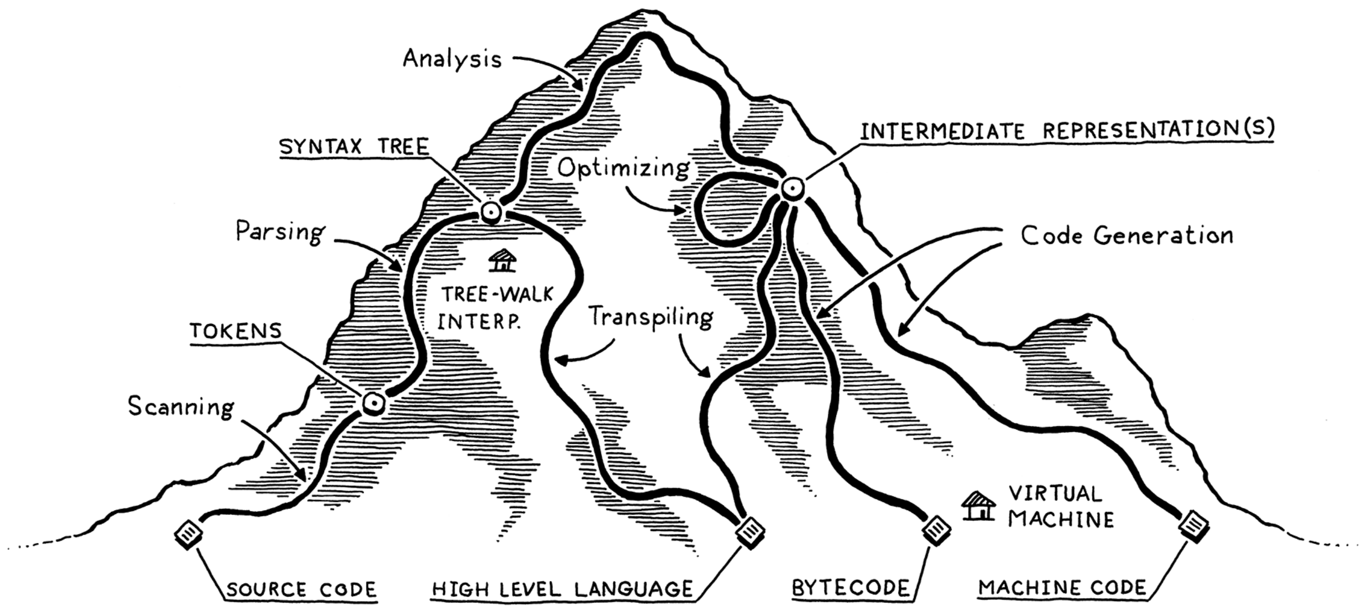
尽管语言设计师所探索的领域辽阔，但他们往往都走到相似的几条路上。 并非每种语言都采用完全相同的路径（有些会采用一种或两种捷径），但除此之外，从海军少将Grace Hopper的第一个COBOL编译器，一直到一些热门的可以转译到JavaScript的语言（JS的 "文档 "甚至完全是由Git仓库中一个编辑得很差的README组成的^1），都呈现出相似的特征，这令人十分欣慰。

> I visualize the network of paths an implementation may choose as climbing a mountain. You start off at the bottom with the program as raw source text, literally just a string of characters. Each phase analyzes the program and transforms it to some higher-level representation where the semantics—what the author wants the computer to do—becomes more apparent.

我把一个语言实现可能选择的路径网络类比为爬山。你从最底层开始，程序是原始的源文本，实际上只是一串字符。每个阶段都会对程序进行分析，并将其转换为更高层次的表现形式，从而使语义（作者希望计算机做什么）变得更加明显。

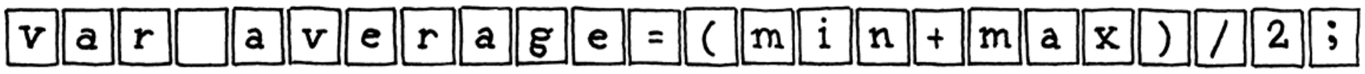
> Eventually we reach the peak. We have a bird's-eye view of the users's program and can see what their code *means*. We begin our descent down the other side of the mountain. We transform this highest-level representation down to successively lower-level forms to get closer and closer to something we know how to make the CPU actually execute.

最终我们达到了峰顶。我们可以鸟瞰用户的程序，可以看到他们的代码含义是什么。我们开始从山的另一边下山。我们将这个最高级的表示形式转化为连续的较低级别的形式，从而越来越接近我们所知道的如何让CPU真正执行的形式。



> Let's trace through each of those trails and points of interest. Our journey begins on the left with the bare text of the user's source code:

让我们追随着这一条条路径和点前进吧。我们的旅程从左边的用户源代码的纯文本开始：



> ## 2.1.1 Scanning

2.1.1 扫描

> The first step is **scanning**, also known as **lexing**, or (if you're trying to impress someone) **lexical analysis**. They all mean pretty much the same thing. I like "lexing" because it sounds like something an evil supervillain would do, but I'll use "scanning" because it seems to be marginally more commonplace.

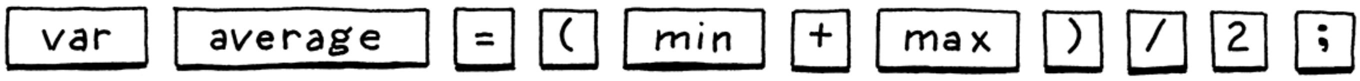
第一步是**扫描**，也就是所谓的**词法分析**（lexing 或者强调写法 lexical analysis）。扫描和词法分析的意思相近。我喜欢词法分析这个描述，因为这听起来像是一个邪恶的超级大坏蛋会做的事情，但我还是用扫描，因为它似乎更常见一些。

> A **scanner** (or **lexer**) takes in the linear stream of characters and chunks them together into a series of something more akin to “words”. In programming languages, each of these words is called a **token**. Some tokens are single characters, like `(` and `,`. Others may be several characters long, like numbers (`123`), string literals (`"hi!"`), and identifiers (`min`).

扫描器(或词法解析器)接收线性字符流，并将它们切分成一系列更类似于“单词”的东西。在编程语言中，这些词的每一个都被称为**词法单元**。有些词法单元是单个字符，比如`(`和`,`。其他的可能是几个字符长的，比如数字（`123`）、字符串字元（`"hi!"`）和标识符（`min`）。

> Some characters in a source file don’t actually mean anything. Whitespace is often insignificant and comments, by definition, are ignored by the language. The scanner usually discards these, leaving a clean sequence of meaningful tokens.

源文件中的一些字符实际上没有任何意义。空格通常是无关紧要的，而注释，从定义就能看出来，会被变成语言忽略。扫描器通常会丢弃这些字符，留下一个干净的有意义的词法单元序列。



> 2.1.2 Parsing

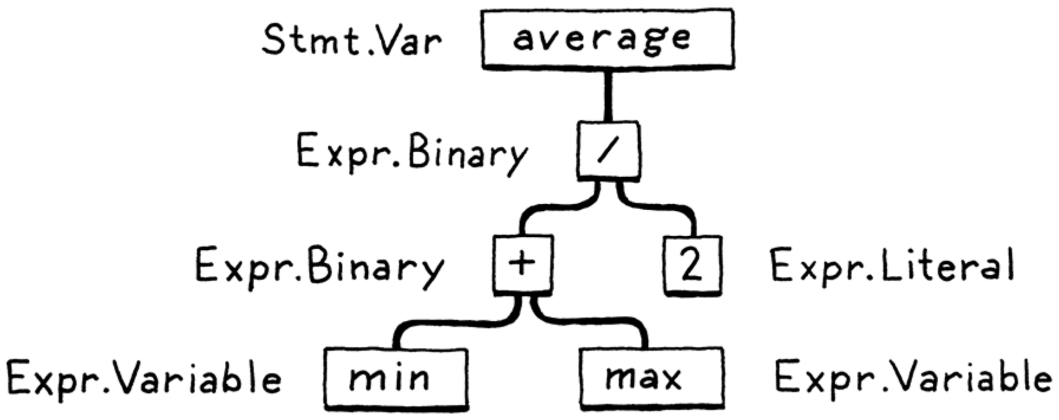
2.1.2 语法分析

> The next step is **parsing**. This is where our syntax gets a **grammar**—the ability to compose larger expressions and statements out of smaller parts. Did you ever diagram sentences in English class? If so, you’ve done what a parser does, except that English has thousands and thousands of “keywords” and an overflowing cornucopia of ambiguity. Programming languages are much simpler.

下一步是**解析**。这就是我们从句法中得到**语法**的地方——语法能够将较小的部分组成较大的表达式和语句。你在英语课上做过语法图解吗？如果有，你就做了解析器所做的事情，区别在于，英语中有成千上万的“关键字”和大量的歧义，而编程语言要简单得多。

> A **parser** takes the flat sequence of tokens and builds a tree structure that mirrors the nested nature of the grammar. These trees have a couple of different names—**“parse tree”** or **“abstract syntax tree”**—depending on how close to the bare syntactic structure of the source language they are. In practice, language hackers usually call them **“syntax trees”**, **“ASTs”**, or often just **“trees”**.

解析器将扁平的词法单元序列转化为树形结构，树形结构能更好地反映语法的嵌套本质。这些树有两个不同的名称:**解析树**或**抽象语法树**，这取决于它们与源语言的语法结构有多接近。在实践中，语言黑客通常称它们为“**语法树**”、“**AST**”，或者干脆直接说“**树**”。



> Parsing has a long, rich history in computer science that is closely tied to the artificial intelligence community. Many of the techniques used today to parse programming languages were originally conceived to parse *human* languages by AI researchers who were trying to get computers to talk to us.

解析在计算机科学中有着悠久而丰富的历史，它与人工智能界有着密切的联系。今天用于解析编程语言的许多技术最初被人工智能研究人员用于解析人类语言，人工智能研究人员试图通过这些技术让计算机能与我们对话。

> It turns out human languages are too messy for the rigid grammars those parsers could handle, but they were a perfect fit for the simpler artificial grammars of programming languages. Alas, we flawed humans still manage to use those simple grammars incorrectly, so the parser's job also includes letting us know when we do by reporting **syntax errors**.

事实证明，人类语言对于只能处理僵化语法的解析器来说太混乱了，但面对编程语言这种简单的人造语法时，解析器表现得十分合适。唉，可惜我们这些有缺陷的人类在使用这些简单的语法时，仍然会不停地出错，因此解析器的工作还包括通过报告**语法错误**让我们知道出错了。

> ### 2.1.3 Static analysis

2.1.3 静态分析

> The first two stages are pretty similar across all implementations. Now, the individual characteristics of each language start coming into play. At this point, we know the syntactic structure of the code—things like which expressions are nested in which others—but we don't know much more than that.

在所有实现中，前两个阶段都非常相似。 现在，每种语言的个性化特征开始发挥作用。 至此，我们知道了代码的语法结构（诸如哪些表达式嵌套在其他表达式中）之类的东西，但是我们知道的也就仅限于此了。

> In an expression like `a + b`, we know we are adding `a` and `b`, but we don't know what those names refer to. Are they local variables? Global? Where are they defined?

在`a + b`这样的表达式中，我们知道我们要把`a`和`b`相加，但我们不知道这些名字指的是什么。它们是局部变量吗？全局变量？它们在哪里被定义？

> The first bit of analysis that most languages do is called **binding** or **resolution**. For each **identifier** we find out where that name is defined and wire the two together. This is where **scope** comes into play—the region of source code where a certain name can be used to refer to a certain declaration.

大多数语言所做的第一点分析叫做**绑定**或**决议**。对于每一个**标识符**，我们都要找出定义该名称的地方，并将两者连接起来。这就是**作用域**的作用——在这个源代码区域中，某个名字可以用来引用某个声明。

> If the language is statically typed, this is when we type check. Once we know where a and b are declared, we can also figure out their types. Then if those types don't support being added to each other, we report a **type error**.

如果语言是静态类型的，这就是我们进行类型检查的时机。一旦我们知道了a和b的声明位置，我们也可以弄清楚它们的类型。然后如果这些类型不支持相加，我们就会报告一个**类型错误**^2。

> Take a deep breath. We have attained the summit of the mountain and a sweeping view of the user's program. All this semantic insight that is visible to us from analysis needs to be stored somewhere. There are a few places we can squirrel it away:
>
> - Often, it gets stored right back as **attributes** on the syntax tree itself—extra fields in the nodes that aren't initialized during parsing but get filled in later.
> - Other times, we may store data in a look-up table off to the side. Typically, the keys to this table are identifiers—names of variables and declarations. In that case, we call it a **symbol table** and the values it associates with each key tell us what that identifier refers to.
> - The most powerful bookkeeping tool is to transform the tree into an entirely new data structure that more directly expresses the semantics of the code. That's the next section.

深吸一口气。 我们已经到达了山顶，并对用户的程序有了全面的了解。从分析中可见的所有语义信息都需要存储在某个地方。我们可以把它存储在几个地方：

- 通常，它会被直接存储在语法树本身的**属性**中——属性是节点中的额外字段，这些字段在解析时不会初始化，但在稍后会进行填充。
- 有时，我们可能会将数据存储在外部的查找表中。 通常，该表的关键字是标识符，即变量和声明的名称。 在这种情况下，我们称其为**符号表**，并且其中与每个键关联的值告诉我们该标识符所指的是什么。
- 最强大的记录工具是将树转化为一个全新的数据结构，更直接地表达代码的语义。这是下一节的内容。

> Everything up to this point is considered the **front end** of the implementation. You might guess everything after this is the **back end**, but no. Back in the days of yore when "front end" and "back end" were coined, compilers were much simpler. Later researchers invented new phases to stuff between the two halves. Rather than discard the old terms, William Wulf and company lumped them into the charming but spatially paradoxical name **middle end**.

到目前为止，所有内容都被视为实现的**前端**。你可能会猜至此以后是**后端**，其实并不是。 在过去的年代，当"前端"和"后端"被创造出来时，编译器要简单得多。 后来，研究人员在两个半部之间引入了新阶段。 威廉·沃尔夫（William Wulf）和他的同伴没有放弃旧术语，而是新添加了一个迷人但有点自相矛盾的名称"**中端**"。

> ## 2.1.4 Intermediate representations

2.1.4 中间码

> You can think of the compiler as a pipeline where each stage's job is to organize the data representing the user's code in a way that makes the next stage simpler to implement. The front end of the pipeline is specific to the source language the program is written in. The back end is concerned with the final architecture where the program will run.

你可以把编译器看成是一条流水线，每个阶段的工作是把代表用户代码的数据组织起来，使下一阶段的实现更加简单。管道的前端是针对程序所使用的源语言编写的。后端关注的是程序运行的最终架构。

> In the middle, the code may be stored in some **intermediate representation** (or **IR**) that isn't tightly tied to either the source or destination forms (hence "intermediate"). Instead, the IR acts as an interface between these two languages.

在中间阶段，代码可能被存储在一些**中间代码**（**intermediate representation**, 也叫**IR**）中，这些中间代码与源文件或目的文件形式都没有紧密的联系（因此叫作 "中间" ）。相反，IR充当了这两种语言之间的接口^3。

> This lets you support multiple source languages and target platforms with less effort. Say you want to implement Pascal, C and Fortran compilers and you want to target x86, ARM, and, I dunno, SPARC. Normally, that means you're signing up to write *nine* full compilers: Pascal→x86, C→ARM, and every other combination.

这可以让你更轻松地支持多种源语言和目标平台。假设你想在x86、ARM、SPARC 平台上实现Pascal、C和Fortran编译器。通常情况下，这意味着你需要写九个完整的编译器：Pascal→x86， C→ARM，以及其他各种组合^4。

> A shared intermediate representation reduces that dramatically. You write *one* front end for each source language that produces the IR. Then *one* back end for each target architecture. Now you can mix and match those to get every combination.

一个共享的中间代码可以大大减少这种情况。你为每个产生IR的源语言写一个前端。然后为每个目标平台写一个后端。现在，你可以将这些混搭起来，得到每一种组合。

> There's another big reason we might want to transform the code into a form that makes the semantics more apparent...

我们希望将代码转化为某种语义更加明确的形式，还有一个重要的原因是。。。

> ## 2.1.5 Optimization

2.1.5 优化

> Once we understand what the user's program means, we are free to swap it out with a different program that has the *same semantics* but implements them more efficiently—we can **optimize** it.

一旦我们理解了用户程序的含义，我们就可以自由地用另一个具有相同语义但实现效率更高的程序来交换它——我们可以对它进行**优化**。

> A simple example is **constant folding**: if some expression always evaluates to the exact same value, we can do the evaluation at compile time and replace the code for the expression with its result. If the user typed in:

一个简单的例子是**常量折叠**：如果某个表达式求值得到的始终是完全相同的值，我们可以在编译时进行求值，并用其结果替换该表达式的代码。 如果用户输入：

```
pennyArea = 3.14159 * (0.75 / 2) * (0.75 / 2);
```

> We can do all of that arithmetic in the compiler and change the code to:

我们可以在编译器中完成所有的算术运算，并将代码更改为：

```
pennyArea = 0.4417860938;
```

> Optimization is a huge part of the programming language business. Many language hackers spend their entire careers here, squeezing every drop of performance they can out of their compilers to get their benchmarks a fraction of a percent faster. It can become a sort of obsession.

优化是编程语言业务的重要组成部分。许多语言黑客把他们的整个职业生涯都花在了这里，竭尽所能地从他们的编译器中挤出每一点性能，以使他们的基准测试速度提高百分之几。有的时候这也会变成一种痴迷, 无法自拔。

> We're mostly going to hop over that rathole in this book. Many successful languages have surprisingly few compile-time optimizations. For example, Lua and CPython generate relatively unoptimized code, and focus most of their performance effort on the runtime.

我们在本书中通常会跳过这些棘手问题。令人惊讶的是许多成功的语言只有很少的编译期优化。 例如，Lua和CPython生成没怎么优化过的代码，并将其大部分性能工作集中在运行时上^5。

> ### 2.1.6 Code generation

2.1.6 代码生成

> We have applied all of the optimizations we can think of to the user's program. The last step is converting it to a form the machine can actually run. In other words **generating code** (or **code gen**), where "code" here usually refers to the kind of primitive assembly-like instructions a CPU runs and not the kind of "source code" a human might want to read.

我们已经将所有可以想到的优化应用到了用户程序中。 最后一步是将其转换为机器可以实际运行的形式。 换句话说，**生成代码**（或**代码生成**），这里的"代码"通常是指CPU运行的类似于汇编的原始指令，而不是人类可能想要阅读的"源代码"。

> Finally, we are in the **back end**, descending the other side of the mountain. From here on out, our representation of the code becomes more and more primitive, like evolution run in reverse, as we get closer to something our simple-minded machine can understand.

最后，我们到了**后端**，从山的另一侧开始向下。 从现在开始，随着我们越来越接近于思维简单的机器可以理解的东西，我们对代码的表示变得越来越原始，就像逆向进化。

> We have a decision to make. Do we generate instructions for a real CPU or a virtual one? If we generate real machine code, we get an executable that the OS can load directly onto the chip. Native code is lightning fast, but generating it is a lot of work. Today's architectures have piles of instructions, complex pipelines, and enough historical baggage to fill a 747's luggage bay.

我们需要做一个决定。 我们是为真实CPU还是虚拟CPU生成指令？ 如果我们生成真实的机器代码，则会得到一个可执行文件，操作系统可以将其直接加载到芯片上。 原生代码快如闪电，但生成它需要大量工作。 当今的体系结构包含大量指令，复杂的管线和足够塞满一架747行李舱的历史包袱。

> Speaking the chip's language also means your compiler is tied to a specific architecture. If your compiler targets x86 machine code, it's not going to run on an ARM device. All the way back in the 60s, during the Cambrian explosion of computer architectures, that lack of portability was a real obstacle.

使用芯片的语言也意味着你的编译器是与特定的架构相绑定的。如果你的编译器以x86机器代码为目标，那么它就无法在ARM设备上运行。追朔到上世纪60年代计算机体系结构 “寒武纪大爆发” 期间，这种缺乏可移植性的情况是一个真正的障碍^6。

> To get around that, hackers like Martin Richards and Niklaus Wirth, of BCPL and Pascal fame, respectively, made their compilers produce *virtual* machine code. Instead of instructions for some real chip, they produced code for a hypothetical, idealized machine. Wirth called this **"p-code"** for "portable", but today, we generally call it **bytecode** because each instruction is often a single byte long.

为了解决这个问题，专家开始让他们的编译器生成虚拟机代码，包括BCPL的设计者Martin Richards以及Pascal设计者Niklaus Wirth。他们不是为真正的芯片编写指令，而是为一个假设的、理想化的机器编写代码。Wirth称这种**p-code**为"可移植代码"，但今天，我们通常称它为**字节码**，因为每条指令通常都是一个字节长。

> These synthetic instructions are designed to map a little more closely to the language's semantics, and not be so tied to the peculiarities of any one computer architecture and its accumulated historical cruft. You can think of it like a dense, binary encoding of the language's low-level operations.

这些合成指令的设计是为了更紧密地映射到语言的语义上，而不必与任何一个计算机体系结构的特性和它积累的历史错误绑定在一起。你可以把它想象成语言底层操作的密集二进制编码。

> ### 2.1.7 Virtual machine

2.1.7 虚拟机

> If your compiler produces bytecode, your work isn't over once that's done. Since there is no chip that speaks that bytecode, it's your job to translate. Again, you have two options. You can write a little mini-compiler for each target architecture that converts the bytecode to native code for that machine. You still have to do work for each chip you support, but this last stage is pretty simple and you get to reuse the rest of the compiler pipeline across all of the machines you support. You're basically using your bytecode as an intermediate representation.

如果你的编译器产生了字节码，你的工作还没有结束。因为没有芯片可以解析这些字节码，因此你还需要进行翻译。同样，你有两个选择。你可以为每个目标体系结构编写一个小型编译器，将字节码转换为该机器的本机代码^7。你仍然需要针对你支持的每个芯片做一些工作，但最后这个阶段非常简单，你可以在你支持的所有机器上重复使用编译器流水线的其余部分。你基本上是把你的字节码作为一种中间代码。

> Or you can write a **virtual machine** (**VM**), a program that emulates a hypothetical chip supporting your virtual architecture at runtime. Running bytecode in a VM is slower than translating it to native code ahead of time because every instruction must be simulated at runtime each time it executes. In return, you get simplicity and portability. Implement your VM in, say, C, and you can run your language on any platform that has a C compiler. This is how the second interpreter we build in this book works.

或者，你可以编写**虚拟机（VM）**^8，该程序可在运行时模拟支持虚拟架构的虚拟芯片。在虚拟机中运行字节码比提前将其翻译成本地代码要慢，因为每条指令每次执行时都必须在运行时模拟。作为回报，你得到的是简单性和可移植性。用比如说C语言实现你的虚拟机，你就可以在任何有C编译器的平台上运行你的语言。这就是我们在本书中构建的第二个解释器的工作原理。

> ### 2.1.8 Runtime

2.1.8 运行时

> We have finally hammered the user's program into a form that we can execute. The last step is running it. If we compiled it to machine code, we simply tell the operating system to load the executable and off it goes. If we compiled it to bytecode, we need to start up the VM and load the program into that.

我们终于将用户程序锤炼成可以执行的形式。最后一步是运行它。如果我们将其编译为机器码，我们只需告诉操作系统加载可执行文件，然后就可以运行了。如果我们将它编译成字节码，我们需要启动VM并将程序加载到其中。

> In both cases, for all but the basest of low-level languages, we usually need some services that our language provides while the program is running. For example, if the language automatically manages memory, we need a garbage collector going in order to reclaim unused bits. If our language supports "instance of" tests so you can see what kind of object you have, then we need some representation to keep track of the type of each object during execution.

在这两种情况下，除了最基本的底层语言外，我们通常需要我们的语言在程序运行时提供一些服务。例如，如果语言自动管理内存，我们需要一个垃圾收集器去回收未使用的比特位。如果我们的语言支持用 "instance of "测试我们拥有什么类型的对象，那么我们就需要一些表示方法来跟踪执行过程中每个对象的类型。

> All of this stuff is going at runtime, so it's called, appropriately, the **runtime**. In a fully compiled language, the code implementing the runtime gets inserted directly into the resulting executable. In, say, Go, each compiled application has its own copy of Go's runtime directly embedded in it. If the language is run inside an interpreter or VM, then the runtime lives there. This is how most implementations of languages like Java, Python, and JavaScript work.

所有这些东西都是在运行时进行的，所以它被恰当地称为，**运行时**。在一个完全编译的语言中，实现运行时的代码会直接插入到生成的可执行文件中。比如说，在Go中，每个编译后的应用程序都有自己的一份Go的运行时副本直接嵌入其中。如果语言是在解释器或虚拟机内运行，那么运行时将驻留于虚拟机中。这也就是Java、Python和JavaScript等大多数语言实现的工作方式。

> ## 2.2 Shortcuts and Alternate Routes

2.2 捷径和备选路线

> That's the long path covering every possible phase you might implement. Many languages do walk the entire route, but there are a few shortcuts and alternate paths.

这是一条漫长的道路，涵盖了你要实现的每个可能的阶段。许多语言的确走完了整条路线，但也有一些捷径和备选路径。

> ### 2.2.1 Single-pass compilers

2.2.1 单遍编译器

> Some simple compilers interleave parsing, analysis, and code generation so that they produce output code directly in the parser, without ever allocating any syntax trees or other IRs. These **single-pass compilers** restrict the design of the language. You have no intermediate data structures to store global

> information about the program, and you don't revisit any previously parsed part of the code. That means as soon as you see some expression, you need to know enough to correctly compile it.

一些简单的编译器将解析、分析和代码生成交织在一起，这样它们就可以直接在解析器中生成输出代码，而无需分配任何语法树或其他IR。这些**单遍编译器**限制了语言的设计。你没有中间数据结构来存储程序的全局信息，也不会重新访问任何之前解析过的代码部分。 这意味着，一旦你看到某个表达式，就需要足够的知识来正确地对其进行编译^9。

> Pascal and C were designed around this limitation. At the time, memory was so precious that a compiler might not even be able to hold an entire *source file* in memory, much less the whole program. This is why Pascal's grammar requires type declarations to appear first in a block. It's why in C you can't call a function above the code that defines it unless you have an explicit forward declaration that tells the compiler what it needs to know to generate code for a call to the later function.

Pascal和C语言就是围绕这个限制而设计的。在当时，内存非常珍贵，一个编译器可能连整个源文件都无法存放在内存中，更不用说整个程序了。这也是为什么Pascal的语法要求类型声明要先出现在一个块中。这也是为什么在C语言中，你不能在定义函数的代码上面调用函数，除非你有一个明确的前向声明，告诉编译器它需要知道什么，以便生成调用后面函数的代码。

> ## 2.2.2 Tree-walk interpreters

2.2.2 树遍历解释器

> Some programming languages begin executing code right after parsing it to an AST (with maybe a bit of static analysis applied). To run the program, the interpreter traverses the syntax tree one branch and leaf at a time, evaluating each node as it goes.

有些编程语言在将代码解析为AST后就开始执行代码（可能应用了一点静态分析）。为了运行程序，解释器每次都会遍历语法树的一个分支和叶子，并在运行过程中计算每个节点。

> This implementation style is common for student projects and little languages, but is not widely used for general-purpose languages since it tends to be slow. Some people use "interpreter" to mean only these kinds of implementations, but others define that word more generally, so I'll use the inarguably explicit **"tree-walk interpreter"** to refer to these. Our first interpreter rolls this way.

这种实现风格在学生项目和小型语言中很常见，但在通用语言中并不广泛使用，因为它往往很慢。有些人使用"解释器"仅指这类实现，但其他人对"解释器"一词的定义更宽泛，因此我将使用没有歧义的"**树遍历解释器**"来指代这些实现。我们的第一个解释器就是这样工作的^10。

> ## 2.2.3 Transpilers

2.2.3 转译器

> Writing a complete back end for a language can be a lot of work. If you have some existing generic IR to target, you could bolt your front end onto that. Otherwise, it seems like you're stuck. But what if you treated some other *source language* as if it were an intermediate representation?

为一种语言编写一个完整的后端可能需要大量的工作。 如果你有一些现有的通用IR作为目标，则可以将前端转换到该IR上。 否则，你可能会陷入困境。 但是，如果你将某些其他*源语言*视为中间代码，该怎么办？

> You write a front end for your language. Then, in the back end, instead of doing all the work to *lower* the semantics to some primitive target language, you produce a string of valid source code for some other language that's about as high level as yours. Then, you use the existing compilation tools for *that* language as your escape route off the mountain and down to something you can execute.

你需要为你的语言编写一个前端。然后，在后端，你可以生成一份与你的语言级别差不多的其他语言的有效源代码字符串，而不是将所有代码*降低*到某个原始目标语言的语义。然后，你可以使用该语言现有的编译工具作为逃离大山的路径，得到某些可执行的内容。

> They used to call this a **source-to-source compiler** or a **transcompiler**. After the rise of languages that compile to JavaScript in order to run in the browser, they've affected the hipster sobriquet **transpiler**.

人们过去称之为**源到源编译器**或**转换编译器**^11。随着那些为了在浏览器中运行而编译成JavaScript的各类语言的兴起，它们有了一个时髦的名字——**转译器**。

> While the first transcompiler translated one assembly language to another, today, most transpilers work on higher-level languages. After the viral spread of UNIX to machines various and sundry, there began a long tradition of compilers that produced C as their output language. C compilers were available everywhere UNIX was and produced efficient code, so targeting C was a good way to get your language running on a lot of architectures.

虽然第一个编译器是将一种汇编语言翻译成另一种汇编语言，但现今，大多数编译器都适用于高级语言。在UNIX广泛运行在各种各样的机器上之后，编译器开始长期以C作为输出语言。C编译器在UNIX存在的地方都可以使用，并能生成有效的代码，因此，以C为目标是让语言在许多体系结构上运行的好方法。

> Web browsers are the "machines" of today, and their "machine code" is JavaScript, so these days it seems almost every language out there has a compiler that targets JS since that's the main way to get your code running in a browser.

Web浏览器是今天的 "机器"，它们的 "机器代码 "是JavaScript，所以现在似乎几乎所有的语言都有一个以JS为目标的编译器，因为这是让你的代码在浏览器中运行的主要方式^12。

> The front end—scanner and parser—of a transpiler looks like other compilers. Then, if the source language is only a simple syntactic skin over the target language, it may skip analysis entirely and go straight to outputting the analogous syntax in the destination language.

转译器的前端（扫描器和解析器）看起来跟其他编译器相似。 然后，如果源语言只是在目标语言在语法方面的换皮版本，则它可能会完全跳过分析，并直接输出目标语言中的类似语法。

> If the two languages are more semantically different, then you'll see more of the typical phases of a full compiler including analysis and possibly even optimization. Then, when it comes to code generation, instead of outputting some binary language like machine code, you produce a string of grammatically correct source (well, destination) code in the target language.

如果两种语言的语义差异较大，那么你就会看到完整编译器的更多典型阶段，包括分析甚至优化。然后，在代码生成阶段，无需输出一些像机器代码一样的二进制语言，而是生成一串语法正确的目标语言的源码（好吧，目标代码）。

> Either way, you then run that resulting code through the output language's existing compilation pipeline and you're good to go.

不管是哪种方式，你再通过目标语言已有的编译流水线运行生成的代码就可以了。

> ### 2.2.4 Just-in-time compilation

2.2.4 即时编译

> This last one is less of a shortcut and more a dangerous alpine scramble best reserved for experts. The fastest way to execute code is by compiling it to machine code, but you might not know what architecture your end user’s machine supports. What to do?

最后一个与其说是捷径，不如说是危险的高山争霸赛，最好留给专家。执行代码最快的方法是将代码编译成机器代码，但你可能不知道你的最终用户的机器支持什么架构。该怎么做呢？

> You can do the same thing that the HotSpot JVM, Microsoft’s CLR and most JavaScript interpreters do. On the end user’s machine, when the program is loaded—either from source in the case of JS, or platform-independent bytecode for the JVM and CLR—you compile it to native for the architecture their computer supports. Naturally enough, this is called **just-in-time compilation.** Most hackers just say “JIT”, pronounced like it rhymes with “fit”.

你可以做和HotSpot JVM、Microsoft的CLR和大多数JavaScript解释器相同的事情。 在终端用户的机器上，当程序加载时（无论是JS源代码还者是平台无关的JVM和CLR字节码），都可以将其编译为对应的本地代码，以适应本机支持的体系结构。 自然地，这被称为**即时编译**。大多数黑客只是说” JIT”，其发音与” fit”押韵。

> The most sophisticated JITs insert profiling hooks into the generated code to see which regions are most performance critical and what kind of data is flowing through them. Then, over time, they will automatically recompile those hot spots with more advanced optimizations.

最复杂的JIT将性能分析钩子插入到生成的代码中，以查看哪些区域对性能最为关键，以及哪些类型的数据正在流经其中。 然后，随着时间的推移，它们将通过更高级的优化功能自动重新编译那些热点部分^13。

> ## 2.3 Compilers and Interpreters

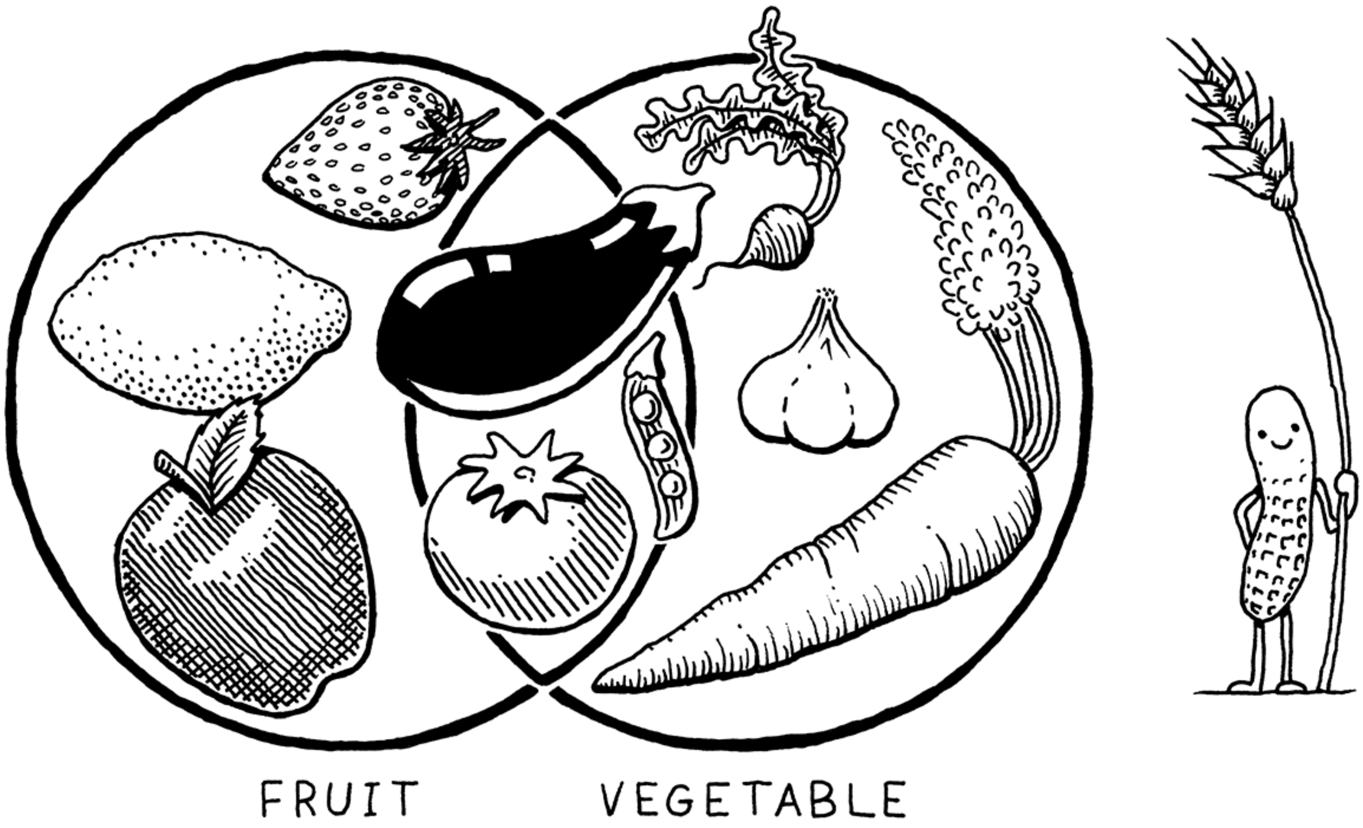
2.3 编译器和解释器

> Now that I’ve stuffed your head with a dictionary’s worth of programming language jargon, we can finally address a question that’s plagued coders since time immemorial: “What’s the difference between a compiler and an interpreter?”

现在我已经向你的脑袋里塞满了一大堆编程语言术语，我们终于可以解决一个自远古以来一直困扰着程序员的问题:编译器和解释器之间有什么区别?

> It turns out this is like asking the difference between a fruit and a vegetable. That seems like a binary either-or choice, but actually “fruit” is a *botanical* term and “vegetable” is *culinary*. One does not strictly imply the negation of the other. There are fruits that aren’t vegetables (apples) and vegetables that are not fruits (carrots), but also edible plants that are both fruits *and* vegetables, like tomatoes.

事实证明，这就像问水果和蔬菜的区别一样。这看上去似乎是一个非此即彼的选择，但实际上 "水果 "是一个植物学术语，"蔬菜 "是烹饪学术语。严格来说，一个并不意味着对另一个的否定。有不是蔬菜的水果（苹果），也有不是水果的蔬菜（胡萝卜），也有既是水果又是蔬菜的可食用植物，比如西红柿^14。



> So, back to languages:
>
> - **Compiling** is an *implementation technique* that involves translating a source language to some other—usually lower-level—form. When you generate bytecode or machine code, you are compiling. When you transpile to another high-level language you are compiling too.
> - When we say a language implementation "is a **compiler**", we mean it translates source code to some other form but doesn't execute it. The user has to take the resulting output and run it themselves.
> - Conversely, when we say an implementation "is an **interpreter**", we mean it takes in source code and executes it immediately. It runs programs "from source".

好，回到语言上：

- **编译**是一种实现技术，其中涉及到将源语言翻译成其他语言——通常是较低级的形式。当你生成字节码或机器代码时，你就是在编译。当你移植到另一种高级语言时，你也在编译。
- 当我们说语言实现"是**编译器**"时，是指它会将源代码转换为其他形式，但不会执行。 用户必须获取结果输出并自己运行。
- 相反，当我们说一个实现"是一个**解释器**"时，是指它接受源代码并立即执行它。 它"从源代码"运行程序。

> Like apples and oranges, some implementations are clearly compilers and *not* interpreters. GCC and Clang take your C code and compile it to machine code. An end user runs that executable directly and may never even know which tool was used to compile it. So those are *compilers* for C.

像苹果和橘子一样，某些实现显然是编译器，而不是解释器。 GCC和Clang接受你的C代码并将其编译为机器代码。 最终用户直接运行该可执行文件，甚至可能永远都不知道使用了哪个工具来编译它。 所以这些是C的*编译器*。

> In older versions of Matz' canonical implementation of Ruby, the user ran Ruby from source. The implementation parsed it and executed it directly by traversing the syntax tree. No other translation occurred, either internally or in any user-visible form. So this was definitely an *interpreter* for Ruby.

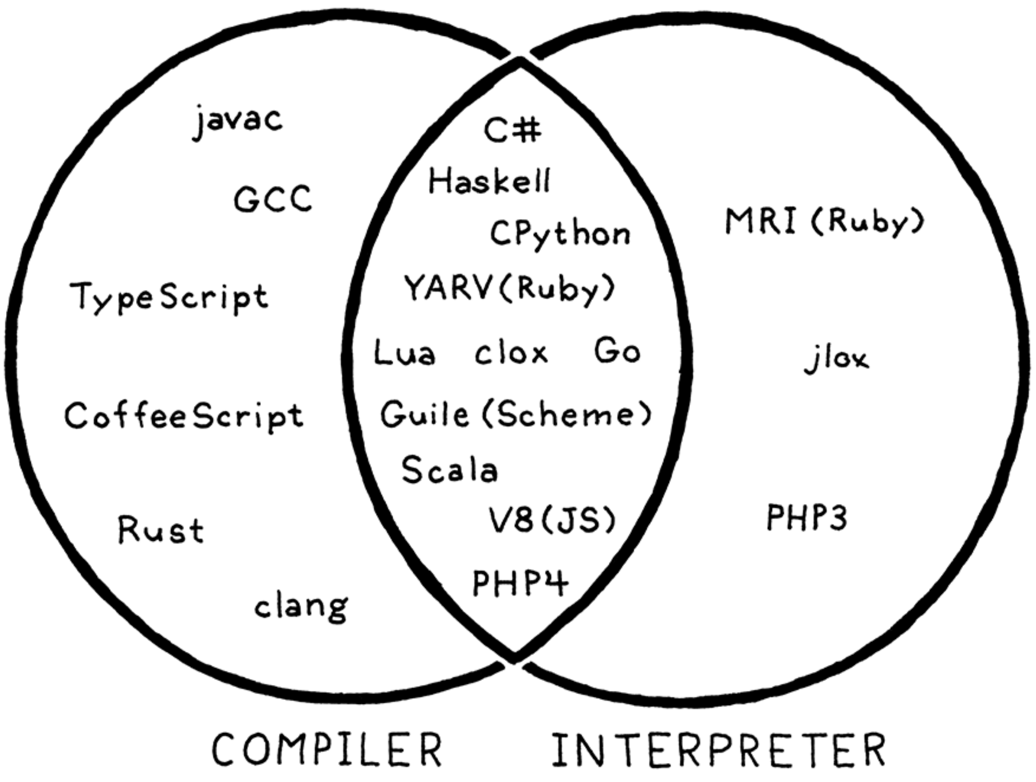
由 Matz 实现的老版本 Ruby 中，用户从源代码中运行Ruby。该实现通过遍历语法树对其进行解析并直接执行。期间都没有发生其他的转换，无论是在实现内部还是以任何用户可见的形式。所以这绝对是一个Ruby的*解释器*。

> But what of CPython? When you run your Python program using it, the code is parsed and converted to an internal bytecode format, which is then executed inside the VM. From the user's perspective, this is clearly an interpreter—they run their program from source. But if you look under CPython's scaly skin, you'll see that there is definitely some compiling going on.

但是 CPython 呢？当你使用它运行你的Python程序时，代码会被解析并转换为内部字节码格式，然后在虚拟机内部执行。从用户的角度来看，这显然是一个解释器——他们是从源代码开始运行自己的程序。但如果你看一下CPython的内部，你会发现肯定有一些编译工作在进行。

> The answer is that it is both. CPython *is* an interpreter, and it *has* a compiler. In practice, most scripting languages work this way, as you can see:

答案是两者兼而有之。 CPython是一个解释器，但他也*有*一个编译器。 实际上，大多数脚本语言都以这种方式工作^15，如你所见：



> That overlapping region in the center is where our second interpreter lives too, since it internally compiles to bytecode. So while this book is nominally about interpreters, we'll cover some compilation too.

中间那个重叠的区域也是我们第二个解释器所在的位置，因为它会在内部编译成字节码。所以，虽然本书名义上是关于解释器的，但我们也会涉及一些编译的内容。

2.4 Our Journey

2.4 我们的旅程

> That's a lot to take in all at once. Don't worry. This isn't the chapter where you're expected to *understand* all of these pieces and parts. I just want you to know that they are out there and roughly how they fit together.

一下子有太多东西要消化掉。别担心。这一章并不是要求你理解所有这些零碎的内容。我只是想让你们知道它们是存在的，以及大致了解它们是如何组合在一起的。

> This map should serve you well as you explore the territory beyond the guided path we take in this book. I want to leave you yearning to strike out on your own and wander all over that mountain.

当你探索本书本书所指导的路径之外的领域时，这张地图应该对你很有用。我希望你自己出击，在那座山里到处游走。

> But, for now, it's time for our own journey to begin. Tighten your bootlaces, cinch up your pack, and come along. From here on out, all you need to focus on is the path in front of you.

但是，现在，是我们自己的旅程开始的时候了。系好你的鞋带，背好你的包，走吧。从这里开始，你需要关注的是你面前的路。

^3: 有几种成熟的IR风格。点击你熟悉的搜索引擎，搜索 "控制流图"、"静态单赋值形式"、"延续传递形式 "和 "三位址码"。^4: 如果你曾经好奇GCC如何支持这么多疯狂的语言和体系结构，例如Motorola 68k上的Modula-3，现在你就明白了。 语言前端针对的是少数IR，主要是GIMPLE和RTL。目标后端如68k，会接受这些IR并生成本机代码。 ^5: 如果你无法抗拒要进入这个领域，可以从以下关键字开始，例如“常量折叠”，“公共表达式消除”，“循环不变代码外提”，“全局值编号”，“强度降低”，“ 聚合量标量替换”，“死码删除”和“循环展开”。^6: 例如，AAD（"ASCII Adjust AX Before Division",除法前ASCII调整AX）指令可以让你执行除法，这听起来很有用。除了该指令将两个二进制编码的十进制数字作为操作数打包到一个16位寄存器中。你最后一次在16位机器上使用BCD是什么时候？ ^7: 这里的基本原则是，你把特定于体系架构的工作推得越靠后，你就可以在不同架构之间共享更多的早期阶段。不过，这里存在一些矛盾。 许多优化（例如寄存器分配和指令选择）在了解特定芯片的优势和功能时才能发挥最佳效果。 弄清楚编译器的哪些部分可以共享，哪些应该针对特定目标是一门艺术。 ^8: 术语“虚拟机”也指另一种抽象。 “系统虚拟机”在软件中模拟整个硬件平台和操作系统。 这就是你可以在Linux机器上玩Windows游戏的原因，也是云提供商为什么可以给客户提供控制自己的“服务器”的用户体验，而无需为每个用户实际分配单独的计算机。在本书中，我们将要讨论的虚拟机类型是“语言虚拟机”或“进程虚拟机”（如果你需要明确的话）。 ^9: 语法导向翻译是一种结构化的技术，用于构建这些一次性编译器。你可以将一个操作与语法的每个片段(通常是生成输出代码的语法片段)相关联。然后，每当解析器匹配该语法块时，它就执行操作，一次构建一个规则的目标代码。 ^10: 一个明显的例外是早期版本的Ruby，它们是树遍历型解释器。在1.9时，Ruby的规范实现从最初的MRI（"Matz' Ruby Interpreter"）切换到了Koichi Sasada的YARV（"Yet Another Ruby VM"）。YARV是一个字节码虚拟机。 ^11: 第一个转编译器XLT86将8080程序集转换为8086程序集。 这看似简单，但请记住8080是8位芯片，而8086是16位芯片，可以将每个寄存器用作一对8位寄存器。XLT86进行了数据流分析，以跟踪源程序中的寄存器使用情况，然后将其有效地映射到8086的寄存器集。它是由悲惨的计算机科学英雄加里·基尔达尔（Gary Kildall）撰写的。 他是最早认识到微型计算机前景的人之一，他创建了PL / M和CP / M，这是它们的第一种高级语言和操作系统。 ^12: JS曾经是在浏览器中执行代码的唯一方式。多亏了Web Assembly，编译器现在有了第二种可以在Web上运行的低级语言。 ^13: 当然，这正是HotSpot JVM名称的来源。 ^14: 花生（连真正的坚果都算不上）和小麦等谷类其实都是水果，但我把这个图画错了。我能说什么呢，我是个软件工程师，不是植物学家。我也许应该抹掉这个花生小家伙，但他太可爱了，我不忍心。 ^15: Go工具更是一个奇葩。如果你运行`go build`，它就会把你的go源代码编译成机器代码然后停止。如果你输入`go run`，它也会这样做，然后立即执行生成的可执行文件。所以，可以说go是一个编译器（你

可以把它当做一个工具来编译代码而不运行）；也可以说是一个解释器（你可以调用它立即从源码中运行一个程序），并且有一个编译器（当你把它当做解释器使用时，它仍然在内部编译）。

CHALLENGES

习题

> 1、Pick an open source implementation of a language you like. Download the source code and poke around in it. Try to find the code that implements the scanner and parser. Are they hand-written, or generated using tools like Lex and Yacc? (`.l` or `.y` files usually imply the latter.)

1、选择一个你喜欢的语言的开源实现。下载源代码，并在其中探索。试着找到实现扫描器和解析器的代码，它们是手写的，还是用Lex和Yacc等工具生成的？（存在`.l`或`.y`文件通常意味着后者）

> 2、Just-in-time compilation tends to be the fastest way to implement a dynamically-typed language, but not all of them use it. What reasons are there to *not* JIT?

2、实时编译往往是实现动态类型语言最快的方法，但并不是所有的语言都使用它。有什么理由不采用JIT呢？

> 3、Most Lisp implementations that compile to C also contain an interpreter that lets them execute Lisp code on the fly as well. Why?

3、大多数可编译为C的Lisp实现也包含一个解释器，该解释器还使它们能够即时执行Lisp代码。 为什么？

3.Lox语言 The Lox Language

> What nicer thing can you do for somebody than make them breakfast?
>
> ——Anthony Bourdain

还有什么能比给别人做顿早餐，更能体现你对他的好呢？

> We'll spend the rest of this book illuminating every dark and sundry corner of the Lox language, but it seems cruel to have you immediately start grinding out code for the interpreter without at least a glimpse of what we're going to end up with.

我们将用本书的其余部分来照亮Lox语言的每一个黑暗和杂乱的角落，但如果让你在对目标一无所知的情况下，就立即开始为解释器编写代码，这似乎很残忍。

> At the same time, I don't want to drag you through reams of language lawyering and specification-ese before you get to touch your text editor. So this will be a gentle, friendly introduction to Lox. It will leave out a lot of details and edge cases. We've got plenty of time for those later.

与此同时，我也不想在您编码之前，就把您拖入大量的语言和规范术语中。所以这是一个温和、友好的Lox介绍，它会省去很多细节和边缘情况^1。后面我们有足够的时间来解决这些问题。

3.1 Hello, Lox

3.1 Hello, Lox

> Here's your very first taste of Lox:

下面是你对Lox的第一次体验：

```
// Your first Lox program!
print "Hello, world!";
```

> As that `//` line comment and the trailing semicolon imply, Lox's syntax is a member of the C family. (There are no parentheses around the string because `print` is a built-in statement, and not a library function.)

正如那句`//`行注释和后面的分号所暗示的那样，Lox的语法是C语言家族的成员之一。（因为`print`是一个内置语句，而不是库函数，所以字符串周围没有括号。）

> Now, I won't claim that C has a *great* syntax. If we wanted something elegant, we'd probably mimic Pascal or Smalltalk. If we wanted to go full Scandinavian-furniture-minimalism, we'd do a Scheme. Those all have their virtues.

这里，我并不是想说C语言具有出色的语法^2。如果我们想要一些优雅的东西，我们可能会模仿Pascal或Smalltalk。如果我们想要完全体现斯堪的纳维亚家具的极简主义风格，我们会实现一个Scheme。这些都有其优点。

> What C-like syntax has instead is something you'll find is often more valuable in a language: *familiarity*. I know you are already comfortable with that style because the two languages we'll be using to *implement* Lox—Java and C—also inherit it. Using a similar syntax for Lox gives you one less thing to learn.

但是，类C的语法所具有的反而是一些在语言中更有价值的东西：*熟悉度*。我知道你已经对这种风格很熟悉了，因为我们将用来实现Lox的两种语言——Java和C——也继承了这种风格。让Lox使用类似的语法，你就少了一件需要学习的事情。

> ## 3.2 A High-Level Language

3.2 高级语言

> While this book ended up bigger than I was hoping, it's still not big enough to fit a huge language like Java in it. In order to fit two complete implementations of Lox in these pages, Lox itself has to be pretty compact.

虽然这本书最终比我所希望的要大，但它仍然不够大，无法将Java这样一门庞大的语言放进去。为了在有限的篇幅里容纳两个完整的Lox实现，Lox本身必须相当紧凑。

> When I think of languages that are small but useful, what comes to mind are high-level "scripting" languages like JavaScript, Scheme, and Lua. Of those three, Lox looks most like JavaScript, mainly because most C-syntax languages do. As we'll learn later, Lox's approach to scoping hews closely to Scheme. The C flavor of Lox we'll build in Part III is heavily indebted to Lua's clean, efficient implementation.

当我想到那些小而有用的语言时，我脑海中浮现的是像JavaScript^3、Scheme和Lua这样的高级 "脚本 "语言。在这三种语言中，Lox看起来最像JavaScript，主要是因为大多数c语法语言都是这样的。稍后我们将了解到，

Lox的范围界定方法与Scheme密切相关。 我们将在第三部分中构建的C风格的Lox很大程度上借鉴了Lua的干净、高效的实现。

> Lox shares two other aspects with those three languages:

Lox与这三种语言有两个共同之处：

> ## 3.2.1 Dynamic typing

3.2.1 动态类型

> Lox is dynamically typed. Variables can store values of any type, and a single variable can even store values of different types at different times. If you try to perform an operation on values of the wrong type—say, dividing a number by a string—then the error is detected and reported at runtime.

Lox是动态类型的。变量可以存储任何类型的值，单个变量甚至可以在不同时间存储不同类型的值。如果尝试对错误类型的值执行操作（例如，将数字除以字符串），则会在运行时检测到错误并报告。

> There are plenty of reasons to like static types, but they don't outweigh the pragmatic reasons to pick dynamic types for Lox. A static type system is a ton of work to learn and implement. Skipping it gives you a simpler language and a shorter book. We'll get our interpreter up and executing bits of code sooner if we defer our type checking to runtime.

喜欢静态类型的原因有很多，但它们都比不上为Lox选择动态类型的实际原因^4。静态类型系统需要学习和实现大量的工作。跳过它会让你的语言更简单，也可以让本书更短。如果我们将类型检查推迟到运行时，我们将可以更快地启动解释器并执行代码。

> ## 3.2.2 Automatic memory management

3.2.2 自动内存管理

> High-level languages exist to eliminate error-prone, low-level drudgery and what could be more tedious than manually managing the allocation and freeing of storage? No one rises and greets the morning sun with, "I can't wait to figure out the correct place to call `free()` for every byte of memory I allocate today!"

高级语言的存在是为了消除容易出错的低级工作，还有什么比手动管理存储的分配和释放更繁琐的呢?没有人会抬起头来迎接早晨的阳光，"我迫不及待想找到正确的位置去调用`free()`方法，来释放掉今天我在内存中申请的每个字节！"

> There are two main techniques for managing memory: **reference counting** and **tracing garbage collection** (usually just called **"garbage collection"** or **"GC"**). Ref counters are much simpler to implement—I think that's why Perl, PHP, and Python all started out using them. But, over time, the limitations of ref counting become too troublesome. All of those languages eventually ended up adding a full tracing GC or at least enough of one to clean up object cycles.

有两种主要的内存管理技术：**引用计数**和**跟踪垃圾收集**（通常仅称为"**垃圾收集**"或" **GC**"）^5。引用计数器的实现要简单得多——我想这就是为什么Perl、PHP和Python一开始都使用该方式的原因。但是，随着时间的流逝，引用计数的限制变得太麻烦了。 所有这些语言最终都添加了完整的跟踪GC或至少一种足以清除对象循环引用的管理方式。

> Tracing garbage collection has a fearsome reputation. It *is* a little harrowing working at the level of raw memory. Debugging a GC can sometimes leave you seeing hex dumps in your dreams. But, remember, this book is about dispelling magic and slaying those monsters, so we *are* going to write our own garbage collector. I think you'll find the algorithm is quite simple and a lot of fun to implement.

追踪式垃圾收集是一个听起来就很可怕的名称。在原始内存的层面上工作是有点折磨人的。调试GC的时候会让你在梦中也能看到hex dumps。但是，请记住，这本书是关于驱散魔法和杀死那些怪物的，所以我们要写出自己的垃圾收集器。我想你会发现这个算法相当简单，而且实现起来很有趣。

3.3 Data Types

3.3 数据类型

In Lox's little universe, the atoms that make up all matter are the built-in data types. There are only a few:

在Lox的小宇宙中，构成所有物质的原子是内置的数据类型。只有几个：

> **Booleans –** You can't code without logic and you can't logic without Boolean values. "True" and "false", the yin and yang of software. Unlike some ancient languages that repurpose an existing type to represent truth and falsehood, Lox has a dedicated Boolean type. We may be roughing it on this expedition, but we aren't *savages*.
>
> There are two Boolean values, obviously, and a literal for each one:

Booleans——没有逻辑就不能编码，没有布尔值也就没有逻辑^6。"真"和"假"，就是软件的阴与阳。 与某些古老的语言重新利用已有类型来表示真假不同，Lox具有专用的布尔类型。在这次探险中，我们可能会有些粗暴，但我们不是野蛮人。

显然，有两个布尔值，每个值都有一个字面量：

```
true;  // Not false.
false; // Not *not* false.
```

> **Numbers –** Lox only has one kind of number: double-precision floating point. Since floating point numbers can also represent a wide range of integers, that covers a lot of territory, while keeping things simple.
>
> Full-featured languages have lots of syntax for numbers—hexadecimal, scientific notation, octal, all sorts of fun stuff. We'll settle for basic integer and decimal literals:

Numbers——Lox只有一种数字：双精度浮点数。 由于浮点数还可以表示各种各样的整数，因此可以覆盖很多领域，同时保持简单。

功能齐全的语言具有多种数字语法-十六进制，科学计数法，八进制和各种有趣的东西。 我们只使用基本的整数和十进制文字：

```
1234;  // An integer.
12.34; // A decimal number.
```

> **Strings –** We've already seen one string literal in the first example. Like most languages, they are enclosed in double quotes:

Strings——在第一个示例中，我们已经看到一个字符串字面量。 与大多数语言一样，它们用双引号引起来：

```
"I am a string";
"";    // The empty string.
"123"; // This is a string, not a number.
```

> As we'll see when we get to implementing them, there is quite a lot of complexity hiding in that innocuous sequence of characters.

我们在实现它们时会看到，在这个看起来无害的字符序列^7中隐藏了相当多的复杂性。

> **Nil –** There's one last built-in value who's never invited to the party but always seems to show up. It represents "no value". It's called "null" in many other languages. In Lox we spell it `nil`. (When we get to implementing it, that will help distinguish when we're talking about Lox's `nil` versus Java or C's `null`.)
>
> There are good arguments for not having a null value in a language since null pointer errors are the scourge of our industry. If we were doing a statically-typed language, it would be worth trying to ban it. In a dynamically-typed one, though, eliminating it is often more annoying than having it.

Nil——还有最后一个内置数据，它从未被邀请参加聚会，但似乎总是会出现。 它代表"没有价值"。在许多其他语言中称为"null"。在Lox中，我们将其拼写为`nil`。（当我们实现它时，这将有助于区分Lox的`nil`与Java或C的`null`）

有一些很好的理由表明在语言中不使用空值是合理的，因为空指针错误是我们行业的祸害。如果我们使用的是静态类型语言，那么禁止它是值得的。然而，在动态类型中，消除它往往比保留它更加麻烦。

> ## 3.4 Expressions

3.4 表达式

> If built-in data types and their literals are atoms, then **expressions** must be the molecules. Most of these will be familiar.

如果内置数据类型及其字面量是原子，那么表达式必须是分子。其中大部分大家都很熟悉。

> ### 3.4.1 Arithmetic

3.4.1 算术运算

> Lox features the basic arithmetic operators you know and love from C and other languages:

Lox具备了您从C和其他语言中了解到的基本算术运算符：

```
add + me;
subtract - me;
```

```
multiply * me;
divide / me;
```

> The subexpressions on either side of the operator are **operands**. Because there are *two* of them, these are called **binary** operators. (It has nothing to do with the ones-and-zeroes use of "binary".) Because the operator is fixed *in* the middle of the operands, these are also called **infix** operators as opposed to **prefix** operators where the operator comes before and **postfix** where it follows the operand.

操作符两边的子表达式都是**操作数**。因为有两个操作数，它们被称为**二元**运算符(这与二进制的1和0二元没有关联)。由于操作符固定在操作数的中间，因此也称为**中缀**操作符，相对的，还有**前缀**操作符(操作符在操作数前面)和**后缀**操作符(操作符在操作数后面)^8。

> One arithmetic operator is actually *both* an infix and a prefix one. The `-` operator can also be used to negate a number:

有一个数学运算符既是中缀运算符也是前缀运算符，`-`运算符可以对数字取负：

```
-negateMe;
```

> All of these operators work on numbers, and it's an error to pass any other types to them. The exception is the `+` operator—you can also pass it two strings to concatenate them.

所有这些操作符都是针对数字的，将任何其他类型操作数传递给它们都是错误的。唯一的例外是`+`运算符——你也可以传给它两个字符串将它们串接起来。

> ### 3.4.2 Comparison and equality

3.4.2 比较与相等

> Moving along, we have a few more operators that always return a Boolean result. We can compare numbers (and only numbers), using Ye Olde Comparison Operators:

接下来，我们有几个返回布尔值的操作符。我们可以使用旧的比较操作符来比较数字(并且只能比较数字)：

```
less < than;
lessThan <= orEqual;
greater > than;
greaterThan >= orEqual;
```

> We can test two values of any kind for equality or inequality:

我们可以测试两个任意类型的值是否相等：

```
1 == 2;         // false.
"cat" != "dog"; // true.
```

> Even different types:

即使是不同类型也可以：

```
314 == "pi"; // false.
```

> Values of different types are *never* equivalent:

不同类型的值*永远不会*相等：

```
123 == "123"; // false.
```

> I'm generally against implicit conversions.

我通常是反对隐式转换的。

> ## 3.4.3 Logical operators

3.4.3 逻辑运算

> The not operator, a prefix `!`, returns `false` if its operand is true, and vice versa:

取非操作符，是前缀操作符`!`，如果操作数是true，则返回false，反之亦然：

```
!true;  // false.
!false; // true.
```

> The other two logical operators really are control flow constructs in the guise of expressions. An `and` expression determines if two values are *both* true. It returns the left operand if it's false, or the right operand otherwise:

其他两个逻辑操作符实际上是表达式伪装下的控制流结构。`and`表达式用于确认两个操作数是否 *都是*true。如果左侧操作数是false，则返回左侧操作数，否则返回右侧操作数：

```
true and false; // false.
true and true;  // true.
```

> And an `or` expression determines if *either* of two values (or both) are true. It returns the left operand if it is true and the right operand otherwise:

`or`表达式用于确认两个操作数中任意一个（或者都是）为true。如果左侧操作数为true，则返回左侧操作数，否则返回右侧操作数：

```
false or false; // false.
true or false;  // true.
```

> The reason and and or are like control flow structures is because they **short-circuit**. Not only does and return the left operand if it is false, it doesn't even *evaluate* the right one in that case. Conversely, ("contrapositively"?) if the left operand of an or is true, the right is skipped.

and和 or之所以像控制流结构，是因为它们会**短路**^9。如果左操作数为假，and不仅会返回左操作数，在这种情况下，它甚至不会计算右操作数。反过来，("相对的"?)如果or的左操作数为真，右操作数就会被跳过。

> ### 3.4.4 Precedence and grouping

3.4.4 优先级与分组

> All of these operators have the same precedence and associativity that you'd expect coming from C. (When we get to parsing, we'll get *way* more precise about that.) In cases where the precedence isn't what you want, you can use () to group stuff:

所有这些操作符都具有与c语言相同的优先级和结合性(当我们开始解析时，会进行更详细的说明)。在优先级不满足要求的情况下，你可以使用()来分组：

```
var average = (min + max) / 2;
```

> Since they aren't very technically interesting, I've cut the remainder of the typical operator menagerie out of our little language. No bitwise, shift, modulo, or conditional operators. I'm not grading you, but you will get bonus points in my heart if you augment your own implementation of Lox with them.

我把其他典型的操作符从我们的小语言中去掉了，因为它们在技术上不是很有趣。没有位运算、移位、取模或条件运算符。我不是在给你打分，但如果你通过自己的方式来完成支持这些运算的Lox实现，你会在我心中得到额外的加分。

> Those are the expression forms (except for a couple related to specific features that we'll get to later), so let's move up a level.

这些都是表达式形式(除了一些与我们将在后面介绍的特定特性相关的)，所以让我们继续。

> ## 3.5 Statements

3.5 语句

> Now we're at statements. Where an expression's main job is to produce a *value*, a statement's job is to produce an *effect*. Since, by definition, statements don't evaluate to a value, to be useful they have to otherwise change the world in some way—usually modifying some state, reading input, or producing output.

现在我们来看语句。表达式的主要作用是产生一个*值*，语句的主要作用是产生一个*效果*。由于根据定义，语句不求值，因此必须以某种方式改变世界（通常是修改某些状态，读取输入或产生输出）才能有用。

> You've seen a couple of kinds of statements already. The first one was:

您已经看到了几种语句。 第一个是：

```
print "Hello, world!";
```

> A `print` statement evaluates a single expression and displays the result to the user. You've also seen some statements like:

`print`语句计算单个表达式并将结果显示给用户^10。您还看到了一些语句，例如：

```
"some expression";
```

> An expression followed by a semicolon (`;`) promotes the expression to statement-hood. This is called (imaginatively enough), an **expression statement**.

表达式后跟分号（;）可以将表达式提升为语句状态。这被称为(很有想象力)**表达式语句**。

> If you want to pack a series of statements where a single one is expected, you can wrap them up in a block:

如果您想将一系列语句打包成一个语句，那么可以将它们打包在一个块中：

```
{
  print "One statement.";
  print "Two statements.";
}
```

> Blocks also affect scoping, which leads us to the next section…

块还会影响作用域，我们将在下一节中进行说明。

3.6 Variables

3.6 变量

> You declare variables using `var` statements. If you omit the initializer, the variable's value defaults to `nil`:

你可以使用`var`语句声明变量。如果你省略了初始化操作，变量的值默认为`nil`^11：

```
var imAVariable = "here is my value";
var iAmNil;
```

> Once declared, you can, naturally, access and assign a variable using its name:

一旦声明完成，你自然就可以通过变量名对其进行访问和赋值：

```
var breakfast = "bagels";
print breakfast; // "bagels".
breakfast = "beignets";
print breakfast; // "beignets".
```

> I won’t get into the rules for variable scope here, because we’re going to spend a surprising amount of time in later chapters mapping every square inch of the rules. In most cases, it works like you expect coming from C or Java.

我不会在这里讨论变量作用域的规则，因为我们在后面的章节中将会花费大量的时间来详细讨论这些规则。在大多数情况下，它的工作方式与您期望的C或Java一样。

> ## 3.7 Control Flow

3.7 控制流

> It’s hard to write useful programs if you can’t skip some code, or execute some more than once. That means control flow. In addition to the logical operators we already covered, Lox lifts three statements straight from C.

如果你不能跳过某些代码，或者不能多次执行某些代码，就很难写出有用的程序^12。这意味着控制流。除了我们已经介绍过的逻辑运算符之外，Lox直接从C中借鉴了三条语句。

> An `if` statement executes one of two statements based on some condition:

`if`语句根据某些条件执行两条语句中的一条：

```
if (condition) {
  print "yes";
} else {
  print "no";
}
```

> A `while` loop executes the body repeatedly as long as the condition expression evaluates to true:

只要条件表达式的计算结果为true，`while`循环就会重复执行循环体^13：

```
var a = 1;
while (a < 10) {
  print a;
  a = a + 1;
}
```

> Finally, we have `for` loops:

最后，还有for循环：

```
for (var a = 1; a < 10; a = a + 1) {
  print a;
}
```

> This loop does the same thing as the previous `while` loop. Most modern languages also have some sort of `for-in` or `foreach` loop for explicitly iterating over various sequence types. In a real language, that's nicer than the crude C-style `for` loop we got here. Lox keeps it basic.

这个循环与之前的 while 循环做同样的事情。大多数现代语言也有某种for-in或foreach循环，用于显式迭代各种序列类型^14。在真正的语言中，这比我们在这里使用的粗糙的C-风格for循环要好。Lox只保持了它的基本功能。

> ## 3.8 Functions

3.8 函数

> A function call expression looks the same as it does in C:

函数调用表达式与C语言中一样：

```
makeBreakfast(bacon, eggs, toast);
```

> You can also call a function without passing anything to it:

你也可以在不传递任何参数的情况下调用一个函数：

```
makeBreakfast();
```

> Unlike, say, Ruby, the parentheses are mandatory in this case. If you leave them off, it doesn't *call* the function, it just refers to it.

与Ruby不同的是，在本例中括号是强制性的。如果你把它们去掉，就不会调用函数，只是指向该函数。

> A language isn't very fun if you can't define your own functions. In Lox, you do that with `fun`:

如果你不能定义自己的函数，一门语言就不能算有趣。在Lox里，你可以通过`fun`完成：

```
fun printSum(a, b) {
  print a + b;
}
```

> Now's a good time to clarify some terminology. Some people throw around "parameter" and "argument" like they are interchangeable and, to many, they are. We're going to spend a lot of time splitting the finest of downy hairs around semantics, so let's sharpen our words. From here on out:
>
> - An **argument** is an actual value you pass to a function when you call it. So a function *call* has an *argument* list. Sometimes you hear **actual parameter** used for these.
> - A **parameter** is a variable that holds the value of the argument inside the body of the function. Thus, a function *declaration* has a *parameter* list. Others call these **formal parameters** or simply **formals**.

现在是澄清一些术语的好时机^15。有些人把 "parameter "和 "argument "混为一谈，好像它们可以互换，而对许多人来说，它们确实可以互换。我们要花很多时间围绕语义学来对其进行分辨，所以让我们在这里把话说清楚：

- **argument**是你在调用函数时传递给它的实际值。所以一个函数 *调用*有一个*argument*列表。有时你会听到有人用**实际参数**指代这些参数。
- **parameter**是一个变量，用于在函数的主体里面存放参数的值。因此，一个函数*声明*有一个*parameter*列表。也有人把这些称为**形式参数**或者干脆称为**形参**。

> The body of a function is always a block. Inside it, you can return a value using a `return` statement:

函数体总是一个块。在其中，您可以使用`return`语句返回一个值：

```
fun returnSum(a, b) {
  return a + b;
}
```

> If execution reaches the end of the block without hitting a `return`, it implicitly returns `nil`.

如果执行到达代码块的末尾而没有`return`语句，则会隐式返回nil。

> ## 3.8.1 Closures

3.8.1 闭包

> Functions are *first class* in Lox, which just means they are real values that you can get a reference to, store in variables, pass around, etc. This works:

在Lox中，函数是一等公民，这意味着它们都是真实的值，你可以对这些值进行引用、存储在变量中、传递等等。下面的代码是有效的：

```
fun addPair(a, b) {
  return a + b;
}

fun identity(a) {
  return a;
}
```

```
print identity(addPair)(1, 2); // Prints "3".
```

> Since function declarations are statements, you can declare local functions inside another function:

由于函数声明是语句，所以可以在另一个函数中声明局部函数：

```
fun outerFunction() {
  fun localFunction() {
    print "I'm local!";
  }

  localFunction();
}
```

> If you combine local functions, first-class functions, and block scope, you run into this interesting situation:

如果将局部函数、头等函数和块作用域组合在一起，就会遇到这种有趣的情况：

```
fun returnFunction() {
  var outside = "outside";

  fun inner() {
    print outside;
  }

  return inner;
}

var fn = returnFunction();
fn();
```

> Here, `inner()` accesses a local variable declared outside of its body in the surrounding function. Is this kosher? Now that lots of languages have borrowed this feature from Lisp, you probably know the answer is yes.

在这里，`inner()`访问了在其函数体外的外部函数中声明的局部变量。这样可行吗?现在很多语言都从Lisp借鉴了这个特性，你应该也知道答案是肯定的。

> For that to work, `inner()` has to "hold on" to references to any surrounding variables that it uses so that they stay around even after the outer function has returned. We call functions that do this **closures**. These days, the term is often used for *any* first-class function, though it's sort of a misnomer if the function doesn't happen to close over any variables.

要做到这一点，`inner()`必须"保留"对它使用的任何周围变量的引用，这样即使在外层函数返回之后，这些变量仍然存在。我们把能做到这一点的函数称为**闭包**^16。现在，这个术语经常被用于任何头类函数，但是如果函

数没有在任何变量上闭包，那就有点用词不当了。

> As you can imagine, implementing these adds some complexity because we can no longer assume variable scope works strictly like a stack where local variables evaporate the moment the function returns. We're going to have a fun time learning how to make these work and do so efficiently.

可以想象，实现这些会增加一些复杂性，因为我们不能再假定变量作用域严格地像堆栈一样工作，在函数返回时局部变量就消失了。我们将度过一段有趣的时间来学习如何使这些工作，并有效地做到这一点。

> ## 3.9 Classes

3.9 类

> Since Lox has dynamic typing, lexical (roughly, "block") scope, and closures, it's about halfway to being a functional language. But as you'll see, it's *also* about halfway to being an object-oriented language. Both paradigms have a lot going for them, so I thought it was worth covering some of each.

因为Lox具有动态类型、词法(粗略地说，就是块)作用域和闭包，所以它离函数式语言只有一半的距离。但正如您将看到的，它离成为一种面向对象的语言也有一半的距离。这两种模式都有很多优点，所以我认为有必要分别介绍一下。

> Since classes have come under fire for not living up to their hype, let me first explain why I put them into Lox and this book. There are really two questions:

类因为没有达到其宣传效果而受到抨击，所以让我先解释一下为什么我把它们放到Lox和这本书中。这里实际上有两个问题：

> ### 3.9.1 Why might any language want to be object oriented?

3.9.1 为什么任何语言都想要面向对象？

> Now that object-oriented languages like Java have sold out and only play arena shows, it's not cool to like them anymore. Why would anyone make a *new* language with objects? Isn't that like releasing music on 8-track?

现在像Java这样的面向对象的语言已经销声匿迹了，只能在舞台上表演，喜欢它们已经不酷了。为什么有人要用对象来做一门新的语言呢？这不就像用磁带[^21]发行音乐一样吗？

> It is true that the "all inheritance all the time" binge of the 90s produced some monstrous class hierarchies, but object-oriented programming is still pretty rad. Billions of lines of successful code have been written in OOP languages, shipping millions of apps to happy users. Likely a majority of working programmers today are using an object-oriented language. They can't all be *that* wrong.

90年代的 "一直都是继承 "的狂潮确实产生了一些畸形的类层次结构，但面向对象的编程还是很流行的。数十亿行成功的代码都是用OOP语言编写的，为用户提供了数百万个应用程序。很可能今天大多数在职程序员都在使用面向对象语言。他们不可能都错得那么离谱。

> In particular, for a dynamically-typed language, objects are pretty handy. We need *some* way of defining compound data types to bundle blobs of stuff together.

特别是，对于动态类型语言来说，对象是非常方便的。我们需要某种方式来定义复合数据类型，用来将一堆数据组合在一起。

> If we can also hang methods off of those, then we avoid the need to prefix all of our functions with the name of the data type they operate on to avoid colliding with similar functions for different types. In, say, Racket, you end up having to name your functions like `hash-copy` (to copy a hash table) and `vector-copy` (to copy a vector) so that they don't step on each other. Methods are scoped to the object, so that problem goes away.

如果我们也能把方法挂在这些对象上，那么我们就不需要把函数操作的数据类型的名字作为函数名称的前缀，以避免与不同类型的类似函数发生冲突。比如说，在Racket中，你最终不得不将你的函数命名为hash-copy(复制一个哈希表)和vector-copy(复制一个向量)，这样它们就不会互相覆盖。方法的作用域是对象，所以这个问题就不存在了。

> ### 3.9.2 Why is Lox object oriented?

3.9.2 为什么Lox是面向对象的？

> I could claim objects are groovy but still out of scope for the book. Most programming language books, especially ones that try to implement a whole language, leave objects out. To me, that means the topic isn't well covered. With such a widespread paradigm, that omission makes me sad.

我可以说对象确实很吸引人，但仍然超出了本书的范围。大多数编程语言的书籍，特别是那些试图实现一门完整语言的书籍，都忽略了对象。对我来说，这意味着这个主题没有被很好地覆盖。对于如此广泛使用的范式，这种遗漏让我感到悲伤。

> Given how many of us spend all day *using* OOP languages, it seems like the world could use a little documentation on how to *make* one. As you'll see, it turns out to be pretty interesting. Not as hard as you might fear, but not as simple as you might presume, either.

鉴于我们很多人整天都在使用OOP语言，似乎这个世界应该有一些关于如何制作OOP语言的文档。正如你将看到的那样，事实证明这很有趣。没有你担心的那么难，但也没有你想象的那么简单。

> ### 3.9.3 Classes or prototypes?

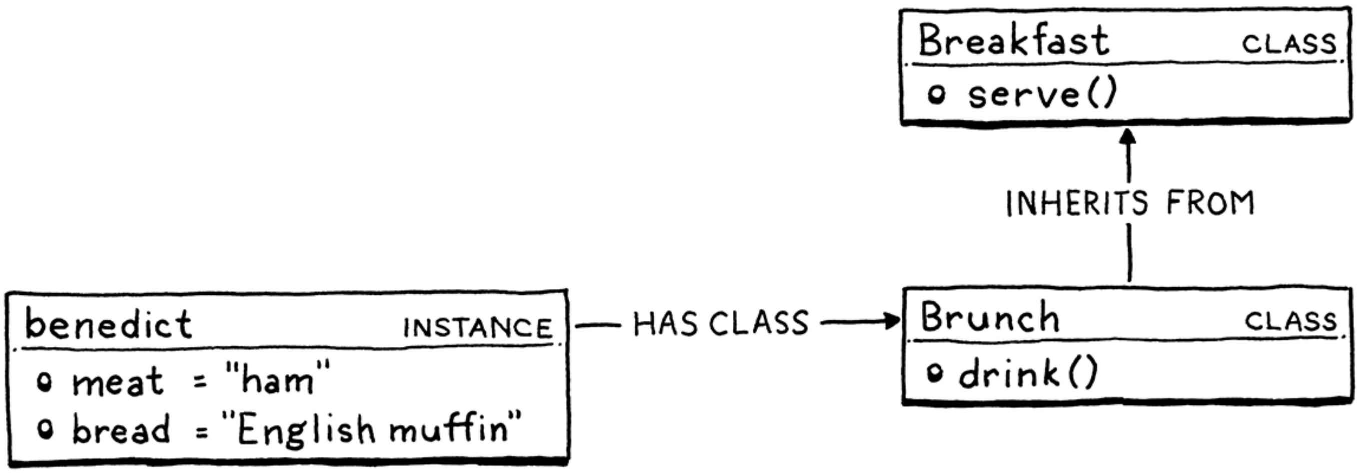
3.9.3 类还是原型？

> When it comes to objects, there are actually two approaches to them, classes and prototypes. Classes came first, and are more common thanks to C++, Java, C#, and friends. Prototypes were a virtually forgotten offshoot until JavaScript accidentally took over the world.

当涉及对象时，实际上有两种方法，类和原型。类最先出现，由于C++、Java、C#和其它近似语言的出现，类更加普遍。直到JavaScript意外地占领了世界之前，原型几乎是一个被遗忘的分支。

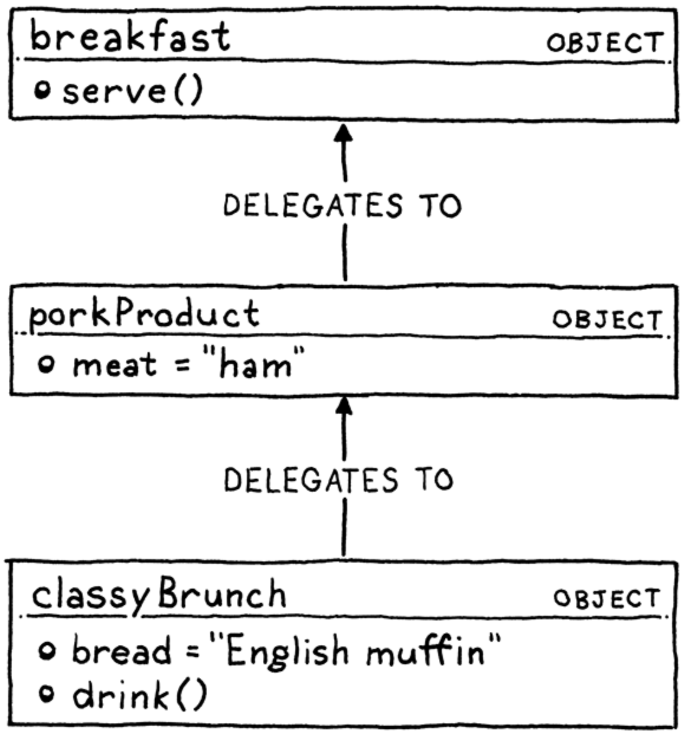
> In a class-based language, there are two core concepts: instances and classes. Instances store the state for each object and have a reference to the instance's class. Classes contain the methods and inheritance chain. To call a method on an instance, there is always a level of indirection. You look up the instance's class and then you find the method *there*:

在基于类的语言中，有两个核心概念：实例和类。 实例存储每个对象的状态，并有一个对实例的类的引用。 类包含方法和继承链。要在实例上调用方法，总是存在一个中间层。您要先查找实例的类，然后在其中找到方法：



> Prototype-based languages merge these two concepts. There are only objects—no classes—and each individual object may contain state and methods. Objects can directly inherit from each other (or “delegate to” in prototypal lingo):

基于原型的语言融合了这两个概念^17。这里只有对象——没有类，而且每个对象都可以包含状态和方法。对象之间可以直接继承（或者用原型语言的术语说是 “委托”）：



> This means prototypal languages are more fundamental in some way than classes. They are really neat to implement because they’re *so* simple. Also, they can express lots of unusual patterns that classes steer you away from.

这意味着原型语言在某些方面比类更基础。 它们实现起来真的很整洁，因为它们很简单。另外，它们还可以表达很多不寻常的模式，而这些模式是类所不具备的。

> But I've looked at a *lot* of code written in prototypal languages—including some of my own devising. Do you know what people generally do with all of the power and flexibility of prototypes? ...They use it to reinvent classes.

但是我看过很多用原型语言写的代码——包括我自己设计的一些代码。你知道人们一般会怎么使用原型的强大功能和灵活性吗？...他们用它来重新发明类。

> I don't know *why* that is, but people naturally seem to prefer a class-based ("Classic"? "Classy"?) style. Prototypes *are* simpler in the language, but they seem to accomplish that only by pushing the complexity onto the user. So, for Lox, we'll save our users the trouble and bake classes right in.

我不知道这是为什么，但人们自然而然地似乎更喜欢基于类的（经典？优雅？）风格。原型在语言中更简单，但它们似乎只是通过将复杂性推给用户来实现的^18。所以，对于Lox来说，我们将省去用户的麻烦，直接把类包含进去。

> ## 3.9.4 Classes in Lox

3.9.4 Lox中的类

> Enough rationale, let's see what we actually have. Classes encompass a constellation of features in most languages. For Lox, I've selected what I think are the brightest stars. You declare a class and its methods like so:

理由已经说够了，来看看我们实际上拥有什么。在大多数语言中，类包含了一系列的特性。对于Lox，我选择了我认为最闪亮的一点。您可以像这样声明一个类及其方法：

```
class Breakfast {
  cook() {
    print "Eggs a-fryin'!";
  }

  serve(who) {
    print "Enjoy your breakfast, " + who + ".";
  }
}
```

> The body of a class contains its methods. They look like function declarations but without the `fun` keyword. When the class declaration is executed, Lox creates a class object and stores that in a variable named after the class. Just like functions, classes are first class in Lox:

类的主体包含其方法。 它们看起来像函数声明，但没有`fun`关键字。 当类声明生效时，Lox将创建一个类对象，并将其存储在以该类命名的变量中。就像函数一样，类在Lox中也是一等公民：

```
// Store it in variables.
var someVariable = Breakfast;

// Pass it to functions.
someFunction(Breakfast);
```

> Next, we need a way to create instances. We could add some sort of new keyword, but to keep things simple, in Lox the class itself is a factory function for instances. Call a class like a function and it produces a new instance of itself:

接下来，我们需要一种创建实例的方法。我们可以添加某种new关键字，但为了简单起见，在Lox中，类本身是实例的工厂函数。像调用函数一样调用一个类，它会生成一个自己的新实例：

```
var breakfast = Breakfast();
print breakfast; // "Breakfast instance".
```

> ### 3.9.5 Instantiation and initialization

3.9.5 实例化和初始化

> Classes that only have behavior aren't super useful. The idea behind object-oriented programming is encapsulating behavior *and state* together. To do that, you need fields. Lox, like other dynamically-typed languages, lets you freely add properties onto objects:

只有行为的类不是非常有用。面向对象编程背后的思想是将行为和状态封装在一起。为此，您需要有字段。Lox和其他动态类型语言一样，允许您自由地向对象添加属性：

```
breakfast.meat = "sausage";
breakfast.bread = "sourdough";
```

> Assigning to a field creates it if it doesn't already exist.

如果一个字段不存在，那么对它进行赋值时就会先创建。

> If you want to access a field or method on the current object from within a method, you use good old this:

如果您想从方法内部访问当前对象上的字段或方法，可以使用this：

```
class Breakfast {
  serve(who) {
    print "Enjoy your " + this.meat + " and " +
        this.bread + ", " + who + ".";
  }

  // ...
}
```

> Part of encapsulating data within an object is ensuring the object is in a valid state when it's created. To do that, you can define an initializer. If your class has a method named init(), it is called

> automatically when the object is constructed. Any parameters passed to the class are forwarded to its initializer:

在对象中封装数据的目的之一是确保对象在创建时处于有效状态。为此，你可以定义一个初始化器。如果您的类中包含一个名为`init()`的方法，则在构造对象时会自动调用该方法。传递给类的任何参数都会转发给它的初始化器：

```
class Breakfast {
  init(meat, bread) {
    this.meat = meat;
    this.bread = bread;
  }

  // ...
}

var baconAndToast = Breakfast("bacon", "toast");
baconAndToast.serve("Dear Reader");
// "Enjoy your bacon and toast, Dear Reader."
```

> 3.9.6 Inheritance

3.9.6 继承

> Every object-oriented language lets you not only define methods, but reuse them across multiple classes or objects. For that, Lox supports single inheritance. When you declare a class, you can specify a class that it inherits from using a less-than (`<`) operator:

在每一种面向对象的语言中，你不仅可以定义方法，而且可以在多个类或对象中重用它们。为此，Lox支持单继承。当你声明一个类时，你可以使用小于(`<`)操作符指定它继承的类^19：

```
class Brunch < Breakfast {
  drink() {
    print "How about a Bloody Mary?";
  }
}
```

> Here, Brunch is the **derived class** or **subclass**, and Breakfast is the **base class** or **superclass**. Every method defined in the superclass is also available to its subclasses:

这里，Brunch是**派生类**或**子类**，而Breakfast是**基类**或**超类**。父类中定义的每个方法对其子类也可用：

```
var benedict = Brunch("ham", "English muffin");
benedict.serve("Noble Reader");
```

> Even the `init()` method gets inherited. In practice, the subclass usually wants to define its own `init()` method too. But the original one also needs to be called so that the superclass can maintain its state. We need some way to call a method on our own *instance* without hitting our own *methods*.

即使是init()方法也会被继承。在实践中，子类通常也想定义自己的`init()`方法。但还需要调用原始的初始化方法，以便超类能够维护其状态[^20]。我们需要某种方式能够调用自己实例上的方法，而无需触发实例自身的方法。

> As in Java, you use `super` for that:

与Java中一样，您可以使用`super`：

```
class Brunch < Breakfast {
  init(meat, bread, drink) {
    super.init(meat, bread);
    this.drink = drink;
  }
}
```

> That's about it for object orientation. I tried to keep the feature set minimal. The structure of the book did force one compromise. Lox is not a *pure* object-oriented language. In a true OOP language every object is an instance of a class, even primitive values like numbers and Booleans.

这就是面向对象的内容。我尽量将功能设置保持在最低限度。本书的结构确实迫使我做了一个妥协。Lox不是一种纯粹的面向对象的语言。在真正的OOP语言中，每个对象都是一个类的实例，即使是像数字和布尔值这样的基本类型。

> Because we don't implement classes until well after we start working with the built-in types, that would have been hard. So values of primitive types aren't real objects in the sense of being instances of classes. They don't have methods or properties. If I were trying to make Lox a real language for real users, I would fix that.

因为我们开始使用内置类型很久之后才会实现类，所以这一点很难实现。因此，从类实例的意义上说，基本类型的值并不是真正的对象。它们没有方法或属性。如果以后我想让Lox成为真正的用户使用的语言，我会解决这个问题。

> ## 3.10 The Standard Library

3.10 标准库

> We're almost done. That's the whole language, so all that's left is the "core" or "standard" library—the set of functionality that is implemented directly in the interpreter and that all user-defined behavior is built on top of.

我们快结束了，这就是整个语言，所剩下的就是"核心"或"标准"库——这是一组直接在解释器中实现的功能集，所有用户定义的行为都是建立在此之上。

> This is the saddest part of Lox. Its standard library goes beyond minimalism and veers close to outright nihilism. For the sample code in the book, we only need to demonstrate that code is running and

> doing what it’s supposed to do. For that, we already have the built-in `print` statement.

这是Lox中最可悲的部分。它的标准库已经超过了极简主义，接近彻底的虚无主义。对于本书中的示例代码，我们只需要证明代码在运行，并且在做它应该做的事。为此，我们已经有了内置的`print`语句。

> Later, when we start optimizing, we’ll write some benchmarks and see how long it takes to execute code. That means we need to track time, so we’ll define one built-in function `clock()` that returns the number of seconds since the program started.

稍后，当我们开始优化时，我们将编写一些基准测试，看看执行代码需要多长时间。这意味着我们需要跟踪时间，因此我们将定义一个内置函数`clock()`，该函数会返回程序启动后的秒数。

> And… that’s it. I know, right? It’s embarrassing.

嗯...就是这样。 我知道，有点尴尬，对吧？

> If you wanted to turn Lox into an actual useful language, the very first thing you should do is flesh this out. String manipulation, trigonometric functions, file I/O, networking, heck, even *reading input from the user* would help. But we don’t need any of that for this book, and adding it wouldn’t teach you anything interesting, so I left it out.

如果您想将Lox变成一门实际可用的语言，那么您应该做的第一件事就是对其充实。 字符串操作、三角函数、文件I / O、网络、扩展，甚至读取用户的输入都将有所帮助。但对于本书来说，我们不需要这些，而且加入这些也不会教给你任何有趣的东西，所以我把它省略了。

> Don’t worry, we’ll have plenty of exciting stuff in the language itself to keep us busy.

别担心，这门语言本身就有很多精彩的内容让我们忙个不停。

^2: 我肯定有偏见，但我认为Lox的语法很干净。 C语言最严重的语法问题就是关于类型的。丹尼斯·里奇（Dennis Ritchie）有个想法叫“声明反映使用”，其中变量声明反映了为获得基本类型的值而必须对变量执行的操作。这主意不错，但是我认为实践中效果不太好。Lox没有静态类型，所以我们避免了这一点。 ^3: 现在，JavaScript已席卷全球，并已用于构建大量应用程序，很难将其视为“小脚本语言”。但是Brendan Eich曾在十天内将第一个JS解释器嵌入了Netscape Navigator，以使网页上的按钮具有动画效果。 从那时起，JavaScript逐渐发展起来，但是它曾经是一种可爱的小语言。因为Eich大概只用了一集MacGyver的时间把JS糅合在一起，所以它有一些奇怪的语义，会有明显的拼凑痕迹。比如变量提升、动态绑定`this`、数组中的漏洞和隐式转换等。我有幸在Lox上多花了点时间，所以它应该更干净一些。 ^4: 毕竟，我们用于实现Lox的两种语言都是静态类型的。 ^5: 在实践中，引用计数和追踪更像是连续体的两端，而不是对立的双方。大多数引用计数系统最终会执行一些跟踪来处理循环，如果你仔细观察的话，分代收集器的写屏障看起来有点像保留调用。有关这方面的更多信息，请参阅垃圾收集统一理论(PDF)。^6: 布尔变量是Lox中唯一以人名George Boole命名的数据类型，这也是为什么 "Boolean "是大写的原因。他死于1864年，比数字计算机把他的代数变成电子信息的时间早了近一个世纪。我很好奇他看到自己的名字出现在数十亿行Java代码中时会怎么想。 ^7: 就连那个 "character "一词也是个骗局。是ASCII码？是Unicode？一个码点，还是一个 "字词群"？字符是如何编码的？每个字符是固定的大小，还是可以变化的？ ^8: 有些操作符有两个以上的操作数，并且操作符与操作数之间是交错的。唯一广泛使用的是C及其相近语言中的“条件”或“三元”操作符:`condition ?thenArm: elseArm;`，有些人称这些为mixfix操作符。有一些语言允许您定义自己的操作符，并控制它们的定位方式——它们的 "固定性"。。^9: 我使用了and和or，而不是&&和||，因为Lox不使用&和|作为位元操作符。不存在单字符形式的情况下引入双字符形式感觉很奇怪。我喜欢用单词来表示运算，也是因为它们实际上是控制流结构，而不是简单的操作符。 ^10: 将 print 融入到语言中，而不是仅仅将其作为一个核心库函数，这是一种入侵。但对我们来说，这是一个很有用的“入侵”：这意味着在我们实现所有定义函数、按名称查找和调用函数所需的机制之前，我们的解释器可以就开始产生输

出。 ^11: 这是一种情况，没有nil并强制每个变量初始化为某个值，会比处理nil本身更麻烦。 ^12: 我们已经有and和or可以进行分支处理，我们可以用递归来重复代码，所以理论上这就足够了。但是，在命令式语言中这样编程会很尴尬。另一方面，Scheme没有内置的循环结构。它确实依赖递归进行重复执行代码。Smalltalk没有内置的分支结构，并且依赖动态分派来选择性地执行代码。 ^13: 我没有在Lox中使用do-while循环，因为它们并不常见，相比while循环也没有多余的内涵。如果你高兴的话，就把它加入到你的实现中去吧。你自己做主。 ^14: 这是我做出的让步，因为本书中的实现是按章节划分的。for-in循环需要迭代器协议中的某种动态分派来处理不同类型的序列，但我们完成控制流之后才能实现这种分派。我们可以回过头来，添加for-in循环，但我认为这样做不会教给你什么超级有趣的东西。 ^15: 说到术语，一些静态类型的语言，比如C语言，会对函数的声明和定义进行区分。声明是将函数的类型和它的名字绑定在一起，所以调用时可以进行类型检查，但不提供函数体。定义也会填入函数的主体，这样就可以进行编译。由于Lox是动态类型的，所以这种区分没有意义。一个函数声明完全指定了函数，包括它的主体。 ^16: Peter J. Landin创造了这个词。没错，几乎一半的编程语言术语都是他创造的。它们中的大部分都出自一篇不可思议的论文 "The Next 700 Programming Languages"。为了实现这类函数，您需要创建一个数据结构，将函数代码和它所需要的周围变量绑定在一起。他称它为“闭包”，是因为函数“闭合”并保留了它需要的变量。 ^17: 实际上，基于类的语言和基于原型的语言之间的界限变得模糊了。JavaScript的“构造函数”概念使您很难定义类对象。 同时，基于类的Ruby非常乐意让您将方法附加到单个实例中。 ^18: Perl的发明家/先知Larry Wall将其称为“水床理论”。某些复杂性是必不可少的，无法消除。 如果在某个位置将其向下推，则在另一个位置会出现膨胀。原型语言并没有消除类的复杂性，因为它们确实让用户通过构建近似类的元编程库来承担这种复杂性。 ^19: 为什么用<操作符？我不喜欢引入一个新的关键字，比如extends。Lox不使用:来做其他事情，所以我也不想保留它。相反，我借鉴了Ruby的做法，使用了<。如果你了解任何类型理论，你会发现这并不是一个完全任意的选择。一个子类的每一个实例也是它的超类的一个实例，但可能有超类的实例不是子类的实例。这意味着，在对象的宇宙中，子类对象的集合比超类的集合要小，尽管类型迷们通常用<:来表示这种关系。 [^20]: Lox不同于不继承构造函数的c++、Java和c#，而是类似于Smalltalk和Ruby，它们继承了构造函数。 [^21]: 这里的8轨音乐指的是磁带。在中国大陆，通常“磁带”或者“录音带”一词都指紧凑音频盒带，因为它的应用非常广泛。在中国台湾，reel-to-reel tape被称为盘式录音带、紧凑音频盒带（Compact audio cassette）被称为卡式录音带、8轨软片（8-track cartridges）被称为匣式录音带。

CHALLENGES

习题

> 1、Write some sample Lox programs and run them (you can use the implementations of Lox in my repository). Try to come up with edge case behavior I didn’t specify here. Does it do what you expect? Why or why not?

1、编写一些示例Lox程序并运行它们(您可以使用我的Lox实现)。试着想出我在这里没有详细说明的边界情况。它是否按照期望运行？为什么？

> 2、This informal introduction leaves a *lot* unspecified. List several open questions you have about the language’s syntax and semantics. What do you think the answers should be?

2、这种非正式的介绍留下了很多未说明的东西。列出几个关于语言语法和语义的开放问题。你认为答案应该是什么？

> 3、Lox is a pretty tiny language. What features do you think it is missing that would make it annoying to use for real programs? (Aside from the standard library, of course.)

3、Lox是一种很小的语言。 您认为缺少哪些功能会使其不适用于实际程序？ （当然，除了标准库。）

DESIGN NOTE: EXPRESSIONS AND STATEMENTS

设计笔记：表达式和语句

> Lox has both expressions and statements. Some languages omit the latter. Instead, they treat declarations and control flow constructs as expressions too. These "everything is an expression" languages tend to have functional pedigrees and include most Lisps, SML, Haskell, Ruby, and CoffeeScript.
>
> To do that, for each "statement-like" construct in the language, you need to decide what value it evaluates to. Some of those are easy:
>
> - An `if` expression evaluates to the result of whichever branch is chosen. Likewise, a `switch` or other multi-way branch evaluates to whichever case is picked.
> - A variable declaration evaluates to the value of the variable.
> - A block evaluates to the result of the last expression in the sequence.
>
> Some get a little stranger. What should a loop evaluate to? A `while` loop in CoffeeScript evaluates to an array containing each element that the body evaluated to. That can be handy, or a waste of memory if you don't need the array.
>
> You also have to decide how these statement-like expressions compose with other expressions—you have to fit them into the grammar's precedence table. For example, Ruby allows:
>
> ```
> puts 1 + if true then 2 else 3 end + 4
> ```
>
> Is this what you'd expect? Is it what your *users* expect? How does this affect how you design the syntax for your "statements"? Note that Ruby has an explicit `end` to tell when the `if` expression is complete. Without it, the `+ 4` would likely be parsed as part of the `else` clause.
>
> Turning every statement into an expression forces you to answer a few hairy questions like that. In return, you eliminate some redundancy. C has both blocks for sequencing statements, and the comma operator for sequencing expressions. It has both the `if` statement and the `?:` conditional operator. If everything was an expression in C, you could unify each of those.
>
> Languages that do away with statements usually also feature **implicit returns**—a function automatically returns whatever value its body evaluates to without need for some explicit `return` syntax. For small functions and methods, this is really handy. In fact, many languages that do have statements have added syntax like `=>` to be able to define functions whose body is the result of evaluating a single expression.
>
> But making *all* functions work that way can be a little strange. If you aren't careful, your function will leak a return value even if you only intend it to produce a side effect. In practice, though, users of these languages don't find it to be a problem.
>
> For Lox, I gave it statements for prosaic reasons. I picked a C-like syntax for familiarity's sake, and trying to take the existing C statement syntax and interpret it like expressions gets weird pretty fast.

Lox既有表达式也有语句。有些语言省略了后者。相对地，它们将声明和控制流结构也视为表达式。这类 "一切都是表达式" 的语言往往具有函数式的血统，包括大多数Lisps、SML、Haskell、Ruby和CoffeeScript。

要做到这一点，对于语言中的每一个 "类似于语句" 的构造，你需要决定它所计算的值是什么。其中有些很简单：

- `if`表达式的计算结果是所选分支的结果。同样，`switch`或其他多路分支的计算结果取决于所选择的情况。
- 变量声明的计算结果是变量的值。
- 块的计算结果是序列中最后一个表达式的结果。

有一些是比较复杂的。循环应该计算什么值？在CoffeeScript中，一个`while`循环计算结果为一个数组，其中包含了循环体中计算到的每个元素。这可能很方便，但如果你不需要这个数组，就会浪费内存。

您还必须决定这些类似语句的表达式如何与其他表达式组合，必须将它们放入语法的优先表中。例如，Ruby允许下面这种写法：

```
puts 1 + if true then 2 else 3 end + 4
```

这是你所期望的吗？这是你的用户所期望的吗？这对你如何设计 "语句 "的语法有什么影响？请注意，Ruby有一个显式的`end`关键字来表明`if`表达式结束。如果没有它，`+4`很可能会被解析为 `else`子句的一部分。

把每个语句都转换成表达式会迫使你回答一些类似这样的复杂问题。作为回报，您消除了一些冗余。C语言中既有用于排序语句的块，以及用于排序表达式的逗号操作符。它既有`if`语句，也有`?:`条件操作符。如果在C语言中所有东西都是表达式，你就可以把它们统一起来。

取消了语句的语言通常还具有**隐式返回**的特点——函数自动返回其函数主体所计算得到的任何值，而不需要显式的`return`语法。对于小型函数和方法来说，这真的很方便。事实上，许多有语句的语言都添加了类似于 `=>` 的语法，以便能够定义函数体是计算单一表达式结果的函数。

但是让所有的函数以这种方式工作可能有点奇怪。即使你只是想让函数产生副作用，如果不小心，函数也可能会泄露返回值。但实际上，这些语言的用户并不觉得这是一个问题。

对于Lox，我在其中添加语句是出于朴素的原因。为了熟悉起见，我选择了一种类似于C的语法，而试图把现有的C语句语法像表达式一样解释，会很快变得奇怪。

4.扫描 Scanning

> Take big bites. Anything worth doing is worth overdoing.
>
> —— Robert A. Heinlein, *Time Enough for Love*

大干特工。每件值得做的事都要尽力做好。

> The first step in any compiler or interpreter is scanning. The scanner takes in raw source code as a series of characters and groups it into a series of chunks we call **tokens**. These are the meaningful "words" and "punctuation" that make up the language's grammar.

任何编译器或解释器的第一步都是扫描^1。扫描器以一系列字符的形式接收原始源代码，并将其分组成一系列的块，我们称之为**标识**（词法单元）。这些是有意义的 "单词 "和 "标点"，它们构成了语言的语法。

> Scanning is a good starting point for us too because the code isn't very hard—pretty much a `switch` statement with delusions of grandeur. It will help us warm up before we tackle some of the more interesting material later. By the end of this chapter, we'll have a full-featured, fast scanner that can take any string of Lox source code and produce the tokens that we'll feed into the parser in the next chapter.

对于我们来说，扫描也是一个很好的起点，因为代码不是很难——相当于有很多分支的`switch`语句。这可以帮助我们在学习更后面有趣的部分之前进行热身。在本章结束时，我们将拥有一个功能齐全、速度快的扫描器，它可以接收任何一串Lox源代码，并产生标记，我们将在下一章把这些标记输入到解析器中。

4.1 The Interpreter Framework

4.1 解释器框架

> Since this is our first real chapter, before we get to actually scanning some code we need to sketch out the basic shape of our interpreter, jlox. Everything starts with a class in Java.

由于这是我们的第一个真正的章节，在我们开始实际扫描代码之前，我们需要先勾勒出我们的解释器jlox的基本形态。在Java中，一切都是从一个类开始的。

【译者注：原作者在代码的侧边栏标注了代码名及对应的操作（创建文件、追加代码、删除代码等），由于翻译版的格式受限，将这部分信息迁移到代码块之前，以带下划线的斜体突出，后同】

lox/Lox.java，创建新文件^2

```java
package com.craftinginterpreters.lox;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class Lox {
  public static void main(String[] args) throws IOException {
    if (args.length > 1) {
      System.out.println("Usage: jlox [script]");
      System.exit(64);
    } else if (args.length == 1) {
      runFile(args[0]);
    } else {
      runPrompt();
    }
  }
}
```

> Stick that in a text file, and go get your IDE or Makefile or whatever set up. I'll be right here when you're ready. Good? OK!

把它贴在一个文本文件里，然后去把你的IDE或者Makefile或者其他工具设置好。我就在这里等你准备好。好了吗？好的！

> Lox is a scripting language, which means it executes directly from source. Our interpreter supports two ways of running code. If you start jlox from the command line and give it a path to a file, it reads the file and executes it.

Lox是一种脚本语言，这意味着它直接从源代码执行。我们的解释器支持两种运行代码的方式。如果从命令行启动jlox并为其提供文件路径，它将读取该文件并执行。

lox/Lox.java，添加到`main()`方法之后

```
private static void runFile(String path) throws IOException {
  byte[] bytes = Files.readAllBytes(Paths.get(path));
  run(new String(bytes, Charset.defaultCharset()));
}
```

> If you want a more intimate conversation with your interpreter, you can also run it interactively. Fire up jlox without any arguments, and it drops you into a prompt where you can enter and execute code one line at a time.

如果你想与你的解释器对话, 可以交互式的启动它。 启动的时候不加任何参数就可以了，它会有一个提示符，你可以在提示符处一次输入并执行一行代码。

lox/Lox.java，添加到`runFile()`方法之后^3

```
private static void runPrompt() throws IOException {
  InputStreamReader input = new InputStreamReader(System.in);
  BufferedReader reader = new BufferedReader(input);

  for (;;) {
    System.out.print("> ");
    String line = reader.readLine();
    if (line == null) break;
    run(line);
  }
}
```

> The `readLine()` function, as the name so helpfully implies, reads a line of input from the user on the command line and returns the result. To kill an interactive command-line app, you usually type Control-D. Doing so signals an “end-of-file” condition to the program. When that happens `readLine()` returns `null`, so we check for that to exit the loop.

`readLine()`函数，顾名思义，读取用户在命令行上的一行输入，并返回结果。要终止交互式命令行应用程序，通常需要输入Control-D。这样做会向程序发出 "文件结束" 的信号。当这种情况发生时，readLine()就会返回null，所以我们检查一下是否存在null以退出循环。

> Both the prompt and the file runner are thin wrappers around this core function:

交互式提示符和文件运行工具都是对这个核心函数的简单包装：

lox/Lox.java，添加到runPrompt()之后

```
  private static void run(String source) {
    Scanner scanner = new Scanner(source);
    List<Token> tokens = scanner.scanTokens();

    // For now, just print the tokens.
    for (Token token : tokens) {
      System.out.println(token);
    }
  }
```

> It's not super useful yet since we haven't written the interpreter, but baby steps, you know? Right now, it prints out the tokens our forthcoming scanner will emit so that we can see if we're making progress.

因为我们还没有写出解释器，所以这些代码还不是很有用，但这只是小步骤，你要明白？现在，它可以打印出我们即将完成的扫描器所返回的标记，这样我们就可以看到我们的解析是否生效。

> ### 4.1.1 Error handling

4.1.1 错误处理

> While we're setting things up, another key piece of infrastructure is *error handling*. Textbooks sometimes gloss over this because it's more a practical matter than a formal computer science-y problem. But if you care about making a language that's actually *usable*, then handling errors gracefully is vital.

当我们设置东西的时候，另一个关键的基础设施是错误处理。教科书有时会掩盖这一点，因为这更多的是一个实际问题，而不是一个正式的计算机科学问题。但是，如果你关心的是如何制作一个真正可用的语言，那么优雅地处理错误是至关重要的。

> The tools our language provides for dealing with errors make up a large portion of its user interface. When the user's code is working, they aren't thinking about our language at all—their headspace is all about *their program*. It's usually only when things go wrong that they notice our implementation.

我们的语言提供的处理错误的工具构成了其用户界面的很大一部分。当用户的代码在工作时，他们根本不会考虑我们的语言——他们的脑子里都是他们的程序。通常只有当程序出现问题时，他们才会注意到我们的实现。

> When that happens, it's up to us to give the user all the information they need to understand what went wrong and guide them gently back to where they are trying to go. Doing that well means thinking about error handling all through the implementation of our interpreter, starting now.

当这种情况发生时，我们就需要向用户提供他们所需要的所有信息，让他们了解哪里出了问题，并引导他们慢慢达到他们想要去的地方。要做好这一点，意味着从现在开始，在解释器的整个实现过程中都要考虑错误处理^4。

lox/Lox.java，添加到run()方法之后

```
  static void error(int line, String message) {
    report(line, "", message);
  }

  private static void report(int line, String where,
                             String message) {
    System.err.println(
        "[line " + line + "] Error" + where + ": " + message);
    hadError = true;
  }
```

> This `error()` function and its `report()` helper tells the user some syntax error occurred on a given line. That is really the bare minimum to be able to claim you even *have* error reporting. Imagine if you accidentally left a dangling comma in some function call and the interpreter printed out:

这个`error()`函数和其工具方法`report()`会告诉用户在某一行上发生了一些语法错误。这其实是最起码的，可以说你有错误报告功能。想象一下，如果你在某个函数调用中不小心留下了一个悬空的逗号，解释器就会打印出来：

```
Error: Unexpected "," somewhere in your code. Good luck finding it!
```

> That's not very helpful. We need to at least point them to the right line. Even better would be the beginning and end column so they know *where* in the line. Even better than *that* is to *show* the user the offending line, like:

这种信息没有多大帮助。我们至少要给他们指出正确的方向。好一些的做法是指出开头和结尾一栏，这样他们就知道这一行的位置了。更好的做法是向用户显示违规的行，比如：

```
Error: Unexpected "," in argument list.

    15 | function(first, second,);
                               ^-- Here.
```

> I'd love to implement something like that in this book but the honest truth is that it's a lot of grungy string manipulation code. Very useful for users, but not super fun to read in a book and not very technically interesting. So we'll stick with just a line number. In your own interpreters, please do as I say and not as I do.

我很想在这本书里实现这样的东西，但老实说，这会引入很多繁琐的字符串操作代码。这些代码对用户来说非常有用，但在书中读起来并不友好，而且技术上也不是很有趣。所以我们还是只用一个行号。在你们自己的解释器中，请按我说的做，而不是按我做的做。

> The primary reason we're sticking this error reporting function in the main Lox class is because of that `hadError` field. It's defined here:

我们在Lox主类中坚持使用这个错误报告功能的主要原因就是因为那个hadError字段。它的定义在这里：

lox/Lox.java 在Lox类中添加：

```
public class Lox {
  static boolean hadError = false;
```

> We'll use this to ensure we don't try to execute code that has a known error. Also, it lets us exit with a non-zero exit code like a good command line citizen should.

我们将以此来确保我们不会尝试执行有已知错误的代码。此外，它还能让我们像一个好的命令行工具那样，用一个非零的结束代码退出。

lox/Lox.java，在runFile()中添加：

```
    run(new String(bytes, Charset.defaultCharset()));

    // Indicate an error in the exit code.
    if (hadError) System.exit(65);
  }
```

> We need to reset this flag in the interactive loop. If the user makes a mistake, it shouldn't kill their entire session.

我们需要在交互式循环中重置此标志。 如果用户输入有误，也不应终止整个会话。

lox/Lox.java，在runPrompt()中添加：

```
      run(line);
      hadError = false;
    }
```

> The other reason I pulled the error reporting out here instead of stuffing it into the scanner and other phases where the error might occur is to remind you that it's good engineering practice to separate the code that *generates* the errors from the code that *reports* them.

我把错误报告拉出来，而不是把它塞进扫描器和其他可能发生错误的阶段，还有另一个原因，是为了提醒您，把产生错误的代码和报告错误的代码分开是一个很好的工程实践。

> Various phases of the front end will detect errors, but it's not really their job to know how to present that to a user. In a full-featured language implementation, you will likely have multiple ways errors get displayed: on stderr, in an IDE's error window, logged to a file, etc. You don't want that code smeared all over your scanner and parser.

前端的各个阶段都会检测到错误，但是它们不需要知道如何向用户展示错误。在一个功能齐全的语言实现中，可能有多种方式展示错误信息：在stderr，在IDE的错误窗口中，记录到文件，等等。您肯定不希望扫描器和解释器中到处充斥着这类代码。

> Ideally, we would have an actual abstraction, some kind of “ErrorReporter” interface that gets passed to the scanner and parser so that we can swap out different reporting strategies. For our simple interpreter here, I didn’t do that, but I did at least move the code for error reporting into a different class.

理想情况下，我们应该有一个实际的抽象，即传递给扫描程序和解析器的某种ErrorReporter接口^5，这样我们就可以交换不同的报告策略。对于我们这里的简单解释器，我没有那样做，但我至少将错误报告代码移到了一个不同的类中。

> With some rudimentary error handling in place, our application shell is ready. Once we have a Scanner class with a `scanTokens()` method, we can start running it. Before we get to that, let’s get more precise about what tokens are.

有了一些基本的错误处理，我们的应用程序外壳已经准备好了。一旦我们有了一个带有 `scanTokens()` 方法的 Scanner 类，我们就可以开始运行它了。在我们开始之前，让我们更精确地了解什么是标记（tokens）。

> ## 4.2 Lexemes and Tokens

4.2 词素和标记（词法单元）

下面是一行lox代码：

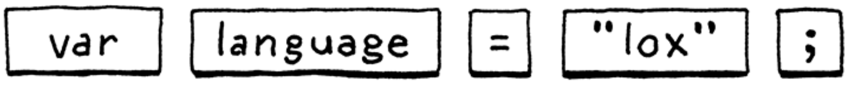
```
var language = "lox";
```

> Here, `var` is the keyword for declaring a variable. That three-character sequence “v-a-r” means something. But if we yank three letters out of the middle of `language`, like “g-u-a”, those don’t mean anything on their own.

在这里，`var`是声明变量的关键字。“v-a-r”这三个字符的序列是有意义的。但如果我们从`language`中间抽出三个字母，比如“g-u-a”，它们本身并没有任何意义。

> That’s what lexical analysis is about. Our job is to scan through the list of characters and group them together into the smallest sequences that still represent something. Each of these blobs of characters is called a **lexeme**. In that example line of code, the lexemes are:

这就是词法分析的意义所在。我们的工作是扫描字符列表，并将它们归纳为具有某些含义的最小序列。每一组字符都被称为词素。在示例代码行中，词素是：



> The lexemes are only the raw substrings of the source code. However, in the process of grouping character sequences into lexemes, we also stumble upon some other useful information. When we take the lexeme and bundle it together with that other data, the result is a token. It includes useful stuff like:

词素只是源代码的原始子字符串。 但是，在将字符序列分组为词素的过程中，我们也会发现了一些其他有用的信息。 当我们获取词素并将其与其他数据捆绑在一起时，结果是一个标记（token，词法单元）。它包含一些有用的内容，比如：

4.2.1 Token type

4.2.1 标记类型

> Keywords are part of the shape of the language's grammar, so the parser often has code like, "If the next token is `while` then do . . . " That means the parser wants to know not just that it has a lexeme for some identifier, but that it has a *reserved* word, and *which* keyword it is.

关键词是语言语法的一部分，所以解析器经常会有这样的代码："如果下一个标记是`while`，那么就……" 。这意味着解析器想知道的不仅仅是它有某个标识符的词素，而是它得到一个*保留词*，以及它是*哪个*关键词。

> The parser could categorize tokens from the raw lexeme by comparing the strings, but that's slow and kind of ugly. Instead, at the point that we recognize a lexeme, we also remember which *kind* of lexeme it represents. We have a different type for each keyword, operator, bit of punctuation, and literal type.

解析器可以通过比较字符串对原始词素中的标记进行分类，但这样做很慢，而且有点难看^6。相反，在我们识别一个词素的时候，我们还要记住它代表的是哪种词素。我们为每个关键字、操作符、标点位和字面量都有不同的类型。

lox/TokenType.java 创建新文件

```java
package com.craftinginterpreters.lox;

enum TokenType {
  // Single-character tokens.
  LEFT_PAREN, RIGHT_PAREN, LEFT_BRACE, RIGHT_BRACE,
  COMMA, DOT, MINUS, PLUS, SEMICOLON, SLASH, STAR,

  // One or two character tokens.
  BANG, BANG_EQUAL,
  EQUAL, EQUAL_EQUAL,
  GREATER, GREATER_EQUAL,
  LESS, LESS_EQUAL,

  // Literals.
  IDENTIFIER, STRING, NUMBER,

  // Keywords.
  AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR,
  PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE,

  EOF
}
```

4.2.2 Literal value

4.2.2 字面量

> There are lexemes for literal values—numbers and strings and the like. Since the scanner has to walk each character in the literal to correctly identify it, it can also convert that textual representation of a

> value to the living runtime object that will be used by the interpreter later.

字面量有对应词素——数字和字符串等。由于扫描器必须遍历文字中的每个字符才能正确识别，所以它还可以将值的文本表示转换为运行时对象，解释器后续将使用该对象。

> 4.2.3 Location information

4.2.3 位置信息

> Back when I was preaching the gospel about error handling, we saw that we need to tell users *where* errors occurred. Tracking that starts here. In our simple interpreter, we note only which line the token appears on, but more sophisticated implementations include the column and length too.

早在我宣讲错误处理的福音时，我们就看到，我们需要告诉用户错误发生在哪里。（用户）从这里开始定位问题。在我们的简易解释器中，我们只说明了标记出现在哪一行上，但更复杂的实现中还应该包括列位置和长度^7。

> We take all of this data and wrap it in a class.

我们将所有这些数据打包到一个类中。

lox/Token.java，创建新文件

```
package com.craftinginterpreters.lox;

class Token {
  final TokenType type;
  final String lexeme;
  final Object literal;
  final int line;

  Token(TokenType type, String lexeme, Object literal, int line) {
    this.type = type;
    this.lexeme = lexeme;
    this.literal = literal;
    this.line = line;
  }

  public String toString() {
    return type + " " + lexeme + " " + literal;
  }
}
```

> Now we have an object with enough structure to be useful for all of the later phases of the interpreter.

现在我们有了一个信息充分的对象，足以支撑解释器的所有后期阶段。

> ## 4.3 Regular Languages and Expressions

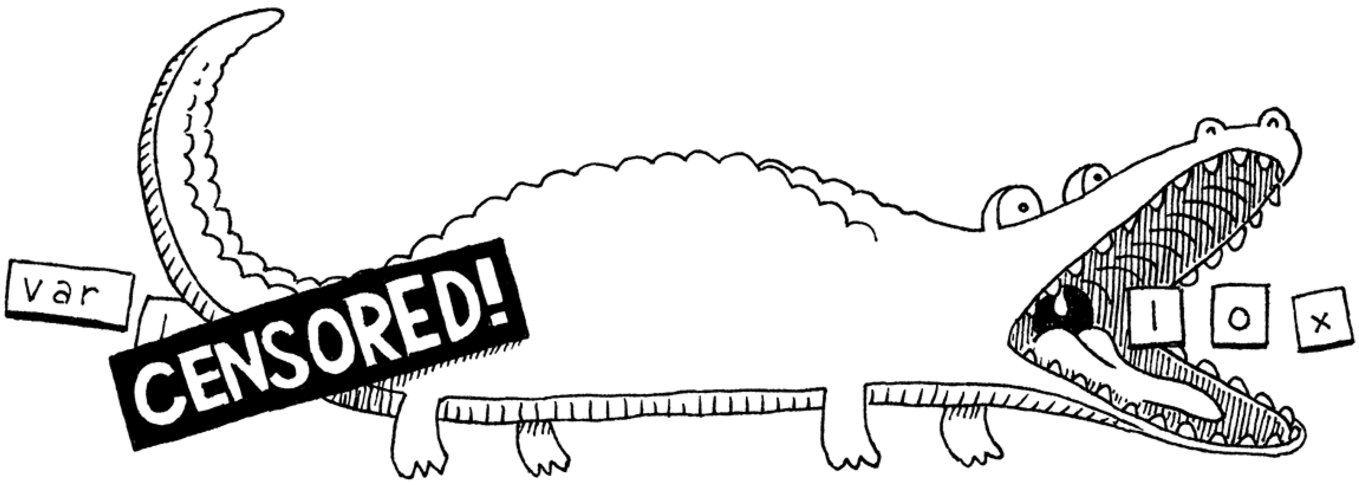
4.3 正则语言和表达式

> Now that we know what we're trying to produce, let's, well, produce it. The core of the scanner is a loop. Starting at the first character of the source code, it figures out what lexeme it belongs to, and consumes it and any following characters that are part of that lexeme. When it reaches the end of that lexeme, it emits a token.

既然我们已知道我们要输出什么，那么，我们就开始吧。扫描器的核心是一个循环。从源码的第一个字符开始，扫描器计算出该字符属于哪个词素，并消费它和属于该词素的任何后续字符。当到达该词素的末尾时，扫描器会输出一个标记（词法单元 token）。

> Then it loops back and does it again, starting from the very next character in the source code. It keeps doing that, eating characters and occasionally, uh, excreting tokens, until it reaches the end of the input.

然后再循环一次，它又循环回来，从源代码中的下一个字符开始再做一次。它一直这样做，吃掉字符，偶尔，呃，排出标记，直到它到达输入的终点。



> The part of the loop where we look at a handful of characters to figure out which kind of lexeme it "matches" may sound familiar. If you know regular expressions, you might consider defining a regex for each kind of lexeme and using those to match characters. For example, Lox has the same rules as C for identifiers (variable names and the like). This regex matches one:

在循环中，我们会查看一些字符，以确定它 "匹配 "的是哪种词素，这部分内容可能听起来很熟悉，但如果你知道正则表达式，你可以考虑为每一种词素定义一个regex，并使用这些regex来匹配字符。例如，Lox对标识符（变量名等）的规则与C语言相同。下面的regex可以匹配一个标识符：

```
[a-zA-Z_][a-zA-Z_0-9]*
```

> If you did think of regular expressions, your intuition is a deep one. The rules that determine how a particular language groups characters into lexemes are called its **lexical grammar**. In Lox, as in most programming languages, the rules of that grammar are simple enough for the language to be classified a **regular language**. That's the same "regular" as in regular expressions.

如果你确实想到了正则表达式，那么你的直觉还是很深刻的。决定一门语言如何将字符分组为词素的规则被称为它的**词法语法**^8。在Lox中，和大多数编程语言一样，该语法的规则非常简单，可以将其归为**正则语言**。这里的正则和正则表达式中的 "正则 "是一样的含义。

> You very precisely *can* recognize all of the different lexemes for Lox using regexes if you want to, and there's a pile of interesting theory underlying why that is and what it means. Tools like Lex or Flex are designed expressly to let you do this—throw a handful of regexes at them, and they give you a complete scanner back.

如果你愿意，你可以非常精确地使用正则表达式来识别Lox的所有不同词组，而且还有一堆有趣的理论来支撑着为什么会这样以及它的意义。像Lex^9或Flex这样的工具就是专门为实现这一功能而设计的——向其中传入一些正则表达式，它可以为您提供完整的扫描器。

> Since our goal is to understand how a scanner does what it does, we won't be delegating that task. We're about handcrafted goods.

由于我们的目标是了解扫描器是如何工作的，所以我们不会把这个任务交给正则表达式。我们要亲自动手实现。

> ## 4.4 The Scanner Class

4.4 Scanner类

事不宜迟，我们先来建一个扫描器吧。

lox/Scanner.java，创建新文件^10

```java
package com.craftinginterpreters.lox;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static com.craftinginterpreters.lox.TokenType.*;

class Scanner {
  private final String source;
  private final List<Token> tokens = new ArrayList<>();

  Scanner(String source) {
    this.source = source;
  }
}
```

> We store the raw source code as a simple string, and we have a list ready to fill with tokens we're going to generate. The aforementioned loop that does that looks like this:

我们将原始的源代码存储为一个简单的字符串，并且我们已经准备了一个列表来保存扫描时产生的标记。前面提到的循环看起来类似于：

lox/Scanner.java，方法Scanner()后添加：

```
  List<Token> scanTokens() {
    while (!isAtEnd()) {
      // We are at the beginning of the next lexeme.
      start = current;
      scanToken();
    }

    tokens.add(new Token(EOF, "", null, line));
    return tokens;
  }
```

> The scanner works its way through the source code, adding tokens until it runs out of characters. Then it appends one final “end of file” token. That isn’t strictly needed, but it makes our parser a little cleaner.

扫描器通过自己的方式遍历源代码，添加标记，直到遍历完所有字符。然后，它在最后附加一个的 "end of file "标记。严格意义上来说，这并不是必须的，但它可以使我们的解析器更加干净。

> This loop depends on a couple of fields to keep track of where the scanner is in the source code.

这个循环依赖于几个字段来跟踪扫描器在源代码中的位置。

lox/Scanner.java，在Scanner类中添加：

```
  private final List<Token> tokens = new ArrayList<>();
// 添加下面三行代码
  private int start = 0;
  private int current = 0;
  private int line = 1;

  Scanner(String source) {
```

> The `start` and `current` fields are offsets that index into the string. The `start` field points to the first character in the lexeme being scanned, and `current` points at the character currently being considered. The `line` field tracks what source line `current` is on so we can produce tokens that know their location.

`start`和`current`字段是指向字符串的偏移量。`start`字段指向被扫描的词素中的第一个字符，`current`字段指向当前正在处理的字符。`line`字段跟踪的是`current`所在的源文件行数，这样我们产生的标记就可以知道其位置。

> Then we have one little helper function that tells us if we’ve consumed all the characters.

然后，我们还有一个辅助函数，用来告诉我们是否已消费完所有字符。

lox/Scanner.java 在scanTokens()方法之后添加：

```
  private boolean isAtEnd() {
    return current >= source.length();
  }
```

4.5 Recognizing Lexemes

4.5 识别词素

> In each turn of the loop, we scan a single token. This is the real heart of the scanner. We'll start simple. Imagine if every lexeme were only a single character long. All you would need to do is consume the next character and pick a token type for it. Several lexemes *are* only a single character in Lox, so let's start with those.

在每一次循环中，我们可以扫描出一个 token。这是扫描器真正的核心。让我们先从简单情况开始。想象一下，如果每个词素只有一个字符长。您所需要做的就是消费下一个字符并为其选择一个 token 类型。在Lox中有一些词素只包含一个字符，所以我们从这些词素开始^11。

lox/Scanner.java添加到scanTokens()方法之后

```
private void scanToken() {
    char c = advance();
    switch (c) {
      case '(': addToken(LEFT_PAREN); break;
      case ')': addToken(RIGHT_PAREN); break;
      case '{': addToken(LEFT_BRACE); break;
      case '}': addToken(RIGHT_BRACE); break;
      case ',': addToken(COMMA); break;
      case '.': addToken(DOT); break;
      case '-': addToken(MINUS); break;
      case '+': addToken(PLUS); break;
      case ';': addToken(SEMICOLON); break;
      case '*': addToken(STAR); break;
    }
  }
```

> Again, we need a couple of helper methods.

同样，我们也需要一些辅助方法。

lox/Scanner.java，添加到 isAtEnd()方法后

```
  private char advance() {
    current++;
    return source.charAt(current - 1);
  }

  private void addToken(TokenType type) {
    addToken(type, null);
  }

  private void addToken(TokenType type, Object literal) {
    String text = source.substring(start, current);
    tokens.add(new Token(type, text, literal, line));
  }
```

> The `advance()` method consumes the next character in the source file and returns it. Where `advance()` is for input, `addToken()` is for output. It grabs the text of the current lexeme and creates a new token for it. We'll use the other overload to handle tokens with literal values soon.

`advance()`方法获取源文件中的下一个字符并返回它。`advance()`用于处理输入，`addToken()`则用于输出。该方法获取当前词素的文本并为其创建一个新 token。我们马上会使用另一个重载方法来处理带有字面值的token。

> ## 4.5.1 Lexical errors

4.5.1 词法错误

> Before we get too far in, let's take a moment to think about errors at the lexical level. What happens if a user throws a source file containing some characters Lox doesn't use, like `@#^`, at our interpreter? Right now, those characters get silently discarded. They aren't used by the Lox language, but that doesn't mean the interpreter can pretend they aren't there. Instead, we report an error.

在我们深入探讨之前，我们先花一点时间考虑一下词法层面的错误。如果用户抛入解释器的源文件中包含一些Lox中不使用的字符——如`@#^`，会发生什么？现在，这些字符被默默抛弃了。它们没有被Lox语言使用，但是不意味着解释器可以假装它们不存在。相反，我们应该报告一个错误：

lox/Scanner.java 在 scanToken()方法中添加：

```
      case '*': addToken(STAR); break;

      default:
        Lox.error(line, "Unexpected character.");
        break;
    }
```

> Note that the erroneous character is still *consumed* by the earlier call to `advance()`. That's important so that we don't get stuck in an infinite loop.

注意，错误的字符仍然会被前面调用的`advance()`方法消费。这一点很重要，这样我们就不会陷入无限循环了。

> Note also that we *keep scanning*. There may be other errors later in the program. It gives our users a better experience if we detect as many of those as possible in one go. Otherwise, they see one tiny error and fix it, only to have the next error appear, and so on. Syntax error Whac-A-Mole is no fun.

另请注意，我们一直在扫描。 程序稍后可能还会出现其他错误。 如果我们能够一次检测出尽可能多的错误，将为我们的用户带来更好的体验。 否则，他们会看到一个小错误并修复它，但是却出现下一个错误，不断重复这个过程。语法错误“打地鼠”一点也不好玩。

> (Don't worry. Since `hadError` gets set, we'll never try to *execute* any of the code, even though we keep going and scan the rest of it.)

(别担心。因为`hadError`进行了赋值，我们永远不会尝试执行任何代码，即使程序在继续运行并扫描代码文件的其余部分。)

> 4.5.2 Operators

4.5.2 操作符

> We have single-character lexemes working, but that doesn't cover all of Lox's operators. What about `!`? It's a single character, right? Sometimes, yes, but if the very next character is an equals sign, then we should instead create a `!=` lexeme. Note that the `!` and `=` are *not* two independent operators. You can't write `!  =` in Lox and have it behave like an inequality operator. That's why we need to scan `!=` as a single lexeme. Likewise, `<`, `>`, and `=` can all be followed by `=` to create the other equality and comparison operators.

我们的单字符词素已经生效了，但是这不能涵盖Lox中的所有操作符。比如`!`，这是单字符，对吧？有时候是的，但是如果下一个字符是等号，那么我们应该改用`!=` 词素。注意，这里的`!`和`=`*不是*两个独立的操作符。在Lox中，你不能写`!  =`来表示不等操作符。这就是为什么我们需要将`!=`作为单个词素进行扫描。同样地，`<`、`>`和`=`都可以与后面跟随的`=`来组合成其他相等和比较操作符。

> For all of these, we need to look at the second character.

对于所有这些情况，我们都需要查看第二个字符。

lox/Scanner.java，在 scanToken()方法中添加

```
      case '*': addToken(STAR); break;
      case '!':
        addToken(match('=') ? BANG_EQUAL : BANG);
        break;
      case '=':
        addToken(match('=') ? EQUAL_EQUAL : EQUAL);
        break;
      case '<':
        addToken(match('=') ? LESS_EQUAL : LESS);
        break;
      case '>':
        addToken(match('=') ? GREATER_EQUAL : GREATER);
        break;
      default:
```

> Those cases use this new method:

这些分支中使用了下面的新方法：

lox/Scanner.java 添加到 scanToken()方法后

```
  private boolean match(char expected) {
    if (isAtEnd()) return false;
    if (source.charAt(current) != expected) return false;

    current++;
    return true;
  }
```

> It's like a conditional `advance()`. We only consume the current character if it's what we're looking for.

这就像一个有条件的`advance()`。只有当前字符是我们正在寻找的字符时，我们才会消费。

> Using `match()`, we recognize these lexemes in two stages. When we reach, for example, `!`, we jump to its switch case. That means we know the lexeme *starts* with `!`. Then we look at the next character to determine if we're on a `!=` or merely a `!`.

使用`match()`，我们分两个阶段识别这些词素。例如，当我们得到`!`时，我们会跳转到它的case分支。这意味着我们知道这个词素是以 `!`开始的。然后，我们查看下一个字符，以确认词素是一个 `!=` 还是仅仅是一个 `!`。

> ## 4.6 Longer Lexemes

4.6 更长的词素

> We're still missing one operator: `/` for division. That character needs a little special handling because comments begin with a slash too.

我们还缺少一个操作符：表示除法的`/`。这个字符需要一些特殊处理，因为注释也是以斜线开头的。

lox/Scanner.java，在scanToken()方法中添加：

```
      break;
      case '/':
        if (match('/')) {
          // A comment goes until the end of the line.
          while (peek() != '\n' && !isAtEnd()) advance();
        } else {
          addToken(SLASH);
        }
        break;
      default:
```

> This is similar to the other two-character operators, except that when we find a second `/`, we don't end the token yet. Instead, we keep consuming characters until we reach the end of the line.

这与其它的双字符操作符是类似的，区别在于我们找到第二个`/`时，还没有结束本次标记。相反，我们会继续消费字符直至行尾。

> This is our general strategy for handling longer lexemes. After we detect the beginning of one, we shunt over to some lexeme-specific code that keeps eating characters until it sees the end.

这是我们处理较长词素的一般策略。当我们检测到一个词素的开头后，我们会分流到一些特定于该词素的代码，这些代码会不断地消费字符，直到结尾。

> We've got another helper:

我们又有了一个辅助函数：

lox/Scanner.java，在match()方法后添加：

```
  private char peek() {
    if (isAtEnd()) return '\0';
    return source.charAt(current);
  }
```

> It's sort of like `advance()`, but doesn't consume the character. This is called **lookahead**. Since it only looks at the current unconsumed character, we have *one character of lookahead*. The smaller this number is, generally, the faster the scanner runs. The rules of the lexical grammar dictate how much lookahead we need. Fortunately, most languages in wide use peek only one or two characters ahead.

这有点像`advance()`方法，只是不会消费字符。这就是所谓的**lookahead(前瞻)**^12。因为它只关注当前未消费的字符，所以我们有*一个前瞻字符*。一般来说，前瞻的字符越少，扫描器运行速度就越快。词法语法的规则决定了我们需要前瞻多少字符。幸运的是，大多数广泛使用的语言只需要提前一到两个字符。

> Comments are lexemes, but they aren't meaningful, and the parser doesn't want to deal with them. So when we reach the end of the comment, we *don't* call `addToken()`. When we loop back around to start the next lexeme, `start` gets reset and the comment's lexeme disappears in a puff of smoke.

注释是词素，但是它们没有含义，而且解析器也不想要处理它们。所以，我们达到注释末尾后，*不会*调用`addToken()`方法。当我们循环处理下一个词素时，`start`已经被重置了，注释的词素就消失在一阵烟雾中了。

> While we're at it, now's a good time to skip over those other meaningless characters: newlines and whitespace.

既然如此，现在正好可以跳过其它那些无意义的字符了：换行和空格。

lox/Scanner.java，在scanToken()方法中添加：

```
          break;
        case ' ':
        case '\r':
        case '\t':
          // Ignore whitespace.
          break;

        case '\n':
          line++;
          break;
        default:
          Lox.error(line, "Unexpected character.");
```

> When encountering whitespace, we simply go back to the beginning of the scan loop. That starts a new lexeme *after* the whitespace character. For newlines, we do the same thing, but we also increment the line counter. (This is why we used `peek()` to find the newline ending a comment instead of `match()`. We want that newline to get us here so we can update `line`.)

当遇到空白字符时，我们只需回到扫描循环的开头。这样就会在空白字符之后开始一个新的词素。对于换行符，我们做同样的事情，但我们也会递增行计数器。(这就是为什么我们使用`peek()` 而不是`match()`来查找注释结尾的换行符。我们到这里希望能读取到换行符，这样我们就可以更新行数了)

> Our scanner is getting smarter. It can handle fairly free-form code like:

我们的扫描器越来越聪明了。它可以处理相当自由形式的代码，如：

```
// this is a comment
(( )){} // grouping stuff
!*+-/=<> <= == // operators
```

> ### 4.6.1 String literals

4.6.1 字符串字面量

> Now that we're comfortable with longer lexemes, we're ready to tackle literals. We'll do strings first, since they always begin with a specific character, `"`.

现在我们对长词素已经很熟悉了，我们可以开始处理字面量了。我们先处理字符串，因为字符串总是以一个特定的字符`"`开头。

lox/Scanner.java，在 scanToken()方法中添加：

```
          break;
        case '"': string(); break;
        default:
```

> That calls:

这里会调用：

lox/Scanner.java，在 *scanToken*()方法之后添加：

```
    private void string() {
      while (peek() != '"' && !isAtEnd()) {
        if (peek() == '\n') line++;
        advance();
      }

      if (isAtEnd()) {
        Lox.error(line, "Unterminated string.");
        return;
      }

      // The closing ".
      advance();

      // Trim the surrounding quotes.
      String value = source.substring(start + 1, current - 1);
      addToken(STRING, value);
    }
```

> Like with comments, we consume characters until we hit the `"` that ends the string. We also gracefully handle running out of input before the string is closed and report an error for that.

与注释类似，我们会一直消费字符，直到`"`结束该字符串。如果输入内容耗尽，我们也会进行优雅的处理，并报告一个对应的错误。

> For no particular reason, Lox supports multi-line strings. There are pros and cons to that, but prohibiting them was a little more complex than allowing them, so I left them in. That does mean we also need to update `line` when we hit a newline inside a string.

没有特别的原因，Lox支持多行字符串。这有利有弊，但禁止换行比允许换行更复杂一些，所以我把它们保留了下来。这意味着当我们在字符串内遇到新行时，我们也需要更新`line`值。

> Finally, the last interesting bit is that when we create the token, we also produce the actual string *value* that will be used later by the interpreter. Here, that conversion only requires a `substring()` to strip off the surrounding quotes. If Lox supported escape sequences like `\n`, we'd unescape those here.

最后，还有一个有趣的地方就是当我们创建标记时，我们也会产生实际的字符串值，该值稍后将被解释器使用。这里，值的转换只需要调用`substring()`剥离前后的引号。如果Lox支持转义序列，比如`\n`，我们会在这里取消转义。

> ### 4.6.2 Number literals

4.6.2 数字字面量

> All numbers in Lox are floating point at runtime, but both integer and decimal literals are supported. A number literal is a series of digits optionally followed by a `.` and one or more trailing digits.

在Lox中，所有的数字在运行时都是浮点数，但是同时支持整数和小数字面量。一个数字字面量就是一系列数位，后面可以跟一个`.`和一或多个尾数^13。

```
1234
12.34
```

我们不允许小数点处于最开始或最末尾，所以下面的格式是不正确的：

```
.1234
1234.
```

> We could easily support the former, but I left it out to keep things simple. The latter gets weird if we ever want to allow methods on numbers like `123.sqrt()`.

我们可以很容易地支持前者，但为了保持简单，我把它删掉了。如果我们要允许对数字进行方法调用，比如`123.sqrt()`，后者会变得很奇怪。

> To recognize the beginning of a number lexeme, we look for any digit. It's kind of tedious to add cases for every decimal digit, so we'll stuff it in the default case instead.

为了识别数字词素的开头，我们会寻找任何一位数字。 为每个十进制数字添加case分支有点乏味，所以我们直接在默认分支中进行处理。

lox/Scanner.java，在 scanToken()方法中替换一行：

```
      default:
        // 替换部分开始
        if (isDigit(c)) {
          number();
        } else {
          Lox.error(line, "Unexpected character.");
        }
        // 替换部分结束
        break;
```

> This relies on this little utility:

这里依赖下面的小工具函数^14：

lox/Scanner.java，在 peek()方法之后添加：

```
  private boolean isDigit(char c) {
    return c >= '0' && c <= '9';
  }
```

> Once we know we are in a number, we branch to a separate method to consume the rest of the literal, like we do with strings.

一旦我们知道当前在处理数字，我们就分支进入一个单独的方法消费剩余的字面量，跟字符串的处理类似。

lox/Scanner.java，在 scanToken()方法后添加：

```
  private void number() {
    while (isDigit(peek())) advance();

    // Look for a fractional part.
    if (peek() == '.' && isDigit(peekNext())) {
      // Consume the "."
      advance();

      while (isDigit(peek())) advance();
    }

    addToken(NUMBER,
        Double.parseDouble(source.substring(start, current)));
  }
```

> We consume as many digits as we find for the integer part of the literal. Then we look for a fractional part, which is a decimal point (`.`) followed by at least one digit. If we do have a fractional part, again, we consume as many digits as we can find.

我们在字面量的整数部分中尽可能多地获取数字。然后我们寻找小数部分，也就是一个小数点(`.`)后面至少跟一个数字。如果确实有小数部分，同样地，我们也尽可能多地获取数字。

> Looking past the decimal point requires a second character of lookahead since we don’t want to consume the `.` until we’re sure there is a digit *after* it. So we add:

在定位到小数点之后需要继续前瞻第二个字符，因为我们只有确认其*后*有数字才会消费`.`。所以我们添加了^15：

lox/Scanner.java，在 peek()方法后添加

```
  private char peekNext() {
    if (current + 1 >= source.length()) return '\0';
    return source.charAt(current + 1);
  }
```

> Finally, we convert the lexeme to its numeric value. Our interpreter uses Java’s `Double` type to represent numbers, so we produce a value of that type. We’re using Java’s own parsing method to convert the lexeme to a real Java double. We could implement that ourselves, but, honestly, unless you’re trying to cram for an upcoming programming interview, it’s not worth your time.

最后，我们将词素转换为其对应的数值。我们的解释器使用Java的`Double`类型来表示数字，所以我们创建一个该类型的值。我们使用Java自带的解析方法将词素转换为真正的Java double。我们可以自己实现，但是，说实话，除非你想为即将到来的编程面试做准备，否则不值得你花时间。

> The remaining literals are Booleans and `nil`, but we handle those as keywords, which gets us to . . .

剩下的词素是Boolean和`nil`，但我们把它们作为关键字来处理，这样我们就来到了......

4.7 Reserved Words and Identifiers

4.7 保留字和标识符

> Our scanner is almost done. The only remaining pieces of the lexical grammar to implement are identifiers and their close cousins, the reserved words. You might think we could match keywords like `or` in the same way we handle multiple-character operators like `<=`.

我们的扫描器基本完成了，词法语法中还需要实现的部分仅剩标识符及其近亲——保留字。你也许会想，我们可以采用与处理`<=`等多字符操作符时相同的方法来匹配关键字，如`or`。

```
case 'o':
  if (peek() == 'r') {
    addToken(OR);
  }
  break;
```

> Consider what would happen if a user named a variable `orchid`. The scanner would see the first two letters, `or`, and immediately emit an `or` keyword token. This gets us to an important principle called **maximal munch**. When two lexical grammar rules can both match a chunk of code that the scanner is looking at, *whichever one matches the most characters wins*.

考虑一下，如果用户将变量命名为`orchid`会发生什么？扫描器会先看到前面的两个字符，然后立刻生成一个`or`标记。这就涉及到了一个重要原则，叫作**maximal munch**(最长匹配)^16。当两个语法规则都能匹配扫描器正在处理的一大块代码时，*哪个规则相匹配的字符最多，就使用哪个规则*。

> That rule states that if we can match `orchid` as an identifier and `or` as a keyword, then the former wins. This is also why we tacitly assumed, previously, that `<=` should be scanned as a single `<=` token and not `<` followed by `=`.

该规则规定，如果我们可以将`orchid`匹配为一个标识符，也可以将`or`匹配为一个关键字，那就采用第一种结果。这也就是为什么我们在前面会默认为，`<=`应该识别为单一的`<=`标记，而不是`<`后面跟了一个`=`。

> Maximal munch means we can't easily detect a reserved word until we've reached the end of what might instead be an identifier. After all, a reserved word *is* an identifier, it's just one that has been claimed by the language for its own use. That's where the term **reserved word** comes from.

最大匹配原则意味着，我们只有扫描完一个可能是标识符的片段，才能确认是否一个保留字。毕竟，保留字也是一个标识符，只是一个已经被语言要求为自己所用的标识符。这也是**保留字**一词的由来。

> So we begin by assuming any lexeme starting with a letter or underscore is an identifier.

所以我们首先假设任何以字母或下划线开头的词素都是一个标识符。

lox/Scanner.java，在 scanToken()中添加代码

```
        default:
        if (isDigit(c)) {
          number();
          // 新增部分开始
        } else if (isAlpha(c)) {
          identifier();
        // 新增部分结束
        } else {
          Lox.error(line, "Unexpected character.");
        }
```

> The rest of the code lives over here:

其它代码如下：

lox/Scanner.java，在 scanToken()方法之后添加：

```
  private void identifier() {
    while (isAlphaNumeric(peek())) advance();

    addToken(IDENTIFIER);
  }
```

> We define that in terms of these helpers:

通过以下辅助函数来定义：

lox/Scanner.java，在 peekNext()方法之后添加：

```
  private boolean isAlpha(char c) {
    return (c >= 'a' && c <= 'z') ||
           (c >= 'A' && c <= 'Z') ||
            c == '_';
  }

  private boolean isAlphaNumeric(char c) {
    return isAlpha(c) || isDigit(c);
  }
```

> That gets identifiers working. To handle keywords, we see if the identifier’s lexeme is one of the reserved words. If so, we use a token type specific to that keyword. We define the set of reserved words in a map.

这样标识符就开始工作了。为了处理关键字，我们要查看标识符的词素是否是保留字之一。如果是，我们就使用该关键字特有的标记类型。我们在map中定义保留字的集合。

lox/Scanner.java，在 Scanner类中添加：

```
  private static final Map<String, TokenType> keywords;

  static {
    keywords = new HashMap<>();
    keywords.put("and",    AND);
    keywords.put("class",  CLASS);
    keywords.put("else",   ELSE);
    keywords.put("false",  FALSE);
    keywords.put("for",    FOR);
    keywords.put("fun",    FUN);
    keywords.put("if",     IF);
    keywords.put("nil",    NIL);
    keywords.put("or",     OR);
    keywords.put("print",  PRINT);
    keywords.put("return", RETURN);
    keywords.put("super",  SUPER);
    keywords.put("this",   THIS);
    keywords.put("true",   TRUE);
    keywords.put("var",    VAR);
    keywords.put("while",  WHILE);
  }
```

> Then, after we scan an identifier, we check to see if it matches anything in the map.

接下来，在我们扫描到标识符之后，要检查是否与map中的某些项匹配。

lox/Scanner.java，在 identifier()方法中替换一行：

```

    while (isAlphaNumeric(peek())) advance();

    // 替换部分开始
    String text = source.substring(start, current);
    TokenType type = keywords.get(text);
    if (type == null) type = IDENTIFIER;
    addToken(type);
    // 替换部分结束
  }
```

> If so, we use that keyword's token type. Otherwise, it's a regular user-defined identifier.

如果匹配的话，就使用关键字的标记类型。否则，就是一个普通的用户定义的标识符。

> And with that, we now have a complete scanner for the entire Lox lexical grammar. Fire up the REPL and type in some valid and invalid code. Does it produce the tokens you expect? Try to come up with some interesting edge cases and see if it handles them as it should.

至此，我们就有了一个完整的扫描器，可以扫描整个Lox词法语法。启动REPL，输入一些有效和无效的代码。它是否产生了你所期望的词法单元？试着想出一些有趣的边界情况，看看它是否能正确地处理它们。

^1: 一直以来，这项工作被称为 "扫描(scanning) "和 "词法分析(lexing)" （"词法分析(lexical analysis)"的简称）。早在计算机还像Winnebagos一样大，但内存比你的手表还小的时候，有些人就用 "扫描 "来指代从磁盘上读取原始源代码字符并在内存中缓冲的那段代码。然后，"lexing "是后续阶段，对字符做有用的操作。现在，将源文件读入内存是很平常的事情，因此在编译器中很少出现不同的阶段。 因此，这两个术语基本上可以互换。 ^2: `System.exit(64)`，对于退出代码，我使用UNIX sysexts .h头文件中定义的约定。这是我能找到的

最接近标准的东西。 ^3: 交互式提示符也被称为REPL(发音像rebel，但替换为p)。它的名称来自于Lisp，实现Lisp非常简单，只需围绕几个内置函数进行循环:`(print (eval (read)))`从嵌套最内的调用向外执行，读取一行输入，求值，打印结果，然后循环并再次执行。 ^4: 说了这么多，对于这个解释器，我们要构建的只是基本框架。我很想谈谈交互式调试器、静态分析器和其它有趣的东西，但是篇幅实在有限。 ^5: 我第一次实现jlox的时候正是如此。最后我把它拆出去了，因为对于本书的最小解释器来说，这有点过度设计了。 ^6: 毕竟，字符串比较最终也会比对单个字符，这不正是扫描器的工作吗？ ^7: 一些标记实现将位置存储为两个数字：从源文件开始到词素开始的偏移量，以及词素的长度。扫描器无论如何都会知道这些数字，因此计算这些数字没有任何开销。通过回头查看源文件并计算前面的换行数，可以将偏移量转换为行和列位置。这听起来很慢，确实如此。然而，只有当你需要向用户实际显示行和列的时候，你才需要这样做。大多数标记从来不会出现在错误信息中。对于这些标记，你花在提前计算位置信息上的时间越少越好。 ^8: 我很痛心要对理论做这么多掩饰，尤其是当它像乔姆斯基谱系和有限状态机那样有趣的时候。但说实话，其他的书比我写得好。*Compilers: Principles, Techniques, and Tools*(常被称为“龙书”)是最经典的参考书。 ^9: Lex是由Mike Lesk和Eric Schmidt创建的。是的，就是那个曾任谷歌执行董事长的Eric Schmidt。我并不是说编程语言是通往财富和名声的必经之路，但我们中至少已经有一位超级亿万富翁。 ^10: 我知道很多人认为静态导入是一种不好的代码风格，但这样我就不必在扫描器和解析器中到处写`TokenType`了。恕我直言，在一本书中，每个字符都很重要 ^11: 想知道这里为什么没有`/`吗？别担心，我们会解决的。 ^12: 技术上来说，`match()`方法也是在做前瞻。`advance()`和`peek()`是基本运算符，`match()`将它们结合起来。 ^13: 因为我们只会根据数字来判断数字字面量，这就意味着`-123`不是一个数字*字面量*。相反，`-123`是一个*表达式*，将`-`应用到数字字面量`123`。在实践中，结果是一样的，尽管它有一个有趣的边缘情况。试想一下，如果我们要在数字上添加方法调用：`print -123.abs();`，这里会输出`-123`，因为负号的优先级低于方法调用。我们可以通过将`-`作为数字字面值的一部分来解决这个问题。但接着考虑：`var n = 123; print -n.abs();`，结果仍然是`-123`，所以现在语言似乎不一致。无论你怎么做，有些情况最后都会变得很奇怪。 ^14: Java标准库中提供了Character.isDigit()，这似乎是个不错的选择。唉，该方法中还允许梵文数字、全宽数字和其他我们不想要的有趣的东西。 ^15: 我本可以让`peek()`方法接受一个参数来表示要前瞻的字符数，而不需要定义两个函数。但这样做就会允许前瞻任意长度的字符。提供两个函数可以让读者更清楚地知道，我们的扫描器最多只能向前看两个字符。 ^16: 看一下这段讨厌的C代码：`---a;`，它有效吗？这取决于扫描器如何分割词素。如果扫描器看到的是`- --a;`，那它就可以被解析。但是这需要扫描器知道代码前后的语法结构，这比我们需要的更复杂。相反，最大匹配原则表明，扫描结果总是：`-- -a;`，它就会这样扫描，尽管这样做会在解析器中导致后面的语法错误。

CHALLENGES

习题

> 1、The lexical grammars of Python and Haskell are not *regular*. What does that mean, and why aren't they?

1、Python和Haskell的语法不是*常规的*。这是什么意思，为什么不是呢？

- Python和Haskell都采用了对缩进敏感的语法，所以它们必须将缩进级别的变动识别为词法标记。这样做需要比较连续行的开头空格数量，这是使用常规语法无法做到的。

> 2、Aside from separating tokens—distinguishing `print foo` from `printfoo`—spaces aren't used for much in most languages. However, in a couple of dark corners, a space *does* affect how code is parsed in CoffeeScript, Ruby, and the C preprocessor. Where and what effect does it have in each of those languages?

2、除了分隔标记——区分`print foo`和`printfoo`——空格在大多数语言中并没有什么用处。在CoffeeScript、Ruby和C预处理器中的一些隐秘的地方，空格确实会影响代码解析方式。在这些语言中，空格在什么地方，会

有什么影响？

> 3、Our scanner here, like most, discards comments and whitespace since those aren't needed by the parser. Why might you want to write a scanner that does *not* discard those? What would it be useful for?

3、我们这里的扫描器和大多数扫描器一样，会丢弃注释和空格，因为解析器不需要这些。什么情况下你会写一个不丢弃这些的扫描器？它有什么用呢？

> 4、Add support to Lox's scanner for C-style `/* ... */` block comments. Make sure to handle newlines in them. Consider allowing them to nest. Is adding support for nesting more work than you expected? Why?

4、为Lox扫描器增加对C样式`/ * ... * /`屏蔽注释的支持。确保要处理其中的换行符。 考虑允许它们嵌套，增加对嵌套的支持是否比你预期的工作更多？ 为什么？

> ## DESIGN NOTE: IMPLICIT SEMICOLONS

设计笔记：隐藏的分号

> Programmers today are spoiled for choice in languages and have gotten picky about syntax. They want their language to look clean and modern. One bit of syntactic lichen that almost every new language scrapes off (and some ancient ones like BASIC never had) is `;` as an explicit statement terminator.
>
> Instead, they treat a newline as a statement terminator where it makes sense to do so. The "where it makes sense" part is the challenging bit. While *most* statements are on their own line, sometimes you need to spread a single statement across a couple of lines. Those intermingled newlines should not be treated as terminators.
>
> Most of the obvious cases where the newline should be ignored are easy to detect, but there are a handful of nasty ones:
>
> - A return value on the next line:
>
>   ```
>   if (condition) return
>   "value"
>   ```
>
>   Is "value" the value being returned, or do we have a `return` statement with no value followed by an expression statement containing a string literal?
>
> - A parenthesized expression on the next line:
>
>   ```
>   func
>   (parenthesized)
>   ```
>
>   Is this a call to `func(parenthesized)`, or two expression statements, one for `func` and one for a parenthesized expression?

- A `-` on the next line:

  ```
  first
  -second
  ```

  Is this `first - second`—an infix subtraction—or two expression statements, one for `first` and one to negate `second`?

In all of these, either treating the newline as a separator or not would both produce valid code, but possibly not the code the user wants. Across languages, there is an unsettling variety of rules used to decide which newlines are separators. Here are a couple:

- Lua completely ignores newlines, but carefully controls its grammar such that no separator between statements is needed at all in most cases. This is perfectly legit:

  ```
  a = 1 b = 2
  ```

  Lua avoids the `return` problem by requiring a `return` statement to be the very last statement in a block. If there is a value after `return` before the keyword `end`, it *must* be for the `return`. For the other two cases, they allow an explicit `;` and expect users to use that. In practice, that almost never happens because there's no point in a parenthesized or unary negation expression statement.

- Go handles newlines in the scanner. If a newline appears following one of a handful of token types that are known to potentially end a statement, the newline is treated like a semicolon, otherwise it is ignored. The Go team provides a canonical code formatter, gofmt, and the ecosystem is fervent about its use, which ensures that idiomatic styled code works well with this simple rule.

- Python treats all newlines as significant unless an explicit backslash is used at the end of a line to continue it to the next line. However, newlines anywhere inside a pair of brackets (`()`, `[]`, or `{}`) are ignored. Idiomatic style strongly prefers the latter.

  This rule works well for Python because it is a highly statement-oriented language. In particular, Python's grammar ensures a statement never appears inside an expression. C does the same, but many other languages which have a "lambda" or function literal syntax do not.

  An example in JavaScript:

  ```
  console.log(function() {
    statement();
  });
  ```

  Here, the `console.log()` *expression* contains a function literal which in turn contains the *statement* `statement();`.

> Python would need a different set of rules for implicitly joining lines if you could get back *into* a statement where newlines should become meaningful while still nested inside brackets.
>
> - JavaScript's "automatic semicolon insertion" rule is the real odd one. Where other languages assume most newlines *are* meaningful and only a few should be ignored in multi-line statements, JS assumes the opposite. It treats all of your newlines as meaningless whitespace *unless* it encounters a parse error. If it does, it goes back and tries turning the previous newline into a semicolon to get something grammatically valid.
>
>   This design note would turn into a design diatribe if I went into complete detail about how that even *works*, much less all the various ways that JavaScript's "solution" is a bad idea. It's a mess. JavaScript is the only language I know where many style guides demand explicit semicolons after every statement even though the language theoretically lets you elide them.
>
> If you're designing a new language, you almost surely *should* avoid an explicit statement terminator. Programmers are creatures of fashion like other humans, and semicolons are as passé as ALL CAPS KEYWORDS. Just make sure you pick a set of rules that make sense for your language's particular grammar and idioms. And don't do what JavaScript did.

现在的程序员已经被越来越多的语言选择宠坏了，对语法也越来越挑剔。他们希望自己的代码看起来干净、现代化。几乎每一种新语言都会放弃一个小的语法点（一些古老的语言，比如BASIC从来没有过），那就是将`;`作为显式的语句结束符。

相对地，它们将"有意义的"换行符看作是语句结束符。这里所说的"有意义的"是有挑战性的部分。尽管*大多数的*语句都是在同一行，但有时你需要将一个语句扩展到多行。这些混杂的换行符不应该被视作结束符。

大多数明显的应该忽略换行的情况都很容易发现，但也有少数讨厌的情况：

- 返回值在下一行：

  ```
  if (condition) return
  "value"
  ```

  "value"是要返回的值吗？还是说我们有一个空的`return`语句，后面跟着包含一个字符串字面量的表达式语句。

- 下一行中有带圆括号的表达式：

  ```
  func
  (parenthesized)
  ```

  这是一个对`func(parenthesized)`的调用，还是两个表达式语句，一个用于`func`，一个用于圆括号表达式？

- "-"号在下一行：

```
first
-second
```

这是一个中缀表达式——`first - second`，还是两个表达式语句，一个是`first`，另一个是对`second`取负？

在所有这些情况下，无论是否将换行符作为分隔符，都会产生有效的代码，但可能不是用户想要的代码。在不同的语言中，有各种不同的规则来决定哪些换行符是分隔符。下面是几个例子：

- Lua完全忽略了换行符，但是仔细地控制了它的语法，因此在大多数情况下，语句之间根本不需要分隔符。这段代码是完全合法的：

  ```
  a = 1 b = 2
  ```

  Lua要求 `return` 语句是一个块中的最后一条语句，从而避免 `return` 问题。如果在关键字`end`之前、`return`之后有一个值，这个值*必须*是用于`return`。对于其他两种情况来说，Lua允许显式的`;`并且期望用户使用它。在实践中，这种情况基本不会发生，因为在小括号或一元否定表达式语句中没有任何意义。

- Go会处理扫描器中的换行。如果在词法单元之后出现换行，并且该词法标记是已知可能结束语句的少数标记类型之一，则将换行视为分号，否则就忽略它。Go团队提供了一个规范的代码格式化程序gofmt，整个软件生态系统非常热衷于使用它，这确保了常用样式的代码能够很好地遵循这个简单的规则。

- Python将所有换行符都视为有效，除非在行末使用明确的反斜杠将其延续到下一行。但是，括号(`()`、`[]`或`{}`)内的任何换行都将被忽略。惯用的代码风格更倾向于后者。

  这条规则对 Python 很有效，因为它是一种高度面向语句的语言。特别是，Python 的语法确保了语句永远不会出现在表达式内。C语言也是如此，但许多其他有 "lambda "或函数字面语法的语言则不然。

  举一个JavaScript中的例子：

  ```
  console.log(function() {
    statement();
  });
  ```

  这里，`console.log()` *表达式*包含一个函数字面量，而这个函数字面量又包含 `statement();` *语句*。

  如果要求*进入*一个嵌套在括号内的语句中，并且要求其中的换行是有意义的，那么Python将需要一套不同的隐式连接行的规则^lambda。

- JavaScript的"自动分号插入"规则才是真正的奇葩。其他语言认为大多数换行符都是有意义的，只有少数换行符在多行语句中应该被忽略，而JS的假设恰恰相反。它将所有的换行符都视为无意义的空白，除非遇到解析错误。如果遇到了，它就会回过头来，尝试把之前的换行变成分号，以期得到正确的语法。

  如果我完全详细地介绍它是如何工作的，那么这个设计说明就会变成一篇设计檄文，更不用说JavaScript的"解决方案"从各种角度看都是个坏主意。真是一团糟。JavaScript是我所知道的唯一（风格指南和语言

> 本身背离）的语言，它的许多风格指南要求在每条语句后都显式地使用分号，但该语言却理论上允许您省略分号。

如果您要设计一种新的语言，则几乎可以肯定应该避免使用显式的语句终止符。 程序员和其他人类一样是时尚的动物，分号和ALL CAPS KEYWORDS(全大写关键字)一样已经过时了。只是要确保您选择了一套适用于您语言的特定语法和习语的规则即可。不要重蹈JavaScript的覆辙。

5.Representing Code 表示代码

> To dwellers in a wood, almost every species of tree has its voice as well as its feature.
>
> —— Thomas Hardy, *Under the Greenwood Tree*

对于森林中的居民来说，几乎每一种树都有它的声音和特点。

> In the last chapter, we took the raw source code as a string and transformed it into a slightly higher-level representation: a series of tokens. The parser we'll write in the next chapter takes those tokens and transforms them yet again, into an even richer, more complex representation.

在上一章中，我们以字符串形式接收原始源代码，并将其转换为一个稍高级别的表示：一系列词法标记。我们在下一章中要编写的解析器，会将这些词法标记再次转换为更丰富、更复杂的表示形式。

> Before we can produce that representation, we need to define it. That's the subject of this chapter. Along the way, we'll cover some theory around formal grammars, feel the difference between functional and object-oriented programming, go over a couple of design patterns, and do some metaprogramming.

在我们能够输出这种表示形式之前，我们需要先对其进行定义。这就是本章的主题^1。在这一过程中，我们将围绕形式化语法进行一些理论讲解，感受函数式编程和面向对象编程的区别，会介绍几种设计模式，并进行一些元编程。

> Before we do all that, let's focus on the main goal—a representation for code. It should be simple for the parser to produce and easy for the interpreter to consume. If you haven't written a parser or interpreter yet, those requirements aren't exactly illuminating. Maybe your intuition can help. What is your brain doing when you play the part of a *human* interpreter? How do you mentally evaluate an arithmetic expression like this:

在做这些事情之前，我们先关注一下主要目标——代码的表示形式。它应该易于解析器生成，也易于解释器使用。如果您还没有编写过解析器或解释器，那么这样的需求描述并不能很好地说明问题。也许你的直觉可以帮助你。当你扮演一个*人类*解释器的角色时，你的大脑在做什么？你如何在心里计算这样的算术表达式：

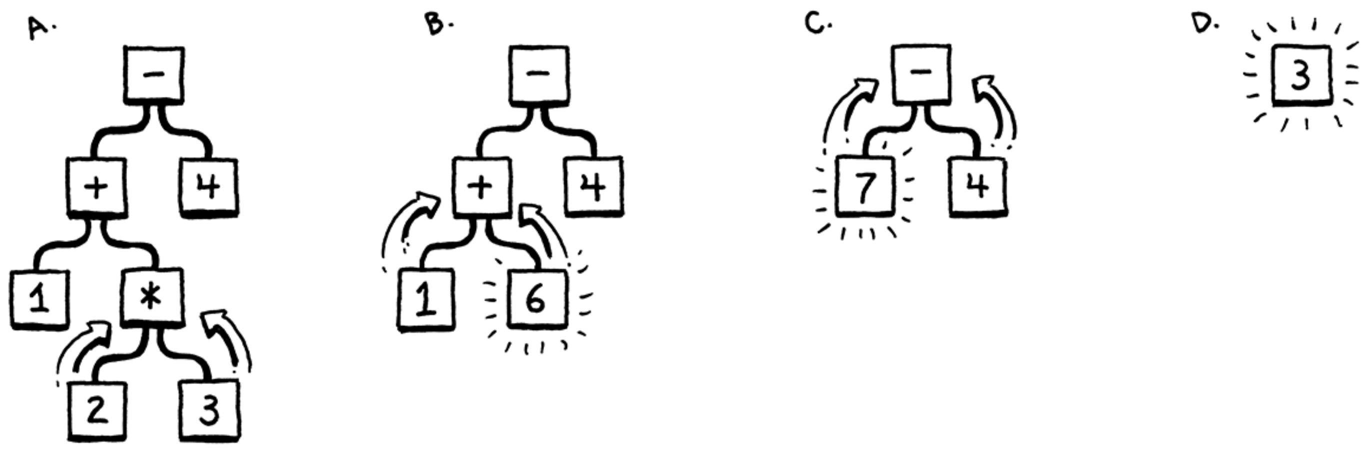
```
1 + 2 * 3 - 4
```

> Because you understand the order of operations—the old "Please Excuse My Dear Aunt Sally" stuff—you know that the multiplication is evaluated before the addition or subtraction. One way to visualize that precedence is using a tree. Leaf nodes are numbers, and interior nodes are operators with branches for each of their operands.

因为你已经理解了操作的顺序——以前的“Please Excuse My Dear Aunt Sally”之类^2，你知道乘法在加减操作之前执行。有一种方法可以将这种优先级进行可视化，那就是使用树^3。叶子节点是数字，内部节点是运算符，它们的每个操作数都对应一个分支。

> In order to evaluate an arithmetic node, you need to know the numeric values of its subtrees, so you have to evaluate those first. That means working your way from the leaves up to the root—a *post-order* traversal:

要想计算一个算术节点，你需要知道它的子树的数值，所以你必须先计算子树的结果。这意味着要从叶节点一直计算到根节点——*后序*遍历：



- A.从完整的树开始，先计算最下面的操作2*3；
- B.现在计算+；
- C.接下来，计算-；
- D.最终得到答案。

> If I gave you an arithmetic expression, you could draw one of these trees pretty easily. Given a tree, you can evaluate it without breaking a sweat. So it intuitively seems like a workable representation of our code is a tree that matches the grammatical structure—the operator nesting—of the language.

如果我给你一个算术表达式，你可以很容易地画出这样的树；给你一棵树，你也可以毫不费力地进行计算。因此，从直观上看，我们的代码的一种可行的表示形式是一棵与语言的语法结构（运算符嵌套）相匹配的树。

> We need to get more precise about what that grammar is then. Like lexical grammars in the last chapter, there is a long ton of theory around syntactic grammars. We're going into that theory a little more than we did when scanning because it turns out to be a useful tool throughout much of the interpreter. We start by moving one level up the Chomsky hierarchy . . .

那么我们需要更精确地了解这个语法是什么。就像上一章的词汇语法一样，围绕句法语法也有一大堆理论。我们要比之前处理扫描时投入更多精力去研究这个理论，因为它在整个解释器的很多地方都是一个有用的工具。我们先从乔姆斯基谱系中往上升一级……

5.1 Context-Free Grammars

5.1 上下文无关语法

> In the last chapter, the formalism we used for defining the lexical grammar—the rules for how characters get grouped into tokens—was called a *regular language*. That was fine for our scanner,

> which emits a flat sequence of tokens. But regular languages aren't powerful enough to handle expressions which can nest arbitrarily deeply.

在上一章中，我们用来定义词法语法（字符如何被分组为词法标记的规则）的形式体系，被称为*正则语言*。这对于我们的扫描器来说没什么问题，因为它输出的是一个扁平的词法标记序列。但正则语言还不够强大，无法处理可以任意深度嵌套的表达式。

> We need a bigger hammer, and that hammer is a **context-free grammar** (**CFG**). It's the next heaviest tool in the toolbox of **formal grammars**. A formal grammar takes a set of atomic pieces it calls its "alphabet". Then it defines a (usually infinite) set of "strings" that are "in" the grammar. Each string is a sequence of "letters" in the alphabet.

我们还需要一个更强大的工具，就是上下文无关语法(**context-free grammar**，CFG)。它是形式化语法的工具箱中下一个最重的工具。一个形式化语法需要一组原子片段，它称之为 "alphabet（字母表）"。然后它定义了一组（通常是无限的）"strings（字符串）"，这些字符串 "包含"在语法中。每个字符串都是字母表中 "letters（字符）"的序列。

> I'm using all those quotes because the terms get a little confusing as you move from lexical to syntactic grammars. In our scanner's grammar, the alphabet consists of individual characters and the strings are the valid lexemes—roughly "words". In the syntactic grammar we're talking about now, we're at a different level of granularity. Now each "letter" in the alphabet is an entire token and a "string" is a sequence of *tokens*—an entire expression.

我这里使用引号是因为当你从词法转到文法语法时，这些术语会让你有点困惑。在我们的扫描器词法中，alphabet（字母表）由单个字符组成，strings（字符串）是有效的词素（粗略的说，就是"单词"）。在现在讨论的句法语法中，我们处于一个不同的粒度水平。现在，字母表中的一个"letters（字符）"是一个完整的词法标记，而"strings（字符串）"是一个词法标记系列——一个完整的表达式。

> Oof. Maybe a table will help:

嗯，使用表格可能更有助于理解：

Terminology **术语**		**Lexical grammar 词法**	**Syntactic grammar 语法**
The "alphabet" is . . . 字母表	→	Characters 字符	Tokens 词法标记
A "string" is . . . 字符串	→	Lexeme or token 词素或词法标记	Expression 表达式
It's implemented by the . . . 实现	→	Scanner 扫描器	Parser 解析器

> A formal grammar's job is to specify which strings are valid and which aren't. If we were defining a grammar for English sentences, "eggs are tasty for breakfast" would be in the grammar, but "tasty breakfast for are eggs" would probably not.

形式化语法的工作是指定哪些字符串有效，哪些无效。如果我们要为英语句子定义一个语法，"eggs are tasty for breakfast "会包含在语法中，但 "tasty breakfast for are eggs "可能不会。

> ### 5.1.1 Rules for grammars

5.1.1 语法规则

> How do we write down a grammar that contains an infinite number of valid strings? We obviously can't list them all out. Instead, we create a finite set of rules. You can think of them as a game that you can "play" in one of two directions.

我们如何写下一个包含无限多有效字符串的语法?我们显然无法一一列举出来。相反，我们创建了一组有限的规则。你可以把它们想象成一场你可以朝两个方向"玩"的游戏。

> If you start with the rules, you can use them to *generate* strings that are in the grammar. Strings created this way are called **derivations** because each is *derived* from the rules of the grammar. In each step of the game, you pick a rule and follow what it tells you to do. Most of the lingo around formal grammars comes from playing them in this direction. Rules are called **productions** because they *produce* strings in the grammar.

如果你从规则入手，你可以用它们*生成*语法中的字符串。以这种方式创建的字符串被称为**推导式**（派生式），因为每个字符串都是从语法规则中*推导*出来的。在游戏的每一步中，你都要选择一条规则，然后按照它告诉你的去做。围绕形式化语法的大部分语言都倾向这种方式。规则被称为**生成式**，因为它们生成了语法中的字符串。

> Each production in a context-free grammar has a **head**—its name—and a **body**, which describes what it generates. In its pure form, the body is simply a list of symbols. Symbols come in two delectable flavors:

上下文无关语法中的每个生成式都有一个**头部**（其名称）和描述其生成内容的**主体**^4。在纯粹的形式上看，主体只是一系列符号。符号有两种：

> - A **terminal** is a letter from the grammar's alphabet. You can think of it like a literal value. In the syntactic grammar we're defining, the terminals are individual lexemes—tokens coming from the scanner like `if` or `1234`.
>
>   These are called "terminals", in the sense of an "end point" because they don't lead to any further "moves" in the game. You simply produce that one symbol.
>
> - A **nonterminal** is a named reference to another rule in the grammar. It means "play that rule and insert whatever it produces here". In this way, the grammar composes.

- **终止符**是语法字母表中的一个字母。你可以把它想象成一个字面值。在我们定义的语法中，终止符是独立的词素——来自扫描器的词法标记，比如 `if` 或 `1234`。

  这些词素被称为"终止符"，表示"终点"，因为它们不会导致游戏中任何进一步的 "动作"。你只是简单地产生了那一个符号。

- 非终止符是对语法中另一条规则的命名引用。它的意思是 "执行那条规则，然后将它产生的任何内容插入这里"。这样，语法就构成了。

> There is one last refinement: you may have multiple rules with the same name. When you reach a nonterminal with that name, you are allowed to pick any of the rules for it, whichever floats your boat.

还有最后一个细节：你可以有多个同名的规则。当你遇到一个该名字的非终止符时，你可以为它选择任何一条规则，随您喜欢。

> To make this concrete, we need a way to write down these production rules. People have been trying to crystallize grammar all the way back to Pāṇini's *Ashtadhyayi*, which codified Sanskrit grammar a mere couple thousand years ago. Not much progress happened until John Backus and company needed a notation for specifying ALGOL 58 and came up with Backus-Naur form (**BNF**). Since then, nearly everyone uses some flavor of BNF, tweaked to their own tastes.

为了让这个规则具体化，我们需要一种方式来写下这些生成规则。人们一直试图将语法具体化，可以追溯到Pāṇini的*Ashtadhyayi*，他在几千年前编纂了梵文语法。直到约翰-巴库斯（John Backus）和公司需要一个声明ALGOL 58的符号，并提出了巴科斯范式（**BNF**），才有了很大的进展。从那时起，几乎每个人都在使用BNF的某种变形，并根据自己的需要进行了调整^5。

> I tried to come up with something clean. Each rule is a name, followed by an arrow (`→`), followed by a sequence of symbols, and finally ending with a semicolon (`;`). Terminals are quoted strings, and nonterminals are lowercase words.

我试图提出一个简单的形式。 每个规则都是一个名称，后跟一个箭头（`→`），后跟一系列符号，最后以分号（`;`）结尾。 终止符是带引号的字符串，非终止符是小写的单词。

> Using that, here's a grammar for breakfast menus:

以此为基础，下面是一个早餐菜单语法：

```
breakfast  → protein "with" breakfast "on the side" ;
breakfast  → protein ;
breakfast  → bread ;

protein    → crispiness "crispy" "bacon" ;
protein    → "sausage" ;
protein    → cooked "eggs" ;

crispiness → "really" ;
crispiness → "really" crispiness ;

cooked     → "scrambled" ;
cooked     → "poached" ;
cooked     → "fried" ;

bread      → "toast" ;
bread      → "biscuits" ;
bread      → "English muffin" ;
```

> We can use this grammar to generate random breakfasts. Let's play a round and see how it works. By age-old convention, the game starts with the first rule in the grammar, here `breakfast`. There are three productions for that, and we randomly pick the first one. Our resulting string looks like:

我们可以使用这个语法来随机生成早餐。我们来玩一轮，看看它是如何工作的。按照老规矩，游戏从语法中的第一个规则开始，这里是`breakfast`。它有三个生成式，我们随机选择第一个。我们得到的字符串是这样的：

```
protein "with" breakfast "on the side"
```

> We need to expand that first nonterminal, `protein`, so we pick a production for that. Let’s pick:

我们需要展开第一个非终止符，`protein`，所有我们要选择它对应的一个生成式。我们选：

```
protein → cooked "eggs" ;
```

> Next, we need a production for `cooked`, and so we pick `"poached"`. That’s a terminal, so we add that. Now our string looks like:

接下来，我们需要 `cooked`的生成式，我们选择 `"poached"`。这是一个终止符，我们加上它。现在我们的字符串是这样的：

```
"poached" "eggs" "with" breakfast "on the side"
```

> The next non-terminal is `breakfast` again. The first `breakfast` production we chose recursively refers back to the `breakfast` rule. Recursion in the grammar is a good sign that the language being defined is context-free instead of regular. In particular, recursion where the recursive nonterminal has productions on both sides implies that the language is not regular.

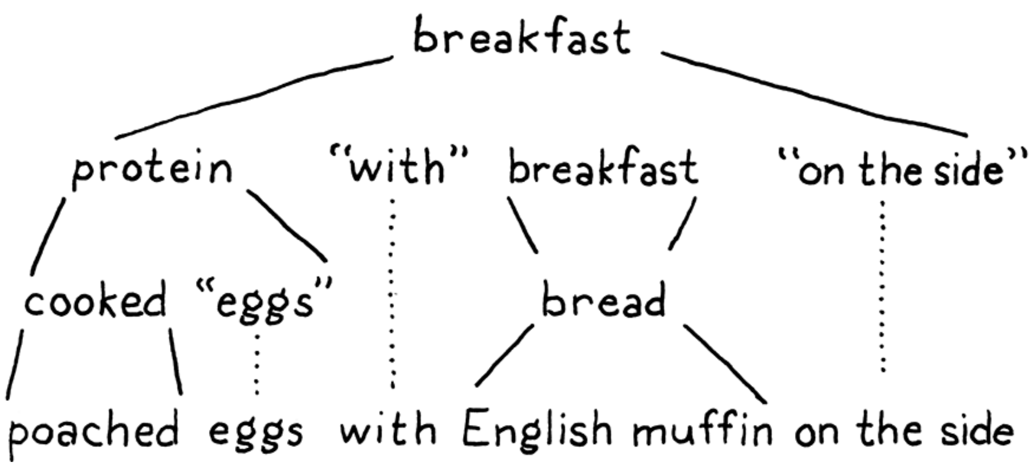
下一个非终止符还是`breakfast` ，我们开始选择的`breakfast` 生成式递归地指向了`breakfast` 规则^6。语法中的递归是一个很好的标志，表明所定义的语言是上下文无关的，而不是正则的。特别是，递归非终止符两边都有生成式的递归，意味着语言不是正则的。

> We could keep picking the first production for `breakfast` over and over again yielding all manner of breakfasts like “bacon with sausage with scrambled eggs with bacon . . . ” We won’t though. This time we’ll pick `bread`. There are three rules for that, each of which contains only a terminal. We’ll pick “English muffin”.

我们可以不断选择`breakfast` 的第一个生成式，以做出各种各样的早餐："bacon with sausage with scrambled eggs with bacon . . . ”，【存疑，按照规则设置，这里应该不会出现以bacon开头的字符串，原文可能有误】但我们不会这样做。这一次我们选择`bread`。有三个对应的规则，每个规则只包含一个终止符。我们选 "English muffin"。

> With that, every nonterminal in the string has been expanded until it finally contains only terminals and we’re left with:

这样一来，字符串中的每一个非终止符都被展开了，直到最后只包含终止符，我们就剩下：



> Throw in some ham and Hollandaise, and you've got eggs Benedict.

再加上一些火腿和荷兰酱，你就得到了松饼蛋。

> Any time we hit a rule that had multiple productions, we just picked one arbitrarily. It is this flexibility that allows a short number of grammar rules to encode a combinatorially larger set of strings. The fact that a rule can refer to itself—directly or indirectly—kicks it up even more, letting us pack an infinite number of strings into a finite grammar.

每当我们遇到具有多个结果的规则时，我们都只是随意选择了一个。 正是这种灵活性允许用少量的语法规则来编码出组合性更强的字符串集。一个规则可以直接或间接地引用它自己，这就更提高了它的灵活性，让我们可以将无限多的字符串打包到一个有限的语法中。

> ### 5.1.2 Enhancing our notation

5.1.2 增强符号

> Stuffing an infinite set of strings in a handful of rules is pretty fantastic, but let's take it further. Our notation works, but it's a little tedious. So, like any good language designer, we'll sprinkle some syntactic sugar on top. In addition to terminals and nonterminals, we'll allow a few other kinds of expressions in the body of a rule:

在少量的规则中可以填充无限多的字符串是相当奇妙的，但是我们可以更进一步。我们的符号是可行的，但有点乏味。所以，就像所有优秀的语言设计者一样，我们会在上面撒一些语法糖。除了终止符和非终止符之外，我们还允许在规则的主体中使用一些其他类型的表达式：

- > Instead of repeating the rule name each time we want to add another production for it, we'll allow a series of productions separated by a pipe (`|`).

  我们将允许一系列由管道符(`|`)分隔的生成式，避免在每次在添加另一个生成式时重复规则名称。

  ```
  bread → "toast" | "biscuits" | "English muffin" ;
  ```

- > Further, we'll allow parentheses for grouping and then allow `|` within that to select one from a series of options within the middle of a production.

  此外，我们允许用括号进行分组，然后在分组中可以用`|`表示从一系列生成式中选择一个。

```
protein → ( "scrambled" | "poached" | "fried" ) "eggs" ;
```

- > Using recursion to support repeated sequences of symbols has a certain appealing purity, but it's kind of a chore to make a separate named sub-rule each time we want to loop. So, we also use a postfix * to allow the previous symbol or group to be repeated zero or more times.

  使用递归来支持符号的重复序列有一定的吸引力，但每次我们要循环的时候，都要创建一个单独的命名子规则，有点繁琐^7。所以，我们也使用后缀*来允许前一个符号或组重复零次或多次。

  ```
  crispiness → "really" "really"* ;
  ```

- > A postfix + is similar, but requires the preceding production to appear at least once.

  后缀+与此类似，但要求前面的生成式至少出现一次。

  ```
  crispiness → "really"+ ;
  ```

- > A postfix ? is for an optional production. The thing before it can appear zero or one time, but not more.

  后缀 ? 表示可选生成式，它之前的生成式可以出现零次或一次，但不能出现多次。

  ```
  breakfast → protein ( "with" breakfast "on the side" )? ;
  ```

> With all of those syntactic niceties, our breakfast grammar condenses down to:

有了所有这些语法上的技巧，我们的早餐语法浓缩为：

```
breakfast → protein ( "with" breakfast "on the side" )?
          | bread ;

protein   → "really"+ "crispy" "bacon"
          | "sausage"
          | ( "scrambled" | "poached" | "fried" ) "eggs" ;

bread     → "toast" | "biscuits" | "English muffin" ;
```

> Not too bad, I hope. If you're used to grep or using regular expressions in your text editor, most of the punctuation should be familiar. The main difference is that symbols here represent entire tokens, not single characters.

我希望还不算太坏。如果你习惯使用grep或在你的文本编辑器中使用正则表达式，大多数的标点符号应该是熟悉的。主要区别在于，这里的符号代表整个标记，而不是单个字符。

> We'll use this notation throughout the rest of the book to precisely describe Lox's grammar. As you work on programming languages, you'll find that context-free grammars (using this or EBNF or some other notation) help you crystallize your informal syntax design ideas. They are also a handy medium for communicating with other language hackers about syntax.

在本书的其余部分中，我们将使用这种表示法来精确地描述Lox的语法。当您使用编程语言时，您会发现上下文无关的语法(使用此语法或EBNF或其他一些符号)可以帮助您将非正式的语法设计思想具体化。它们也是与其他语言黑客交流语法的方便媒介。

> The rules and productions we define for Lox are also our guide to the tree data structure we're going to implement to represent code in memory. Before we can do that, we need an actual grammar for Lox, or at least enough of one for us to get started.

我们为Lox定义的规则和生成式也是我们将要实现的树数据结构（用于表示内存中的代码）的指南。 在此之前，我们需要为Lox编写一个实际的语法，或者至少要有一个足够上手的语法。

> ### 5.1.3 A Grammar for Lox expressions

5.1.3 Lox表达式语法

> In the previous chapter, we did Lox's entire lexical grammar in one fell swoop. Every keyword and bit of punctuation is there. The syntactic grammar is larger, and it would be a real bore to grind through the entire thing before we actually get our interpreter up and running.

在上一章中，我们一气呵成地完成了Lox的全部词汇语法，包括每一个关键词和标点符号。但句法语法的规模更大，如果在我们真正启动并运行解释器之前，就要把整个语法啃完，那就太无聊了。

> Instead, we'll crank through a subset of the language in the next couple of chapters. Once we have that mini-language represented, parsed, and interpreted, then later chapters will progressively add new features to it, including the new syntax. For now, we are going to worry about only a handful of expressions:

相反，我们将在接下来的几章中摸索该语言的一个子集。一旦我们可以对这个迷你语言进行表示、解析和解释，那么在之后的章节中将逐步为它添加新的特性，包括新的语法。现在，我们只关心几个表达式：

- > **Literals.** Numbers, strings, Booleans, and `nil`.

  字面量。数字、字符串、布尔值以及`nil`。

- > **Unary expressions.** A prefix `!` to perform a logical not, and `-` to negate a number.

  一元表达式。前缀`!`执行逻辑非运算，`-`对数字求反。

- > **Binary expressions.** The infix arithmetic (`+`, `-`, `*`, `/`) and logic operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) we know and love.

  二元表达式。我们已经知道的中缀算术符（`+`, `-`, `*`, `/`）和逻辑运算符（`==`, `! =`, `<`, `<=`, `>`, `> =`）。

- > **Parentheses.** A pair of `(` and `)` wrapped around an expression.

括号。表达式前后的一对`(`和`)`。

> That gives us enough syntax for expressions like:

这已经为表达式提供了足够的语法，例如：

```
1 - (2 * 3) < 4 == false
```

> Using our handy dandy new notation, here's a grammar for those:

使用我们的新符号，下面是语法的表示：

```
expression     → literal
               | unary
               | binary
               | grouping ;

literal        → NUMBER | STRING | "true" | "false" | "nil" ;
grouping       → "(" expression ")" ;
unary          → ( "-" | "!" ) expression ;
binary         → expression operator expression ;
operator       → "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+"  | "-"  | "*" | "/" ;
```

> There's one bit of extra metasyntax here. In addition to quoted strings for terminals that match exact lexemes, we `CAPITALIZE` terminals that are a single lexeme whose text representation may vary. `NUMBER` is any number literal, and `STRING` is any string literal. Later, we'll do the same for `IDENTIFIER`.

这里有一点额外的元语法。除了与精确词素相匹配的终止符会加引号外，我们还对表示单一词素的终止符进行`大写化`，这些词素的文本表示方式可能会有所不同。`NUMBER`是任何数字字面量，`STRING`是任何字符串字面量。稍后，我们将对`IDENTIFIER`进行同样的处理^8。

> This grammar is actually ambiguous, which we'll see when we get to parsing it. But it's good enough for now.

这个语法实际上是有歧义的，我们在解析它时就会看到这一点。但现在这已经足够了。

> ## 5.2 Implementing Syntax Trees

5.2 实现语法树

> Finally, we get to write some code. That little expression grammar is our skeleton. Since the grammar is recursive—note how `grouping`, `unary`, and `binary` all refer back to `expression`—our data structure will form a tree. Since this structure represents the syntax of our language, it's called a **syntax tree**.

最后，我们要写一些代码。这个小小的表达式语法就是我们的骨架。由于语法是递归的——请注意`grouping`, `unary`, 和 `binary` 都是指回`expression`的——我们的数据结构将形成一棵树。因为这个结构代表了我们语言的语法，所以叫做**语法树**^9。

> Our scanner used a single Token class to represent all kinds of lexemes. To distinguish the different kinds—think the number `123` versus the string `"123"`—we included a simple TokenType enum. Syntax trees are not so homogeneous. Unary expressions have a single operand, binary expressions have two, and literals have none.

我们的扫描器使用一个单一的 Token 类来表示所有类型的词素。为了区分不同的种类——想想数字 `123` 和字符串 `"123"`——我们创建了一个简单的 TokenType 枚举。语法树并不是那么同质的^10。一元表达式只有一个操作数，二元表达式有两个操作数，而字面量则没有。

> We *could* mush that all together into a single Expression class with an arbitrary list of children. Some compilers do. But I like getting the most out of Java's type system. So we'll define a base class for expressions. Then, for each kind of expression—each production under `expression`—we create a subclass that has fields for the nonterminals specific to that rule. This way, we get a compile error if we, say, try to access the second operand of a unary expression.

我们 *可以*将所有这些内容整合到一个包含任意子类列表的 Expression 类中。有些编译器会这么做。但我希望充分利用Java的类型系统。所以我们将为表达式定义一个基类。然后，对于每一种表达式——`expression`下的每一个生成式——我们创建一个子类，这个子类有该规则所特有的非终止符字段。这样，如果试图访问一元表达式的第二个操作数，就会得到一个编译错误。

> Something like this:

类似这样^11：

```java
package com.craftinginterpreters.lox;

abstract class Expr {
  static class Binary extends Expr {
    Binary(Expr left, Token operator, Expr right) {
      this.left = left;
      this.operator = operator;
      this.right = right;
    }

    final Expr left;
    final Token operator;
    final Expr right;
  }

  // Other expressions...
}
```

> Expr is the base class that all expression classes inherit from. As you can see from `Binary`, the subclasses are nested inside of it. There's no technical need for this, but it lets us cram all of the classes into a single Java file.

Expr是所有表达式类继承的基类。从`Binary`中可以看到，子类都嵌套在它的内部。这在技术上没有必要，但它允许我们将所有类都塞进一个Java文件中。

> ### 5.2.1 Disoriented objects

5.2.1 非面向对象

> You'll note that, much like the Token class, there aren't any methods here. It's a dumb structure. Nicely typed, but merely a bag of data. This feels strange in an object-oriented language like Java. Shouldn't the class *do stuff*?

你会注意到，（表达式类）像Token类一样，其中没有任何方法。这是一个很愚蠢的结构，巧妙的类型封装，但仅仅是一包数据。这在Java这样的面向对象语言中会有些奇怪，难道类不是应该*做一些事情*吗？

> The problem is that these tree classes aren't owned by any single domain. Should they have methods for parsing since that's where the trees are created? Or interpreting since that's where they are consumed? Trees span the border between those territories, which means they are really owned by *neither*.

问题在于这些树类不属于任何单个的领域。树是在解析的时候创建的，难道类中应该有解析对应的方法？或者因为树结构在解释的时候被消费，其中是不是要提供解释相关的方法？树跨越了这些领域之间的边界，这意味着它们实际上不属于任何一方。

> In fact, these types exist to enable the parser and interpreter to *communicate*. That lends itself to types that are simply data with no associated behavior. This style is very natural in functional languages like Lisp and ML where *all* data is separate from behavior, but it feels odd in Java.

事实上，这些类型的存在是为了让解析器和解释器能够*进行交流*。这就适合于那些只是简单的数据而没有相关行为的类型。这种风格在Lisp和ML这样的函数式语言中是非常自然的，因为在这些语言中，*所有的*数据和行为都是分开的，但是在Java中感觉很奇怪。

> Functional programming aficionados right now are jumping up to exclaim "See! Object-oriented languages are a bad fit for an interpreter!" I won't go that far. You'll recall that the scanner itself was admirably suited to object-orientation. It had all of the mutable state to keep track of where it was in the source code, a well-defined set of public methods, and a handful of private helpers.

函数式编程的爱好者们现在都跳起来惊呼："看吧！面向对象的语言不适合作为解释器！"我不会那么过分的。您可能还记得，扫描器本身非常适合面向对象。它包含所有的可变状态来跟踪其在源代码中的位置、一组定义良好的公共方法和少量的私有辅助方法。

> My feeling is that each phase or part of the interpreter works fine in an object-oriented style. It is the data structures that flow between them that are stripped of behavior.

我的感觉是，在面向对象的风格下，解释器的每个阶段或部分都能正常工作。只不过在它们之间流动的数据结构剥离了行为。

> ### 5.2.2 Metaprogramming the trees

5.2.2 节点树元编程

> Java can express behavior-less classes, but I wouldn't say that it's particularly great at it. Eleven lines of code to stuff three fields in an object is pretty tedious, and when we're all done, we're going to have 21 of these classes.

Java可以表达无行为的类，但很难说它特别擅长。用11行代码在一个对象中填充3个字段是相当乏味的，当我们全部完成后，我们将有21个这样的类。

> I don’t want to waste your time or my ink writing all that down. Really, what is the essence of each subclass? A name, and a list of typed fields. That’s it. We’re smart language hackers, right? Let’s automate.

我不想浪费你的时间或我的墨水把这些都写下来。真的，每个子类的本质是什么?一个名称和一个字段列表而已。我们是聪明的语言黑客，对吧?我们把它自动化^12。

> Instead of tediously handwriting each class definition, field declaration, constructor, and initializer, we’ll hack together a script that does it for us. It has a description of each tree type—its name and fields—and it prints out the Java code needed to define a class with that name and state.

与其繁琐地手写每个类的定义、字段声明、构造函数和初始化器，我们一起编写一个脚本来完成任务。 它具有每种树类型（名称和字段）的描述，并打印出定义具有该名称和状态的类所需的Java代码。

> This script is a tiny Java command-line app that generates a file named “Expr.java”:

该脚本是一个微型Java命令行应用程序，它生成一个名为“ Expr.java”的文件：

tool/GenerateAst.java ，创建新文件

```
package com.craftinginterpreters.tool;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.List;

public class GenerateAst {
  public static void main(String[] args) throws IOException {
    if (args.length != 1) {
      System.err.println("Usage: generate_ast <output directory>");
      System.exit(64);
    }
    String outputDir = args[0];
  }
}
```

> Note that this file is in a different package, `.tool` instead of `.lox`. This script isn’t part of the interpreter itself. It’s a tool *we*, the people hacking on the interpreter, run ourselves to generate the syntax tree classes. When it’s done, we treat “Expr.java” like any other file in the implementation. We are merely automating how that file gets authored.

注意，这个文件在另一个包中，是`.tool`而不是`.lox`。这个脚本并不是解释器本身的一部分，它是一个工具，我们这种编写解释器的人，通过运行该脚本来生成语法树类。完成后，我们把“Expr.java”与实现中的其它文件进行相同的处理。我们只是自动化了文件的生成方式。

> To generate the classes, it needs to have some description of each type and its fields.

为了生成类，还需要对每种类型及其字段进行一些描述。

tool/GenerateAst.java ，在 main()方法中添加

```
    String outputDir = args[0];
    // 新增部分开始
    defineAst(outputDir, "Expr", Arrays.asList(
      "Binary   : Expr left, Token operator, Expr right",
      "Grouping : Expr expression",
      "Literal  : Object value",
      "Unary    : Token operator, Expr right"
    ));
    // 新增部分结束
  }
```

> For brevity’s sake, I jammed the descriptions of the expression types into strings. Each is the name of the class followed by `:` and the list of fields, separated by commas. Each field has a type and a name.

为简便起见，我将表达式类型的描述放入了字符串中。 每一项都包括类的名称，后跟`:`和以逗号分隔的字段列表。 每个字段都有一个类型和一个名称。

> The first thing `defineAst()` needs to do is output the base Expr class.

`defineAst()`需要做的第一件事是输出基类Expr。

tool/GenerateAst.java，在 main()方法后添加：

```
  private static void defineAst(
      String outputDir, String baseName, List<String> types)
      throws IOException {
    String path = outputDir + "/" + baseName + ".java";
    PrintWriter writer = new PrintWriter(path, "UTF-8");

    writer.println("package com.craftinginterpreters.lox;");
    writer.println();
    writer.println("import java.util.List;");
    writer.println();
    writer.println("abstract class " + baseName + " {");

    writer.println("}");
    writer.close();
  }
```

> When we call this, `baseName` is “Expr”, which is both the name of the class and the name of the file it outputs. We pass this as an argument instead of hardcoding the name because we’ll add a separate family of classes later for statements.

我们调用这个函数时，`baseName`是“Expr”，它既是类的名称，也是它输出的文件的名称。我们将它作为参数传递，而不是对名称进行硬编码，因为稍后我们将为语句添加一个单独的类族。

> Inside the base class, we define each subclass.

在基类内部，我们定义每个子类。

tool/GenerateAst.java，在 defineAst()类中添加^13：

```
    writer.println("abstract class " + baseName + " {");
    // 新增部分开始
    // The AST classes.
    for (String type : types) {
      String className = type.split(":")[0].trim();
      String fields = type.split(":")[1].trim();
      defineType(writer, baseName, className, fields);
    }
    // 新增部分结束
    writer.println("}");
```

> That code, in turn, calls:

这段代码依次调用：

tool/GenerateAst.java，在 defineAst()后面添加：

```
  private static void defineType(
      PrintWriter writer, String baseName,
      String className, String fieldList) {
    writer.println("  static class " + className + " extends " +
        baseName + " {");

    // Constructor.
    writer.println("    " + className + "(" + fieldList + ") {");

    // Store parameters in fields.
    String[] fields = fieldList.split(", ");
    for (String field : fields) {
      String name = field.split(" ")[1];
      writer.println("      this." + name + " = " + name + ";");
    }

    writer.println("    }");

    // Fields.
    writer.println();
    for (String field : fields) {
      writer.println("    final " + field + ";");
    }

    writer.println("  }");
  }
```

> There we go. All of that glorious Java boilerplate is done. It declares each field in the class body. It defines a constructor for the class with parameters for each field and initializes them in the body.

好了。所有的Java模板都完成了。它在类体中声明了每个字段。它为类定义了一个构造函数，为每个字段提供参数，并在类体中对其初始化。

> Compile and run this Java program now and it blasts out a new ".java" file containing a few dozen lines of code. That file's about to get even longer.

现在编译并运行这个Java程序，它会生成一个新的". Java"文件，其中包含几十行代码。那份文件还会变得更长^14。

5.3 Working with Trees

5.3 处理树结构

> Put on your imagination hat for a moment. Even though we aren't there yet, consider what the interpreter will do with the syntax trees. Each kind of expression in Lox behaves differently at runtime. That means the interpreter needs to select a different chunk of code to handle each expression type. With tokens, we can simply switch on the TokenType. But we don't have a "type" enum for the syntax trees, just a separate Java class for each one.

先想象一下吧。尽管我们还没有到那一步，但请考虑一下解释器将如何处理语法树。Lox中的每种表达式在运行时的行为都不一样。这意味着解释器需要选择不同的代码块来处理每种表达式类型。对于词法标记，我们可以简单地根据`TokenType`进行转换。但是我们并没有为语法树设置一个 "type "枚举，只是为每个语法树单独设置一个Java类。

> We could write a long chain of type tests:

我们可以编写一长串类型测试：

```
if (expr instanceof Expr.Binary) {
  // ...
} else if (expr instanceof Expr.Grouping) {
  // ...
} else // ...
```

> But all of those sequential type tests are slow. Expression types whose names are alphabetically later would take longer to execute because they'd fall through more `if` cases before finding the right type. That's not my idea of an elegant solution.

但所有这些顺序类型测试都很慢。类型名称按字母顺序排列在后面的表达式，执行起来会花费更多的时间，因为在找到正确的类型之前，它们会遇到更多的`if`情况。这不是我认为的优雅解决方案。

> We have a family of classes and we need to associate a chunk of behavior with each one. The natural solution in an object-oriented language like Java is to put those behaviors into methods on the classes themselves. We could add an abstract `interpret()` method on Expr which each subclass would then implement to interpret itself.

我们有一个类族，我们需要将一组行为与每个类关联起来。在Java这样的面向对象语言中，最自然的解决方案是将这些行为放入类本身的方法中。我们可以在Expr上添加一个抽象的`interpret()`方法，然后每个子类都要实现这个方法来解释自己^15。

> This works alright for tiny projects, but it scales poorly. Like I noted before, these tree classes span a few domains. At the very least, both the parser and interpreter will mess with them. As you'll see later, we need to do name resolution on them. If our language was statically typed, we'd have a type checking pass.

这对于小型项目来说还行，但它的扩展性很差。就像我之前提到的，这些树类跨越了几个领域。至少，解析器和解释器都会对它们进行干扰。稍后您将看到，我们需要对它们进行名称解析。如果我们的语言是静态类型的，我们还需要做类型检查。

> If we added instance methods to the expression classes for every one of those operations, that would smush a bunch of different domains together. That violates separation of concerns and leads to hard-to-maintain code.

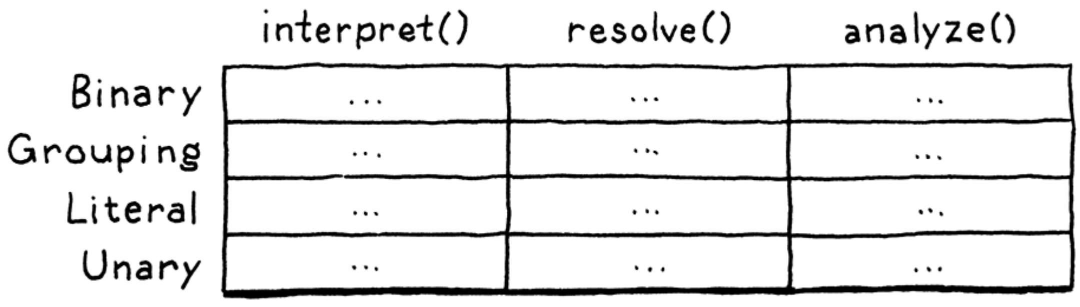
如果我们为每一个操作的表达式类中添加实例方法，就会将一堆不同的领域混在一起。这违反了关注点分离原则，并会产生难以维护的代码。

> ### 5.3.1 The expression problem

5.3.1 表达式问题

> This problem is more fundamental than it may seem at first. We have a handful of types, and a handful of high-level operations like "interpret". For each pair of type and operation, we need a specific implementation. Picture a table:

这个问题比起初看起来更基础。我们有一些类型，和一些高级操作，比如"解释"。对于每一对类型和操作，我们都需要一个特定的实现。画一个表:

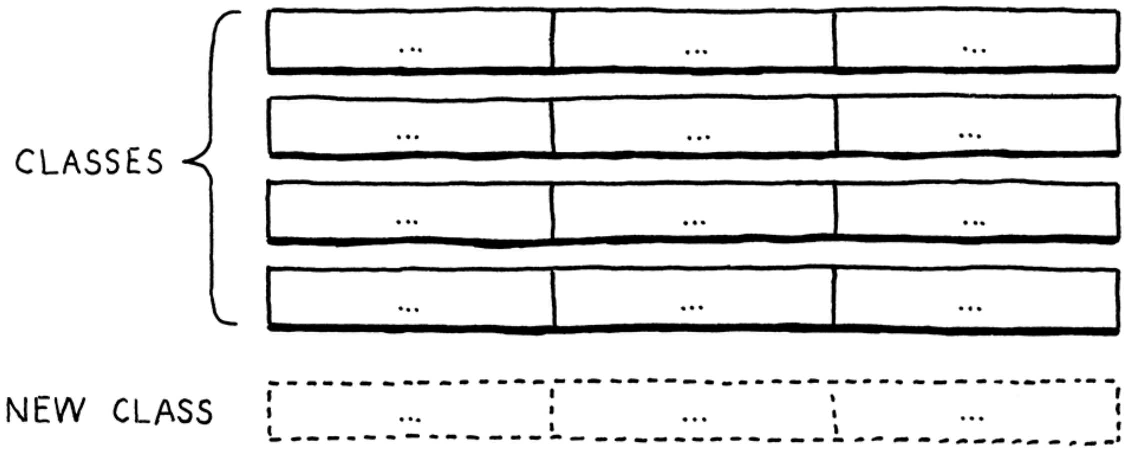


> Rows are types, and columns are operations. Each cell represents the unique piece of code to implement that operation on that type.

行是类型，列是操作。每个单元格表示在该类型上实现该操作的唯一代码段。

> An object-oriented language like Java assumes that all of the code in one row naturally hangs together. It figures all the things you do with a type are likely related to each other, and the language makes it easy to define them together as methods inside the same class.

像Java这样的面向对象的语言，假定一行中的所有代码都自然地挂在一起。它认为你对一个类型所做的所有事情都可能是相互关联的，而使用这类语言可以很容易将它们一起定义为同一个类里面的方法。



> This makes it easy to extend the table by adding new rows. Simply define a new class. No existing code has to be touched. But imagine if you want to add a new *operation*—a new column. In Java, that means cracking open each of those existing classes and adding a method to it.

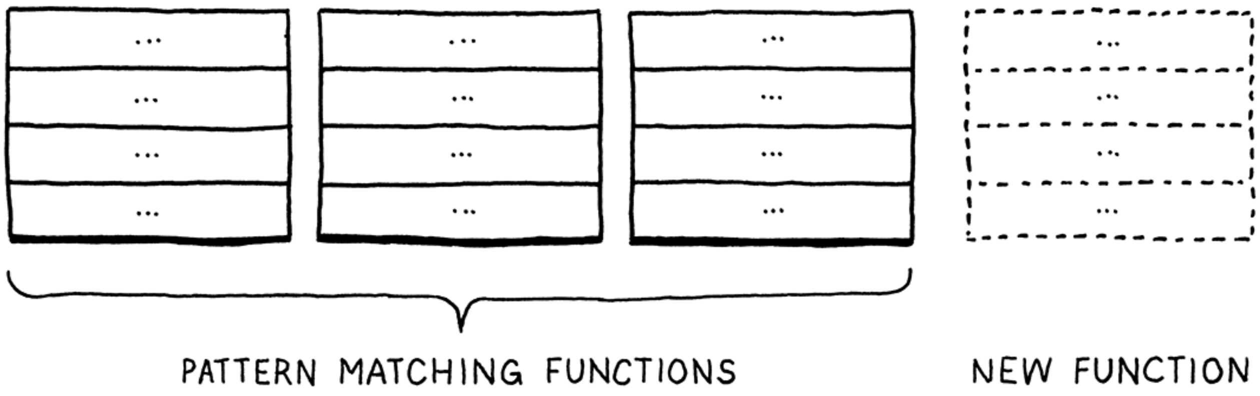
这种情况下，向表中加入新行来扩展列表是很容易的，简单地定义一个新类即可，不需要修改现有的代码。但是，想象一下，如果你要添加一个新操作（新的一列）。在Java中，这意味着要拆开已有的那些类并向其中添加方法。

> Functional paradigm languages in the ML family flip that around. There, you don't have classes with methods. Types and functions are totally distinct. To implement an operation for a number of different types, you define a single function. In the body of that function, you use *pattern matching*—sort of a type-based switch on steroids—to implement the operation for each type all in one place.

ML家族中的函数式范型反过来了^16。在这些语言中，没有带方法的类，类型和函数是完全独立的。要为许多不同类型实现一个操作，只需定义一个函数。在该函数体中，您可以使用*模式匹配*（某种基于类型的switch操作）在同一个函数中实现每个类型对应的操作。

> This makes it trivial to add new operations—simply define another function that pattern matches on all of the types.

这使得添加新操作非常简单——只需定义另一个与所有类型模式匹配的的函数即可。



> But, conversely, adding a new type is hard. You have to go back and add a new case to all of the pattern matches in all of the existing functions.

但是，反过来说，添加新类型是困难的。您必须回头向已有函数中的所有模式匹配添加一个新的case。

> Each style has a certain “grain” to it. That’s what the paradigm name literally says—an object-oriented language wants you to *orient* your code along the rows of types. A functional language instead encourages you to lump each column’s worth of code together into a *function*.

每种风格都有一定的 "纹路"。这就是范式名称的字面意思——面向对象的语言希望你按照类型的行来*组织*你的代码。而函数式语言则鼓励你把每一列的代码都归纳为一个*函数*。

> A bunch of smart language nerds noticed that neither style made it easy to add *both* rows and columns to the table. They called this difficulty the “expression problem” because—like we are now—they first ran into it when they were trying to figure out the best way to model expression syntax tree nodes in a compiler.

一群聪明的语言迷注意到，这两种风格都不容易向表格中添加行和列。他们称这个困难为“表达式问题”^17。就像我们现在一样，他们是在试图找出在编译器中建模表达式语法树节点的最佳方法时，第一次遇到了该问题。

> People have thrown all sorts of language features, design patterns, and programming tricks to try to knock that problem down but no perfect language has finished it off yet. In the meantime, the best we can do is try to pick a language whose orientation matches the natural architectural seams in the program we’re writing.

人们已经抛出了各种各样的语言特性、设计模式和编程技巧，试图解决这个问题，但还没有一种完美的语言能够解决它。与此同时，我们所能做的就是尽量选择一种与我们正在编写的程序的自然架构相匹配的语言。

> Object-orientation works fine for many parts of our interpreter, but these tree classes rub against the grain of Java. Fortunately, there’s a design pattern we can bring to bear on it.

面向对象在我们的解释器的许多部分都可以正常工作，但是这些树类与Java的本质背道而驰。 幸运的是，我们可以采用一种设计模式来解决这个问题。

> ### 5.3.2 The Visitor pattern

5.3.2 访问者模式

> The **Visitor pattern** is the most widely misunderstood pattern in all of *Design Patterns*, which is really saying something when you look at the software architecture excesses of the past couple of decades.

访问者模式是所有*设计模式*中最容易被误解的模式，当您回顾过去几十年的软件架构泛滥状况时，会发现确实如此。

> The trouble starts with terminology. The pattern isn’t about “visiting”, and the “accept” method in it doesn’t conjure up any helpful imagery either. Many think the pattern has to do with traversing trees, which isn’t the case at all. We *are* going to use it on a set of classes that are tree-like, but that’s a coincidence. As you’ll see, the pattern works as well on a single object.

问题出在术语上。这个模式不是关于“visiting（访问）”，它的 “accept”方法也没有让人产生任何有用的想象。许多人认为这种模式与遍历树有关，但事实并非如此。我们确实要在一组树结构的类上使用它，但这只是一个巧合。如您所见，该模式在单个对象上也可以正常使用。

> The Visitor pattern is really about approximating the functional style within an OOP language. It lets us add new columns to that table easily. We can define all of the behavior for a new operation on a set of

> types in one place, without having to touch the types themselves. It does this the same way we solve almost every problem in computer science: by adding a layer of indirection.

访问者模式实际上近似于OOP语言中的函数式。它让我们可以很容易地向表中添加新的列。我们可以在一个地方定义针对一组类型的新操作的所有行为，而不必触及类型本身。这与我们解决计算机科学中几乎所有问题的方式相同：添加中间层。

> Before we apply it to our auto-generated Expr classes, let’s walk through a simpler example. Say we have two kinds of pastries: beignets and crullers.

在将其应用到自动生成的Expr类之前，让我们先看一个更简单的例子。比方说我们有两种点心:Beignet(卷饼)和Cruller(油酥卷)。

```
 abstract class Pastry {
  }

  class Beignet extends Pastry {
  }

  class Cruller extends Pastry {
  }
```

> We want to be able to define new pastry operations—cooking them, eating them, decorating them, etc.—without having to add a new method to each class every time. Here’s how we do it. First, we define a separate interface.

我们希望能够定义新的糕点操作（烹饪，食用，装饰等），而不必每次都向每个类添加新方法。我们是这样做的。首先，我们定义一个单独的接口^18。

```
  interface PastryVisitor {
    void visitBeignet(Beignet beignet);
    void visitCruller(Cruller cruller);
  }
```

> Each operation that can be performed on pastries is a new class that implements that interface. It has a concrete method for each type of pastry. That keeps the code for the operation on both types all nestled snugly together in one class.

可以对糕点执行的每个操作都是实现该接口的新类。 它对每种类型的糕点都有具体的方法。 这样一来，针对两种类型的操作代码都紧密地嵌套在一个类中。

> Given some pastry, how do we route it to the correct method on the visitor based on its type? Polymorphism to the rescue! We add this method to Pastry:

给定一个糕点，我们如何根据其类型将其路由到访问者的正确方法？多态性拯救了我们！我们在Pastry中添加这个方法：

```java
abstract class Pastry {
  abstract void accept(PastryVisitor visitor);
}
```

> Each subclass implements it.

每个子类都需要实现该方法：

```java
class Beignet extends Pastry {
  @Override
  void accept(PastryVisitor visitor) {
    visitor.visitBeignet(this);
  }
}
```

> And:

以及：

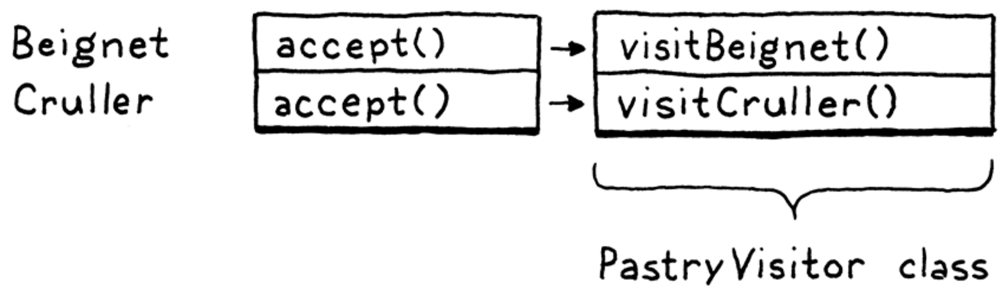
```java
class Cruller extends Pastry {
  @Override
  void accept(PastryVisitor visitor) {
    visitor.visitCruller(this);
  }
}
```

> To perform an operation on a pastry, we call its `accept()` method and pass in the visitor for the operation we want to execute. The pastry—the specific subclass's overriding implementation of `accept()`—turns around and calls the appropriate visit method on the visitor and passes *itself* to it.

要对糕点执行一个操作，我们就调用它的`accept()`方法，并将我们要执行的操作vistor作为参数传入该方法。pastry类——特定子类对`accept()`的重写实现——会反过来，在visitor上调用合适的visit方法，并将*自身*作为参数传入。

> That's the heart of the trick right there. It lets us use polymorphic dispatch on the *pastry* classes to select the appropriate method on the *visitor* class. In the table, each pastry class is a row, but if you look at all of the methods for a single visitor, they form a *column*.

这就是这个技巧的核心所在。它让我们可以在*pastry*类上使用多态派遣，在*visitor*类上选择合适的方法。对应在表格中，每个pastry类都是一行，但如果你看一个visitor的所有方法，它们就会形成一*列*。



> We added one accept() method to each class, and we can use it for as many visitors as we want without ever having to touch the pastry classes again. It's a clever pattern.

我们为每个类添加了一个accept （） 方法，我们可以根据需要将其用于任意数量的访问者，而无需再次修改*pastry*类。 这是一个聪明的模式。

> ### 5.3.3 Visitors for expressions

5.3.3 表达式访问者

> OK, let's weave it into our expression classes. We'll also refine the pattern a little. In the pastry example, the visit and accept() methods don't return anything. In practice, visitors often want to define operations that produce values. But what return type should accept() have? We can't assume every visitor class wants to produce the same type, so we'll use generics to let each implementation fill in a return type.

好的，让我们将它编入表达式类中。我们还要对这个模式进行一下完善。在糕点的例子中，visit和accept()方法没有返回任何东西。在实践中，访问者通常希望定义能够产生值的操作。但accept()应该具有什么返回类型呢？我们不能假设每个访问者类都想产生相同的类型，所以我们将使用泛型来让每个实现类自行填充一个返回类型。

> First, we define the visitor interface. Again, we nest it inside the base class so that we can keep everything in one file.

首先，我们定义访问者接口。同样，我们把它嵌套在基类中，以便将所有的内容都放在一个文件中。

tool/GenerateAst.java，在 defineAst()方法中添加：

```
    writer.println("abstract class " + baseName + " {");
    // 新增部分开始
    defineVisitor(writer, baseName, types);
    // 新增部分结束
    // The AST classes.
```

> That function generates the visitor interface.

这个函数会生成visitor接口。

tool/GenerateAst.java，在 defineAst()方法后添加：

```
  private static void defineVisitor(
      PrintWriter writer, String baseName, List<String> types) {
    writer.println("  interface Visitor<R> {");

    for (String type : types) {
      String typeName = type.split(":")[0].trim();
      writer.println("    R visit" + typeName + baseName + "(" +
          typeName + " " + baseName.toLowerCase() + ");");
    }

    writer.println("  }");
  }
```

> Here, we iterate through all of the subclasses and declare a visit method for each one. When we define new expression types later, this will automatically include them.

在这里，我们遍历所有的子类，并为每个子类声明一个visit方法。当我们以后定义新的表达式类型时，会自动包含这些内容。

> Inside the base class, we define the abstract `accept()` method.

在基类中，定义抽象 `accept()` 方法。

tool/GenerateAst.java，在 defineAst()方法中添加：

```
      defineType(writer, baseName, className, fields);
    }
    // 新增部分开始
    // The base accept() method.
    writer.println();
    writer.println("  abstract <R> R accept(Visitor<R> visitor);");
    // 新增部分结束
    writer.println("}");
```

> Finally, each subclass implements that and calls the right visit method for its own type.

最后，每个子类都实现该方法，并调用其类型对应的visit方法。

tool/GenerateAst.java，在 defineType()方法中添加：

```
    writer.println("    }");
    // 新增部分开始
    // Visitor pattern.
    writer.println();
    writer.println("    @Override");
    writer.println("    <R> R accept(Visitor<R> visitor) {");
    writer.println("      return visitor.visit" +
        className + baseName + "(this);");
    writer.println("    }");
    // 新增部分结束
    // Fields.
```

> There we go. Now we can define operations on expressions without having to muck with the classes or our generator script. Compile and run this generator script to output an updated “Expr.java” file. It contains a generated Visitor interface and a set of expression node classes that support the Visitor pattern using it.

这下好了。现在我们可以在表达式上定义操作，而且无需对类或生成器脚本进行修改。编译并运行这个生成器脚本，输出一个更新后的 "Expr.java "文件。该文件中包含一个生成的Visitor接口和一组使用该接口支持Visitor模式的表达式节点类。

> Before we end this rambling chapter, let’s implement that Visitor interface and see the pattern in action.

在结束这杂乱的一章之前，我们先实现一下这个Visitor接口，看看这个模式的运行情况。

> ## 5.4 A (Not Very) Pretty Printer

5.4 一个（不是很）漂亮的打印器

> When we debug our parser and interpreter, it’s often useful to look at a parsed syntax tree and make sure it has the structure we expect. We could inspect it in the debugger, but that can be a chore.

当我们调试解析器和解释器时，查看解析后的语法树并确保其与期望的结构一致通常是很有用的。我们可以在调试器中进行检查，但那可能有点难。

> Instead, we’d like some code that, given a syntax tree, produces an unambiguous string representation of it. Converting a tree to a string is sort of the opposite of a parser, and is often called “pretty printing” when the goal is to produce a string of text that is valid syntax in the source language.

相反，我们需要一些代码，在给定语法树的情况下，生成一个明确的字符串表示。将语法树转换为字符串是解析器的逆向操作，当我们的目标是产生一个在源语言中语法有效的文本字符串时，通常被称为 "漂亮打印"。

> That’s not our goal here. We want the string to very explicitly show the nesting structure of the tree. A printer that returned `1 + 2 * 3` isn’t super helpful if what we’re trying to debug is whether operator precedence is handled correctly. We want to know if the `+` or `*` is at the top of the tree.

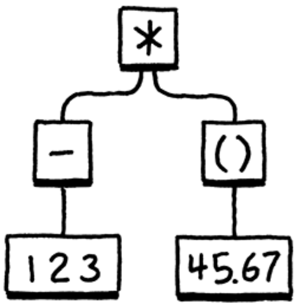
这不是我们的目标。我们希望字符串非常明确地显示树的嵌套结构。如果我们要调试的是操作符的优先级是否处理正确，那么返回`1 + 2 * 3`的打印器并没有什么用，我们想知道`+`或`*`是否在语法树的顶部。

> To that end, the string representation we produce isn’t going to be Lox syntax. Instead, it will look a lot like, well, Lisp. Each expression is explicitly parenthesized, and all of its subexpressions and tokens are contained in that.

因此，我们生成的字符串表示形式不是Lox语法。相反，它看起来很像Lisp。每个表达式都被显式地括起来，并且它的所有子表达式和词法标记都包含在其中。

> Given a syntax tree like:

给定一个语法树，如：



> It produces:

输出结果为：

```
(* (- 123) (group 45.67))
```

> Not exactly “pretty”, but it does show the nesting and grouping explicitly. To implement this, we define a new class.

不是很“漂亮”，但是它确实明确地展示了嵌套和分组。为了实现这一点，我们定义了一个新类。

lox/AstPrinter.java，创建新文件：

```
package com.craftinginterpreters.lox;

class AstPrinter implements Expr.Visitor<String> {
  String print(Expr expr) {
    return expr.accept(this);
  }
}
```

> As you can see, it implements the visitor interface. That means we need visit methods for each of the expression types we have so far.

如你所见，它实现了visitor接口。这意味着我们需要为我们目前拥有的每一种表达式类型提供visit方法。

lox/AstPrinter.java，在 print()方法后添加：

```
    return expr.accept(this);
  }
  // 新增部分开始
  @Override
  public String visitBinaryExpr(Expr.Binary expr) {
    return parenthesize(expr.operator.lexeme,
                        expr.left, expr.right);
  }

  @Override
  public String visitGroupingExpr(Expr.Grouping expr) {
    return parenthesize("group", expr.expression);
  }

  @Override
  public String visitLiteralExpr(Expr.Literal expr) {
    if (expr.value == null) return "nil";
    return expr.value.toString();
  }

  @Override
  public String visitUnaryExpr(Expr.Unary expr) {
    return parenthesize(expr.operator.lexeme, expr.right);
  }
    // 新增部分结束
}
```

> Literal expressions are easy—they convert the value to a string with a little check to handle Java's `null` standing in for Lox's `nil`. The other expressions have subexpressions, so they use this `parenthesize()` helper method:

字面量表达式很简单——它们将值转换为一个字符串，并通过一个小检查用Java中的`null`代替Lox中的`nil`。其他表达式有子表达式，所以它们要使用`parenthesize()`这个辅助方法：

lox/AstPrinter.java，在 visitUnaryExpr()方法后添加：

```
    private String parenthesize(String name, Expr... exprs) {
      StringBuilder builder = new StringBuilder();

      builder.append("(").append(name);
      for (Expr expr : exprs) {
        builder.append(" ");
        builder.append(expr.accept(this));
      }
      builder.append(")");

      return builder.toString();
    }
```

> It takes a name and a list of subexpressions and wraps them all up in parentheses, yielding a string like:

它接收一个名称和一组子表达式作为参数，将它们全部包装在圆括号中，并生成一个如下的字符串：

```
(+ 1 2)
```

> Note that it calls `accept()` on each subexpression and passes in itself. This is the recursive step that lets us print an entire tree.

请注意，它在每个子表达式上调用`accept()`并将自身传递进去。 这是递归步骤，可让我们打印整棵树。

> We don't have a parser yet, so it's hard to see this in action. For now, we'll hack together a little `main()` method that manually instantiates a tree and prints it.

我们还没有解析器，所以很难看到它的实际应用。现在，我们先使用一个`main()`方法来手动实例化一个树并打印它。

lox/AstPrinter.java，在 parenthesize()方法后添加：

```
  public static void main(String[] args) {
    Expr expression = new Expr.Binary(
        new Expr.Unary(
            new Token(TokenType.MINUS, "-", null, 1),
            new Expr.Literal(123)),
        new Token(TokenType.STAR, "*", null, 1),
        new Expr.Grouping(
            new Expr.Literal(45.67)));

    System.out.println(new AstPrinter().print(expression));
  }
```

> If we did everything right, it prints:

如果我们都做对了，它就会打印：

```
(* (- 123) (group 45.67))
```

> You can go ahead and delete this method. We won't need it. Also, as we add new syntax tree types, I won't bother showing the necessary visit methods for them in AstPrinter. If you want to (and you want the Java compiler to not yell at you), go ahead and add them yourself. It will come in handy in the next chapter when we start parsing Lox code into syntax trees. Or, if you don't care to maintain AstPrinter, feel free to delete it. We won't need it again.

您可以继续删除这个方法，我们后面不再需要它了。另外，当我们添加新的语法树类型时，我不会在AstPrinter中展示它们对应的visit方法。如果你想这样做(并且希望Java编译器不会报错)，那么你可以自行添加这些方法。在下一章，当我们开始将Lox代码解析为语法树时，它将会派上用场。或者，如果你不想维护AstPrinter，可以随意删除它。我们不再需要它了。

^2: 在美国，运算符优先级常缩写为**PEMDAS**，分别表示*P*arentheses(括号), *E*xponents(指数), *M*ultiplication/*D*ivision(乘除), *A*ddition/*S*ubtraction(加减)。为了便于记忆，将缩写词扩充为**Please Excuse My Dear Aunt Sally**”。 ^3: 这并不是说树是我们代码的唯一可能的表示方式。在第三部分，我们将生成字节码，这是另一种对人类不友好但更接近机器的表示方式。 ^4: 将头部限制为单个符号是上下文无关语法的定义特性。更强大的形式，如无限制文法，允许在头部和主体中都包含一系列的符号。 ^5: 是的，我们需要为定义语法的规则定义一个语法。我们也应该指定这个元语法吗?我们用什么符号来表示它?从上到下都是语言 ^6: 想象一下，我们在这里递归扩展几次`breakfast`规则，比如 "bacon with bacon with bacon with . . ." ，为了正确地完成这个字符串，我们需要在结尾处添加同等数量的 "on the side "词组。跟踪所需尾部的数量超出了正则语法的能力范围。正则语法可以表达*重复*，但它们无法*统计*有多少重复，但是这（种跟踪）对于确保字符串的`with`和`on the side`部分的数量相同是必要的。 ^7: Scheme编程语言就是这样工作的。它根本没有内置的循环功能。

相反，所有重复都用递归来表示。 ^8: 如果你愿意，可以尝试使用这个语法生成一些表达式，就像我们之前用早餐语法做的那样。生成的表达式你觉得对吗？你能让它生成任何错误的东西，比如1+/3吗？ ^9: 特别是，我们要定义一个**抽象语法树（AST）**。在**解析树**中，每一个语法生成式都成为树中的一个节点。AST省略了后面阶段不需要的生成式。 ^10: 词法单元也不是完全同质的。字面值的标记存储值，但其他类型的词素不需要该状态。我曾经见过一些扫描器使用不同的类来处理字面量和其他类型的词素，但我认为我应该把事情简单化。 ^11: 我尽量避免在代码中使用缩写，因为这会让不知道其含义的读者犯错误。但是在我所研究过的编译器中，“Expr”和“Stmt”是如此普遍，我最好现在就开始让您习惯它们。 ^12: 我从Jython和IronPython的创建者Jim Hugunin那里得到了编写语法树类脚本的想法。真正的脚本语言比Java更适合这种情况，但我尽量不向您提供太多的语言。 ^13: 这不是世界上最优雅的字符串操作代码，但也很好。它只在我们给它的类定义集上运行。稳健性不是优先考虑的问题。 ^14: 附录II包含了在我们完成jlox的实现并定义了它的所有语法树节点之后，这个脚本生成的代码。 ^15: 这就是Erich Gamma等人在《设计模式:可重用的面向对象软件的元素》一书中所谓的解释器模式。 ^16: ML，是元语言(metalanguage)的简称，它是由Robin Milner和他的朋友们创建的，是伟大的编程语言家族的主要分支之一。它的子程序包括SML、Caml、OCaml、Haskell和F#。甚至Scala、Rust和Swift都有很强的相似性。就像Lisp一样，它也是那种充满了好点子的语言之一，即使在40多年后的今天，语言设计者仍然在重新发现它们。 ^17: 诸如Common Lisp的CLOS， Dylan和Julia这样的支持多方法(多分派)的语言都能轻松添加新类型和操作。它们通常牺牲的是静态类型检查或单独编译。 ^18: 在设计模式中，这两种方法的名字都叫`visit()`，很容易混淆，需要依赖重载来区分不同方法。这也导致一些读者认为正确的visit方法是在运行时根据其参数类型选择的。事实并非如此。与重写不同，重载是在编译时静态分派的。为每个方法使用不同的名称使分派更加明显，同时还向您展示了如何在不支持重载的语言中应用此模式。

CHALLENGES

习题

> 1、Earlier, I said that the `|`, `*`, and `+` forms we added to our grammar metasyntax were just syntactic sugar. Take this grammar:
>
> ```
> expr → expr ( "(" ( expr ( "," expr )* )? ")" | "." IDENTIFIER )+
>      | IDENTIFIER
>      | NUMBER
> ```
>
> Produce a grammar that matches the same language but does not use any of that notational sugar.
>
> *Bonus:* What kind of expression does this bit of grammar encode?

1、之前我说过，我们在语法元语法中添加的`|`、`*`、`+`等形式只是语法糖。以这个语法为例:

```
expr → expr ( "(" ( expr ( "," expr )* )? ")" | "." IDENTIFIER )+
     | IDENTIFIER
     | NUMBER
```

生成一个与同一语言相匹配的语法，但不要使用任何语法糖。

附加题：这一点语法表示了什么样的表达式？

> 2、The Visitor pattern lets you emulate the functional style in an object-oriented language. Devise a complementary pattern for a functional language. It should let you bundle all of the operations on one type together and let you define new types easily.
>
> (SML or Haskell would be ideal for this exercise, but Scheme or another Lisp works as well.)

2、Visitor 模式让你可以在面向对象的语言中模仿函数式。为函数式语言设计一个互补的模式，该模式让你可以将一个类型上的所有操作捆绑在一起，并轻松扩展新的类型。

(SML或Haskell是这个练习的理想选择，但Scheme或其它Lisp方言也可以。)

> 3、In Reverse Polish Notation (RPN), the operands to an arithmetic operator are both placed before the operator, so `1 + 2` becomes `1 2 +`. Evaluation proceeds from left to right. Numbers are pushed onto an implicit stack. An arithmetic operator pops the top two numbers, performs the operation, and pushes the result. Thus, this:
>
> ```
> (1 + 2) * (4 - 3)
> ```
>
> in RPN becomes:
>
> ```
> 1 2 + 4 3 - *
> ```
>
> Define a visitor class for our syntax tree classes that takes an expression, converts it to RPN, and returns the resulting string.

3、在逆波兰表达式(RPN)中，算术运算符的操作数都放在运算符之前，所以`1 + 2`变成了`1 2 +`。计算时从左到右进行，操作数被压入隐式栈。算术运算符弹出前两个数字，执行运算，并将结果推入栈中。因此,

```
(1 + 2) * (4 - 3)
```

在RPN中变为了

```
1 2 + 4 3 - *
```

为我们的语法树类定义一个Vistor类，该类接受一个表达式，将其转换为RPN，并返回结果字符串。

6.Parsing Expressions 解析表达式

> *Grammar, which knows how to control even kings.*
>
> ——Molière

语法，它甚至知道如何控制国王。（莫里哀）

> This chapter marks the first major milestone of the book. Many of us have cobbled together a mishmash of regular expressions and substring operations to extract some sense out of a pile of text. The code was probably riddled with bugs and a beast to maintain. Writing a *real* parser—one with decent error handling, a coherent internal structure, and the ability to robustly chew through a sophisticated syntax—is considered a rare, impressive skill. In this chapter, you will attain it.

本章是本书的第一个重要里程碑。我们中的许多人都曾将正则表达式和字符串操作糅合在一起，以便从一堆文本中提取一些信息。这些代码可能充满了错误，而且很难维护。编写一个真正的解析器^1——具有良好的错误处理、一致的内部结构和能够健壮地分析复杂语法的能力——被认为是一种罕见的、令人印象深刻的技能。在这一章中，你将获得这种技能。

> It's easier than you think, partially because we front-loaded a lot of the hard work in the last chapter. You already know your way around a formal grammar. You're familiar with syntax trees, and we have some Java classes to represent them. The only remaining piece is parsing—transmogrifying a sequence of tokens into one of those syntax trees.

这比想象中要简单，部分是因为我们在上一章中提前完成了很多困难的工作。你已经对形式化语法了如指掌，也熟悉了语法树，而且我们有一些Java类来表示它们。唯一剩下的部分是解析——将一个标记序列转换成这些语法树中的一个。

> Some CS textbooks make a big deal out of parsers. In the '60s, computer scientists—understandably tired of programming in assembly language—started designing more sophisticated, human-friendly languages like Fortran and ALGOL. Alas, they weren't very *machine*-friendly for the primitive computers of the time.

一些CS教科书在解析器上大做文章。在60年代，计算机科学家——他们理所当然地厌倦了用汇编语言编程——开始设计更复杂的、对人类友好的语言，比如Fortran和ALGOL^2。唉，对于当时原始的计算机来说，这些语言对机器并不友好。

> These pioneers designed languages that they honestly weren't even sure how to write compilers for, and then did groundbreaking work inventing parsing and compiling techniques that could handle these new, big languages on those old, tiny machines.

这些先驱们设计了一些语言，说实话，他们甚至不知道如何编写编译器。然后他们做了开创性的工作，发明了解析和编译技术，可以在那些老旧、小型的机器上处理这些新的、大型的语言。

> Classic compiler books read like fawning hagiographies of these heroes and their tools. The cover of *Compilers: Principles, Techniques, and Tools* literally has a dragon labeled "complexity of compiler design" being slain by a knight bearing a sword and shield branded "LALR parser generator" and "syntax directed translation". They laid it on thick.

经典的编译书读起来就像是对这些英雄和他们的工具的吹捧传记。《编译器:原理、技术和工具》（*Compilers: Principles, Techniques, and Tools*）的封面上有一条标记着"编译器设计复杂性"的龙，被一个手持剑和盾的骑士杀死，剑和盾上标记着"LALR解析器生成器"和"语法制导翻译"。他们在过分吹捧。

> A little self-congratulation is well-deserved, but the truth is you don't need to know most of that stuff to bang out a high quality parser for a modern machine. As always, I encourage you to broaden your education and take it in later, but this book omits the trophy case.

稍微的自我祝贺是当之无愧的，但事实是，你不需要知道其中的大部分知识，就可以为现代机器制作出高质量的解析器。一如既往，我鼓励你先扩大学习范围，以后再慢慢接受它，但这本书省略了奖杯箱。

6.1 Ambiguity and the Parsing Game

6.1 歧义与解析游戏

> In the last chapter, I said you can “play” a context-free grammar like a game in order to *generate* strings. Parsers play that game in reverse. Given a string—a series of tokens—we map those tokens to terminals in the grammar to figure out which rules could have generated that string.

在上一章中，我说过你可以像“玩”游戏一样使用上下文无关的语法来*生成*字符串。解析器则以相反的方式玩游戏。给定一个字符串(一系列语法标记)，我们将这些标记映射到语法中的终止符，以确定哪些规则可能生成该字符串。

> The “could have” part is interesting. It’s entirely possible to create a grammar that is *ambiguous*, where different choices of productions can lead to the same string. When you’re using the grammar to *generate* strings, that doesn’t matter much. Once you have the string, who cares how you got to it?

"可能产生 "这部分很有意思。我们完全有可能创建一个*模棱两可*的语法，在这个语法中，不同的生成式可能会得到同一个字符串。当你使用该语法来*生成*字符串时，这一点不太重要。一旦你有了字符串，谁还会在乎你是怎么得到它的呢？

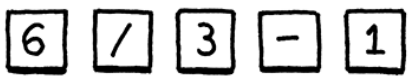
> When parsing, ambiguity means the parser may misunderstand the user’s code. As we parse, we aren’t just determining if the string is valid Lox code, we’re also tracking which rules match which parts of it so that we know what part of the language each token belongs to. Here’s the Lox expression grammar we put together in the last chapter:

但是在解析时，歧义意味着解析器可能会误解用户的代码。当我们进行解析时，我们不仅要确定字符串是不是有效的Lox代码，还要记录哪些规则与代码的哪些部分相匹配，以便我们知道每个标记属于语言的哪一部分。下面是我们在上一章整理的Lox表达式语法：

```
expression     → literal
               | unary
               | binary
               | grouping ;

literal        → NUMBER | STRING | "true" | "false" | "nil" ;
grouping       → "(" expression ")" ;
unary          → ( "-" | "!" ) expression ;
binary         → expression operator expression ;
operator       → "==" | "!=" | "<" | "<=" | ">" | ">="
               | "+"  | "-"  | "*" | "/" ;
```

> This is a valid string in that grammar:

下面是一个满足语法的有效字符串：



> But there are two ways we could have generated it. One way is:

但是，有两种方式可以生成该字符串。其一是：

> 1. Starting at `expression`, pick `binary`.
> 2. For the left-hand `expression`, pick `NUMBER`, and use `6`.
> 3. For the operator, pick `"/"`.
> 4. For the right-hand `expression`, pick `binary` again.
> 5. In that nested `binary` expression, pick `3 - 1`.

1. 从`expression`开始，选择`binary`。
2. 对于左边的`expression`，选择`NUMBER`，并且使用`6`。
3. 对于操作符，选择`/`。
4. 对于右边的`expression`，再次选择`binary`。
5. 在内层的`binary` 表达式中，选择`3-1`。

> Another is:

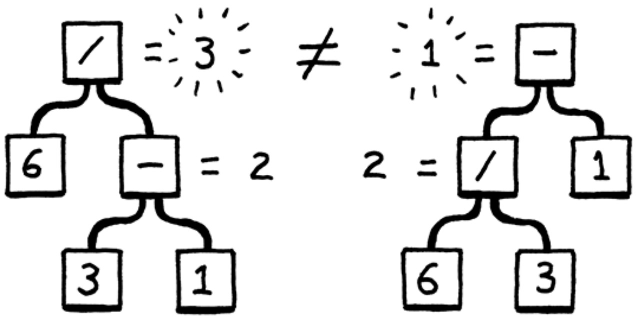
其二是：

> 1. Starting at `expression`, pick `binary`.
> 2. For the left-hand `expression`, pick `binary` again.
> 3. In that nested `binary` expression, pick `6 / 3`.
> 4. Back at the outer `binary`, for the operator, pick `"-"`.
> 5. For the right-hand `expression`, pick `NUMBER`, and use `1`.

1. 从`expression`开始，选择`binary`。
2. 对于左边的`expression`，再次选择`binary`。
3. 在内层的`binary` 表达式中，选择`6/3`。
4. 返回外层的`binary` ，对于操作符，选择`-`。
5. 对于右边的`expression`，选择`NUMBER`，并且使用`1`。

> Those produce the same *strings*, but not the same *syntax trees*:

它们产生相同的字符串，但对应的是不同的 *语法树*：



> In other words, the grammar allows seeing the expression as `(6 / 3) - 1` or `6 / (3 - 1)`. The `binary` rule lets operands nest any which way you want. That in turn affects the result of evaluating the parsed tree. The way mathematicians have addressed this ambiguity since blackboards were first invented is by defining rules for precedence and associativity.

换句话说，这个语法可以将该表达式看作是 `(6 / 3) - 1`或`6 / (3 - 1)`。`binary` 规则运行操作数以任意方式嵌套，这反过来又会影响解析数的计算结果。自从黑板被发明以来，数学家们解决这种模糊性的方法就是定义优先级和结合性规则。

- > **Precedence** determines which operator is evaluated first in an expression containing a mixture of different operators. Precedence rules tell us that we evaluate the `/` before the `-` in the above example. Operators with higher precedence are evaluated before operators with lower precedence. Equivalently, higher precedence operators are said to “bind tighter”.

- **优先级**决定了在一个包含不同运算符的混合表达式中，哪个运算符先被执行^3。优先级规则告诉我们，在上面的例子中，我们在`-`之前先计算`/`。优先级较高的运算符在优先级较低的运算符之前计算。同样，优先级较高的运算符被称为 "更严格的绑定"。

- > **Associativity** determines which operator is evaluated first in a series of the *same* operator. When an operator is **left-associative** (think “left-to-right”), operators on the left evaluate before those on the right. Since `-` is left-associative, this expression:

- **结合性**决定在一系列相同操作符中先计算哪个操作符。如果一个操作符是**左结合**的(可以认为是“从左到右”)时，左边的操作符在右边的操作符之前计算。因为`-`是左结合的，下面的表达式：

  ```
  5 - 3 - 1
  ```

  > is equivalent to:

  等价于：

  ```
  (5 - 3) - 1
  ```

  > Assignment, on the other hand, is **right-associative**. This:

  另一方面，赋值是**右结合**的。如：

  ```
  a = b = c
  ```

  > is equivalent to:

  等价于：

  ```
  a = (b = c)
  ```

> Without well-defined precedence and associativity, an expression that uses multiple operators is ambiguous—it can be parsed into different syntax trees, which could in turn evaluate to different results. We’ll fix that in Lox by applying the same precedence rules as C, going from lowest to highest.

如果没有明确定义的优先级和结合性，使用多个运算符的表达式可能就会变得有歧义——它可以被解析为不同的语法树，而这些语法树又可能会计算出不同的结果。我们在Lox中会解决这个问题，使用与C语言相同的优先级规则，从低到高分别是：

Name	Operators	Associates
Equality 等于	`==` `!=`	Left 左结合
Comparison 比较	`>` `>=` `<` `<=`	Left 左结合
Term 加减运算	`-` `+`	Left 左结合
Factor 乘除运算	`/` `*`	Left 左结合
Unary 一元运算符	`!` `-`	Right 右结合

> Right now, the grammar stuffs all expression types into a single `expression` rule. That same rule is used as the non-terminal for operands, which lets the grammar accept any kind of expression as a subexpression, regardless of whether the precedence rules allow it.

现在，该语法将所有表达式类型都添加到一个 `expression`规则中。这条规则同样作用于操作数中的非终止符，这使得语法中可以接受任何类型的表达式作为子表达式，而不管优先级规则是否允许。

> We fix that by stratifying the grammar. We define a separate rule for each precedence level.

我们通过对语法进行分层来解决这个问题。我们为每个优先级定义一个单独的规则^4。

```
expression     → ...
equality       → ...
comparison     → ...
term           → ...
factor         → ...
unary          → ...
primary        → ...
```

> Each rule here only matches expressions at its precedence level or higher. For example, `unary` matches a unary expression like `!negated` or a primary expression like `1234`. And `term` can match `1 + 2` but also `3 * 4 / 5`. The final `primary` rule covers the highest-precedence forms—literals and parenthesized expressions.

此处的每个规则仅匹配其当前优先级或更高优先级的表达式。 例如，`unary` 匹配一元表达式（如 `!negated`）或主表达式（如`1234`）。`term`可以匹配`1 + 2`，但也可以匹配`3 * 4 /5`。最后的`primary` 规则涵盖优先级最高的形式——字面量和括号表达式。

> We just need to fill in the productions for each of those rules. We'll do the easy ones first. The top `expression` rule matches any expression at any precedence level. Since `equality` has the lowest precedence, if we match that, then it covers everything.

我们只需要填写每条规则的生成式。我们先从简单的开始。顶级的`expression` 规则可以匹配任何优先级的表达式。由于`equality`的优先级最低，只要我们匹配了它，就涵盖了一切^5。

```
expression     → equality
```

> Over at the other end of the precedence table, a primary expression contains all the literals and grouping expressions.

在优先级表的另一端，`primary`表达式包括所有的字面量和分组表达式。

```
primary        → NUMBER | STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

> A unary expression starts with a unary operator followed by the operand. Since unary operators can nest—`!!true` is a valid if weird expression—the operand can itself be a unary operator. A recursive rule handles that nicely.

一元表达式以一元操作符开头，后跟操作数。因为一元操作符可以嵌套——`!!true`虽奇怪也是可用的表达式——这个操作数本身可以是一个一元表达式。递归规则可以很好地解决这个问题。

```
unary          → ( "!" | "-" ) unary ;
```

> But this rule has a problem. It never terminates.

但是这条规则有一个问题，它永远不会终止。

> Remember, each rule needs to match expressions at that precedence level *or higher*, so we also need to let this match a primary expression.

请记住，每个规则都需要匹配该优先级或更高优先级的表达式，因此我们还需要使其与主表达式匹配。

```
unary          → ( "!" | "-" ) unary
               | primary ;
```

> That works.

这样就可以了。

> The remaining rules are all binary operators. We'll start with the rule for multiplication and division. Here's a first try:

剩下的规则就是二元运算符。我们先从乘法和除法的规则开始。下面是第一次尝试：

```
factor         → factor ( "/" | "*" ) unary
               | unary ;
```

> The rule recurses to match the left operand. That enables the rule to match a series of multiplication and division expressions like `1 * 2 / 3`. Putting the recursive production on the left side and `unary` on the right makes the rule left-associative and unambiguous.

该规则递归匹配左操作数，这样一来，就可以匹配一系列乘法和除法表达式，例如 `1 * 2 / 3`。将递归生成式放在左侧并将`unary` 放在右侧，可以使该规则具有左关联性和明确性^6。

> All of this is correct, but the fact that the first symbol in the body of the rule is the same as the head of the rule means this production is **left-recursive**. Some parsing techniques, including the one we're going to use, have trouble with left recursion. (Recursion elsewhere, like we have in `unary` and the indirect recursion for grouping in `primary` are not a problem.)

所有这些都是正确的，但规则主体中的第一个符号与规则头部相同意味着这个生成式是**左递归**的。一些解析技术，包括我们将要使用的解析技术，在处理左递归时会遇到问题。(其他地方的递归，比如在`unary`中，以及在`primary`分组中的间接递归都不是问题。)

> There are many grammars you can define that match the same language. The choice for how to model a particular language is partially a matter of taste and partially a pragmatic one. This rule is correct, but not optimal for how we intend to parse it. Instead of a left recursive rule, we'll use a different one.

你可以定义很多符合同一种语言的语法。如何对某一特定语言进行建模，一部分是品味问题，一部分是实用主义问题。这个规则是正确的，但对于我们后续的解析来说它并不是最优的。我们将使用不同的规则来代替左递归规则。

```
factor         → unary ( ( "/" | "*" ) unary )* ;
```

> We define a factor expression as a flat *sequence* of multiplications and divisions. This matches the same syntax as the previous rule, but better mirrors the code we'll write to parse Lox. We use the same structure for all of the other binary operator precedence levels, giving us this complete expression grammar:

我们将因子表达式定义为乘法和除法的扁平*序列*。这与前面的规则语法相同，但更好地反映了我们将编写的解析Lox的代码。我们对其它二元运算符的优先级使用相同的结构，从而得到下面这个完整的表达式语法：

```
expression     → equality ;
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;
unary          → ( "!" | "-" ) unary
               | primary ;
primary        → NUMBER | STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

This grammar is more complex than the one we had before, but in return we have eliminated the previous one's ambiguity. It's just what we need to make a parser.

这个语法比我们以前的那个更复杂，但反过来我们也消除了前一个语法定义中的歧义。这正是我们制作解析器时所需要的。

6.2 Recursive Descent Parsing

6.2 递归下降分析

> There is a whole pack of parsing techniques whose names are mostly combinations of “L” and “R”—LL(k), LR(1), LALR—along with more exotic beasts like parser combinators, Earley parsers, the shunting yard algorithm, and packrat parsing. For our first interpreter, one technique is more than sufficient: **recursive descent**.

现在有一大堆解析技术，它们的名字大多是 "L "和 "R "的组合——LL(k)、LR(1)、LALR——还有更多的异类，比如解析器组合子、Earley parsers、分流码算法和packrat解析。对于我们的第一个解释器来说，一种技术已经足够了：**递归下降**。

> Recursive descent is the simplest way to build a parser, and doesn’t require using complex parser generator tools like Yacc, Bison or ANTLR. All you need is straightforward handwritten code. Don’t be fooled by its simplicity, though. Recursive descent parsers are fast, robust, and can support sophisticated error handling. In fact, GCC, V8 (the JavaScript VM in Chrome), Roslyn (the C# compiler written in C#) and many other heavyweight production language implementations use recursive descent. It rocks.

递归下降是构建解析器最简单的方法，不需要使用复杂的解析器生成工具，如Yacc、Bison或ANTLR。你只需要直接手写代码。但是不要被它的简单性所欺骗，递归下降解析器速度快、健壮，并且可以支持复杂的错误处理。事实上，GCC、V8 (Chrome中的JavaScript VM)、Roslyn(用c#编写的c#编译器)和许多其他重量级产品语言实现都使用了递归下降技术。它很好用。

> Recursive descent is considered a **top-down parser** because it starts from the top or outermost grammar rule (here `expression`) and works its way down into the nested subexpressions before finally reaching the leaves of the syntax tree. This is in contrast with bottom-up parsers like LR that start with primary expressions and compose them into larger and larger chunks of syntax.

递归下降被认为是一种**自顶向下解析器**，因为它从最顶部或最外层的语法规则(这里是`expression`)开始，一直向下进入嵌套子表达式，最后到达语法树的叶子。这与LR等自下而上的解析器形成鲜明对比，后者从初级表达式(primary)开始，将其组成越来越大的语法块^7。

> A recursive descent parser is a literal translation of the grammar’s rules straight into imperative code. Each rule becomes a function. The body of the rule translates to code roughly like:

递归下降解析器是一种将语法规则直接翻译成命令式代码的文本翻译器。每个规则都会变成一个函数，规则主体翻译成代码大致是这样的：

Grammar notation	**Code representation**
Terminal	Code to match and consume a token 匹配并消费一个语法标记
Nonterminal	Call to that rule’s function 调用规则对应的函数
`|`	`if` or `switch` statement if或switch语句

Grammar notation	Code representation
`*` or `+`	`while` or `for` loop while或for循环
`?`	`if` statement if语句

> The descent is described as “recursive” because when a grammar rule refers to itself—directly or indirectly—that translates to a recursive function call.

下降被“递归”修饰是因为，如果一个规则引用自身（直接或间接）就会变为递归的函数调用。

> ### 6.2.1 The parser class

6.2.1 Parser类

> Each grammar rule becomes a method inside this new class:

每个语法规则都成为新类中的一个方法:

lox/Parser.java，创建新文件：

```java
package com.craftinginterpreters.lox;

import java.util.List;

import static com.craftinginterpreters.lox.TokenType.*;

class Parser {
  private final List<Token> tokens;
  private int current = 0;

  Parser(List<Token> tokens) {
    this.tokens = tokens;
  }
}
```

> Like the scanner, the parser consumes a flat input sequence, only now we’re reading tokens instead of characters. We store the list of tokens and use `current` to point to the next token eagerly waiting to be parsed.

与扫描器一样，解析器也是消费一个扁平的输入序列，只是这次我们要读取的是语法标记而不是字符。我们会保存标记列表并使用`current`指向待解析的下一个标记。

> We’re going to run straight through the expression grammar now and translate each rule to Java code. The first rule, `expression`, simply expands to the `equality` rule, so that’s straightforward.

我们现在要直接执行表达式语法，并将每一条规则翻译为Java代码。第一条规则`expression`，简单地展开为`equality`规则，所以很直接：

lox/Parser.java，在 Parser()方法后添加：

```
  private Expr expression() {
    return equality();
  }
```

> Each method for parsing a grammar rule produces a syntax tree for that rule and returns it to the caller. When the body of the rule contains a nonterminal—a reference to another rule—we call that other rule's method.

每个解析语法规则的方法都会生成该规则对应的语法树，并将其返回给调用者。当规则主体中包含一个非终止符——对另一条规则的引用时，我们就会调用另一条规则对应的方法^8。

> The rule for equality is a little more complex.

`equality`规则有一点复杂：

```
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
```

> In Java, that becomes:

在Java中，这会变成：

lox/Parser.java，在 expression()后面添加：

```
  private Expr equality() {
    Expr expr = comparison();

    while (match(BANG_EQUAL, EQUAL_EQUAL)) {
      Token operator = previous();
      Expr right = comparison();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

> Let's step through it. The first `comparison` nonterminal in the body translates to the first call to `comparison()` in the method. We take that result and store it in a local variable.

让我们一步步来。规则体中的第一个 `comparison` 非终止符变成了方法中对 `comparison()` 的第一次调用。我们获取结果并将其保存在一个局部变量中。

> Then, the `( ... )*` loop in the rule maps to a `while` loop. We need to know when to exit that loop. We can see that inside the rule, we must first find either a `!=` or `==` token. So, if we *don't* see one of those, we must be done with the sequence of equality operators. We express that check using a handy `match()` method.

然后，规则中的`( ... )*`循环映射为一个`while`循环。我们需要知道何时退出这个循环。可以看到，在规则体中，我们必须先找到一个 `!=` 或`==`标记。因此，如果我们*没有*看到其中任一标记，我们必须结束相等(不相等)运算符的序列。我们使用一个方便的`match()`方法来执行这个检查。

lox/Parser.java，在 equality()方法后添加：

```
  private boolean match(TokenType... types) {
    for (TokenType type : types) {
      if (check(type)) {
        advance();
        return true;
      }
    }

    return false;
  }
```

> This checks to see if the current token has any of the given types. If so, it consumes the token and returns `true`. Otherwise, it returns `false` and leaves the current token alone. The `match()` method is defined in terms of two more fundamental operations.

这个检查会判断当前的标记是否属于给定的类型之一。如果是，则消费该标记并返回`true`；否则，就返回`false`并保留当前标记。`match()`方法是由两个更基本的操作来定义的。

> The `check()` method returns `true` if the current token is of the given type. Unlike `match()`, it never consumes the token, it only looks at it.

如果当前标记属于给定类型，则`check()`方法返回`true`。与`match()`不同的是，它从不消费标记，只是读取。

lox/Parser.java，在 match()方法后添加：

```
  private boolean check(TokenType type) {
    if (isAtEnd()) return false;
    return peek().type == type;
  }
```

> The `advance()` method consumes the current token and returns it, similar to how our scanner's corresponding method crawled through characters.

`advance()`方法会消费当前的标记并返回它，类似于扫描器中对应方法处理字符的方式。

lox/Parser.java，在 check()方法后添加：

```
  private Token advance() {
    if (!isAtEnd()) current++;
    return previous();
  }
```

> These methods bottom out on the last handful of primitive operations.

这些方法最后都归结于几个基本操作。

lox/Parser.java，在 advance()后添加：

```
  private boolean isAtEnd() {
    return peek().type == EOF;
  }

  private Token peek() {
    return tokens.get(current);
  }

  private Token previous() {
    return tokens.get(current - 1);
  }
```

> `isAtEnd()` checks if we've run out of tokens to parse. `peek()` returns the current token we have yet to consume, and `previous()` returns the most recently consumed token. The latter makes it easier to use `match()` and then access the just-matched token.

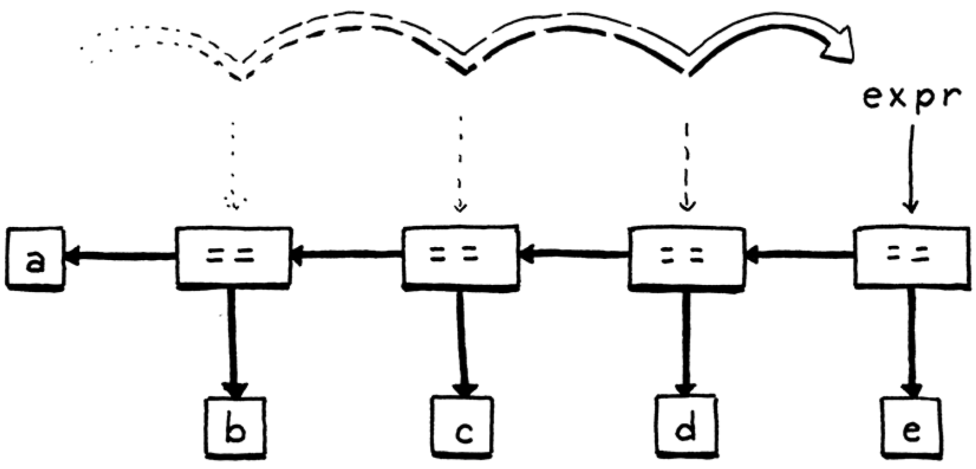
`isAtEnd()`检查我们是否处理完了待解析的标记。`peek()`方法返回我们还未消费的当前标记，而`previous()`会返回最近消费的标记。后者让我们更容易使用`match()`，然后访问刚刚匹配的标记。

> That's most of the parsing infrastructure we need. Where were we? Right, so if we are inside the `while` loop in `equality()`, then we know we have found a `!=` or `==` operator and must be parsing an equality expression.

这就是我们需要的大部分解析基本工具。我们说到哪里了？对，如果我们在`equality()`的`while`循环中，也就能知道我们已经找到了一个`!=`或`==`操作符，并且一定是在解析一个等式表达式。

> We grab the matched operator token so we can track which kind of equality expression we have. Then we call `comparison()` again to parse the right-hand operand. We combine the operator and its two operands into a new `Expr.Binary` syntax tree node, and then loop around. For each iteration, we store the resulting expression back in the same `expr` local variable. As we zip through a sequence of equality expressions, that creates a left-associative nested tree of binary operator nodes.

我们获取到匹配的操作符标记，这样就可以知道我们要处理哪一类等式表达式。之后，我们再次调用`comparison()`解析右边的操作数。我们将操作符和它的两个操作数组合成一个新的`Expr.Binary`语法树节点，然后开始循环。对于每一次迭代，我们都将结果表达式存储在同一个`expr`局部变量中。在对等式表达式序列进行压缩时，会创建一个由二元操作符节点组成的左结合嵌套树^9。



> The parser falls out of the loop once it hits a token that's not an equality operator. Finally, it returns the expression. Note that if the parser never encounters an equality operator, then it never enters the loop. In that case, the `equality()` method effectively calls and returns `comparison()`. In that way, this method matches an equality operator *or anything of higher precedence*.

一旦解析器遇到一个不是等式操作符的标记，就会退出循环。最后，它会返回对应的表达式。请注意，如果解析器从未遇到过等式操作符，它就永远不会进入循环。在这种情况下，`equality()`方法有效地调用并返回`comparison()`。这样一来，这个方法就会匹配一个等式运算符或*任何更高优先级的表达式*。

> Moving on to the next rule . . .

继续看下一个规则。

```
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
```

> Translated to Java:

翻译成Java：

lox/Parser.java，在 equality()方法后添加：

```
  private Expr comparison() {
    Expr expr = term();

    while (match(GREATER, GREATER_EQUAL, LESS, LESS_EQUAL)) {
      Token operator = previous();
      Expr right = term();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

> The grammar rule is virtually identical to `equality` and so is the corresponding code. The only differences are the token types for the operators we match, and the method we call for the operands—

> now `term()` instead of `comparison()`. The remaining two binary operator rules follow the same pattern.

语法规则与`equality`几乎完全相同，相应的代码也是如此。唯一的区别是匹配的操作符的标记类型，而且现在获取操作数时调用的方法是`term()`而不是`comparison()`。其余两个二元操作符规则遵循相同的模式。

> In order of precedence, first addition and subtraction:

按照优先级顺序，先做加减法：

lox/Parser.java，在 comparison()方法后添加：

```
  private Expr term() {
    Expr expr = factor();

    while (match(MINUS, PLUS)) {
      Token operator = previous();
      Expr right = factor();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

> And finally, multiplication and division:

最后，是乘除法：

lox/Parser.java，在 term()方法后面添加：

```
  private Expr factor() {
    Expr expr = unary();

    while (match(SLASH, STAR)) {
      Token operator = previous();
      Expr right = unary();
      expr = new Expr.Binary(expr, operator, right);
    }

    return expr;
  }
```

> That's all of the binary operators, parsed with the correct precedence and associativity. We're crawling up the precedence hierarchy and now we've reached the unary operators.

这就是所有的二元运算符，已经按照正确的优先级和结合性进行了解析。接下来，按照优先级层级，我们要处理一元运算符了。

```
unary          → ( "!" | "-" ) unary
               | primary ;
```

> The code for this is a little different.

该规则对应的代码有些不同。

lox/Parser.java，在 factor()方法后添加：

```
  private Expr unary() {
    if (match(BANG, MINUS)) {
      Token operator = previous();
      Expr right = unary();
      return new Expr.Unary(operator, right);
    }

    return primary();
  }
```

> Again, we look at the current token to see how to parse. If it's a `!` or `-`, we must have a unary expression. In that case, we grab the token and then recursively call `unary()` again to parse the operand. Wrap that all up in a unary expression syntax tree and we're done.

同样的，我们先检查当前的标记以确认要如何进行解析^10。如果是`!`或`-`，我们一定有一个一元表达式。在这种情况下，我们使用当前的标记递归调用`unary()`来解析操作数。将所有这些都包装到一元表达式语法树中，我们就完成了。

> Otherwise, we must have reached the highest level of precedence, primary expressions.

否则，我们就达到了最高级别的优先级，即基本表达式。

```
primary        → NUMBER | STRING | "true" | "false" | "nil"
               | "(" expression ")" ;
```

> Most of the cases for the rule are single terminals, so parsing is straightforward.

该规则中大部分都是终止符，可以直接进行解析。

lox/Parser.java，在 unary()方法后添加：

```
  private Expr primary() {
    if (match(FALSE)) return new Expr.Literal(false);
    if (match(TRUE)) return new Expr.Literal(true);
    if (match(NIL)) return new Expr.Literal(null);

    if (match(NUMBER, STRING)) {
      return new Expr.Literal(previous().literal);
    }

    if (match(LEFT_PAREN)) {
      Expr expr = expression();
      consume(RIGHT_PAREN, "Expect ')' after expression.");
      return new Expr.Grouping(expr);
    }
  }
```

The interesting branch is the one for handling parentheses. After we match an opening `(` and parse the expression inside it, we *must* find a `)` token. If we don't, that's an error.

有趣的一点是处理括号的分支。当我们匹配了一个开头`(`并解析了里面的表达式后，我们必须找到一个`)`标记。如果没有找到，那就是一个错误。

6.3 Syntax Errors

6.3 语法错误

> A parser really has two jobs:

解析器实际上有两项工作：

1. > Given a valid sequence of tokens, produce a corresponding syntax tree.

   给定一个有效的标记序列，生成相应的语法树。

2. > Given an *invalid* sequence of tokens, detect any errors and tell the user about their mistakes.

   给定一个*无效*的标记序列，检测错误并告知用户。

> Don't underestimate how important the second job is! In modern IDEs and editors, the parser is constantly reparsing code—often while the user is still editing it—in order to syntax highlight and support things like auto-complete. That means it will encounter code in incomplete, half-wrong states *all the time.*

不要低估第二项工作的重要性！在现代的IDE和编辑器中，为了语法高亮显示和支持自动补齐等功能，当用户还在编辑代码时，解析器就会不断地重新解析代码。这也意味着解析器*总是*会遇到不完整的、半错误状态的代码。

> When the user doesn't realize the syntax is wrong, it is up to the parser to help guide them back onto the right path. The way it reports errors is a large part of your language's user interface. Good syntax error handling is hard. By definition, the code isn't in a well-defined state, so there's no infallible way to know what the user *meant* to write. The parser can't read your mind.

当用户没有意识到语法错误时，解析器要帮助引导他们回到正确的道路上。在你的语言的人机交互中，错误反馈占据了很大的比重。良好的语法错误处理是很难的。根据定义，代码并不是处于良好定义的状态，所以没有可靠的方法能够知道用户*想要*写什么。解析器无法读懂你的思想。

> There are a couple of hard requirements for when the parser runs into a syntax error. A parser must:

当解析器遇到语法错误时，有几个硬性要求。解析器必须能够：

- > **Detect and report the error.** If it doesn't detect the error and passes the resulting malformed syntax tree on to the interpreter, all manner of horrors may be summoned.

  检测并报告错误。如果它没有检测到错误，并将由此产生的畸形语法树传递给解释器，就会出现各种可怕的情况。

- > **Avoid crashing or hanging.** Syntax errors are a fact of life, and language tools have to be robust in the face of them. Segfaulting or getting stuck in an infinite loop isn't allowed. While the source may not be valid *code*, it's still a valid *input to the parser* because users use the parser to learn what syntax is allowed.

  避免崩溃或挂起。语法错误是生活中不可避免的事实，面对语法错误，语言工具必须非常健壮。段错误或陷入无限循环是不允许的。虽然源代码可能不是有效的*代码*，但它仍然是*解析器的有效输入*，因为用户使用解析器来了解什么是允许的语法。

> Those are the table stakes if you want to get in the parser game at all, but you really want to raise the ante beyond that. A decent parser should:

如果你想参与到解析器的游戏中来，这些就是桌面的筹码，但你真的想提高赌注，除了这些。一个像样的解析器还应该：

- > **Be fast.** Computers are thousands of times faster than they were when parser technology was first invented. The days of needing to optimize your parser so that it could get through an entire source file during a coffee break are over. But programmer expectations have risen as quickly, if not faster. They expect their editors to reparse files in milliseconds after every keystroke.

  要快。计算机的速度比最初发明解析器技术时快了几千倍。那种需要优化解析器，以便它能在喝咖啡的时候处理完整个源文件的日子已经一去不复返了。但是程序员的期望值也上升得同样快，甚至更快。他们希望他们的编辑器能在每次击键后的几毫秒内回复文件。

- > **Report as many distinct errors as there are.** Aborting after the first error is easy to implement, but it's annoying for users if every time they fix what they think is the one error in a file, a new one appears. They want to see them all.

  尽可能多地报告出不同的错误。在第一个错误后中止是很容易实现的，但是如果每次当用户修复文件中的一个错误时，又出现了另一个新的错误，这对用户来说是很烦人的。他们希望一次看到所有的错误。

- > **Minimize *cascaded* errors.** Once a single error is found, the parser no longer really knows what's going on. It tries to get itself back on track and keep going, but if it gets confused, it may report a slew of ghost errors that don't indicate other real problems in the code. When the first error is fixed, those phantoms disappear, because they reflect only the parser's own confusion. Cascaded errors are annoying because they can scare the user into thinking their code is in a worse state than it is.

  最小化*级联*错误。一旦发现一个错误，解析器就不再能知道发生了什么。它会试图让自己回到正轨并继续工作，但如果它感到混乱，它可能会报告大量的幽灵错误，而这些错误并不表明代码中存在其它问题。当第一个错误被修正后，这些幽灵错误就消失了，因为它们只反映了解析器自身的混乱。级联错误很烦人，因为它们会让用户害怕，让用户认为自己的代码比实际情况更糟糕。

> The last two points are in tension. We want to report as many separate errors as we can, but we don't want to report ones that are merely side effects of an earlier one.

最后两点是相互矛盾的。我们希望尽可能多地报告单独的错误，但我们不想报告那些只是由早期错误的副作用导致的错误。

> The way a parser responds to an error and keeps going to look for later errors is called **error recovery**. This was a hot research topic in the '60s. Back then, you'd hand a stack of punch cards to the secretary and come back the next day to see if the compiler succeeded. With an iteration loop that slow, you *really* wanted to find every single error in your code in one pass.

解析器对一个错误做出反应，并继续去寻找后面的错误的方式叫做**错误恢复**。这在60年代是一个热门的研究课题。那时，你需要把一叠打孔卡交给秘书，第二天再来看看编译器是否成功。在迭代循环如此缓慢的情况下，你真的会想在一次执行中找到代码中的每个错误。

> Today, when parsers complete before you've even finished typing, it's less of an issue. Simple, fast error recovery is fine.

如今，解析器在您甚至还没有完成输入之前就完成解析了，这不再是一个问题。 简单，快速的错误恢复就可以了。

> ### 6.3.1 Panic mode error recovery

6.3.1 恐慌模式错误恢复

> Of all the recovery techniques devised in yesteryear, the one that best stood the test of time is called—somewhat alarmingly—**panic mode**. As soon as the parser detects an error, it enters panic mode. It knows at least one token doesn't make sense given its current state in the middle of some stack of grammar productions.

在过去设计的所有恢复技术中，最能经受住时间考验的一种叫做**恐慌模式**（有点令人震惊）。一旦解析器检测到一个错误，它就会进入恐慌模式。它知道至少有一个token是没有意义的，因为它目前的状态是在一些语法生成式的堆栈中间。

> Before it can get back to parsing, it needs to get its state and the sequence of forthcoming tokens aligned such that the next token does match the rule being parsed. This process is called **synchronization**.

在程序继续进行解析之前，它需要将自己的状态和即将到来的标记序列对齐，使下一个标记能够匹配正则解析的规则。这个过程称为**同步**。

> To do that, we select some rule in the grammar that will mark the synchronization point. The parser fixes its parsing state by jumping out of any nested productions until it gets back to that rule. Then it synchronizes the token stream by discarding tokens until it reaches one that can appear at that point in the rule.

为此，我们在语法中选择一些规则来标记同步点。解析器会跳出所有嵌套的生成式直到回退至该规则中，来修复其解析状态。然后，它会丢弃标记，直到遇到一个可以匹配该规则的标记，以此来同步标记流。

> Any additional real syntax errors hiding in those discarded tokens aren't reported, but it also means that any mistaken cascaded errors that are side effects of the initial error aren't *falsely* reported either,

> which is a decent trade-off.

这些被丢弃的标记中隐藏的其它真正的语法错误都不会被报告，但是这也意味着由初始错误引起的其它级联错误也不会被*错误地*报告出来，这是个不错的权衡。

> The traditional place in the grammar to synchronize is between statements. We don't have those yet, so we won't actually synchronize in this chapter, but we'll get the machinery in place for later.

语法中传统的要同步的地方是语句之间。我们还没有这些，所以我们不会在这一章中真正地同步，但我们会在以后把这些机制准备好。

> ### 6.3.2 Entering panic mode

6.3.2 进入恐慌模式

> Back before we went on this side trip around error recovery, we were writing the code to parse a parenthesized expression. After parsing the expression, it looks for the closing `)` by calling `consume()`. Here, finally, is that method:

在我们讨论错误恢复之前，我们正在编写解析括号表达式的代码。在解析表达式之后，会调用`consume()`方法查找收尾的`)`。这里，终于可以实现那个方法了：

lox/Parser.java，在 match()方法后添加：

```java
  private Token consume(TokenType type, String message) {
    if (check(type)) return advance();

    throw error(peek(), message);
  }
```

> It's similar to `match()` in that it checks to see if the next token is of the expected type. If so, it consumes it and everything is groovy. If some other token is there, then we've hit an error. We report it by calling this:

它和 `match()`方法类似，检查下一个标记是否是预期的类型。如果是，它就会消费该标记，一切都很顺利。如果是其它的标记，那么我们就遇到了错误。我们通过调用下面的方法来报告错误：

lox/Parser.java，在 previous()方法后添加：

```java
  private ParseError error(Token token, String message) {
    Lox.error(token, message);
    return new ParseError();
  }
```

> First, that shows the error to the user by calling:

首先，通过调用下面的方法向用户展示错误信息：

lox/Lox.java，在 report()方法后添加：

```
  static void error(Token token, String message) {
    if (token.type == TokenType.EOF) {
      report(token.line, " at end", message);
    } else {
      report(token.line, " at '" + token.lexeme + "'", message);
    }
  }
```

> This reports an error at a given token. It shows the token’s location and the token itself. This will come in handy later since we use tokens throughout the interpreter to track locations in code.

该方法会报告给定标记处的错误。它显示了标记的位置和标记本身。这在以后会派上用场，因为我们在整个解释器中使用标记来跟踪代码中的位置。

> After we report the error, the user knows about their mistake, but what does the *parser* do next? Back in `error()`, we create and return a ParseError, an instance of this new class:

在我们报告错误后，用户知道了他们的错误，但接下来解析器要做什么呢？回到`error()`方法中，我们创建并返回了一个`ParseError`，是下面这个新类的实例:

lox/Parser.java，在 Parser 中嵌入内部类：

```
class Parser {
  // 新增部分开始
  private static class ParseError extends RuntimeException {}
  // 新增部分结束
  private final List<Token> tokens;
```

> This is a simple sentinel class we use to unwind the parser. The `error()` method *returns* the error instead of *throwing* it because we want to let the calling method inside the parser decide whether to unwind or not. Some parse errors occur in places where the parser isn’t likely to get into a weird state and we don’t need to synchronize. In those places, we simply report the error and keep on truckin’.

这是一个简单的哨兵类，我们用它来帮助解析器摆脱错误。`error()`方法是*返回*错误而不是*抛出*错误，因为我们希望解析器内的调用方法决定是否要跳脱出该错误。有些解析错误发生在解析器不可能进入异常状态的地方，这时我们就不需要同步。在这些地方，我们只需要报告错误，然后继续解析。

> For example, Lox limits the number of arguments you can pass to a function. If you pass too many, the parser needs to report that error, but it can and should simply keep on parsing the extra arguments instead of freaking out and going into panic mode.

例如，Lox限制了你可以传递给一个函数的参数数量。如果你传递的参数太多，解析器需要报告这个错误，但它可以而且应该继续解析额外的参数，而不是惊慌失措，进入恐慌模式^11。

> In our case, though, the syntax error is nasty enough that we want to panic and synchronize. Discarding tokens is pretty easy, but how do we synchronize the parser’s own state?

但是，在我们的例子中，语法错误非常严重，以至于我们要进入恐慌模式并进行同步。丢弃标记非常简单，但是我们如何同步解析器自己的状态呢？

> 6.3.3 Synchronizing a recursive descent parser

6.3.3 同步递归下降解析器

> With recursive descent, the parser's state—which rules it is in the middle of recognizing—is not stored explicitly in fields. Instead, we use Java's own call stack to track what the parser is doing. Each rule in the middle of being parsed is a call frame on the stack. In order to reset that state, we need to clear out those call frames.

在递归下降中，解析器的状态（即它正在识别哪个规则）不是显式存储在字段中的。相反，我们使用Java自身的调用栈来跟踪解析器正在做什么。每一条正在被解析的规则都是栈上的一个调用帧。为了重置状态，我们需要清除这些调用帧。

> The natural way to do that in Java is exceptions. When we want to synchronize, we *throw* that ParseError object. Higher up in the method for the grammar rule we are synchronizing to, we'll catch it. Since we synchronize on statement boundaries, we'll catch the exception there. After the exception is caught, the parser is in the right state. All that's left is to synchronize the tokens.

在Java中，最自然的实现方式是异常。当我们想要同步时，我们抛出ParseError对象。在我们正同步的语法规则的方法上层，我们将捕获它。因为我们在语句边界上同步，所以我们可以在那里捕获异常。捕获异常后，解析器就处于正确的状态。剩下的就是同步标记了。

> We want to discard tokens until we're right at the beginning of the next statement. That boundary is pretty easy to spot—it's one of the main reasons we picked it. *After* a semicolon, we're probably finished with a statement. Most statements start with a keyword—`for`, `if`, `return`, `var`, etc. When the *next* token is any of those, we're probably about to start a statement.

我们想要丢弃标记，直至达到下一条语句的开头。这个边界很容易发现——这也是我们选其作为边界的原因。在*分号之后*，我们可能就结束了一条语句^12。大多数语句都通过一个关键字开头——`for`、`if`、`return`、`var`等等。当下一个标记是其中之一时，我们可能就要开始一条新语句了。

> This method encapsulates that logic:

下面的方法封装了这个逻辑：

lox/Parser.java，在 error()方法后添加：

```
  private void synchronize() {
    advance();

    while (!isAtEnd()) {
      if (previous().type == SEMICOLON) return;

      switch (peek().type) {
        case CLASS:
        case FUN:
        case VAR:
        case FOR:
        case IF:
        case WHILE:
        case PRINT:
        case RETURN:
          return;
      }

      advance();
    }
  }
```

> It discards tokens until it thinks it has found a statement boundary. After catching a ParseError, we'll call this and then we are hopefully back in sync. When it works well, we have discarded tokens that would have likely caused cascaded errors anyway, and now we can parse the rest of the file starting at the next statement.

该方法会不断丢弃标记，直到它发现一个语句的边界。在捕获一个ParseError后，我们会调用该方法，然后我们就有望回到同步状态。当它工作顺利时，我们就已经丢弃了无论如何都可能会引起级联错误的语法标记，现在我们可以从下一条语句开始解析文件的其余部分。

> Alas, we don't get to see this method in action, since we don't have statements yet. We'll get to that in a couple of chapters. For now, if an error occurs, we'll panic and unwind all the way to the top and stop parsing. Since we can parse only a single expression anyway, that's no big loss.

唉，我们还没有看到这个方法的实际应用，因为我们目前还没有语句。我们会在后面几章中开始引入语句。现在，如果出现错误，我们就会进入恐慌模式，一直跳出到最顶层，并停止解析。由于我们只能解析一个表达式，所以这并不是什么大损失。

6.4 Wiring up the Parser

6.4 调整解析器

> We are mostly done parsing expressions now. There is one other place where we need to add a little error handling. As the parser descends through the parsing methods for each grammar rule, it eventually hits `primary()`. If none of the cases in there match, it means we are sitting on a token that can't start an expression. We need to handle that error too.

我们现在基本上已经完成了对表达式的解析。我们还需要在另一个地方添加一些错误处理。当解析器在每个语法规则的解析方法中下降时，它最终会进入`primary()`。如果该方法中的case都不匹配，就意味着我们正面对一个不是表达式开头的语法标记。我们也需要处理这个错误。

lox/Parser.java，在 primary()方法中添加：

```
    if (match(LEFT_PAREN)) {
      Expr expr = expression();
      consume(RIGHT_PAREN, "Expect ')' after expression.");
      return new Expr.Grouping(expr);
    }
    // 新增部分开始
    throw error(peek(), "Expect expression.");
    // 新增部分结束
  }
```

> With that, all that remains in the parser is to define an initial method to kick it off. That method is called, naturally enough, `parse()`.

这样，解析器中剩下的工作就是定义一个初始方法来启动它。这个方法自然应该叫做`parse()`。

lox/Parser.java，在 Parser()方法后添加：

```
  Expr parse() {
    try {
      return expression();
    } catch (ParseError error) {
      return null;
    }
  }
```

> We'll revisit this method later when we add statements to the language. For now, it parses a single expression and returns it. We also have some temporary code to exit out of panic mode. Syntax error recovery is the parser's job, so we don't want the ParseError exception to escape into the rest of the interpreter.

稍后在向语言中添加语句时，我们将重新审视这个方法。目前，它只解析一个表达式并返回它。我们还有一些临时代码用于退出恐慌模式。语法错误恢复是解析器的工作，所以我们不希望ParseError异常逃逸到解释器的其它部分。

> When a syntax error does occur, this method returns `null`. That's OK. The parser promises not to crash or hang on invalid syntax, but it doesn't promise to return a *usable syntax tree* if an error is found. As soon as the parser reports an error, `hadError` gets set, and subsequent phases are skipped.

当确实出现语法错误时，该方法会返回`null`。这没关系。解析器承诺不会因为无效语法而崩溃或挂起，但它不承诺在发现错误时返回一个*可用的语法树*。一旦解析器报告错误，就会对`hadError`赋值，然后跳过后续阶段。

> Finally, we can hook up our brand new parser to the main Lox class and try it out. We still don't have an interpreter, so for now, we'll parse to a syntax tree and then use the AstPrinter class from the last chapter to display it.

最后，我们可以将全新的解析器挂到Lox主类并进行试验。我们仍然还没有解释器，所以现在，我们将表达式解析为一个语法树，然后使用上一章中的AstPrinter类来显示它。

> Delete the old code to print the scanned tokens and replace it with this:

删除打印已扫描标记的旧代码，将其替换为：

lox/Lox.java，在 run()方法中，替换其中5行

```
    List<Token> tokens = scanner.scanTokens();
    // 替换部分开始
    Parser parser = new Parser(tokens);
    Expr expression = parser.parse();

    // Stop if there was a syntax error.
    if (hadError) return;

    System.out.println(new AstPrinter().print(expression));
    // 替换部分结束
  }
```

> Congratulations, you have crossed the threshold! That really is all there is to handwriting a parser. We'll extend the grammar in later chapters with assignment, statements, and other stuff, but none of that is any more complex than the binary operators we tackled here.

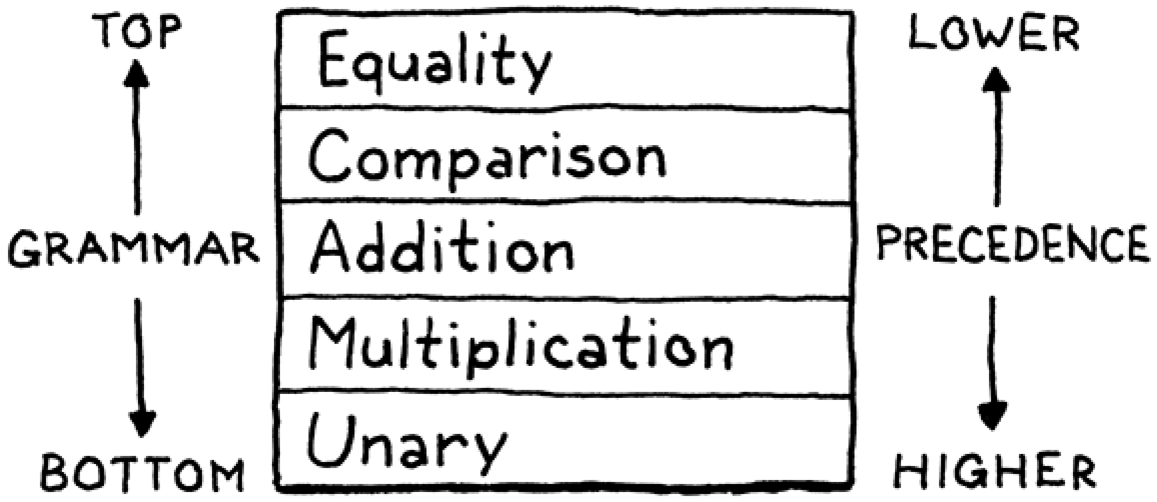
祝贺你，你已经跨过了门槛!这就是手写解析器的全部内容^13。我们将在后面的章节中扩展赋值、语句和其它特性对应的语法，但这些都不会比我们本章处理的二元操作符更复杂。

> Fire up the interpreter and type in some expressions. See how it handles precedence and associativity correctly? Not bad for less than 200 lines of code.

启动解释器并输入一些表达式。查看它是如何正确处理优先级和结合性的?这对于不到200行代码来说已经很不错了。

^1: 英语中的"Parse "来自古法语 "pars"，意为 "语言的一部分"。它的意思是取一篇文章，把每一个词都映射到语言的语法上。我们在这里使用它也是这个意思，只不过我们的语言比古法语更现代一些。 ^2: 可以想见，在那些老机器上进行汇编编程是多么痛苦，以至于他们认为Fortran是一种改进。 ^3: 虽然现在并不常见，但有些语言规定某些运算符之间没有相对优先级。这种语言中，在表达式中混合使用这些操作符而不使用显式分组是一种语法错误。同样，有些运算符是**非结合**的。这意味着在语句序列中多次使用该操作符是错误的。例如，Perl的范围操作符是非结合的，所以`a ..b`是可以的，但是`a ..b . .c`是错误的。 ^4: 一些解析器生成器并没有将优先级直接写入语法规则中，而是允许你保持同样的模糊但简单的语法，然后在旁边添加一点明确的操作符优先级元数据，以消除歧义。 ^5: 我们可以取消`expression`，而只是在其他包含表达式的规则中使用`equality`，但使用`expression`会使这些其他规则可读性更好。另外，在后面的章节中，当我们将语法扩展到包括赋值和逻辑运算符时，我们只需要改变`expression`的生成式，而不需要修改每条包含`expression`的规则。 ^6: 原则上，你把乘法当作左关联还是右关联都没有关系——无论你使用哪种方式都可以得到相同的结果。但是，在精度有限的情况下，舍入和溢出意味着关联性会影响乘法序列的计算结果。如`print 0. 1 * (0. 2 * 0. 3);`和`print (0.1 * 0.2) * 0.3;`，在Lox等使用IEEE 754双精度浮点数的语言中，第一个算式的计算结果是`0.006`，而第二个算式的计算结果是`0.006000000000000001`。有时，这种微小的差异很重要。可以在这里了解更多信息。

^7: 该方法之所以被称为"递归 *下降*"，是因为它是沿着语法*向下*运行的。令人困惑的是，在谈论"高"和"低"优先级时，我们也使用方向来比喻，但是方向却是相反的。在自顶向下的解析器中，首先达到优先级最低的表达式，因为其中可能包含优先级更高的子表达式。



CS的人真的需要聚在一起理清他们的隐喻。甚至不要让我开始讨论堆栈向哪个方向生长，或者为什么树的根在上面。 ^8: 这就是为什么左递归对于递归下降是有问题的。左递归规则的函数会立即调用自身，并循环往复，直到解析器遇到堆栈溢出并崩溃。 ^9: 解析`a==b==c==d`。对于每一次迭代，使用前一个子式结果作为左操作数并创建一个新的二元表达式。 ^10: 解析器提前观察即将到来的标记来决定如何解析，这就把递归下降纳入了**预测性解析器**的范畴。 ^11: 另一种处理常见语法错误的方法是**错误生成式**。你可以使用一个能*成功*匹配*错误*语法的规则来扩充语法。解析器可以对其进行安全地解析，但是不会生成语法树，而是会报告一个错误。
举例来说，有些语言中有一元运算符`+`，如`+123`，但是Lox不支持。当解析器在表达式的开头遇到一个`+`时，我们不必感到困惑，我们可以扩展一元规则来允许该语法。

```
unary → ( "!" | "-" | "+" ) unary | primary ;
```

这样解析器就会消费`+`标记，而不是进入恐慌模式或让解析器陷入奇怪的状态。
错误生成式的效果很好。因为你作为解析器的作者，知道代码的*如何*出错的以及用户想要做什么。这意味着你可以给出一个更有用的信息来帮助用户回到正轨，比如，"不支持一元'+'表达式"。成熟的解析器往往会积累错误生成式，因为它们可以帮助用户修复常见的错误。 ^12: 我说 "可能 "是因为我们可以在for循环中碰到分隔子句的分号。我们的同步并不完美，但这没关系。我们已经准确地报告了第一个错误，所以之后的一切都算是 "尽力而为 "了。 ^13: 你可能会定义一个比Lox更复杂的语法，使用递归下降法难以对其解析。当你可能需要预先查看大量的标记以弄清你面临的情况时，预测性解析就变得很棘手。实际上，大多数语言都是为了避免这种情况而设计的。 即使情况并非如此，您通常也可以毫不费力地解决问题。 既然您可以使用递归下降来解析C ++（许多C ++编译器都可以做到），那么您就可以解析任何内容。

CHALLENGES

习题

> 1、In C, a block is a statement form that allows you to pack a series of statements where a single one is expected. The comma operator is an analogous syntax for expressions. A comma-separated series of expressions can be given where a single expression is expected (except inside a function call’s argument list). At runtime, the comma operator evaluates the left operand and discards the result. Then it evaluates and returns the right operand.
>
> Add support for comma expressions. Give them the same precedence and associativity as in C. Write the grammar, and then implement the necessary parsing code.

1、在C语言中，块是一种语句形式，它允许你把一系列的语句打包作为一个语句来使用。逗号运算符是表达式的类似语法。可以在需要单个表达式的地方给出以逗号分隔的表达式序列(函数调用的参数列表除外)。在运行时，逗号操作符计算左操作数并丢弃结果。然后计算并返回右操作数。

添加对逗号表达式的支持。赋予它们与c语言中相同的优先级和结合性。编写语法，然后实现必要的解析代码。

> 2、Likewise, add support for the C-style conditional or “ternary” operator `?:`. What precedence level is allowed between the `?` and `:`? Is the whole operator left-associative or right-associative?

2、同样，添加对C风格的条件操作符或 "三元 "操作符`?:`的支持。在`?`和`:`之间采用什么优先级顺序？整个操作符是左关联还是右关联？

> 3、Add error productions to handle each binary operator appearing without a left-hand operand. In other words, detect a binary operator appearing at the beginning of an expression. Report that as an error, but also parse and discard a right-hand operand with the appropriate precedence.

3、添加错误生成式处理没有左操作数的二元操作符。换句话说，检测出现在表达式开头的二元操作符。将其作为错误报告给用户，同时也要解析并丢弃具有相应优先级的右操作数。

DESIGN NOTE: LOGIC VERSUS HISTORY

> Let’s say we decide to add bitwise `&` and `|` operators to Lox. Where should we put them in the precedence hierarchy? C—and most languages that follow in C’s footsteps—place them below `==`. This is widely considered a mistake because it means common operations like testing a flag require parentheses.
>
> ```
> if (flags & FLAG_MASK == SOME_FLAG) { ... } // Wrong.
> if ((flags & FLAG_MASK) == SOME_FLAG) { ... } // Right.
> ```
>
> Should we fix this for Lox and put bitwise operators higher up the precedence table than C does? There are two strategies we can take.
>
> You almost never want to use the result of an `==` expression as the operand to a bitwise operator. By making bitwise bind tighter, users don’t need to parenthesize as often. So if we do that, and users assume the precedence is chosen logically to minimize parentheses, they’re likely to infer it correctly.
>
> This kind of internal consistency makes the language easier to learn because there are fewer edge cases and exceptions users have to stumble into and then correct. That’s good, because before users can use our language, they have to load all of that syntax and semantics into their heads. A simpler, more rational language *makes sense*.
>
> But, for many users there is an even faster shortcut to getting our language’s ideas into their wetware—*use concepts they already know*. Many newcomers to our language will be coming from some other language or languages. If our language uses some of the same syntax or semantics as those, there is much less for the user to learn (and *unlearn*).
>
> This is particularly helpful with syntax. You may not remember it well today, but way back when you learned your very first programming language, code probably looked alien and unapproachable. Only through painstaking effort did you learn to read and accept it. If you design a novel syntax for your new language, you force users to start that process all over again.

> Taking advantage of what users already know is one of the most powerful tools you can use to ease adoption of your language. It's almost impossible to overestimate how valuable this is. But it faces you with a nasty problem: What happens when the thing the users all know *kind of sucks*? C's bitwise operator precedence is a mistake that doesn't make sense. But it's a *familiar* mistake that millions have already gotten used to and learned to live with.
>
> Do you stay true to your language's own internal logic and ignore history? Do you start from a blank slate and first principles? Or do you weave your language into the rich tapestry of programming history and give your users a leg up by starting from something they already know?
>
> There is no perfect answer here, only trade-offs. You and I are obviously biased towards liking novel languages, so our natural inclination is to burn the history books and start our own story.
>
> In practice, it's often better to make the most of what users already know. Getting them to come to your language requires a big leap. The smaller you can make that chasm, the more people will be willing to cross it. But you can't *always* stick to history, or your language won't have anything new and compelling to give people a *reason* to jump over.

设计笔记：逻辑和历史

假设我们决定在Lox中添加位元`&`和`|`运算符。我们应该将它们放在优先级层次结构的哪个位置？ C（以及大多数跟随C语言步伐的语言）将它们放在`==`之下。 目前普遍认为这是一个错误，因为这意味着检测标志位等常用操作都需要加括号。

```
if (flags & FLAG_MASK == SOME_FLAG) { ... } // Wrong.
if ((flags & FLAG_MASK) == SOME_FLAG) { ... } // Right.
```

我们是否应该在 Lox 中修正这个问题，为位运算符赋予比 C 中更高的优先级？我们可以采取两种策略。

几乎可以肯定你不会想把`==`表达式的计算结果当作位运算的操作数。将位运算操作绑定更紧密，用户就不需要像以前那样经常使用括号。所以如果我们这样做，并且用户认为优先级的选择是合乎逻辑的，是为了尽量减少小括号，他们很可能会正确地推断出来。

这种内部一致性使语言更容易学习，因为用户需要纠正的边界情况和异常变少了。这很好，因为用户在使用我们的语言之前，需要先理解所有的语法和语义。一个更简单、更合理的语言是*有意义的*。

但是，对于许多用户来说，有一个更快的捷径，可以将我们语言的思想融入他们的湿件中——*使用他们已经知道的概念*。许多我们语言的新用户都使用过其它一门或多门语言。如果我们的语言使用了与那些语言相同的一些语法或语义，那么用户需要学习（和*忘掉*）的东西就会少很多。

这对词法语法特别有帮助。您现在可能不太记得了，但是回想一下您学习第一门编程语言时，代码看起来似乎很陌生且难以理解。 只有通过艰苦的努力，您才学会阅读和接受它。 如果你为你的新语言设计了一种新颖的语法，你就会迫使用户重新开始这个过程。

利用用户已经知道的知识，是你可以用来简化语言采用的最强大的工具之一。这一点的价值怎么估计都不过分。但它也给你带来了一个棘手的问题：如果用户都知道的东西*有点糟糕*时，会发生什么？C语言的位运算操作符优先是一个没有意义的错误。但这是一个数以百万计的人已经习惯并学会忍受的熟悉错误。

你是否忠于语言的内在逻辑而忽略历史？你是从一张白纸和基本原则开始的吗？还是把你的语言编织到丰富的编程历史中去，从用户已经知道的东西开始，使您的用户受益？

这里没有完美的答案，只有权衡取舍。你和我显然都倾向于喜欢新奇的语言，所以我们的自然倾向是烧掉历史书，开始我们自己的故事。

在实践中，充分利用用户已经知道的知识往往更好。让他们来使用你的语言需要一个大的跨越。两个语言间的鸿沟越小，人们就越愿意跨越它。但你不能总是拘泥于历史，否则你的语言就不会有什么新颖的、令人信服的东西让用户们有理由跳过去。

7.表达式求值

> You are my creator, but I am your master; Obey!
>
> —— Mary Shelley, *Frankenstein*

你是我的创造者，但我是你的主人，听话！

——Mary Shelley, *科学怪人*

> If you want to properly set the mood for this chapter, try to conjure up a thunderstorm, one of those swirling tempests that likes to yank open shutters at the climax of the story. Maybe toss in a few bolts of lightning. In this chapter, our interpreter will take breath, open its eyes, and execute some code.

如果你想为这一章适当地设定气氛，试着想象一场雷雨，那种在故事高潮时喜欢拉开百叶窗的漩涡式暴风雨。也许再加上几道闪电。在这一章中，我们的解释器将开始呼吸，睁开眼睛，并执行一些代码。

> There are all manner of ways that language implementations make a computer do what the user's source code commands. They can compile it to machine code, translate it to another high-level language, or reduce it to some bytecode format for a virtual machine to run. For our first interpreter, though, we are going to take the simplest, shortest path and execute the syntax tree itself.

对于语言实现来说，有各种方式可以使计算机执行用户的源代码命令。它们可以将其编译为机器代码，将其翻译为另一种高级语言，或者将其还原为某种字节码格式，以便在虚拟机中执行。不过对于我们的第一个解释

器，我们要选择最简单、最短的一条路，也就是执行语法树本身。

> Right now, our parser only supports expressions. So, to “execute” code, we will evaluate an expression and produce a value. For each kind of expression syntax we can parse—literal, operator, etc.—we need a corresponding chunk of code that knows how to evaluate that tree and produce a result. That raises two questions:

现在，我们的解释器只支持表达式。因此，为了“执行”代码，我们要计算一个表达式时并生成一个值。对于我们可以解析的每一种表达式语法——字面量，操作符等——我们都需要一个相应的代码块，该代码块知道如何计算该语法树并产生结果。这也就引出了两个问题：

1. > What kinds of values do we produce?

   我们要生成什么类型的值？

2. > How do we organize those chunks of code?

   我们如何组织这些代码块？

> Taking them on one at a time . . .

让我们来逐个击破。

> ## 7 . 1 Representing Values

7.1 值描述

> In Lox, values are created by literals, computed by expressions, and stored in variables. The user sees these as *Lox* objects, but they are implemented in the underlying language our interpreter is written in. That means bridging the lands of Lox’s dynamic typing and Java’s static types. A variable in Lox can store a value of any (Lox) type, and can even store values of different types at different points in time. What Java type might we use to represent that?

在Lox中，值由字面量创建，由表达式计算，并存储在变量中。用户将其视作Lox对象^1，但它们是用编写解释器的底层语言实现的。这意味着要在Lox的动态类型和Java的静态类型之间架起桥梁。Lox中的变量可以存储任何（Lox）类型的值，甚至可以在不同时间存储不同类型的值。我们可以用什么Java类型来表示？

> Given a Java variable with that static type, we must also be able to determine which kind of value it holds at runtime. When the interpreter executes a `+` operator, it needs to tell if it is adding two numbers or concatenating two strings. Is there a Java type that can hold numbers, strings, Booleans, and more? Is there one that can tell us what its runtime type is? There is! Good old java.lang.Object.

给定一个具有该静态类型的Java变量，我们还必须能够在运行时确定它持有哪种类型的值。当解释器执行 `+`运算符时，它需要知道它是在将两个数字相加还是在拼接两个字符串。有没有一种Java类型可以容纳数字、字符串、布尔值等等？有没有一种类型可以告诉我们它的运行时类型是什么？有的! 就是老牌的 `java.lang.Object`。

> In places in the interpreter where we need to store a Lox value, we can use Object as the type. Java has boxed versions of its primitive types that all subclass Object, so we can use those for Lox’s built-in types:

在解释器中需要存储Lox值的地方，我们可以使用Object作为类型。Java已经将其基本类型的所有子类对象装箱了，因此我们可以将它们用作Lox的内置类型：

Lox type Lox类	Java representation Java表示
Any Lox value	Object
`nil`	`null`
Boolean	Boolean
number	Double
string	String

> Given a value of static type Object, we can determine if the runtime value is a number or a string or whatever using Java's built-in `instanceof` operator. In other words, the JVM's own object representation conveniently gives us everything we need to implement Lox's built-in types. We'll have to do a little more work later when we add Lox's notions of functions, classes, and instances, but Object and the boxed primitive classes are sufficient for the types we need right now.

给定一个静态类型为Object的值，我们可以使用Java内置的`instanceof`操作符来确定运行时的值是数字、字符串或其它什么。换句话说，JVM自己的对象表示方便地为我们提供了实现Lox内置类型所需的一切^2。当稍后添加Lox的函数、类和实例等概念时，我们还必须做更多的工作，但Object和基本类型的包装类足以满足我们现在的需要。

7.2 Evaluating Expressions

7.2 表达式求值

> Next, we need blobs of code to implement the evaluation logic for each kind of expression we can parse. We could stuff that code into the syntax tree classes in something like an `interpret()` method. In effect, we could tell each syntax tree node, "Interpret thyself". This is the Gang of Four's Interpreter design pattern. It's a neat pattern, but like I mentioned earlier, it gets messy if we jam all sorts of logic into the tree classes.

接下来，我们需要大量的代码实现我们可解析的每种表达式对应的求值逻辑。我们可以把这些代码放在语法树的类中，比如添加一个`interpret()`方法。然后，我们可以告诉每一个语法树节点"解释你自己"，这就是四人组的解释器模式。这是一个整洁的模式，但正如我前面提到的，如果我们将各种逻辑都塞进语法树类中，就会变得很混乱。

> Instead, we're going to reuse our groovy Visitor pattern. In the previous chapter, we created an AstPrinter class. It took in a syntax tree and recursively traversed it, building up a string which it ultimately returned. That's almost exactly what a real interpreter does, except instead of concatenating strings, it computes values.

相反，我们将重用我们的访问者模式。在前面的章节中，我们创建了一个AstPrinter类。它接受一个语法树，并递归地遍历它，构建一个最终返回的字符串。这几乎就是一个真正的解释器所做的事情，只不过解释器不是连接字符串，而是计算值。

> We start with a new class.

我们先创建一个新类。

lox/Interpreter.java，创建新文件：

```
package com.craftinginterpreters.lox;

class Interpreter implements Expr.Visitor<Object> {
}
```

> The class declares that it's a visitor. The return type of the visit methods will be Object, the root class that we use to refer to a Lox value in our Java code. To satisfy the Visitor interface, we need to define visit methods for each of the four expression tree classes our parser produces. We'll start with the simplest . . .

这个类声明它是一个访问者。访问方法的返回类型将是Object，即我们在Java代码中用来引用Lox值的根类。为了实现Visitor接口，我们需要为解析器生成的四个表达式树类中分别定义访问方法。我们从最简单的开始...

> ### 7.2.1 Evaluating literals

7.2.1 字面量求值

> The leaves of an expression tree—the atomic bits of syntax that all other expressions are composed of—are literals. Literals are almost values already, but the distinction is important. A literal is a *bit of syntax* that produces a value. A literal always appears somewhere in the user's source code. Lots of values are produced by computation and don't exist anywhere in the code itself. Those aren't literals. A literal comes from the parser's domain. Values are an interpreter concept, part of the runtime's world.

一个表达式树的叶子节点（构成其它表达式的语法原子单位）是字面量^3。字面符号几乎已经是值了，但两者的区别很重要。字面量是产生一个值的语法单元。字面量总是出现在用户的源代码中的某个地方。而很多值是通过计算产生的，并不存在于代码中的任何地方，这些都不是字面量。字面量来自于解析器领域，而值是一个解释器的概念，是运行时世界的一部分。

> So, much like we converted a literal *token* into a literal *syntax tree node* in the parser, now we convert the literal tree node into a runtime value. That turns out to be trivial.

因此，就像我们在解析器中将字面量*标记*转换为字面量*语法树节点*一样，现在我们将字面量树节点转换为运行时值。这其实很简单。

lox/Interpreter.java，在 Interpreter类中添加：

```
  @Override
  public Object visitLiteralExpr(Expr.Literal expr) {
    return expr.value;
  }
```

> We eagerly produced the runtime value way back during scanning and stuffed it in the token. The parser took that value and stuck it in the literal tree node, so to evaluate a literal, we simply pull it back

> out.

我们早在扫描过程中就即时生成了运行时的值，并把它放进了语法标记中。解析器获取该值并将其插入字面量语法树节点中，所以要对字面量求值，我们只需把它存的值取出来。

> ## 7.2.2 Evaluating parentheses

7.2.2 括号求值

> The next simplest node to evaluate is grouping—the node you get as a result of using explicit parentheses in an expression.

下一个要求值的节点是分组——在表达式中显式使用括号时产生的语法树节点。

lox/Interpreter.java，在 Interpreter类中添加：

```
@Override
public Object visitGroupingExpr(Expr.Grouping expr) {
  return evaluate(expr.expression);
}
```

> A grouping node has a reference to an inner node for the expression contained inside the parentheses. To evaluate the grouping expression itself, we recursively evaluate that subexpression and return it.

一个分组节点中包含一个引用指向对应于括号内的表达式的内部节点^4。要想计算括号表达式，我们只需要递归地对子表达式求值并返回结果即可。

> We rely on this helper method which simply sends the expression back into the interpreter's visitor implementation:

我们依赖于下面这个辅助方法，它只是将表达式发送回解释器的访问者实现中：

lox/Interpreter.java，在 Interpreter类中添加：

```
private Object evaluate(Expr expr) {
  return expr.accept(this);
```

> ## 7.2.3 Evaluating unary expressions

7.2.3 一元表达式求值

> Like grouping, unary expressions have a single subexpression that we must evaluate first. The difference is that the unary expression itself does a little work afterwards.

像分组表达式一样，一元表达式也有一个必须先求值的子表达式。不同的是，一元表达式自身在完成求值之后还会做一些工作。

lox/Interpreter.java，在 visitLiteralExpr()方法后添加：

```java
  @Override
  public Object visitUnaryExpr(Expr.Unary expr) {
    Object right = evaluate(expr.right);

    switch (expr.operator.type) {
      case MINUS:
        return -(double)right;
    }

    // Unreachable.
    return null;
  }
```

> First, we evaluate the operand expression. Then we apply the unary operator itself to the result of that. There are two different unary expressions, identified by the type of the operator token.

首先，我们计算操作数表达式，然后我们将一元操作符作用于子表达式的结果。我们有两种不同的一元表达式，由操作符标记的类型来区分。

> Shown here is `-`, which negates the result of the subexpression. The subexpression must be a number. Since we don't *statically* know that in Java, we cast it before performing the operation. This type cast happens at runtime when the `-` is evaluated. That's the core of what makes a language dynamically typed right there.

这里展示的是`-`，它会对子表达式的结构取负。子表达式结果必须是数字。因为我们在Java中无法*静态地*知道这一点，所以我们在执行操作之前先对其进行强制转换。这个类型转换是在运行时对`-`求值时发生的。这就是将语言动态类型化的核心所在。

> You can start to see how evaluation recursively traverses the tree. We can't evaluate the unary operator itself until after we evaluate its operand subexpression. That means our interpreter is doing a **post-order traversal**—each node evaluates its children before doing its own work.

你可以看到求值过程是如何递归遍历语法树的。在对一元操作符本身进行计算之前，我们必须先对其操作数子表达式求值。这表明，解释器正在进行**后序遍历**——每个节点在自己求值之前必须先对子节点求值。

> The other unary operator is logical not.

另一个一元操作符是逻辑非。

lox/Interpreter.java，在visitUnaryExpr()方法中添加：

```java
        switch (expr.operator.type) {
      // 新增部分开始
          case BANG:
        return !isTruthy(right);
      // 新增部分结束
      case MINUS:
```

> The implementation is simple, but what is this “truthy” thing about? We need to make a little side trip to one of the great questions of Western philosophy: *What is truth?*

实现很简单，但是这里的“真实”指的是什么呢？我们需要简单地讨论一下西方哲学中的一个伟大问题：什么是真理？

> ## 7.2.4 Truthiness and falsiness

7.2.4 真与假

> OK, maybe we’re not going to really get into the universal question, but at least inside the world of Lox, we need to decide what happens when you use something other than `true` or `false` in a logic operation like `!` or any other place where a Boolean is expected.

好吧，我们不会真正深入这个普世的问题，但是至少在Lox的世界中，我们需要确定当您在逻辑运算（如`!`或其他任何需要布尔值的地方）中使用非`true`或`false`以外的东西时会发生什么？

> We *could* just say it’s an error because we don’t roll with implicit conversions, but most dynamically typed languages aren’t that ascetic. Instead, they take the universe of values of all types and partition them into two sets, one of which they define to be “true”, or “truthful”, or (my favorite) “truthy”, and the rest which are “false” or “falsey”. This partitioning is somewhat arbitrary and gets weird in a few languages.

我们*可以*说这是一个错误，因为我们没有使用隐式转换，但是大多数动态类型语言并不那么严格。相反，他们把所有类型的值分成两组，其中一组他们定义为“真”，其余为“假”。这种划分有些武断，在一些语言中会变得很奇怪^5。

> Lox follows Ruby’s simple rule: `false` and `nil` are falsey, and everything else is truthy. We implement that like so:

Lox遵循Ruby的简单规则：`false`和`nil`是假的，其他都是真的。我们是这样实现的：

lox/Interpreter.java，在 visitUnaryExpr()方法后添加：

```java
  private boolean isTruthy(Object object) {
    if (object == null) return false;
    if (object instanceof Boolean) return (boolean)object;
    return true;
  }
```

> ## 7.2.5 Evaluating binary operators

7.2.5 二元操作符求值

> On to the last expression tree class, binary operators. There’s a handful of them, and we’ll start with the arithmetic ones.

来到最后的表达式树类——二元操作符，其中包含很多运算符，我们先从数学运算开始。

lox/Interpreter.java，在 evaluate()方法后添加^6：

```
  @Override
  public Object visitBinaryExpr(Expr.Binary expr) {
    Object left = evaluate(expr.left);
    Object right = evaluate(expr.right);

    switch (expr.operator.type) {
      case MINUS:
        return (double)left - (double)right;
      case SLASH:
        return (double)left / (double)right;
      case STAR:
        return (double)left * (double)right;
    }

    // Unreachable.
    return null;
  }
```

> I think you can figure out what's going on here. The main difference from the unary negation operator is that we have two operands to evaluate.

我想你能理解这里的实现。与一元取负运算符的主要区别是，我们有两个操作数要计算。

> I left out one arithmetic operator because it's a little special.

我漏掉了一个算术运算符，因为它有点特殊。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    switch (expr.operator.type) {
      case MINUS:
        return (double)left - (double)right;
      // 新增部分开始
      case PLUS:
        if (left instanceof Double && right instanceof Double) {
          return (double)left + (double)right;
        }

        if (left instanceof String && right instanceof String) {
          return (String)left + (String)right;
        }

        break;
      // 新增部分结束
      case SLASH:
```

> The `+` operator can also be used to concatenate two strings. To handle that, we don't just assume the operands are a certain type and *cast* them, we dynamically *check* the type and choose the appropriate operation. This is why we need our object representation to support `instanceof`.

+操作符也可以用来拼接两个字符串。为此，我们不能只是假设操作数是某种类型并将其强制转换，而是要动态地检查操作数类型并选择适当的操作。这就是为什么我们需要对象表示能支持`instanceof`。

> Next up are the comparison operators.

接下来是比较操作符。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    switch (expr.operator.type) {
            // 新增部分开始
      case GREATER:
        return (double)left > (double)right;
      case GREATER_EQUAL:
        return (double)left >= (double)right;
      case LESS:
        return (double)left < (double)right;
      case LESS_EQUAL:
        return (double)left <= (double)right;
      // 新增部分结束
      case MINUS:
```

> They are basically the same as arithmetic. The only difference is that where the arithmetic operators produce a value whose type is the same as the operands (numbers or strings), the comparison operators always produce a Boolean.

它们基本上与算术运算符相同。唯一的区别是，算术运算符产生的值的类型与操作数（数字或字符串）相同，而比较运算符总是产生一个布尔值。

> The last pair of operators are equality.

最后一对是等式运算符。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case BANG_EQUAL: return !isEqual(left, right);
      case EQUAL_EQUAL: return isEqual(left, right);
```

> Unlike the comparison operators which require numbers, the equality operators support operands of any type, even mixed ones. You can't ask Lox if 3 is *less* than `"three"`, but you can ask if it's *equal* to it.

与需要数字的比较运算符不同，等式运算符支持任何类型的操作数，甚至是混合类型。你不能问Lox 3是否*小于*`"three"`，但你可以问它3是否等于`"three"`。

> Like truthiness, the equality logic is hoisted out into a separate method.

与真假判断一样，相等判断也被提取到了单独的方法中。

lox/Interpreter.java，在 isTruthy()方法后添加：

```
  private boolean isEqual(Object a, Object b) {
    if (a == null && b == null) return true;
    if (a == null) return false;

    return a.equals(b);
  }
```

> This is one of those corners where the details of how we represent Lox objects in terms of Java matter. We need to correctly implement *Lox's* notion of equality, which may be different from Java's.

这是我们使用Java表示Lox对象的细节一角。我们需要正确地实现Lox的相等概念，这可能与Java中不同。

> Fortunately, the two are pretty similar. Lox doesn't do implicit conversions in equality and Java does not either. We do have to handle `nil`/`null` specially so that we don't throw a NullPointerException if we try to call `equals()` on `null`. Otherwise, we're fine. Java's `.equals()` method on Boolean, Double, and String have the behavior we want for Lox.

幸运的是，这两者很相似。Lox不会在等式中做隐式转换，Java也不会。我们必须对 `nil`/`null` 做特殊处理，这样就不会在对`null`调用`equals()`方法时抛出NullPointerException。其它情况下，都是没问题的。Java中的`.equals()`方法对Boolean、Double和 String的处理都符合Lox的要求^7。

> And that's it! That's all the code we need to correctly interpret a valid Lox expression. But what about an *invalid* one? In particular, what happens when a subexpression evaluates to an object of the wrong type for the operation being performed?

就这样了! 这就是我们要正确解释一个有效的Lox表达式所需要的全部代码。但是*无效的*表达式呢？尤其是，当一个子表达式的计算结果类型与待执行的操作不符时会发生什么？

> ## 7.3 Runtime Errors

7.3 运行时错误

> I was cavalier about jamming casts in whenever a subexpression produces an Object and the operator requires it to be a number or a string. Those casts can fail. Even though the user's code is erroneous, if we want to make a usable language, we are responsible for handling that error gracefully.

每当子表达式产生一个对象，而运算符要求它是一个数字或字符串时，我都会轻率地插入强制类型转换。这些类型转换可能会失败。如果我们想做出一个可用的语言，即使用户的代码是错误的，我们也有责任优雅地处理这个错误^8。

> It's time for us to talk about **runtime errors**. I spilled a lot of ink in the previous chapters talking about error handling, but those were all *syntax* or *static* errors. Those are detected and reported before *any* code is executed. Runtime errors are failures that the language semantics demand we detect and report while the program is running (hence the name).

现在是时候讨论**运行时错误**了。在前面的章节中，我花了很多笔墨讨论错误处理，但这些都是语法或静态错误。这些都是在代码执行之前进行检测和报告的。运行时错误是语言语义要求我们在程序运行时检测和报告的故障（因此得名）。

> Right now, if an operand is the wrong type for the operation being performed, the Java cast will fail and the JVM will throw a ClassCastException. That unwinds the whole stack and exits the application, vomiting a Java stack trace onto the user. That's probably not what we want. The fact that Lox is implemented in Java should be a detail hidden from the user. Instead, we want them to understand that a *Lox* runtime error occurred, and give them an error message relevant to our language and their program.

现在，如果操作数对于正在执行的操作来说是错误的类型，那么Java转换将失败，JVM将抛出一个ClassCastException。这将跳脱出整个调用堆栈并退出应用程序，然后向用户抛出Java堆栈跟踪信息。这可能不是我们想要的。Lox是用Java实现的这一事实应该是一个对用户隐藏的细节。相反，我们希望他们理解此时发生的是Lox运行时错误，并给他们一个与我们的语言和他们的程序相关的错误信息。

> The Java behavior does have one thing going for it, though. It correctly stops executing any code when the error occurs. Let's say the user enters some expression like:

不过，Java的行为确实有一个优点。当错误发生时，它会正确地停止执行代码。比方说，用户输入了一些表达式，比如：

```
2 * (3 / -"muffin")
```

> You can't negate a muffin, so we need to report a runtime error at that inner `-` expression. That in turn means we can't evaluate the `/` expression since it has no meaningful right operand. Likewise for the `*`. So when a runtime error occurs deep in some expression, we need to escape all the way out.

你无法对"muffin"取负，所以我们需要在内部的`-`表达式中报告一个运行时错误。这又意味着我们无法计算`/`表达式，因为它的右操作数无意义，对于`*`表达式也是如此。因此，当某个表达式深处出现运行时错误时，我们需要一直跳出到最外层。

> We could print a runtime error and then abort the process and exit the application entirely. That has a certain melodramatic flair. Sort of the programming language interpreter equivalent of a mic drop.

我们可以打印一个运行时错误，然后中止进程并完全退出应用程序。这有一点戏剧性，有点像编程语言解释器中的 "mic drop"。

> Tempting as that is, we should probably do something a little less cataclysmic. While a runtime error needs to stop evaluating the *expression*, it shouldn't kill the *interpreter*. If a user is running the REPL and has a typo in a line of code, they should still be able to keep the session going and enter more code after that.

尽管这种处理方式很诱人，我们或许应该做一些不那么灾难性的事情。虽然运行时错误需要停止对表达式的计算，但它不应该杀死解释器。如果用户正在运行REPL，并且在一行代码中出现了错误，他们应该仍然能够保持会话，并在之后继续输入更多的代码。

> ### 7.3.1 Detecting runtime errors

7.3.1 检测运行时错误

> Our tree-walk interpreter evaluates nested expressions using recursive method calls, and we need to unwind out of all of those. Throwing an exception in Java is a fine way to accomplish that. However, instead of using Java's own cast failure, we'll define a Lox-specific one so that we can handle it how we want.

我们的树遍历型解释器通过递归方法调用计算嵌套的表达式，而且我们需要能够跳脱出所有的调用层。在Java中抛出异常是实现这一点的好方法。但是，我们不使用Java自己的转换失败错误，而是定义一个Lox专用的错误，这样我们就可以按照我们想要的方式处理它。

> Before we do the cast, we check the object's type ourselves. So, for unary `-`, we add:

在进行强制转换之前，我们先自己检查对象的类型。因此，对于一元操作符`-`，我们需要添加代码：

lox/Interpreter.java，在visitUnaryExpr()方法中添加：

```
      case MINUS:
        // 新增部分开始
        checkNumberOperand(expr.operator, right);
        // 新增部分结束
        return -(double)right;
```

> The code to check the operand is:

检查操作数的代码如下：

lox/Interpreter.java，在 visitUnaryExpr()方法后添加：

```
  private void checkNumberOperand(Token operator, Object operand) {
    if (operand instanceof Double) return;
    throw new RuntimeError(operator, "Operand must be a number.");
  }
```

> When the check fails, it throws one of these:

当检查失败时，代码会抛出一个以下的错误：

lox/RuntimeError.java，新建源代码文件：

```
package com.craftinginterpreters.lox;

class RuntimeError extends RuntimeException {
  final Token token;

  RuntimeError(Token token, String message) {
    super(message);
    this.token = token;
  }
}
```

> Unlike the Java cast exception, our class tracks the token that identifies where in the user's code the runtime error came from. As with static errors, this helps the user know where to fix their code.

与Java转换异常不同，我们的类会跟踪语法标记，可以指明用户代码中抛出运行时错误的位置^9。与静态错误一样，这有助于用户知道去哪里修复代码。

> We need similar checking for the binary operators. Since I promised you every single line of code needed to implement the interpreters, I'll run through them all.

我们需要对二元运算符进行类似的检查。既然我答应了要展示实现解释器所需的每一行代码，那么我就把它们逐一介绍一遍。

> Greater than:

大于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case GREATER:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left > (double)right;
```

> Greater than or equal to:

大于等于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case GREATER_EQUAL:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left >= (double)right;
```

> Less than:

小于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case LESS:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left < (double)right;
```

> Less than or equal to:

小于等于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case LESS_EQUAL:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left <= (double)right;
```

> Subtraction:

减法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case MINUS:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left - (double)right;
```

> Division:

除法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case SLASH:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left / (double)right;
```

> Multiplication:

乘法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case STAR:
              // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left * (double)right;
```

> All of those rely on this validator, which is virtually the same as the unary one:

所有这些都依赖于下面这个验证器，它实际上与一元验证器相同^10：

lox/Interpreter.java，在 checkNumberOperand()方法后添加：

```
  private void checkNumberOperands(Token operator, Object left, Object right) {
    if (left instanceof Double && right instanceof Double) return;

    throw new RuntimeError(operator, "Operands must be numbers.");
  }
```

> The last remaining operator, again the odd one out, is addition. Since + is overloaded for numbers and strings, it already has code to check the types. All we need to do is fail if neither of the two success cases match.

剩下的最后一个运算符，也是最奇怪的一个，就是加法。由于+已经对数字和字符串进行重载，其中已经有检查类型的代码。我们需要做的就是在这两种情况都不匹配时失败。

lox/Interpreter.java，在 visitBinaryExpr()方法中替换一行：

```
          return (String)left + (String)right;
        }
        // 替换部分开始
        throw new RuntimeError(expr.operator,
            "Operands must be two numbers or two strings.");
        // 替换部分结束
      case SLASH:
```

> That gets us detecting runtime errors deep in the bowels of the evaluator. The errors are getting thrown. The next step is to write the code that catches them. For that, we need to wire up the Interpreter class into the main Lox class that drives it.

这样我们就可以在计算器的内部检测运行时错误。错误已经被抛出了。下一步就是编写能捕获这些错误的代码。为此，我们需要将Interpreter类连接到驱动它的Lox主类中。

> ## 7.4 Hooking Up the Interpreter

7.4 连接解释器

> The visit methods are sort of the guts of the Interpreter class, where the real work happens. We need to wrap a skin around them to interface with the rest of the program. The Interpreter's public API is simply one method.

visit方法是Interpreter类的核心部分，真正的工作是在这里进行的。我们需要给它们包上一层皮，以便与程序的其他部分对接。解释器的公共API只是一种方法。

lox/Interpreter.java，在 Interpreter类中添加：

```
  void interpret(Expr expression) {
    try {
      Object value = evaluate(expression);
      System.out.println(stringify(value));
    } catch (RuntimeError error) {
      Lox.runtimeError(error);
    }
  }
```

> This takes in a syntax tree for an expression and evaluates it. If that succeeds, `evaluate()` returns an object for the result value. `interpret()` converts that to a string and shows it to the user. To convert a Lox value to a string, we rely on:

该方法会接收一个表达式对应的语法树，并对其进行计算。如果成功了，`evaluate()`方法会返回一个对象作为结果值。`interpret()`方法将结果转为字符串并展示给用户。要将Lox值转为字符串，我们要依赖下面的方法：

lox/Interpreter.java，在 isEqual()方法后添加：

```
  private String stringify(Object object) {
    if (object == null) return "nil";

    if (object instanceof Double) {
      String text = object.toString();
      if (text.endsWith(".0")) {
        text = text.substring(0, text.length() - 2);
      }
      return text;
    }

    return object.toString();
  }
```

> This is another of those pieces of code like `isTruthy()` that crosses the membrane between the user's view of Lox objects and their internal representation in Java.

这是一段像`isTruthy()`一样的代码，它连接了Lox对象的用户视图和它们在Java中的内部表示。

> It's pretty straightforward. Since Lox was designed to be familiar to someone coming from Java, things like Booleans look the same in both languages. The two edge cases are `nil`, which we represent using Java's `null`, and numbers.

这很简单。由于Lox的设计旨在使Java使用者熟悉，因此Boolean之类的东西在两种语言中看起来是一样的。只有两种边界情况是`nil`(我们用Java的`null`表示)和数字。

> Lox uses double-precision numbers even for integer values. In that case, they should print without a decimal point. Since Java has both floating point and integer types, it wants you to know which one you're using. It tells you by adding an explicit `.0` to integer-valued doubles. We don't care about that, so we hack it off the end.

Lox即使对整数值也使用双精度数字^11。在这种情况下，打印时应该不带小数点。 由于Java同时具有浮点型和整型，它希望您知道正在使用的是哪一种类型。它通过在整数值的双数上添加一个明确的`.0`来告知用户。我们不关心这个，所以我们把它去掉。

> ### 7.4.1 Reporting runtime errors

7.4.1 报告运行时错误

> If a runtime error is thrown while evaluating the expression, `interpret()` catches it. This lets us report the error to the user and then gracefully continue. All of our existing error reporting code lives in the Lox class, so we put this method there too:

如果在计算表达式时出现了运行时错误，`interpret()`方法会将其捕获。这样我们可以向用户报告这个错误，然后优雅地继续执行。我们现有的所有错误报告代码都在Lox类中，所以我们也把这个方法放在其中：

lox/Lox.java，在 error()方法后添加：

```
static void runtimeError(RuntimeError error) {
  System.err.println(error.getMessage() +
      "\n[line " + error.token.line + "]");
  hadRuntimeError = true;
}
```

> We use the token associated with the RuntimeError to tell the user what line of code was executing when the error occurred. Even better would be to give the user an entire call stack to show how they *got* to be executing that code. But we don't have function calls yet, so I guess we don't have to worry about it.

我们使用与RuntimeError关联的标记来告诉用户错误发生时正在执行哪一行代码。更好的做法是给用户一个完整的调用堆栈，来显示他们是如何执行该代码的。但我们目前还没有函数调用，所以我想我们不必担心这个问题。

> After showing the error, `runtimeError()` sets this field:

展示错误之后，`runtimeError()`会设置以下字段：

lox/Lox.java，在 Lox类中添加：

```
static boolean hadError = false;
// 新增部分开始
static boolean hadRuntimeError = false;
// 新增部分结束
public static void main(String[] args) throws IOException {
```

> That field plays a small but important role.

这个字段担任着很小但很重要的角色。

lox/Lox.java，在 runFile()方法中添加：

```
    run(new String(bytes, Charset.defaultCharset()));

    // Indicate an error in the exit code.
    if (hadError) System.exit(65);
    // 新增部分开始
    if (hadRuntimeError) System.exit(70);
    // 新增部分结束
  }
```

> If the user is running a Lox script from a file and a runtime error occurs, we set an exit code when the process quits to let the calling process know. Not everyone cares about shell etiquette, but we do.

如果用户从文件中运行Lox脚本，并且发生了运行时错误，我们在进程退出时设置一个退出码，以便让调用进程知道。不是每个人都在乎shell的规矩，但我们在乎^12。

> ## 7.4.2 Running the interpreter

7.4.2 运行解释器

> Now that we have an interpreter, the Lox class can start using it.

现在我们有了解释器，Lox类可以开始使用它了。

lox/Lox.java，在 Lox类中添加：

```
public class Lox {
  // 新增部分开始
  private static final Interpreter interpreter = new Interpreter();
  // 新增部分结束
  static boolean hadError = false;
```

> We make the field static so that successive calls to `run()` inside a REPL session reuse the same interpreter. That doesn't make a difference now, but it will later when the interpreter stores global variables. Those variables should persist throughout the REPL session.

我们把这个字段设置为静态的，这样在一个REPL会话中连续调用`run()`时就会重复使用同一个解释器。目前这一点没有什么区别，但以后当解释器需要存储全局变量时就会有区别。这些全局变量应该在整个REPL会话中持续存在。

> Finally, we remove the line of temporary code from the last chapter for printing the syntax tree and replace it with this:

最后，我们删除上一章中用于打印语法树的那行临时代码，并将其替换为：

lox/Lox.java，在 run()方法中替换一行：

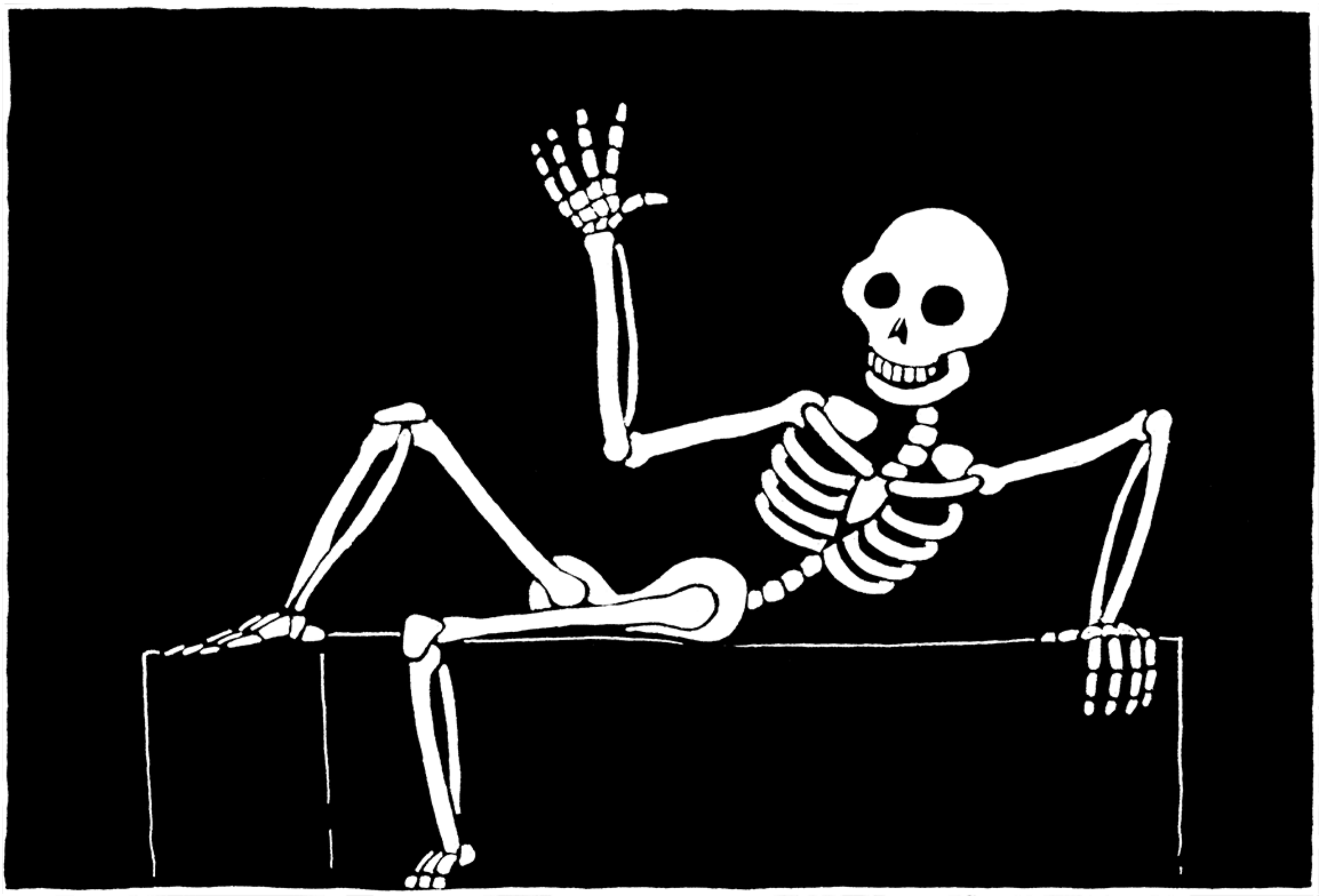
```
    // Stop if there was a syntax error.
    if (hadError) return;
    // 替换部分开始
    interpreter.interpret(expression);
    // 替换部分结束
  }
```

> We have an entire language pipeline now: scanning, parsing, and execution. Congratulations, you now have your very own arithmetic calculator.

我们现在有一个完整的语言管道：扫描、解析和执行。恭喜你，你现在有了你自己的算术计算器。

> As you can see, the interpreter is pretty bare bones. But the Interpreter class and the Visitor pattern we've set up today form the skeleton that later chapters will stuff full of interesting guts—variables, functions, etc. Right now, the interpreter doesn't do very much, but it's alive!

如您所见，这个解释器是非常简陋的。但是我们今天建立的解释器类和访问者模式构成了一个骨架，后面的章节中将填充入有趣的内容（变量，函数等）。现在，解释器的功能并不多，但它是活的!



^1: 在这里，我基本可以互换地使用 "值 "和 "对象"。稍后在C解释器中，我们会对它们稍作区分，但这主要是针对实现的两个不同方面（本地数据和堆分配数据）使用不同的术语。从用户的角度来看，这些术语是同义的。
^2: 我们需要对值做的另一件事是管理它们的内存，Java也能做到这一点。方便的对象表示和非常好的垃圾收集

器是我们用Java编写第一个解释器的主要原因。 ^3: 在下一章，当我们实现变量时，我们将添加标识符表达式，它也是叶子节点。 ^4: 有些解析器不为圆括号单独定义树节点。相应地，在解析带圆括号的表达式时，它们只返回内部表达式的节点。在Lox中，我们确实为圆括号创建了一个节点，因为稍后我们需要用它来正确处理赋值表达式的左值。 ^5: 在JavaScript中，字符串是真的，但空字符串不是。数组是真的，但空数组是……也是真的。数字0是假的，但字符串 "0 "是真的。
在 Python 中，空字符串是假的，就像在 JS 中一样，但其他空序列也是假的。
在PHP中，数字0和字符串 "0 "都是假的。大多数其他非空字符串是真实的。明白了吗？ ^6: 你是否注意到我们在这里固定了语言语义的一个细微的点？在二元表达式中，我们按从左到右的顺序计算操作数。如果这些操作数有副作用，那这个选择应该是用户可见的，所以这不是一个简单的实现细节。如果我们希望我们的两个解释器是一致的（提示：我们是一致的），我们就需要确保 clox 也是这样做的。 ^7: 你希望这个表达式的计算结果是什么？`(0 / 0) == (0 / 0)`。根据IEEE 754（它规定了双精度数的行为），用0除以0会得到特殊的**NaN**（不是一个数字）值。奇怪的是，NaN不等于它自己。
在Java中，基本类型double的`==`操作满足该规范，但是封装类Double的`equals()`方法不满足。Lox使用了后者，因此不遵循IEEE。这类微妙的不兼容问题占据了语言开发者生活中令人沮丧的一部分。 ^8: 我们完全可以不检测或报告一个类型错误。当你在C语言中把一个指针转换到与实际被指向的数据不匹配的类型上，C语言就是这样做的。C语言通过允许这样的操作获得了灵活性和速度，但它也是出了名的危险。一旦你错误地解释了内存中的数据，一切都完了。很少有现代语言接受这样的不安全操作。相反，大多数语言都是**内存安全**的，并通过静态和运行时检查的组合，确保程序永远不会错误地解释存储在内存中的值。 ^9: 我承认 "RuntimeError "这个名字令人困惑，因为Java定义了一个RuntimeException类。关于构建解释器的一件恼人的事情就是，您使用的名称经常与实现语言中已经使用的名称冲突。等我们支持Lox类就好了。 ^10: 另一个微妙的语义选择：在检查两个操作数的类型之前，我们先计算这两个操作数。假设我们有一个函数`say()`，它会打印其介绍的参数，然后返回。 我们使用这个函数写出表达式：`say("left") - say("right");`。我们的解释器在报告运行时错误之前会先打印"left"和"right"。相对地，我们也可以指定在计算右操作数之前先检查左操作数。 ^11: 同样，我们要处理这种数字的边界情况，以确保jlox和clox的工作方式相同。像这样处理语言的一个奇怪的边界可能会让你抓狂，但这是工作的一个重要部分。用户会有意或无意地依赖于这些细节，如果实现不一致，他们的程序在不同的解释器上运行时将会中断。 ^12: 如果用户正在运行REPL，则我们不必跟踪运行时错误。在错误被报告之后，我们只需要循环，让用户输入新的代码，然后继续执行。

CHALLENGES

习题

> 1、Allowing comparisons on types other than numbers could be useful. The operators might have a reasonable interpretation for strings. Even comparisons among mixed types, like `3 < "pancake"` could be handy to enable things like ordered collections of heterogeneous types. Or it could simply lead to bugs and confusion.
>
> Would you extend Lox to support comparing other types? If so, which pairs of types do you allow and how do you define their ordering? Justify your choices and compare them to other languages.

1、允许对数字之外的类型进行比较可能是个有用的特性。操作符可能对字符串有合理的解释。即使是混合类型之间的比较，如`3<"pancake"`，也可以方便地支持异构类型的有序集合。否则可能导致错误和混乱。

你是否会扩展Lox以支持对其他类型的比较？如果是，您允许哪些类型间的比较，以及如何定义它们的顺序？证明你的选择并与其他语言进行比较。

> 2、Many languages define `+` such that if *either* operand is a string, the other is converted to a string and the results are then concatenated. For example, `"scone" + 4` would yield `scone4`. Extend the

> code in `visitBinaryExpr()` to support that.

2、许多语言对+的定义是，如果其中一个操作数是字符串，另一个操作数就会被转换成字符串，然后将两个结果拼接起来。例如，`"scone "+4`的结果应该是`scone4`。扩展`visitBinaryExpr()`中的代码以支持该特性。

> 3、What happens right now if you divide a number by zero? What do you think should happen? Justify your choice. How do other languages you know handle division by zero, and why do they make the choices they do?
>
> Change the implementation in `visitBinaryExpr()` to detect and report a runtime error for this case.

3、如果你用一个数除以0会发生什么？你认为应该发生什么？证明你的选择。你知道的其他语言是如何处理除零的，为什么他们会做出这样的选择？

更改`visitBinaryExpr()`中的实现代码，以检测并报告运行时错误。

DESIGN NOTE: STATIC AND DYNAMIC TYPING

设计笔记：静态类型和动态类型

> Some languages, like Java, are statically typed which means type errors are detected and reported at compile time before any code is run. Others, like Lox, are dynamically typed and defer checking for type errors until runtime right before an operation is attempted. We tend to consider this a black-and-white choice, but there is actually a continuum between them.
>
> It turns out even most statically typed languages do *some* type checks at runtime. The type system checks most type rules statically, but inserts runtime checks in the generated code for other operations.
>
> For example, in Java, the *static* type system assumes a cast expression will always safely succeed. After you cast some value, you can statically treat it as the destination type and not get any compile errors. But downcasts can fail, obviously. The only reason the static checker can presume that casts always succeed without violating the language's soundness guarantees, is because the cast is checked *at runtime* and throws an exception on failure.
>
> A more subtle example is covariant arrays in Java and C#. The static subtyping rules for arrays allow operations that are not sound. Consider:
>
> ```
> Object[] stuff = new Integer[1];
> stuff[0] = "not an int!";
> ```
>
> This code compiles without any errors. The first line upcasts the Integer array and stores it in a variable of type Object array. The second line stores a string in one of its cells. The Object array type statically allows that—strings *are* Objects—but the actual Integer array that `stuff` refers to at runtime should never have a string in it! To avoid that catastrophe, when you store a value in an array, the JVM does a *runtime* check to make sure it's an allowed type. If not, it throws an ArrayStoreException.
>
> Java could have avoided the need to check this at runtime by disallowing the cast on the first line. It could make arrays *invariant* such that an array of Integers is *not* an array of Objects. That's statically sound, but it prohibits common and safe patterns of code that only read from arrays. Covariance is safe

> if you never *write* to the array. Those patterns were particularly important for usability in Java 1.0 before it supported generics.
>
> James Gosling and the other Java designers traded off a little static safety and performance—those array store checks take time—in return for some flexibility.
>
> There are few modern statically typed languages that don’t make that trade-off *somewhere*. Even Haskell will let you run code with non-exhaustive matches. If you find yourself designing a statically typed language, keep in mind that you can sometimes give users more flexibility without sacrificing *too* many of the benefits of static safety by deferring some type checks until runtime.
>
> On the other hand, a key reason users choose statically typed languages is because of the confidence the language gives them that certain kinds of errors can *never* occur when their program is run. Defer too many type checks until runtime, and you erode that confidence.

有些语言，如Java，是静态类型的，这意味着在任何代码运行之前，会在编译时检测和报告类型错误。其他语言，如Lox，是动态类型的，将类型错误的检查推迟到运行时尝试执行具体操作之前。我们倾向于认为这是一个非黑即白的选择，但实际上它们之间是连续统一的。

事实证明，大多数静态类型的语言也会在运行时进行一些类型检查。类型系统会静态地检查多数类型规则，但在生成的代码中插入了运行时检查以支持其它操作。

例如，在Java中，静态类型系统会假定强制转换表达式总是能安全地成功执行。在转换某个值之后，可以将其静态地视为目标类型，而不会出现任何编译错误。但向下转换显然会失败。静态检查器之所以能够在不违反语言的合理性保证的情况下假定转换总是成功的，唯一原因是，强制转换操作会在运行时进行类型检查，并在失败时抛出异常。

一个更微妙的例子是Java和c#中的协变数组。数组的静态子类型规则允许不健全的操作。考虑以下代码：

```
Object[] stuff = new Integer[1];
stuff[0] = "not an int!";
```

这段代码在编译时没有任何错误。第一行代码将整数数组向上转换并存储到一个对象数组类型的变量中。第二行代码将字符串存储在其中一个单元格里。对象数组类型静态地允许该操作——字符串也是对象——但是`stuff`在运行时引用的整数数组中不应该包含字符串！为了避免这种灾难，当你在数组中存储一个值时，JVM会进行运行时检查，以确保该值是允许的类型。如果不是，则抛出ArrayStoreException。

Java可以通过禁止对第一行进行强制转换来避免在运行时检查这一点。它可以使数组保持不变，这样整型数组就不是对象数组。这在静态类型角度是合理的，但它禁止了只从数组中读取数据的常见安全的代码模式。如果你从来不向数组写入内容，那么协变是安全的。在支持泛型之前，这些模式对于Java 1.0的可用性尤为重要。

James Gosling和其他Java设计师牺牲了一点静态安全和性能（这些数组存储检查需要花费时间）来换取一些灵活性。

几乎所有的现代静态类型语言都在某些方面做出了权衡。即使Haskell也允许您运行非穷举性匹配的代码。如果您自己正在设计一种静态类型语言，请记住，有时你可以通过将一些类型检查推迟到运行时来给用户更多的灵活性，而不会牺牲静态安全的太多好处。

另一方面，用户选择静态类型语言的一个关键原因是，这种语言让他们相信：在他们的程序运行时，某些类型的错误永远不会发生。将过多的类型检查推迟到运行时，就会破坏用户的这种信心。

8.表达式和状态 Statements and State

> *All my life, my heart has yearned for a thing I cannot name.*
>
> —— André Breton, *Mad Love*

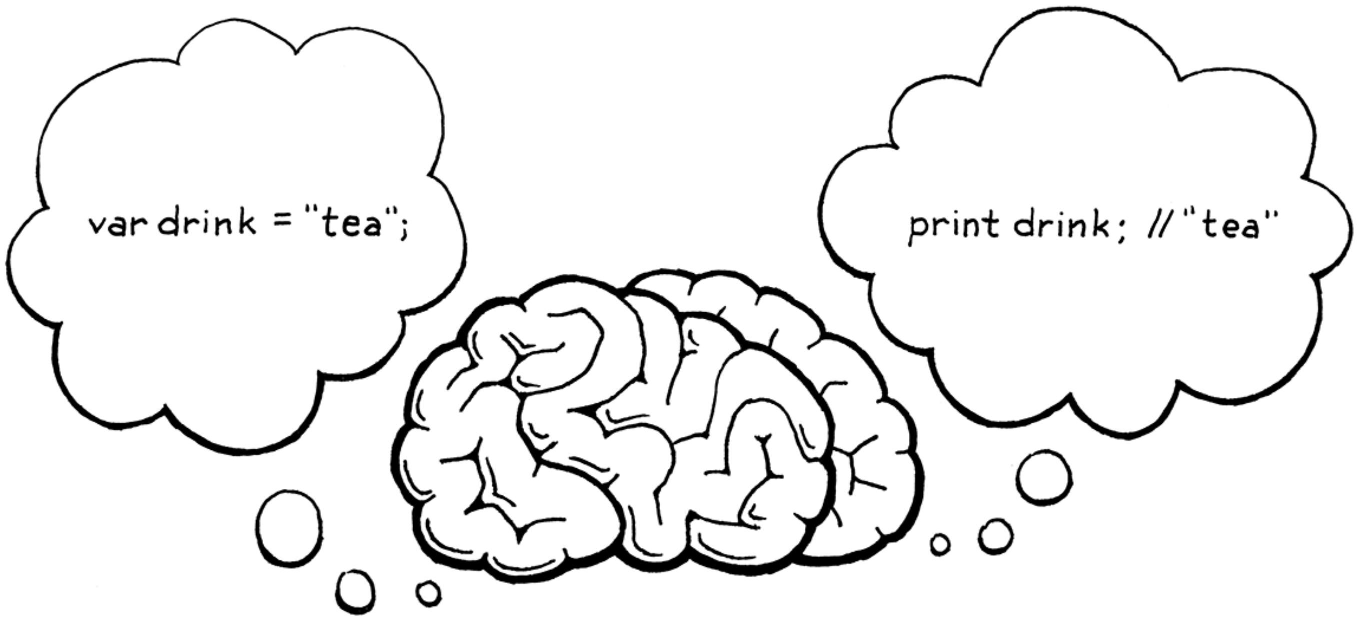
终我一生，我们的内心都在渴求一种我无法名状的东西。

> The interpreter we have so far feels less like programming a real language and more like punching buttons on a calculator. “Programming” to me means building up a system out of smaller pieces. We can’t do that yet because we have no way to bind a name to some data or function. We can’t compose software without a way to refer to the pieces.

到目前为止，我们提供解释器的感觉不太像是在使用一种真正的语言进行编程，更像是在计算器上按按钮。对我来说，"编程 "意味着用较小的部分构建出一个系统。我们目前还不支持这样做，因为我们还无法将一个名称绑定到某个数据或函数。我们不能在无法引用小片段的情况下编写软件。

> To support bindings, our interpreter needs internal state. When you define a variable at the beginning of the program and use it at the end, the interpreter has to hold on to the value of that variable in the meantime. So in this chapter, we will give our interpreter a brain that can not just process, but *remember*.

为了支持绑定，我们的解释器需要保存内部状态。如果你在程序开始处定义了一个变量，并在结束处使用它，那么解释器必须在这期间保持该变量的值。所以在这一章中，我们会给解释器一个大脑，它不仅可以运算，而且可以*记忆*。



> State and statements go hand in hand. Since statements, by definition, don’t evaluate to a value, they need to do something else to be useful. That something is called a **side effect**. It could mean producing user-visible output or modifying some state in the interpreter that can be detected later. The latter makes them a great fit for defining variables or other named entities.

状态和语句是相辅相成的。因为根据定义，语句不会计算出一个具体值，而是需要做一些事情来发挥作用。这些事情被称为**副作用(side effect)**。它可能意味着产生用户可见的输出，或者修改解释器中的一些状态，而这些状态后续可以被检测到。第二个特性使得语句非常适合于定义变量或其他命名实体。

> In this chapter, we'll do all of that. We'll define statements that produce output (`print`) and create state (`var`). We'll add expressions to access and assign to variables. Finally, we'll add blocks and local scope. That's a lot to stuff into one chapter, but we'll chew through it all one bite at a time.

在这一章中，我们会实现所有这些。我们会定义可以产生输出和创建状态的语句，然后会添加表达式来访问和赋值给这些变量，最后，我们会引入代码块和局部作用域。这一章要讲的内容太多了，但是我们会一点一点地把它们嚼碎。

8.1 Statements

8.1 语句

> We start by extending Lox's grammar with statements. They aren't very different from expressions. We start with the two simplest kinds:

我们首先扩展Lox的语法以支持语句。 语句与表达式并没有很大的不同，我们从两种最简单的类型开始：

1. > An **expression statement** lets you place an expression where a statement is expected. They exist to evaluate expressions that have side effects. You may not notice them, but you use them all the time in C, Java, and other languages. Any time you see a function or method call followed by a `;`, you're looking at an expression statement.

   表达式语句可以让您将表达式放在需要语句的位置。它们的存在是为了计算有副作用的表达式。您可能没有注意到它们，但其实你在C、Java和其他语言中一直在使用表达式语句^1。如果你看到一个函数或方法调用后面跟着一个`;`，您看到的其实就是一个表达式语句。

2. > A **`print` statement** evaluates an expression and displays the result to the user. I admit it's weird to bake printing right into the language instead of making it a library function. Doing so is a concession to the fact that we're building this interpreter one chapter at a time and want to be able to play with it before it's all done. To make print a library function, we'd have to wait until we had all of the machinery for defining and calling functions before we could witness any side effects.

   `print`语句会计算一个表达式，并将结果展示给用户。我承认把`print`直接放进语言中，而不是把它变成一个库函数，这很奇怪^2。这样做是基于本书的编排策略的让步，即我们会以章节为单位逐步构建这个解释器，并希望能够在完成解释器的所有功能之前能够使用它。如果让`print`成为一个标准库函数，我们必须等到拥有了定义和调用函数的所有机制之后，才能看到它发挥作用。

> New syntax means new grammar rules. In this chapter, we finally gain the ability to parse an entire Lox script. Since Lox is an imperative, dynamically typed language, the "top level" of a script is simply a list of statements. The new rules are:

新的词法意味着新的语法规则。在本章中，我们终于获得了解析整个Lox脚本的能力。由于Lox是一种命令式的、动态类型的语言，所以脚本的"顶层"也只是一组语句。新的规则如下：

```
program        → statement* EOF ;

statement      → exprStmt
               | printStmt ;

exprStmt       → expression ";" ;
printStmt      → "print" expression ";" ;
```

> The first rule is now `program`, which is the starting point for the grammar and represents a complete Lox script or REPL entry. A program is a list of statements followed by the special “end of file” token. The mandatory end token ensures the parser consumes the entire input and doesn’t silently ignore erroneous unconsumed tokens at the end of a script.

现在第一条规则是`program`，这也是语法的起点，代表一个完整的Lox脚本或REPL输入项。程序是一个语句列表，后面跟着特殊的“文件结束”(EOF)标记。强制性的结束标记可以确保解析器能够消费所有输入内容，而不会默默地忽略脚本结尾处错误的、未消耗的标记。

> Right now, `statement` only has two cases for the two kinds of statements we’ve described. We’ll fill in more later in this chapter and in the following ones. The next step is turning this grammar into something we can store in memory—syntax trees.

目前，`statement`只有两种情况，分别对应于我们描述的两类语句。我们将在本章后面和接下来的章节中补充更多内容。接下来就是将这个语法转化为我们可以存储在内存中的东西——语法树。。

> ### 8.1.1 Statement syntax trees

8.1.1 Statement语法树

> There is no place in the grammar where both an expression and a statement are allowed. The operands of, say, `+` are always expressions, never statements. The body of a `while` loop is always a statement.

语法中没有地方既允许使用表达式，也允许使用语句。 操作符（如`+`）的操作数总是表达式，而不是语句。`while`循环的主体总是一个语句。

> Since the two syntaxes are disjoint, we don’t need a single base class that they all inherit from. Splitting expressions and statements into separate class hierarchies enables the Java compiler to help us find dumb mistakes like passing a statement to a Java method that expects an expression.

因为这两种语法是不相干的，所以我们不需要提供一个它们都继承的基类。将表达式和语句拆分为单独的类结构，可使Java编译器帮助我们发现一些愚蠢的错误，例如将语句传递给需要表达式的Java方法。

> That means a new base class for statements. As our elders did before us, we will use the cryptic name “Stmt”. With great foresight, I have designed our little AST metaprogramming script in anticipation of this. That’s why we passed in “Expr” as a parameter to `defineAst()`. Now we add another call to define Stmt and its subclasses.

这意味着要为语句创建一个新的基类。正如我们的前辈那样，我们将使用“Stmt”这个隐秘的名字。我很有远见，在设计我们的AST元编程脚本时就已经预见到了这一点。这就是为什么我们把“Expr”作为参数传给了`defineAst()`。现在我们添加另一个方法调用来定义`Stmt`和它的子类。

tool/GenerateAst.java，在 main()方法中新增：

```
      "Unary    : Token operator, Expr right"
    ));
    // 新增部分开始
    defineAst(outputDir, "Stmt", Arrays.asList(
      "Expression : Expr expression",
      "Print      : Expr expression"
    ));
    // 新增部分结束
  }
```

新节点对应的生成代码可以参考附录： Appendix II: Expression statement, Print statement.

> Run the AST generator script and behold the resulting "Stmt.java" file with the syntax tree classes we need for expression and `print` statements. Don't forget to add the file to your IDE project or makefile or whatever.

运行AST生成器脚本，查看生成的`Stmt.java`文件，其中包含表达式和`print`语句所需的语法树类。不要忘记将该文件添加到IDE项目或makefile或其他文件中。

> ### 8.1.2 Parsing statements

8.1.2 解析语句

> The parser's `parse()` method that parses and returns a single expression was a temporary hack to get the last chapter up and running. Now that our grammar has the correct starting rule, `program`, we can turn `parse()` into the real deal.

解析器的`parse()`方法会解析并返回一个表达式，这是一个临时方案，是为了让上一章的代码能启动并运行起来。现在，我们的语法已经有了正确的起始规则，即`program`，我们可以正式编写`parse()`方法了。

lox/Parser.java，parse()方法，替换7行：

```
  List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      statements.add(statement());
    }

    return statements;
  }
```

> This parses a series of statements, as many as it can find until it hits the end of the input. This is a pretty direct translation of the `program` rule into recursive descent style. We must also chant a minor prayer to the Java verbosity gods since we are using ArrayList now.

该方法会尽可能多地解析一系列语句，直到命中输入内容的结尾为止。这是一种非常直接的将`program`规则转换为递归下降风格的方式。由于我们现在使用ArrayList，所以我们还必须向Java的冗长之神做一个小小的祈祷。

lox/Parser.java，新增代码：

```
package com.craftinginterpreters.lox;
// 新增部分开始
import java.util.ArrayList;
// 新增部分结束
import java.util.List;
```

> A program is a list of statements, and we parse one of those statements using this method:

一个程序就是一系列的语句，而我们可以通过下面的方法解析每一条语句：

lox/Parser.java，在 expression()方法后添加：

```
  private Stmt statement() {
    if (match(PRINT)) return printStatement();

    return expressionStatement();
  }
```

> A little bare bones, but we'll fill it in with more statement types later. We determine which specific statement rule is matched by looking at the current token. A `print` token means it's obviously a `print` statement.

这是一个简单的框架，但是稍后我们将会填充更多的语句类型。我们通过查看当前标记来确定匹配哪条语句规则。`print`标记意味着它显然是一个`print`语句。

> If the next token doesn't look like any known kind of statement, we assume it must be an expression statement. That's the typical final fallthrough case when parsing a statement, since it's hard to proactively recognize an expression from its first token.

如果下一个标记看起来不像任何已知类型的语句，我们就认为它一定是一个表达式语句。这是解析语句时典型的最终失败分支，因为我们很难通过第一个标记主动识别出一个表达式。

> Each statement kind gets its own method. First `print`:

每种语句类型都有自己的方法。首先是`print`：

lox/Parser.java，在 statement()方法后添加：

```
  private Stmt printStatement() {
    Expr value = expression();
    consume(SEMICOLON, "Expect ';' after value.");
    return new Stmt.Print(value);
  }
```

> Since we already matched and consumed the `print` token itself, we don’t need to do that here. We parse the subsequent expression, consume the terminating semicolon, and emit the syntax tree.

因为我们已经匹配并消费了`print`标记本身，所以这里不需要重复消费。我们先解析随后的表达式，消费表示语句终止的分号，并生成语法树。

> If we didn’t match a `print` statement, we must have one of these:

如果我们没有匹配到`print`语句，那一定是一条下面的语句：

lox/Parser.java，在 printStatement()方法后添加：

```
  private Stmt expressionStatement() {
    Expr expr = expression();
    consume(SEMICOLON, "Expect ';' after expression.");
    return new Stmt.Expression(expr);
  }
```

> Similar to the previous method, we parse an expression followed by a semicolon. We wrap that Expr in a Stmt of the right type and return it.

与前面的方法类似，我们解析一个后面带分号的表达式。我们将Expr封装在一个正确类型的Stmt中，并返回它。

> ### 8.1.3 Executing statements

8.1.3 执行语句

> We’re running through the previous couple of chapters in microcosm, working our way through the front end. Our parser can now produce statement syntax trees, so the next and final step is to interpret them. As in expressions, we use the Visitor pattern, but we have a new visitor interface, Stmt.Visitor, to implement since statements have their own base class.

我们在前面几章一步一步地慢慢完成了解释器的前端工作。我们的解析器现在可以产生语句语法树，所以下一步，也是最后一步，就是对其进行解释。和表达式一样，我们使用的是Visitor模式，但是我们需要实现一个新的访问者接口`Stmt.Visitor`，因为语句有自己的基类。

> We add that to the list of interfaces Interpreter implements.

我们将其添加到Interpreter实现的接口列表中。

lox/Interpreter.java，替换1行^3：

```
// 替换部分开始
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
```

```
// 替换部分结束
  void interpret(Expr expression) {
```

> Unlike expressions, statements produce no values, so the return type of the visit methods is Void, not Object. We have two statement types, and we need a visit method for each. The easiest is expression statements.

与表达式不同，语句不会产生值，因此visit方法的返回类型是`Void`，而不是`Object`。我们有两种语句类型，每种类型都需要一个visit方法。最简单的是表达式语句：

lox/Interpreter.java，在 evaluate()方法后添加：

```
  @Override
  public Void visitExpressionStmt(Stmt.Expression stmt) {
    evaluate(stmt.expression);
    return null;
  }
```

> We evaluate the inner expression using our existing `evaluate()` method and discard the value. Then we return `null`. Java requires that to satisfy the special capitalized Void return type. Weird, but what can you do?

我们使用现有的`evaluate()`方法计算内部表达式，并丢弃其结果值。然后我们返回`null`，因为Java要求为特殊的大写Void返回类型返回该值。很奇怪，但你能有什么办法呢？

> The `print` statement's visit method isn't much different.

`print`语句的visit方法没有太大的不同。

lox/Interpreter.java，在 visitExpressionStmt()方法后添加：

```
  @Override
  public Void visitPrintStmt(Stmt.Print stmt) {
    Object value = evaluate(stmt.expression);
    System.out.println(stringify(value));
    return null;
  }
```

> Before discarding the expression's value, we convert it to a string using the `stringify()` method we introduced in the last chapter and then dump it to stdout.

在丢弃表达式的值之前，我们使用上一章引入的`stringify()`方法将其转换为字符串，然后将其输出到stdout。

> Our interpreter is able to visit statements now, but we have some work to do to feed them to it. First, modify the old `interpret()` method in the Interpreter class to accept a list of statements—in other words, a program.

我们的解释器现在可以处理语句了，但是我们还需要做一些工作将语句输入到解释器中。首先，修改Interpreter类中原有的`interpret()` 方法，让其能够接受一组语句——即一段程序。

lox/Interpreter.java，修改 interpret()方法，替换8行：

```
  void interpret(List<Stmt> statements) {
    try {
      for (Stmt statement : statements) {
        execute(statement);
      }
    } catch (RuntimeError error) {
      Lox.runtimeError(error);
    }
  }
```

> This replaces the old code which took a single expression. The new code relies on this tiny helper method:

这段代码替换了原先处理单个表达式的旧代码。新代码依赖于下面的小辅助方法。

lox/Interpreter.java，在 evaluate()方法后添加：

```
  private void execute(Stmt stmt) {
    stmt.accept(this);
  }
```

> That’s the statement analogue to the `evaluate()` method we have for expressions. Since we’re working with lists now, we need to let Java know.

这类似于处理表达式的`evaluate()`方法，这是这里处理语句。因为我们要使用列表，所以我们需要在Java中引入一下。

lox/Interpreter.java

```
package com.craftinginterpreters.lox;
// 新增部分开始
import java.util.List;
// 新增部分结束
class Interpreter implements Expr.Visitor<Object>,
```

> The main Lox class is still trying to parse a single expression and pass it to the interpreter. We fix the parsing line like so:

Lox主类中仍然是只解析单个表达式并将其传给解释器。我们将其修正如下：

lox/Lox.java，在 run()方法中替换一行：

```
    Parser parser = new Parser(tokens);
    // 替换部分开始
    List<Stmt> statements = parser.parse();
    // 替换部分结束
    // Stop if there was a syntax error.
```

> And then replace the call to the interpreter with this:

然后将对解释器的调用替换如下：

lox/Lox.java，在 run()方法中替换一行：

```
    if (hadError) return;
    // 替换部分开始
    interpreter.interpret(statements);
    // 替换部分结束
  }
```

> Basically just plumbing the new syntax through. OK, fire up the interpreter and give it a try. At this point, it's worth sketching out a little Lox program in a text file to run as a script. Something like:

基本就是对新语法进行遍历。 OK，启动解释器并测试一下。 现在有必要在文本文件中草拟一个小的Lox程序来作为脚本运行。 就像是：

```
print "one";
print true;
print 2 + 1;
```

> It almost looks like a real program! Note that the REPL, too, now requires you to enter a full statement instead of a simple expression. Don't forget your semicolons.

它看起来就像一个真实的程序！ 请注意，REPL现在也要求你输入完整的语句，而不是简单的表达式。 所以不要忘记后面的分号。

8.2 Global Variables

8.2 全局变量

> Now that we have statements, we can start working on state. Before we get into all of the complexity of lexical scoping, we'll start off with the easiest kind of variables—globals. We need two new constructs.

现在我们已经有了语句，可以开始处理状态了。在深入探讨语法作用域的复杂性之前，我们先从最简单的变量（全局变量）开始^4。我们需要两个新的结构。

1. > A **variable declaration** statement brings a new variable into the world.

   变量声明语句用于创建一个新变量。

```
var beverage = "espresso";
```

> This creates a new binding that associates a name (here “beverage”) with a value (here, the string `"espresso"`).

该语句将创建一个新的绑定，将一个名称（这里是 `beverage`）和一个值（这里是字符串 `"espresso"`）关联起来。

2. > Once that’s done, a **variable expression** accesses that binding. When the identifier “beverage” is used as an expression, it looks up the value bound to that name and returns it.

   一旦声明完成，**变量表达式**就可以访问该绑定。当标识符“beverage”被用作一个表达式时，程序会查找与该名称绑定的值并返回。

   ```
   print beverage; // "espresso".
   ```

> Later, we’ll add assignment and block scope, but that’s enough to get moving.

稍后，我们会添加赋值和块作用域，但是这些已经足够继续后面的学习了。

> ## 8.2.1 Variable syntax

8.2.1 变量语法

> As before, we’ll work through the implementation from front to back, starting with the syntax. Variable declarations are statements, but they are different from other statements, and we’re going to split the statement grammar in two to handle them. That’s because the grammar restricts where some kinds of statements are allowed.

与前面一样，我们将从语法开始，从前到后依次完成实现。变量声明是一种语句，但它们不同于其他语句，我们把statement语法一分为二来处理该情况。这是因为语法要限制某个位置上哪些类型的语句是被允许的。

> The clauses in control flow statements—think the then and else branches of an `if` statement or the body of a `while`—are each a single statement. But that statement is not allowed to be one that declares a name. This is OK:

控制流语句中的子句——比如，`if`或`while`语句体中的`then`和`else`分支——都是一个语句。但是这个语句不应该是一个声明名称的语句。下面的代码是OK的：

```
if (monday) print "Ugh, already?";
```

> But this is not:

但是下面的代码不行：

```
if (monday) var beverage = "espresso";
```

> We *could* allow the latter, but it's confusing. What is the scope of that `beverage` variable? Does it persist after the `if` statement? If so, what is its value on days other than Monday? Does the variable exist at all on those days?

我们也 *可以*允许后者，但是会令人困惑。 `beverage`变量的作用域是什么？`if`语句结束之后它是否还继续存在？如果存在的话，在其它条件下它的值是什么？这个变量是否在其它情形下也一直存在？

> Code like this is weird, so C, Java, and friends all disallow it. It's as if there are two levels of "precedence" for statements. Some places where a statement is allowed—like inside a block or at the top level—allow any kind of statement, including declarations. Others allow only the "higher" precedence statements that don't declare names.

这样的代码有点奇怪，所以C、Java及类似语言中都不允许这种写法。语句就好像有两个“优先级”。有些允许语句的地方——比如在代码块内或程序顶层^5——可以允许任何类型的语句，包括变量声明。而其他地方只允许那些不声明名称的、优先级更高的语句。

> To accommodate the distinction, we add another rule for kinds of statements that declare names.

为了适应这种区别，我们为声明名称的语句类型添加了另一条规则：

```
program        → declaration* EOF ;

declaration    → varDecl
               | statement ;

statement      → exprStmt
               | printStmt ;
```

> Declaration statements go under the new `declaration` rule. Right now, it's only variables, but later it will include functions and classes. Any place where a declaration is allowed also allows non-declaring statements, so the `declaration` rule falls through to `statement`. Obviously, you can declare stuff at the top level of a script, so `program` routes to the new rule.

声明语句属于新的 `declaration`规则。目前，这里只有变量，但是后面还会包含函数和类。任何允许声明的地方都允许一个非声明式的语句，所以 `declaration` 规则会下降到`statement`。显然，你可以在脚本的顶层声明一些内容，所以`program`规则需要路由到新规则。

> The rule for declaring a variable looks like:

声明一个变量的规则如下：

```
varDecl        → "var" IDENTIFIER ( "=" expression )? ";" ;
```

> Like most statements, it starts with a leading keyword. In this case, `var`. Then an identifier token for the name of the variable being declared, followed by an optional initializer expression. Finally, we put a bow on it with the semicolon.

像大多数语句一样，它以一个前置关键字开头，这里是`var`。然后是一个标识符标记，作为声明变量的名称，后面是一个可选的初始化式表达式。最后，以一个分号作为结尾。

> To access a variable, we define a new kind of primary expression.

为了访问变量，我们还需要定义一个新类型的基本表达式：

```
primary        → "true" | "false" | "nil"
               | NUMBER | STRING
               | "(" expression ")"
               | IDENTIFIER ;
```

> That `IDENTIFIER` clause matches a single identifier token, which is understood to be the name of the variable being accessed.

`IDENTIFIER` 子语句会匹配单个标识符标记，该标记会被理解为正在访问的变量的名称。

> These new grammar rules get their corresponding syntax trees. Over in the AST generator, we add a new statement tree for a variable declaration.

这些新的语法规则需要其相应的语法树。在AST生成器中，我们为变量声明添加一个新的语句树。

tool/GenerateAst.java，在 main()方法中添加一行，前一行需要加`,`：

```
      "Expression : Expr expression",
      "Print      : Expr expression",
      // 新增部分开始
      "Var        : Token name, Expr initializer"
      // 新增部分结束
    ));
```

> It stores the name token so we know what it’s declaring, along with the initializer expression. (If there isn’t an initializer, that field is `null`.)

这里存储了名称标记，以便我们知道该语句声明了什么，此外还有初始化表达式（如果没有，字段就是`null`）。

> Then we add an expression node for accessing a variable.

然后我们添加一个表达式节点用于访问变量。

tool/GenerateAst.java，在 main()方法中添加一行，前一行需要加`,`：

```
      "Literal  : Object value",
      "Unary    : Token operator, Expr right",
      // 新增部分开始
      "Variable : Token name"
      // 新增部分结束
    ));
```

> It's simply a wrapper around the token for the variable name. That's it. As always, don't forget to run the AST generator script so that you get updated "Expr.java" and "Stmt.java" files.

这只是对变量名称标记的简单包装，就是这样。像往常一样，别忘了运行AST生成器脚本，这样你就能得到更新的 "Expr.java "和 "Stmt.java "文件。

> 8.2.2 Parsing variables

8.2.2 解析变量

> Before we parse variable statements, we need to shift around some code to make room for the new `declaration` rule in the grammar. The top level of a program is now a list of declarations, so the entrypoint method to the parser changes.

在解析变量语句之前，我们需要修改一些代码，为语法中的新规则`declaration`腾出一些空间。现在，程序的最顶层是声明语句的列表，所以解析器方法的入口需要更改：

lox/Parser.java，在 parse()方法中替换1行：

```
    List<Stmt> parse() {
    List<Stmt> statements = new ArrayList<>();
    while (!isAtEnd()) {
      // 替换部分开始
      statements.add(declaration());
      // 替换部分结束
    }

    return statements;
  }
```

> That calls this new method:

这里会调用下面的新方法：

lox/Parser.java，在 expression()方法后添加：

```
  private Stmt declaration() {
    try {
      if (match(VAR)) return varDeclaration();

      return statement();
    } catch (ParseError error) {
      synchronize();
      return null;
    }
  }
```

> Hey, do you remember way back in that earlier chapter when we put the infrastructure in place to do error recovery? We are finally ready to hook that up.

你还记得前面的章节中，我们建立了一个进行错误恢复的框架吗？现在我们终于可以用起来了。

> This `declaration()` method is the method we call repeatedly when parsing a series of statements in a block or a script, so it's the right place to synchronize when the parser goes into panic mode. The whole body of this method is wrapped in a try block to catch the exception thrown when the parser begins error recovery. This gets it back to trying to parse the beginning of the next statement or declaration.

当我们解析块或脚本中的 一系列语句时， `declaration()` 方法会被重复调用。因此当解析器进入恐慌模式时，它就是进行同步的正确位置。该方法的整个主体都封装在一个try块中，以捕获解析器开始错误恢复时抛出的异常。这样可以让解析器跳转到解析下一个语句或声明的开头。

> The real parsing happens inside the try block. First, it looks to see if we're at a variable declaration by looking for the leading `var` keyword. If not, it falls through to the existing `statement()` method that parses `print` and expression statements.

真正的解析工作发生在try块中。首先，它通过查找前面的`var`关键字判断是否是变量声明语句。如果不是的话，就会进入已有的`statement()`方法中，解析`print`和语句表达式。

> Remember how `statement()` tries to parse an expression statement if no other statement matches? And `expression()` reports a syntax error if it can't parse an expression at the current token? That chain of calls ensures we report an error if a valid declaration or statement isn't parsed.

还记得 `statement()` 会在没有其它语句匹配时会尝试解析一个表达式语句吗？而`expression()`如果无法在当前语法标记处解析表达式，则会抛出一个语法错误？这一系列调用链可以保证在解析无效的声明或语句时会报告错误。

> When the parser matches a `var` token, it branches to:

当解析器匹配到一个`var`标记时，它会跳转到：

lox/Parser.java，在 printStatement()方法后添加：

```
  private Stmt varDeclaration() {
    Token name = consume(IDENTIFIER, "Expect variable name.");

    Expr initializer = null;
    if (match(EQUAL)) {
      initializer = expression();
    }
    consume(SEMICOLON, "Expect ';' after variable declaration.");
    return new Stmt.Var(name, initializer);
  }
```

> As always, the recursive descent code follows the grammar rule. The parser has already matched the `var` token, so next it requires and consumes an identifier token for the variable name.

与之前一样，递归下降代码会遵循语法规则。解析器已经匹配了`var`标记，所以接下来要消费一个标识符标记作为变量的名称。

> Then, if it sees an `=` token, it knows there is an initializer expression and parses it. Otherwise, it leaves the initializer `null`. Finally, it consumes the required semicolon at the end of the statement. All this gets wrapped in a Stmt.Var syntax tree node and we're groovy.

然后，如果找到`=`标记，解析器就知道后面有一个初始化表达式，并对其进行解析。否则，它会将初始器保持为`null`。最后，会消费语句末尾所需的分号。然后将所有这些都封装到一个Stmt.Var语法树节点中。

> Parsing a variable expression is even easier. In `primary()`, we look for an identifier token.

解析变量表达式甚至更简单。在`primary()`中，我们需要查找一个标识符标记。

lox/Parser.java，在 primary()方法中添加：

```
      return new Expr.Literal(previous().literal);
    }
    // 新增部分开始
    if (match(IDENTIFIER)) {
      return new Expr.Variable(previous());
    }
    // 新增部分结束
    if (match(LEFT_PAREN)) {
```

> That gives us a working front end for declaring and using variables. All that's left is to feed it into the interpreter. Before we get to that, we need to talk about where variables live in memory.

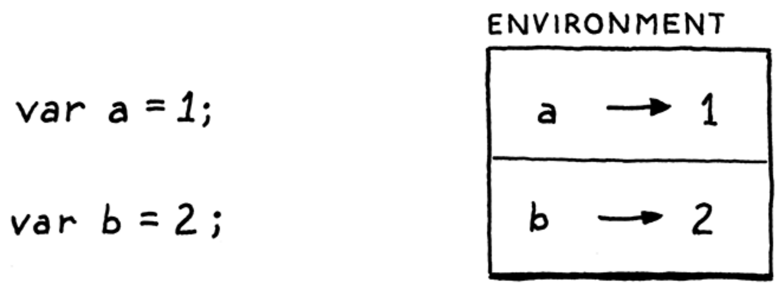
这为我们提供了声明和使用变量的可用前端，剩下的就是将其接入解释器中。在此之前，我们需要讨论变量在内存中的位置。

> ## 8.3 Environments

8.3 环境

> The bindings that associate variables to values need to be stored somewhere. Ever since the Lisp folks invented parentheses, this data structure has been called an **environment**.

变量与值之间的绑定关系需要保存在某个地方。自从Lisp发明圆括号以来，这种数据结构就被称为**环境。**



> You can think of it like a map where the keys are variable names and the values are the variable's, uh, values. In fact, that's how we'll implement it in Java. We could stuff that map and the code to manage it right into Interpreter, but since it forms a nicely delineated concept, we'll pull it out into its own class.

你可以把它想象成一个映射，其中键是变量名称，值就是变量的值^6。实际上，这也就是我们在Java中采用的实现方式。我们可以直接在解释器中加入该映射及其管理代码，但是因为它形成了一个很好的概念，我们可以将其提取到单独的类中。

> Start a new file and add:

打开新文件，添加以下代码：

lox/Environment.java，创建新文件

```java
package com.craftinginterpreters.lox;

import java.util.HashMap;
import java.util.Map;

class Environment {
  private final Map<String, Object> values = new HashMap<>();
}
```

> There's a Java Map in there to store the bindings. It uses bare strings for the keys, not tokens. A token represents a unit of code at a specific place in the source text, but when it comes to looking up variables, all identifier tokens with the same name should refer to the same variable (ignoring scope for now). Using the raw string ensures all of those tokens refer to the same map key.

其中使用一个Java Map来保存绑定关系。这里使用原生字符串作为键，而不是使用标记。一个标记表示源文本中特定位置的一个代码单元，但是在查找变量时，具有相同名称的标识符标记都应该指向相同的变量（暂时忽略作用域）。使用原生字符串可以保证所有这些标记都会指向相同的映射键。

> There are two operations we need to support. First, a variable definition binds a new name to a value.

我们需要支持两个操作。首先，是变量定义操作，可以将一个新的名称与一个值进行绑定。

lox/Environment.java，在 Environment类中添加：

```java
  void define(String name, Object value) {
    values.put(name, value);
  }
```

> Not exactly brain surgery, but we have made one interesting semantic choice. When we add the key to the map, we don’t check to see if it’s already present. That means that this program works:

不算困难，但是我们这里也做出了一个有趣的语义抉择。当我们向映射中添加键时，没有检查该键是否已存在。这意味着下面的代码是有效的：

```
var a = "before";
print a; // "before".
var a = "after";
print a; // "after".
```

> A variable statement doesn’t just define a *new* variable, it can also be used to *re*define an existing variable. We could choose to make this an error instead. The user may not intend to redefine an existing variable. (If they did mean to, they probably would have used assignment, not `var`.) Making redefinition an error would help them find that bug.

变量语句不仅可以定义一个新变量，也可以用于重新定义一个已有的变量。我们可以选择将其作为一个错误来处理。用户可能不打算重新定义已有的变量（如果他们想这样做，可能会使用赋值，而不是`var`），将重定义作为错误可以帮助用户发现这个问题。

> However, doing so interacts poorly with the REPL. In the middle of a REPL session, it’s nice to not have to mentally track which variables you’ve already defined. We could allow redefinition in the REPL but not in scripts, but then users would have to learn two sets of rules, and code copied and pasted from one form to the other might not work.

然而，这样做与REPL的交互很差。在与REPL的交互中，最好是让用户不必在脑子记录已经定义了哪些变量。我们可以在REPL中允许重定义，在脚本中不允许。但是这样一来，用户就不得不学习两套规则，而且一种形式的代码复制粘贴到另一种形式后可能无法运行^7。

> So, to keep the two modes consistent, we’ll allow it—at least for global variables. Once a variable exists, we need a way to look it up.

所以，为了保证两种模式的统一，我们选择允许重定义——至少对于全局变量如此。一旦一个变量存在，我们就需要可以查找该变量的方法。

lox/Environment.java，在 Environment类中添加：

```
class Environment {
  private final Map<String, Object> values = new HashMap<>();
  // 新增部分开始
  Object get(Token name) {
    if (values.containsKey(name.lexeme)) {
      return values.get(name.lexeme);
    }

    throw new RuntimeError(name,
        "Undefined variable '" + name.lexeme + "'.");
  }
  // 新增部分结束
  void define(String name, Object value) {
```

> This is a little more semantically interesting. If the variable is found, it simply returns the value bound to it. But what if it's not? Again, we have a choice:

这在语义上更有趣一些。如果找到了这个变量，只需要返回与之绑定的值。但如果没有找到呢？我们又需要做一个选择：

- > Make it a syntax error.

  抛出语法错误

- > Make it a runtime error.

  抛出运行时错误

- > Allow it and return some default value like `nil`.

  允许该操作并返回默认值（如`nil`）

> Lox is pretty lax, but the last option is a little *too* permissive to me. Making it a syntax error—a compile-time error—seems like a smart choice. Using an undefined variable is a bug, and the sooner you detect the mistake, the better.

Lox是很宽松的，但最后一个选项对我来说有点过于宽松了。把它作为语法错误（一个编译时的错误）似乎是一个明智的选择。使用未定义的变量确实是一个错误，用户越早发现这个错误就越好。

> The problem is that *using* a variable isn't the same as *referring* to it. You can refer to a variable in a chunk of code without immediately evaluating it if that chunk of code is wrapped inside a function. If we make it a static error to *mention* a variable before it's been declared, it becomes much harder to define recursive functions.

问题在于，*使用*一个变量并不等同于*引用*它。如果代码块封装在函数中，则可以在代码块中引用变量，而不必立即对其求值。如果我们把引用未声明的变量当作一个静态错误，那么定义递归函数就变得更加困难了。

> We could accommodate single recursion—a function that calls itself—by declaring the function's own name before we examine its body. But that doesn't help with mutually recursive procedures that call each other. Consider:

通过在检查函数体之前先声明函数名称，我们可以支持单一递归——调用自身的函数。但是，这无法处理互相调用的递归程序^8。考虑以下代码：

```
fun isOdd(n) {
  if (n == 0) return false;
  return isEven(n - 1);
}

fun isEven(n) {
  if (n == 0) return true;
  return isOdd(n - 1);
}
```

> The `isEven()` function isn't defined by the time we are looking at the body of `isOdd()` where it's called. If we swap the order of the two functions, then `isOdd()` isn't defined when we're looking at `isEven()`'s body.

当我们查看`isOdd()`方法时， `isEven()` 方法被调用的时候还没有被声明。如果我们交换着两个函数的顺序，那么在查看`isEven()`方法体时会发现`isOdd()`方法未被定义^9。

> Since making it a *static* error makes recursive declarations too difficult, we'll defer the error to runtime. It's OK to refer to a variable before it's defined as long as you don't *evaluate* the reference. That lets the program for even and odd numbers work, but you'd get a runtime error in:

因为将其当作*静态*错误会使递归声明过于困难，因此我们把这个错误推迟到运行时。在一个变量被定义之前引用它是可以的，只要你不对引用进行*求值*。这样可以让前面的奇偶数代码正常工作。但是执行以下代码时，你会得到一个运行时错误：

```
print a;
var a = "too late!";
```

> As with type errors in the expression evaluation code, we report a runtime error by throwing an exception. The exception contains the variable's token so we can tell the user where in their code they messed up.

与表达式计算代码中的类型错误一样，我们通过抛出一个异常来报告运行时错误。异常中包含变量的标记，以便我们告诉用户代码的什么位置出现了错误。

> ## 8.3.1 Interpreting global variables

8.3.1 解释全局变量

> The Interpreter class gets an instance of the new Environment class.

Interpreter类会获取Environment类的一个实例。

lox/Interpreter.java，在 Interpreter类中添加：

```
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
  // 添加部分开始
  private Environment environment = new Environment();
  // 添加部分结束
  void interpret(List<Stmt> statements) {
```

> We store it as a field directly in Interpreter so that the variables stay in memory as long as the interpreter is still running.

我们直接将它作为一个字段存储在解释器中，这样，只要解释器仍在运行，变量就会留在内存中。

> We have two new syntax trees, so that's two new visit methods. The first is for declaration statements.

我们有两个新的语法树，所以这就是两个新的访问方法。第一个是关于声明语句的。

lox/Interpreter.java，在 visitPrintStmt()方法后添加：

```
  @Override
  public Void visitVarStmt(Stmt.Var stmt) {
    Object value = null;
    if (stmt.initializer != null) {
      value = evaluate(stmt.initializer);
    }

    environment.define(stmt.name.lexeme, value);
    return null;
  }
```

> If the variable has an initializer, we evaluate it. If not, we have another choice to make. We could have made this a syntax error in the parser by *requiring* an initializer. Most languages don't, though, so it feels a little harsh to do so in Lox.

如果该变量有初始化式，我们就对其求值。如果没有，我们就需要做一个选择。我们可以通过在解析器中*要求*初始化式令其成为一个语法错误。但是，大多数语言都不会这么做，所以在Lox中这样做感觉有点苛刻。

> We could make it a runtime error. We'd let you define an uninitialized variable, but if you accessed it before assigning to it, a runtime error would occur. It's not a bad idea, but most dynamically typed languages don't do that. Instead, we'll keep it simple and say that Lox sets a variable to `nil` if it isn't explicitly initialized.

我们可以使其成为运行时错误。我们允许您定义一个未初始化的变量，但如果您在对其赋值之前访问它，就会发生运行时错误。这不是一个坏主意，但是大多数动态类型的语言都不会这样做。相反，我们使用最简单的方式。或者说，如果变量没有被显式初始化，Lox会将变量设置为`nil`。

```
var a;
print a; // "nil".
```

> Thus, if there isn't an initializer, we set the value to `null`, which is the Java representation of Lox's `nil` value. Then we tell the environment to bind the variable to that value.

因此，如果没有初始化式，我们将值设为`null`，这也是Lox中的`nil`值的Java表示形式。然后，我们告诉环境上下文将变量与该值进行绑定。

> Next, we evaluate a variable expression.

接下来，我们要对变量表达式求值。

lox/Interpreter.java，在 visitUnaryExpr()方法后添加：

```
  @Override
  public Object visitVariableExpr(Expr.Variable expr) {
    return environment.get(expr.name);
  }
```

> This simply forwards to the environment which does the heavy lifting to make sure the variable is defined. With that, we've got rudimentary variables working. Try this out:

这里只是简单地将操作转发到环境上下文中，环境做了一些繁重的工作保证变量已被定义。这样，我们就可以支持基本的变量操作了。尝试以下代码：

```
var a = 1;
var b = 2;
print a + b;
```

> We can't reuse *code* yet, but we can start to build up programs that reuse *data*.

我们还不能复用代码，但是我们可以构建能够复用数据的程序。

8.4 Assignment

8.4 赋值

> It's possible to create a language that has variables but does not let you reassign—or **mutate**—them. Haskell is one example. SML supports only mutable references and arrays—variables cannot be reassigned. Rust steers you away from mutation by requiring a `mut` modifier to enable assignment.

你可以创建一种语言，其中有变量，但是不支持对该变量重新赋值（或更改）。Haskell就是一个例子。SML只支持可变引用和数组——变量不能被重新赋值。Rust则通过要求`mut`标识符开启赋值，从而引导用户远离可更改变量。

> Mutating a variable is a side effect and, as the name suggests, some language folks think side effects are dirty or inelegant. Code should be pure math that produces values—crystalline, unchanging ones—like an act of divine creation. Not some grubby automaton that beats blobs of data into shape, one imperative grunt at a time.

更改变量是一种副作用，顾名思义，一些语言专家认为副作用是肮脏或不优雅的。代码应该是纯粹的数学，它会产生值——纯净的、不变的值——就像上帝造物一样。而不是一些肮脏的自动机器，将数据块转换成各种形式，一次执行一条命令。

Lox is not so austere. Lox is an imperative language, and mutation comes with the territory. Adding support for assignment doesn't require much work. Global variables already support redefinition, so most of the machinery is there now. Mainly, we're missing an explicit assignment notation.

Lox没有这么严苛。Lox是一个命令式语言，可变性是与生俱来的，添加对赋值操作的支持并不需要太多工作。全局变量已经支持了重定义，所以该机制的大部分功能已经存在。主要的是，我们缺少显式的赋值符号。

> 8.4.1 Assignment syntax

8.4.1 赋值语法

> That little = syntax is more complex than it might seem. Like most C-derived languages, assignment is an expression and not a statement. As in C, it is the lowest precedence expression form. That means the rule slots between `expression` and `equality` (the next lowest precedence expression).

这个小小的=语法比看起来要更复杂。像大多数C派生语言一样，赋值是一个表达式，而不是一个语句。和C语言中一样，它是优先级最低的表达式形式。这意味着该规则在语法中处于 `expression` 和`equality`（下一个优先级的表达式）之间。

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment
               | equality ;
```

> This says an `assignment` is either an identifier followed by an = and an expression for the value, or an `equality` (and thus any other) expression. Later, `assignment` will get more complex when we add property setters on objects, like:

这就是说，一个`assignment`（赋值式）要么是一个标识符，后跟一个=和一个对应值的表达式；要么是一个等式（也就是任何其它）表达式。稍后，当我们在对象中添加属性设置式时，赋值将会变得更加复杂，比如：

```
instance.field = "value";
```

> The easy part is adding the new syntax tree node.

最简单的部分就是添加新的语法树节点。

tool/GenerateAst.java，在 main()方法中添加：

```
    defineAst(outputDir, "Expr", Arrays.asList(
      // 新增部分开始
      "Assign   : Token name, Expr value",
      // 新增部分结束
      "Binary   : Expr left, Token operator, Expr right",
```

> It has a token for the variable being assigned to, and an expression for the new value. After you run the AstGenerator to get the new Expr.Assign class, swap out the body of the parser's existing `expression()` method to match the updated rule.

其中包含被赋值变量的标记，一个计算新值的表达式。运行AstGenerator得到新的`Expr.Assign`类之后，替换掉解析器中现有的`expression()`方法的方法体，以匹配最新的规则。

lox/Parser.java，在 expression()方法中替换一行：

```
    private Expr expression() {
      // 替换部分开始
      return assignment();
      // 替换部分结束
    }
```

> Here is where it gets tricky. A single token lookahead recursive descent parser can't see far enough to tell that it's parsing an assignment until *after* it has gone through the left-hand side and stumbled onto the =. You might wonder why it even needs to. After all, we don't know we're parsing a + expression until after we've finished parsing the left operand.

这里开始变得棘手。单个标记前瞻递归下降解析器直到解析完左侧标记并且遇到=标记*之后*，才能判断出来正在解析的是赋值语句。你可能会想，为什么需要这样做？毕竟，我们也是完成左操作数的解析之后才知道正在解析的是+表达式。

> The difference is that the left-hand side of an assignment isn't an expression that evaluates to a value. It's a sort of pseudo-expression that evaluates to a "thing" you can assign to. Consider:

区别在于，赋值表达式的左侧不是可以求值的表达式，而是一种伪表达式，计算出的是一个你可以赋值的"东西"。考虑以下代码：

```
var a = "before";
a = "value";
```

> On the second line, we don't *evaluate* a (which would return the string "before"). We figure out what variable a refers to so we know where to store the right-hand side expression's value. The classic terms for these two constructs are **l-value** and **r-value**. All of the expressions that we've seen so far that produce values are r-values. An l-value "evaluates" to a storage location that you can assign into.

在第二行中，我们不会对a进行求值（如果求值会返回"before"）。我们要弄清楚a指向的是什么变量，这样我们就知道该在哪里保存右侧表达式的值。这两个概念的经典术语是**左值**和**右值**。到目前为止，我们看到的所有产生值的表达式都是右值。左值"计算"会得到一个存储位置，你可以向其赋值。

> We want the syntax tree to reflect that an l-value isn't evaluated like a normal expression. That's why the Expr.Assign node has a *Token* for the left-hand side, not an Expr. The problem is that the parser doesn't know it's parsing an l-value until it hits the =. In a complex l-value, that may occur many tokens later.

我们希望语法树能够反映出左值不会像常规表达式那样计算。这也是为什么Expr.Assign节点的左侧是一个Token，而不是Expr。问题在于，解析器直到遇到=才知道正在解析一个左值。在一个复杂的左值中，可能在出现很多标记之后才能识别到。

```
makeList().head.next = node;
```

> We have only a single token of lookahead, so what do we do? We use a little trick, and it looks like this:

我们只会前瞻一个标记，那我们该怎么办呢？我们使用一个小技巧，看起来像下面这样^10：

lox/Parser.java，在 expressionStatement()方法后添加：

```
  private Expr assignment() {
    Expr expr = equality();

    if (match(EQUAL)) {
      Token equals = previous();
      Expr value = assignment();

      if (expr instanceof Expr.Variable) {
        Token name = ((Expr.Variable)expr).name;
        return new Expr.Assign(name, value);
      }

      error(equals, "Invalid assignment target.");
    }

    return expr;
  }
```

> Most of the code for parsing an assignment expression looks similar to that of the other binary operators like `+`. We parse the left-hand side, which can be any expression of higher precedence. If we find an `=`, we parse the right-hand side and then wrap it all up in an assignment expression tree node.

解析赋值表达式的大部分代码看起来与解析其它二元运算符（如`+`）的代码类似。我们解析左边的内容，它可以是任何优先级更高的表达式。如果我们发现一个`=`，就解析右侧内容，并把它们封装到一个复杂表达式树节点中。

> One slight difference from binary operators is that we don't loop to build up a sequence of the same operator. Since assignment is right-associative, we instead recursively call `assignment()` to parse the right-hand side.

与二元运算符的一个细微差别在于，我们不会循环构建相同操作符的序列。因为赋值操作是右关联的，所以我们递归调用 `assignment()`来解析右侧的值。

> The trick is that right before we create the assignment expression node, we look at the left-hand side expression and figure out what kind of assignment target it is. We convert the r-value expression node into an l-value representation.

诀窍在于，在创建赋值表达式节点之前，我们先查看左边的表达式，弄清楚它是什么类型的赋值目标。然后我们将右值表达式节点转换为左值的表示形式。

> This conversion works because it turns out that every valid assignment target happens to also be valid syntax as a normal expression. Consider a complex field assignment like:

这种转换是有效的，因为事实证明，每个有效的赋值目标正好也是符合普通表达式的有效语法^11。考虑一个复杂的属性赋值操作，如下：

```
newPoint(x + 2, 0).y = 3;
```

> The left-hand side of that assignment could also work as a valid expression.

该赋值表达式的左侧也是一个有效的表达式。

```
newPoint(x + 2, 0).y;
```

> The first example sets the field, the second gets it.

第一个例子设置该字段，第二个例子获取该字段。

> This means we can parse the left-hand side *as if it were* an expression and then after the fact produce a syntax tree that turns it into an assignment target. If the left-hand side expression isn't a valid assignment target, we fail with a syntax error. That ensures we report an error on code like this:

这意味着，我们可以像解析表达式一样解析左侧内容，然后生成一个语法树，将其转换为赋值目标。如果左边的表达式不是一个有效的赋值目标，就会出现一个语法错误^12。这样可以确保在遇到类似下面的代码时会报告错误：

```
a + b = c;
```

> Right now, the only valid target is a simple variable expression, but we'll add fields later. The end result of this trick is an assignment expression tree node that knows what it is assigning to and has an expression subtree for the value being assigned. All with only a single token of lookahead and no backtracking.

现在，唯一有效的赋值目标就是一个简单的变量表达式，但是我们后面会添加属性字段。这个技巧的最终结果是一个赋值表达式树节点，该节点知道要向什么赋值，并且有一个表达式子树用于计算要使用的值。所有这些都只用了一个前瞻标记，并且没有回溯。

8.4.2 Assignment semantics

> We have a new syntax tree node, so our interpreter gets a new visit method.

我们有了一个新的语法树节点，所以我们的解释器也需要一个新的访问方法。

lox/Interpreter.java，在 visitVarStmt()方法后添加：

```
  @Override
  public Object visitAssignExpr(Expr.Assign expr) {
    Object value = evaluate(expr.value);
    environment.assign(expr.name, value);
    return value;
  }
```

> For obvious reasons, it's similar to variable declaration. It evaluates the right-hand side to get the value, then stores it in the named variable. Instead of using `define()` on Environment, it calls this new method:

很明显，这与变量声明很类似。首先，对右侧表达式运算以获取值，然后将其保存到命名变量中。这里不使用Environment中的 `define()`，而是调用下面的新方法：

lox/Environment.java，在 get()方法后添加：

```
void assign(Token name, Object value) {
  if (values.containsKey(name.lexeme)) {
    values.put(name.lexeme, value);
    return;
  }

  throw new RuntimeError(name,
      "Undefined variable '" + name.lexeme + "'.");
}
```

> The key difference between assignment and definition is that assignment is not allowed to create a *new* variable. In terms of our implementation, that means it's a runtime error if the key doesn't already exist in the environment's variable map.

赋值与定义的主要区别在于，赋值操作不允许创建新变量。就我们的实现而言，这意味着如果环境的变量映射中不存在变量的键，那就是一个运行时错误^13。

> The last thing the `visit()` method does is return the assigned value. That's because assignment is an expression that can be nested inside other expressions, like so:

`visit()`方法做的最后一件事就是返回要赋给变量的值。这是因为赋值是一个表达式，可以嵌套在其他表达式里面，就像这样:

```
var a = 1;
print a = 2; // "2".
```

> Our interpreter can now create, read, and modify variables. It's about as sophisticated as early BASICs. Global variables are simple, but writing a large program when any two chunks of code can accidentally step on each other's state is no fun. We want *local* variables, which means it's time for *scope*.

我们的解释器现在可以创建、读取和修改变量。这和早期的BASICs一样复杂。全局变量很简单，但是在编写一个大型程序时，任何两块代码都可能不小心修改对方的状态，这就不好玩了。我们需要*局部*变量，这意味着是时候讨论*作用域了*。

8.5 Scope

8.5 作用域

> A **scope** defines a region where a name maps to a certain entity. Multiple scopes enable the same name to refer to different things in different contexts. In my house, “Bob” usually refers to me. But maybe in your town you know a different Bob. Same name, but different dudes based on where you say it.

作用域定义了名称映射到特定实体的一个区域。多个作用域允许同一个名称在不同的上下文中指向不同的内容。在我家，“Bob”通常指的是我自己，但是在你的身边，你可能认识另外一个Bob。同一个名字，基于你的所知所见指向了不同的人。

> **Lexical scope** (or the less commonly heard **static scope**) is a specific style of scoping where the text of the program itself shows where a scope begins and ends. In Lox, as in most modern languages, variables are lexically scoped. When you see an expression that uses some variable, you can figure out which variable declaration it refers to just by statically reading the code.

词法作用域（或者比较少见的**静态作用域**）是一种特殊的作用域定义方式，程序本身的文本显示了作用域的开始和结束位置^14。Lox，和大多数现代语言一样，变量在词法作用域内有效。当你看到使用了某些变量的表达式时，你通过静态地阅读代码就可以确定其指向的变量声明。

> For example:

举例来说：

```
{
  var a = "first";
  print a; // "first".
}

{
  var a = "second";
  print a; // "second".
}
```

> Here, we have two blocks with a variable `a` declared in each of them. You and I can tell just from looking at the code that the use of `a` in the first `print` statement refers to the first `a`, and the second one refers to the second.

这里，我们在两个块中都定义了一个变量`a`。我们可以从代码中看出，在第一个`print`语句中使用的`a`指的是第一个`a`，第二个语句指向的是第二个变量。

FIRST BLOCK
a
"first"
SECOND BLOCK
a
"second"

> This is in contrast to **dynamic scope** where you don’t know what a name refers to until you execute the code. Lox doesn’t have dynamically scoped *variables*, but methods and fields on objects are dynamically scoped.

这与**动态作用域**形成了对比，在动态作用域中，直到执行代码时才知道名称指向的是什么。Lox没有动态作用域*变量*，但是对象上的方法和字段是动态作用域的。

```
class Saxophone {
  play() {
    print "Careless Whisper";
  }
}

class GolfClub {
  play() {
    print "Fore!";
  }
}

fun playIt(thing) {
  thing.play();
}
```

When `playIt()` calls `thing.play()`, we don't know if we're about to hear "Careless Whisper" or "Fore!" It depends on whether you pass a Saxophone or a GolfClub to the function, and we don't know that until runtime.

当`playIt()`调用`thing.play()`时，我们不知道我们将要听到的是 "Careless Whisper "还是 "Fore!" 。这取决于你向函数传递的是Saxophone还是GolfClub，而我们在运行时才知道这一点。

> Scope and environments are close cousins. The former is the theoretical concept, and the latter is the machinery that implements it. As our interpreter works its way through code, syntax tree nodes that affect scope will change the environment. In a C-ish syntax like Lox's, scope is controlled by curly-braced blocks. (That's why we call it **block scope**.)

作用域和环境是近亲，前者是理论概念，而后者是实现它的机制。当我们的解释器处理代码时，影响作用域的语法树节点会改变环境上下文。在像Lox这样的类C语言语法中，作用域是由花括号的块控制的。（这就是为什么我们称它为**块范围**）。

```
{
  var a = "in block";
}
print a; // Error! No more "a".
```

> The beginning of a block introduces a new local scope, and that scope ends when execution passes the closing `}`. Any variables declared inside the block disappear.

块的开始引入了一个新的局部作用域，当执行通过结束的`}`时，这个作用域就结束了。块内声明的任何变量都会消失。

8.5.1 Nesting and shadowing

8.5.1 嵌套和遮蔽

> A first cut at implementing block scope might work like this:

实现块作用域的第一步可能是这样的：

1. > As we visit each statement inside the block, keep track of any variables declared.

   当访问块内的每个语句时，跟踪所有声明的变量。

2. > After the last statement is executed, tell the environment to delete all of those variables.

   执行完最后一条语句后，告诉环境将这些变量全部删除。

> That would work for the previous example. But remember, one motivation for local scope is encapsulation—a block of code in one corner of the program shouldn't interfere with some other block. Check this out:

这对前面的例子是可行的。但是请记住，局部作用域的一个目的是封装——程序中一个块内的代码，不应该干扰其他代码块。看看下面的例子：

```
// How loud?
var volume = 11;

// Silence.
volume = 0;

// Calculate size of 3x4x5 cuboid.
{
  var volume = 3 * 4 * 5;
  print volume;
}
```

> Look at the block where we calculate the volume of the cuboid using a local declaration of `volume`. After the block exits, the interpreter will delete the *global* `volume` variable. That ain't right. When we exit the block, we should remove any variables declared inside the block, but if there is a variable with the same name declared outside of the block, *that's a different variable*. It shouldn't get touched.

请看这个代码块，在这里我们声明了一个局部变量`volume`来计算长方体的体积。该代码块退出后，解释器将删除*全局*`volume`变量。这是不对的。当我们退出代码块时，我们应该删除在块内声明的所有变量，但是如果在代码块外声明了相同名称的变量，那就是一个不同的变量。它不应该被删除。

> When a local variable has the same name as a variable in an enclosing scope, it **shadows** the outer one. Code inside the block can't see it any more—it is hidden in the "shadow" cast by the inner one—but it's still there.

当局部变量与外围作用域中的变量具有相同的名称时，内部变量会遮蔽外部变量。代码块内部不能再看到外部变量——它被遮蔽在内部变量的阴影中——但它仍然是存在的。

> When we enter a new block scope, we need to preserve variables defined in outer scopes so they are still around when we exit the inner block. We do that by defining a fresh environment for each block containing only the variables defined in that scope. When we exit the block, we discard its environment and restore the previous one.

当进入一个新的块作用域时，我们需要保留在外部作用域中定义的变量，这样当我们退出内部代码块时这些外部变量仍然存在。为此，我们为每个代码块定义一个新的环境，该环境只包含该作用域中定义的变量。当我们退出代码块时，我们将丢弃其环境并恢复前一个环境。

> We also need to handle enclosing variables that are *not* shadowed.

我们还需要处理没有被遮蔽的外围变量。

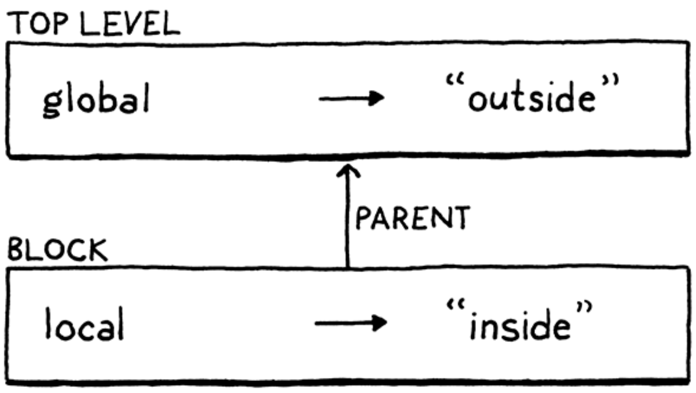
```
var global = "outside";
{
  var local = "inside";
  print global + local;
}
```

> Here, `global` lives in the outer global environment and `local` is defined inside the block's environment. In that `print` statement, both of those variables are in scope. In order to find them, the interpreter must search not only the current innermost environment, but also any enclosing ones.

这段代码中，`global`在外部全局环境中，`local`则在块环境中定义。在执行print`语句时，这两个变量都在作用域内。为了找到它们，解释器不仅要搜索当前最内层的环境，还必须搜索所有外围的环境。

> We implement this by chaining the environments together. Each environment has a reference to the environment of the immediately enclosing scope. When we look up a variable, we walk that chain from innermost out until we find the variable. Starting at the inner scope is how we make local variables shadow outer ones.

我们通过将环境链接在一起来实现这一点。每个环境都有一个对直接外围作用域的环境的引用。当我们查找一个变量时，我们从最内层开始遍历环境链直到找到该变量。从内部作用域开始，就是我们使局部变量遮蔽外部变量的方式。



> Before we add block syntax to the grammar, we'll beef up our Environment class with support for this nesting. First, we give each environment a reference to its enclosing one.

在我们添加块语法之前，我们要强化Environment类对这种嵌套的支持。首先，我们在每个环境中添加一个对其外围环境的引用。

lox/Environment.java，在 Environment类中添加：

```
class Environment {
  // 新增部分开始
  final Environment enclosing;
  // 新增部分结束
  private final Map<String, Object> values = new HashMap<>();
```

> This field needs to be initialized, so we add a couple of constructors.

这个字段需要初始化，所以我们添加两个构造函数。

lox/Environment.java，在 Environment类中添加：

```
  Environment() {
    enclosing = null;
  }

  Environment(Environment enclosing) {
    this.enclosing = enclosing;
  }
```

> The no-argument constructor is for the global scope’s environment, which ends the chain. The other constructor creates a new local scope nested inside the given outer one.

无参构造函数用于全局作用域环境，它是环境链的结束点。另一个构造函数用来创建一个嵌套在给定外部作用域内的新的局部作用域。

> We don’t have to touch the `define()` method—a new variable is always declared in the current innermost scope. But variable lookup and assignment work with existing variables and they need to walk the chain to find them. First, lookup:

我们不必修改`define()`方法——因为新变量总是在当前最内层的作用域中声明。但是变量的查找和赋值是结合已有的变量一起处理的，需要遍历环境链以找到它们。首先是查找操作：

lox/Environment.java，在 get()方法中添加：

```
      return values.get(name.lexeme);
    }
    // 新增部分开始
    if (enclosing != null) return enclosing.get(name);
    // 新增部分结束
    throw new RuntimeError(name,
        "Undefined variable '" + name.lexeme + "'.");
```

> If the variable isn't found in this environment, we simply try the enclosing one. That in turn does the same thing recursively, so this will ultimately walk the entire chain. If we reach an environment with no enclosing one and still don't find the variable, then we give up and report an error as before.

如果当前环境中没有找到变量，就在外围环境中尝试。然后递归地重复该操作，最终会遍历完整个链路。如果我们到达了一个没有外围环境的环境，并且仍然没有找到这个变量，那我们就放弃，并且像之前一样报告一个错误。

> Assignment works the same way.

赋值也是如此。

lox/Environment.java，在 assign()方法中添加：

```
      values.put(name.lexeme, value);
      return;
    }
    // 新增部分开始
    if (enclosing != null) {
      enclosing.assign(name, value);
      return;
    }
    // 新增部分结束
    throw new RuntimeError(name,
```

> Again, if the variable isn't in this environment, it checks the outer one, recursively.

同样，如果变量不在此环境中，它会递归地检查外围环境。

> ## 8.5.2 Block syntax and semantics

8.5.2 块语法和语义

> Now that Environments nest, we're ready to add blocks to the language. Behold the grammar:

现在环境已经嵌套了，我们就准备向语言中添加块了。请看以下语法：

```
statement      → exprStmt
               | printStmt
               | block ;

block          → "{" declaration* "}" ;
```

> A block is a (possibly empty) series of statements or declarations surrounded by curly braces. A block is itself a statement and can appear anywhere a statement is allowed. The syntax tree node looks like this:

块是由花括号包围的一系列语句或声明(可能是空的)。块本身就是一条语句，可以出现在任何允许语句的地方。语法树节点如下所示。

tool/GenerateAst.java，在 main()方法中添加：

```
    defineAst(outputDir, "Stmt", Arrays.asList(
      // 新增部分开始
      "Block      : List<Stmt> statements",
      // 新增部分结束
      "Expression : Expr expression",
```

> It contains the list of statements that are inside the block. Parsing is straightforward. Like other statements, we detect the beginning of a block by its leading token—in this case the `{`. In the `statement()` method, we add:

它包含块中语句的列表。解析很简单。与其他语句一样，我们通过块的前缀标记(在本例中是`{`)来检测块的开始。在`statement()`方法中，我们添加代码：

lox/Parser.java，在 statement()方法中添加：

```
    if (match(PRINT)) return printStatement();
    // 新增部分开始
    if (match(LEFT_BRACE)) return new Stmt.Block(block());
    // 新增部分结束
    return expressionStatement();
```

> All the real work happens here:

真正的工作都在这里进行：

lox/Parser.java，在 expressionStatement()方法后添加：

```
  private List<Stmt> block() {
    List<Stmt> statements = new ArrayList<>();

    while (!check(RIGHT_BRACE) && !isAtEnd()) {
      statements.add(declaration());
    }

    consume(RIGHT_BRACE, "Expect '}' after block.");
    return statements;
  }
```

> We create an empty list and then parse statements and add them to the list until we reach the end of the block, marked by the closing `}`. Note that the loop also has an explicit check for `isAtEnd()`. We have to be careful to avoid infinite loops, even when parsing invalid code. If the user forgets a closing `}`, the parser needs to not get stuck.

我们先创建一个空列表，然后解析语句并将其放入列表中，直至遇到块的结尾（由`}`符号标识）^15。注意，该循环还有一个明确的`isAtEnd()`检查。我们必须小心避免无限循环，即使在解析无效代码时也是如此。如果用户忘记了结尾的`}`，解析器需要保证不能被阻塞。

> That's it for syntax. For semantics, we add another visit method to Interpreter.

语法到此为止。对于语义，我们要在Interpreter中添加另一个访问方法。

lox/Interpreter.java，在 execute()方法后添加：

```
  @Override
  public Void visitBlockStmt(Stmt.Block stmt) {
    executeBlock(stmt.statements, new Environment(environment));
    return null;
  }
```

> To execute a block, we create a new environment for the block's scope and pass it off to this other method:

要执行一个块，我们先为该块作用域创建一个新的环境，然后将其传入下面这个方法：

lox/Interpreter.java，在execute()方法后添加：

```
  void executeBlock(List<Stmt> statements,
                    Environment environment) {
    Environment previous = this.environment;
    try {
      this.environment = environment;

      for (Stmt statement : statements) {
        execute(statement);
      }
    } finally {
      this.environment = previous;
    }
  }
```

> This new method executes a list of statements in the context of a given environment. Up until now, the `environment` field in Interpreter always pointed to the same environment—the global one. Now, that field represents the *current* environment. That's the environment that corresponds to the innermost scope containing the code to be executed.

这个新方法会在给定的环境上下文中执行一系列语句。在此之前，解释器中的 `environment` 字段总是指向相同的环境——全局环境。现在，这个字段会指向*当前*环境，也就是与要执行的代码的最内层作用域相对应的环境^16。

> To execute code within a given scope, this method updates the interpreter's `environment` field, visits all of the statements, and then restores the previous value. As is always good practice in Java, it

> restores the previous environment using a finally clause. That way it gets restored even if an exception is thrown.

为了在给定作用域内执行代码，该方法会先更新解释器的 `environment` 字段，执行所有的语句，然后恢复之前的环境。基于Java中一贯的优良传统，它使用`finally`子句来恢复先前的环境。这样一来，即使抛出了异常，环境也会被恢复。

> Surprisingly, that's all we need to do in order to fully support local variables, nesting, and shadowing. Go ahead and try this out:

出乎意料的是，这就是我们为了完全支持局部变量、嵌套和遮蔽所需要做的全部事情。试运行下面的代码：

```
var a = "global a";
var b = "global b";
var c = "global c";
{
  var a = "outer a";
  var b = "outer b";
  {
    var a = "inner a";
    print a;
    print b;
    print c;
  }
  print a;
  print b;
  print c;
}
print a;
print b;
print c;
```

> Our little interpreter can remember things now. We are inching closer to something resembling a full-featured programming language.

我们的小解释器现在可以记住东西了，我们距离全功能编程语言又近了一步。

```
a = 3;   // OK.
(a) = 3; // Error.
```

^14: "词法"来自希腊语" lexikos"，意思是"与单词有关"。当我们在编程语言中使用它时，通常意味着您无需执行任何操作即可从源代码本身中获取到一些东西。词法作用域是随着ALGOL出现的。早期的语言通常是动态作用域的。当时的计算机科学家认为，动态作用域的执行速度更快。今天，多亏了早期的Scheme研究者，我们知道这不是真的。甚至可以说，情况恰恰相反。变量的动态作用域仍然存在于某些角落。Emacs Lisp默认为变量的动态作用域。Clojure中的`binding`宏也提供了。JavaScript中普遍不被喜欢的`with`语句将对象上的属性转换为动态作用域变量。 ^15: 让`block()`返回原始的语句列表，并在`statement()`方法中将该列表封装在Stmt.Block中，这看起来有点奇怪。我这样做是因为稍后我们会重用`block()`来解析函数体，我们当然不希望函数体被封装在Stmt.Block中。 ^16: 手动修改和恢复一个可变的`environment`字段感觉很不优雅。另一种经典方法是显式

地将环境作为参数传递给每个访问方法。如果要"改变"环境，就在沿树向下递归时传入一个不同的环境。你不必恢复旧的环境，因为新的环境存在于 Java 堆栈中，当解释器从块的访问方法返回时，该环境会被隐式丢弃。我曾考虑过在jlox中这样做，但在每一个访问方法中加入一个环境参数，这有点繁琐和冗长。为了让这本书更简单，我选择了可变字段。

CHALLENGES

习题

> 1、The REPL no longer supports entering a single expression and automatically printing its result value. That's a drag. Add support to the REPL to let users type in both statements and expressions. If they enter a statement, execute it. If they enter an expression, evaluate it and display the result value.

1、REPL不再支持输入一个表达式并自动打印其结果值。这是个累赘。在 REPL 中增加支持，让用户既可以输入语句又可以输入表达式。如果他们输入一个语句，就执行它。如果他们输入一个表达式，则对表达式求值并显示结果值。

> 2、Maybe you want Lox to be a little more explicit about variable initialization. Instead of implicitly initializing variables to `nil`, make it a runtime error to access a variable that has not been initialized or assigned to, as in:

2、也许你希望Lox对变量的初始化更明确一些。与其隐式地将变量初始化为nil，不如将访问一个未被初始化或赋值的变量作为一个运行时错误，如：

```
// No initializers.
var a;
var b;

a = "assigned";
print a; // OK, was assigned first.

print b; // Error!
```

> 3、What does the following program do?

3、下面的代码会怎么执行？

```
var a = 1;
{
  var a = a + 2;
  print a;
}
```

> What did you *expect* it to do? Is it what you think it should do? What does analogous code in other languages you are familiar with do? What do you think users will expect this to do?

你期望它怎么执行？它是按照你的想法执行的吗？你所熟悉的其他语言中的类似代码怎么执行？你认为用户会期望它怎么执行？

DESIGN NOTE: IMPLICIT VARIABLE DECLARATION

设计笔记：隐式变量声明

> Lox has distinct syntax for declaring a new variable and assigning to an existing one. Some languages collapse those to only assignment syntax. Assigning to a non-existent variable automatically brings it into being. This is called **implicit variable declaration** and exists in Python, Ruby, and CoffeeScript, among others. JavaScript has an explicit syntax to declare variables, but can also create new variables on assignment. Visual Basic has an option to enable or disable implicit variables.
>
> When the same syntax can assign or create a variable, each language must decide what happens when it isn't clear about which behavior the user intends. In particular, each language must choose how implicit declaration interacts with shadowing, and which scope an implicitly declared variable goes into.
>
> - In Python, assignment always creates a variable in the current function's scope, even if there is a variable with the same name declared outside of the function.
> - Ruby avoids some ambiguity by having different naming rules for local and global variables. However, blocks in Ruby (which are more like closures than like "blocks" in C) have their own scope, so it still has the problem. Assignment in Ruby assigns to an existing variable outside of the current block if there is one with the same name. Otherwise, it creates a new variable in the current block's scope.
> - CoffeeScript, which takes after Ruby in many ways, is similar. It explicitly disallows shadowing by saying that assignment always assigns to a variable in an outer scope if there is one, all the way up to the outermost global scope. Otherwise, it creates the variable in the current function scope.
> - In JavaScript, assignment modifies an existing variable in any enclosing scope, if found. If not, it implicitly creates a new variable in the *global* scope.
>
> The main advantage to implicit declaration is simplicity. There's less syntax and no "declaration" concept to learn. Users can just start assigning stuff and the language figures it out.
>
> Older, statically typed languages like C benefit from explicit declaration because they give the user a place to tell the compiler what type each variable has and how much storage to allocate for it. In a dynamically typed, garbage-collected language, that isn't really necessary, so you can get away with making declarations implicit. It feels a little more "scripty", more "you know what I mean".
>
> But is that a good idea? Implicit declaration has some problems.
>
> - A user may intend to assign to an existing variable, but may have misspelled it. The interpreter doesn't know that, so it goes ahead and silently creates some new variable and the variable the user wanted to assign to still has its old value. This is particularly heinous in JavaScript where a typo will create a *global* variable, which may in turn interfere with other code.
> - JS, Ruby, and CoffeeScript use the presence of an existing variable with the same name—even in an outer scope—to determine whether or not an assignment creates a new variable or assigns to an existing one. That means adding a new variable in a surrounding scope can change the

> meaning of existing code. What was once a local variable may silently turn into an assignment to that new outer variable.
> - In Python, you may *want* to assign to some variable outside of the current function instead of creating a new variable in the current one, but you can't.
>
> Over time, the languages I know with implicit variable declaration ended up adding more features and complexity to deal with these problems.
>
> - Implicit declaration of global variables in JavaScript is universally considered a mistake today. "Strict mode" disables it and makes it a compile error.
> - Python added a `global` statement to let you explicitly assign to a global variable from within a function. Later, as functional programming and nested functions became more popular, they added a similar `nonlocal` statement to assign to variables in enclosing functions.
> - Ruby extended its block syntax to allow declaring certain variables to be explicitly local to the block even if the same name exists in an outer scope.
>
> Given those, I think the simplicity argument is mostly lost. There is an argument that implicit declaration is the right *default* but I personally find that less compelling.
>
> My opinion is that implicit declaration made sense in years past when most scripting languages were heavily imperative and code was pretty flat. As programmers have gotten more comfortable with deep nesting, functional programming, and closures, it's become much more common to want access to variables in outer scopes. That makes it more likely that users will run into the tricky cases where it's not clear whether they intend their assignment to create a new variable or reuse a surrounding one.
>
> So I prefer explicitly declaring variables, which is why Lox requires it.

Lox使用不同的语法来声明新变量和为已有变量赋值。有些语言将其简化为只有赋值语法。对一个不存在的变量进行赋值时会自动生成该变量。这被称为**隐式变量声明**，存在于Python、Ruby和CoffeeScript以及其他语言中。JavaScript有一个显式的语法来声明变量，但是也可以在赋值时创建新变量。Visual Basic有一个选项可以启用或禁用隐式变量。

当同样的语法既可以对变量赋值，也可以创建变量时，语言实现就必须决定在不清楚用户的预期行为时该怎么办。特别是，每种语言必须选择隐式变量声明与变量遮蔽的交互方式，以及隐式变量应该属于哪个作用域。

- 在Python中，赋值总是会在当前函数的作用域内创建一个变量，即使在函数外部声明了同名变量。
- Ruby通过对局部变量和全局变量使用不同的命名规则，避免了一些歧义。 但是，Ruby中的块（更像闭包，而不是C中的"块"）具有自己的作用域，因此仍然存在问题。在Ruby中，如果已经存在一个同名的变量，则赋值会赋给当前块之外的现有变量。否则，就会在当前块的作用域中创建一个新变量。
- CoffeeScript在许多方面都效仿Ruby，这一点也类似。它明确禁止变量遮蔽，要求赋值时总是优先赋给外部作用域中现有的变量（一直到最外层的全局作用域）。如果变量不存在的话，它会在当前函数作用域中创建新变量。
- 在JavaScript中，赋值会修改任意外部作用域中的一个现有变量（如果能找到该变量的话）。如果变量不存在，它就隐式地在全局作用域内创建一个新的变量。

隐式声明的主要优点是简单。语法较少，无需学习"声明"概念。用户可以直接开始赋值，然后语言就能解决其它问题。

像C这样较早的静态类型语言受益于显式声明，是因为它们给用户提供了一个地方，让他们告诉编译器每个变量的类型以及为它分配多少存储空间。在动态类型、垃圾收集的语言中，这其实是没有必要的，所以你可以通

过隐式声明来实现。这感觉更 "脚本化"，更像是 "你懂我的意思吧"。

但这是就个好主意吗？隐式声明还存在一些问题。

- 用户可能打算为现有变量赋值，但是出现拼写错误。解释器不知道这一点，所以它悄悄地创建了一些新变量，而用户想要赋值的变量仍然是原来的值。这在JavaScript中尤其令人讨厌，因为一个拼写错误会创建一个全局变量，这反过来又可能会干扰其它代码。
- JS、Ruby和CoffeeScript通过判断是否存在同名变量——包括外部作用域——来确定赋值是创建新变量还是赋值给现有变量。这意味着在外围作用域中添加一个新变量可能会改变现有代码的含义，原先的局部变量可能会默默地变成对新的外部变量的赋值。
- 在Python中，你可能想要赋值给当前函数之外的某个变量，而不是在当前函数中创建一个新变量，但是你做不到。

随着时间的推移，我所知道的具有隐式变量声明的语言最后都增加了更多的功能和复杂性来处理这些问题。

- 现在，普遍认为JavaScript中全局变量的隐式声明是一个错误。"Strict mode "禁用了它，并将其成为一个编译错误。
- Python添加了一个`global`语句，让用户可以在函数内部显式地赋值给一个全局变量。后来，随着函数式编程和嵌套函数越来越流行，他们添加了一个类似的`nonlocal`语句来赋值给外围函数中的变量。
- Ruby扩展了它的块语法，允许在块中显式地声明某些变量，即使外部作用域中存在同名的变量。

考虑到这些，我认为简单性的论点已经失去了意义。有一种观点认为隐式声明是正确的默认选项，但我个人认为这种说法不太有说服力。

我的观点是，隐式声明在过去的几年里是有意义的，当时大多数脚本语言都是非常命令式的，代码是相当简单直观的。随着程序员对深度嵌套、函数式编程和闭包越来越熟悉，访问外部作用域中的变量变得越来越普遍。这使得用户更有可能遇到棘手的情况，即不清楚他们的赋值是要创建一个新变量还是重用外围的已有变量。

所以我更喜欢显式声明变量，这就是Lox要这样做的原因。

9.控制流 Control Flow

> Logic, like whiskey, loses its beneficial effect when taken in too large quantities.
>
> —— Edward John Moreton Drax Plunkett, Lord Dunsany

逻辑和威士忌一样，如果摄入太多，就会失去其有益的效果。

> Compared to last chapter's grueling marathon, today is a lighthearted frolic through a daisy meadow. But while the work is easy, the reward is surprisingly large.

与上一章艰苦的马拉松相比，这一章就是在雏菊草地上的轻松嬉戏。虽然工作很简单，但回报却惊人的大。

> Right now, our interpreter is little more than a calculator. A Lox program can only do a fixed amount of work before completing. To make it run twice as long you have to make the source code twice as lengthy. We're about to fix that. In this chapter, our interpreter takes a big step towards the programming language major leagues: *Turing-completeness*.

现在，我们的解释器只不过是一个计算器而已。一个Lox程序在结束之前只能做固定的工作量。要想让它的运行时间延长一倍，你就必须让源代码的长度增加一倍。我们即将解决这个问题。在本章中，我们的解释器向编程语言大联盟迈出了一大步：图灵完备性。

9.1Turing Machines (Briefly)

9.1 图灵机（简介）

> In the early part of last century, mathematicians stumbled into a series of confusing paradoxes that led them to doubt the stability of the foundation they had built their work upon. To address that crisis, they went back to square one. Starting from a handful of axioms, logic, and set theory, they hoped to rebuild mathematics on top of an impervious foundation.

在上世纪初，数学家们陷入了一系列令人困惑的悖论之中，导致他们对自己工作所依赖的基础的稳定性产生怀疑^1。为了解决这一危机，他们又回到了原点。他们希望从少量的公理、逻辑和集合理论开始，在一个不透水的地基上重建数学。

> They wanted to rigorously answer questions like, “Can all true statements be proven?”, “Can we compute all functions that we can define?”, or even the more general question, “What do we mean when we claim a function is ‘computable’?”

他们想要严格地回答这样的问题：“所有真实的陈述都可以被证明吗？”，“我们可以计算我们能定义的所有函数吗？”，甚至是更一般性的问题，“当我们声称一个函数是'可计算的'时，代表什么意思？”

> They presumed the answer to the first two questions would be “yes”. All that remained was to prove it. It turns out that the answer to both is “no”, and astonishingly, the two questions are deeply intertwined. This is a fascinating corner of mathematics that touches fundamental questions about what brains are able to do and how the universe works. I can’t do it justice here.

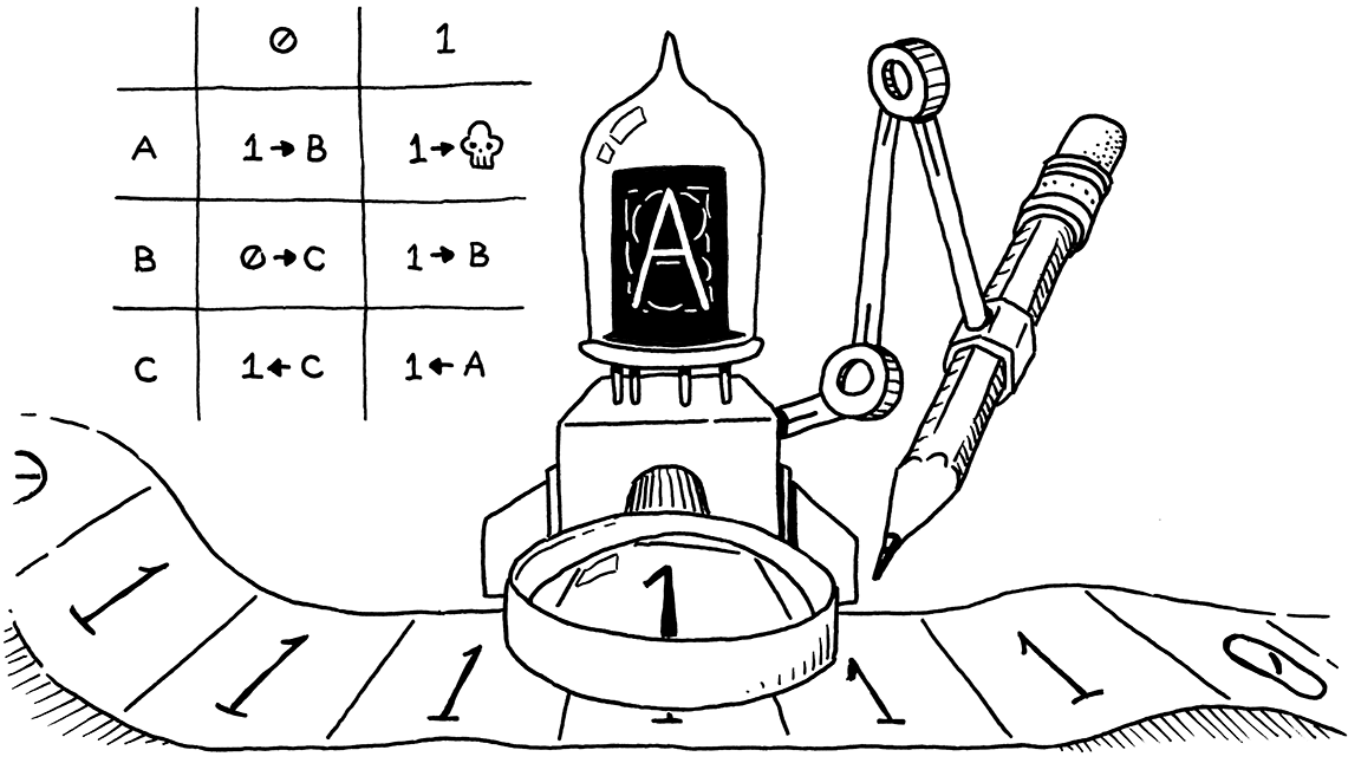
他们认为前两个问题的答案应该是“是”，剩下的就是去证明它。但事实证明这两个问题的答案都是“否”。而且令人惊讶的是，这两个问题是深深地交织在一起的。这是数学的一个迷人的角落，它触及了关于大脑能够做什么和宇宙如何运作的基本问题。我在这里说不清楚。

> What I do want to note is that in the process of proving that the answer to the first two questions is “no”, Alan Turing and Alonzo Church devised a precise answer to the last question—a definition of what kinds of functions are computable. They each crafted a tiny system with a minimum set of machinery that is still powerful enough to compute any of a (very) large class of functions.

我想指出的是，在证明前两个问题的答案是 "否 "的过程中，艾伦·图灵和阿隆佐·邱奇为最后一个问题设计了一个精确的答案，即定义了什么样的函数是可计算的。他们各自设计了一个具有最小机械集的微型系统，该系统仍然强大到足以计算一个超大类函数中的任何一个。

> These are now considered the “computable functions”. Turing’s system is called a **Turing machine**. Church’s is the **lambda calculus**. Both are still widely used as the basis for models of computation and, in fact, many modern functional programming languages use the lambda calculus at their core.

这些现在被认为是“可计算函数”。图灵的系统被称为**图灵机**^2，邱奇的系统是**lambda演算**。这两种方法仍然被广泛用作计算模型的基础，事实上，许多现代函数式编程语言的核心都是lambda演算。



> Turing machines have better name recognition—there's no Hollywood film about Alonzo Church yet—but the two formalisms are equivalent in power. In fact, any programming language with some minimal level of expressiveness is powerful enough to compute *any* computable function.

图灵机的知名度更高——目前还没有关于阿隆佐·邱奇的好莱坞电影，但这两种形式在能力上是等价的。事实上，任何具有最低表达能力的编程语言都足以计算任何可计算函数。

> You can prove that by writing a simulator for a Turing machine in your language. Since Turing proved his machine can compute any computable function, by extension, that means your language can too. All you need to do is translate the function into a Turing machine, and then run that on your simulator.

你可以用自己的语言为图灵机编写一个模拟器来证明这一点。由于图灵证明了他的机器可以计算任何可计算函数，推而广之，这意味着你的语言也可以。你所需要做的就是把函数翻译成图灵机，然后在你的模拟器上运行它。

> If your language is expressive enough to do that, it's considered **Turing-complete**. Turing machines are pretty dang simple, so it doesn't take much power to do this. You basically need arithmetic, a little control flow, and the ability to allocate and use (theoretically) arbitrary amounts of memory. We've got the first. By the end of this chapter, we'll have the second.

如果你的语言有足够的表达能力来做到这一点，它就被认为是**图灵完备**的。图灵机非常简单，所以它不需要太多的能力。您基本上只需要算术、一点控制流以及分配和使用(理论上)任意数量内存的能力。我们已经具备了第一个条件^3。在本章结束时，我们将具备第二个条件。

9.2Conditional Execution

9.2 条件执行

> Enough history, let's jazz up our language. We can divide control flow roughly into two kinds:

说完了历史，现在让我们把语言优化一下。我们大致可以把控制流分为两类：

- > **Conditional** or **branching control flow** is used to *not* execute some piece of code. Imperatively, you can think of it as jumping *ahead* over a region of code.

  条件或**分支控制流**是用来不执行某些代码的。意思是，你可以把它看作是跳过了代码的一个区域。

- > **Looping control flow** executes a chunk of code more than once. It jumps *back* so that you can do something again. Since you don't usually want *infinite* loops, it typically has some conditional logic to know when to stop looping as well.

  循环控制流是用于多次执行一块代码的。它会*向回*跳转，从而能再次执行某些代码。用户通常不需要无限循环，所以一般也会有一些条件逻辑用于判断何时停止循环。

> Branching is simpler, so we'll start there. C-derived languages have two main conditional execution features, the `if` statement and the perspicaciously named "conditional" operator (`?:`). An `if` statement lets you conditionally execute statements and the conditional operator lets you conditionally execute expressions.

分支更简单一些，所以我们先从分支开始实现。C衍生语言中包含两个主要的条件执行功能，即`if`语句和"条件"运算符（`?:`）^4。`if`语句使你可以按条件执行语句，而条件运算符使你可以按条件执行表达式。

> For simplicity's sake, Lox doesn't have a conditional operator, so let's get our `if` statement on. Our statement grammar gets a new production.

为了简单起见，Lox没有条件运算符，所以让我们直接开始`if`语句吧。我们的语句语法需要一个新的生成式。

```
statement      → exprStmt
               | ifStmt
               | printStmt
               | block ;

ifStmt         → "if" "(" expression ")" statement
               ( "else" statement )? ;
```

> An `if` statement has an expression for the condition, then a statement to execute if the condition is truthy. Optionally, it may also have an `else` keyword and a statement to execute if the condition is falsey. The syntax tree node has fields for each of those three pieces.

if语句有一个表达式作为条件，然后是一个在条件为真时要执行的语句。另外，它还可以有一个`else`关键字和条件为假时要执行的语句。语法树节点中对语法的这三部分都有对应的字段。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Expression : Expr expression",
      // 新增部分开始
      "If         : Expr condition, Stmt thenBranch," +
                  " Stmt elseBranch",
      // 新增部分结束
      "Print      : Expr expression",
```

> Like other statements, the parser recognizes an `if` statement by the leading `if` keyword.

与其它语句类似，解析器通过开头的`if`关键字来识别`if`语句。

lox/Parser.java，在 statement()方法中添加：

```
  private Stmt statement() {
    // 新增部分开始
    if (match(IF)) return ifStatement();
    // 新增部分结束
    if (match(PRINT)) return printStatement();
```

> When it finds one, it calls this new method to parse the rest:

如果发现了`if`关键字，就调用下面的新方法解析其余部分^5：

lox/Parser.java，在 statement()方法后添加：

```
  private Stmt ifStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'if'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after if condition.");

    Stmt thenBranch = statement();
    Stmt elseBranch = null;
    if (match(ELSE)) {
      elseBranch = statement();
    }

    return new Stmt.If(condition, thenBranch, elseBranch);
  }
```

> As usual, the parsing code hews closely to the grammar. It detects an else clause by looking for the preceding `else` keyword. If there isn't one, the `elseBranch` field in the syntax tree is `null`.

跟之前一样，解析代码严格遵循语法。它通过查找前面的`else`关键字来检测`else`子句。如果没有，语法树中的`elseBranch`字段为`null`。

> That seemingly innocuous optional else has, in fact, opened up an ambiguity in our grammar. Consider:

实际上，这个看似无伤大雅的可选项在我们的语法中造成了歧义。考虑以下代码：

```
if (first) if (second) whenTrue(); else whenFalse();
```

> Here's the riddle: Which `if` statement does that else clause belong to? This isn't just a theoretical question about how we notate our grammar. It actually affects how the code executes:

谜题是这样的:这里的`else`子句属于哪个`if`语句?这不仅仅是一个关于如何标注语法的理论问题。它实际上会影响代码的执行方式：

- > If we attach the else to the first `if` statement, then `whenFalse()` is called if `first` is falsey, regardless of what value `second` has.

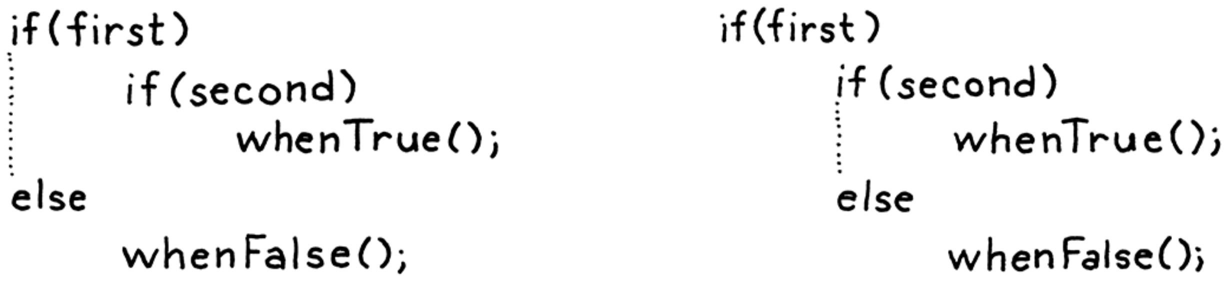
  如果我们将`else`语句关联到第一个`if`语句，那么当`first`为假时，无论`second`的值是多少，都将调用`whenFalse()`。

- > If we attach it to the second `if` statement, then `whenFalse()` is only called if `first` is truthy and `second` is falsey.

  如果我们将`else`语句关联到第二个`if`语句，那么只有当`first`为真并且`second`为假时，才会调用`whenFalse()`。

> Since else clauses are optional, and there is no explicit delimiter marking the end of the `if` statement, the grammar is ambiguous when you nest `if`s in this way. This classic pitfall of syntax is called the **dangling else** problem.

由于`else`子句是可选的，而且没有明确的分隔符来标记`if`语句的结尾，所以当你以这种方式嵌套`if`时，语法是不明确的。这种典型的语法陷阱被称为悬空的else问题。



> It *is* possible to define a context-free grammar that avoids the ambiguity directly, but it requires splitting most of the statement rules into pairs, one that allows an `if` with an `else` and one that doesn't. It's annoying.

也可以定义一个上下文无关的语法来直接避免歧义，但是需要将大部分语句规则拆分成对，一个是允许带有`else`的`if`语句，另一个不允许。这很烦人。

> Instead, most languages and parsers avoid the problem in an ad hoc way. No matter what hack they use to get themselves out of the trouble, they always choose the same interpretation—the `else` is bound to the nearest `if` that precedes it.

相反，大多数语言和解析器都以一种特殊的方式避免了这个问题。不管他们用什么方法来解决这个问题，他们总是选择同样的解释——`else`与前面最近的`if`绑定在一起。

> Our parser conveniently does that already. Since `ifStatement()` eagerly looks for an `else` before returning, the innermost call to a nested series will claim the else clause for itself before returning to the outer `if` statements.

我们的解析器已经很方便地做到了这一点。因为 `ifStatement()`在返回之前会继续寻找一个`else`子句，连续嵌套的最内层调用在返回外部的`if`语句之前，会先为自己声明`else`语句。

> Syntax in hand, we are ready to interpret.

语法就绪了，我们可以开始解释了。

lox/Interpreter.java，在 visitExpressionStmt()后添加：

```
  @Override
  public Void visitIfStmt(Stmt.If stmt) {
    if (isTruthy(evaluate(stmt.condition))) {
      execute(stmt.thenBranch);
    } else if (stmt.elseBranch != null) {
      execute(stmt.elseBranch);
    }
    return null;
  }
```

> The interpreter implementation is a thin wrapper around the self-same Java code. It evaluates the condition. If truthy, it executes the then branch. Otherwise, if there is an else branch, it executes that.

解释器实现就是对相同的Java代码的简单包装。它首先对条件表达式进行求值。如果为真，则执行`then`分支。否则，如果有存在`else`分支，就执行该分支。

> If you compare this code to how the interpreter handles other syntax we've implemented, the part that makes control flow special is that Java `if` statement. Most other syntax trees always evaluate their subtrees. Here, we may not evaluate the then or else statement. If either of those has a side effect, the choice not to evaluate it becomes user visible.

如果你把这段代码与解释器中我们已实现的处理其它语法的代码进行比较，会发现控制流中特殊的地方就在于Java的`if`语句。其它大多数语法树总是会对子树求值，但是这里，我们可能会不执行`then`语句或`else`语句。如果其中任何一个语句有副作用，那么选择不执行某条语句就是用户可见的。

9.3Logical Operators

9.3 逻辑操作符

> Since we don't have the conditional operator, you might think we're done with branching, but no. Even without the ternary operator, there are two other operators that are technically control flow constructs —the logical operators `and` and `or`.

由于我们没有条件运算符，你可能认为我们已经完成分支开发了，但其实还没有。虽然没有三元运算符，但是还有两个其它操作符在技术上是控制流结构——逻辑运算符`and`和`or`。

> These aren't like other binary operators because they **short-circuit**. If, after evaluating the left operand, we know what the result of the logical expression must be, we don't evaluate the right operand. For example:

它们与其它二进制运算符不同，是因为它们会短路。如果在计算左操作数之后，我们已经确切知道逻辑表达式的结果，那么就不再计算右操作数。例如：

```
false and sideEffect();
```

> For an and expression to evaluate to something truthy, both operands must be truthy. We can see as soon as we evaluate the left false operand that that isn't going to be the case, so there's no need to evaluate sideEffect() and it gets skipped.

对于一个and表达式来说，两个操作数都必须是真，才能得到结果为真。我们只要看到左侧的false操作数，就知道结果不会是真，也就不需要对sideEffect()求值，会直接跳过它。

> This is why we didn't implement the logical operators with the other binary operators. Now we're ready. The two new operators are low in the precedence table. Similar to || and && in C, they each have their own precedence with or lower than and. We slot them right between assignment and equality.

这就是为什么我们没有在实现其它二元运算符的时候一起实现逻辑运算符。现在我们已经准备好了。这两个新的运算符在优先级表中的位置很低，类似于C语言中的||和&&，它们都有各自的优先级，or低于and。我们把这两个运算符插入assignment 和 equality之间。

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment
               | logic_or ;
logic_or       → logic_and ( "or" logic_and )* ;
logic_and      → equality ( "and" equality )* ;
```

> Instead of falling back to equality, assignment now cascades to logic_or. The two new rules, logic_or and logic_and, are similar to other binary operators. Then logic_and calls out to equality for its operands, and we chain back to the rest of the expression rules.

assignment 现在不是落到 equality，而是继续进入logic_or。两个新规则，logic_or 和 logic_and，与其它二元运算符类似。然后logic_and会调用equality计算其操作数，然后我们就链入了表达式规则的其它部分。

> We could reuse the existing Expr.Binary class for these two new expressions since they have the same fields. But then visitBinaryExpr() would have to check to see if the operator is one of the logical operators and use a different code path to handle the short circuiting. I think it's cleaner to define a new class for these operators so that they get their own visit method.

对于这两个新表达式，我们可以重用Expr.Binary类，因为他们具有相同的字段。但是这样的话，visitBinaryExpr() 方法中必须检查运算符是否是逻辑运算符，并且要使用不同的代码处理短路逻辑。我认为更整洁的方法是为这些运算符定义一个新类，这样它们就有了自己的visit方法。

tool/GenerateAst.java，在main()方法中添加：

```
      "Literal  : Object value",
      // 新增部分开始
      "Logical  : Expr left, Token operator, Expr right",
      // 新增部分结束
      "Unary    : Token operator, Expr right",
```

> To weave the new expressions into the parser, we first change the parsing code for assignment to call `or()`.

为了将新的表达式加入到解析器中，我们首先将赋值操作的解析代码改为调用`or()`方法。

lox/Parser.java,在 assignment()方法中替换一行：

```
  private Expr assignment() {
    // 新增部分开始
    Expr expr = or();
    // 新增部分结束
    if (match(EQUAL)) {
```

> The code to parse a series of `or` expressions mirrors other binary operators.

解析一系列`or`语句的代码与其它二元运算符相似。

lox/Parser.java，在 assignment()方法后添加：

```
  private Expr or() {
    Expr expr = and();

    while (match(OR)) {
      Token operator = previous();
      Expr right = and();
      expr = new Expr.Logical(expr, operator, right);
    }

    return expr;
  }
```

> Its operands are the next higher level of precedence, the new `and` expression.

它的操作数是位于下一优先级的新的`and`表达式。

lox/Parser.java，在 or()方法后添加：

```
  private Expr and() {
    Expr expr = equality();

    while (match(AND)) {
      Token operator = previous();
      Expr right = equality();
      expr = new Expr.Logical(expr, operator, right);
    }

    return expr;
  }
```

> That calls `equality()` for its operands, and with that, the expression parser is all tied back together again. We're ready to interpret.

这里会调用 `equality()` 计算操作数，这样一来，表达式解析器又重新绑定到了一起。我们已经准备好进行解释了。

lox/Interpreter.java，在 visitLiteralExpr()方法后添加：

```
  @Override
  public Object visitLogicalExpr(Expr.Logical expr) {
    Object left = evaluate(expr.left);

    if (expr.operator.type == TokenType.OR) {
      if (isTruthy(left)) return left;
    } else {
      if (!isTruthy(left)) return left;
    }

    return evaluate(expr.right);
  }
```

> If you compare this to the earlier chapter's `visitBinaryExpr()` method, you can see the difference. Here, we evaluate the left operand first. We look at its value to see if we can short-circuit. If not, and only then, do we evaluate the right operand.

如果你把这个方法与前面章节的`visitBinaryExpr()`方法相比较，就可以看出其中的区别。这里，我们先计算左操作数。然后我们查看结果值，判断是否可以短路。当且仅当不能短路时，我们才计算右侧的操作数。

> The other interesting piece here is deciding what actual value to return. Since Lox is dynamically typed, we allow operands of any type and use truthiness to determine what each operand represents. We apply similar reasoning to the result. Instead of promising to literally return `true` or `false`, a logic operator merely guarantees it will return a value with appropriate truthiness.

另一个有趣的部分是决定返回什么实际值。由于Lox是动态类型的，我们允许任何类型的操作数，并使用真实性来确定每个操作数代表什么。我们对结果采用类似的推理。逻辑运算符并不承诺会真正返回`true`或`false`，而只是保证它将返回一个具有适当真实性的值。

> Fortunately, we have values with proper truthiness right at hand—the results of the operands themselves. So we use those. For example:

幸运的是，我们手边就有具有适当真实性的值——即操作数本身的结果，所以我们可以直接使用它们。如：

```
print "hi" or 2; // "hi".
print nil or "yes"; // "yes".
```

> On the first line, `"hi"` is truthy, so the `or` short-circuits and returns that. On the second line, `nil` is falsey, so it evaluates and returns the second operand, `"yes"`.

在第一行，`“hi”`是真的，所以`or`短路并返回它。在第二行，`nil`是假的，因此它计算并返回第二个操作数`“yes”`。

> That covers all of the branching primitives in Lox. We're ready to jump ahead to loops. You see what I did there? *Jump. Ahead.* Get it? See, it's like a reference to . . . oh, forget it.

这样就完成了Lox中的所有分支原语，我们准备实现循环吧。

9.4While Loops

9.4 While循环

> Lox features two looping control flow statements, `while` and `for`. The `while` loop is the simpler one, so we'll start there. Its grammar is the same as in C.

Lox有两种类型的循环控制流语句，分别是`while`和`for`。`while`循环更简单一点，我们先从它开始.

```
statement      → exprStmt
               | ifStmt
               | printStmt
               | whileStmt
               | block ;

whileStmt      → "while" "(" expression ")" statement ;
```

> We add another clause to the statement rule that points to the new rule for while. It takes a `while` keyword, followed by a parenthesized condition expression, then a statement for the body. That new grammar rule gets a syntax tree node.

我们在`statement`规则中添加一个子句，指向while对应的新规则`whileStmt`。该规则接收一个`while`关键字，后跟一个带括号的条件表达式，然后是循环体对应的语句。新语法规则需要定义新的语法树节点。

tool/GenerateAst.java,在 main()方法中新增，前一行后添加","

```
      "Print      : Expr expression",
      "Var        : Token name, Expr initializer",
      // 新增部分开始
      "While      : Expr condition, Stmt body"
      // 新增部分结束
    ));
```

> The node stores the condition and body. Here you can see why it's nice to have separate base classes for expressions and statements. The field declarations make it clear that the condition is an expression and the body is a statement.

该节点中保存了条件式和循环体。这里就可以看出来为什么表达式和语句最好要有单独的基类。字段声明清楚地表明了，条件是一个表达式，循环主体是一个语句。

> Over in the parser, we follow the same process we used for `if` statements. First, we add another case in `statement()` to detect and match the leading keyword.

在解析器中，我们遵循与`if`语句相同的处理步骤。首先，在 `statement()` 添加一个case分支检查并匹配开头的关键字。

lox/Parser.java，在statement()方法中添加：

```
    if (match(PRINT)) return printStatement();
    // 新增部分开始
    if (match(WHILE)) return whileStatement();
    // 新增部分结束
    if (match(LEFT_BRACE)) return new Stmt.Block(block());
```

> That delegates the real work to this method:

实际的工作委托给下面的方法：

lox/Parser.java，在 varDeclaration()方法后添加：

```
  private Stmt whileStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'while'.");
    Expr condition = expression();
    consume(RIGHT_PAREN, "Expect ')' after condition.");
    Stmt body = statement();

    return new Stmt.While(condition, body);
  }
```

> The grammar is dead simple and this is a straight translation of it to Java. Speaking of translating straight to Java, here’s how we execute the new syntax:

语法非常简单，这里将其直接翻译为Java。说到直接翻译成Java，下面是我们执行新语法的方式：

lox/Interpreter.java，在 visitVarStmt()方法后添加：

```
  @Override
  public Void visitWhileStmt(Stmt.While stmt) {
    while (isTruthy(evaluate(stmt.condition))) {
      execute(stmt.body);
    }
    return null;
  }
```

> Like the visit method for `if`, this visitor uses the corresponding Java feature. This method isn’t complex, but it makes Lox much more powerful. We can finally write a program whose running time isn’t strictly bound by the length of the source code.

和`if`的访问方法一样，这里的访问方法使用了相应的Java特性。这个方法并不复杂，但它使Lox变得更加强大。我们终于可以编写一个运行时间不受源代码长度严格限制的程序了。

9.5For Loops

9.5 For循环

> We're down to the last control flow construct, Ye Olde C-style `for` loop. I probably don't need to remind you, but it looks like this:

我们已经到了最后一个控制流结构，即老式的C语言风格`for`循环。我可能不需要提醒你，但还是要说它看起来是这样的：

```
for (var i = 0; i < 10; i = i + 1) print i;
```

> In grammarese, that's:

在语法中，是这样的：

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | whileStmt
               | block ;

forStmt        → "for" "(" ( varDecl | exprStmt | ";" )
                 expression? ";"
                 expression? ")" statement ;
```

> Inside the parentheses, you have three clauses separated by semicolons:

在括号内，有三个由分号分隔的子语句：

1. > The first clause is the *initializer*. It is executed exactly once, before anything else. It's usually an expression, but for convenience, we also allow a variable declaration. In that case, the variable is scoped to the rest of the `for` loop—the other two clauses and the body.

   第一个子句是*初始化式*。它只会在任何其它操作之前执行一次。它通常是一个表达式，但是为了便利，我们也允许一个变量声明。在这种情况下，变量的作用域就是`for`循环的其它部分——其余两个子式和循环体。

2. > Next is the *condition*. As in a `while` loop, this expression controls when to exit the loop. It's evaluated once at the beginning of each iteration, including the first. If the result is truthy, it executes the loop body. Otherwise, it bails.

   接下来是*条件表达式*。与`while`循环一样，这个表达式控制了何时退出循环。它会在每次循环开始之前执行一次（包括第一次）。如果结果是真，就执行循环体；否则，就结束循环。

> 3. The last clause is the *increment*. It's an arbitrary expression that does some work at the end of each loop iteration. The result of the expression is discarded, so it must have a side effect to be useful. In practice, it usually increments a variable.

最后一个子句是*增量式*。它是一个任意的表达式，会在每次循环结束的时候做一些工作。因为表达式的结果会被丢弃，所以它必须有副作用才能有用。在实践中，它通常会对变量进行递增。

> Any of these clauses can be omitted. Following the closing parenthesis is a statement for the body, which is typically a block.

这些子语句都可以忽略。在右括号之后是一个语句作为循环体，通常是一个代码块。

> 9.5.1Desugaring

9.5.1 语法脱糖

> That's a lot of machinery, but note that none of it does anything you couldn't do with the statements we already have. If `for` loops didn't support initializer clauses, you could just put the initializer expression before the `for` statement. Without an increment clause, you could simply put the increment expression at the end of the body yourself.

这里包含了很多配件，但是请注意，它所做的任何事情中，没有一件是无法用已有的语句实现的。如果`for`循环不支持初始化子句，你可以在`for`语句之前加一条初始化表达式。如果没有增量子语句，你可以直接把增量表达式放在循环体的最后。

> In other words, Lox doesn't *need* `for` loops, they just make some common code patterns more pleasant to write. These kinds of features are called **syntactic sugar**. For example, the previous `for` loop could be rewritten like so:

换句话说，Lox不*需要*`for`循环，它们只是让一些常见的代码模式更容易编写。这类功能被称为**语法糖**^6。例如，前面的`for`循环可以改写成这样：

```
{
  var i = 0;
  while (i < 10) {
    print i;
    i = i + 1;
  }
}
```

> This script has the exact same semantics as the previous one, though it's not as easy on the eyes. Syntactic sugar features like Lox's `for` loop make a language more pleasant and productive to work in. But, especially in sophisticated language implementations, every language feature that requires back-end support and optimization is expensive.

虽然这个脚本不太容易看懂，但这个脚本与之前那个语义完全相同。像Lox中的`for`循环这样的语法糖特性可以使语言编写起来更加愉快和高效。但是，特别是在复杂的语言实现中，每一个需要后端支持和优化的语言特性都是代价昂贵的。

> We can have our cake and eat it too by **desugaring**. That funny word describes a process where the front end takes code using syntax sugar and translates it to a more primitive form that the back end already knows how to execute.

我们可以通过**脱糖**来吃这个蛋糕。这个有趣的词描述了这样一个过程：前端接收使用了语法糖的代码，并将其转换成后端知道如何执行的更原始的形式。

> We're going to desugar `for` loops to the `while` loops and other statements the interpreter already handles. In our simple interpreter, desugaring really doesn't save us much work, but it does give me an excuse to introduce you to the technique. So, unlike the previous statements, we *won't* add a new syntax tree node. Instead, we go straight to parsing. First, add an import we'll need soon.

我们将把`for`循环脱糖为`while`循环和其它解释器可处理的其它语句。在我们的简单解释器中，脱糖真的不能为我们节省很多工作，但它确实给了我一个契机来向你介绍这一技术。因此，与之前的语句不同，我们不会为`for`循环添加一个新的语法树节点。相反，我们会直接进行解析。首先，先引入一个我们要用到的依赖：

lox/Parser.java，添加代码：

```
import java.util.ArrayList;
// 新增部分开始
import java.util.Arrays;
// 新增部分结束
import java.util.List;
```

> Like every statement, we start parsing a `for` loop by matching its keyword.

像每个语句一样，我们通过匹配`for`关键字来解析循环。

lox/Parser.java,在 statement()方法中新增：

```
  private Stmt statement() {
    // 新增部分开始
    if (match(FOR)) return forStatement();
    // 新增部分结束
    if (match(IF)) return ifStatement();
```

> Here is where it gets interesting. The desugaring is going to happen here, so we'll build this method a piece at a time, starting with the opening parenthesis before the clauses.

接下来是有趣的部分，脱糖也是在这里发生的，所以我们会一点点构建这个方法，首先从子句之前的左括号开始。

lox/Parser.java，在 statement()方法后添加：

```
  private Stmt forStatement() {
    consume(LEFT_PAREN, "Expect '(' after 'for'.");
    // More here...
  }
```

> The first clause following that is the initializer.

接下来的第一个子句是初始化式。

lox/Parser.java，在 forStatement()方法中替换一行：

```
      consume(LEFT_PAREN, "Expect '(' after 'for'.");
      // 替换部分开始
      Stmt initializer;
      if (match(SEMICOLON)) {
        initializer = null;
      } else if (match(VAR)) {
        initializer = varDeclaration();
      } else {
        initializer = expressionStatement();
      }
      // 替换部分结束
    }
```

> If the token following the `(` is a semicolon then the initializer has been omitted. Otherwise, we check for a `var` keyword to see if it's a variable declaration. If neither of those matched, it must be an expression. We parse that and wrap it in an expression statement so that the initializer is always of type Stmt.

如果`(`后面的标记是分号，那么初始化式就被省略了。否则，我们就检查`var`关键字，看它是否是一个变量声明。如果这两者都不符合，那么它一定是一个表达式。我们对其进行解析，并将其封装在一个表达式语句中，这样初始化器就必定属于Stmt类型。

> Next up is the condition.

接下来是条件表达式。

lox/Parser.java，在 forStatement()方法中添加代码：

```
        initializer = expressionStatement();
      }
      // 新增部分开始
      Expr condition = null;
      if (!check(SEMICOLON)) {
        condition = expression();
      }
      consume(SEMICOLON, "Expect ';' after loop condition.");
      // 新增部分结束
    }
```

> Again, we look for a semicolon to see if the clause has been omitted. The last clause is the increment.

同样，我们查找分号检查子句是否被忽略。最后一个子句是增量语句。

lox/Parser.java，在 forStatement()方法中添加：

```
    consume(SEMICOLON, "Expect ';' after loop condition.");
    // 新增部分开始
    Expr increment = null;
    if (!check(RIGHT_PAREN)) {
      increment = expression();
    }
    consume(RIGHT_PAREN, "Expect ')' after for clauses.");
    // 新增部分结束
  }
```

> It's similar to the condition clause except this one is terminated by the closing parenthesis. All that remains is the body.

它类似于条件式子句，只是这个子句是由右括号终止的。剩下的就是循环主体了。

lox/Parser.java，在 forStatement()方法中添加代码：

```
    consume(RIGHT_PAREN, "Expect ')' after for clauses.");
    // 新增部分开始
    Stmt body = statement();

    return body;
    // 新增部分结束
  }
```

> We've parsed all of the various pieces of the `for` loop and the resulting AST nodes are sitting in a handful of Java local variables. This is where the desugaring comes in. We take those and use them to synthesize syntax tree nodes that express the semantics of the `for` loop, like the hand-desugared example I showed you earlier.

我们已经解析了`for`循环的所有部分，得到的AST节点也存储在一些Java本地变量中。这里也是脱糖开始的地方。我们利用这些变量来合成表示`for`循环语义的语法树节点，就像前面展示的手工脱糖的例子一样。

> The code is a little simpler if we work backward, so we start with the increment clause.

如果我们从后向前处理，代码会更简单一些，所以我们从增量子句开始。

lox/Parser.java，在 forStatement()方法中新增：

```
    Stmt body = statement();
    // 新增部分开始
    if (increment != null) {
      body = new Stmt.Block(
          Arrays.asList(
              body,
              new Stmt.Expression(increment)));
    }
    // 新增部分结束
    return body;
```

> The increment, if there is one, executes after the body in each iteration of the loop. We do that by replacing the body with a little block that contains the original body followed by an expression statement that evaluates the increment.

如果存在增量子句的话，会在循环的每个迭代中在循环体结束之后执行。我们用一个代码块来代替循环体，这个代码块中包含原始的循环体，后面跟一个执行增量子语句的表达式语句。

lox/Parser.java，在 forStatement()方法中新增代码：

```
      }
      // 新增部分开始
      if (condition == null) condition = new Expr.Literal(true);
      body = new Stmt.While(condition, body);
      // 新增部分结束
      return body;
```

> Next, we take the condition and the body and build the loop using a primitive `while` loop. If the condition is omitted, we jam in `true` to make an infinite loop.

接下来，我们获取条件式和循环体，并通过基本的`while`语句构建对应的循环。如果条件式被省略了，我们就使用`true`来创建一个无限循环。

lox/Parser.java，在 forStatement()方法中新增：

```
      body = new Stmt.While(condition, body);
      // 新增部分开始
      if (initializer != null) {
        body = new Stmt.Block(Arrays.asList(initializer, body));
      }
      // 新增部分结束
      return body;
```

> Finally, if there is an initializer, it runs once before the entire loop. We do that by, again, replacing the whole statement with a block that runs the initializer and then executes the loop.

最后，如果有初始化式，它会在整个循环之前运行一次。我们的做法是，再次用代码块来替换整个语句，该代码块中首先运行一个初始化式，然后执行循环。

> That's it. Our interpreter now supports C-style `for` loops and we didn't have to touch the Interpreter class at all. Since we desugared to nodes the interpreter already knows how to visit, there is no more

> work to do.

就是这样。我们的解释器现在已经支持了C语言风格的`for`循环，而且我们根本不需要修改解释器类。因为我们通过脱糖将其转换为了解释器已经知道如何访问的节点，所以无需做其它的工作。

> Finally, Lox is powerful enough to entertain us, at least for a few minutes. Here's a tiny program to print the first 21 elements in the Fibonacci sequence:

最后，Lox已强大到足以娱乐我们，至少几分钟。下面是一个打印斐波那契数列前21个元素的小程序：

```
var a = 0;
var temp;

for (var b = 1; a < 10000; b = temp + b) {
  print a;
  temp = a;
  a = b;
}
```

^2: 图灵把他的发明称为 "a-machines"，表示"automatic(自动)"。他并没有自吹自擂到把自己的名字放入其中。后来的数学家们为他做了这些。这就是你如何在成名的同时还能保持谦虚。 ^3: 我们也基本上具备第三个条件了。你可以创建和拼接任意大小的字符串，因此也就可以存储无界内存。但我们还无法访问字符串的各个部分。 ^4: 条件操作符也称为三元操作符，因为它是C语言中唯一接受三个操作数的操作符。 ^5: 条件周围的圆括号只有一半是有用的。您需要在条件和then语句之间设置某种分隔符，否则解析器无法判断是否到达条件表达式的末尾。但是 `if` 后面的小括号并没有什么用处。Dennis Ritchie 把它放在那里是为了让他可以使用 `)`作为结尾的分隔符，而且不会出现不对称的小括号。其他语言，比如Lua和一些BASICs，使用`then`这样的关键字作为结束分隔符，在条件表达式之前没有任何内容。而Go和Swift则要求语句必须是一个带括号的块，这样就可以使用语句开头的`{`来判断条件表达式是否结束。 ^6: 这个令人愉快的短语是由Peter J. Landin在1964年创造的，用来描述ALGOL等语言支持的一些很好的表达式形式是如何在更基本但可能不太令人满意的lambda演算的基础上增添一些小甜头的。

> ## CHALLENGES

习题

> 1、A few chapters from now, when Lox supports first-class functions and dynamic dispatch, we technically won't *need* branching statements built into the language. Show how conditional execution can be implemented in terms of those. Name a language that uses this technique for its control flow.

1、在接下来的几章中，当Lox支持一级函数和动态调度时，从技术上讲，我们就不需要在语言中内置分支语句。说明如何用这些特性来实现条件执行。说出一种在控制流中使用这种技术的语言。

> 2、Likewise, looping can be implemented using those same tools, provided our interpreter supports an important optimization. What is it, and why is it necessary? Name a language that uses this technique for iteration.

2、同样地，只要我们的解释器支持一个重要的优化，循环也可以用这些工具来实现。它是什么？为什么它是必要的？请说出一种使用这种技术进行迭代的语言。

> 3、Unlike Lox, most other C-style languages also support `break` and `continue` statements inside loops. Add support for `break` statements.
>
> The syntax is a `break` keyword followed by a semicolon. It should be a syntax error to have a `break` statement appear outside of any enclosing loop. At runtime, a `break` statement causes execution to jump to the end of the nearest enclosing loop and proceeds from there. Note that the `break` may be nested inside other blocks and `if` statements that also need to be exited.

3、与Lox不同，大多数其他C风格语言也支持循环内部的`break`和`continue`语句。添加对`break`语句的支持。

语法是一个`break`关键字，后面跟一个分号。如果`break`语句出现在任何封闭的循环之后，那就应该是一个语法错误。在运行时，`break`语句会跳转到最内层的封闭循环的末尾，并从那里开始继续执行。注意，`break`语句可以嵌套在其它需要退出的代码块和`if`语句中。

> ## DESIGN NOTE: SPOONFULS OF SYNTACTIC SUGAR

设计笔记：一些语法糖

> When you design your own language, you choose how much syntactic sugar to pour into the grammar. Do you make an unsweetened health food where each semantic operation maps to a single syntactic unit, or some decadent dessert where every bit of behavior can be expressed ten different ways? Successful languages inhabit all points along this continuum.
>
> On the extreme acrid end are those with ruthlessly minimal syntax like Lisp, Forth, and Smalltalk. Lispers famously claim their language “has no syntax”, while Smalltalkers proudly show that you can fit the entire grammar on an index card. This tribe has the philosophy that the *language* doesn’t need syntactic sugar. Instead, the minimal syntax and semantics it provides are powerful enough to let library code be as expressive as if it were part of the language itself.
>
> Near these are languages like C, Lua, and Go. They aim for simplicity and clarity over minimalism. Some, like Go, deliberately eschew both syntactic sugar and the kind of syntactic extensibility of the previous category. They want the syntax to get out of the way of the semantics, so they focus on keeping both the grammar and libraries simple. Code should be obvious more than beautiful.
>
> Somewhere in the middle you have languages like Java, C#, and Python. Eventually you reach Ruby, C++, Perl, and D—languages which have stuffed so much syntax into their grammar, they are running out of punctuation characters on the keyboard.
>
> To some degree, location on the spectrum correlates with age. It’s relatively easy to add bits of syntactic sugar in later releases. New syntax is a crowd pleaser, and it’s less likely to break existing programs than mucking with the semantics. Once added, you can never take it away, so languages tend to sweeten with time. One of the main benefits of creating a new language from scratch is it gives you an opportunity to scrape off those accumulated layers of frosting and start over.
>
> Syntactic sugar has a bad rap among the PL intelligentsia. There’s a real fetish for minimalism in that crowd. There is some justification for that. Poorly designed, unneeded syntax raises the cognitive load without adding enough expressiveness to carry its weight. Since there is always pressure to cram new features into the language, it takes discipline and a focus on simplicity to avoid bloat. Once you add some syntax, you’re stuck with it, so it’s smart to be parsimonious.

> At the same time, most successful languages do have fairly complex grammars, at least by the time they are widely used. Programmers spend a ton of time in their language of choice, and a few niceties here and there really can improve the comfort and efficiency of their work.
>
> Striking the right balance—choosing the right level of sweetness for your language—relies on your own sense of taste.

当你设计自己的语言时，你可以选择在语法中注入多少语法糖。你是要做一种不加糖、每个语法操作都对应单一的语法单元的健康食品？还是每一点行为都可以用10种不同方式实现的堕落的甜点？把这两种情况看作是两端的话，成功的语言分布在这个连续体的每个中间点。

极端尖刻的一侧是那些语法极少的语言，如Lisp、Forth和SmallTalk。Lisp的拥趸广泛声称他们的语言 "没有语法"，而Smalltalk的人则自豪地表示，你可以把整个语法放在一张索引卡上。这个部落的理念是，语言不需要句法糖。相反，它所提供的最小的语法和语义足够强大，足以让库中的代码像语言本身的一部分一样具有表现力。

接近这些的是像C、Lua和Go这样的语言。他们的目标是简单和清晰，而不是极简主义。有些语言，如Go，故意避开了语法糖和前一类语言的语法扩展性。他们希望语法不受语义的影响，所以他们专注于保持语法和库的简单性。代码应该是明显的，而不是漂亮的。

介于之间的是Java、C#和Python等语言。最终，你会看到Ruby、C++、Perl和D-语言，它们在语法中塞入了太多的句法规则，以至于键盘上的标点符号都快用完了。

在某种程度上，频谱上的位置与年龄相关。在后续的版本中增加一些语法糖是比较容易的。新的语法很容易让人喜欢，而且与修改语义相比，它更不可能破坏现有的程序。一旦加进去，你就再也不能把它去掉了，所以随着时间的推移，语言会变得越来越甜。从头开始创建一门新语言的主要好处之一是，它给了你一个机会去刮掉那些累积的糖霜并重新开始。

语法糖在PL知识分子中名声不佳。那群人对极简主义有一种真正的迷恋。这是有一定道理的。设计不良的、不必要的语法增加了认知负荷，却没有增加相匹配的表达能力。因为一直会有向语言中添加新特性的压力，所以需要自律并专注于简单，以避免臃肿。一旦你添加了一些语法，你就会被它困住，所以明智的做法是要精简。

同时，大多数成功的语言都有相当复杂的语法，至少在它们被广泛使用的时候是这样。程序员在他们所选择的语言上花费了大量的时间，一些随处可见的细节确实可以提高他们工作时的舒适度和效率。

找到正确的平衡——为你的语言选择适当的甜度——取决于你自己的品味。

10.函数Functions

> And that is also the way the human mind works—by the compounding of old ideas into new structures that become new ideas that can themselves be used in compounds, and round and round endlessly, growing ever more remote from the basic earthbound imagery that is each language’s soil.
>
> —— Douglas R. Hofstadter, *I Am a Strange Loop*

这也是人类思维的运作方式——将旧的想法复合成为新结构，成为新的想法，而这些想法本身又可以被用于复合，循环往复，无休无止，越来越远离每一种语言赖以生存的基本的土壤。

> This chapter marks the culmination of a lot of hard work. The previous chapters add useful functionality in their own right, but each also supplies a piece of a puzzle. We’ll take those pieces—expressions,

> statements, variables, control flow, and lexical scope—add a couple more, and assemble them all into support for real user-defined functions and function calls.

这一章标志着很多艰苦工作的一个高潮。在前面的章节中，各自添加了一些有用的功能，但是每一章也都提供了一个拼图的碎片。我们整理这些碎片——表达式、语句、变量、控制流和词法作用域，再加上其它功能，并把他们组合起来，以支持真正的用户定义函数和函数调用。

> ## 10.1 Function Calls

10.1 函数调用

> You're certainly familiar with C-style function call syntax, but the grammar is more subtle than you may realize. Calls are typically to named functions like:

你肯定熟悉C语言风格的函数调用语法，但其语法可能比你意识到的更微妙。调用通常是指向命名的函数，例如：

```
average(1, 2);
```

> But the name of the function being called isn't actually part of the call syntax. The thing being called—the **callee**—can be any expression that evaluates to a function. (Well, it does have to be a pretty *high precedence* expression, but parentheses take care of that.) For example:

但是被调用函数的名称实际上并不是调用语法的一部分。被调用者（ **callee** ）可以是计算结果为一个函数的任何表达式。(好吧，它必须是一个非常高优先级的表达式，但是圆括号可以解决这个问题。)例如：

```
getCallback()();
```

> There are two call expressions here. The first pair of parentheses has `getCallback` as its callee. But the second call has the entire `getCallback()` expression as its callee. It is the parentheses following an expression that indicate a function call. You can think of a call as sort of like a postfix operator that starts with `(`.

这里有两个函数调用。第一对括号将`getCallback`作为其被调用者。但是第二对括号将整个`getCallback()` 表达式作为其被调用者。表达式后面的小括号表示函数调用，你可以把调用看作是一种以`(`开头的后缀运算符。

> This "operator" has higher precedence than any other operator, even the unary ones. So we slot it into the grammar by having the `unary` rule bubble up to a new `call` rule.

这个"运算符"比其它运算符（包括一元运算符）有更高的优先级。所以我们通过让`unary`规则跳转到新的`call`规则，将其添加到语法中^1。

```
unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")" )* ;
```

> This rule matches a primary expression followed by zero or more function calls. If there are no parentheses, this parses a bare primary expression. Otherwise, each call is recognized by a pair of parentheses with an optional list of arguments inside. The argument list grammar is:

该规则匹配一个基本表达式，后面跟着0个或多个函数调用。如果没有括号，则解析一个简单的基本表达式。否则，每一对圆括号都表示一个函数调用，圆括号内有一个可选的参数列表。参数列表语法是：

```
arguments      → expression ( "," expression )* ;
```

> This rule requires at least one argument expression, followed by zero or more other expressions, each preceded by a comma. To handle zero-argument calls, the `call` rule itself considers the entire `arguments` production to be optional.

这个规则要求至少有一个参数表达式，后面可以跟0个或多个其它表达式，每两个表达式之间用`,`分隔。为了处理无参调用，`call`规则本身认为整个`arguments`生成式是可选的。

> I admit, this seems more grammatically awkward than you'd expect for the incredibly common "zero or more comma-separated things" pattern. There are some sophisticated metasyntaxes that handle this better, but in our BNF and in many language specs I've seen, it is this cumbersome.

我承认，对于极其常见的 "零或多个逗号分隔的事物 "模式来说，这在语法上似乎比你想象的更难处理。有一些复杂的元语法可以更好地处理这个问题，但在我们的BNF和我见过的许多语言规范中，它就是如此的麻烦。

> Over in our syntax tree generator, we add a new node.

在我们的语法树生成器中，我们添加一个新节点。

tool/GenerateAst.java，在 main()方法中添加代码：

```
      "Binary   : Expr left, Token operator, Expr right",
      // 新增部分开始
      "Call     : Expr callee, Token paren, List<Expr> arguments",
      // 新增部分结束
      "Grouping : Expr expression",
```

> It stores the callee expression and a list of expressions for the arguments. It also stores the token for the closing parenthesis. We'll use that token's location when we report a runtime error caused by a function call.

它存储了被调用者表达式和参数表达式列表，同时也保存了右括号标记。当我们报告由函数调用引起的运行时错误时，会使用该标记的位置。

> Crack open the parser. Where `unary()` used to jump straight to `primary()`, change it to call, well, `call()`.

打开解析器，原来`unary()`直接跳转到`primary()`方法，将其修改为调用`call()`。

lox/Parser.java，在 unary()方法中替换一行：

```
      return new Expr.Unary(operator, right);
    }
    // 替换部分开始
    return call();
    // 替换部分结束
  }
```

> Its definition is:

该方法定义为：

lox/Parser.java，在 unary()方法后添加^2：

```
  private Expr call() {
    Expr expr = primary();

    while (true) {
      if (match(LEFT_PAREN)) {
        expr = finishCall(expr);
      } else {
        break;
      }
    }

    return expr;
  }
```

> The code here doesn't quite line up with the grammar rules. I moved a few things around to make the code cleaner—one of the luxuries we have with a handwritten parser. But it's roughly similar to how we parse infix operators. First, we parse a primary expression, the "left operand" to the call. Then, each time we see a `(`, we call `finishCall()` to parse the call expression using the previously parsed expression as the callee. The returned expression becomes the new `expr` and we loop to see if the result is itself called.

这里的代码与语法规则并非完全一致。为了保持代码简洁，我调整了一些东西——这是我们手写解析器的优点之一。但它与我们解析中缀运算符的方式类似。首先，我们解析一个基本表达式，即调用的左操作数。然后，每次看到`(`，我们就调用`finishCall()`解析调用表达式，并使用之前解析出的表达式作为被调用者。返回的表达式成为新的`expr`，我们循环检查其结果是否被调用。

> The code to parse the argument list is in this helper:

解析参数列表的代码在下面的工具方法中：

lox/Parser.java，在 unary()方法后添加：

```
  private Expr finishCall(Expr callee) {
    List<Expr> arguments = new ArrayList<>();
    if (!check(RIGHT_PAREN)) {
      do {
        arguments.add(expression());
      } while (match(COMMA));
    }

    Token paren = consume(RIGHT_PAREN,
                          "Expect ')' after arguments.");

    return new Expr.Call(callee, paren, arguments);
  }
```

> This is more or less the `arguments` grammar rule translated to code, except that we also handle the zero-argument case. We check for that case first by seeing if the next token is `)`. If it is, we don't try to parse any arguments.

这或多或少是`arguments` 语法规则翻译成代码的结果，除了我们这里还处理了无参情况。我们首先判断下一个标记是否`)`来检查这种情况。如果是，我们就不会尝试解析任何参数。

> Otherwise, we parse an expression, then look for a comma indicating that there is another argument after that. We keep doing that as long as we find commas after each expression. When we don't find a comma, then the argument list must be done and we consume the expected closing parenthesis. Finally, we wrap the callee and those arguments up into a call AST node.

如果不是，我们就解析一个表达式，然后寻找逗号（表明后面还有一个参数）。只要我们在表达式后面发现逗号，就会继续解析表达式。当我们找不到逗号时，说明参数列表已经结束，我们继续消费预期的右括号。最终，我们将被调用者和这些参数封装成一个函数调用的AST节点。

> ### 10.1.1 Maximum argument counts

10.1.1 最大参数数量

> Right now, the loop where we parse arguments has no bound. If you want to call a function and pass a million arguments to it, the parser would have no problem with it. Do we want to limit that?

现在，我们解析参数的循环是没有边界的。如果你想调用一个函数并向其传递一百万个参数，解析器不会有任何问题。我们要对此进行限制吗？

> Other languages have various approaches. The C standard says a conforming implementation has to support *at least* 127 arguments to a function, but doesn't say there's any upper limit. The Java specification says a method can accept *no more than* 255 arguments.

其它语言采用了不同的策略。C语言标准要求在符合标准的实现中，一个函数至少要支持127个参数，但是没有指定任何上限。Java规范规定一个方法可以接受不超过255个参数^3。

> Our Java interpreter for Lox doesn't really need a limit, but having a maximum number of arguments will simplify our bytecode interpreter in Part III. We want our two interpreters to be compatible with each other, even in weird corner cases like this, so we'll add the same limit to jlox.

Lox的Java解释器实际上并不需要限制，但是设置一个最大的参数数量限制可以简化第三部分中的字节码解释器。即使是在这样奇怪的地方里，我们也希望两个解释器能够相互兼容，所以我们为jlox添加同样的限制。

lox/Parser.java，在 finishCall()方法中添加：

```
      do {
        // 新增部分开始
        if (arguments.size() >= 255) {
          error(peek(), "Can't have more than 255 arguments.");
        }
        // 新增部分结束
        arguments.add(expression());
```

> Note that the code here *reports* an error if it encounters too many arguments, but it doesn't *throw* the error. Throwing is how we kick into panic mode which is what we want if the parser is in a confused state and doesn't know where it is in the grammar anymore. But here, the parser is still in a perfectly valid state—it just found too many arguments. So it reports the error and keeps on keepin' on.

请注意，如果发现参数过多，这里的代码会*报告*一个错误，但是不会*抛出*该错误。抛出错误是进入恐慌模式的方法，如果解析器处于混乱状态，不知道自己在语法中处于什么位置，那这就是我们想要的。但是在这里，解析器仍然处于完全有效的状态，只是发现了太多的参数。所以它会报告这个错误，并继续执行解析。

> ### 10.1.2 Interpreting function calls

10.1.2 解释函数调用

> We don't have any functions we can call, so it seems weird to start implementing calls first, but we'll worry about that when we get there. First, our interpreter needs a new import.

我们还没有任何可以调用的函数，所以先实现函数调用似乎有点奇怪，但是这个问题我们后面再考虑。首先，我们的解释器需要引入一个新依赖。

lox/Interpreter.java

```
// 新增部分开始
import java.util.ArrayList;
// 新增部分结束
import java.util.List;
```

> As always, interpretation starts with a new visit method for our new call expression node.

跟之前一样，解释工作从新的调用表达式节点对应的新的visit方法开始^4。

lox/Interpreter.java，在 visitBinaryExpr()方法后添加：

```
  @Override
  public Object visitCallExpr(Expr.Call expr) {
    Object callee = evaluate(expr.callee);

    List<Object> arguments = new ArrayList<>();
    for (Expr argument : expr.arguments) {
      arguments.add(evaluate(argument));
    }

    LoxCallable function = (LoxCallable)callee;
    return function.call(this, arguments);
  }
```

> First, we evaluate the expression for the callee. Typically, this expression is just an identifier that looks up the function by its name, but it could be anything. Then we evaluate each of the argument expressions in order and store the resulting values in a list.

首先，对被调用者的表达式求值。通常情况下，这个表达式只是一个标识符，可以通过它的名字来查找函数。但它可以是任何东西。然后，我们依次对每个参数表达式求值，并将结果值存储在一个列表中。

> Once we've got the callee and the arguments ready, all that remains is to perform the call. We do that by casting the callee to a LoxCallable and then invoking a `call()` method on it. The Java representation of any Lox object that can be called like a function will implement this interface. That includes user-defined functions, naturally, but also class objects since classes are "called" to construct new instances. We'll also use it for one more purpose shortly.

一旦我们准备好被调用者和参数，剩下的就是执行函数调用。我们将被调用者转换为LoxCallable，然后对其调用`call()`方法来实现。任何可以像函数一样被调用的Lox对象的Java表示都要实现这个接口。这自然包括用户定义的函数，但也包括类对象，因为类会被 "调用 "来创建新的实例。稍后我们还将把它用于另一个目的。

> There isn't too much to this new interface.

这个新接口中没有太多内容。

lox/LoxCallable.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

interface LoxCallable {
  Object call(Interpreter interpreter, List<Object> arguments);
}
```

> We pass in the interpreter in case the class implementing `call()` needs it. We also give it the list of evaluated argument values. The implementer's job is then to return the value that the call expression produces.

我们会传入解释器，以防实现`call()`方法的类会需要它。我们也会提供已求值的参数值列表。接口实现者的任务就是返回调用表达式产生的值。

> ## 10.1.3 Call type errors

10.1.3 调用类型错误

> Before we get to implementing LoxCallable, we need to make the visit method a little more robust. It currently ignores a couple of failure modes that we can’t pretend won’t occur. First, what happens if the callee isn’t actually something you can call? What if you try to do this:

在实现`LoxCallable`之前，必须先强化一下我们的visit方法。这个方法忽略了两个可能出现的错误场景。第一个，如果被调用者无法被调用，会发生什么？比如：

```
"totally not a function"();
```

> Strings aren’t callable in Lox. The runtime representation of a Lox string is a Java string, so when we cast that to LoxCallable, the JVM will throw a ClassCastException. We don’t want our interpreter to vomit out some nasty Java stack trace and die. Instead, we need to check the type ourselves first.

在Lox中，字符串不是可调用的数据类型。Lox字符串在运行时中的本质其实是java字符串，所以当我们把它当作`LoxCallable` 处理的时候，JVM就会抛出`ClassCastException`。我们并不想让我们的解释器吐出一坨java堆栈信息然后挂掉。所以，我们自己必须先做一次类型检查。

lox/Interpreter.java，在visitCallExpr接口中新增：

```
    // 新增部分开始
    if (!(callee instanceof LoxCallable)) {
      throw new RuntimeError(expr.paren,
          "Can only call functions and classes.");
    }
    // 新增部分结束
    LoxCallable function = (LoxCallable)callee;
```

We still throw an exception, but now we’re throwing our own exception type, one that the interpreter knows to catch and report gracefully.

我们的实现同样也是抛出错误，但它们能够被解释器捕获并优雅地展示出来。

> ### 10.1.4 Checking arity

10.1.4 检查元数

> The other problem relates to the function’s **arity**. Arity is the fancy term for the number of arguments a function or operation expects. Unary operators have arity one, binary operators two, etc. With functions, the arity is determined by the number of parameters it declares.

另一个问题与函数的**元数**有关。元数是一个花哨的术语，指一个函数或操作所期望的参数数量。一元运算符的元数是1，二元运算符是2，等等。对于函数来说，元数由函数声明的参数数量决定。

```
fun add(a, b, c) {
  print a + b + c;
}
```

> This function defines three parameters, a, b, and c, so its arity is three and it expects three arguments. So what if you try to call it like this:

这个函数定义了三个形参，a、b 和c，所以它的元数是3，而且它期望有3个参数。那么如果你用下面的方式调用该函数会怎样：

```
add(1, 2, 3, 4); // Too many.
add(1, 2);       // Too few.
```

> Different languages take different approaches to this problem. Of course, most statically typed languages check this at compile time and refuse to compile the code if the argument count doesn't match the function's arity. JavaScript discards any extra arguments you pass. If you don't pass enough, it fills in the missing parameters with the magic sort-of-like-null-but-not-really value `undefined`. Python is stricter. It raises a runtime error if the argument list is too short or too long.

不同的语言对这个问题采用了不同的方法。当然，大多数静态类型的语言在编译时都会检查这个问题，如果实参与函数元数不匹配，则拒绝编译代码。JavaScript会丢弃你传递的所有多余参数。如果你没有传入的参数数量不足，它就会用神奇的与`null`类似但并不相同的值`undefined`来填补缺少的参数。Python更严格。如果参数列表太短或太长，它会引发一个运行时错误。

> I think the latter is a better approach. Passing the wrong number of arguments is almost always a bug, and it's a mistake I do make in practice. Given that, the sooner the implementation draws my attention to it, the better. So for Lox, we'll take Python's approach. Before invoking the callable, we check to see if the argument list's length matches the callable's arity.

我认为后者是一种更好的方法。传递错误的参数数量几乎总是一个错误，这也是我在实践中确实犯的一个错误。有鉴于此，语言实现能越早引起用户的注意就越好。所以对于Lox，我们将采取Python的方法。在执行可调用方法之前，我们检查参数列表的长度是否与可调用方法的元数相符。

lox/Interpreter.java，在 visitCallExpr()方法中添加代码：

```
    LoxCallable function = (LoxCallable)callee;
    // 新增部分开始
    if (arguments.size() != function.arity()) {
      throw new RuntimeError(expr.paren, "Expected " +
          function.arity() + " arguments but got " +
          arguments.size() + ".");
    }
    // 新增部分结束
    return function.call(this, arguments);
```

> That requires a new method on the LoxCallable interface to ask it its arity.

这就需要在`LoxCallable`接口中增加一个新方法来查询函数的元数。

lox/LoxCallable.java，在LoxCallable接口中新增：

```
interface LoxCallable {
  // 新增部分开始
  int arity();
  // 新增部分结束
  Object call(Interpreter interpreter, List<Object> arguments);
```

> We *could* push the arity checking into the concrete implementation of `call()`. But, since we'll have multiple classes implementing LoxCallable, that would end up with redundant validation spread across a few classes. Hoisting it up into the visit method lets us do it in one place.

我们可以在`call()`方法的具体实现中做元数检查。但是，由于我们会有多个实现LoxCallable的类，这将导致冗余的验证分散在多个类中。把它提升到访问方法中，这样我们可以在一个地方完成该功能。

> ## 10.2 Native Functions

10.2 原生函数（本地函数）

> We can theoretically call functions, but we have no functions to call yet. Before we get to user-defined functions, now is a good time to introduce a vital but often overlooked facet of language implementations—**native functions**. These are functions that the interpreter exposes to user code but that are implemented in the host language (in our case Java), not the language being implemented (Lox).

理论上我们可以调用函数了，但是我们还没有可供调用的函数。在我们实现用户自定义函数之前，现在正好可以介绍语言实现中一个重要但经常被忽视的方面——**原生函数（本地函数）**。这些函数是解释器向用户代码公开的，但它们是用宿主语言(在我们的例子中是Java)实现的，而不是正在实现的语言(Lox)。

> Sometimes these are called **primitives**, **external functions**, or **foreign functions**. Since these functions can be called while the user's program is running, they form part of the implementation's runtime. A lot of programming language books gloss over these because they aren't conceptually interesting. They're mostly grunt work.

有时这些函数也被称为**原语**、**外部函数**或**外来函数**^5。由于这些函数可以在用户程序运行的时候被调用，因此它们构成了语言运行时的一部分。许多编程语言书籍都掩盖了这些内容，因为它们在概念上并不有趣。它们主要是一些比较繁重的工作。

> But when it comes to making your language actually good at doing useful stuff, the native functions your implementation provides are key. They provide access to the fundamental services that all programs are defined in terms of. If you don't provide native functions to access the file system, a user's going to have a hell of a time writing a program that reads and displays a file.

但是说到让你的语言真正擅长做有用的事情，语言提供的本地函数是关键^6。本地函数提供了对基础服务的访问，所有的程序都是根据这些服务来定义的。如果你不提供访问文件系统的本地函数，那么用户在写一个读取和显示文件的程序时就会有很大的困难。

> Many languages also allow users to provide their own native functions. The mechanism for doing so is called a **foreign function interface** (**FFI**), **native extension**, **native interface**, or something along those lines. These are nice because they free the language implementer from providing access to every

> single capability the underlying platform supports. We won't define an FFI for jlox, but we will add one native function to give you an idea of what it looks like.

许多语言还允许用户提供自己的本地函数。这样的机制称为**外来函数接口(FFI)**、**本机扩展**、**本机接口**或类似的东西。这些机制很好，因为它们使语言实现者无需提供对底层平台所支持的每一项功能的访问。我们不会为 jlox 定义一个 FFI，但我们会添加一个本地函数，让你知道它是什么样子。

> ### 10.2.1 Telling time

10.2.1 报时

> When we get to Part III and start working on a much more efficient implementation of Lox, we're going to care deeply about performance. Performance work requires measurement, and that in turn means **benchmarks**. These are programs that measure the time it takes to exercise some corner of the interpreter.

当我们进入第三部分，开始着手开发更有效的Lox实现时，我们就会非常关心性能。性能需要测量，这也就意味着需要**基准测试**。这些代码就是用于测量解释器执行某些代码时所花费的时间。

> We could measure the time it takes to start up the interpreter, run the benchmark, and exit, but that adds a lot of overhead—JVM startup time, OS shenanigans, etc. That stuff does matter, of course, but if you're just trying to validate an optimization to some piece of the interpreter, you don't want that overhead obscuring your results.

我们可以测量启动解释器、运行基准测试代码并退出所消耗的时间，但是这其中包括很多时间开销——JVM启动时间，操作系统欺诈等等。当然，这些东西确实很重要，但如果您只是试图验证对解释器某个部分的优化，你肯定不希望这些多余的时间开销掩盖你的结果。

> A nicer solution is to have the benchmark script itself measure the time elapsed between two points in the code. To do that, a Lox program needs to be able to tell time. There's no way to do that now—you can't implement a useful clock "from scratch" without access to the underlying clock on the computer.

一个更好的解决方案是让基准脚本本身度量代码中两个点之间的时间间隔。要做到这一点，Lox程序需要能够报时。现在没有办法做到这一点——如果不访问计算机上的底层时钟，就无法从头实现一个可用的时钟。

> So we'll add `clock()`, a native function that returns the number of seconds that have passed since some fixed point in time. The difference between two successive invocations tells you how much time elapsed between the two calls. This function is defined in the global scope, so let's ensure the interpreter has access to that.

所以我们要添加`clock()`，这是一个本地函数，用于返回自某个固定时间点以来所经过的秒数。两次连续调用之间的差值可用告诉你两次调用之间经过了多少时间。这个函数被定义在全局作用域内，以确保解释器能够访问这个函数。

lox/Interpreter.java，在 Interpreter类中，替换一行：

```
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
  // 替换部分开始
  final Environment globals = new Environment();
  private Environment environment = globals;
  // 替换部分结束
  void interpret(List<Stmt> statements) {
```

> The `environment` field in the interpreter changes as we enter and exit local scopes. It tracks the *current* environment. This new `globals` field holds a fixed reference to the outermost global environment.

解释器中的`environment`字段会随着进入和退出局部作用域而改变，它会跟随当前环境。新加的`globals`字段则固定指向最外层的全局作用域。

> When we instantiate an Interpreter, we stuff the native function in that global scope.

当我们实例化一个解释器时，我们将全局作用域中添加本地函数。

lox/Interpreter.java，在 Interpreter类中新增：

```
  private Environment environment = globals;
  // 新增部分开始
  Interpreter() {
    globals.define("clock", new LoxCallable() {
      @Override
      public int arity() { return 0; }

      @Override
      public Object call(Interpreter interpreter,
                         List<Object> arguments) {
        return (double)System.currentTimeMillis() / 1000.0;
      }

      @Override
      public String toString() { return "<native fn>"; }
    });
  }
  // 新增部分结束
  void interpret(List<Stmt> statements) {
```

> This defines a variable named “clock”. Its value is a Java anonymous class that implements LoxCallable. The `clock()` function takes no arguments, so its arity is zero. The implementation of `call()` calls the corresponding Java function and converts the result to a double value in seconds.

这里有一个名为`clock`的变量，它的值是一个实现LoxCallable接口的Java匿名类。这里的`clock()`函数不接受参数，所以其元数为0。`call()`方法的实现是直接调用Java函数并将结果转换为以秒为单位的double值。

> If we wanted to add other native functions—reading input from the user, working with files, etc.—we could add them each as their own anonymous class that implements LoxCallable. But for the book, this one is really all we need.

如果我们想要添加其它本地函数——读取用户输入，处理文件等等——我们可以依次为它们提供实现LoxCallable接口的匿名类。但是在本书中，这个函数足以满足需要。

> Let's get ourselves out of the function-defining business and let our users take over . . .

让我们从函数定义的事务中解脱出来，由用户来接管吧。

10.3 Function Declarations

10.3 函数声明

> We finally get to add a new production to the `declaration` rule we introduced back when we added variables. Function declarations, like variables, bind a new name. That means they are allowed only in places where a declaration is permitted.

我们终于可以在添加变量时就引入的`declaration`规则中添加产生式了。就像变量一样，函数声明也会绑定一个新的名称。这意味中它们只能出现在允许声明的地方。

```
declaration    → funDecl
               | varDecl
               | statement ;
```

> The updated `declaration` rule references this new rule:

更新后的`declaration`引用了下面的新规则：

```
funDecl        → "fun" function ;
function       → IDENTIFIER "(" parameters? ")" block ;
```

> The main `funDecl` rule uses a separate helper rule `function`. A function *declaration statement* is the `fun` keyword followed by the actual function-y stuff. When we get to classes, we'll reuse that `function` rule for declaring methods. Those look similar to function declarations, but aren't preceded by `fun`.

主要的`funDecl`规则使用了一个单独的辅助规则`function`。函数*声明语句*是`fun`关键字后跟实际的函数体内容。等到我们实现类的时候，将会复用`function`规则来声明方法。这些方法与函数声明类似，但是前面没有`fun`。

> The function itself is a name followed by the parenthesized parameter list and the body. The body is always a braced block, using the same grammar rule that block statements use. The parameter list uses this rule:

函数本身是一个名称，后跟带括号的参数列表和函数体。函数体是一个带花括号的块，可以使用与块语句相同的语法。参数列表则使用以下规则：

```
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;
```

> It's like the earlier `arguments` rule, except that each parameter is an identifier, not an expression. That's a lot of new syntax for the parser to chew through, but the resulting AST node isn't too bad.

这就类似于前面的`arguments` 规则，区别在于参数是一个标识符，而不是一个表达式。这对于解析器来说是很多要处理的新语法，但是生成的AST节点没这么复杂。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Expression : Expr expression",
      // 新增部分开始
      "Function   : Token name, List<Token> params," +
                  " List<Stmt> body",
      // 新增部分结束
      "If         : Expr condition, Stmt thenBranch," +
```

> A function node has a name, a list of parameters (their names), and then the body. We store the body as the list of statements contained inside the curly braces.

函数节点有一个名称、一个参数列表(参数的名称)，然后是函数主体。我们将函数主体存储为包含在花括号中的语句列表。

> Over in the parser, we weave in the new declaration.

在解析器中，我们把新的声明添加进去。

lox/Parser.java，在 declaration()方法中添加：

```
    try {
      // 新增部分开始
      if (match(FUN)) return function("function");
      // 新增部分结束
      if (match(VAR)) return varDeclaration();
```

> Like other statements, a function is recognized by the leading keyword. When we encounter `fun`, we call `function`. That corresponds to the `function` grammar rule since we already matched and consumed the `fun` keyword. We'll build the method up a piece at a time, starting with this:

像其它语句一样，函数是通过前面的关键字来识别的。当我们遇到`fun`时，我们就调用`function`。这步操作对应于`function`语法规则，因为我们已经匹配并消费了`fun`关键字。我们会一步步构建这个方法，首先从下面的代码开始：

lox/Parser.java，在 expressionStatement()方法后添加：

```
  private Stmt.Function function(String kind) {
    Token name = consume(IDENTIFIER, "Expect " + kind + " name.");
  }
```

> Right now, it only consumes the identifier token for the function's name. You might be wondering about that funny little `kind` parameter. Just like we reuse the grammar rule, we'll reuse the

> function() method later to parse methods inside classes. When we do that, we'll pass in "method" for kind so that the error messages are specific to the kind of declaration being parsed.

现在，它只是消费了标识符标记作为函数名称。你可能会对这里的kind参数感到疑惑。就像我们复用语法规则一样，稍后我们也会复用function()方法来解析类中的方法。到时候，我们会在kind参数中传入 "method"，这样错误信息就会针对被解析的声明类型来展示。

> Next, we parse the parameter list and the pair of parentheses wrapped around it.

接下来，我们要解析参数列表和包裹着它们的一对小括号。

lox/Parser.java，在 function()方法中添加：

```java
    Token name = consume(IDENTIFIER, "Expect " + kind + " name.");
    // 新增部分开始
    consume(LEFT_PAREN, "Expect '(' after " + kind + " name.");
    List<Token> parameters = new ArrayList<>();
    if (!check(RIGHT_PAREN)) {
      do {
        if (parameters.size() >= 255) {
          error(peek(), "Can't have more than 255 parameters.");
        }

        parameters.add(
            consume(IDENTIFIER, "Expect parameter name."));
      } while (match(COMMA));
    }
    consume(RIGHT_PAREN, "Expect ')' after parameters.");
    // 新增部分结束
  }
```

> This is like the code for handling arguments in a call, except not split out into a helper method. The outer if statement handles the zero parameter case, and the inner while loop parses parameters as long as we find commas to separate them. The result is the list of tokens for each parameter's name.

这就像在函数调用中处理参数的代码一样，只是没有拆分到一个辅助方法中。外部的if语句用于处理零参数的情况，内部的while会循环解析参数，只要能找到分隔参数的逗号。其结果是包含每个参数名称的标记列表。

> Just like we do with arguments to function calls, we validate at parse time that you don't exceed the maximum number of parameters a function is allowed to have.

就像我们处理函数调用的参数一样，我们在解析时验证是否超过了一个函数所允许的最大参数数。

> Finally, we parse the body and wrap it all up in a function node.

最后，我们解析函数主体，并将其封装为一个函数节点。

lox/Parser.java，在 function()方法中添加：

```
    consume(RIGHT_PAREN, "Expect ')' after parameters.");
    // 新增部分开始
    consume(LEFT_BRACE, "Expect '{' before " + kind + " body.");
    List<Stmt> body = block();
    return new Stmt.Function(name, parameters, body);
    // 新增部分结束
  }
```

> Note that we consume the `{` at the beginning of the body here before calling `block()`. That's because `block()` assumes the brace token has already been matched. Consuming it here lets us report a more precise error message if the `{` isn't found since we know it's in the context of a function declaration.

请注意，在调用`block()`方法之前，我们已经消费了函数体开头的`{`。这是因为`block()`方法假定大括号标记已经匹配了。在这里消费该标记可以让我们在找不到`{`的情况下报告一个更精确的错误信息，因为我们知道当前是在一个函数声明的上下文中。

> ## 10.4 Function Objects

10.4 函数对象

> We've got some syntax parsed so usually we're ready to interpret, but first we need to think about how to represent a Lox function in Java. We need to keep track of the parameters so that we can bind them to argument values when the function is called. And, of course, we need to keep the code for the body of the function so that we can execute it.

我们已经解析了一些语法，通常我们要开始准备解释了，但是我们首先需要思考一下，在Java中如何表示一个Lox函数。我们需要跟踪形参，以便在函数被调用时可以将形参与实参值进行绑定。当然，我们也要保留函数体的代码，以便我们可以执行它。

> That's basically what the Stmt.Function class is. Could we just use that? Almost, but not quite. We also need a class that implements LoxCallable so that we can call it. We don't want the runtime phase of the interpreter to bleed into the front end's syntax classes so we don't want Stmt.Function itself to implement that. Instead, we wrap it in a new class.

这基本上就是Stmt.Function的内容。我们可以用这个吗？差不多，但还不够。我们还需要一个实现LoxCallable的类，以便我们可以调用它。我们不希望解释器的运行时阶段渗入到前端语法类中，所以我们不希望使用Stmt.Function本身来实现它。相反，我们将它包装在一个新类中。

lox/LoxFunction.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

class LoxFunction implements LoxCallable {
  private final Stmt.Function declaration;
  LoxFunction(Stmt.Function declaration) {
    this.declaration = declaration;
    }
  }
```

> We implement the `call()` of LoxCallable like so:

使用如下方式实现LoxCallable的`call()`方法：

lox/LoxFunction.java，在 LoxFunction()方法后添加：

```
  @Override
  public Object call(Interpreter interpreter,
                     List<Object> arguments) {
    Environment environment = new Environment(interpreter.globals);
    for (int i = 0; i < declaration.params.size(); i++) {
      environment.define(declaration.params.get(i).lexeme,
          arguments.get(i));
    }

    interpreter.executeBlock(declaration.body, environment);
    return null;
  }
```

> This handful of lines of code is one of the most fundamental, powerful pieces of our interpreter. As we saw in the chapter on statements and state, managing name environments is a core part of a language implementation. Functions are deeply tied to that.

这几行代码是我们的解释器中最基本、最强大的部分之一。正如我们在上一章中所看到的，管理名称环境是语言实现中的核心部分。函数与此紧密相关。

> Parameters are core to functions, especially the fact that a function *encapsulates* its parameters—no other code outside of the function can see them. This means each function gets its own environment where it stores those variables.

参数是函数的核心，尤其是考虑到函数*封装*了其参数——函数之外的代码看不到这些参数。这意味着每个函数都会维护自己的环境，其中存储着那些变量。

> Further, this environment must be created dynamically. Each function *call* gets its own environment. Otherwise, recursion would break. If there are multiple calls to the same function in play at the same time, each needs its *own* environment, even though they are all calls to the same function.

此外，这个环境必须是动态创建的。每次函数 *调用*都会获得自己的环境，否则，递归就会中断。如果在同一时刻对相同的函数有多次调用，那么每个调用都需要自身的环境，即便它们都是对相同函数的调用。

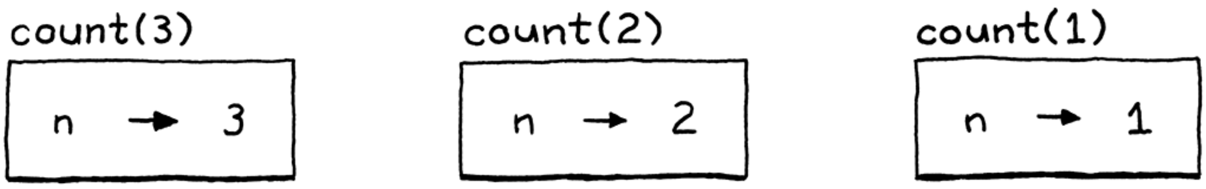
> For example, here’s a convoluted way to count to three:

举例来说，下面是一个计数到3的复杂方法：

```
fun count(n) {
  if (n > 1) count(n - 1);
  print n;
}

count(3);
```

> Imagine we pause the interpreter right at the point where it's about to print 1 in the innermost nested call. The outer calls to print 2 and 3 haven't printed their values yet, so there must be environments somewhere in memory that still store the fact that n is bound to 3 in one context, 2 in another, and 1 in the innermost, like:

假设一下，如果我们在最内层的嵌套调用中即将打印1的时候暂停了解释器。打印2和3的外部调用还没有打印出它们的值，所以在内存的某个地方一定有环境仍然存储着这样的数据：n在一个上下文中被绑定到3，在另一个上下文中被绑定到2，而在最内层调用中绑定为1，比如：



> That's why we create a new environment at each *call*, not at the function *declaration*. The `call()` method we saw earlier does that. At the beginning of the call, it creates a new environment. Then it walks the parameter and argument lists in lockstep. For each pair, it creates a new variable with the parameter's name and binds it to the argument's value.

这就是为什么我们在每次*调用*时创建一个新的环境，而不是在函数声明时创建。我们前面看到的`call()`方法就是这样做的。在调用开始的时候，它创建了一个新环境。然后它以同步的方式遍历形参和实参列表。对于每一对参数，它用形参的名字创建一个新的变量，并将其与实参的值绑定。

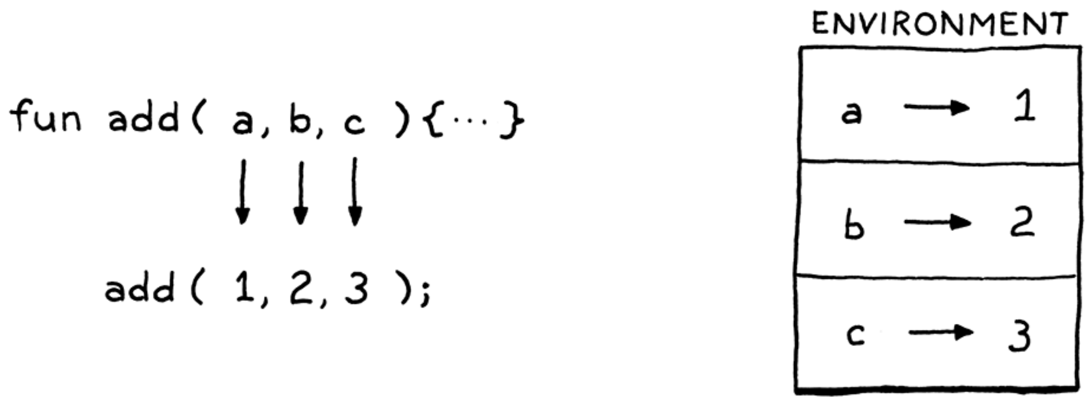
> So, for a program like this:

所以，对于类似下面这样的代码：

```
fun add(a, b, c) {
  print a + b + c;
}

add(1, 2, 3);
```

> At the point of the call to `add()`, the interpreter creates something like this:

在调用`add()`时，解释器会创建类似下面这样的内容：



> Then `call()` tells the interpreter to execute the body of the function in this new function-local environment. Up until now, the current environment was the environment where the function was being called. Now, we teleport from there inside the new parameter space we've created for the function.

然后`call()`会告诉解释器在这个新的函数局部环境中执行函数体。在此之前，当前环境是函数被调用的位置所处的环境。现在，我们转入了为函数创建的新的参数空间中。

> This is all that's required to pass data into the function. By using different environments when we execute the body, calls to the same function with the same code can produce different results.

这就是将数据传入函数所需的全部内容。通过在执行函数主体时使用不同的环境，用同样的代码调用相同的函数可以产生不同的结果。

> Once the body of the function has finished executing, `executeBlock()` discards that function-local environment and restores the previous one that was active back at the callsite. Finally, `call()` returns `null`, which returns `nil` to the caller. (We'll add return values later.)

一旦函数的主体执行完毕，`executeBlock()`就会丢弃该函数的本地环境，并恢复调用该函数前的活跃环境。最后，`call()`方法会返回`null`，它向调用者返回`nil`。（我们会在稍后添加返回值）

> Mechanically, the code is pretty simple. Walk a couple of lists. Bind some new variables. Call a method. But this is where the crystalline *code* of the function declaration becomes a living, breathing *invocation*. This is one of my favorite snippets in this entire book. Feel free to take a moment to meditate on it if you're so inclined.

从机制上讲，这段代码是非常简单的。遍历几个列表，绑定一些新变量，调用一个方法。但这就是将代码块变成有生命力的调用执行的地方。这是我在整本书中最喜欢的片段之一。如果你愿意的话，可以花点时间好好思考一下。

> Done? OK. Note when we bind the parameters, we assume the parameter and argument lists have the same length. This is safe because `visitCallExpr()` checks the arity before calling `call()`. It relies on the function reporting its arity to do that.

完成了吗？好的。注意当我们绑定参数时，我们假设参数和参数列表具有相同的长度。这是安全的，因为`visitCallExpr()`在调用`call()`之前会检查元数。它依靠报告其元数的函数来做到这一点。

lox/LoxFunction.java，在 LoxFunction()方法后添加：

```
  @Override
  public int arity() {
    return declaration.params.size();
  }
```

> That's most of our object representation. While we're in here, we may as well implement `toString()`.

这基本就是我们的函数对象表示了。既然已经到了这一步，我们也可以实现toString()。

lox/LoxFunction.java，在 LoxFunction()方法后添加：

```
  @Override
  public String toString() {
    return "<fn " + declaration.name.lexeme + ">";
  }
```

> This gives nicer output if a user decides to print a function value.

如果用户要打印函数的值，该方法能提供一个更漂亮的输出值。

```
fun add(a, b) {
  print a + b;
}

print add; // "<fn add>".
```

> ## 10.4.1 Interpreting function declarations

10.4.1 解释函数声明

> We'll come back and refine LoxFunction soon, but that's enough to get started. Now we can visit a function declaration.

我们很快就会回头来完善LoxFunction，但是现在已足够开始进行解释了。现在，我们可以访问函数声明节点了。

lox/Interpreter.java，在 visitExpressionStmt()方法后添加：

```
  @Override
  public Void visitFunctionStmt(Stmt.Function stmt) {
    LoxFunction function = new LoxFunction(stmt);
    environment.define(stmt.name.lexeme, function);
    return null;
  }
```

> This is similar to how we interpret other literal expressions. We take a function *syntax node*—a compile-time representation of the function—and convert it to its runtime representation. Here, that's a LoxFunction that wraps the syntax node.

这类似于我们介绍其它文本表达式的方式。我们会接收一个函数 *语法*节点——函数的编译时表示形式——然后将其转换为运行时表示形式。在这里就是一个封装了语法节点的LoxFunction实例。

> Function declarations are different from other literal nodes in that the declaration *also* binds the resulting object to a new variable. So, after creating the LoxFunction, we create a new binding in the current environment and store a reference to it there.

函数声明与其它文本节点的不同之处在于，声明还会将结果对象绑定到一个新的变量。因此，在创建LoxFunction之后，我们在当前环境中创建一个新的绑定，并在其中保存对该函数的引用。

> With that, we can define and call our own functions all within Lox. Give it a try:

这样，我们就可以在Lox中定义和调用我们自己的函数。试一下：

```
fun sayHi(first, last) {
  print "Hi, " + first + " " + last + "!";
}

sayHi("Dear", "Reader");
```

> I don't know about you, but that looks like an honest-to-God programming language to me.

我不知道你怎么想的，但对我来说，这看起来像是一种虔诚的编程语言。

10.5 Return Statements

10.5 Return语句

> We can get data into functions by passing parameters, but we've got no way to get results back *out*. If Lox were an expression-oriented language like Ruby or Scheme, the body would be an expression whose value is implicitly the function's result. But in Lox, the body of a function is a list of statements which don't produce values, so we need dedicated syntax for emitting a result. In other words, `return` statements. I'm sure you can guess the grammar already.

我们可以通过传递参数将数据输入函数中，但是我们没有办法将结果*传出来*。如果Lox是像Ruby或Scheme那样的面向表达式的语言，那么函数体就是一个表达式，其值就隐式地作为函数的结果。但是在Lox中，函数体是一个不产生值的语句列表，所有我们需要专门的语句来发出结果。换句话说，就是`return`语句。我相信你已经能猜出语法了。

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | returnStmt
               | whileStmt
               | block ;

returnStmt     → "return" expression? ";" ;
```

> We've got one more—the final, in fact—production under the venerable `statement` rule. A `return` statement is the `return` keyword followed by an optional expression and terminated with a semicolon.

我们又得到一个`statement`规则下的新产生式（实际上也是最后一个）。一个`return`语句就是一个`return`关键字，后跟一个可选的表达式，并以一个分号结尾。

> The return value is optional to support exiting early from a function that doesn't return a useful value. In statically typed languages, "void" functions don't return a value and non-void ones do. Since Lox is dynamically typed, there are no true void functions. The compiler has no way of preventing you from taking the result value of a call to a function that doesn't contain a `return` statement.

返回值是可选的，用以支持从一个不返回有效值的函数中提前退出。在静态类型语言中，void函数不返回值，而非void函数返回值。由于Lox是动态类型的，所以没有真正的void函数。在调用一个不包含`return`语句的函数时，编译器没有办法阻止你获取其结果值。

```
fun procedure() {
  print "don't return anything";
}

var result = procedure();
print result; // ?
```

> This means every Lox function must return *something*, even if it contains no `return` statements at all. We use `nil` for this, which is why LoxFunction's implementation of `call()` returns `null` at the end. In that same vein, if you omit the value in a `return` statement, we simply treat it as equivalent to:

这意味着每个Lox函数都要返回一些内容，即使其中根本不包含`return`语句。我们使用`nil`，这就是为什么LoxFunction的`call()`实现在最后返回`null`。同样，如果你省略了`return` 语句中的值，我们将其视为等价于：

```
return nil;
```

> Over in our AST generator, we add a new node.

在AST生成器中，添加一个新节点。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Print      : Expr expression",
      // 新增部分开始
      "Return     : Token keyword, Expr value",
      // 新增部分结束
      "Var        : Token name, Expr initializer",
```

> It keeps the `return` keyword token so we can use its location for error reporting, and the value being returned, if any. We parse it like other statements, first by recognizing the initial keyword.

其中保留了`return`关键字标记（这样我们可以使用该标记的位置来报告错误），以及返回的值（如果有的话）。我们像解析其它语句一样来解析它，首先识别起始的关键字。

lox/Parser.java，在 statement()方法中添加：

```
    if (match(PRINT)) return printStatement();
    // 新增部分开始
    if (match(RETURN)) return returnStatement();
    // 新增部分结束
    if (match(WHILE)) return whileStatement();
```

> That branches out to:

分支会跳转到：

lox/Parser.java，在 printStatement()方法后添加：

```
  private Stmt returnStatement() {
    Token keyword = previous();
    Expr value = null;
    if (!check(SEMICOLON)) {
      value = expression();
    }

    consume(SEMICOLON, "Expect ';' after return value.");
    return new Stmt.Return(keyword, value);
  }
```

> After snagging the previously consumed `return` keyword, we look for a value expression. Since many different tokens can potentially start an expression, it's hard to tell if a return value is *present*. Instead, we check if it's *absent*. Since a semicolon can't begin an expression, if the next token is that, we know there must not be a value.

在捕获先前消耗的`return`关键字之后，我们会寻找一个值表达式。因为很多不同的标记都可以引出一个表达式，所以很难判断是否存在返回值。相反，我们检查它是否不存在。因为分号不能作为表达式的开始，如果下一个标记是分号，我们就知道一定没有返回值。

> ### 10.5.1 Returning from calls

10.5.1 从函数调用中返回

> Interpreting a `return` statement is tricky. You can return from anywhere within the body of a function, even deeply nested inside other statements. When the return is executed, the interpreter needs to

> jump all the way out of whatever context it’s currently in and cause the function call to complete, like some kind of jacked up control flow construct.

解释`return`语句是很棘手的。你可以从函数体中的任何位置返回，甚至是深深嵌套在其它语句中的位置。当返回语句被执行时，解释器需要完全跳出当前所在的上下文，完成函数调用，就像某种顶层的控制流结构。

> For example, say we’re running this program and we’re about to execute the `return` statement:

举例来说，假设我们正在运行下面的代码，并且我们即将执行`return`语句：

```
fun count(n) {
  while (n < 100) {
    if (n == 3) return n; // <--
    print n;
    n = n + 1;
  }
}

count(1);
```

> The Java call stack currently looks roughly like this:

Java调用栈目前看起来大致如下所示：

```
Interpreter.visitReturnStmt()
Interpreter.visitIfStmt()
Interpreter.executeBlock()
Interpreter.visitBlockStmt()
Interpreter.visitWhileStmt()
Interpreter.executeBlock()
LoxFunction.call()
Interpreter.visitCallExpr()
```

> We need to get from the top of the stack all the way back to `call()`. I don’t know about you, but to me that sounds like exceptions. When we execute a `return` statement, we’ll use an exception to unwind the interpreter past the visit methods of all of the containing statements back to the code that began executing the body.

我们需要从栈顶一直回退到`call()`。我不知道你怎么想，但是对我来说，这听起来很像是异常。当我们执行`return`语句时，我们会使用一个异常来解开解释器，经过所有函数内含语句的visit方法，一直回退到开始执行函数体的代码。

> The visit method for our new AST node looks like this:

新的AST节点的visit方法如下所示：

lox/Interpreter.java，在 visitPrintStmt()方法后添加：

```java
  @Override
  public Void visitReturnStmt(Stmt.Return stmt) {
    Object value = null;
    if (stmt.value != null) value = evaluate(stmt.value);

    throw new Return(value);
  }
```

> If we have a return value, we evaluate it, otherwise, we use `nil`. Then we take that value and wrap it in a custom exception class and throw it.

如果我们有返回值，就对其求值，否则就使用`nil`。然后我们取这个值并将其封装在一个自定义的异常类中，并抛出该异常。

lox/Return.java，创建新文件：

```java
package com.craftinginterpreters.lox;

class Return extends RuntimeException {
  final Object value;

  Return(Object value) {
    super(null, null, false, false);
    this.value = value;
  }
}
```

> This class wraps the return value with the accoutrements Java requires for a runtime exception class. The weird super constructor call with those `null` and `false` arguments disables some JVM machinery that we don't need. Since we're using our exception class for control flow and not actual error handling, we don't need overhead like stack traces.

这个类使用Java运行时异常类来封装返回值。其中那个奇怪的带有`null`和`false`的父类构造器方法，禁用了一些我们不需要的JVM机制。因为我们只是使用该异常类来控制流，而不是真正的错误处理，所以我们不需要像堆栈跟踪这样的开销。

> We want this to unwind all the way to where the function call began, the `call()` method in LoxFunction.

我们希望可以一直跳出到函数调用开始的地方，也就是LoxFunction中的`call()`方法。

lox/LoxFunction.java，在 call()方法中替换一行：

```java
          arguments.get(i));
    }
    // 替换部分开始
    try {
      interpreter.executeBlock(declaration.body, environment);
    } catch (Return returnValue) {
      return returnValue.value;
    }
    // 替换部分结束
    return null;
```

> We wrap the call to `executeBlock()` in a try-catch block. When it catches a return exception, it pulls out the value and makes that the return value from `call()`. If it never catches one of these exceptions, it means the function reached the end of its body without hitting a `return` statement. In that case, it implicitly returns `nil`.

我们将对`executeBlock()`的调用封装在一个try-catch块中。当捕获一个返回异常时，它会取出其中的值并将其作为`call()`方法的返回值。如果没有捕获任何异常，意味着函数到达了函数体的末尾，而且没有遇到`return`语句。在这种情况下，隐式地返回`nil`。

> Let's try it out. We finally have enough power to support this classic example—a recursive function to calculate Fibonacci numbers:

我们来试一下。我们终于有能力支持这个经典的例子——递归函数计算Fibonacci数^7：

```
fun fib(n) {
  if (n <= 1) return n;
  return fib(n - 2) + fib(n - 1);
}

for (var i = 0; i < 20; i = i + 1) {
  print fib(i);
}
```

> This tiny program exercises almost every language feature we have spent the past several chapters implementing—expressions, arithmetic, branching, looping, variables, functions, function calls, parameter binding, and returns.

这个小程序练习了我们在过去几章中实现的几乎所有语言特性，包括表达式、算术运算、分支、循环、变量、函数、函数调用、参数绑定和返回。

> ## 10.6 Local Functions and Closures

10.6 局部函数和闭包

> Our functions are pretty full featured, but there is one hole to patch. In fact, it's a big enough gap that we'll spend most of the next chapter sealing it up, but we can get started here.

我们的函数功能已经相当全面了，但是还有一个漏洞需要修补。实际上，这是一个很大的问题，我们将会在下一章中花费大部分时间来修补它，但是我们可以从这里开始。

> LoxFunction's implementation of `call()` creates a new environment where it binds the function's parameters. When I showed you that code, I glossed over one important point: What is the *parent* of that environment?

LoxFunction中的`call()`实现创建了一个新的环境，并在其中绑定了函数的参数。当我向你展示这段代码时，我忽略了一个重要的问题：这个环境的父类是什么？

> Right now, it is always `globals`, the top-level global environment. That way, if an identifier isn't defined inside the function body itself, the interpreter can look outside the function in the global scope to find it. In the Fibonacci example, that's how the interpreter is able to look up the recursive call to `fib` inside the function's own body—`fib` is a global variable.

目前，它始终是`globals`，即顶级的全局环境。这样，如果一个标识符不是在函数体内部定义的，解释器可以在函数外部的全局作用域中查找它。在Fibonacci的例子中，这就是解释器如何能够在函数体中实现对`fib`的递归调用——`fib`是一个全局变量。

> But recall that in Lox, function declarations are allowed *anywhere* a name can be bound. That includes the top level of a Lox script, but also the inside of blocks or other functions. Lox supports **local functions** that are defined inside another function, or nested inside a block.

但请记住，在Lox中，允许在可以绑定名字的*任何地方*进行函数声明。其中包括Lox脚本的顶层，但也包括块或其他函数的内部。Lox支持在另一个函数内定义或在一个块内嵌套的**局部函数**。

> Consider this classic example:

考虑下面这个经典的例子：

```
fun makeCounter() {
  var i = 0;
  fun count() {
    i = i + 1;
    print i;
  }

  return count;
}

var counter = makeCounter();
counter(); // "1".
counter(); // "2".
```

> Here, `count()` uses `i`, which is declared outside of itself in the containing function `makeCounter()`. `makeCounter()` returns a reference to the `count()` function and then its own body finishes executing completely.

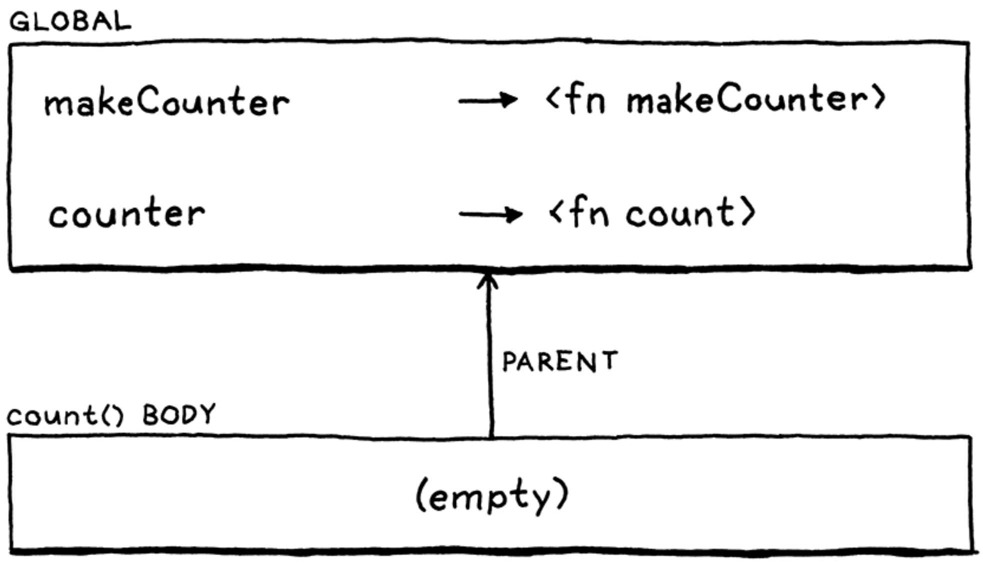
这个例子中，`count()`使用了`i`，它是在该函数外部的 `makeCounter()`声明的。`makeCounter()` 返回对`count()`函数的引用，然后它的函数体就执行完成了。

> Meanwhile, the top-level code invokes the returned `count()` function. That executes the body of `count()`, which assigns to and reads `i`, even though the function where `i` was defined has already exited.

同时，顶层代码调用了返回的`count()`函数。这就执行了`count()`函数的主体，它会对`i`赋值并读取`i`，尽管定义`i`的函数已经退出。

> If you've never encountered a language with nested functions before, this might seem crazy, but users do expect it to work. Alas, if you run it now, you get an undefined variable error in the call to `counter()` when the body of `count()` tries to look up `i`. That's because the environment chain in effect looks like this:

如果你以前从未遇到过带有嵌套函数的语言，那么这可能看起来很疯狂，但用户确实希望它能工作。唉，如果你现在运行它，当`count()`的函数体试图查找`i`时，会在对`counter()`的调用中得到一个未定义的变量错误，这是因为当前的环境链看起来像是这样的：

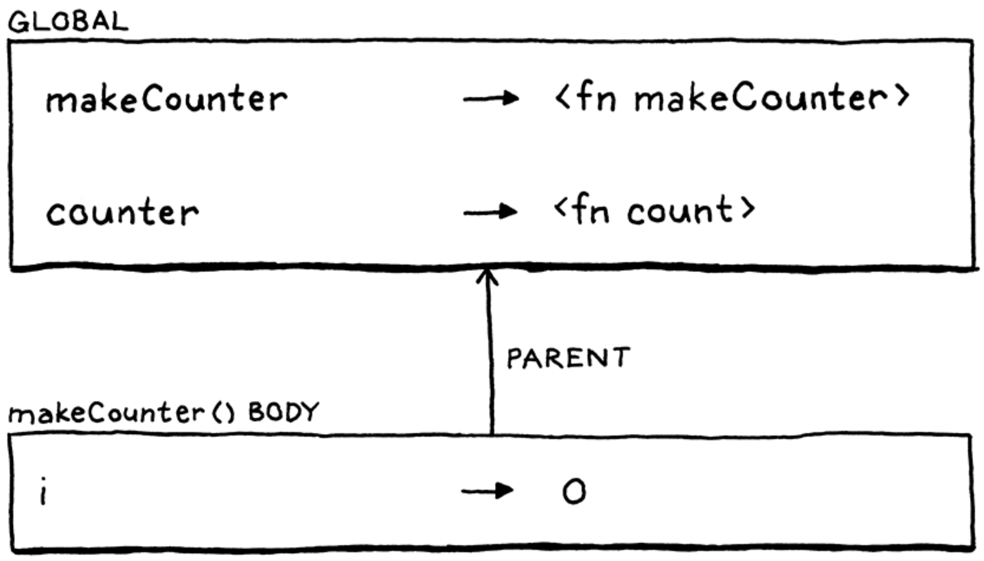


> When we call `count()` (through the reference to it stored in `counter`), we create a new empty environment for the function body. The parent of that is the global environment. We lost the environment for `makeCounter()` where `i` is bound.

当我们调用`count()`时（通过`counter`中保存的引用），我们会为函数体创建一个新的空环境，它的父环境就是全局环境。我们丢失了`i`所在的`makeCounter()`环境。

> Let's go back in time a bit. Here's what the environment chain looked like right when we declared `count()` inside the body of `makeCounter()`:

我们把时间往回拨一点。我们在`makeCounter()`的函数体中声明`count()`时，环境链的样子是下面这样：



> So at the point where the function is declared, we can see `i`. But when we return from `makeCounter()` and exit its body, the interpreter discards that environment. Since the interpreter doesn't keep the environment surrounding `count()` around, it's up to the function object itself to hang on to it.

所以，在函数声明的地方，我们可以看到`i`。但是当我们从`makeCounter()`返回并退出其主体时，解释器会丢弃这个环境。因为解释器不会保留`count()`外围的环境，所以要靠函数对象本身来保存它。

> This data structure is called a **closure** because it "closes over" and holds on to the surrounding variables where the function is declared. Closures have been around since the early Lisp days, and language hackers have come up with all manner of ways to implement them. For jlox, we'll do the simplest thing that works. In LoxFunction, we add a field to store an environment.

这种数据结构被称为**闭包**，因为它 "封闭 "并保留着函数声明的外围变量。闭包早在Lisp时代就已经存在了，语言黑客们想出了各种方法来实现闭包。在jlox中，我们将采用最简单的方式。在LoxFunction中，我们添加一个字段来存储环境。

lox/LoxFunction.java，在 LoxFunction类中添加：

```
  private final Stmt.Function declaration;
  // 新增部分开始
  private final Environment closure;
  // 新增部分结束
  LoxFunction(Stmt.Function declaration) {
```

> We initialize that in the constructor.

我们在构造函数中对其初始化。

lox/LoxFunction.java，在 LoxFunction()构造方法中替换一行：

```
  //替换部分开始
  LoxFunction(Stmt.Function declaration, Environment closure) {
    this.closure = closure;
    // 替换部分结束
    this.declaration = declaration;
```

> When we create a LoxFunction, we capture the current environment.

当我们创建LoxFunction时，我们会捕获当前环境。

lox/Interpreter.java，在 visitFunctionStmt()方法中替换一行：

```
    public Void visitFunctionStmt(Stmt.Function stmt) {
      // 替换部分开始
      LoxFunction function = new LoxFunction(stmt, environment);
      // 替换部分结束
      environment.define(stmt.name.lexeme, function);
```

> This is the environment that is active when the function is *declared* not when it's *called*, which is what we want. It represents the lexical scope surrounding the function declaration. Finally, when we call the function, we use that environment as the call's parent instead of going straight to `globals`.

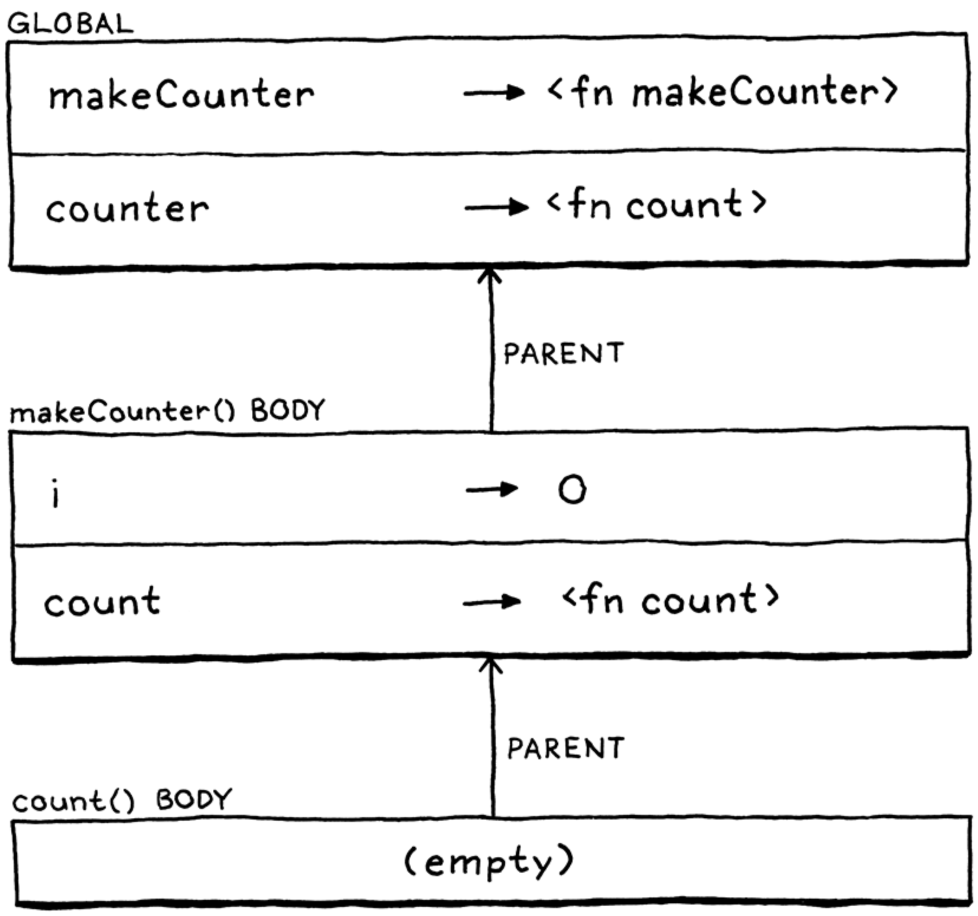
这是函数声明时生效的环境，而不是函数被调用时的环境，这正是我们想要的。它代表了函数声明时的词法作用域。最后，当我们调用函数时，我们使用该环境作为调用的父环境，而不是直接使用`globals`。

lox/LoxFunction.java，在 call()方法中替换一行：

```
                         List<Object> arguments) {
    // 替换部分开始
    Environment environment = new Environment(closure);
    // 替换部分结束
    for (int i = 0; i < declaration.params.size(); i++) {
```

> This creates an environment chain that goes from the function's body out through the environments where the function is declared, all the way out to the global scope. The runtime environment chain matches the textual nesting of the source code like we want. The end result when we call that function looks like this:

这样就创建了一个环境链，从函数体开始，经过函数被声明的环境，然后到全局作用域。运行时环境链与源代码的文本嵌套相匹配，跟我们想要的一致。当我们调用该函数时，最终的结果是这样的：



> Now, as you can see, the interpreter can still find `i` when it needs to because it's in the middle of the environment chain. Try running that `makeCounter()` example now. It works!

如你所见，现在解释器可以在需要的时候找到`i`，因为它在环境链中。现在尝试运行makeCounter()的例子，起作用了！

> Functions let us abstract over, reuse, and compose code. Lox is much more powerful than the rudimentary arithmetic calculator it used to be. Alas, in our rush to cram closures in, we have let a tiny bit of dynamic scoping leak into the interpreter. In the next chapter, we will explore deeper into lexical scope and close that hole.

函数让我们对代码进行抽象、重用和编排。Lox比之前的初级算术计算器要强大得多。唉，在我们匆匆忙忙支持闭包时，已经让一小部分动态作用域泄露到解释器中了。在下一章中，我们将深入探索词法作用域，堵住这个漏洞。

^1: 该规则中使用`*`符号匹配类似`fn(1)(2)(3)`的系列函数调用。这样的代码不是常见的C语言风格，但是在ML衍生的语言族中很常见。在ML中，定义接受多个参数的函数的常规方式是将其定义为一系列嵌套函数。每个函数接受一个参数并返回一个新函数。该函数使用下一个参数，返回另一个函数，以此类推。最终，一旦所有参数都被使用，最后一个函数就完成了操作。这种风格被称为柯里化，是以Haskell Curry（他的名字出现在另一个广为人知的函数式语言中）的名字命名的，它被直接整合到语言的语法中，所以它不像这里看起来那么奇怪。 ^2: 这段代码可以简化为`while (match(LEFT_PAREN))`形式，而不是使用这种愚蠢的`while (true)` 和`break`形式。但是不用担心，稍后使用解析器处理对象属性的时候，这种写法就有意义了。 ^3: 如果该方法是一个实例方法，则限制为254个参数。因为`this`（方法的接收者）就像一个被隐式传递给方法的参数一样，所以也会占用一个参数位置。 ^4: 这是另一个微妙的语义选择。由于参数表达式可能有副作用，因此它们的执行顺序可能是用户可见的。即便如此，有些语言如Scheme和C并没有指定顺序。这样编译器可以自由地重新排序以提高效率，但这意味着如果参数没有按照用户期望的顺序计算，用户可能会感到不愉快。 ^5: 奇怪的是，这些函数的两个名称native和foreign是反义词。也许这取决于选择这个词的人的角度。如果您认为自己生活在运行时实现中(在我们的例子中是Java)，那么用它编写的函数就是本机的。但是，如果您站在语言用户的角度，那么运行时就是用其他"外来"语言实现的。或者本机指的是底层硬件的机器代码语言。在Java中，本机方法是用C或c++实现并编译为本机机器码的方法。 ^6: 几乎每种语言都提供的一个经典的本地函数是将文本打印到标准输出。在Lox中，我将`print`作为了内置语句，以便可以在前面的章节中看到代码结果。一旦我们有了函数，我们就可以删除之前的`print`语法并用一个本机函数替换它，从而简化语言。但这意味着书中前面的例子不能在后面章节的解释器上运行，反之亦然。所以，在这本书中，我不去修改它。但是，如果您正在为自己的语言构建一个解释器，您可能需要考虑一下。 ^7: 你可能会注意到这是很慢的。显然，递归并不是计算斐波那契数的最有效方法，但作为一个微基准测试，它很好地测试了我们的解释器实现函数调用的速度。

> ## CHALLENGES

习题

> 1、Our interpreter carefully checks that the number of arguments passed to a function matches the number of parameters it expects. Since this check is done at runtime on every call, it has a performance cost. Smalltalk implementations don't have that problem. Why not?

1、解释器会仔细检查传给函数的实参数量是否与期望的形参数量匹配。由于该检查是在运行时，针对每一次调用执行的，所以会有性能成本。Smalltalk的实现则没有这个问题。为什么呢？

> 2、Lox's function declaration syntax performs two independent operations. It creates a function and also binds it to a name. This improves usability for the common case where you do want to associate a name with the function. But in functional-styled code, you often want to create a function to immediately pass it to some other function or return it. In that case, it doesn't need a name.

> Languages that encourage a functional style usually support **anonymous functions** or **lambdas**—an expression syntax that creates a function without binding it to a name. Add anonymous function syntax to Lox so that this works:
>
> ```
> fun thrice(fn) {
>   for (var i = 1; i <= 3; i = i + 1) {
>     fn(i);
>   }
> }
>
> thrice(fun (a) {
>   print a;
> });
> // "1".
> // "2".
> // "3".
> ```
>
> How do you handle the tricky case of an anonymous function expression occurring in an expression statement:
>
> ```
> fun () {};
> ```

2、Lox的函数声明语法执行了两个独立的操作。它创建了一个函数，并将其与一个名称绑定。这提高了常见情况下的可用性，即你确实想把一个名字和函数联系起来。但在函数式的代码中，你经常想创建一个函数，以便立即将它传递给其他函数或返回它。在这种情况下，它不需要一个名字。

鼓励函数式风格的语言通常支持**匿名函数**或**lambdas**——一个创建函数而不用将其与名称绑定的表达式语法。在Lox中加入匿名函数的语法，已支持下面的代码：

```
fun thrice(fn) {
  for (var i = 1; i <= 3; i = i + 1) {
    fn(i);
  }
}

thrice(fun (a) {
  print a;
});
// "1".
// "2".
// "3".
```

如何处理在表达式语句中出现匿名函数表达式的棘手情况：

```
fun () {};
```

> 3、Is this program valid?
>
> ```
> fun scope(a) {
>   var a = "local";
> }
> ```
>
> In other words, are a function's parameters in the *same* scope as its local variables, or in an outer scope? What does Lox do? What about other languages you are familiar with? What do you think a language *should* do?

3、下面的代码可用吗？

```
fun scope(a) {
  var a = "local";
}
```

换句话说，一个函数的参数是跟它的局部变量在同一个作用域内，还是在一个外部作用域内？Lox 是怎么做的？你所熟悉的其他语言呢？你认为一种语言应该怎么做？

11.解析和绑定 Resolving and Binding

> Once in a while you find yourself in an odd situation. You get into it by degrees and in the most natural way but, when you are right in the midst of it, you are suddenly astonished and ask yourself how in the world it all came about.
>
> —— Thor Heyerdahl, *Kon-Tiki*

你也许偶尔会发现自己处于一种奇怪的情况。你曾以最自然的方式逐渐进入其中，但当你身处其中时，你会突然感到惊讶，并问自己这一切到底是怎么发生的。

> Oh, no! Our language implementation is taking on water! Way back when we added variables and blocks, we had scoping nice and tight. But when we later added closures, a hole opened in our formerly waterproof interpreter. Most real programs are unlikely to slip through this hole, but as language implementers, we take a sacred vow to care about correctness even in the deepest, dampest corners of the semantics.

哦，不! 我们的语言实现正在进水! 在我们刚添加变量和代码块时，我们把作用域控制的很好很严密。但是当我们后来添加闭包之后，我们以前防水的解释器上就出现了一个洞。大多数真正的程序都不可能从这个洞里溜走，但是作为语言实现者，我们要立下神圣的誓言，即使在语义的最深处、最潮湿的角落里也要关心正确性。【译者注：这一段好中二，其实原文中有很多地方都有类似的中二之魂燃烧瞬间】

> We will spend this entire chapter exploring that leak, and then carefully patching it up. In the process, we will gain a more rigorous understanding of lexical scoping as used by Lox and other languages in the C tradition. We'll also get a chance to learn about *semantic analysis*—a powerful technique for extracting meaning from the user's source code without having to run it.

我们将用整整一章的时间来探索这个漏洞，然后小心翼翼地把它补上。在这个过程中，我们将对Lox和其他C语言传统中使用的词法范围有一个更严格的理解。我们还将有机会学习语义分析——这是一种强大的技术，用于从用户的源代码中提取语义而无需运行它。

11.1 Static Scope

11.1 静态作用域

> A quick refresher: Lox, like most modern languages, uses *lexical* scoping. This means that you can figure out which declaration a variable name refers to just by reading the text of the program. For example:

快速复习一下：Lox和大多数现代语言一样，使用词法作用域。这意味着你可以通过阅读代码文本找到变量名字指向的是哪个声明。例如：

```
var a = "outer";
{
  var a = "inner";
  print a;
}
```

> Here, we know that the a being printed is the variable declared on the previous line, and not the global one. Running the program doesn't—*can't*—affect this. The scope rules are part of the *static* semantics of the language, which is why they're also called *static scope*.

这里，我们知道打印的a是上一行声明的变量，而不是全局变量。运行代码并不会（也不能）影响这一点。作用域规则是语言的静态语义的一部分，这也就是为什么它们被称为静态作用域。

> I haven't spelled out those scope rules, but now is the time for precision:

我还没有详细说明这些作用域规则，但是现在是时候详细说明一下了^1：

> **A variable usage refers to the preceding declaration with the same name in the innermost scope that encloses the expression where the variable is used.**

变量指向的是使用变量的表达式外围环境中，前面具有相同名称的最内层作用域中的变量声明。

> There's a lot to unpack in that:

其中有很多东西需要解读：

- > I say "variable usage" instead of "variable expression" to cover both variable expressions and assignments. Likewise with "expression where the variable is used".

  我说的是"变量使用"而不是"变量表达式"，是为了涵盖变量表达式和赋值两种情况。类似于"使用变量的表达式"。

- > "Preceding" means appearing before *in the program text*.

  "前面"意味着出现在*程序文本*之前。

```
var a = "outer";
{
  print a;
  var a = "inner";
}
```

> Here, the a being printed is the outer one since it appears before the print statement that uses it. In most cases, in straight line code, the declaration preceding in *text* will also precede the usage in *time*. But that's not always true. As we'll see, functions may defer a chunk of code such that its *dynamic temporal* execution no longer mirrors the *static textual* ordering.

这里，打印的a是外层的，因为它在使用该变量的print语句之前。在大多数情况下，在单行代码中，文本中靠前的变量声明在时间上也先于变量使用。但并不总是如此。正如我们将看到的，函数可以推迟代码块，以使其动态执行的时间不受静态文本顺序的约束^2。

- > "Innermost" is there because of our good friend shadowing. There may be more than one variable with the given name in enclosing scopes, as in:

  "最内层"之所以存在，是因为我们的好朋友——变量遮蔽的缘故。在外围作用域中可能存在多个具有给定名称的变量。如：

  ```
  var a = "outer";
  {
    var a = "inner";
    print a;
  }
  ```

  > Our rule disambiguates this case by saying the innermost scope wins.

  我们通过优先使用最内层作用域的方式来消除这种歧义。

> Since this rule makes no mention of any runtime behavior, it implies that a variable expression always refers to the same declaration through the entire execution of the program. Our interpreter so far *mostly* implements the rule correctly. But when we added closures, an error snuck in.

由于这条规则没有提及任何运行时行为，它意味着一个变量表达式在程序的整个执行过程中总是指向同一声明。到目前为止，我们的解释器基本正确实现了这一规则。但是当我们添加了闭包后，一个错误悄悄出现了。

```
var a = "global";
{
  fun showA() {
    print a;
  }

  showA();
  var a = "block";
  showA();
}
```

> Before you type this in and run it, decide what you think it *should* print.

在你执行这段代码之前，先思考一下它*应该*输出什么^3。

> OK . . . got it? If you're familiar with closures in other languages, you'll expect it to print "global" twice. The first call to `showA()` should definitely print "global" since we haven't even reached the declaration of the inner `a` yet. And by our rule that a variable expression always resolves to the same variable, that implies the second call to `showA()` should print the same thing.

好的……清楚了吗？如果你熟悉其它语言中的闭包，你可能期望会输出两次"global"。对`showA()` 的第一次调用肯定会打印 "global"，因为我们甚至还没有执行到内部变量 `a` 的声明。而根据我们的规则，一个变量表达式总是解析为同一个变量，这意味着对 `showA()` 的第二次调用也应该打印出同样的内容。

> Alas, it prints:

唉，它输出的是：

```
global
block
```

> Let me stress that this program never reassigns any variable and contains only a single `print` statement. Yet, somehow, that `print` statement for a never-assigned variable prints two different values at different points in time. We definitely broke something somewhere.

我要强调一下，这个代码中从未重新分配任何变量，并且只包含一个`print`语句。然而，不知何故，对于这个从未分配过的变量，`print`语句在不同的时间点上打印了两个不同的值。我们肯定在什么地方出了问题。

> ### 11.1.1 Scopes and mutable environments

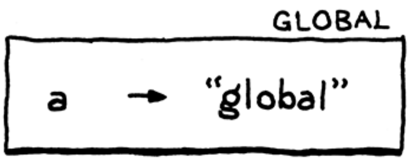
11.1.1 作用域和可变环境

> In our interpreter, environments are the dynamic manifestation of static scopes. The two mostly stay in sync with each other—we create a new environment when we enter a new scope, and discard it when we leave the scope. There is one other operation we perform on environments: binding a variable in one. This is where our bug lies.

在我们的解释器中，环境是静态作用域的动态表现。这两者大多情况下保持同步——当我们进入一个新的作用域时，我们会创建一个新的环境，当我们离开这个作用域时，我们会丢弃它。在环境中还有一个可执行的操作：在环境中绑定一个变量。这就是我们的问题所在。

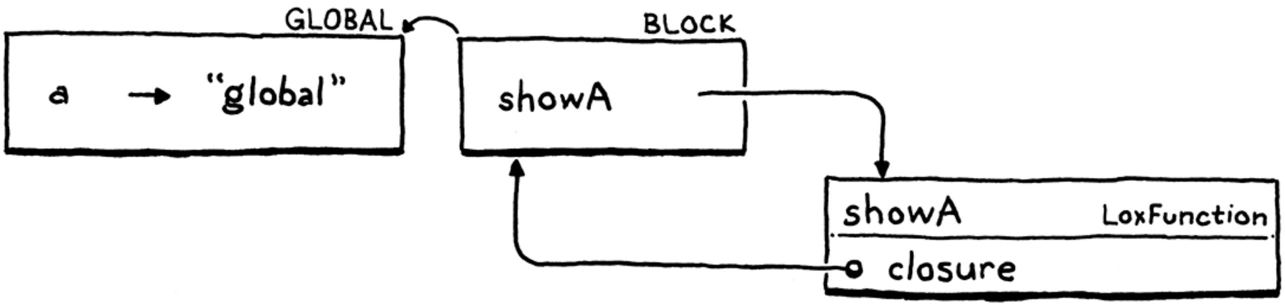
> Let's walk through that problematic example and see what the environments look like at each step. First, we declare `a` in the global scope.

让我们通过这个有问题的例子，看看每一步的环境是什么样的。首先，我们在全局作用域内声明`a`。



> That gives us a single environment with a single variable in it. Then we enter the block and execute the declaration of `showA()`.

这为我们提供了一个环境，其中只有一个变量。然后我们进入代码块，并执行`showA()`的声明。

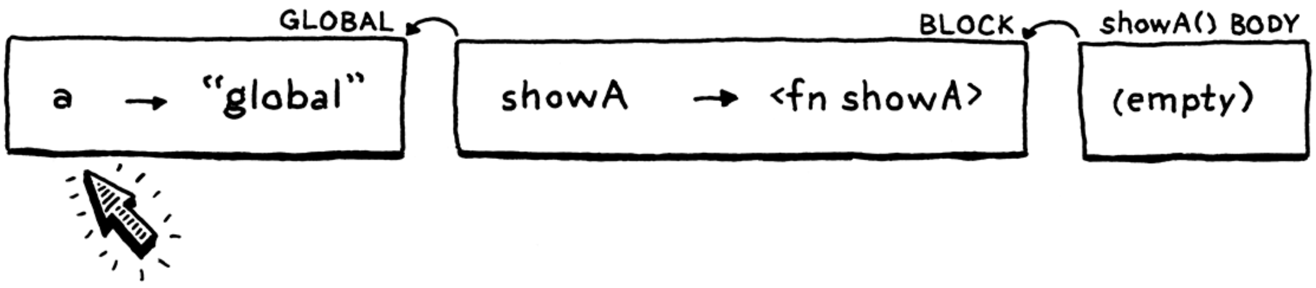


> We get a new environment for the block. In that, we declare one name, `showA`, which is bound to the LoxFunction object we create to represent the function. That object has a `closure` field that captures the environment where the function was declared, so it has a reference back to the environment for the block.

我们得到一个对应该代码块的新环境。在这个环境中，我们声明了一个名称`showA`，它绑定到为表示函数而创建的LoxFunction对象。该对象中有一个`closure`字段，用于捕获函数声明时的环境，因此它有一个指向该代码块环境的引用。

> Now we call `showA()`.

现在我们调用`showA()`。



> The interpreter dynamically creates a new environment for the function body of `showA()`. It's empty since that function doesn't declare any variables. The parent of that environment is the function's closure—the outer block environment.

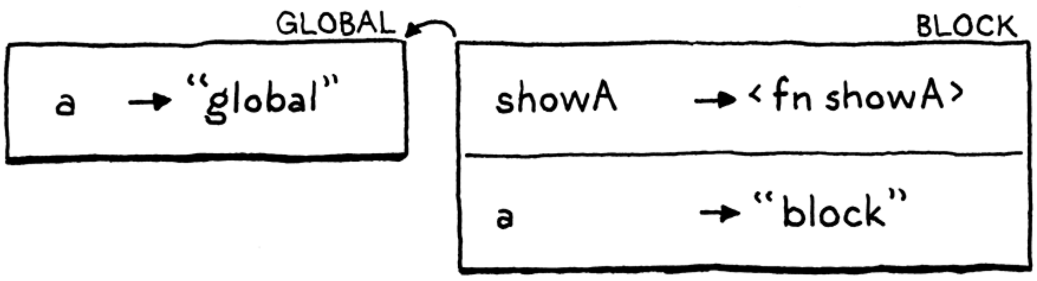
解释器为showA()的函数体动态地创建了一个新环境。它是空的，因为该函数没有声明任何变量。该环境的父环境是该函数的闭包——外部的代码块环境。

> Inside the body of `showA()`, we print the value of `a`. The interpreter looks up this value by walking the chain of environments. It gets all the way to the global environment before finding it there and printing `"global"`. Great.

在showA()函数体中，输出a的值。解释器通过遍历环境链来查找这个值。它会一直到达全局环境，在其中找到变量a并打印“global”。太好了。

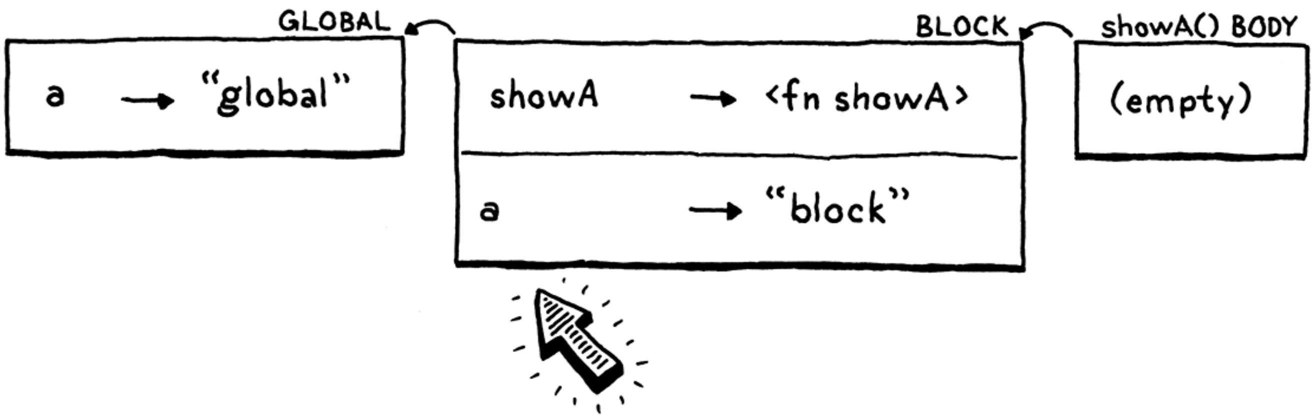
> Next, we declare the second a, this time inside the block.

接下来，我们声明第二个a，这次是在代码块内。



> It's in the same block—the same scope—as showA(), so it goes into the same environment, which is also the same environment showA()'s closure refers to. This is where it gets interesting. We call showA() again.

它和showA()在同一个代码块中——同一个作用域，所以它进入了同一个环境，也就是showA()的闭包所指向的环境。这就是有趣的地方了。我们再次调用showA()。



> We create a new empty environment for the body of showA() again, wire it up to that closure, and run the body. When the interpreter walks the chain of environments to find a, it now discovers the *new* a in the block environment. Boo.

我们再次为showA()的函数体创建了一个新的空环境，将其连接到该闭包，并运行函数体。当解释器遍历环境链去查找a时，它会发现代码块环境中新的变量a。

> I chose to implement environments in a way that I hoped would agree with your informal intuition around scopes. We tend to consider all of the code within a block as being within the same scope, so our interpreter uses a single environment to represent that. Each environment is a mutable hash table. When a new local variable is declared, it gets added to the existing environment for that scope.

我选择了一种实现环境的方式，希望它能够与您对作用域的非正式直觉相一致。我们倾向于认为一个块中的所有代码在同一个作用域中，所以我们的解释器使用了一个环境来表示它。每个环境都是一个可变的hash表。当一个新的局部变量被声明时，它会被加入该作用域的现有环境中。

> That intuition, like many in life, isn't quite right. A block is not necessarily all the same scope. Consider:

就像生活中的很多直觉一样，这种直觉并不完全正确。一个代码块并不一定都是同一个作用域。考虑一下：

```
{
  var a;
  // 1.
  var b;
  // 2.
}
```

> At the first marked line, only a is in scope. At the second line, both a and b are. If you define a "scope" to be a set of declarations, then those are clearly not the same scope—they don't contain the same declarations. It's like each var statement splits the block into two separate scopes, the scope before the variable is declared and the one after, which includes the new variable.

在标记的第一行，作用域中只有a。在第二行时，a和b都在其中。如果将作用域定义为一组声明，那么它们显然不是相同的作用域——它们不包含相同的声明。这就好像是var语句将代码块分割成了两个独立的作用域，变量声明前的作用域和包含新变量的作用域^4。

> But in our implementation, environments do act like the entire block is one scope, just a scope that changes over time. Closures do not like that. When a function is declared, it captures a reference to the current environment. The function *should* capture a frozen snapshot of the environment *as it existed at the moment the function was declared*. But instead, in the Java code, it has a reference to the actual mutable environment object. When a variable is later declared in the scope that environment corresponds to, the closure sees the new variable, even though the declaration does *not* precede the function.

但是在我们的实现中，环境确实表现得像整个代码块是一个作用域，只是这个作用域会随时间变化。而闭包不是这样的。当函数被声明时，它会捕获一个指向当前环境的引用。函数*应该*捕获一个冻结的环境快照，就像它存在于函数被声明的那一瞬间。但是事实上，在Java代码中，它引用的是一个实际可变的环境对象。当后续在该环境所对应的作用域内声明一个变量时，闭包会看到该变量，即使变量声明*没有*出现在函数之前。

> ## 11.1.2 Persistent environments

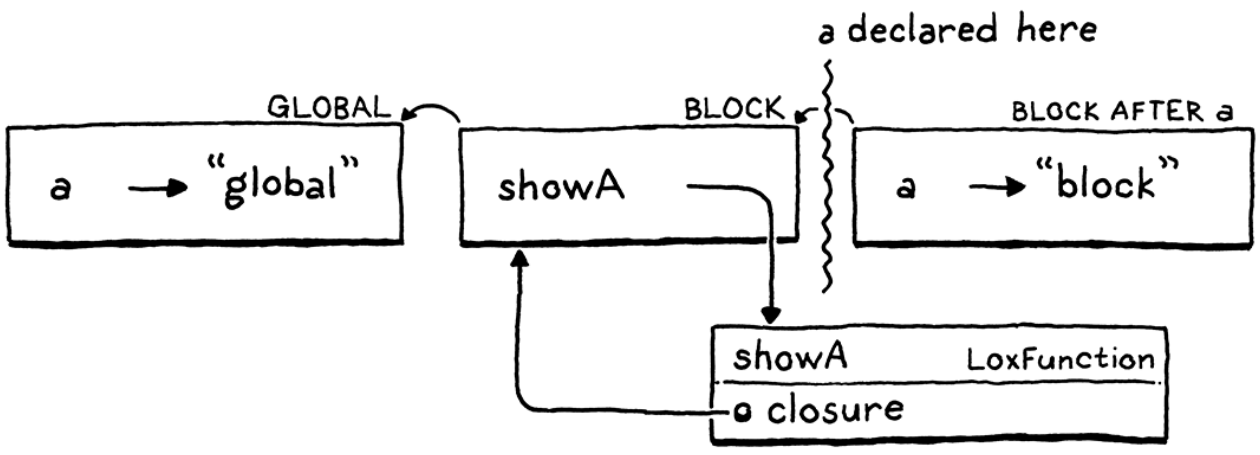
11.1.2 持久环境

> There is a style of programming that uses what are called **persistent data structures**. Unlike the squishy data structures you're familiar with in imperative programming, a persistent data structure can never be directly modified. Instead, any "modification" to an existing structure produces a brand new object that contains all of the original data and the new modification. The original is left unchanged.

有一种编程风格，使用所谓的**持久性数据结构**。与你在命令式编程中所熟悉的模糊的数据结构不同，持久化数据结构永远不能被直接修改。相应地，对现有结构的任何 "修改 "都会产生一个全新的对象，其中包含所有的原始数据和新的修改。而原有的对象则保持不变^5。

> If we were to apply that technique to Environment, then every time you declared a variable it would return a *new* environment that contained all of the previously declared variables along with the one new name. Declaring a variable would do the implicit "split" where you have an environment before the variable is declared and one after:

如果我们将这一技术应用于环境，那么每次你声明一个变量时，都会返回一个新的环境，其中包含所有先前声明的变量和一个新名称。声明一个变量会执行隐式分割，在声明变量之前与之后都有一个环境：



> A closure retains a reference to the Environment instance in play when the function was declared. Since any later declarations in that block would produce new Environment objects, the closure wouldn't see the new variables and our bug would be fixed.

当函数被声明时，闭包保留对正在运行的Environment实例的引用。由于该代码块中后续的任何声明都会生成新的Environment对象，闭包就不会看到新的变量，我们的问题也得到修复。

> This is a legit way to solve the problem, and it's the classic way to implement environments in Scheme interpreters. We could do that for Lox, but it would mean going back and changing a pile of existing code.

这是解决该问题的合法方式，也是在Scheme解释器中实现变量环境的经典方式。对于Lox，我们可以这样做，但是这意味着要回头修改一大堆现有的代码。

> I won't drag you through that. We'll keep the way we represent environments the same. Instead of making the data more statically structured, we'll bake the static resolution into the access *operation* itself.

我不会把你拖下水的。我们将保持表示环境的方式不变。我们不会让数据变得更加静态结构化，而是将静态解析嵌入访问操作本身。

11.2 Semantic Analysis

11.2 语义分析

> Our interpreter **resolves** a variable—tracks down which declaration it refers to—each and every time the variable expression is evaluated. If that variable is swaddled inside a loop that runs a thousand times, that variable gets re-resolved a thousand times.

我们的解释器每次对变量表达式求值时，都会**解析**变量——追踪它所指向的声明。如果这个变量被包在一个运行1000次的循环中，那么该变量就会被重复解析1000次。

> We know static scope means that a variable usage always resolves to the same declaration, which can be determined just by looking at the text. Given that, why are we doing it dynamically every time? Doing so doesn't just open the hole that leads to our annoying bug, it's also needlessly slow.

我们知道静态作用域意味着一个变量的使用总是解析到同一个声明，而且可以通过查看文本来确定。既然如此，我们为什么每次都要动态地解析呢？这样做不仅仅导致了这个恼人的bug，而且也造成了不必要的低效。

> A better solution is to resolve each variable use *once*. Write a chunk of code that inspects the user's program, finds every variable mentioned, and figures out which declaration each refers to. This process is an example of a **semantic analysis**. Where a parser tells only if a program is grammatically correct (a *syntactic* analysis), semantic analysis goes farther and starts to figure out what pieces of the program actually mean. In this case, our analysis will resolve variable bindings. We'll know not just that an expression *is* a variable, but *which* variable it is.

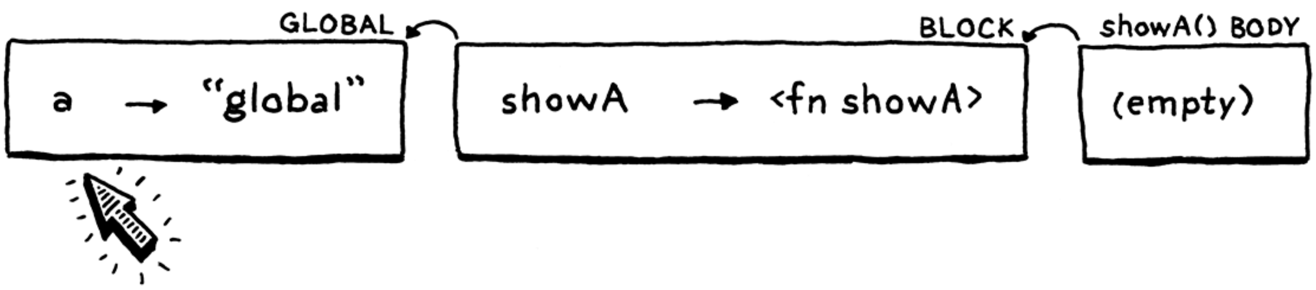
一个更好的解决方案是一次性解析每个变量的使用。编写一段代码，检查用户的程序，找到所提到的每个变量，并找出每个变量引用的是哪个声明。这个过程是**语义分析**的一个例子。解析器只能分析程序在语法上是否正确(语法分析)，而语义分析则更进一步，开始弄清楚程序的各个部分的实际含义。在这种情况下，我们的分析将解决变量绑定的问题。我们不仅要知道一个表达式是一个变量，还要知道它是哪个变量。

> There are a lot of ways we could store the binding between a variable and its declaration. When we get to the C interpreter for Lox, we'll have a *much* more efficient way of storing and accessing local variables. But for jlox, I want to minimize the collateral damage we inflict on our existing codebase. I'd hate to throw out a bunch of mostly fine code.

有很多方法可以存储变量及其声明直接的绑定关系。当我们使用Lox的C解释器时，我们将有一种更有效的方式来存储和访问局部变量。但是对于jlox来说，我想尽量减少对现有代码库的附带损害。我不希望扔掉一堆基本上都很好的代码。

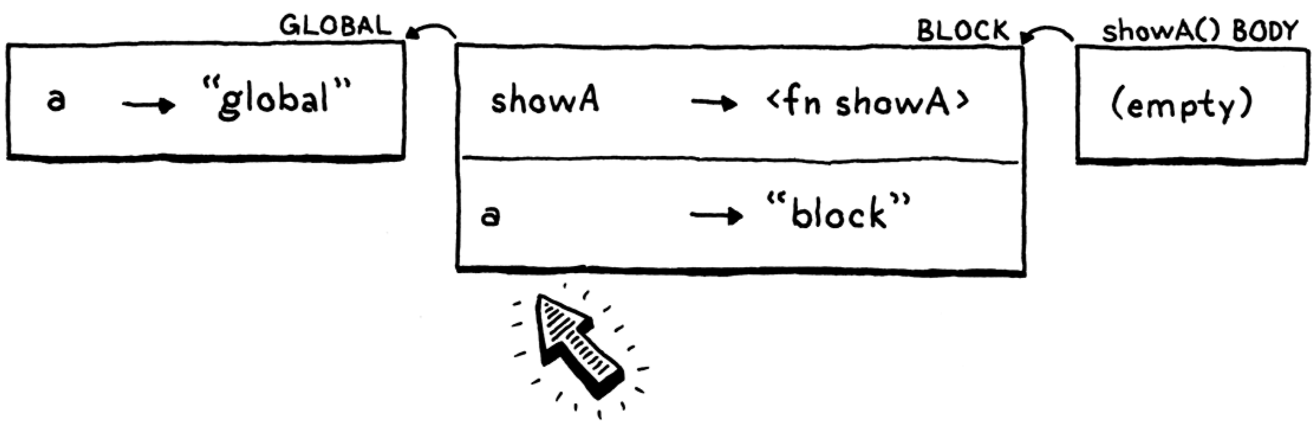
> Instead, we'll store the resolution in a way that makes the most out of our existing Environment class. Recall how the accesses of a are interpreted in the problematic example.

相对地，我们将以最充分利用现有Environment类的方式来存储解析结果。回想一下，在有问题的例子中，a的访问是如何被解释的。



> In the first (correct) evaluation, we look at three environments in the chain before finding the global declaration of a. Then, when the inner a is later declared in a block scope, it shadows the global one.

在第一次（正确的）求值中，我们会检查链中的环境，并找到a的全局声明。然后，当内部的a在块作用域中声明时，它会遮蔽全局的变量a。



> The next lookup walks the chain, finds a in the *second* environment and stops there. Each environment corresponds to a single lexical scope where variables are declared. If we could ensure a variable lookup always walked the *same* number of links in the environment chain, that would ensure that it found the same variable in the same scope every time.

下一次查找会遍历环境链，在第二个环境中找到a并停止。每个环境都对应于一个声明变量的词法作用域。如果我们能够保证变量查找总是在环境链上遍历相同数量的链接，也就可以保证每次都可以在相同的作用域中找到相同的变量。

> To "resolve" a variable usage, we only need to calculate how many "hops" away the declared variable will be in the environment chain. The interesting question is *when* to do this calculation—or, put differently, where in our interpreter's implementation do we stuff the code for it?

要"解析"一个变量使用，我们只需要计算声明的变量在环境链中有多少"跳"。有趣的问题是在什么时候进行这个计算——或者换句话说，在解释器的实现中，这段代码要添加到什么地方？

> Since we're calculating a static property based on the structure of the source code, the obvious answer is in the parser. That is the traditional home, and is where we'll put it later in clox. It would work here too, but I want an excuse to show you another technique. We'll write our resolver as a separate pass.

因为我们是根据源代码的结构来计算一个静态属性，所以答案显然是在解析器中。那是传统的选择，也是我们以后在 clox 中实现它的地方。在这里同样也适用，但是我想给你展示另一种技巧。我们会单独写一个解析器。

> ### 11.2.1 A variable resolution pass

11.2.1 变量解析过程

> After the parser produces the syntax tree, but before the interpreter starts executing it, we'll do a single walk over the tree to resolve all of the variables it contains. Additional passes between parsing and execution are common. If Lox had static types, we could slide a type checker in there. Optimizations are often implemented in separate passes like this too. Basically, any work that doesn't rely on state that's only available at runtime can be done in this way.

在解析器生成语法树之后，解释器执行语法树之前，我们会对语法树再进行一次遍历，以解析其中包含的变量。在解析和执行之间的额外遍历是很常见的。如果Lox中有静态类型，我们可以插入一个类型检查器。优化也经常是在类似单独的遍历过程中实现的。基本上，任何不依赖于运行时状态的工作都可以通过这种方式完成。

> Our variable resolution pass works like a sort of mini-interpreter. It walks the tree, visiting each node, but a static analysis is different from a dynamic execution:

我们的变量解析工作就像一个小型的解释器。它会遍历整棵树，访问每个节点，但是静态分析与动态执行还是不同的：

- > **There are no side effects.** When the static analysis visits a print statement, it doesn't actually print anything. Calls to native functions or other operations that reach out to the outside world are stubbed out and have no effect.

  没有副作用。当静态分析处理一个`print`语句时，它并不会打印任何东西。对本地函数或其它与外部世界联系的操作也会被终止，并且没有任何影响。

- **There is no control flow.** Loops are visited only once. Both branches are visited in `if` statements. Logic operators are not short-circuited.

  没有控制流。循环只会被处理一次，`if`语句中的两个分支都会处理，逻辑操作符也不会做短路处理^6。

11.3 A Resolver Class

11.3 Resolver类

> Like everything in Java, our variable resolution pass is embodied in a class.

与Java中的所有内容一样，我们将变量解析处理也放在一个类中。

lox/Resolver.java，创建新文件：

```java
package com.craftinginterpreters.lox;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Stack;

class Resolver implements Expr.Visitor<Void>, Stmt.Visitor<Void> {
  private final Interpreter interpreter;

  Resolver(Interpreter interpreter) {
    this.interpreter = interpreter;
  }
}
```

> Since the resolver needs to visit every node in the syntax tree, it implements the visitor abstraction we already have in place. Only a few kinds of nodes are interesting when it comes to resolving variables:

因为解析器需要处理语法树中的每个节点，所以它实现了我们已有的访问者抽象。在解析变量时，有几个节点是比较特殊的：

- > A block statement introduces a new scope for the statements it contains.

  块语句为它所包含的语句引入了一个新的作用域。

- > A function declaration introduces a new scope for its body and binds its parameters in that scope.

  函数声明为其函数体引入了一个新的作用域，并在该作用域中绑定了它的形参。

- > A variable declaration adds a new variable to the current scope.

  变量声明将一个新变量加入到当前作用域中。

- > Variable and assignment expressions need to have their variables resolved.

  变量定义和赋值表达式需要解析它们的变量值。

> The rest of the nodes don't do anything special, but we still need to implement visit methods for them that traverse into their subtrees. Even though a + expression doesn't *itself* have any variables to resolve, either of its operands might.

其余的节点不做任何特别的事情，但是我们仍然需要为它们实现visit方法，以遍历其子树。尽管+表达式本身没有任何变量需要解析，但是它的任一操作数都可能需要。

> ### 11.3.1 Resolving blocks

11.3.1 解析代码块

> We start with blocks since they create the local scopes where all the magic happens.

我们从块语法开始，因为它们创建了局部作用域——魔法出现的地方。

lox/Resolver.java，在 Resolver()方法后添加：

```
  @Override
  public Void visitBlockStmt(Stmt.Block stmt) {
    beginScope();
    resolve(stmt.statements);
    endScope();
    return null;
  }
```

> This begins a new scope, traverses into the statements inside the block, and then discards the scope. The fun stuff lives in those helper methods. We start with the simple one.

这里会开始一个新的作用域，遍历块中的语句，然后丢弃该作用域。有趣的部分都在这些辅助方法中。我们先看一个简单的。

lox/Resolver.java，在 Resolver()方法后添加：

```
  void resolve(List<Stmt> statements) {
    for (Stmt statement : statements) {
      resolve(statement);
    }
  }
```

> This walks a list of statements and resolves each one. It in turn calls:

它会遍历语句列表，并解析其中每一条语句。它会进一步调用：

lox/Resolver.java，在 visitBlockStmt()方法后添加：

```
  private void resolve(Stmt stmt) {
    stmt.accept(this);
  }
```

> While we're at it, let's add another overload that we'll need later for resolving an expression.

在此过程中，让我们添加一个后续解析表达式时会用到的重载方法。

lox/Resolver.java，在 resolve(Stmt stmt)方法后添加：

```
  private void resolve(Expr expr) {
    expr.accept(this);
  }
```

> These methods are similar to the `evaluate()` and `execute()` methods in Interpreter—they turn around and apply the Visitor pattern to the given syntax tree node.

这些方法与解释器中的 `evaluate()`和`execute()`方法类似——它们会反过来将访问者模式应用到语法树节点。

> The real interesting behavior is around scopes. A new block scope is created like so:

真正有趣的部分是围绕作用域的。一个新的块作用域是这样创建的：

lox/Resolver.java，在 resolve()方法后添加：

```
  private void beginScope() {
    scopes.push(new HashMap<String, Boolean>());
  }
```

> Lexical scopes nest in both the interpreter and the resolver. They behave like a stack. The interpreter implements that stack using a linked list—the chain of Environment objects. In the resolver, we use an actual Java Stack.

词法作用域在解释器和解析器中都有使用。它们的行为像一个栈。解释器是使用链表（Environment对象组成的链）来实现栈的，在解析器中，我们使用一个真正的Java Stack。

lox/Resolver.java，在 Resolver类中添加：

```
  private final Interpreter interpreter;
  // 新增部分开始
  private final Stack<Map<String, Boolean>> scopes = new Stack<>();
  // 新增部分结束
  Resolver(Interpreter interpreter) {
```

> This field keeps track of the stack of scopes currently, uh, in scope. Each element in the stack is a Map representing a single block scope. Keys, as in Environment, are variable names. The values are Booleans, for a reason I'll explain soon.

这个字段会记录当前作用域内的栈。栈中的每个元素是代表一个块作用域的Map。与Environment中一样，键是变量名。值是布尔值，原因我很快会解释。

> The scope stack is only used for local block scopes. Variables declared at the top level in the global scope are not tracked by the resolver since they are more dynamic in Lox. When resolving a variable, if we can't find it in the stack of local scopes, we assume it must be global.

作用域栈只用于局部块作用域。解析器不会跟踪在全局作用域的顶层声明的变量，因为它们在Lox中是更动态的。当解析一个变量时，如果我们在本地作用域栈中找不到它，我们就认为它一定是全局的。

> Since scopes are stored in an explicit stack, exiting one is straightforward.

由于作用域被存储在一个显式的栈中，退出作用域很简单。

lox/Resolver.java，在 beginScope()方法后添加：

```
  private void endScope() {
    scopes.pop();
  }
```

> Now we can push and pop a stack of empty scopes. Let's put some things in them.

现在我们可以在一个栈中压入和弹出一个空作用域，接下来我们往里面放些内容。

> ### 11.3.2 Resolving variable declarations

11.3.2 解析变量声明

> Resolving a variable declaration adds a new entry to the current innermost scope's map. That seems simple, but there's a little dance we need to do.

解析一个变量声明，会在当前最内层的作用域map中添加一个新的条目。这看起来很简单，但是我们需要做一些小动作。

lox/Resolver.java，在 visitBlockStmt()方法后添加：

```
  @Override
  public Void visitVarStmt(Stmt.Var stmt) {
    declare(stmt.name);
    if (stmt.initializer != null) {
      resolve(stmt.initializer);
    }
    define(stmt.name);
    return null;
  }
```

> We split binding into two steps, declaring then defining, in order to handle funny edge cases like this:

我们将绑定分为两个步骤，先声明，然后定义，以便处理类似下面这样的边界情况：

```
var a = "outer";
{
  var a = a;
}
```

> What happens when the initializer for a local variable refers to a variable with the same name as the variable being declared? We have a few options:

当局部变量的初始化式指向一个与当前声明变量名称相同的变量时，会发生什么？我们有几个选择：

1. > **Run the initializer, then put the new variable in scope.** Here, the new local `a` would be initialized with "outer", the value of the *global* one. In other words, the previous declaration would desugar to:

   1 **运行初始化式，然后将新的变量放入作用域中。** 在这个例子中，新的局部变量`a`会使用"outer"（全局变量`a`的值）初始化。换句话说，前面的声明脱糖后如下：

   ```
   var temp = a; // Run the initializer.
   var a;        // Declare the variable.
   a = temp;     // Initialize it.
   ```

2. > **Put the new variable in scope, then run the initializer.** This means you could observe a variable before it's initialized, so we would need to figure out what value it would have then. Probably `nil`. That means the new local `a` would be re-initialized to its own implicitly initialized value, `nil`. Now the desugaring would look like:

   2 **将新的变量放入作用域中，然后运行初始化式。** 这意味着你可以在变量被初始化之前观察到它，所以当我们需要计算出它的值时，这个值其实是`nil`。这意味着新的局部变量`a`将被重新初始化为它自己的隐式初始化值`nil`。现在，脱糖后的结果如下：

```
var a; // Define the variable.
a = a; // Run the initializer.
```

3. > **Make it an error to reference a variable in its initializer.** Have the interpreter fail either at compile time or runtime if an initializer mentions the variable being initialized.

   3 **在初始化式中引用一个变量是错误的。** 如果初始化式使用了要初始化的变量，则解释器在编译时或运行时都会失败。

> Do either of those first two options look like something a user actually *wants*? Shadowing is rare and often an error, so initializing a shadowing variable based on the value of the shadowed one seems unlikely to be deliberate.

前两个选项中是否有用户真正*想要*的？变量遮蔽很少见，而且通常是一个错误，所以根据被遮蔽的变量值来初始化一个遮蔽的变量，似乎不太可能是有意为之。

> The second option is even less useful. The new variable will *always* have the value `nil`. There is never any point in mentioning it by name. You could use an explicit `nil` instead.

第二个选项就更没用了。新变量的值总是`nil`。通过名称来引用没有任何意义。你可以使用一个隐式的`nil`来代替。

> Since the first two options are likely to mask user errors, we'll take the third. Further, we'll make it a compile error instead of a runtime one. That way, the user is alerted to the problem before any code is run.

由于前两个选项可能会掩盖用户的错误，我们将采用第三个选项。此外，我们要将其作为一个编译错误而不是运行时错误。这样一来，在代码运行之前，用户就会收到该问题的警报。

> In order to do that, as we visit expressions, we need to know if we're inside the initializer for some variable. We do that by splitting binding into two steps. The first is **declaring** it.

要做到这一点，当我们访问表达式时，我们需要知道当前是否在某个变量的初始化式中。我们通过将绑定拆分为两步来实现。首先是**声明**。

lox/Resolver.java，在 endScope()方法后添加：

```
  private void declare(Token name) {
    if (scopes.isEmpty()) return;

    Map<String, Boolean> scope = scopes.peek();
    scope.put(name.lexeme, false);
  }
```

> Declaration adds the variable to the innermost scope so that it shadows any outer one and so that we know the variable exists. We mark it as "not ready yet" by binding its name to `false` in the scope map. The value associated with a key in the scope map represents whether or not we have finished resolving that variable's initializer.

声明将变量添加到最内层的作用域，这样它就会遮蔽任何外层作用域，我们也就知道了这个变量的存在。我们通过在作用域map中将其名称绑定到`false`来表明该变量“尚未就绪”。作用域map中与key相关联的值代表的是我们是否已经结束了对变量初始化式的解析。

> After declaring the variable, we resolve its initializer expression in that same scope where the new variable now exists but is unavailable. Once the initializer expression is done, the variable is ready for prime time. We do that by **defining** it.

在声明完变量后，我们在变量当前存在但是不可用的作用域中解析变量的初始化表达式。一旦初始化表达式完成，变量也就绪了。我们通过**define**来实现。

lox/Resolver.java，在 declare()方法后添加：

```
  private void define(Token name) {
    if (scopes.isEmpty()) return;
    scopes.peek().put(name.lexeme, true);
  }
```

> We set the variable's value in the scope map to `true` to mark it as fully initialized and available for use. It's alive!

我们在作用域map中将变量的值置为`true`，以标记它已完全初始化并可使用。它有了生命！

> ### 11.3.3 Resolving variable expressions

11.3.3 解析变量表达式

> Variable declarations—and function declarations, which we'll get to—write to the scope maps. Those maps are read when we resolve variable expressions.

变量声明——以及我们即将讨论的函数声明——会向作用域map中写数据。在我们解析变量表达式时，需要读取这些map。

lox/Resolver.java，在 visitVarStmt()方法后添加：

```
  @Override
  public Void visitVariableExpr(Expr.Variable expr) {
    if (!scopes.isEmpty() &&
        scopes.peek().get(expr.name.lexeme) == Boolean.FALSE) {
      Lox.error(expr.name,
          "Can't read local variable in its own initializer.");
    }

    resolveLocal(expr, expr.name);
    return null;
  }
```

> First, we check to see if the variable is being accessed inside its own initializer. This is where the values in the scope map come into play. If the variable exists in the current scope but its value is `false`, that means we have declared it but not yet defined it. We report that error.

首先，我们要检查变量是否在其自身的初始化式中被访问。这也就是作用域map中的值发挥作用的地方。如果当前作用域中存在该变量，但是它的值是`false`，意味着我们已经声明了它，但是还没有定义它。我们会报告一个错误出来。

> After that check, we actually resolve the variable itself using this helper:

在检查之后，我们实际上使用了这个辅助方法来解析变量：

lox/Resolver.java，在 define()方法后添加：

```
  private void resolveLocal(Expr expr, Token name) {
    for (int i = scopes.size() - 1; i >= 0; i--) {
      if (scopes.get(i).containsKey(name.lexeme)) {
        interpreter.resolve(expr, scopes.size() - 1 - i);
        return;
      }
    }
  }
```

> This looks, for good reason, a lot like the code in Environment for evaluating a variable. We start at the innermost scope and work outwards, looking in each map for a matching name. If we find the variable, we resolve it, passing in the number of scopes between the current innermost scope and the scope where the variable was found. So, if the variable was found in the current scope, we pass in 0. If it's in the immediately enclosing scope, 1. You get the idea.

这看起来很像是Environment中对变量求值的代码。我们从最内层的作用域开始，向外扩展，在每个map中寻找一个可以匹配的名称。如果我们找到了这个变量，我们就对其解析，传入当前最内层作用域和变量所在作用域之间的作用域的数量。所以，如果变量在当前作用域中找到该变量，则传入0；如果在紧邻的外网作用域中找到，则传1。明白了吧。

> If we walk through all of the block scopes and never find the variable, we leave it unresolved and assume it's global. We'll get to the implementation of that `resolve()` method a little later. For now, let's keep on cranking through the other syntax nodes.

如果我们遍历了所有的作用域也没有找到这个变量，我们就不解析它，并假定它是一个全局变量。稍后我们将讨论`resolve()`方法的实现。现在，让我们继续浏览其他语法节点。

> ### 11.3.4 Resolving assignment expressions

11.3.4 解析赋值表达式

> The other expression that references a variable is assignment. Resolving one looks like this:

另一个引用变量的表达式就是赋值表达式。解析方法如下：

lox/Resolver.java，在 visitVarStmt()方法后添加：

```
@Override
public Void visitAssignExpr(Expr.Assign expr) {
  resolve(expr.value);
  resolveLocal(expr, expr.name);
  return null;
}
```

> First, we resolve the expression for the assigned value in case it also contains references to other variables. Then we use our existing `resolveLocal()` method to resolve the variable that’s being assigned to.

首先，我们解析右值的表达式，以防它还包含对其它变量的引用。然后使用现有的 `resolveLocal()` 方法解析待赋值的变量。

> ### 11.3.5 Resolving function declarations

11.3.5 解析函数声明

> Finally, functions. Functions both bind names and introduce a scope. The name of the function itself is bound in the surrounding scope where the function is declared. When we step into the function’s body, we also bind its parameters into that inner function scope.

最后是函数。函数既绑定名称又引入了作用域。函数本身的名称被绑定在函数声明时所在的作用域中。当我们进入函数体时，我们还需要将其参数绑定到函数内部作用域中。

lox/Resolver.java，在 visitBlockStmt()方法后添加：

```
@Override
public Void visitFunctionStmt(Stmt.Function stmt) {
  declare(stmt.name);
  define(stmt.name);

  resolveFunction(stmt);
  return null;
}
```

> Similar to `visitVariableStmt()`, we declare and define the name of the function in the current scope. Unlike variables, though, we define the name eagerly, before resolving the function’s body. This lets a function recursively refer to itself inside its own body.

与`visitVariableStmt()`类似，我们在当前作用域中声明并定义函数的名称。与变量不同的是，我们在解析函数体之前，就急切地定义了这个名称。这样函数就可以在自己的函数体中递归地使用自身。

> Then we resolve the function’s body using this:

那么我们可以使用下面的方法来解析函数体：

lox/Resolver.java，在 resolve()方法后添加：

```
private void resolveFunction(Stmt.Function function) {
  beginScope();
  for (Token param : function.params) {
    declare(param);
    define(param);
  }
  resolve(function.body);
  endScope();
}
```

> It's a separate method since we will also use it for resolving Lox methods when we add classes later. It creates a new scope for the body and then binds variables for each of the function's parameters.

这是一个单独的方法，因为我们以后添加类时，还需要使用它来解析Lox方法。它为函数体创建一个新的作用域，然后为函数的每个参数绑定变量。

> Once that's ready, it resolves the function body in that scope. This is different from how the interpreter handles function declarations. At *runtime*, declaring a function doesn't do anything with the function's body. The body doesn't get touched until later when the function is called. In a *static* analysis, we immediately traverse into the body right then and there.

一旦就绪，它就会在这个作用域中解析函数体。这与解释器处理函数声明的方式不同。在*运行时*，声明一个函数不会对函数体做任何处理。直到后续函数被调用时，才会触及主体。在*静态*分析中，我们会立即遍历函数体。

> ### 11.3.6 Resolving the other syntax tree nodes

11.3.6 解析其它语法树节点

> That covers the interesting corners of the grammars. We handle every place where a variable is declared, read, or written, and every place where a scope is created or destroyed. Even though they aren't affected by variable resolution, we also need visit methods for all of the other syntax tree nodes in order to recurse into their subtrees. Sorry this bit is boring, but bear with me. We'll go kind of "top down" and start with statements.

这涵盖了语法中很多有趣的部分。我们处理了声明、读取、写入遍历，创建、销毁作用域的部分。虽然其它部分不受遍历解析的影响，我们也需要为其它语法树节点提供visit方法，以便递归到它们的子树。抱歉，这部分内容很枯燥，但请耐心听我讲。我们采用"自上而下"的方式，从语句开始。

> An expression statement contains a single expression to traverse.

一个表达式语句中包含一个需要遍历的表达式。

lox/Resolver.java，在 visitBlockStmt()方法后添加：

```
@Override
public Void visitExpressionStmt(Stmt.Expression stmt) {
  resolve(stmt.expression);
```

```
    return null;
  }
```

> An if statement has an expression for its condition and one or two statements for the branches.

if语句包含一个条件表达式，以及一个或两个分支语句。

lox/Resolver.java，在 visitFunctionStmt()方法后添加：

```
  @Override
  public Void visitIfStmt(Stmt.If stmt) {
    resolve(stmt.condition);
    resolve(stmt.thenBranch);
    if (stmt.elseBranch != null) resolve(stmt.elseBranch);
    return null;
  }
```

> Here, we see how resolution is different from interpretation. When we resolve an if statement, there is no control flow. We resolve the condition and *both* branches. Where a dynamic execution steps only into the branch that *is* run, a static analysis is conservative—it analyzes any branch that *could* be run. Since either one could be reached at runtime, we resolve both.

在这里，我们可以看到解析与解释是不同的。当我们解析if语句时，没有控制流。我们会解析条件表达式和两个分支表达式。动态执行则只会进入*正在执行*的分支，而静态分析是保守的——它会分析所有*可能执行*的分支。因为任何一个分支在运行时都可能被触及，所以我们要对两者都进行解析。

> Like expression statements, a print statement contains a single subexpression.

与表达式语句类似，print语句也包含一个子表达式。

lox/Resolver.java，在 visitIfStmt()方法后添加：

```
  @Override
  public Void visitPrintStmt(Stmt.Print stmt) {
    resolve(stmt.expression);
    return null;
  }
```

> Same deal for return.

return语句也是相同的。

lox/Resolver.java，在 visitPrintStmt()方法后添加：

```
  @Override
  public Void visitReturnStmt(Stmt.Return stmt) {
    if (stmt.value != null) {
      resolve(stmt.value);
    }

    return null;
  }
```

> As in `if` statements, with a `while` statement, we resolve its condition and resolve the body exactly once.

与`if`语句一样，对于`while`语句，我们会解析其条件，并解析一次循环体。

lox/Resolver.java，在 visitVarStmt()方法后添加：

```
  @Override
  public Void visitWhileStmt(Stmt.While stmt) {
    resolve(stmt.condition);
    resolve(stmt.body);
    return null;
  }
```

> That covers all the statements. On to expressions . . .

这样就涵盖了所有的语句。接下来是表达式......

> Our old friend the binary expression. We traverse into and resolve both operands.

我们的老朋友二元表达式。我们要遍历并解析两个操作数。

lox/Resolver.java，在 visitAssignExpr()方法后添加：

```
  @Override
  public Void visitBinaryExpr(Expr.Binary expr) {
    resolve(expr.left);
    resolve(expr.right);
    return null;
  }
```

> Calls are similar—we walk the argument list and resolve them all. The thing being called is also an expression (usually a variable expression), so that gets resolved too.

调用也是类似的——我们遍历参数列表并解析它们。被调用的对象也是一个表达式（通常是一个变量表达式），所以它也会被解析。

lox/Resolver.java，在 visitBinaryExpr()方法后添加：

```
  @Override
  public Void visitCallExpr(Expr.Call expr) {
    resolve(expr.callee);
    for (Expr argument : expr.arguments) {
      resolve(argument);
    }

    return null;
  }
```

> Parentheses are easy.

括号表达式比较简单。

lox/Resolver.java，在 visitCallExpr()方法后添加：

```
  @Override
  public Void visitGroupingExpr(Expr.Grouping expr) {
    resolve(expr.expression);
    return null;
  }
```

> Literals are easiest of all.

字面量表达式是最简单的。

lox/Resolver.java，在 visitGroupingExpr()方法后添加：

```
  @Override
  public Void visitLiteralExpr(Expr.Literal expr) {
    return null;
  }
```

> A literal expression doesn't mention any variables and doesn't contain any subexpressions so there is no work to do.

字面表达式中没有使用任何变量，也不包含任何子表达式，所以也不需要做任何事情。

> Since a static analysis does no control flow or short-circuiting, logical expressions are exactly the same as other binary operators.

因为静态分析没有控制流或短路处理，逻辑表达式与其它的二元运算符是一样的。

lox/Resolver.java，在 visitLiteralExpr()方法后添加：

```
  @Override
  public Void visitLogicalExpr(Expr.Logical expr) {
    resolve(expr.left);
    resolve(expr.right);
    return null;
  }
```

> And, finally, the last node. We resolve its one operand.

接下来是最后一个节点，我们解析它的一个操作数。

lox/Resolver.java，在 visitLogicalExpr()方法后添加：

```
  @Override
  public Void visitUnaryExpr(Expr.Unary expr) {
    resolve(expr.right);
    return null;
  }
```

> With all of these visit methods, the Java compiler should be satisfied that Resolver fully implements Stmt.Visitor and Expr.Visitor. Now is a good time to take a break, have a snack, maybe a little nap.

有了这些visit方法，Java编译器应该会认为Resolver完全实现了Stmt.Visitor 和 Expr.Visitor。现在是时候休息一下了。

11.4 Interpreting Resolved Variables

11.4 解释已解析的变量

> Let's see what our resolver is good for. Each time it visits a variable, it tells the interpreter how many scopes there are between the current scope and the scope where the variable is defined. At runtime, this corresponds exactly to the number of *environments* between the current one and the enclosing one where the interpreter can find the variable's value. The resolver hands that number to the interpreter by calling this:

让我们看看解析器有什么用处。每次访问一个变量时，它都会告诉解释器，在当前作用域和变量定义的作用域之间隔着多少层作用域。在运行时，这正好对应于当前环境与解释器可以找到变量值的外围环境之间的*environments*数量。解析器通过调用下面的方法将这个数字传递给解释器：

lox/Interpreter.java，在 execute()方法后添加：

```
  void resolve(Expr expr, int depth) {
    locals.put(expr, depth);
  }
```

> We want to store the resolution information somewhere so we can use it when the variable or assignment expression is later executed, but where? One obvious place is right in the syntax tree node itself. That's a fine approach, and that's where many compilers store the results of analyses like this.

我们要把解析信息存储在某个地方，这样在执行变量表达式和赋值表达式时就可以使用它，但是要存在哪里呢？一个明显的位置就是语法树节点本身。这是一个很好的方法，许多编译器都是在这里存储类似的分析结果

的。

> We could do that, but it would require mucking around with our syntax tree generator. Instead, we'll take another common approach and store it off to the side in a map that associates each syntax tree node with its resolved data.

我们可以这样做，但是需要对我们的语法树生成器进行修改。相反，我们会采用另一种常见的方法，将其存储在一个map中，将每个语法树节点与其解析的数据关联起来。

> Interactive tools like IDEs often incrementally reparse and re-resolve parts of the user's program. It may be hard to find all of the bits of state that need recalculating when they're hiding in the foliage of the syntax tree. A benefit of storing this data outside of the nodes is that it makes it easy to *discard* it—simply clear the map.

像IDE这种的交互式工具经常会增量地对用户的部分代码进行重新分析和解析。当这些状态隐藏在语法树的枝叶中时，可能很难找到所有需要重新计算的状态。将这些数据存储在节点之外的好处之一就是，可以很容易地丢弃这部分数据——只需要清除map即可。

lox/Interpreter.java，在 *Interpreter*类中添加

```
  private Environment environment = globals;
  // 新增部分开始
  private final Map<Expr, Integer> locals = new HashMap<>();
  // 新增部分结束
  Interpreter() {
```

> You might think we'd need some sort of nested tree structure to avoid getting confused when there are multiple expressions that reference the same variable, but each expression node is its own Java object with its own unique identity. A single monolithic map doesn't have any trouble keeping them separated.

你可能认为我们需要某种嵌套的树状结构，以避免在有多个表达式引用同一个变量时出现混乱，但是每个表达式节点都有其对应的Java对象，具有唯一性标识。一个简单的map就足以将它们全部区分开来。

> As usual, using a collection requires us to import a couple of names.

与之前一样，使用集合需要先引入一些包名称。

lox/Interpreter.java，添加：

```
import java.util.ArrayList;
// 新增部分开始
import java.util.HashMap;
// 新增部分结束
import java.util.List;
```

> And:

还有：

lox/Interpreter.java，添加：

```
import java.util.List;
// 新增部分开始
import java.util.Map;
// 新增部分结束
class Interpreter implements Expr.Visitor<Object>,
```

11.4.1Accessing a resolved variable

11.4.1 访问已解析的变量

> Our interpreter now has access to each variable's resolved location. Finally, we get to make use of that. We replace the visit method for variable expressions with this:

我们的解释器现在可以访问每个变量的解析位置。最后，我们可以利用这一点了，将变量表达式的visit方法替换如下：

lox/Interpreter.java，在 visitVariableExpr()方法中替换一行：

```
  public Object visitVariableExpr(Expr.Variable expr) {
    // 替换部分开始
    return lookUpVariable(expr.name, expr);
    // 替换部分结束
  }
```

> That delegates to:

这里引用了：

lox/Interpreter.java，在 visitVariableExpr()方法后添加：

```
  private Object lookUpVariable(Token name, Expr expr) {
    Integer distance = locals.get(expr);
    if (distance != null) {
      return environment.getAt(distance, name.lexeme);
    } else {
      return globals.get(name);
    }
  }
```

> There are a couple of things going on here. First, we look up the resolved distance in the map. Remember that we resolved only *local* variables. Globals are treated specially and don't end up in the map (hence the name `locals`). So, if we don't find a distance in the map, it must be global. In that

> case, we look it up, dynamically, directly in the global environment. That throws a runtime error if the variable isn't defined.

这里有几件事要做。首先，我们在map中查找已解析的距离值。要记住，我们只解析了本地变量。全局变量被特殊处理了，不会出现了map中（所以它的名字叫`locals`）。所以，如果我们没有在map中找到变量对应的距离值，它一定是全局变量。在这种情况下，我们直接在全局environment中查找。如果变量没有被定义，就会产生一个运行时错误。

> If we *do* get a distance, we have a local variable, and we get to take advantage of the results of our static analysis. Instead of calling `get()`, we call this new method on Environment:

如果我们*确实*查到了一个距离值，那这就是个局部变量，我们可以利用静态分析的结果。我们不会调用`get()`方法，而是调用下面这个Environment中的新方法：

lox/Environment.java，在 define()方法后添加：

```
  Object getAt(int distance, String name) {
    return ancestor(distance).values.get(name);
  }
```

> The old `get()` method dynamically walks the chain of enclosing environments, scouring each one to see if the variable might be hiding in there somewhere. But now we know exactly which environment in the chain will have the variable. We reach it using this helper method:

原先的`get()`方法会动态遍历外围的环境链，搜索每一个环境，查看变量是否包含在其中。但是现在我们明确知道链路中的哪个环境中会包含该变量。我们使用下面的辅助方法直达这个环境：

lox/Environment.java，在 define()方法后添加：

```
  Environment ancestor(int distance) {
    Environment environment = this;
    for (int i = 0; i < distance; i++) {
      environment = environment.enclosing;
    }

    return environment;
  }
```

> This walks a fixed number of hops up the parent chain and returns the environment there. Once we have that, `getAt()` simply returns the value of the variable in that environment's map. It doesn't even have to check to see if the variable is there—we know it will be because the resolver already found it before.

该方法在环境链中经过确定的跳数之后，返回对应的环境。一旦我们有了环境，`getAt()`方法就可以直接返回对应环境map中的变量值。甚至不需要检查变量是否存在——我们知道它是存在的，因为解析器之前已经确认过了^7。

> ## 11.4.2 Assigning to a resolved variable

11.4.2 赋值已解析的变量

> We can also use a variable by assigning to it. The changes to visiting an assignment epression are similar.

我们也可以通过赋值来使用一个变量。赋值表达式对应的visit方法的修改也是类似的。

lox/Interpreter.java，在 visitAssignExpr()方法中替换一行：

```
  public Object visitAssignExpr(Expr.Assign expr) {
    Object value = evaluate(expr.value);
    // 替换部分开始
    Integer distance = locals.get(expr);
    if (distance != null) {
      environment.assignAt(distance, expr.name, value);
    } else {
      globals.assign(expr.name, value);
    }
    // 替换部分结束
    return value;
```

> Again, we look up the variable's scope distance. If not found, we assume it's global and handle it the same way as before. Otherwise, we call this new method:

又一次，我们要查找变量的作用域距离。如果没有找到，我们就假定它是全局变量并采用跟之前一样的方式来处理；否则，我们使用下面的新方法：

lox/Environment.java，在 getAt()方法后添加：

```
  void assignAt(int distance, Token name, Object value) {
    ancestor(distance).values.put(name.lexeme, value);
  }
```

> As `getAt()` is to `get()`, `assignAt()` is to `assign()`. It walks a fixed number of environments, and then stuffs the new value in that map.

正如`getAt()` 与`get()`的关系，`assignAt()` 对应于`assign()`。它会遍历固定数量的环境，然后在其map中塞入新的值。

> Those are the only changes to Interpreter. This is why I chose a representation for our resolved data that was minimally invasive. All of the rest of the nodes continue working as they did before. Even the code for modifying environments is unchanged.

解释器就只需要做这些调整。这也就是为什么我为解析数据选择了一种侵入性最小的表示方法。其余所有节点都跟之前一样，甚至连修改环境的代码也没有改动。

> ### 11.4.3 Running the resolver

11.4.3 运行解析器

> We do need to actually *run* the resolver, though. We insert the new pass after the parser does its magic.

不过，我们确实需要*运行*解析器。我们在解析器完成工作之后插入一次解析器处理。

lox/Lox.java，在 run()方法中添加代码：

```
    // Stop if there was a syntax error.
    if (hadError) return;
    // 新增部分开始
    Resolver resolver = new Resolver(interpreter);
    resolver.resolve(statements);
    // 新增部分结束
    interpreter.interpret(statements);
```

> We don't run the resolver if there are any parse errors. If the code has a syntax error, it's never going to run, so there's little value in resolving it. If the syntax is clean, we tell the resolver to do its thing. The resolver has a reference to the interpreter and pokes the resolution data directly into it as it walks over variables. When the interpreter runs next, it has everything it needs.

如果前面的分析中存在任何错误，我们都不会运行解析器。如果代码有语法错误，它就不会运行，所以解析它的价值不大。如果语法是干净的，我们就告诉解析器做该做的事。解析器中有一个对解释器的引用，当它遍历变量时，会将解析数据直接放入解释器中。解释器后续运行时，它就具备了所需的一切数据。

> At least, that's true if the resolver *succeeds*. But what about errors during resolution?

退一步讲，如果解析器成功了，这么说就是对的。但是如果解析过程中出现错误会怎么办？

> ## 11 . 5 Resolution Errors

11.5 解析错误

> Since we are doing a semantic analysis pass, we have an opportunity to make Lox's semantics more precise, and to help users catch bugs early before running their code. Take a look at this bad boy:

由于我们正在进行语义分析，因此我们有机会使Lox 的语义更加精确，以帮助用户在执行代码之前及早发现错误。看一下下面这个坏代码：

```
fun bad() {
  var a = "first";
  var a = "second";
}
```

> We do allow declaring multiple variables with the same name in the *global* scope, but doing so in a local scope is probably a mistake. If they knew the variable already existed, they would have assigned

> to it instead of using var. And if they *didn't* know it existed, they probably didn't intend to overwrite the previous one.

我们确实允许在*全局*作用域内声明多个同名的变量，但在局部作用域内这样做可能是错误的。如果用户知道变量已经存在，就应该使用赋值操作而不是var。如果他们不知道变量的存在，他们可能并不想覆盖之前的变量。

> We can detect this mistake statically while resolving.

我们可以在解析的时候静态地检测到这个错误。

lox/Resolver.java，在 declare()方法中添加：

```
      Map<String, Boolean> scope = scopes.peek();
      // 新增部分开始
      if (scope.containsKey(name.lexeme)) {
        Lox.error(name,
            "Already variable with this name in this scope.");
      }
      // 新增部分结束
      scope.put(name.lexeme, false);
```

> When we declare a variable in a local scope, we already know the names of every variable previously declared in that same scope. If we see a collision, we report an error.

当我们在局部作用域中声明一个变量时，我们已经知道了之前在同一作用域中声明的每个变量的名字。如果我们看到有冲突，我们就报告一个错误。

> ### 11.5.1 Invalid return errors

11.5.1 无效返回错误

> Here's another nasty little script:

这是另一个讨人厌的小脚本：

```
return "at top level";
```

> This executes a return statement, but it's not even inside a function at all. It's top-level code. I don't know what the user *thinks* is going to happen, but I don't think we want Lox to allow this.

这里执行了一个return语句，但它甚至根本不在函数内部。这是一个顶层代码。我不知道用户认为会发生什么，但是我认为我们不希望Lox允许这种做法。

> We can extend the resolver to detect this statically. Much like we track scopes as we walk the tree, we can track whether or not the code we are currently visiting is inside a function declaration.

我们可以对解析器进行扩展来静态检测这种错误。就像我们遍历语法树时跟踪作用域一样，我们也可以跟踪当前访问的代码是否在一个函数声明内部。

lox/Resolver.java，在 Resolver类中添加代码：

```
  private final Stack<Map<String, Boolean>> scopes = new Stack<>();
  // 新增部分开始
  private FunctionType currentFunction = FunctionType.NONE;
  // 新增部分结束
  Resolver(Interpreter interpreter) {
```

> Instead of a bare Boolean, we use this funny enum:

我们不是使用一个简单的Boolean值，而是使用下面这个有趣的枚举：

lox/Resolver.java，在 Resolver()方法后添加：

```
  private enum FunctionType {
    NONE,
    FUNCTION
  }
```

> It seems kind of dumb now, but we'll add a couple more cases to it later and then it will make more sense. When we resolve a function declaration, we pass that in.

现在看来又点蠢，但是我们稍后会添加更多案例，到时候它将更有意义。当我们解析函数声明时，将其作为参数传入。

lox/Resolver.java，在 visitFunctionStmt()方法中，替换一行：

```
    define(stmt.name);
    // 替换部分开始
    resolveFunction(stmt, FunctionType.FUNCTION);
    // 替换部分结束
    return null;
```

> Over in `resolveFunction()`, we take that parameter and store it in the field before resolving the body.

在`resolveFunction()`中，我们接受该参数，并在解析函数体之前将其保存在字段中。

lox/Resolver.java，在 resolveFunction()方法中替换一行：

```
  // 替换部分开始
    private void resolveFunction(
      Stmt.Function function, FunctionType type) {
    FunctionType enclosingFunction = currentFunction;
    currentFunction = type;
    // 替换部分结束
    beginScope();
```

> We stash the previous value of the field in a local variable first. Remember, Lox has local functions, so you can nest function declarations arbitrarily deeply. We need to track not just that we’re in a function, but *how many* we’re in.

我们先把该字段的旧值存在一个局部变量中。记住，Lox中有局部函数，所以你可以任意深度地嵌套函数声明。我们不仅需要跟踪是否在一个函数内部，还要记录我们在*多少*函数内部。

> We could use an explicit stack of FunctionType values for that, but instead we’ll piggyback on the JVM. We store the previous value in a local on the Java stack. When we’re done resolving the function body, we restore the field to that value.

我们可以使用一个显式的FunctionType值堆栈来进行记录，但我们会借助JVM的力量。我们将前一个值保存在Java堆栈中的一个局部变量。当我们完成函数体的解析之后，我们将该字段恢复为之前的值。

lox/Resolver.java，在 resolveFunction()方法中添加代码：

```
    endScope();
    // 新增部分开始
    currentFunction = enclosingFunction;
    // 新增部分结束
  }
```

> Now that we can always tell whether or not we’re inside a function declaration, we check that when resolving a `return` statement.

既然我们能知道是否在一个函数声明中，那我们就可以在解析`return`语句时进行检查。

lox/Resolver.java，在 visitReturnStmt()方法中添加代码：

```
  public Void visitReturnStmt(Stmt.Return stmt) {
    // 新增部分开始
    if (currentFunction == FunctionType.NONE) {
      Lox.error(stmt.keyword, "Can't return from top-level code.");
    }
    // 新增部分结束
    if (stmt.value != null) {
```

> Neat, right?

很简洁，对吧？

> There’s one more piece. Back in the main Lox class that stitches everything together, we are careful to not run the interpreter if any parse errors are encountered. That check runs *before* the resolver so that we don’t try to resolve syntactically invalid code.

还有一件事。回到将所有部分整合到一起的主类Lox中，我们很小心，如果遇到任何解析错误就不会运行解释器。这个检查是在解析器*之前*运行的，这样我们就不需要再去尝试解析语法无效的代码。

> But we also need to skip the interpreter if there are resolution errors, so we add *another* check.

但是如果在解析变量时存在错误，也需要跳过解释器，所以我们添加*另*一个检查。

lox/Lox.java，在 run()方法中添加代码：

```
    resolver.resolve(statements);
    // 新增部分开始
    // Stop if there was a resolution error.
    if (hadError) return;
    // 新增部分结束
    interpreter.interpret(statements);
```

> You could imagine doing lots of other analysis in here. For example, if we added `break` statements to Lox, we would probably want to ensure they are only used inside loops.

你可以想象在这里做很多其它分析。例如，我们在Lox中添加了`break`语句，而我们可能想确保它只能在循环体中使用。

> We could go farther and report warnings for code that isn't necessarily *wrong* but probably isn't useful. For example, many IDEs will warn if you have unreachable code after a `return` statement, or a local variable whose value is never read. All of that would be pretty easy to add to our static visiting pass, or as separate passes.

我们还可以更进一步，对那些不一定是错误但可能没有用的代码提出警告。举例来说，如果在`return`语句后有不可触及的代码，很多IDE都会发出警告，或者是一个局部变量的值从没有被使用过。所有这些都可以很简单地添加到我们的静态分析过程中，或者作为单独的分析过程^8。

> But, for now, we'll stick with that limited amount of analysis. The important part is that we fixed that one weird annoying edge case bug, though it might be surprising that it took this much work to do it.

但是，就目前而言，我们会坚持这种有限的分析。重要的是，我们修复了一个奇怪又烦人的边界情况bug，尽管花费了这么多精力可能有些令人意外。

^2: 在JavaScript中，使用var声明的变量被隐式提升到块的开头，在代码块中对该名称的任何使用都将指向该变量，即使变量使用出现在声明之前。当你用JavaScript写如下代码时：`{ console.log(a); var a = "value"; }`。它实际相当于：`{ var a; // Hoist. console.log(a); a = "value"; }`。这意味着在某些情况下，您可以在其初始化程序运行之前读取一个变量——一个令人讨厌的错误源。后来添加了用于声明变量的备用`let`语法来解决这个问题。 ^3: 我知道，这完全是一个病态的、人为的程序。这太奇怪了。没有一个理性的人会写这样的代码。唉，如果你长期从事编程语言的工作，你的生活中会有比你想象的更多的时间花在处理这种古怪的代码片段上。 ^4: 一些语言中明确进行了这种分割。在Scheme和ML中，当你用`let`声明一个局部变量时，还描述了新变量在作用域内的后续代码。不存在隐含的 "块的其余部分"。 ^5: 为每个操作复制结构，这听起来可能会浪费大量的内存和时间。在实践中，持久性数据结构在不同的"副本"之间共享大部分的数据。 ^6: 变量解析对每个节点只触及一次，因此其性能是*O(n)*，其中n是语法树中节点的个数。更复杂的分析可能会有更大的复杂性，但是大多数都被精心设计成线性或接近线性。如果编译器随着用户程序的增长而呈指数级变慢，那将是一个很尴尬的失礼。 ^7: 解释器假定变量在map中存在的做法有点像是盲飞。解释器相信解析器完

成了工作并正确地解析了变量。这意味着这两个类之间存在深度耦合。在解析器中，涉及作用域的每一行代码都必须与解释器中修改环境的代码完全匹配。我对这种耦合有切身体会，因为当我在为本书写代码时，我遇到了几个微妙的错误，即解析器代码和解释器代码有点不同步。跟踪这些问题是很困难的。一个行之有效的方法就是，在解释器中使用显式的断言——通过Java的assert或其它验证工具——确认解析器已经具备它所期望的值。 ^8: 要选择将多少个不同的静态分析纳入单个处理过程中是很困难的。许多小的、孤立的过程（每个过程都有自己的职责）实现和维护都比较简单。然而，遍历语法树本身是有实际运行时间成本的，所以将多个分析绑定到一个过程中通常会更快。

CHALLENGES

习题

> 1、Why is it safe to eagerly define the variable bound to a function's name when other variables must wait until after they are initialized before they can be used?

1、为什么先定义与函数名称绑定的变量是安全的，而其它变量必须等到初始化后才能使用？

> 2、How do other languages you know handle local variables that refer to the same name in their initializer, like:

2、你知道其它语言中是如何处理局部变量在初始化式中引用了相同名称变量的情况？比如：

```
var a = "outer";
{
  var a = a;
}
```

> Is it a runtime error? Compile error? Allowed? Do they treat global variables differently? Do you agree with their choices? Justify your answer.

这是一个运行时错误？编译错误？还是允许这种操作？它们对待全局变量的方式有区别吗？你是否认同它们的选择？证明你的答案。

> 3、Extend the resolver to report an error if a local variable is never used.

3、对解析器进行扩展，如果局部变量没有被使用就报告一个错误。

> 4、Our resolver calculates *which* environment the variable is found in, but it's still looked up by name in that map. A more efficient environment representation would store local variables in an array and look them up by index.
>
> Extend the resolver to associate a unique index for each local variable declared in a scope. When resolving a variable access, look up both the scope the variable is in and its index and store that. In the interpreter, use that to quickly access a variable by its index instead of using a map.

4、我们的解析器会计算出变量是在哪个环境中找到的，但是它仍然需要根据名称在对应的map中查找。一个更有效的环境表示形式是将局部变量保存在一个数组中，并通过索引来查找它们。

扩展解析器，为作用域中声明的每个局部变量关联一个唯一的索引。当解析一个变量的访问时，查找变量所在的作用域及对应的索引，并保存起来。在解释器中，使用这个索引快速的访问一个变量。

12.类 Classes

> One has no right to love or hate anything if one has not acquired a thorough knowledge of its nature. Great love springs from great knowledge of the beloved object, and if you know it but little you will be able to love it only a little or not at all.
>
> ——Leonardo da Vinci

如果一个人没有完全了解任何事物的本质，他就没有权利去爱或恨它。伟大的爱来自于对所爱之物的深刻了解，如果你对它知之甚少，你就只能爱一点点，或者根本不爱它。（列奥纳多·达·芬奇）

> We're eleven chapters in, and the interpreter sitting on your machine is nearly a complete scripting language. It could use a couple of built-in data structures like lists and maps, and it certainly needs a core library for file I/O, user input, etc. But the language itself is sufficient. We've got a little procedural language in the same vein as BASIC, Tcl, Scheme (minus macros), and early versions of Python and Lua.

我们已经完成了11章，你机器上的解释器几乎是一个完整的脚本语言实现了。它可以使用一些内置的数据结构，如列表和map，当然还需要一个用于文件IO、用户输入等的核心库。但作为语言本身已经足够了。我们有一个与BASIC、Tcl、Scheme（不包括宏）以及早期版本的Python和Lua相同的小程序语言。

> If this were the '80s, we'd stop here. But today, many popular languages support "object-oriented programming". Adding that to Lox will give users a familiar set of tools for writing larger programs. Even if you personally don't like OOP, this chapter and the next will help you understand how others design and build object systems.

如果现在是80年代，我们就可以到此为止。但是现在，很多流行的语言都支持"面向对象编程"。在Lox中添加该功能，可以为用户提供一套熟悉的工具来编写大型程序。即使你个人不喜欢OOP，这一章和下一章将帮助你理解别人是如何设计和构建对象系统的^1。

12.1 OOP and Classes

> There are three broad paths to object-oriented programming: classes, prototypes, and multimethods. Classes came first and are the most popular style. With the rise of JavaScript (and to a lesser extent Lua), prototypes are more widely known than they used to be. I'll talk more about those later. For Lox, we're taking the, ahem, classic approach.

面向对象编程有三大途径：类、原型和多方法^2。类排在第一位，是最流行的风格。随着JavaScript（其次是Lua）的兴起，原型也比以前更加广为人知。稍后我们会更多地讨论这些问题。对于Lox，我们采取的是经典的方法。

> Since you've written about a thousand lines of Java code with me already, I'm assuming you don't need a detailed introduction to object orientation. The main goal is to bundle data with the code that acts on it. Users do that by declaring a *class* that:

既然你已经跟我一起编写了大约1000行Java代码，我假设你不需要对面向对象进行详细介绍。OOP的主要目标就是将数据与作用于数据的代码捆绑在一起。用户通过声明一个类来实现这一点：

1. > Exposes a *constructor* to create and initialize new *instances* of the class

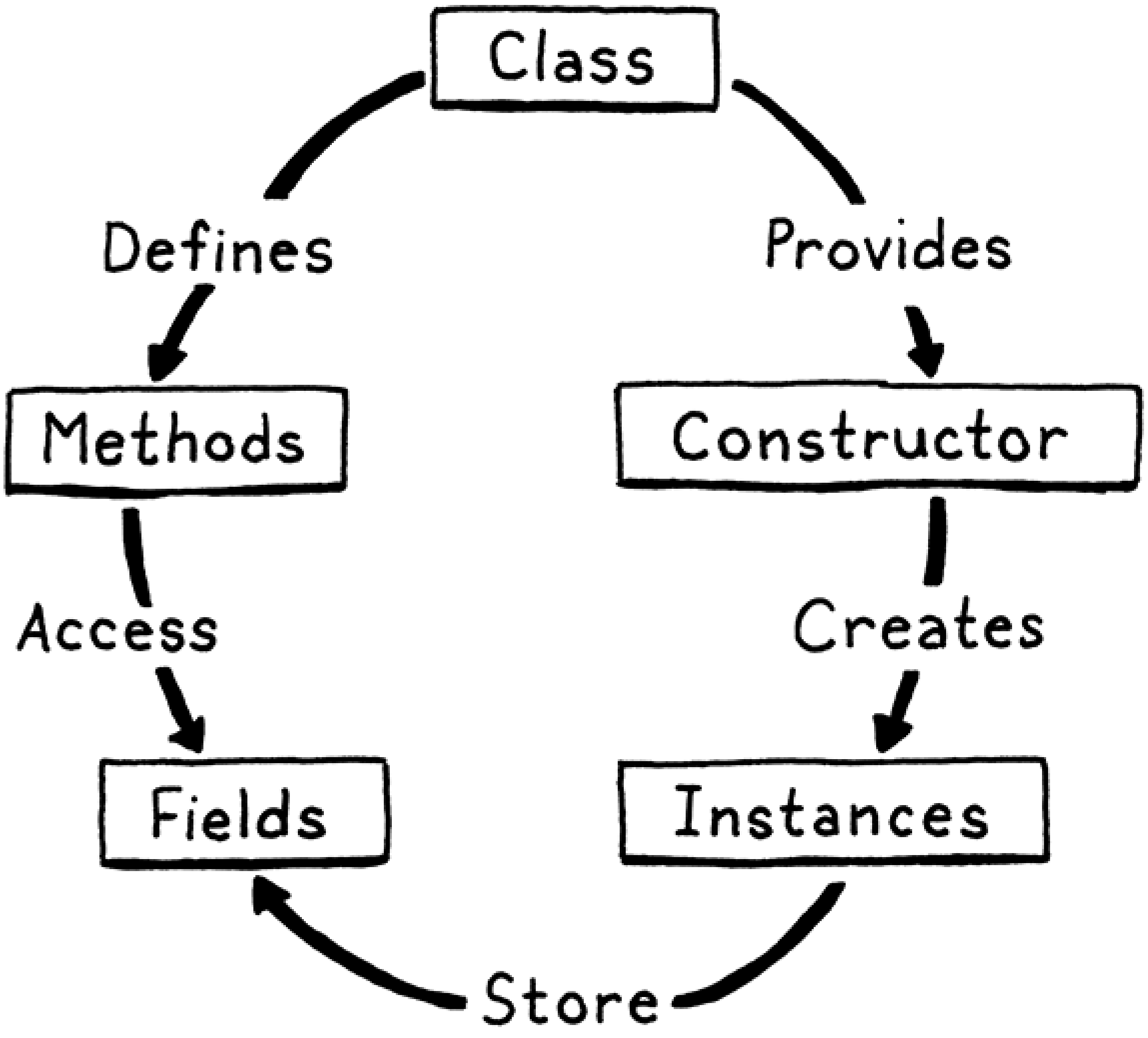
暴露 *构造函数*以创建和初始化该类的新实例

2. > Provides a way to store and access *fields* on instances

   提供在实例上存储和访问 *字段*的方法。

3. > Defines a set of *methods* shared by all instances of the class that operate on each instances' state.

   定义一组由类的所有实例共享的 *方法*，这些方法对各个实例的状态进行操作。



> That's about as minimal as it gets. Most object-oriented languages, all the way back to Simula, also do inheritance to reuse behavior across classes. We'll add that in the next chapter. Even kicking that out, we still have a lot to get through. This is a big chapter and everything doesn't quite come together until we have all of the above pieces, so gather your stamina.

这大概是最低要求。大多数面向对象的语言（一直追溯到Simula），也都是通过继承来跨类重用行为。我们会在下一章中添加该功能。即使剔除了这些，我们仍然有很多东西需要完成。这是一个很大的章节，直到我们完成上述所有内容之后，才能把所有东西整合到一起。所以请集中精力。

12.2 Class Declarations

> Like we do, we're gonna start with syntax. A `class` statement introduces a new name, so it lives in the `declaration` grammar rule.

跟之前一样，我们从语法开始。`class`语句引入了一个新名称，所以它应该在`declaration` 语法规则中。

```
declaration    → classDecl
               | funDecl
               | varDecl
               | statement ;

classDecl      → "class" IDENTIFIER "{" function* "}" ;
```

> The new `classDecl` rule relies on the `function` rule we defined earlier. To refresh your memory:

新的`classDecl`规则依赖于前面定义的`function`规则。复习一下：

```
function       → IDENTIFIER "(" parameters? ")" block ;
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;
```

> In plain English, a class declaration is the `class` keyword, followed by the class's name, then a curly-braced body. Inside that body is a list of method declarations. Unlike function declarations, methods don't have a leading `fun` keyword. Each method is a name, parameter list, and body. Here's an example:

用简单的英语来说，类声明就是`class`关键字，后跟类的名称，然后是一对花括号包含的主体。在这个主体中，有一个方法声明的列表。与函数声明不同的是，方法没有前导的`fun`关键字。每个方法就是一个名称、参数列表和方法主体。下面是一个例子：

```
class Breakfast {
  cook() {
    print "Eggs a-fryin'!";
  }

  serve(who) {
    print "Enjoy your breakfast, " + who + ".";
  }
}
```

> Like most dynamically typed languages, fields are not explicitly listed in the class declaration. Instances are loose bags of data and you can freely add fields to them as you see fit using normal imperative code.

像大多数动态类型的语言一样，字段没有在类的声明中明确列出。实例是松散的数据包，你可以使用正常的命令式代码自由地向其中添加字段。

> Over in our AST generator, the `classDecl` grammar rule gets its own statement node.

在AST生成器中，`classDecl`语法规则有自己的语句节点。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Block      : List<Stmt> statements",
      // 新增部分开始
      "Class      : Token name, List<Stmt.Function> methods",
      // 新增部分结束
      "Expression : Expr expression",
```

> It stores the class's name and the methods inside its body. Methods are represented by the existing Stmt.Function class that we use for function declaration AST nodes. That gives us all the bits of state that we need for a method: name, parameter list, and body.

它存储了类的名称和其主体内的方法。方法使用现有的表示函数声明的Stmt.Function类来表示。这就为我们提供了一个方法所需的所有状态：名称、参数列表和方法体。

> A class can appear anywhere a named declaration is allowed, triggered by the leading `class` keyword.

类可以出现在任何允许名称声明的地方，由前导的`class`关键字来触发。

lox/Parser.java，在 declaration()方法中添加：

```
    try {
      // 新增部分开始
      if (match(CLASS)) return classDeclaration();
      // 新增部分结束
      if (match(FUN)) return function("function");
```

> That calls out to:

进一步调用：

lox/Parser.java，在 declaration()方法后添加：

```
  private Stmt classDeclaration() {
    Token name = consume(IDENTIFIER, "Expect class name.");
    consume(LEFT_BRACE, "Expect '{' before class body.");

    List<Stmt.Function> methods = new ArrayList<>();
    while (!check(RIGHT_BRACE) && !isAtEnd()) {
      methods.add(function("method"));
    }

    consume(RIGHT_BRACE, "Expect '}' after class body.");

    return new Stmt.Class(name, methods);
  }
```

> There's more meat to this than most of the other parsing methods, but it roughly follows the grammar. We've already consumed the `class` keyword, so we look for the expected class name next, followed by the opening curly brace. Once inside the body, we keep parsing method declarations until we hit the closing brace. Each method declaration is parsed by a call to `function()`, which we defined back in the chapter where functions were introduced.

这比其它大多数解析方法有更多的内容，但它大致上遵循了语法。我们已经使用了`class`关键字，所以我们接下来会查找预期的类名，然后是左花括号。一旦进入主体，我们就继续解析方法声明，直到碰到右花括号。每个方法声明是通过调用`function()`方法来解析的，我们在介绍函数的那一章中定义了该函数。

> Like we do in any open-ended loop in the parser, we also check for hitting the end of the file. That won't happen in correct code since a class should have a closing brace at the end, but it ensures the parser doesn't get stuck in an infinite loop if the user has a syntax error and forgets to correctly end the class body.

就像我们在解析器中的所有开放式循环中的操作一样，我们也要检查是否到达文件结尾。这在正确的代码是不会发生的，因为类的结尾应该有一个右花括号，但它可以确保在用户出现语法错误而忘记正确结束类的主体时，解析器不会陷入无限循环。

> We wrap the name and list of methods into a Stmt.Class node and we're done. Previously, we would jump straight into the interpreter, but now we need to plumb the node through the resolver first.

我们将名称和方法列表封装到Stmt.Class节点中，这样就完成了。以前，我们会直接进入解释器中，但是现在我们需要先进入分析器中对节点进行分析。【译者注：为了区分parse和resolve，这里将resolver称为分析器，用于对代码中的变量进行分析】

lox/Resolver.java，在 visitBlockStmt()方法后添加：

```
  @Override
  public Void visitClassStmt(Stmt.Class stmt) {
    declare(stmt.name);
    define(stmt.name);
    return null;
  }
```

> We aren't going to worry about resolving the methods themselves yet, so for now all we need to do is declare the class using its name. It's not common to declare a class as a local variable, but Lox permits it, so we need to handle it correctly.

我们还不用担心针对方法本身的分析，我们目前需要做的是使用类的名称来声明这个类。将类声明为一个局部变量并不常见，但是Lox中允许这样做，所以我们需要正确处理。

> Now we interpret the class declaration.

现在我们解释一下类的声明。

lox/Interpreter.java，在 visitBlockStmt()方法后添加：

```
  @Override
  public Void visitClassStmt(Stmt.Class stmt) {
    environment.define(stmt.name.lexeme, null);
    LoxClass klass = new LoxClass(stmt.name.lexeme);
    environment.assign(stmt.name, klass);
    return null;
  }
```

> This looks similar to how we execute function declarations. We declare the class's name in the current environment. Then we turn the class *syntax node* into a LoxClass, the *runtime* representation of a class. We circle back and store the class object in the variable we previously declared. That two-stage variable binding process allows references to the class inside its own methods.

这看起来类似于我们执行函数声明的方式。我们在当前环境中声明该类的名称。然后我们把类的*语法节点*转换为LoxClass，即类的*运行时*表示。我们回过头来，将类对象存储在我们之前声明的变量中。这个二阶段的变量绑定过程允许在类的方法中引用其自身。

> We will refine it throughout the chapter, but the first draft of LoxClass looks like this:

我们会在整个章节中对其进行完善，但是LoxClass的初稿看起来如下：

lox/LoxClass.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;
import java.util.Map;

class LoxClass {
  final String name;

  LoxClass(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
}
```

> Literally a wrapper around a name. We don't even store the methods yet. Not super useful, but it does have a `toString()` method so we can write a trivial script and test that class objects are actually being parsed and executed.

字面上看，就是一个对name的包装。我们甚至还没有保存类中的方法。不算很有用，但是它确实有一个`toString()`方法，所以我们可以编写一个简单的脚本，测试类对象是否真的被解析和执行。

```
class DevonshireCream {
  serveOn() {
    return "Scones";
  }
}

print DevonshireCream; // Prints "DevonshireCream".
```

12.3 Creating Instances

12.3 创建实例

> We have classes, but they don't do anything yet. Lox doesn't have "static" methods that you can call right on the class itself, so without actual instances, classes are useless. Thus instances are the next step.

我们有了类，但是它们还不能做任何事。Lox没有可以直接在类本身调用的"静态"方法，所以如果没有实例，类是没有用的。因此，下一步就是实例化。

> While some syntax and semantics are fairly standard across OOP languages, the way you create new instances isn't. Ruby, following Smalltalk, creates instances by calling a method on the class object itself, a recursively graceful approach. Some, like C++ and Java, have a `new` keyword dedicated to birthing a new object. Python has you "call" the class itself like a function. (JavaScript, ever weird, sort of does both.)

虽然一些语法和语义在OOP语言中是相当标准的，但创建新实例的方式并不是。Ruby，继Smalltalk之后，通过调用类对象本身的一个方法来创建实例，这是一种递归的优雅方法^3。有些语言，像C++和Java，有一个`new`关键字专门用来创建一个新的对象。Python让你像调用函数一样"调用"类本身。(JavaScript，永远都是那么奇怪，两者兼而有之)

> I took a minimal approach with Lox. We already have class objects, and we already have function calls, so we'll use call expressions on class objects to create new instances. It's as if a class is a factory function that generates instances of itself. This feels elegant to me, and also spares us the need to introduce syntax like `new`. Therefore, we can skip past the front end straight into the runtime.

我在Lox中采用了一种最简单的方法。我们已经有了类对象，也有了函数调用，所以我们直接使用类对象的调用表达式来创建新的实例。这就好像类是一个生产自身实例的工厂函数。这让我感觉很优雅，也不需要引入`new`这样的语法。因此，我们可以跳过前端直接进入运行时。

> Right now, if you try this:

现在，如果你试着运行下面的代码：

```
class Bagel {}
Bagel();
```

> You get a runtime error. `visitCallExpr()` checks to see if the called object implements `LoxCallable` and reports an error since LoxClass doesn't. Not *yet*, that is.

你会得到一个运行时错误。`visitCallExpr()`方法会检查被调用的对象是否实现了`LoxCallable` 接口，因为LoxClass没有实现所以会报错。只是目前还没有。

lox/LoxClass.java，替换一行：

```
import java.util.Map;
// 替换部分开始
class LoxClass implements LoxCallable {
// 替换部分结束
  final String name;
```

> Implementing that interface requires two methods.

实现该接口需要两个方法。

lox/LoxClass.java，在 toString()方法后添加:

```
  @Override
  public Object call(Interpreter interpreter,
                     List<Object> arguments) {
    LoxInstance instance = new LoxInstance(this);
    return instance;
  }

  @Override
  public int arity() {
    return 0;
  }
```

> The interesting one is `call()`. When you "call" a class, it instantiates a new LoxInstance for the called class and returns it. The `arity()` method is how the interpreter validates that you passed the right number of arguments to a callable. For now, we'll say you can't pass any. When we get to user-defined constructors, we'll revisit this.

有趣的是`call()`。当你"调用"一个类时，它会为被调用的类实例化一个新的LoxInstance并返回。`arity()` 方法是解释器用于验证你是否向callable中传入了正确数量的参数。现在，我们会说你不用传任何参数。当我们讨论用户自定义的构造函数时，我们再重新考虑这个问题。

> That leads us to LoxInstance, the runtime representation of an instance of a Lox class. Again, our first implementation starts small.

这就引出了LoxInstance，它是Lox类实例的运行时表示。同样，我们的第一个实现从小处着手。

lox/LoxInstance.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.HashMap;
import java.util.Map;

class LoxInstance {
  private LoxClass klass;

  LoxInstance(LoxClass klass) {
    this.klass = klass;
  }

  @Override
  public String toString() {
    return klass.name + " instance";
  }
}
```

> Like LoxClass, it's pretty bare bones, but we're only getting started. If you want to give it a try, here's a script to run:

和LoxClass一样，它也是相当简陋的，但我们才刚刚开始。如果你想测试一下，可以运行下面的脚本：

```
class Bagel {}
var bagel = Bagel();
print bagel; // Prints "Bagel instance".
```

> This program doesn't do much, but it's starting to do *something*.

这段程序没有做太多事，但是已经开始做*一些事情了*。

12.4 Properties on Instances

12.4 实例属性

> We have instances, so we should make them useful. We're at a fork in the road. We could add behavior first—methods—or we could start with state—properties. We're going to take the latter because, as we'll see, the two get entangled in an interesting way and it will be easier to make sense of them if we get properties working first.

我们有了实例，所以我们应该让它们发挥作用。我们正处于一个岔路口。我们可以首先添加行为（方法），或者我们可以先从状态（属性）开始。我们将选择后者，因为我们后面将会看到，这两者以一种有趣的方式纠缠在一起，如果我们先支持属性，就会更容易理解它们。

> Lox follows JavaScript and Python in how it handles state. Every instance is an open collection of named values. Methods on the instance's class can access and modify properties, but so can outside code. Properties are accessed using a `.` syntax.

Lox遵循了JavaScript和Python处理状态的方式。每个实例都是一个开放的命名值集合。实例类中的方法可以访问和修改属性，但外部代码也可以^4。属性通过.语法进行访问。

```
someObject.someProperty
```

> An expression followed by `.` and an identifier reads the property with that name from the object the expression evaluates to. That dot has the same precedence as the parentheses in a function call expression, so we slot it into the grammar by replacing the existing `call` rule with:

一个后面跟着.和一个标识符的表达式，会从表达式计算出的对象中读取该名称对应的属性。这个点符号与函数调用表达式中的括号具有相同的优先级，所以我们要将该符号加入语法时，可以替换已有的`call`规则如下：

```
call           → primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
```

> After a primary expression, we allow a series of any mixture of parenthesized calls and dotted property accesses. “Property access” is a mouthful, so from here on out, we’ll call these “get expressions”.

在基本表达式之后，我们允许跟一系列括号调用和点属性访问的任何混合。属性访问有点拗口，所以自此以后，我们称其为“get表达式”。

> ### 12.4.1 Get expressions

12.4.1 Get表达式

> The syntax tree node is:

语法树节点是：

tool/GenerateAst.java，在 main()方法中添加：

```
      "Call     : Expr callee, Token paren, List<Expr> arguments",
      // 新增部分开始
      "Get      : Expr object, Token name",
      // 新增部分结束
      "Grouping : Expr expression",
```

> Following the grammar, the new parsing code goes in our existing `call()` method.

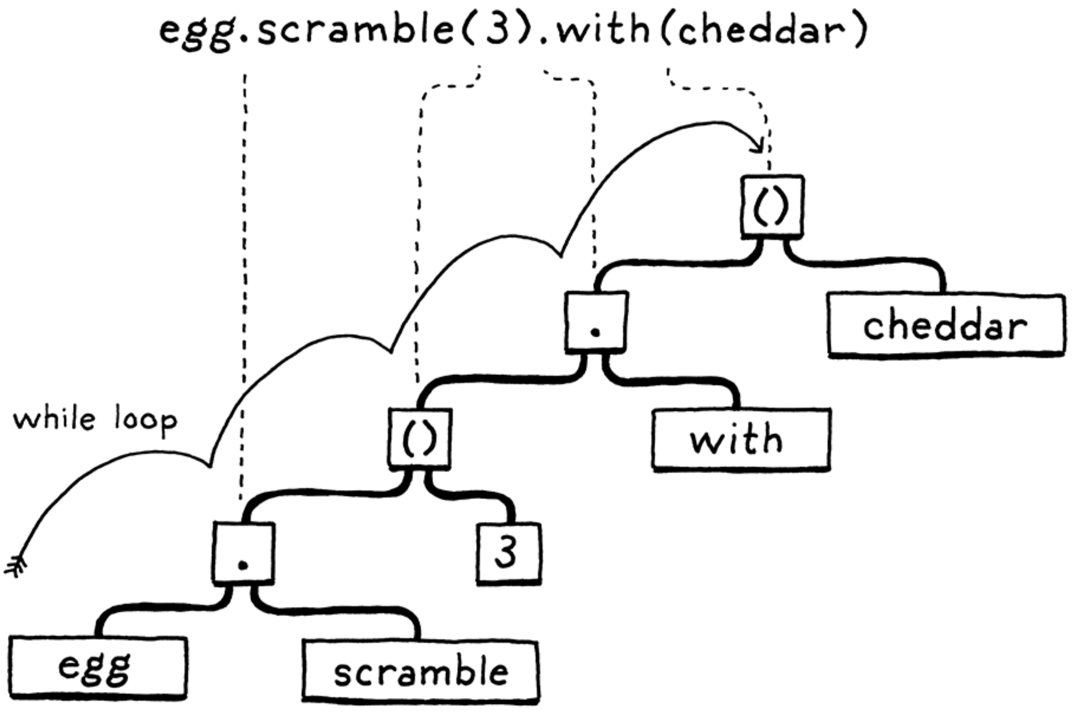
按照语法，在现有的`call()`方法中加入新的解析代码。

lox/Parser.java，在 call()方法中添加代码：

```
    while (true) {
      if (match(LEFT_PAREN)) {
        expr = finishCall(expr);
      // 新增部分开始
      } else if (match(DOT)) {
        Token name = consume(IDENTIFIER,
            "Expect property name after '.'.");
        expr = new Expr.Get(expr, name);
      // 新增部分结束
      } else {
        break;
      }
    }
```

> The outer while loop there corresponds to the * in the grammar rule. We zip along the tokens building up a chain of calls and gets as we find parentheses and dots, like so:

外面的while循环对应于语法规则中的*。随着查找括号和点，我们会沿着标记构建一系列的call和get，就像：



> Instances of the new Expr.Get node feed into the resolver.

新的Expr.Get节点实例会被送入分析器。

lox/Resolver.java，在visitCallExpr()方法后添加：

```
  @Override
  public Void visitGetExpr(Expr.Get expr) {
    resolve(expr.object);
    return null;
  }
```

> OK, not much to that. Since properties are looked up dynamically, they don't get resolved. During resolution, we recurse only into the expression to the left of the dot. The actual property access happens in the interpreter.

好吧，没什么好说的。因为属性是动态查找的，所以不会解析它们。在解析过程中，我们只递归到点符左边的表达式中。实际的属性访问发生在解释器中。

lox/Interpreter.java，在 visitCallExpr()方法后添加：

```
@Override
public Object visitGetExpr(Expr.Get expr) {
  Object object = evaluate(expr.object);
  if (object instanceof LoxInstance) {
    return ((LoxInstance) object).get(expr.name);
  }

  throw new RuntimeError(expr.name,
      "Only instances have properties.");
}
```

> First, we evaluate the expression whose property is being accessed. In Lox, only instances of classes have properties. If the object is some other type like a number, invoking a getter on it is a runtime error.

首先，我们对属性被访问的表达式求值。在Lox中，只有类的实例才具有属性。如果对象是其它类型（如数字），则对其执行getter是运行时错误。

> If the object is a LoxInstance, then we ask it to look up the property. It must be time to give LoxInstance some actual state. A map will do fine.

如果该对象是LoxInstance，我们就要求它去查找该属性。现在必须给LoxInstance一些实际的状态了。一个map就行了。

lox/LoxInstance.java，在 LoxInstance类中添加：

```
private LoxClass klass;
// 新增部分开始
private final Map<String, Object> fields = new HashMap<>();
// 新增部分结束
LoxInstance(LoxClass klass) {
```

> Each key in the map is a property name and the corresponding value is the property's value. To look up a property on an instance:

map中的每个键是一个属性名称，对应的值就是该属性的值。查找实例中的一个属性：

lox/LoxInstance.java，在 LoxInstance()方法后添加：

```
Object get(Token name) {
  if (fields.containsKey(name.lexeme)) {
    return fields.get(name.lexeme);
  }
```

```
    throw new RuntimeError(name,
        "Undefined property '" + name.lexeme + "'.");
  }
```

> An interesting edge case we need to handle is what happens if the instance doesn't *have* a property with the given name. We could silently return some dummy value like `nil`, but my experience with languages like JavaScript is that this behavior masks bugs more often than it does anything useful. Instead, we'll make it a runtime error.

我们需要处理的一个有趣的边缘情况是，如果这个实例中*不包含*给定名称的属性，会发生什么。我们可以悄悄返回一些假值，如`nil`，但是根据我对JavaScript等语言的经验，这种行为只是掩盖了错误，而没有做任何有用的事。相反，我们将它作为一个运行时错误。

> So the first thing we do is see if the instance actually has a field with the given name. Only then do we return it. Otherwise, we raise an error.

因此，我们首先要做的就是看看这个实例中是否真的包含给定名称的字段。只有这样，我们才会返回其值。其它情况下，我们会引发一个错误。

> Note how I switched from talking about "properties" to "fields". There is a subtle difference between the two. Fields are named bits of state stored directly in an instance. Properties are the named, uh, *things*, that a get expression may return. Every field is a property, but as we'll see later, not every property is a field.

注意我是如何从讨论"属性"转换到讨论"字段"的。这两者之间有一个微妙的区别。字段是直接保存在实例中的命名状态。属性是get表达式可能返回的已命名的*东西*。每个字段都是一个属性，但是正如我们稍后将看到的，并非每个属性都是一个字段。

> In theory, we can now read properties on objects. But since there's no way to actually stuff any state into an instance, there are no fields to access. Before we can test out reading, we must support writing.

理论上，我们现在可以读取对象的属性。但是由于没有办法将任何状态真正填充到实例中，所以也没有字段可以访问。在我们测试读取之前，我们需要先支持写入。

> ### 12.4.2 Set expressions

12.4.2 Set表达式

> Setters use the same syntax as getters, except they appear on the left side of an assignment.

setter和getter使用相同的语法，区别只是它们出现在赋值表达式的左侧。

```
someObject.someProperty = value;
```

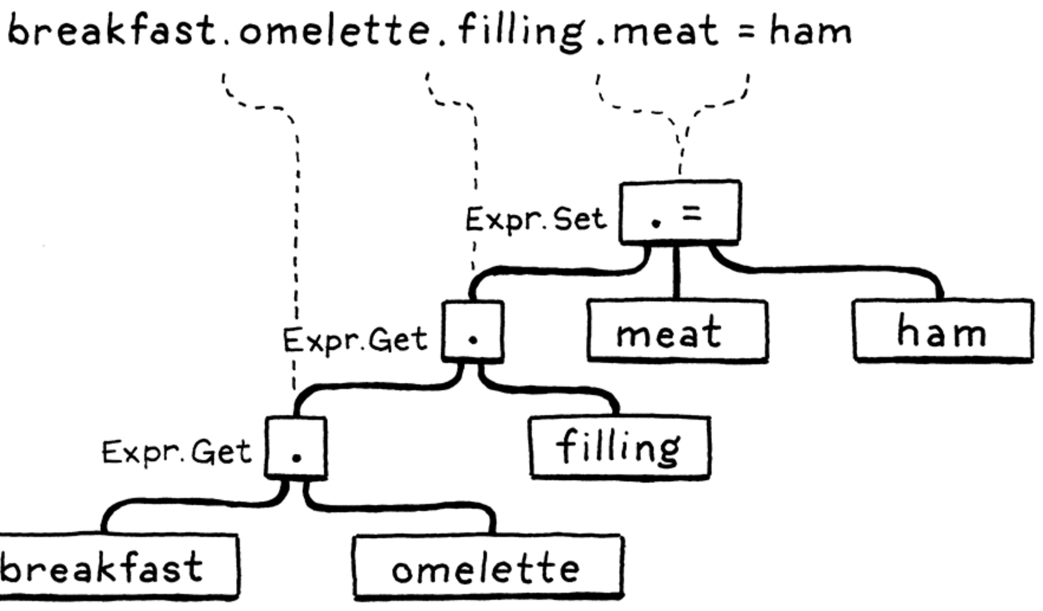
> In grammar land, we extend the rule for assignment to allow dotted identifiers on the left-hand side.

在语言方面，我们扩展了赋值规则，允许在左侧使用点标识符。

```
assignment     → ( call "." )? IDENTIFIER "=" assignment
               | logic_or ;
```

> Unlike getters, setters don't chain. However, the reference to `call` allows any high-precedence expression before the last dot, including any number of *getters*, as in:

与getter不同，setter不使用链。但是，对`call` 规则的引用允许在最后的点符号之前出现任何高优先级的表达式，包括任何数量的*getters*，如：



> Note here that only the *last* part, the `.meat` is the *setter*. The `.omelette` and `.filling` parts are both *get* expressions.

注意，这里只有最后一部分`.meat`是*setter*。`.omelette`和`.filling`部分都是*get*表达式。

> Just as we have two separate AST nodes for variable access and variable assignment, we need a second setter node to complement our getter node.

就像我们有两个独立的AST节点用于变量访问和变量赋值一样，我们也需要一个setter节点来补充getter节点。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Logical  : Expr left, Token operator, Expr right",
      // 新增部分开始
      "Set      : Expr object, Token name, Expr value",
      // 新增部分结束
      "Unary    : Token operator, Expr right",
```

> In case you don't remember, the way we handle assignment in the parser is a little funny. We can't easily tell that a series of tokens is the left-hand side of an assignment until we reach the `=`. Now that our assignment grammar rule has `call` on the left side, which can expand to arbitrarily large expressions, that final `=` may be many tokens away from the point where we need to know we're parsing an assignment.

也许你不记得了，我们在解析器中处理赋值的方法有点奇怪。在遇到=之前，我们无法轻易判断一系列标记是否是一个赋值表达式的左侧部分。现在我们的赋值语法规则在左侧添加了`call`，它可以扩展为任意大的表达式，最后的=可能与我们需要知道是否正在解析赋值表达式的地方隔着很多标记。

> Instead, the trick we do is parse the left-hand side as a normal expression. Then, when we stumble onto the equal sign after it, we take the expression we already parsed and transform it into the correct syntax tree node for the assignment.

相对地，我们的技巧就是把左边的表达式作为一个正常表达式来解析。然后，当我们在后面发现等号时，我们就把已经解析的表达式转换为正确的赋值语法树节点。

> We add another clause to that transformation to handle turning an Expr.Get expression on the left into the corresponding Expr.Set.

我们在该转换中添加另一个子句，将左边的Expr.Get表达式转化为相应的Expr.Set表达式。

lox/Parser.java，在 assignment()方法中添加：

```
          return new Expr.Assign(name, value);
        // 新增部分开始
        } else if (expr instanceof Expr.Get) {
          Expr.Get get = (Expr.Get)expr;
          return new Expr.Set(get.object, get.name, value);
        // 新增部分结束
        }
```

> That's parsing our syntax. We push that node through into the resolver.

这就是语法解析。我们将该节点推入分析器中。

lox/Resolver.java，在 visitLogicalExpr()方法后添加：

```
  @Override
  public Void visitSetExpr(Expr.Set expr) {
    resolve(expr.value);
    resolve(expr.object);
    return null;
  }
```

> Again, like Expr.Get, the property itself is dynamically evaluated, so there's nothing to resolve there. All we need to do is recurse into the two subexpressions of Expr.Set, the object whose property is being set, and the value it's being set to.

同样，像Expr.Get一样，属性本身是动态计算的，所以没有什么需要分析的。我们只需要递归到Expr.Set的两个子表达式中，即被设置属性的对象和它被设置的值。

> That leads us to the interpreter.

这又会把我们引向解释器。

lox/Interpreter.java，在 visitLogicalExpr()方法后添加：

```
  @Override
  public Object visitSetExpr(Expr.Set expr) {
    Object object = evaluate(expr.object);

    if (!(object instanceof LoxInstance)) {
      throw new RuntimeError(expr.name,
                             "Only instances have fields.");
    }

    Object value = evaluate(expr.value);
    ((LoxInstance)object).set(expr.name, value);
    return value;
  }
```

> We evaluate the object whose property is being set and check to see if it's a LoxInstance. If not, that's a runtime error. Otherwise, we evaluate the value being set and store it on the instance. That relies on a new method in LoxInstance.

我们先计算出被设置属性的对象，然后检查它是否是一个LoxInstance。如果不是，这就是一个运行时错误。否则，我们计算设置的值，并将其保存到该实例中。这一步依赖于LoxInstance中的一个新方法。

lox/LoxInstance.java，在 get()方法后添加：

```
  void set(Token name, Object value) {
    fields.put(name.lexeme, value);
  }
```

> No real magic here. We stuff the values straight into the Java map where fields live. Since Lox allows freely creating new fields on instances, there's no need to see if the key is already present.

这里没什么复杂的。我们把这些值之间塞入字段所在的Java map中。由于Lox允许在实例上自由创建新字段，所以不需要检查键是否已经存在。

> ## 12.5 Methods on Classes

12.5 类中的方法

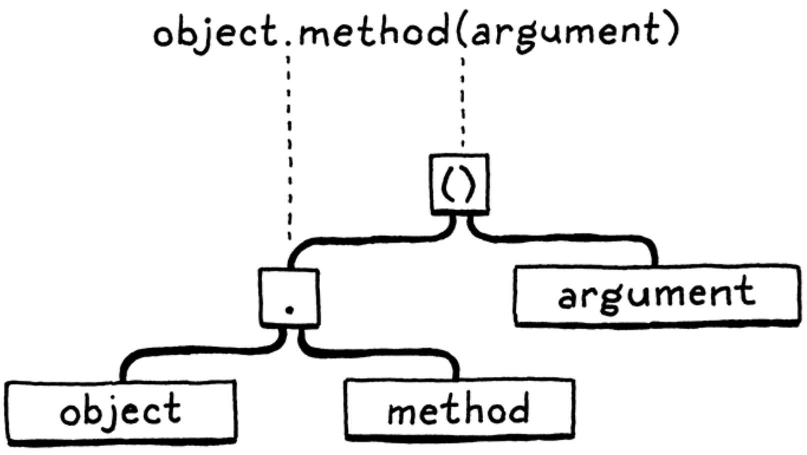
> You can create instances of classes and stuff data into them, but the class itself doesn't really *do* anything. Instances are just maps and all instances are more or less the same. To make them feel like instances *of classes*, we need behavior—methods.

你可以创建类的实例并将数据填入其中，但是类本身实际上并不能做任何事。实例只是一个map，而且所有的实例都是大同小异的。为了让它们更像是*类*的实例，我们需要行为——方法。

> Our helpful parser already parses method declarations, so we're good there. We also don't need to add any new parser support for method *calls*. We already have `.` (getters) and `()` (function calls). A

> “method call” simply chains those together.

我们的解析器已经解析了方法声明，所以我们在这部分做的不错。我们也不需要为方法*调用*添加任何新的解析器支持。我们已经有了`.`(getter)和`()`(函数调用)。“方法调用”只是简单地将这些串在一起。



> That raises an interesting question. What happens when those two expressions are pulled apart? Assuming that `method` in this example is a method on the class of `object` and not a field on the instance, what should the following piece of code do?

这引出了一个有趣的问题。当这两个表达式分开时会发生什么？假设这个例子中的方法`method`是`object`的类中的一个方法，而不是实例中的 一个字段，下面的代码应该做什么？

```
var m = object.method;
m(argument);
```

> This program “looks up” the method and stores the result—whatever that is—in a variable and then calls that object later. Is this allowed? Can you treat a method like it’s a function on the instance?

这个程序会“查找”该方法，并将结果（不管是什么）存储到一个变量中，稍后会调用该对象。允许这样吗？你能将方法作为实例中的一个函数来对待吗？

> What about the other direction?

另一个方向呢？

```
class Box {}

fun notMethod(argument) {
  print "called function with " + argument;
}

var box = Box();
box.function = notMethod;
box.function("argument");
```

> This program creates an instance and then stores a function in a field on it. Then it calls that function using the same syntax as a method call. Does that work?

这个程序创建了一个实例，然后在它的一个字段中存储了一个函数。然后使用与方法调用相同的语法来调用该函数。这样做有用吗？

> Different languages have different answers to these questions. One could write a treatise on it. For Lox, we'll say the answer to both of these is yes, it does work. We have a couple of reasons to justify that. For the second example—calling a function stored in a field—we want to support that because first-class functions are useful and storing them in fields is a perfectly normal thing to do.

不同的语言对这些问题有不同的答案。人们可以就此写一篇论文。对于Lox来说，这两个问题的答案都是肯定的，它确实有效。我们有几个理由来证明这一点。对于第二个例子——调用存储在字段中的函数——我们想要支持它，是因为头等函数是有用的，而且将它们存储在字段中是一件很正常的事情。

> The first example is more obscure. One motivation is that users generally expect to be able to hoist a subexpression out into a local variable without changing the meaning of the program. You can take this:

第一个例子就比较晦涩了。一个场景是，用户通常希望能够在不改变程序含义的情况下，将子表达式赋值到一个局部变量中。你可以这样做：

```
breakfast(omelette.filledWith(cheese), sausage);
```

> And turn it into this:

并将其变成这样：

```
var eggs = omelette.filledWith(cheese);
breakfast(eggs, sausage);
```

> And it does the same thing. Likewise, since the `.` and the `()` in a method call *are* two separate expressions, it seems you should be able to hoist the *lookup* part into a variable and then call it later. We need to think carefully about what the *thing* you get when you look up a method is, and how it behaves, even in weird cases like:

它做的是同样的事情。同样，由于方法调用中的`.`和`()`是两个独立的表达式，你似乎应该把查询部分提取到一个变量中，然后再调用它^5。我们需要仔细思考，当你查找一个方法时你得到的东西是什么，它如何作用，甚至是在一些奇怪的情况下，比如：

```
class Person {
  sayName() {
    print this.name;
  }
}

var jane = Person();
```

```
jane.name = "Jane";

var method = jane.sayName;
method(); // ?
```

> If you grab a handle to a method on some instance and call it later, does it “remember” the instance it was pulled off from? Does `this` inside the method still refer to that original object?

如果你在某个实例上获取了一个方法的句柄，并在稍后再调用它，它是否能“记住”它是从哪个实例中提取出来的？方法内部的`this`是否仍然指向原始的那个对象？

> Here’s a more pathological example to bend your brain:

下面有一个更变态的例子，可以摧毁你的大脑：

```
class Person {
  sayName() {
    print this.name;
  }
}

var jane = Person();
jane.name = "Jane";

var bill = Person();
bill.name = "Bill";

bill.sayName = jane.sayName;
bill.sayName(); // ?
```

> Does that last line print “Bill” because that’s the instance that we *called* the method through, or “Jane” because it’s the instance where we first grabbed the method?

最后一行会因为*调用*方法的实体是bill而打印“Bill”，还是因为我们第一次获取方法的实例是jane而打印“Jane”。

> Equivalent code in Lua and JavaScript would print “Bill”. Those languages don’t really have a notion of “methods”. Everything is sort of functions-in-fields, so it’s not clear that `jane` “owns” `sayName` any more than `bill` does.

在Lua和JavaScript中，同样的代码会打印 "Bill"。这些语言并没有真正的“方法”的概念。所有东西都类似于字段中的函数，所以并不清楚`jane` 是否更应该比`bill`“拥有”`sayName`。

> Lox, though, has real class syntax so we do know which callable things are methods and which are functions. Thus, like Python, C#, and others, we will have methods “bind” `this` to the original instance when the method is first grabbed. Python calls these **bound methods**.

不过，Lox有真正的类语法，所以我们确实知道哪些可调用的东西是方法，哪些是函数。因此，像Python、C#和其他语言一样，当方法第一次被获取时，我们会让方法与原始实例`this`进行 "绑定"。Python将这些绑定的方法称为**bound methods**（绑定方法）。

> In practice, that's usually what you want. If you take a reference to a method on some object so you can use it as a callback later, you want to remember the instance it belonged to, even if that callback happens to be stored in a field on some other object.

在实践中，这通常也是你想要的。如果你获取到了某个对象中一个方法的引用，这样你以后就可以把它作为一个回调函数使用，你想要记住它所属的实例，即使这个回调被存储在其它对象的字段中。

> OK, that's a lot of semantics to load into your head. Forget about the edge cases for a bit. We'll get back to those. For now, let's get basic method calls working. We're already parsing the method declarations inside the class body, so the next step is to resolve them.

好吧，这里有很多语义需要装到你的脑子里。暂时先不考虑那些边缘情况了，我们以后再讲。现在，让我们先把基本的方法调用做好。我们已经解析了类主体内的方法声明，所以下一步就是对其分析。

lox/Resolver.java，在 visitClassStmt()方法内添加^6：

```
    define(stmt.name);
    // 新增部分开始
    for (Stmt.Function method : stmt.methods) {
      FunctionType declaration = FunctionType.METHOD;
      resolveFunction(method, declaration);
    }
    // 新增部分结束
    return null;
```

> We iterate through the methods in the class body and call the `resolveFunction()` method we wrote for handling function declarations already. The only difference is that we pass in a new FunctionType enum value.

我们遍历类主体中的方法，并调用我们已经写好的用来处理函数声明的`resolveFunction()`方法。唯一的区别在于，我们传入了一个新的FunctionType枚举值。

lox/Resolver.java，在 FunctionType 枚举中添加代码，在上一行末尾添加`,`：

```
    NONE,
    FUNCTION,
    // 新增部分开始
    METHOD
    // 新增部分结束
  }
```

> That's going to be important when we resolve `this` expressions. For now, don't worry about it. The interesting stuff is in the interpreter.

这一点在我们分析`this`表达式时很重要。现在还不用担心这个问题。有趣的部分在解释器中。

lox/Interpreter.java，在 visitClassStmt()方法中替换一行：

```
    environment.define(stmt.name.lexeme, null);
    // 替换部分开始
    Map<String, LoxFunction> methods = new HashMap<>();
    for (Stmt.Function method : stmt.methods) {
      LoxFunction function = new LoxFunction(method, environment);
      methods.put(method.name.lexeme, function);
    }

    LoxClass klass = new LoxClass(stmt.name.lexeme, methods);
    // 替换部分结束
    environment.assign(stmt.name, klass);
```

> When we interpret a class declaration statement, we turn the syntactic representation of the class—its AST node—into its runtime representation. Now, we need to do that for the methods contained in the class as well. Each method declaration blossoms into a LoxFunction object.

当我们解释一个类声明语句时，我们把类的语法表示（其AST节点）变成它的运行时表示。现在，我们也需要对类中包含的方法进行这样的操作。每个方法声明都会变成一个LoxFunction对象。

> We take all of those and wrap them up into a map, keyed by the method names. That gets stored in LoxClass.

我们把所有这些都打包到一个map中，以方法名称作为键。这些数据存储在LoxClass中。

lox/LoxClass.java，在类 LoxClass 中，替换4行：

```
  final String name;
  // 替换部分开始
  private final Map<String, LoxFunction> methods;

  LoxClass(String name, Map<String, LoxFunction> methods) {
    this.name = name;
    this.methods = methods;
  }
  // 替换部分结束
  @Override
  public String toString() {
```

> Where an instance stores state, the class stores behavior. LoxInstance has its map of fields, and LoxClass gets a map of methods. Even though methods are owned by the class, they are still accessed through instances of that class.

实例存储状态，类存储行为。LoxInstance包含字段的map，而LoxClass包含方法的map。虽然方法是归类所有，但仍然是通过类的实例来访问。

lox/LoxInstance.java，在 get()方法中添加：

```
  Object get(Token name) {
    if (fields.containsKey(name.lexeme)) {
      return fields.get(name.lexeme);
    }
    // 新增部分开始
    LoxFunction method = klass.findMethod(name.lexeme);
    if (method != null) return method;
    // 新增部分结束
    throw new RuntimeError(name,
        "Undefined property '" + name.lexeme + "'.");
```

> When looking up a property on an instance, if we don't find a matching field, we look for a method with that name on the instance's class. If found, we return that. This is where the distinction between "field" and "property" becomes meaningful. When accessing a property, you might get a field—a bit of state stored on the instance—or you could hit a method defined on the instance's class.

在实例上查找属性时，如果我们没有找到匹配的字段，我们就在实例的类中查找是否包含该名称的方法。如果找到，我们就返回该方法^7。这就是"字段"和"属性"之间的区别变得有意义的地方。当访问一个属性时，你可能会得到一个字段（存储在实例上的状态值），或者你会得到一个实例类中定义的方法。

> The method is looked up using this:

方法是通过下面的代码进行查找的：

lox/LoxClass.java，在 LoxClass()方法后添加：

```
  LoxFunction findMethod(String name) {
    if (methods.containsKey(name)) {
      return methods.get(name);
    }

    return null;
  }
```

> You can probably guess this method is going to get more interesting later. For now, a simple map lookup on the class's method table is enough to get us started. Give it a try:

你大概能猜到这个方法后面会变得更有趣。但是现在，在类的方法表中进行简单的映射查询就足够了。试一下：

```
class Bacon {
  eat() {
    print "Crunch crunch crunch!";
  }
}

Bacon().eat(); // Prints "Crunch crunch crunch!".
```

12.6 This

> We can define both behavior and state on objects, but they aren't tied together yet. Inside a method, we have no way to access the fields of the "current" object—the instance that the method was called on—nor can we call other methods on that same object.

我们可以在对象上定义行为和状态，但是它们并没有被绑定在一起。在一个方法中，我们没有办法访问"当前"对象（调用该方法的实例）的字段，也不能调用同一个对象的其它方法。

> To get at that instance, it needs a name. Smalltalk, Ruby, and Swift use "self". Simula, C++, Java, and others use "this". Python uses "self" by convention, but you can technically call it whatever you like.

为了获得这个实例，它需要一个名称。Smalltalk、Ruby和Swift使用 "self"。Simula、C++、Java等使用 "this"。Python按惯例使用 "self"，但从技术上讲，你可以随便叫它什么。

> For Lox, since we generally hew to Java-ish style, we'll go with "this". Inside a method body, a `this` expression evaluates to the instance that the method was called on. Or, more specifically, since methods are accessed and then invoked as two steps, it will refer to the object that the method was *accessed* from.

对于Lox来说，因为我们通常遵循Java风格，我们会使用"this"。在方法体中，`this`表达式计算结果为调用该方法的实例。或者，更确切地说，由于方法是分为两个步骤进行访问和调用的，因此它会引用调用方法的对象。

> That makes our job harder. Peep at:

这使得我们的工作更加困难。请看：

```
class Egotist {
  speak() {
    print this;
  }
}

var method = Egotist().speak;
method();
```

> On the second-to-last line, we grab a reference to the `speak()` method off an instance of the class. That returns a function, and that function needs to remember the instance it was pulled off of so that *later*, on the last line, it can still find it when the function is called.

在倒数第二行，我们从该类的一个实例中获取到了指向`speak()` 的引用。这个操作会返回一个函数，并且该函数需要记住它来自哪个实例，这样稍后在最后一行，当函数被调用时，它仍然可用找到对应实例。

> We need to take `this` at the point that the method is accessed and attach it to the function somehow so that it stays around as long as we need it to. Hmm . . . a way to store some extra data that hangs around a function, eh? That sounds an awful lot like a *closure*, doesn't it?

我们需要在方法被访问时获取到`this`，并将其附到函数上，这样当我们需要的时候它就一直存在。嗯...一种存储函数周围的额外数据的方法，嗯？听起来很像一个闭包，不是吗？

> If we defined this as a sort of hidden variable in an environment that surrounds the function returned when looking up a method, then uses of this in the body would be able to find it later. LoxFunction already has the ability to hold on to a surrounding environment, so we have the machinery we need.

如果我们把this定义为在查找方法时返回的函数外围环境中的一个隐藏变量，那么稍后在方法主体中使用this时就可以找到它了。LoxFunction已经具备了保持外围环境的能力，所以我们已经有了需要的机制。

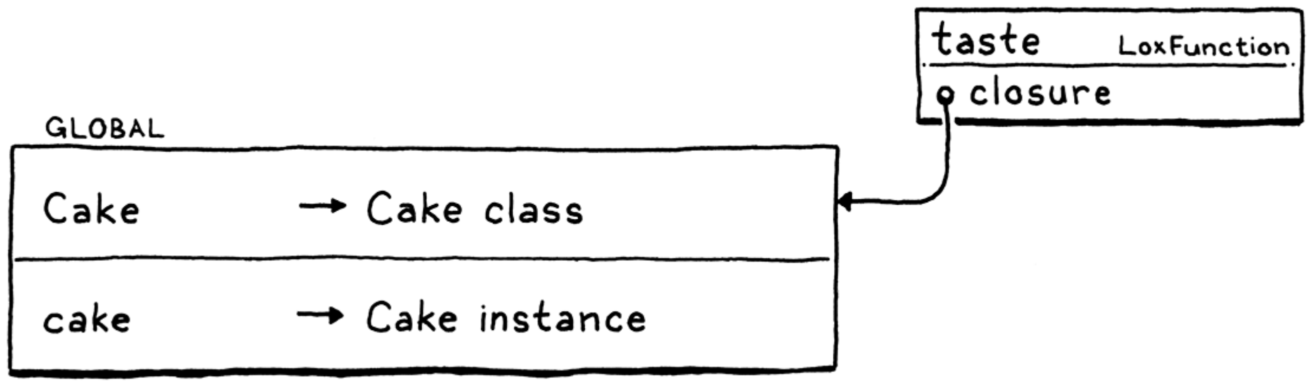
> Let's walk through an example to see how it works:

我们通过一个例子来看看它是如何工作的：

```
class Cake {
  taste() {
    var adjective = "delicious";
    print "The " + this.flavor + " cake is " + adjective + "!";
  }
}

var cake = Cake();
cake.flavor = "German chocolate";
cake.taste(); // Prints "The German chocolate cake is delicious!".
```

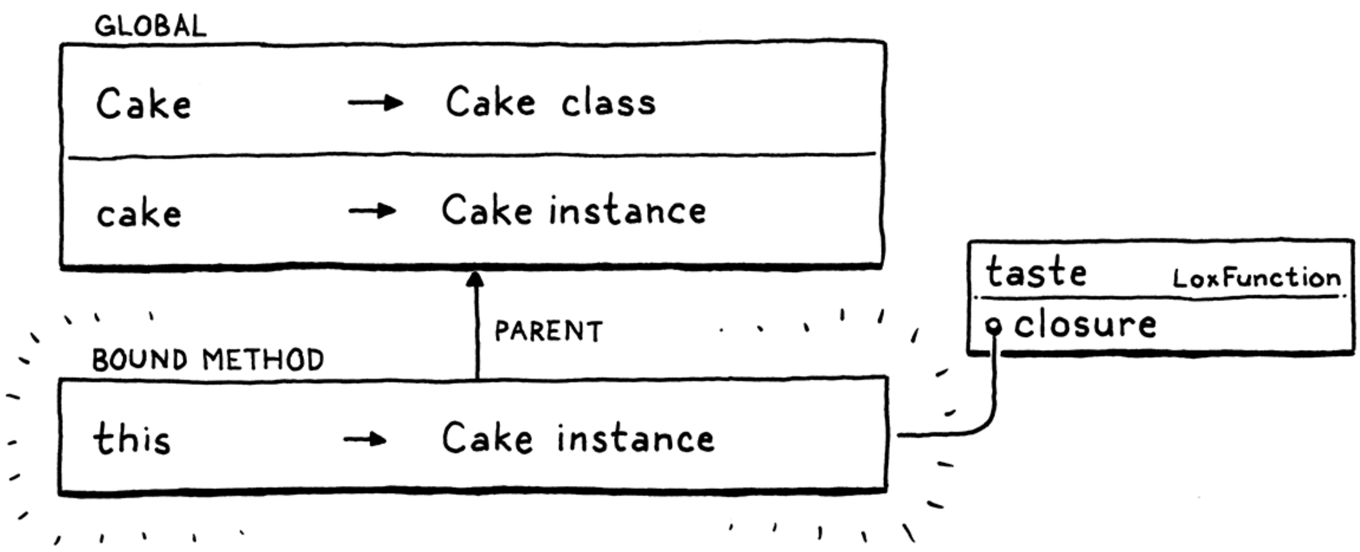
> When we first evaluate the class definition, we create a LoxFunction for taste(). Its closure is the environment surrounding the class, in this case the global one. So the LoxFunction we store in the class's method map looks like so:

当我们第一次执行类定义时，我们为taste()创建了一个LoxFunction。它的闭包是类外围的环境，在这个例子中就是全局环境。所以我们在类的方法map中保存的LoxFunction看起来像是这样的：



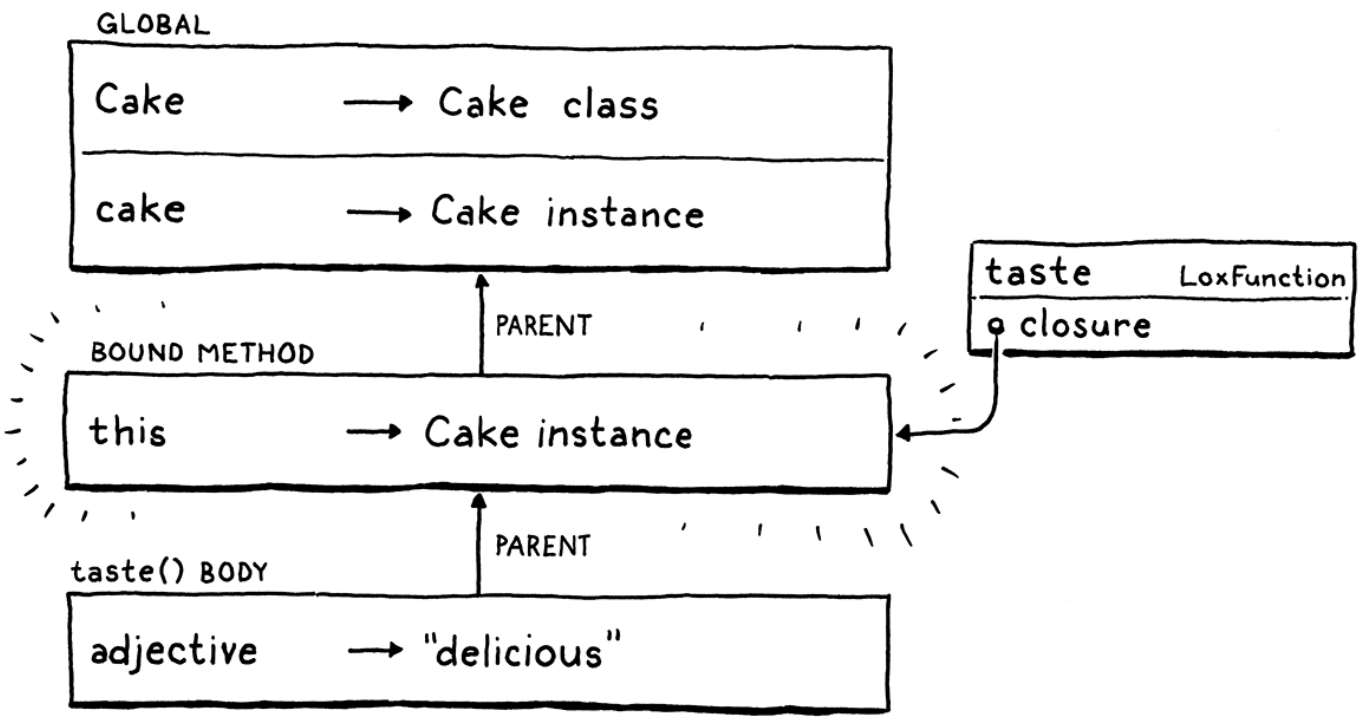
> When we evaluate the cake.taste get expression, we create a new environment that binds this to the object the method is accessed from (here, cake). Then we make a *new* LoxFunction with the same code as the original one but using that new environment as its closure.

当我们执行cake.taste这个get表达式时，我们会创建一个新的环境，其中将this绑定到了访问该方法的对象（这里是cake）。然后我们创建一个*新*的LoxFunction，它的代码与原始的代码相同，但是使用新环境作为其闭包。



> This is the LoxFunction that gets returned when evaluating the get expression for the method name. When that function is later called by a `()` expression, we create an environment for the method body as usual.

这个是在执行方法名的get表达式时返回的LoxFunction。当这个函数稍后被一个`()`表达式调用时，我们像往常一样为方法主体创建一个环境。



> The parent of the body environment is the environment we created earlier to bind `this` to the current object. Thus any use of `this` inside the body successfully resolves to that instance.

主体环境的父环境，也就是我们先前创建并在其中将`this`绑定到当前对象的那个环境。因此，在函数主体内使用`this`都可以成功解析到那个实例。

> Reusing our environment code for implementing `this` also takes care of interesting cases where methods and functions interact, like:

重用环境代码来实现`this`时，也需要注意方法和函数交互的情况，比如：

```
class Thing {
  getCallback() {
    fun localFunction() {
      print this;
    }

    return localFunction;
  }
}

var callback = Thing().getCallback();
callback();
```

> In, say, JavaScript, it's common to return a callback from inside a method. That callback may want to hang on to and retain access to the original object—the `this` value—that the method was associated with. Our existing support for closures and environment chains should do all this correctly.

例如，在JavaScript中，在一个方法中返回一个回调函数是很常见的。这个回调函数可能希望保留对方法所关联的原对象（`this`值）的访问。我们现有的对闭包和环境链的支持应该可以正确地做到这一点。

> Let's code it up. The first step is adding new syntax for `this`.

让我们把它写出来。第一步是为`this`添加新的语法。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Set      : Expr object, Token name, Expr value",
      // 新增部分开始
      "This     : Token keyword",
      // 新增部分结束
      "Unary    : Token operator, Expr right",
```

> Parsing is simple since it's a single token which our lexer already recognizes as a reserved word.

解析很简单，因为它是已经被词法解析器当作关键字识别出来的单个词法标记。

lox/Parser.java，在 primary()方法中添加：

```
      return new Expr.Literal(previous().literal);
    }
    // 新增部分开始
    if (match(THIS)) return new Expr.This(previous());
    // 新增部分结束
    if (match(IDENTIFIER)) {
```

> You can start to see how `this` works like a variable when we get to the resolver.

当进入分析器后，就可以看到 `this` 是如何像变量一样工作的。

lox/Resolver.java，在 visitSetExpr()方法后添加：

```
  @Override
  public Void visitThisExpr(Expr.This expr) {
    resolveLocal(expr, expr.keyword);
    return null;
  }
```

> We resolve it exactly like any other local variable using "this" as the name for the "variable". Of course, that's not going to work right now, because "this" *isn't* declared in any scope. Let's fix that over in `visitClassStmt()`.

我们使用`this`作为"变量"的名称，并像其它局部变量一样对其分析。当然，现在这是行不通的，因为"this"没有在任何作用域进行声明。我们在`visitClassStmt()`方法中解决这个问题。

lox/Resolver.java，在visitClassStmt()方法中添加：

```
    define(stmt.name);
    // 新增部分开始
    beginScope();
    scopes.peek().put("this", true);
    // 新增部分结束
    for (Stmt.Function method : stmt.methods) {
```

> Before we step in and start resolving the method bodies, we push a new scope and define "this" in it as if it were a variable. Then, when we're done, we discard that surrounding scope.

在我们开始分析方法体之前，我们推入一个新的作用域，并在其中像定义变量一样定义"this"。然后，当我们完成后，会丢弃这个外围作用域。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
    }
    // 新增部分开始
    endScope();
    // 新增部分结束
    return null;
```

> Now, whenever a `this` expression is encountered (at least inside a method) it will resolve to a "local variable" defined in an implicit scope just outside of the block for the method body.

现在，只要遇到`this`表达式（至少是在方法内部），它就会解析为一个"局部变量"，该变量定义在方法体块之外的隐含作用域中。

> **The resolver has a new *scope* for `this`, so the interpreter needs to create a corresponding *environment* for it. Remember, we always have to keep the resolver's scope chains and the**

> **interpreter's linked environments in sync with each other. At runtime, we create the environment after we find the method on the instance. We replace the previous line of code that simply returned the meth**od's LoxFunction with this:

分析器对`this`有一个新的*作用域*，所以解释器需要为它创建一个对应的*环境*。记住，我们必须始终保持分析器的作用域链与解释器的链式环境保持同步。在运行时，我们在找到实例上的方法后创建环境。我们把之前那行直接返回方法对应LoxFunction的代码替换如下：

lox/LoxInstance.java，在 get()方法中替换一行：

```
    LoxFunction method = klass.findMethod(name.lexeme);
    // 替换部分开始
    if (method != null) return method.bind(this);
    // 替换部分结束
    throw new RuntimeError(name,
        "Undefined property '" + name.lexeme + "'.");
```

> Note the new call to `bind()`. That looks like so:

注意这里对`bind()`的新调用。该方法看起来是这样的：

lox/LoxFunction.java，在 LoxFunction()方法后添加：

```
  LoxFunction bind(LoxInstance instance) {
    Environment environment = new Environment(closure);
    environment.define("this", instance);
    return new LoxFunction(declaration, environment);
  }
```

> There isn't much to it. We create a new environment nestled inside the method's original closure. Sort of a closure-within-a-closure. When the method is called, that will become the parent of the method body's environment.

这没什么好说的。我们基于方法的原始闭包创建了一个新的环境。就像是闭包内的闭包。当方法被调用时，它将变成方法体对应环境的父环境。

> We declare "this" as a variable in that environment and bind it to the given instance, the instance that the method is being accessed from. *Et voilà*, the returned LoxFunction now carries around its own little persistent world where "this" is bound to the object.

我们将`this`声明为该环境中的一个变量，并将其绑定到给定的实例（即方法被访问时的实例）上。就是这样，现在返回的LoxFunction带着它自己的小持久化世界，其中的"this"被绑定到对象上。

> The remaining task is interpreting those `this` expressions. Similar to the resolver, it is the same as interpreting a variable expression.

剩下的任务就是解释那些`this`表达式。与分析器类似，与解释变量表达式是一样的。

lox/Interpreter.java，在 visitSetExpr()方法后添加：

```
  @Override
  public Object visitThisExpr(Expr.This expr) {
    return lookUpVariable(expr.keyword, expr);
  }
```

> Go ahead and give it a try using that cake example from earlier. With less than twenty lines of code, our interpreter handles `this` inside methods even in all of the weird ways it can interact with nested classes, functions inside methods, handles to methods, etc.

来吧，用前面那个蛋糕的例子试一试。通过添加不到20行代码，我们的解释器就能处理方法内部的`this`，甚至能以各种奇怪的方式与嵌套类、方法内部的函数、方法句柄等进行交互。

> 12.6.1 Invalid uses of this

12.6.1 this的无效使用

> Wait a minute. What happens if you try to use `this` *outside* of a method? What about:

等一下，如果你尝试在方法之外使用`this`会怎么样？比如：

```
print this;
```

> Or:

或者：

```
fun notAMethod() {
  print this;
}
```

> There is no instance for `this` to point to if you're not in a method. We could give it some default value like `nil` or make it a runtime error, but the user has clearly made a mistake. The sooner they find and fix that mistake, the happier they'll be.

如果你不在一个方法中，就没有可供`this`指向的实例。我们可以给它一些默认值如`nil`或者抛出一个运行时错误，但是用户显然犯了一个错误。他们越早发现并纠正这个错误，就会越高兴。

> Our resolution pass is a fine place to detect this error statically. It already detects `return` statements outside of functions. We'll do something similar for `this`. In the vein of our existing FunctionType enum, we define a new ClassType one.

我们的分析过程是一个静态检测这个错误的好地方。它已经检测了函数之外的`return`语句。我们可以针对`this`做一些类似的事情。在我们现有的FunctionType枚举的基础上，我们定义一个新的ClassType枚举。

lox/Resolver.java，在 FunctionType枚举后添加：

```
  }
  // 新增部分开始
  private enum ClassType {
    NONE,
    CLASS
  }

  private ClassType currentClass = ClassType.NONE;
  // 新增部分结束
  void resolve(List<Stmt> statements) {
```

> Yes, it could be a Boolean. When we get to inheritance, it will get a third value, hence the enum right now. We also add a corresponding field, `currentClass`. Its value tells us if we are currently inside a class declaration while traversing the syntax tree. It starts out `NONE` which means we aren't in one.

是的，它可以是一个布尔值。当我们谈到继承时，它会扩展第三个值，因此使用了枚举。我们还添加了一个相应的字段`currentClass`。它的值告诉我们，在遍历语法树时，我们目前是否在一个类声明中。它一开始是`NONE`，意味着我们不在类中。

> When we begin to resolve a class declaration, we change that.

当我们开始分析一个类声明时，我们会改变它。

<u>lox/Resolver.java，在 visitClassStmt()方法中添加：</u>

```
  public Void visitClassStmt(Stmt.Class stmt) {
    // 新增部分开始
    ClassType enclosingClass = currentClass;
    currentClass = ClassType.CLASS;
    // 新增部分结束
    declare(stmt.name);
```

> As with `currentFunction`, we store the previous value of the field in a local variable. This lets us piggyback onto the JVM to keep a stack of `currentClass` values. That way we don't lose track of the previous value if one class nests inside another.

与`currentFunction`一样，我们将字段的前一个值存储在一个局部变量中。这样我们可以在JVM中保持一个`currentClass`的栈。如果一个类嵌套在另一个类中，我们就不会丢失对前一个值的跟踪。

> Once the methods have been resolved, we "pop" that stack by restoring the old value.

一旦这么方法完成了分析，我们通过恢复旧值来"弹出"堆栈。

<u>lox/Resolver.java, 在 visitClassStmt()方法中添加：</u>

```
    endScope();
    // 新增部分开始
    currentClass = enclosingClass;
    // 新增部分结束
    return null;
```

> When we resolve a `this` expression, the `currentClass` field gives us the bit of data we need to report an error if the expression doesn't occur nestled inside a method body.

当我们解析`this`表达式时，如果表达式没有出现在一个方法体内，`currentClass`就为我们提供了报告错误所需的数据。

lox/Resolver.java，在 visitThisExpr()方法中添加：

```
  public Void visitThisExpr(Expr.This expr) {
    // 新增部分开始
    if (currentClass == ClassType.NONE) {
      Lox.error(expr.keyword,
          "Can't use 'this' outside of a class.");
      return null;
    }
    // 新增部分结束
    resolveLocal(expr, expr.keyword);
```

> That should help users use `this` correctly, and it saves us from having to handle misuse at runtime in the interpreter.

这应该能帮助用户正确地使用`this`，并且它使我们不必在解释器运行时中处理这个误用问题。

12.7 Constructors and Initializers

12.7 构造函数和初始化

> We can do almost everything with classes now, and as we near the end of the chapter we find ourselves strangely focused on a beginning. Methods and fields let us encapsulate state and behavior together so that an object always *stays* in a valid configuration. But how do we ensure a brand new object *starts* in a good state?

我们现在几乎可以用类来做任何事情，而当我们接近本章结尾时，却发现自己奇怪地专注于开头。方法和字段让我们把状态和行为封装在一起，这样一个对象就能始终保持在有效的配置状态。但我们如何确保一个全新的对象是以良好的状态开始的？

> For that, we need constructors. I find them one of the trickiest parts of a language to design, and if you peer closely at most other languages, you'll see cracks around object construction where the seams of the design don't quite fit together perfectly. Maybe there's something intrinsically messy about the moment of birth.

为此，我们需要构造函数。我发现它们是语言设计中最棘手的部分之一，如果你仔细观察大多数其它语言，就会发现围绕着对象构造的缺陷，设计的接缝并不完全吻合^8。也许在一开始就存在本质上的混乱。

> "Constructing" an object is actually a pair of operations:

“构造”一个对象实际上是一对操作：

1. > The runtime *allocates* the memory required for a fresh instance. In most languages, this operation is at a fundamental level beneath what user code is able to access.

   运行时为一个新的实例*分配*所需的内存。在多数语言中，这个操作是在用户代码可以访问的层面之下的基础层完成的^9。

2. > Then, a user-provided chunk of code is called which *initializes* the unformed object.

   然后，用户提供的一大块代码被调用，以初始化未成形的对象。

> The latter is what we tend to think of when we hear “constructor”, but the language itself has usually done some groundwork for us before we get to that point. In fact, our Lox interpreter already has that covered when it creates a new LoxInstance object.

当我们听到“构造函数”时，我们往往会想到后者，但语言本身在此之前通常已经为我们做了一些基础工作。事实上，我们的Lox解释器在创建一个新的LoxInstance对象时已经涵盖了这一点。

> We’ll do the remaining part—user-defined initialization—now. Languages have a variety of notations for the chunk of code that sets up a new object for a class. C++, Java, and C# use a method whose name matches the class name. Ruby and Python call it `init()`. The latter is nice and short, so we’ll do that.

我们现在要做的是剩下的部分——用户自定义的初始化。对于为类建立新对象的这块代码，不同的语言有不同的说法。C++、Java和C#使用一个名字与类名相匹配的方法。Ruby 和 Python 称之为 `init()`。后者又好又简短，所以我们采用它。

> In LoxClass’s implementation of LoxCallable, we add a few more lines.

在LoxClass的LoxCallable实现中，我们再增加几行。

lox/LoxClass.java,在 call()方法中添加：

```java
                     List<Object> arguments) {
    LoxInstance instance = new LoxInstance(this);
    // 新增部分开始
    LoxFunction initializer = findMethod("init");
    if (initializer != null) {
      initializer.bind(instance).call(interpreter, arguments);
    }
    // 新增部分结束
    return instance;
```

> When a class is called, after the LoxInstance is created, we look for an “init” method. If we find one, we immediately bind and invoke it just like a normal method call. The argument list is forwarded along.

当一个类被调用时，在LoxInstance被创建后，我们会寻找一个 "init "方法。如果我们找到了，我们就会立即绑定并调用它，就像普通的方法调用一样。参数列表直接透传。

> That argument list means we also need to tweak how a class declares its arity.

这个参数列表意味着我们也需要调整类声明其元数的方式。

```
  public int arity() {
```

lox/LoxClass.java,在 arity()方法中替换一行:

```
  public int arity() {
    // 替换部分开始
    LoxFunction initializer = findMethod("init");
    if (initializer == null) return 0;
    return initializer.arity();
    // 替换部分结束
  }
```

> If there is an initializer, that method's arity determines how many arguments you must pass when you call the class itself. We don't *require* a class to define an initializer, though, as a convenience. If you don't have an initializer, the arity is still zero.

如果有初始化方法,该方法的元数就决定了在调用类本身的时候需要传入多少个参数。但是,为了方便起见,我们并不要求类定义初始化方法。如果你没有初始化方法,元数仍然是0。

> That's basically it. Since we bind the `init()` method before we call it, it has access to `this` inside its body. That, along with the arguments passed to the class, are all you need to be able to set up the new instance however you desire.

基本上就是这样了。因为我们在调用`init()`方法之前已经将其绑定,所以它可以在方法体内访问`this`。这样,连同传递给类的参数,你就可以按照自己的意愿设置新实例了。

> ### 12.7.1 Invoking init() directly

12.7.1 直接执行init()

> As usual, exploring this new semantic territory rustles up a few weird creatures. Consider:

像往常一样,探索这一新的语义领域会催生出一些奇怪的事物。考虑一下:

```
class Foo {
  init() {
    print this;
  }
}

var foo = Foo();
print foo.init();
```

> Can you “re-initialize” an object by directly calling its `init()` method? If you do, what does it return? A reasonable answer would be `nil` since that’s what it appears the body returns.

你能否通过直接调用对象的`init()`方法对其进行“重新初始化”？如果可以，它的返回值是什么？一个合理的答案应该是`nil`，因为这是方法主体返回的内容。

> However—and I generally dislike compromising to satisfy the implementation—it will make clox’s implementation of constructors much easier if we say that `init()` methods always return `this`, even when directly called. In order to keep jlox compatible with that, we add a little special case code in LoxFunction.

然而，我通常不喜欢为满足实现而妥协^10，如果我们让`init()`方法总是返回`this`（即使是被直接调用时），它会使clox中的构造函数实现更加简单。为了保持jlox与之兼容，我们在LoxFunction中添加了一些针对特殊情况的代码。

lox/LoxFunction.java，在 call()方法中添加：

```
      return returnValue.value;
    }
    // 新增部分开始
    if (isInitializer) return closure.getAt(0, "this");
    // 新增部分结束
    return null;
```

> If the function is an initializer, we override the actual return value and forcibly return `this`. That relies on a new `isInitializer` field.

如果该函数是一个初始化方法，我们会覆盖实际的返回值并强行返回`this`。这个操作依赖于一个新的`isInitializer`字段。

lox/LoxFunction.java，在 LoxFunction类中，替换一行：

```
  private final Environment closure;
  // 替换部分开始
  private final boolean isInitializer;

  LoxFunction(Stmt.Function declaration, Environment closure, boolean
isInitializer) {
    this.isInitializer = isInitializer;
    // 替换部分结束
    this.closure = closure;
    this.declaration = declaration;
```

> We can’t simply see if the name of the LoxFunction is “init” because the user could have defined a *function* with that name. In that case, there *is* no `this` to return. To avoid *that* weird edge case, we’ll directly store whether the LoxFunction represents an initializer method. That means we need to go back and fix the few places where we create LoxFunctions.

我们不能简单地检查LoxFunction的名字是否为"init"，因为用户可能已经定义了一个同名的*函数*。在这种情况下，是没有this可供返回的。为了避免这种奇怪的边缘情况，我们将直接存储LoxFunction是否表示一个初始化方法。这意味着我们需要回头修正我们创建LoxFunctions的几个地方。

lox/Interpreter.java，在 visitFunctionStmt()方法中，替换一行：

```
  public Void visitFunctionStmt(Stmt.Function stmt) {
    // 替换部分开始
    LoxFunction function = new LoxFunction(stmt, environment, false);
    // 替换部分结束
    environment.define(stmt.name.lexeme, function);
```

> For actual function declarations, `isInitializer` is always false. For methods, we check the name.

对于实际的函数声明， `isInitializer`取值总是false。对于方法来说，我们检查其名称。

lox/Interpreter.java，在 visitClassStmt()方法中，替换一行：

```
    for (Stmt.Function method : stmt.methods) {
      // 替换部分开始
      LoxFunction function = new LoxFunction(method, environment,
          method.name.lexeme.equals("init"));
      // 替换部分结束
      methods.put(method.name.lexeme, function);
```

> And then in `bind()` where we create the closure that binds `this` to a method, we pass along the original method's value.

然后在`bind()`方法，在创建闭包并将`this`绑定到新方法时，我们将原始方法的值传递给新方法。

lox/LoxFunction.java，在 bind()方法中，替换一行：

```
    environment.define("this", instance);
    // 替换部分开始
    return new LoxFunction(declaration, environment,
                           isInitializer);
    // 替换部分结束
  }
```

> ### 12.7.2 Returning from init()

12.7.2 从init()返回

> We aren't out of the woods yet. We've been assuming that a user-written initializer doesn't explicitly return a value because most constructors don't. What should happen if a user tries:

我们还没有走出困境。我们一直假设用户编写的初始化方法不会显式地返回一个值，因为大多数构造函数都不会。如果用户尝试这样做会发生什么：

```
class Foo {
  init() {
    return "something else";
  }
}
```

> It’s definitely not going to do what they want, so we may as well make it a static error. Back in the resolver, we add another case to FunctionType.

这肯定不会按照用户的期望执行，所以我们不妨把它作为一种静态错误。回到分析器中，我们为FunctionType添加另一种情况。

lox/Resolver.java，在 FunctionType枚举中添加：

```
    FUNCTION,
    // 新增部分开始
    INITIALIZER,
    // 新增部分结束
    METHOD
```

> We use the visited method’s name to determine if we’re resolving an initializer or not.

我们通过被访问方法的名称来确定我们是否在分析一个初始化方法。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
      FunctionType declaration = FunctionType.METHOD;
      // 新增部分开始
      if (method.name.lexeme.equals("init")) {
        declaration = FunctionType.INITIALIZER;
      }
      // 新增部分结束
      resolveFunction(method, declaration);
```

> When we later traverse into a `return` statement, we check that field and make it an error to return a value from inside an `init()` method.

当我们稍后遍历`return`语句时，我们会检查该字段，如果从`init()`方法内部返回一个值时就抛出一个错误。

lox/Resolver.java，在 visitReturnStmt()方法中添加：

```
    if (stmt.value != null) {
      // 新增部分开始
      if (currentFunction == FunctionType.INITIALIZER) {
        Lox.error(stmt.keyword,
            "Can't return a value from an initializer.");
      }
      // 新增部分结束
      resolve(stmt.value);
```

> We're *still* not done. We statically disallow returning a *value* from an initializer, but you can still use an empty early `return`.

我们*仍然*没有结束。我们静态地禁止了从初始化方法返回一个值，但是你仍然可用使用一个空的`return`。

```
class Foo {
  init() {
    return;
  }
}
```

> That is actually kind of useful sometimes, so we don't want to disallow it entirely. Instead, it should return `this` instead of `nil`. That's an easy fix over in LoxFunction.

有时候这实际上是有用的，所以我们不想完全禁止它。相对地，它应该返回`this`而不是`nil`。这在LoxFunction中很容易解决。

lox/LoxFunction.java，在 call()方法中添加：

```
    } catch (Return returnValue) {
      // 新增部分开始
      if (isInitializer) return closure.getAt(0, "this");
      // 新增部分结束
      return returnValue.value;
```

> If we're in an initializer and execute a `return` statement, instead of returning the value (which will always be `nil`), we again return `this`.

如果我们在一个初始化方法中执行`return`语句时，我们仍然返回`this`，而不是返回值（该值始终是`nil`）。

> Phew! That was a whole list of tasks but our reward is that our little interpreter has grown an entire programming paradigm. Classes, methods, fields, `this`, and constructors. Our baby language is looking awfully grown-up.

吁！这是一大堆任务，但是我们的收获是，我们的小解释器已经成长为一个完整的编程范式。类、方法、字段、`this`以及构造函数，我们的语言看起来已经非常成熟了。

^2: Multimethods是你最不可能熟悉的方法。我很想多谈论一下它们——我曾经围绕它们设计了一个业余语言，它们特别棒——但是我只能装下这么多页面了。如果你想了解更多，可以看看CLOS (Common Lisp中的对象系统), Dylan, Julia, 或 Raku。

```
// 方式1
fun callback(a, b, c) {
  object.method(a, b, c);
}

takeCallback(callback);

// 方式2
takeCallback(object.method);
```

^9: C++中的 "placement new "是一个罕见的例子，在这种情况下，分配的内存被暴露出来供程序员使用。
^10: 也许"不喜欢"这个说法太过激了。让语言实现的约束和资源影响语言的设计是合理的。一天只有这么多时间，如果在这里或那里偷工减料可以让你在更短的时间内为用户提供更多的功能，这可能会大大提高用户的幸福感和工作效率。诀窍在于，要弄清楚哪些弯路不会导致你的用户和未来的自己不会咒骂你的短视行为

CHALLENGES

习题

1. > We have methods on instances, but there is no way to define "static" methods that can be called directly on the class object itself. Add support for them. Use a `class` keyword preceding the method to indicate a static method that hangs off the class object.

   我们有实例上的方法，但是没有办法定义可以直接在类对象上调用的"静态"方法。添加对它们的支持，在方法之前使用`class`关键字指示该方法是一个挂载在类对象上的静态方法。

   ```
   class Math {
     class square(n) {
       return n * n;
     }
   }

   print Math.square(3); // Prints "9".
   ```

   > You can solve this however you like, but the "metaclasses" used by Smalltalk and Ruby are a particularly elegant approach. *Hint: Make LoxClass extend LoxInstance and go from there.*

   你可以用你喜欢的方式解决这问题，但是Smalltalk和Ruby使用的"metaclasses" 是一种特别优雅的方法。*提示：让LoxClass继承LoxInstance，然后开始实现。*

2. > Most modern languages support "getters" and "setters"—members on a class that look like field reads and writes but that actually execute user-defined code. Extend Lox to support getter methods. These are declared without a parameter list. The body of the getter is executed when a property with that name is accessed.

   大多数现代语言都支持"getters"和"setters"——类中的成员，看起来像是字段的读写，但实际上执行的用户自定义的代码。扩展Lox以支持getter方法。这些方法在声明时没有参数列表。当访问具有该名称的属

性时，会执行getter的主体。

```
class Circle {
  init(radius) {
    this.radius = radius;
  }

  area {
    return 3.141592653 * this.radius * this.radius;
  }
}

var circle = Circle(4);
print circle.area; // Prints roughly "50.2655".
```

3. > Python and JavaScript allow you to freely access an object's fields from outside of its own methods. Ruby and Smalltalk encapsulate instance state. Only methods on the class can access the raw fields, and it is up to the class to decide which state is exposed. Most statically typed languages offer modifiers like `private` and `public` to control which parts of a class are externally accessible on a per-member basis.
   >
   > What are the trade-offs between these approaches and why might a language prefer one or the other?

   Python和JavaScript允许你从对象自身的方法之外的地方自由访问对象的字段。Ruby和Smalltalk封装了实例状态。只有类上的方法可以访问原始字段，并且由类来决定哪些状态被暴露。大多数静态类型的语言都提供了像`private`和`public`这样的修饰符，以便按成员维度控制类的哪些部分可以被外部访问。

   这些方式之间的权衡是什么？为什么一门语言可能会更偏爱某一种方法？

DESIGN NOTE: PROTOTYPES AND POWER

设计笔记：原型与功率

> In this chapter, we introduced two new runtime entities, LoxClass and LoxInstance. The former is where behavior for objects lives, and the latter is for state. What if you could define methods right on a single object, inside LoxInstance? In that case, we wouldn't need LoxClass at all. LoxInstance would be a complete package for defining the behavior and state of an object.
>
> We'd still want some way, without classes, to reuse behavior across multiple instances. We could let a LoxInstance *delegate* directly to another LoxInstance to reuse its fields and methods, sort of like inheritance.
>
> Users would model their program as a constellation of objects, some of which delegate to each other to reflect commonality. Objects used as delegates represent "canonical" or "prototypical" objects that others refine. The result is a simpler runtime with only a single internal construct, LoxInstance.

That's where the name **prototypes** comes from for this paradigm. It was invented by David Ungar and Randall Smith in a language called Self. They came up with it by starting with Smalltalk and following the above mental exercise to see how much they could pare it down.

Prototypes were an academic curiosity for a long time, a fascinating one that generated interesting research but didn't make a dent in the larger world of programming. That is, until Brendan Eich crammed prototypes into JavaScript, which then promptly took over the world. Many (many) words have been written about prototypes in JavaScript. Whether that shows that prototypes are brilliant or confusing—or both!—is an open question.

Including more than a handful by yours truly.

I won't get into whether or not I think prototypes are a good idea for a language. I've made languages that are prototypal and class-based, and my opinions of both are complex. What I want to discuss is the role of *simplicity* in a language.

Prototypes are simpler than classes—less code for the language implementer to write, and fewer concepts for the user to learn and understand. Does that make them better? We language nerds have a tendency to fetishize minimalism. Personally, I think simplicity is only part of the equation. What we really want to give the user is *power*, which I define as:

```
power = breadth × ease ÷ complexity
```

None of these are precise numeric measures. I'm using math as analogy here, not actual quantification.

- **Breadth** is the range of different things the language lets you express. C has a lot of breadth—it's been used for everything from operating systems to user applications to games. Domain-specific languages like AppleScript and Matlab have less breadth.
- **Ease** is how little effort it takes to make the language do what you want. "Usability" might be another term, though it carries more baggage than I want to bring in. "Higher-level" languages tend to have more ease than "lower-level" ones. Most languages have a "grain" to them where some things feel easier to express than others.
- **Complexity** is how big the language (including its runtime, core libraries, tools, ecosystem, etc.) is. People talk about how many pages are in a language's spec, or how many keywords it has. It's how much the user has to load into their wetware before they can be productive in the system. It is the antonym of simplicity.

Reducing complexity *does* increase power. The smaller the denominator, the larger the resulting value, so our intuition that simplicity is good is valid. However, when reducing complexity, we must take care not to sacrifice breadth or ease in the process, or the total power may go down. Java would be a strictly *simpler* language if it removed strings, but it probably wouldn't handle text manipulation tasks well, nor would it be as easy to get things done.

The art, then, is finding *accidental* complexity that can be omitted—language features and interactions that don't carry their weight by increasing the breadth or ease of using the language.

If users want to express their program in terms of categories of objects, then baking classes into the language increases the ease of doing that, hopefully by a large enough margin to pay for the added

> complexity. But if that isn’t how users are using your language, then by all means leave classes out.

在本章中，我们引入了两个新的运行时实体，LoxClass和LoxInstance。前者是对象的行为所在，后者则是状态所在。如果你可以在LoxInstance的单个对象中定义方法，会怎么样？这种情况下，我们根本就不需要LoxClass。LoxInstance将是一个用于定义对象行为和状态的完整包。

我们仍然需要一些方法，在没有类的情况下，可以跨多个实例重用对象行为。我们可以让一个LoxInstance直接委托给另一个LoxInstance来重用它的字段和方法，有点像继承。

用户可以将他们的程序建模为一组对象，其中一些对象相互委托以反映共性。用作委托的对象代表“典型”或“原型”对象，会被其它对象完善。结果就是会有一个更简单的运行时，只有一个内部结构LoxInstance。

这就是这种范式的名称“原型”的由来。它是由David Ungar和Randall Smith在一种叫做Self的语言中发明的。他们从Smalltalk开始，按照上面的练习，看他们能把它缩减到什么程度，从而想到了这个方法。

长期以来，原型一直是学术上的探索，它是一个引人入胜的东西，也产生了有趣的研究，但是并没有在更大的编程世界中产生影响。直到Brendan Eich把原型塞进JavaScript，然后迅速风靡世界。关于JavaScript中的原型，人们已经写了很多（许多）文字。这是否能够表明原型是出色的还是令人困惑的，或者兼而有之？这是一个开放的问题。

我不会去讨论原型对于一门语言来说是不是一个好主意。基于原型和基于类的语言我都做过，我对两者的看法很复杂。我想讨论的是*简单性*在一门语言中的作用。

原型比类更简单——语言实现者要编写的代码更少，语言用户要学习和理解的概念更少。这是否意味着它让语言变得更好呢？我们这些语言书呆子有一种迷恋极简主义的倾向。就我个人而言，我认为简单性只是一部分。我们真正想给用户的是功率，我将其定义为：

```
power = breadth × ease ÷ complexity
功率 = 广度 × 易用性 ÷ 复杂性
```

这些都不是精确的数字度量。我这里用数学作比喻，而不是实际的量化。

- **广度**是语言可以表达的不同事物的范围。C语言具有很大的广度——从操作系统到用户应用程序再到游戏，它被广泛使用。像AppleScript和Matlab这样的特定领域语言的广度相对较小。
- 易用性是指用户付出多少努力就可以用语言做想做的事。“可用性Usability”是另一个概念，它包含的内容比我想要表达的更多。“高级”语言往往比“低级”语言更容易使用。大多数语言都有一个核心，对它们来说，有些东西比其它的更容易表达。
- 复杂性是指语言的规模（包括其运行时、核心库、工具、生态等）有多大。人们谈论一种语言的规范有多少页，或者它有多少个关键词。这是指用户在使用系统之前，必须在先学习多少东西，才能产生效益。它是简单性的反义词。

降低复杂性确实可以提高功率，分母越小，得到的值就越大，所以我们直觉认为“简单的是好的”是对的。然而，在降低复杂性时，我们必须注意不要在这个过程中牺牲广度或易用性，否则总功率可能会下降。如果去掉字符串，Java将变成一种严格意义上的简单语言，但它可能无法很好地处理文本操作任务，也不会那么容易完成事情。

因此，关键就在于找到可以省略的意外复杂性，也就是哪些没有通过增加语言广度或语言易用性来体现其重要性的语言特性与交互。

如果用户想用对象的类别来表达他们的程序，那么在语言中加入类就能提高这类操作的便利性，希望能有足够大的提升幅度来弥补所增加的复杂性。但如果这不是用户使用您的语言的方式，那么无论如何都不要使用类。

13.继承 Inheritance

> Once we were blobs in the sea, and then fishes, and then lizards and rats and then monkeys, and hundreds of things in between. This hand was once a fin, this hand once had claws! In my human mouth I have the pointy teeth of a wolf and the chisel teeth of a rabbit and the grinding teeth of a cow! Our blood is as salty as the sea we used to live in! When we're frightened, the hair on our skin stands up, just like it did when we had fur. We are history! Everything we've ever been on the way to becoming us, we still are.
>
> —— Terry Pratchett, *A Hat Full of Sky*

我们曾经是海里一团一团的东西，然后是鱼，然后是蜥蜴、老鼠、猴子，以及介于其间的数百种形态。这只手曾经是鳍，这只手曾经是爪子！在我人类的嘴里，有狼的尖牙，有兔子的凿齿，还有牛的磨牙！我们的血和我们曾经生活的大海一样咸！当我们受到惊吓时，我们皮肤上的毛发会竖起来，就像我们有毛时一样。我们就是历史！我们在成为我们的路上曾拥有的一切，我们仍然拥有。

> Can you believe it? We've reached the last chapter of Part II. We're almost done with our first Lox interpreter. The previous chapter was a big ball of intertwined object-orientation features. I couldn't separate those from each other, but I did manage to untangle one piece. In this chapter, we'll finish off Lox's class support by adding inheritance.

你能相信吗？我们已经到了第二部分的最后一章。我们几乎已经完成了第一个Lox解释器。上一章中是一大堆错综复杂的面向对象特性。我无法将这些内容完全解开，但是我设法拆出来一块。在这一章，我们会添加继承来完成Lox中对类的支持。

> Inheritance appears in object-oriented languages all the way back to the first one, Simula. Early on, Kristen Nygaard and Ole-Johan Dahl noticed commonalities across classes in the simulation programs they wrote. Inheritance gave them a way to reuse the code for those similar parts.

继承出现在面向对象语言中，可以追溯到第一种语言Simula。早些时候，克里斯汀·尼加德(Kristen Nygaard)和奥勒-约翰·达尔(Ole-Johan Dahl)注意到，在他们编写的模拟程序中，不同类之间存在共性。继承为他们提供了一种重用相似部分代码的方法。

> ## 13 . 1 Superclasses and Subclasses

13.1 超类和子类

> Given that the concept is "inheritance", you would hope they would pick a consistent metaphor and call them "parent" and "child" classes, but that would be too easy. Way back when, C. A. R. Hoare coined the term "subclass" to refer to a record type that refines another type. Simula borrowed that term to refer to a *class* that inherits from another. I don't think it was until Smalltalk came along that someone flipped the Latin prefix to get "superclass" to refer to the other side of the relationship. From C++, you also hear "base" and "derived" classes. I'll mostly stick with "superclass" and "subclass".

鉴于这个概念叫"继承"，你可能希望他们会选择一个一致的比喻，把类称为"父"类和"子"类，但这太简单了。早在很久以前，C. A. R. Hoare就创造了"subclass"这个术语，指的是完善另一种类型的记录类型。Simula借用了这

个术语来指代一个继承自另一个类的类。我认为直到Smalltalk出现后，才有人将这个词的拉丁前缀取反义^1，用超类（superclass）指代这种关系的另一方。

> Our first step towards supporting inheritance in Lox is a way to specify a superclass when declaring a class. There's a lot of variety in syntax for this. C++ and C# place a `:` after the subclass's name, followed by the superclass name. Java uses `extends` instead of the colon. Python puts the superclass(es) in parentheses after the class name. Simula puts the superclass's name *before* the `class` keyword.

我们在Lox中支持继承的第一步是找到声明类时指定超类的方法。这方面有很多不同的语法。C++和C#在子类的名字后面加一个`:`，然后是超类的名字。Java 使用 `extends` 而不是冒号。Python 则把超类放在类名后面的小括号里。Simula 把超类的名字放在关键字`class`之前。

> This late in the game, I'd rather not add a new reserved word or token to the lexer. We don't have `extends` or even `:`, so we'll follow Ruby and use a less-than sign (`<`).

游戏已经到了后期，我宁愿不在词法分析器中添加新的保留字或标记。我们没有`extends`或`:`，所以我们遵循Ruby来使用小于号（`<`）。

```
class Doughnut {
  // General doughnut stuff...
}

class BostonCream < Doughnut {
  // Boston Cream-specific stuff...
}
```

> To work this into the grammar, we add a new optional clause in our existing `classDecl` rule.

为了在语法中实现这一点，我们在目前的`classDecl`规则中加一个新的可选子句。

```
classDecl      → "class" IDENTIFIER ( "<" IDENTIFIER )?
                 "{" function* "}" ;
```

> After the class name, you can have a `<` followed by the superclass's name. The superclass clause is optional because you don't *have* to have a superclass. Unlike some other object-oriented languages like Java, Lox has no root "Object" class that everything inherits from, so when you omit the superclass clause, the class has *no* superclass, not even an implicit one.

在类的名称后面，可以有一个`<`，后跟超类的名称。超类子句是可选的，因为一个类不一定要有超类。与Java等面向对象的语言不同，Lox没有所有东西都继承的一个根"Object"类，所以当你省略超类子句时，该类就没有超类，甚至连隐含的都没有。

> We want to capture this new syntax in the class declaration's AST node.

我们想在类声明的AST节点中捕捉这个新语法。

tool/GenerateAst.java，在 main()方法中，替换一行：

```
      "Block      : List<Stmt> statements",
      // 替换部分开始
      "Class      : Token name, Expr.Variable superclass, List<Stmt.Function>
methods",
      // 替换部分结束
      "Expression : Expr expression",
```

> You might be surprised that we store the superclass name as an Expr.Variable, not a Token. The grammar restricts the superclass clause to a single identifier, but at runtime, that identifier is evaluated as a variable access. Wrapping the name in an Expr.Variable early on in the parser gives us an object that the resolver can hang the resolution information off of.

你可能会惊讶，我们把超类的名字存为一个Expr.Variable，而不是一个Token。语法将一个超类子句限制为一个标识符，但是在运行时，这个标识符是当作变量访问来执行的。在解析器早期将名称封装在Expr.Variable内部，这样可以给我们提供一个对象，在分析器中可以将分析信息附加在其中。

> The new parser code follows the grammar directly.

新的解析器代码直接遵循语法。

lox/Parser.java，在 classDeclaration()中添加：

```
    Token name = consume(IDENTIFIER, "Expect class name.");
    // 新增部分开始
    Expr.Variable superclass = null;
    if (match(LESS)) {
      consume(IDENTIFIER, "Expect superclass name.");
      superclass = new Expr.Variable(previous());
    }
    // 新增部分结束
    consume(LEFT_BRACE, "Expect '{' before class body.");
```

> Once we've (possibly) parsed a superclass declaration, we store it in the AST.

一旦我们（可能）解析到一个超类声明，就将其保存到AST节点中。

lox/Parser.java，在 classDeclaration()方法中，替换一行：

```
    consume(RIGHT_BRACE, "Expect '}' after class body.");
    // 替换部分开始
    return new Stmt.Class(name, superclass, methods);
    // 替换部分结束
  }
```

> If we didn't parse a superclass clause, the superclass expression will be `null`. We'll have to make sure the later passes check for that. The first of those is the resolver.

如果我们没有解析到超类子句，超类表达式将是`null`。我们必须确保后面的操作会对其进行检查。首先是分析器。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
    define(stmt.name);
    // 新增部分开始
    if (stmt.superclass != null) {
      resolve(stmt.superclass);
    }
    // 新增部分结束
    beginScope();
```

> The class declaration AST node has a new subexpression, so we traverse into and resolve that. Since classes are usually declared at the top level, the superclass name will most likely be a global variable, so this doesn't usually do anything useful. However, Lox allows class declarations even inside blocks, so it's possible the superclass name refers to a local variable. In that case, we need to make sure it's resolved.

类声明的AST节点有一个新的子表达式，所以我们要遍历并分析它。因为类通常是在顶层声明的，超类的名称很可能是一个全局变量，所以这一步通常没有什么作用。然而，Lox运行在区块内的类声明，所以超类名称有可能指向一个局部变量。在那种情况下，我们需要保证能被它被分析。

> Because even well-intentioned programmers sometimes write weird code, there's a silly edge case we need to worry about while we're in here. Take a look at this:

即使是善意的程序员有时也会写出奇怪的代码，所以在这里我们需要考虑一个愚蠢的边缘情况。看看这个：

```
class Oops < Oops {}
```

> There's no way this will do anything useful, and if we let the runtime try to run this, it will break the expectation the interpreter has about there not being cycles in the inheritance chain. The safest thing is to detect this case statically and report it as an error.

这种代码不可能做什么有用的事情，如果我们尝试让运行时去执行它，将会打破解释器对继承链中没有循环的期望。最安全的做法是静态地检测这种情况，并将其作为一个错误报告出来。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
    define(stmt.name);
    // 新增部分开始
    if (stmt.superclass != null &&
        stmt.name.lexeme.equals(stmt.superclass.name.lexeme)) {
      Lox.error(stmt.superclass.name,
          "A class can't inherit from itself.");
    }
    // 新增部分结束
    if (stmt.superclass != null) {
```

> Assuming the code resolves without error, the AST travels to the interpreter.

如果代码分析没有问题，AST节点就会被传递到解释器。

lox/Interpreter.java，在 visitClassStmt()方法中添加：

```
  public Void visitClassStmt(Stmt.Class stmt) {
    // 新增部分开始
    Object superclass = null;
    if (stmt.superclass != null) {
      superclass = evaluate(stmt.superclass);
      if (!(superclass instanceof LoxClass)) {
        throw new RuntimeError(stmt.superclass.name,
            "Superclass must be a class.");
      }
    }
    // 新增部分结束
    environment.define(stmt.name.lexeme, null);
```

> If the class has a superclass expression, we evaluate it. Since that could potentially evaluate to some other kind of object, we have to check at runtime that the thing we want to be the superclass is actually a class. Bad things would happen if we allowed code like:

如果类中有超类表达式，我们就对其求值。因为我们可能会得到其它类型的对象，我们在运行时必须检查我们希望作为超类的对象是否确实是一个类。如果我们允许下面这样的代码，就会发生不好的事情：

```
var NotAClass = "I am totally not a class";

class Subclass < NotAClass {} // ?!
```

> Assuming that check passes, we continue on. Executing a class declaration turns the syntactic representation of a class—its AST node—into its runtime representation, a LoxClass object. We need to plumb the superclass through to that too. We pass the superclass to the constructor.

假设检查通过，我们继续。执行类声明语句会把类的语法表示（AST节点）转换为其运行时表示（一个LoxClass对象）。我们也需要把超类对象传入该类对象中。我们将超类传递给构造函数。

lox/Interpreter.java，在 visitClassStmt()方法中替换一行：

```
      methods.put(method.name.lexeme, function);
    }
    // 替换部分开始
    LoxClass klass = new LoxClass(stmt.name.lexeme,
        (LoxClass)superclass, methods);
    // 替换部分结束
    environment.assign(stmt.name, klass);
```

> The constructor stores it in a field.

构造函数将它存储到一个字段中。

lox/LoxClass.java，LoxClass()构造函数中，替换一行：

```
  // 替换部分开始
  LoxClass(String name, LoxClass superclass,
           Map<String, LoxFunction> methods) {
    this.superclass = superclass;
    // 替换部分结束
    this.name = name;
```

> Which we declare here:

字段我们在这里声明：

lox/LoxClass.java，在 LoxClass类中添加：

```
  final String name;
  // 新增部分开始
  final LoxClass superclass;
  // 新增部分结束
  private final Map<String, LoxFunction> methods;
```

> With that, we can define classes that are subclasses of other classes. Now, what does having a superclass actually *do?*

有了这个，我们就可以定义一个类作为其它类的子类。现在，拥有一个超类究竟有什么用呢？

13.2 Inheriting Methods

13.2 继承方法

> Inheriting from another class means that everything that's true of the superclass should be true, more or less, of the subclass. In statically typed languages, that carries a lot of implications. The sub*class* must also be a sub*type*, and the memory layout is controlled so that you can pass an instance of a subclass to a function expecting a superclass and it can still access the inherited fields correctly.

继承自另一个类，意味着对于超类适用的一切，对于子类或多或少也应该适用。在静态类型的语言中，这包含了很多含义。子类也必须是一个子类型，而且内存布局是可控的，这样你就可以把一个子类实例传递给一个期望超类的函数，而它仍然可以正确地访问继承的字段。

> Lox is a dynamically typed language, so our requirements are much simpler. Basically, it means that if you can call some method on an instance of the superclass, you should be able to call that method

> when given an instance of the subclass. In other words, methods are inherited from the superclass.

Lox是一种动态类型的语言，所以我们的要求要简单得多。基本上，这意味着如果你能在超类的实例上调用某些方法，那么当给你一个子类的实例时，你也应该能调用这个方法。换句话说，方法是从超继承的。

> This lines up with one of the goals of inheritance—to give users a way to reuse code across classes. Implementing this in our interpreter is astonishingly easy.

这符合继承的目标之一——为用户提供一种跨类重用代码的方式。在我们的解释器中实现这一点是非常容易的。

lox/LoxClass.java，在 findMethod()方法中添加：

```
      return methods.get(name);
    }
    // 新增部分开始
    if (superclass != null) {
      return superclass.findMethod(name);
    }
    // 新增部分结束
    return null;
```

> That's literally all there is to it. When we are looking up a method on an instance, if we don't find it on the instance's class, we recurse up through the superclass chain and look there. Give it a try:

这就是它的全部内容。当我们在一个实例上查找一个方法时，如果我们在实例的类中找不到它，就沿着超类继承链递归查找。试一下这个：

```
class Doughnut {
  cook() {
    print "Fry until golden brown.";
  }
}

class BostonCream < Doughnut {}

BostonCream().cook();
```

> There we go, half of our inheritance features are complete with only three lines of Java code.

好了，一半的继承特性只用了三行Java代码就完成了。

13.3 Calling Superclass Methods

13.3 调用超类方法

> In `findMethod()` we look for a method on the current class *before* walking up the superclass chain. If a method with the same name exists in both the subclass and the superclass, the subclass one takes

> precedence or **overrides** the superclass method. Sort of like how variables in inner scopes shadow outer ones.

在`findMethod()`方法中，我们首先在当前类中查找，然后遍历超类链。如果在子类和超类中包含相同的方法，那么子类中的方法将优先于或**覆盖**超类的方法。这有点像内部作用域中的变量对外部作用域的遮蔽。

> That’s great i**f the subclass wants to *replace* some superclass behavior completely. But, in practice, subclasses often want to *refine* the superclass’s behavior. They want to do a little work specific to the subclass, but also execute the original superclass behav**ior too.

如果子类想要完全 *替换*超类的某些行为，那就正好。但是，在实践中，子类通常想改进超类的行为。他们想要做一些专门针对子类的操作，但是也想要执行原来超类中的行为。

> However, since the subclass has overridden the method, there’s no way to refer to the original one. If the subclass method tries to call it by name, it will just recursively hit its own override. We need a way to say “Call this method, but look for it directly on my superclass and ignore my override”. Java uses `super` for this, and we’ll use that same syntax in Lox. Here is an example:

然而，由于子类已经重写了该方法，所有没有办法指向原始的方法。如果子类的方法试图通过名字来调用它，将会递归到自身的重写方法上。我们需要一种方式来表明“调用这个方法，但是要直接在我的超类上寻找，忽略我内部的重写方法”。Java中使用`super`实现这一点，我们在Lox中使用相同的语法。下面是一个例子：

```
class Doughnut {
  cook() {
    print "Fry until golden brown.";
  }
}

class BostonCream < Doughnut {
  cook() {
    super.cook();
    print "Pipe full of custard and coat with chocolate.";
  }
}

BostonCream().cook();
```

> If you run this, it should print:

如果你运行该代码，应该打印出：

```
Fry until golden brown.
Pipe full of custard and coat with chocolate.
```

> We have a new expression form. The `super` keyword, followed by a dot and an identifier, looks for a method with that name. Unlike calls on `this`, the search starts at the superclass.

我们有了一个新的表达式形式。super关键字，后跟一个点和一个标识符，以使用该名称查找方法。与this调用不同，该搜索是从超类开始的。

> 13.3.1 Syntax

13.3.1 语法

> With this, the keyword works sort of like a magic variable, and the expression is that one lone token. But with super, the subsequent . and property name are inseparable parts of the super expression. You can't have a bare super token all by itself.

在this使用中，关键字有点像一个魔法变量，而表达式是一个单独的标记。但是对于super，随后的.和属性名是super表达式不可分割的一部分。你不可能只有一个单独的super标记。

```
print super; // Syntax error.
```

> So the new clause we add to the primary rule in our grammar includes the property access as well.

因此，我们在语法中的primary规则添加新子句时要包含属性访问。

```
primary        → "true" | "false" | "nil" | "this"
               | NUMBER | STRING | IDENTIFIER | "(" expression ")"
               | "super" "." IDENTIFIER ;
```

> Typically, a super expression is used for a method call, but, as with regular methods, the argument list is *not* part of the expression. Instead, a super *call* is a super *access* followed by a function call. Like other method calls, you can get a handle to a superclass method and invoke it separately.

通常情况下，super表达式用于方法调用，但是，与普通方法一样，参数列表并不是表达式的一部分。相反，super调用是一个super属性访问，然后跟一个函数调用。与其它方法调用一样，你可以获得超类方法的句柄，然后单独运行它。

```
var method = super.cook;
method();
```

> So the super expression itself contains only the token for the super keyword and the name of the method being looked up. The corresponding syntax tree node is thus:

因此，super表达式本身只包含super关键字和要查找的方法名称。对应的语法树节点为：

tool/GenerateAst.java，在 main()方法中添加：

```
      "Set      : Expr object, Token name, Expr value",
      // 新增部分开始
      "Super    : Token keyword, Token method",
      // 新增部分结束
      "This     : Token keyword",
```

> Following the grammar, the new parsing code goes inside our existing `primary()` method.

按照语法，需要在我们现有的`primary`方法中添加新代码。

lox/Parser.java，在 primary()方法中添加：

```
      return new Expr.Literal(previous().literal);
    }
    // 新增部分开始
    if (match(SUPER)) {
      Token keyword = previous();
      consume(DOT, "Expect '.' after 'super'.");
      Token method = consume(IDENTIFIER,
          "Expect superclass method name.");
      return new Expr.Super(keyword, method);
    }
    // 新增部分结束
    if (match(THIS)) return new Expr.This(previous());
```

> A leading `super` keyword tells us we've hit a `super` expression. After that we consume the expected `.` and method name.

开头的`super`关键字告诉我们遇到了一个`super`表达式，之后我们消费预期中的`.`和方法名称。

> 13.3.2 Semantics

13.3.2 语义

> Earlier, I said a `super` expression starts the method lookup from "the superclass", but *which* superclass? The naïve answer is the superclass of `this`, the object the surrounding method was called on. That coincidentally produces the right behavior in a lot of cases, but that's not actually correct. Gaze upon:

之前，我说过`super`表达式从"超类"开始查找方法，但是是哪个超类？一个不太成熟的答案是方法被调用时的外围对象`this`的超类。在很多情况下，这碰巧产生了正确的行为，但实际上这是不正确的。请看：

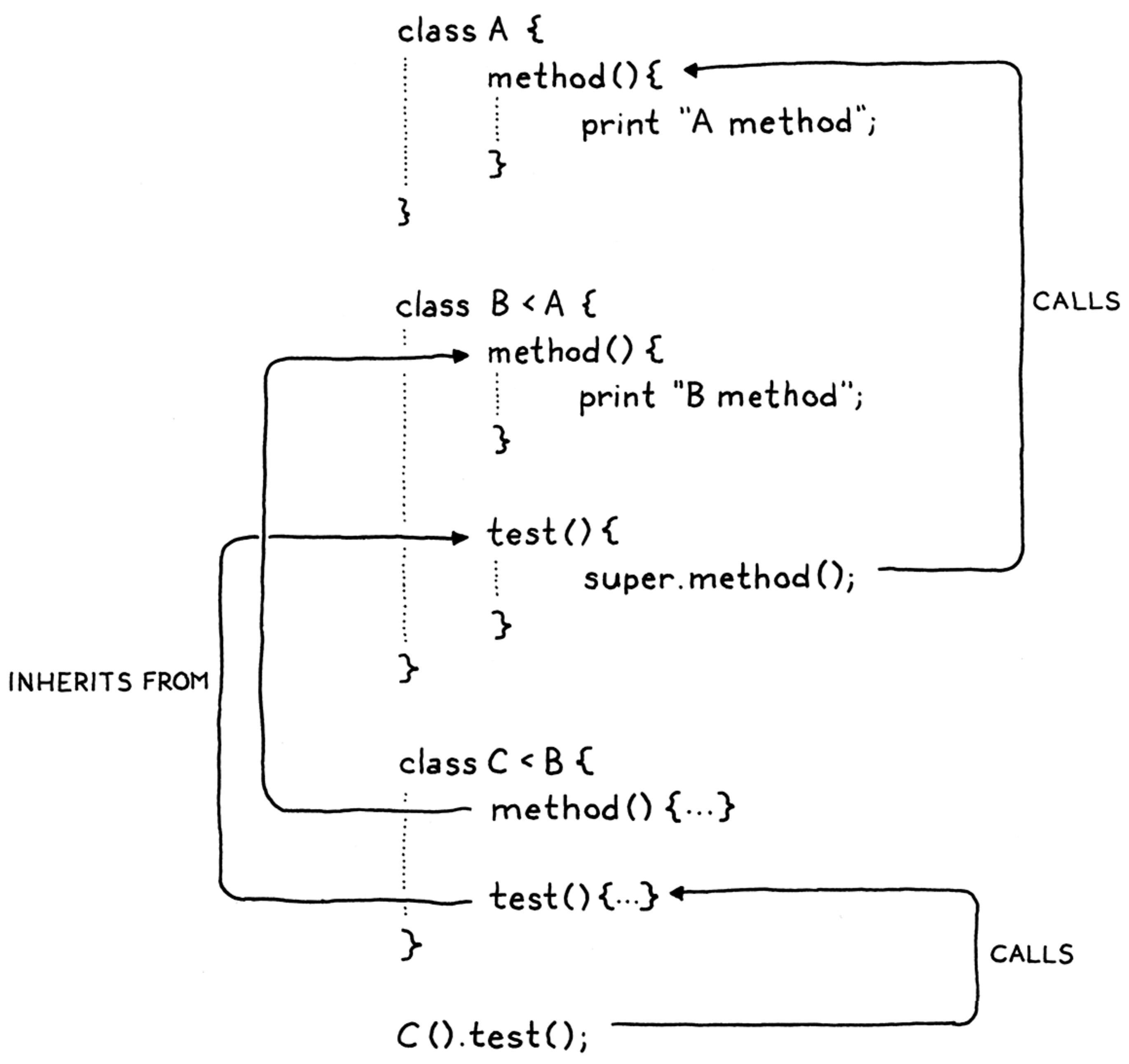
```
class A {
  method() {
    print "A method";
  }
}

class B < A {
  method() {
    print "B method";
  }
  test() {
    super.method();
  }
}

class C < B {}

C().test();
```

> Translate this program to Java, C#, or C++ and it will print "A method", which is what we want Lox to do too. When this program runs, inside the body of `test()`, `this` is an instance of C. The superclass of C is B, but that is *not* where the lookup should start. If it did, we would hit B's `method()`.

将这个程序转换为Java、c#或c++，它将输出"A method"，这也是我们希望Lox做的。当这个程序运行时，在`test`方法体中，`this`是C的一个实例，C是超类是B，但这不是查找应该开始的地方。如果是这样，我们就会命中B的`method()`。

> Instead, lookup should start on the superclass of *the class containing the `super` expression*. In this case, since `test()` is defined inside B, the `super` expression inside it should start the lookup on *B*'s superclass—A.

相反，查找应该从包含`super`表达式的类的超类开始。在这个例子中，由于`test()`是在B中定义的，它内部的`super`表达式应该在B的超类A中开始查找。



> The execution flow looks something like this:

执行流程看起来是这样的：

1. > We call `test()` on an instance of C.

   我们在C的一个实例上调用`test()`。

2. > That enters the `test()` method inherited from B. That calls `super.method()`.

   这就进入了从B中继承的`test()`方法，其中又会调用`super.method()`。

3. > The superclass of B is A, so that chains to `method()` on A, and the program prints “A method”.

   B的超类是A，所以链接到A中的`method()`，程序会打印出“A method”。

> Thus, in order to evaluate a `super` expression, we need access to the superclass of the class definition surrounding the call. Alack and alas, at the point in the interpreter where we are executing a `super` expression, we don’t have that easily available.

因此，为了对`super`表达式求值，我们需要访问围绕方法调用的类的超类。可惜的是，在解释器中执行`super`表达式的地方，我们并没有那么容易获得。

> We *could* add a field to LoxFunction to store a reference to the LoxClass that owns that method. The interpreter would keep a reference to the currently executing LoxFunction so that we could look it up later when we hit a `super` expression. From there, we'd get the LoxClass of the method, then its superclass.

我们可以从LoxFunction添加一个字段，以存储指向拥有该方法的LoxClass的引用。解释器会保存当前正在执行的LoxFunction的引用，这样稍后在遇到`super`表达式时就可以找到它。从它开始，可以得到方法的LoxClass，然后找到它的超类。

> That's a lot of plumbing. In the last chapter, we had a similar problem when we needed to add support for `this`. In that case, we used our existing environment and closure mechanism to store a reference to the current object. Could we do something similar for storing the superclass? Well, I probably wouldn't be talking about it if the answer was no, so . . . yes.

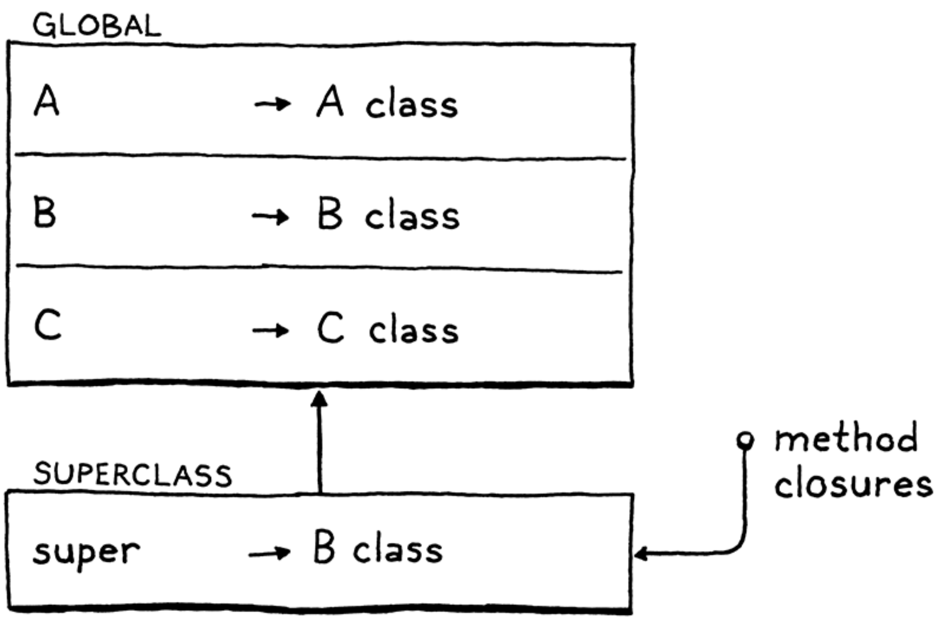
这需要很多管道。在上一章中，我们添加对`this`的支持时遇到了类似的问题。在那种情况下，我们使用已有的环境和闭包机制保存了指向当前对象的引用。那我们是否可以做类似的事情来存储超类？嗯，如果答案是否定的，我就不会问这个问题了，所以……是的。

> One important difference is that we bound `this` when the method was *accessed*. The same method can be called on different instances and each needs its own `this`. With `super` expressions, the superclass is a fixed property of the *class declaration itself*. Every time you evaluate some `super` expression, the superclass is always the same.

一个重要的区别是，我们在方法被访问时绑定了`this`。同一个方法可以在不同的实例上被调用，而且每个实例都需要有自己的`this`。对于`super`表达式，超类是*类声明本身*的一个固定属性。每次对某个`super`表达式求值时，超类都是同一个。

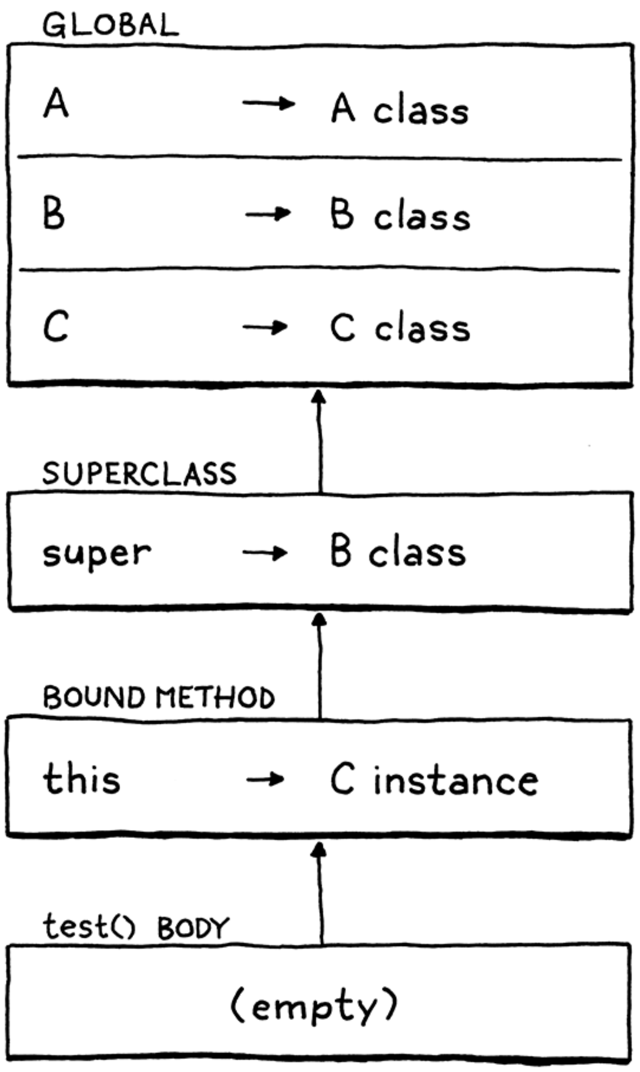
> That means we can create the environment for the superclass once, when the class definition is executed. Immediately before we define the methods, we make a new environment to bind the class's superclass to the name `super`.

这意味着我们可以在执行类定义时，为超类创建一个环境。在定义方法之前，我们创建一个新环境，将类的超类与名称`super`绑定。



> When we create the LoxFunction runtime representation for each method, that is the environment they will capture in their closure. Later, when a method is invoked and `this` is bound, the superclass environment becomes the parent for the method’s environment, like so:

当我们为每个方法创建LoxFunction运行时表示时，也就是这个方法闭包中获取的环境。之后，放方法被调用时会绑定`this`，超类环境会成为方法环境的父环境，就像这样：



> That's a lot of machinery, but we'll get through it a step at a time. Before we can get to creating the environment at runtime, we need to handle the corresponding scope chain in the resolver.

这是一个复杂的机制，但是我们会一步一步完成它。在我们可以在运行时创建环境之前，我们需要在分析器中处理对应的作用域。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
      resolve(stmt.superclass);
    }
    // 新增部分开始
    if (stmt.superclass != null) {
      beginScope();
      scopes.peek().put("super", true);
    }
    // 新增部分结束
    beginScope();
```

> If the class declaration has a superclass, then we create a new scope surrounding all of its methods. In that scope, we define the name "super". Once we're done resolving the class's methods, we discard that scope.

如果该类声明有超类，那么我们就在其所有方法的外围创建一个新的作用域。在这个作用域中，我们会定义名称`super`。一旦我们完成了对该类中方法的分析，就丢弃这个作用域。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
    endScope();
    // 新增部分开始
    if (stmt.superclass != null) endScope();
    // 新增部分结束
    currentClass = enclosingClass;
```

> It's a minor optimization, but we only create the superclass environment if the class actually *has* a superclass. There's no point creating it when there isn't a superclass since there'd be no superclass to store in it anyway.

这是一个小优化，但是我们只在类真的有超类时才会创建超类环境。在没有超类的情况下，创建超类环境是没有意义的，因为无论如何里面都不会存储超类。

> With "super" defined in a scope chain, we are able to resolve the `super` expression itself.

在作用域链中定义`super`后，我们就能够分析`super`表达式了。

lox/Resolver.java，在 visitSetExpr()方法后添加：

```
  @Override
  public Void visitSuperExpr(Expr.Super expr) {
    resolveLocal(expr, expr.keyword);
    return null;
  }
```

> We resolve the `super` token exactly as if it were a variable. The resolution stores the number of hops along the environment chain that the interpreter needs to walk to find the environment where the superclass is stored.

我们把`super`标记当作一个变量进行分析。分析结果保存了解释器要在环境链上找到超类所在的环境需要的跳数。

> This code is mirrored in the interpreter. When we evaluate a subclass definition, we create a new environment.

这段代码在解释器中也有对应。当我们执行子类定义时，创建一个新环境。

lox/Interpreter.java，在 visitClassStmt()方法中添加：

```
        throw new RuntimeError(stmt.superclass.name,
            "Superclass must be a class.");
      }
    }

    environment.define(stmt.name.lexeme, null);
    // 新增部分开始
    if (stmt.superclass != null) {
      environment = new Environment(environment);
      environment.define("super", superclass);
    }
    // 新增部分结束
    Map<String, LoxFunction> methods = new HashMap<>();
```

> Inside that environment, we store a reference to the superclass—the actual LoxClass object for the superclass which we have now that we are in the runtime. Then we create the LoxFunctions for each method. Those will capture the current environment—the one where we just bound “super”—as their closure, holding on to the superclass like we need. Once that’s done, we pop the environment.

在这个环境中，我们保存指向超类的引用——即我们在运行时现在拥有的超类的实际LoxClass对象。然后我们为每个方法创建LoxFunction。这些函数将捕获当前环境（也就是我们刚刚绑定“super”的那个）作为其闭包，像我们需要的那样维系着超类。一旦这些完成，我们就弹出环境。

lox/Interpreter.java，在 visitClassStmt()方法中添加：

```
    LoxClass klass = new LoxClass(stmt.name.lexeme,
        (LoxClass)superclass, methods);
    // 新增部分开始
    if (superclass != null) {
      environment = environment.enclosing;
    }
    // 新增部分结束
    environment.assign(stmt.name, klass);
```

> We're ready to interpret `super` expressions themselves. There are a few moving parts, so we'll build this method up in pieces.

我们现在已经准备好解释`super`表达式了。这会分为很多部分，所以我们逐步构建这个方法。

lox/Interpreter.java，在 visitSetExpr()方法后添加：

```
  @Override
  public Object visitSuperExpr(Expr.Super expr) {
    int distance = locals.get(expr);
    LoxClass superclass = (LoxClass)environment.getAt(
        distance, "super");
  }
```

> First, the work we've been leading up to. We look up the surrounding class's superclass by looking up "super" in the proper environment.

首先，我们要做之前铺垫的工作。我们通过在适当环境中查找"super"来找到外围类的超类。

> When we access a method, we also need to bind `this` to the object the method is accessed from. In an expression like `doughnut.cook`, the object is whatever we get from evaluating `doughnut`. In a `super` expression like `super.cook`, the current object is implicitly the *same* current object that we're using. In other words, `this`. Even though we are looking up the *method* on the superclass, the *instance* is still `this`.

当我们访问方法时，还需要将`this`与访问该方法的对象进行绑定。在像`doughnut.cook`这样的表达式中，对象是我们通过对`doughnut`求值得到的内容。在像`super.cook`这样的`super`表达式中，当前对象隐式地与我们正使用的当前对象相同。换句话说，就是`this`。即使我们在超类中查找方法，*实例*仍然是`this`。

> Unfortunately, inside the `super` expression, we don't have a convenient node for the resolver to hang the number of hops to `this` on. Fortunately, we do control the layout of the environment chains. The environment where "this" is bound is always right inside the environment where we store "super".

不幸的是，在`super`表达式中，我们没有一个方便的节点可以让分析器将`this`对应的跳数保存起来。幸运的是，我们可以控制环境链的布局。绑定`this`的环境总是存储在保存`super`的环境中。

lox/Interpreter.java，在 visitSuperExpr()方法中添加：

```
    LoxClass superclass = (LoxClass)environment.getAt(
        distance, "super");
    // 新增部分开始
    LoxInstance object = (LoxInstance)environment.getAt(
        distance - 1, "this");
    // 新增部分结束
  }
```

> Offsetting the distance by one looks up “this” in that inner environment. I admit this isn’t the most elegant code, but it works.

将距离偏移1，在那个内部环境中查找“this”。我承认这个代码不是最优雅的，但是它是有效的。

> Now we’re ready to look up and bind the method, starting at the superclass.

现在我们准备查找并绑定方法，从超类开始。

lox/Interpreter.java，在 visitSuperExpr()方法中添加：

```
    LoxInstance object = (LoxInstance)environment.getAt(
        distance - 1, "this");
    // 新增部分开始
    LoxFunction method = superclass.findMethod(expr.method.lexeme);
    return method.bind(object);
    // 新增部分结束
  }
```

> This is almost exactly like the code for looking up a method of a get expression, except that we call `findMethod()` on the superclass instead of on the class of the current object.

这几乎与查找get表达式方法的代码完全一样，区别在于，我们是在超类上调用`findMethod()`，而不是在当前对象的类。

> That’s basically it. Except, of course, that we might *fail* to find the method. So we check for that too.

基本上就是这样了。当然，除了我们可能找不到方法之外。所以，我们要对其检查。

lox/Interpreter.java，在 visitSuperExpr()方法中添加：

```
    LoxFunction method = superclass.findMethod(expr.method.lexeme);
    // 新增部分开始
    if (method == null) {
      throw new RuntimeError(expr.method,
          "Undefined property '" + expr.method.lexeme + "'.");
    }
    // 新增部分结束
    return method.bind(object);
  }
```

> There you have it! Take that BostonCream example earlier and give it a try. Assuming you and I did everything right, it should fry it first, then stuff it with cream.

这就对了！试着运行一下前面那个BostonCream的例子。如果你我都做对了，它的结果应该是：

```
Fry until golden brown.
Pipe full of custard and coat with chocolate.
```

> 13.3.3 Invalid uses of super

13.3.3 super的无效使用

> As with previous language features, our implementation does the right thing when the user writes correct code, but we haven't bulletproofed the intepreter against bad code. In particular, consider:

像以前的语言特性一样，当用户写出正确的代码时，我们的语言实现也会做成正确的事情，但我们还没有在解释器中对错误代码进行防御。具体来说，考虑以下代码：

```
class Eclair {
  cook() {
    super.cook();
    print "Pipe full of crème pâtissière.";
  }
}
```

> This class has a `super` expression, but no superclass. At runtime, the code for evaluating `super` expressions assumes that "super" was successfully resolved and will be found in the environment. That's going to fail here because there is no surrounding environment for the superclass since there is no superclass. The JVM will throw an exception and bring our interpreter to its knees.

这个类中有一个`super`表达式，但是没有超类。在运行时，计算`super`表达式的代码假定`super`已经被成功分析，并且可以在环境中找到超类。但是在这里会失败，因为没有超类，也就没有超类对应的外围环境。JVM会抛出一个异常，我们的解释器也会因此崩溃。

> Heck, there are even simpler broken uses of super:

见鬼，还有更简单的`super`错误用法：

```
super.notEvenInAClass();
```

> We could handle errors like these at runtime by checking to see if the lookup of "super" succeeded. But we can tell statically—just by looking at the source code—that Eclair has no superclass and thus no `super` expression will work inside it. Likewise, in the second example, we know that the `super` expression is not even inside a method body.

我们可以在运行时通过检查"super"是否查找成功而处理此类错误。但是我们可以只通过查看源代码静态地知道，Eclair没有超类，因此也就没有`super`表达式可以在其中生效。同样的，在第二个例子中，我们知道`super`表达式甚至不在方法体内。

> Even though Lox is dynamically typed, that doesn't mean we want to defer *everything* to runtime. If the user made a mistake, we'd like to help them find it sooner rather than later. So we'll report these errors

> statically, in the resolver.

尽管Lox是动态类型的，但这并不意味着我们要将一切都推迟到运行时。如果用户犯了错误，我们希望能帮助他们尽早发现，所以我们会在分析器中静态地报告这些错误。

> First, we add a new case to the enum we use to keep track of what kind of class is surrounding the current code being visited.

首先，在我们用来追踪当前访问代码外围类的类型的枚举中添加一个新值。

lox/Resolver.java，在 ClassType枚举中添加代码，首先在上一行后面加","：

```
    NONE,
    CLASS,
    // 新增部分开始
    SUBCLASS
    // 新增部分结束
  }
```

> We'll use that to distinguish when we're inside a class that has a superclass versus one that doesn't. When we resolve a class declaration, we set that if the class is a subclass.

我们将用它来区分我们是否在一个有超类的类中。当我们分析一个类的声明时，如果该类是一个子类，我们就设置该值。

lox/Resolver.java，在 visitClassStmt()方法中添加：

```
    if (stmt.superclass != null) {
      // 新增部分开始
      currentClass = ClassType.SUBCLASS;
      // 新增部分结束
      resolve(stmt.superclass);
```

> Then, when we resolve a `super` expression, we check to see that we are currently inside a scope where that's allowed.

然后，当我们分析`super`表达式时，会检查当前是否在一个允许使用`super`表达式的作用域中。

lox/Resolver.java，在 visitSuperExpr()方法中添加：

```
  public Void visitSuperExpr(Expr.Super expr) {
    // 新增部分开始
    if (currentClass == ClassType.NONE) {
      Lox.error(expr.keyword,
          "Can't use 'super' outside of a class.");
    } else if (currentClass != ClassType.SUBCLASS) {
      Lox.error(expr.keyword,
          "Can't use 'super' in a class with no superclass.");
    }
    // 新增部分结束
    resolveLocal(expr, expr.keyword);
```

> If not—oopsie!—the user made a mistake.

如果不是，那就是用户出错了。

13.4 Conclusion

13.4 总结

> We made it! That final bit of error handling is the last chunk of code needed to complete our Java implementation of Lox. This is a real accomplishment and one you should be proud of. In the past dozen chapters and a thousand or so lines of code, we have learned and implemented . . .

我们成功了！最后的错误处理是完成Lox语言的Java实现所需的最后一块代码。这是一项真正的成就，你应该为此感到自豪。在过去的十几章和一千多行代码中，我们已经学习并实现了：

- tokens and lexing, 标记与词法
- abstract syntax trees, 抽象语法树
- recursive descent parsing, 递归下降分析
- prefix and infix expressions, 前缀、中缀表达式
- runtime representation of objects, 对象的运行时表示
- interpreting code using the Visitor pattern, 使用Visitor模式解释代码
- lexical scope, 词法作用域
- environment chains for storing variables, 保存变量的环境链
- control flow, 控制流
- functions with parameters, 有参函数
- closures, 闭包
- static variable resolution and error detection, 静态变量分析与错误检查
- classes, 类
- constructors, 构造函数
- fields, 字段
- methods, and finally, 方法
- inheritance. 继承

> We did all of that from scratch, with no external dependencies or magic tools. Just you and I, our respective text editors, a couple of collection classes in the Java standard library, and the JVM runtime.

所有这些都是我们从头开始做的，没有借助外部依赖和神奇工具。只有你和我，我们的文本编辑器，Java标准库中的几个集合类，以及JVM运行时。

> This marks the end of Part II, but not the end of the book. Take a break. Maybe write a few fun Lox programs and run them in your interpreter. (You may want to add a few more native methods for things like reading user input.) When you're refreshed and ready, we'll embark on our next adventure.

这标志着第二部分的结束，但不是这本书的结束。休息一下，也许可以编写几个Lox程序在你的解释器中运行一下（你可能需要添加一些本地方法来支持读取用户的输入等操作）。当你重新振作之后，我们将开始下一次冒险。

> ## CHALLENGES

习题

1. > Lox supports only *single inheritance*—a class may have a single superclass and that's the only way to reuse methods across classes. Other languages have explored a variety of ways to more freely reuse and share capabilities across classes: mixins, traits, multiple inheritance, virtual inheritance, extension methods, etc.

> If you were to add some feature along these lines to Lox, which would you pick and why? If you're feeling courageous (and you should be at this point), go ahead and add it.

Lox只支持*单继承*——一个类可以有一个超类，这是唯一跨类复用方法的方式。其它语言中已经探索出了各种方法来更自由地跨类重用和共享功能：mixins, traits, multiple inheritance, virtual inheritance, extension methods, 等等。

如果你要在Lox中添加一些类似的功能，你会选择哪种，为什么？如果你有勇气的话（这时候你应该有勇气了），那就去添加它。

2. > In Lox, as in most other object-oriented languages, when looking up a method, we start at the bottom of the class hierarchy and work our way up—a subclass's method is preferred over a superclass's. In order to get to the superclass method from within an overriding method, you use `super`.

   在Lox中，与其它大多数面向对象语言一样，当查找一个方法时，我们从类的底层开始向上查找——子类的方法优先于超类的方法。为了在覆盖方法中访问超类方法，你可以使用`super`。

   > The language BETA takes the opposite approach. When you call a method, it starts at the *top* of the class hierarchy and works *down*. A superclass method wins over a subclass method. In order to get to the subclass method, the superclass method can call `inner`, which is sort of like the inverse of `super`. It chains to the next method down the hierarchy.

   BEAT语言采用了相反的方法。当你调用一个方法时，它从类继承结构的顶层开始向下寻找。超类方法的优先级高于子类方法。为了访问子类的方法，超类方法可以调用`inner`，这有点像是`super`的反义词。它与继承层次结构中的下一级方法相连接。

   > The superclass method controls when and where the subclass is allowed to refine its behavior. If the superclass method doesn't call `inner` at all, then the subclass has no way of overriding or modifying the superclass's behavior.

   超类方法控制着子类何时何地可以改进其行为。如果超类方法根本没有调用`inner`，那么子类就无法覆盖或修改超类的行为。

   > Take out Lox's current overriding and `super` behavior and replace it with BETA's semantics. In short:

   去掉Lox目前的覆盖和`super`行为，用BEAT的语义来替换。简而言之：

   - > When calling a method on a class, prefer the method *highest* on the class's inheritance chain.

     当调用类上的方法时，优先选择类继承链中最高的方法。

   - > Inside the body of a method, a call to `inner` looks for a method with the same name in the nearest subclass along the inheritance chain between the class containing the `inner` and the class of `this`. If there is no matching method, the `inner` call does nothing.

     在方法体内部，`inner`调用会在继承链中包含`inner`的类和包含`this`的类之间，查找具有相同名称的最近的子类中的方法。如果没有匹配的方法，`inner`调用不做任何事情。

   > For example:

举例来说：

```
class Doughnut {
  cook() {
    print "Fry until golden brown.";
    inner();
    print "Place in a nice box.";
  }
}

class BostonCream < Doughnut {
  cook() {
    print "Pipe full of custard and coat with chocolate.";
  }
}

BostonCream().cook();
```

> This should print:

这应该输出：

```
Fry until golden brown.
Pipe full of custard and coat with chocolate.
Place in a nice box.
```

3. > In the chapter where I introduced Lox, I challenged you to come up with a couple of features you think the language is missing. Now that you know how to build an interpreter, implement one of those features.

   在介绍Lox的那一章，我让你想出几个你认为该语言缺少的功能。现在你知道了如何构建一个解释器，请实现其中的一个功能。

14.字节码块 Chunks of Bytecode

> If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.
>
> ——Donald Knuth

如果你发现你几乎把所有的时间都花在了理论上，那就开始把一些注意力转向实际的东西；这会提高你的理论水平。如果你发现你几乎把所有的时间都花在了实践上，那就开始把一些注意力转向理论上的东西；这将改善你的实践。（高德纳）

> We already have ourselves a complete implementation of Lox with jlox, so why isn't the book over yet? Part of this is because jlox relies on the JVM to do lots of things for us. If we want to understand how an interpreter works all the way down to the metal, we need to build those bits and pieces ourselves.

我们已经有了一个Lox 的完整实现jlox，那么为什么这本书还没有结束呢？部分原因是jlox依赖JVM为我们做很多事情^1。如果我们想要了解一个解释器是如何工作的，我们就需要自己构建这些零碎的东西。

> An even more fundamental reason that jlox isn't sufficient is that it's too damn slow. A tree-walk interpreter is fine for some kinds of high-level, declarative languages. But for a general-purpose, imperative language—even a "scripting" language like Lox—it won't fly. Take this little script:

jlox不够用的一个更根本的原因在于，它太慢了。树遍历解释器对于某些高级的声明式语言来说是不错的，但是对于通用的命令式语言——即使是Lox这样的"脚本"语言——这是行不通的。以下面的小脚本为例^2：

```
fun fib(n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}

var before = clock();
print fib(40);
var after = clock();
print after - before;
```

> On my laptop, that takes jlox about 72 seconds to execute. An equivalent C program finishes in half a second. Our dynamically typed scripting language is never going to be as fast as a statically typed language with manual memory management, but we don't need to settle for more than *two orders of magnitude* slower.

在我的笔记本电脑上，jlox大概需要72秒的时间来执行。一个等价的C程序在半秒内可以完成。我们的动态类型的脚本语言永远不可能像手动管理内存的静态类型语言那样快，但我们没必要满足于慢两个数量级以上的速度。

> We could take jlox and run it in a profiler and start tuning and tweaking hotspots, but that will only get us so far. The execution model—walking the AST—is fundamentally the wrong design. We can't micro-optimize that to the performance we want any more than you can polish an AMC Gremlin into an SR-71 Blackbird.

我们可以把jlox放在性能分析器中运行，并进行调优和调整热点，但这也只能到此为止了。它的执行模型（遍历AST）从根本上说就是一个错误的设计。我们无法将其微优化到我们想要的性能，就像你无法将AMC Gremlin打磨成SR-71 Blackbird一样。

> We need to rethink the core model. This chapter introduces that model, bytecode, and begins our new interpreter, clox.

我们需要重新考虑核心模型。本章将介绍这个模型——字节码，并开始我们的新解释器，clox。

14.1Bytecode?

14.1 字节码？

> In engineering, few choices are without trade-offs. To best understand why we're going with bytecode, let's stack it up against a couple of alternatives.

在工程领域，很少有选择是不需要权衡的。为了更好地理解我们为什么要使用字节码，让我们将它与几个备选方案进行比较。

> 14.1.1Why not walk the AST?

14.1.1 为什么不遍历AST？

> Our existing interpreter has a couple of things going for it:

我们目前的解释器有几个优点：

- > Well, first, we already wrote it. It’s done. And the main reason it’s done is because this style of interpreter is *really simple to implement*. The runtime representation of the code directly maps to the syntax. It’s virtually effortless to get from the parser to the data structures we need at runtime.

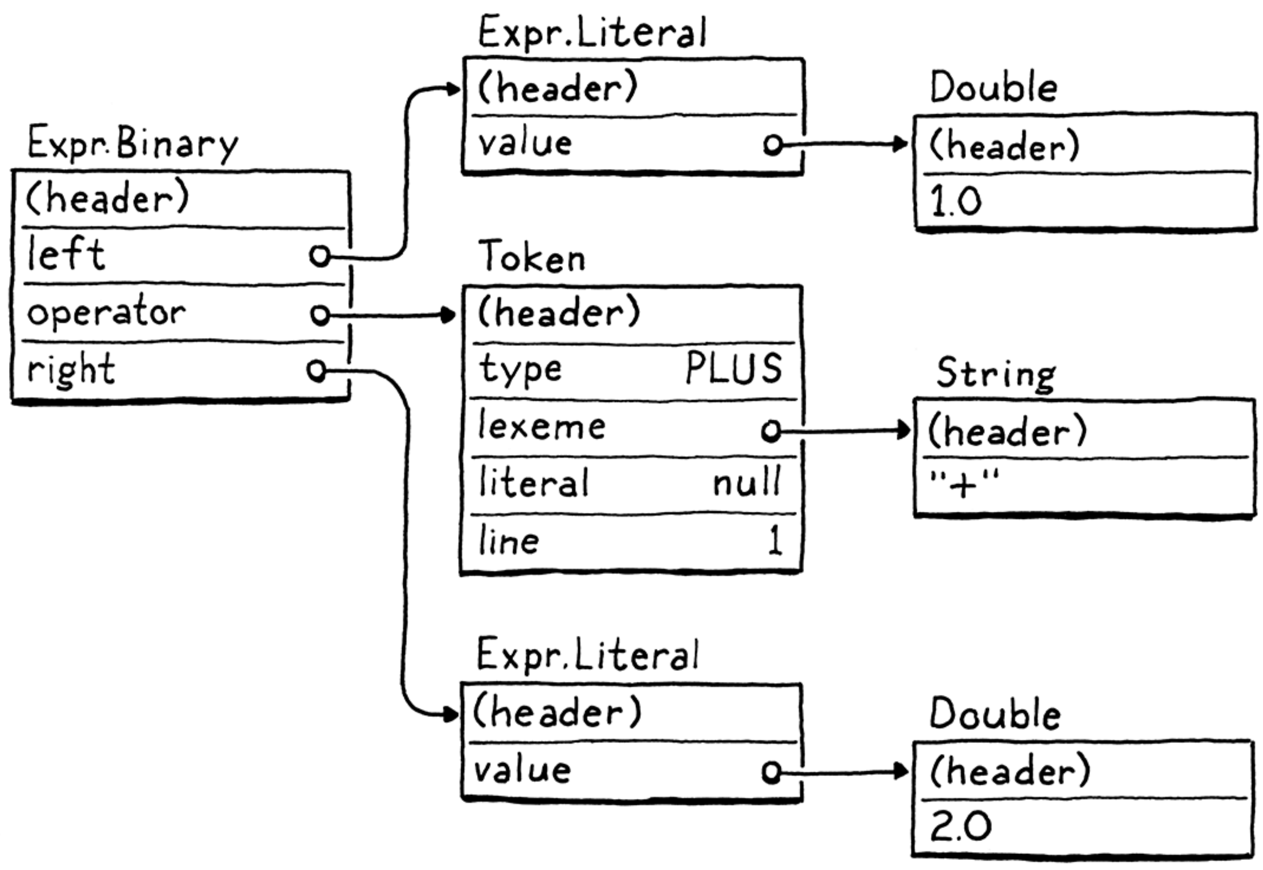
  嗯，首先我们已经写好了，它已经完成了。它能完成的主要原因是这种风格的解释器*实现起来非常简单*。代码的运行时表示直接映射到语法。从解析器到我们在运行时需要的数据结构，几乎都毫不费力。

- It’s *portable*. Our current interpreter is written in Java and runs on any platform Java supports. We could write a new implementation in C using the same approach and compile and run our language on basically every platform under the sun.

  它是可移植的。我们目前的解释器是使用Java编写的，可以在Java支持的任何平台上运行。我们可以用同样的方法在C语言中编写一个新的实现，并在世界上几乎所有平台上编译并运行我们的语言。

> Those are real advantages. But, on the other hand, it’s *not memory-efficient*. Each piece of syntax becomes an AST node. A tiny Lox expression like `1 + 2` turns into a slew of objects with lots of pointers between them, something like:

这些是真正的优势。但是，另一方面，它的内存使用效率不高。每一段语法都会变成一个AST节点。像`1+2`这样的Lox表达式会变成一连串的对象，对象之间有很多指针，就像^3：



> Each of those pointers adds an extra 32 or 64 bits of overhead to the object. Worse, sprinkling our data across the heap in a loosely connected web of objects does bad things for *spatial locality*.

每个指针都会给对象增加32或64比特的开销。更糟糕的是，将我们的数据散布在一个松散连接的对象网络中的堆上，会对空间局部性造成影响。

> Modern CPUs process data way faster than they can pull it from RAM. To compensate for that, chips have multiple layers of caching. If a piece of memory it needs is already in the cache, it can be loaded more quickly. We're talking upwards of 100 *times* faster.

现代CPU处理数据的速度远远超过它们从RAM中提取数据的速度。为了弥补这一点，芯片中有多层缓存。如果它需要的一块存储数据已经在缓存中，它就可以更快地被加载。我们谈论的是100倍以上的提速。

> How does data get into that cache? The machine speculatively stuffs things in there for you. Its heuristic is pretty simple. Whenever the CPU reads a bit of data from RAM, it pulls in a whole little bundle of adjacent bytes and stuffs them in the cache.

数据是如何进入缓存的？机器会推测性地为你把数据塞进去。它的启发式方法很简单。每当CPU从RAM中读取数据时，它就会拉取一块相邻的字节并放到缓存中。

> If our program next requests some data close enough to be inside that cache line, our CPU runs like a well-oiled conveyor belt in a factory. We *really* want to take advantage of this. To use the cache effectively, the way we represent code in memory should be dense and ordered like it's read.

如果我们的程序接下来请求一些在缓存行中的数据，那么我们的CPU就能像工厂里一条运转良好的传送带一样运行。我们真的很想利用这一点。为了有效的利用缓存，我们在内存中表示代码的方式应该像读取时一样紧密而有序。

> Now look up **at that tree. Those sub-objects could be *anywhere*. Every step the tree-walker takes where it follows a reference to a child node may step outside the bounds of the cache and force the CPU to stall until a new lump of data can be slurped in from RAM. Just the *overhead* of those tree nodes with all of their pointer fields and object headers tends to push o**bjects away from each other and out of the cache.

现在抬头看看那棵树。这些子对象可能在任何地方。树遍历器的每一步都会引用子节点，都可能会超出缓存的范围，并迫使CPU暂停，直到从RAM中拉取到新的数据块（才会继续执行）。仅仅是这些树形节点及其所有指针字段和对象头的开销，就会把对象彼此推离，并将其推出缓存区。

> Our AST walker has other overhead too around interface dispatch and the Visitor pattern, but the locality issues alone are enough to justify a better code representation.

我们的AST遍历器在接口调度和Visitor模式方面还有其它开销，但仅仅是局部性问题就足以证明使用更好的代码表示是合理的。

> 14.1.2Why not compile to native code?

14.1.2 为什么不编译成本地代码？

> If you want to go *real* fast, you want to get all of those layers of indirection out of the way. Right down to the metal. Machine code. It even *sounds* fast. *Machine code.*

如果你想真正快，就要摆脱所有的中间层，一直到最底层——机器码。听起来就很快，*机器码*。

> Compiling directly to the native instruction set the chip supports is what the fastest languages do. Targeting native code has been the most efficient option since way back in the early days when engineers actually handwrote programs in machine code.

最快的语言所做的是直接把代码编译为芯片支持的本地指令集。从早期工程师真正用机器码手写程序以来，以本地代码为目标一直是最有效的选择。

> If you've never written any machine code, or its slightly more human-palatable cousin assembly code before, I'll give you the gentlest of introductions. Native code is a dense series of operations, encoded directly in binary. Each instruction is between one and a few bytes long, and is almost mind-numbingly low level. "Move a value from this address to this register." "Add the integers in these two registers." Stuff like that.

如果你以前从来没有写过任何机器码，或者是它略微讨人喜欢的近亲汇编语言，那我给你做一个简单的介绍。本地代码是一系列密集的操作，直接用二进制编码。每条指令的长度都在一到几个字节之间，而且几乎是令人头疼的底层指令。“将一个值从这个地址移动到这个寄存器”“将这两个寄存器中的整数相加”，诸如此类。

> The CPU cranks through the instructions, decoding and executing each one in order. There is no tree structure like our AST, and control flow is handled by jumping from one point in the code directly to another. No indirection, no overhead, no unnecessary skipping around or chasing pointers.

通过解码和按顺序执行指令来操作CPU。没有像AST那样的树状结构，控制流是通过从代码中的一个点跳到另一个点来实现的。没有中间层，没有开销，没有不必要的跳转或指针寻址。

> Lightning fast, but that performance comes at a cost. First of all, compiling to native code ain't easy. Most chips in wide use today have sprawling Byzantine architectures with heaps of instructions that

> accreted over decades. They require sophisticated register allocation, pipelining, and instruction scheduling.

闪电般的速度，但这种性能是有代价的。首先，编译成本地代码并不容易。如今广泛使用的大多数芯片都有着庞大的拜占庭式架构，其中包含了几十年来积累的大量指令。它们需要复杂的寄存器分配、流水线和指令调度。

> And, of course, you've thrown portability out. Spend a few years mastering some architecture and that still only gets you onto *one* of the several popular instruction sets out there. To get your language on all of them, you need to learn all of their instruction sets and write a separate back end for each one.

当然，你可以把可移植性抛在一边。花费几年时间掌握一些架构，但这仍然只能让你接触到一些流行的指令集。为了让你的语言能在所有的架构上运行，你需要学习所有的指令集，并为每个指令集编写一个单独的后端^4。

> ### 14.1.3What is bytecode?

14.1.3 什么是字节码？

> Fix those two points in your mind. On one end, a tree-walk interpreter is simple, portable, and slow. On the other, native code is complex and platform-specific but fast. Bytecode sits in the middle. It retains the portability of a tree-walker—we won't be getting our hands dirty with assembly code in this book. It sacrifices *some* simplicity to get a performance boost in return, though not as fast as going fully native.

记住这两点。一方面，树遍历解释器简单、可移植，而且慢。另一方面，本地代码复杂且特定与平台，但是很快。字节码位于中间。它保留了树遍历型的可移植性——在本书中我们不会编写汇编代码，同时它牺牲了一些简单性来换取性能的提升，虽然没有完全的本地代码那么快。

> Structurally, bytecode resembles machine code. It's a dense, linear sequence of binary instructions. That keeps overhead low and plays nice with the cache. However, it's a much simpler, higher-level instruction set than any real chip out there. (In many bytecode formats, each instruction is only a single byte long, hence "bytecode".)

结构上讲，字节码类似于机器码。它是一个密集的、线性的二进制指令序列。这样可以保持较低的开销，并可以与高速缓存配合得很好。然而，它是一个更简单、更高级的指令集，比任何真正的芯片都要简单。（在很多字节码格式中，每条指令只有一个字节长，因此称为"字节码"）

> Imagine you're writing a native compiler from some source language and you're given carte blanche to define the easiest possible architecture to target. Bytecode is kind of like that. It's an idealized fantasy instruction set that makes your life as the compiler writer easier.

想象一下，你在用某种源语言编写一个本地编译器，并且你可以全权定义一个尽可能简单的目标架构。字节码就有点像这样，它是一个理想化的幻想指令集，可以让你作为编译器作者的生活更轻松。

> The problem with a fantasy architecture, of course, is that it doesn't exist. We solve that by writing an *emulator*—a simulated chip written in software that interprets the bytecode one instruction at a time. A *virtual machine (VM)*, if you will.

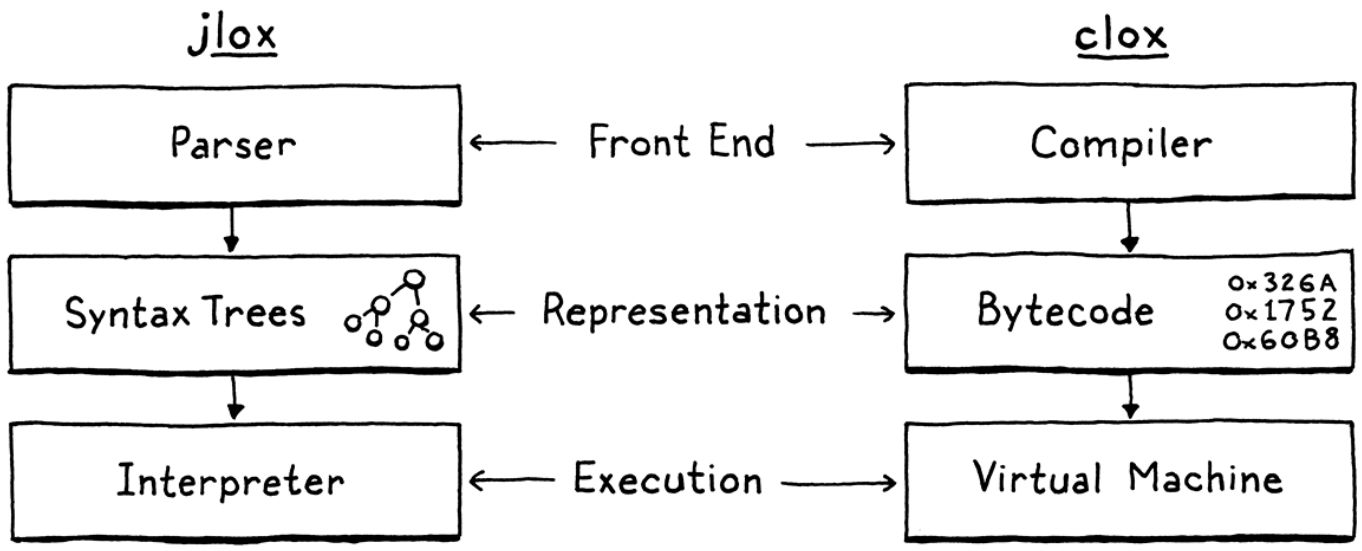
当然，幻想架构的问题在于它并不存在。我们提供编写模拟器来解决这个问题，这个模拟器是一个用软件编写的芯片，每次会解释字节码的一条指令。如果你愿意的话，可以叫它*虚拟机（VM）*。

> That emulation layer adds overhead, which is a key reason bytecode is slower than native code. But in return, it gives us portability. Write our VM in a language like C that is already supported on all the machines we care about, and we can run our emulator on top of any hardware we like.

模拟层增加了开销，这是字节码比本地代码慢的一个关键原因。但作为回报，它为我们提供了可移植性^5。用像C这样的语言来编写我们的虚拟机，它已经被我们所关心的所有机器所支持，这样我们就可以在任何我们喜欢的硬件上运行我们的模拟器。

> This is the path we'll take with our new interpreter, clox. We'll follow in the footsteps of the main implementations of Python, Ruby, Lua, OCaml, Erlang, and others. In many ways, our VM's design will parallel the structure of our previous interpreter:

这就是我们的新解释器clox要走的路。我们将追随Python、Ruby、Lua、OCaml、Erlang和其它主要语言实现的脚步。在许多方面，我们的VM设计将与之前的解释器结构并行。



> Of course, we won't implement the phases strictly in order. Like our previous interpreter, we'll bounce around, building up the implementation one language feature at a time. In this chapter, we'll get the skeleton of the application in place and create the data structures needed to store and represent a chunk of bytecode.

当然，我们不会严格按照顺序实现这些阶段。像我们之前的解释器一样，我们会反复地构建实现，每次只构建一种语言特性。在这一章中，我们将了解应用程序的框架，并创建用于存储和表示字节码块的数据结构。

14.2Getting Started

14.2 开始

> Where else to begin, but at `main()`? Fire up your trusty text editor and start typing.

除了`main()`还能从哪里开始呢？启动你的文本编辑器，开始输入。

main.c，创建新文件：

```
#include "common.h"

int main(int argc, const char* argv[]) {
  return 0;
}
```

> From this tiny seed, we will grow our entire VM. Since C provides us with so little, we first need to spend some time amending the soil. Some of that goes into this header:

从这颗小小的种子开始，我们将成长为整个VM。由于C提供给我们的东西太少，我们首先需要花费一些时间来培育土壤。其中一部分就在下面的header中。

common.h，创建新文件：

```
#ifndef clox_common_h
#define clox_common_h

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#endif
```

> There are a handful of types and constants we'll use throughout the interpreter, and this is a convenient place to put them. For now, it's the venerable `NULL`, `size_t`, the nice C99 Boolean `bool`, and explicit-sized integer types—`uint8_t` and friends.

在整个解释器中，我们会使用一些类型和常量，这是一个方便放置它们的地方。现在，它是古老的`NULL`、`size_t`, C99中的布尔类型`bool`，以及显式声明大小的整数类型——`uint8_t`和它的朋友们。

14.3Chunks of Instructions

14.3 指令块

> Next, we need a module to define our code representation. I've been using "chunk" to refer to sequences of bytecode, so let's make that the official name for that module.

接下来，我们需要一个模块来定义我们的代码表示形式。我一直使用"chunk"指代字节码序列，所以我们把它作为该模块的正式名称。

chunk.h，创建新文件：

```
#ifndef clox_chunk_h
#define clox_chunk_h

#include "common.h"

#endif
```

> In our bytecode format, each instruction has a one-byte **operation code** (universally shortened to **opcode**). That number controls what kind of instruction we're dealing with—add, subtract, look up variable, etc. We define those here:

在我们的字节码格式中，每个指令都有一个字节的**操作码**（通常简称为**opcode**）。这个数字控制我们要处理的指令类型——加、减、查找变量等。我们在这块定义这些：

chunk.h，添加代码：

```
#include "common.h"
// 新增部分开始
typedef enum {
  OP_RETURN,
} OpCode;
// 新增部分结束
#endif
```

> For now, we start with a single instruction, `OP_RETURN`. When we have a full-featured VM, this instruction will mean "return from the current function". I admit this isn't exactly useful yet, but we have to start somewhere, and this is a particularly simple instruction, for reasons we'll get to later.

现在，我们从一条指令`OP_RETURN`开始。当我们有一个全功能的VM时，这个指令意味着"从当前函数返回"。我承认这还不是完全有用，但是我们必须从某个地方开始下手，而这是一个特别简单的指令，原因我们会在后面讲到。

> ### 14.3.1 A dynamic array of instructions

14.3.1 指令动态数组

> Bytecode is a series of instructions. Eventually, we'll store some other data along with the instructions, so let's go ahead and create a struct to hold it all.

字节码是一系列指令。最终，我们会与指令一起存储一些其它数据，所以让我们继续创建一个结构体来保存所有这些数据。

chunk.h，在枚举 OpCode后添加：

```
} OpCode;
// 新增部分开始
typedef struct {
  uint8_t* code;
} Chunk;
// 新增部分结束
#endif
```

> At the moment, this is simply a wrapper around an array of bytes. Since we don't know how big the array needs to be before we start compiling a chunk, it must be dynamic. Dynamic arrays are one of my favorite data structures. That sounds like claiming vanilla is my favorite ice cream flavor, but hear me out. Dynamic arrays provide:

目前，这只是一个字节数组的简单包装。由于我们在开始编译块之前不知道数组需要多大，所以它必须是动态的。动态数组是我最喜欢的数据结构之一。这听起来就像是在说香草是我最喜爱的冰淇淋口味，但请听我说完。动态数组提供了：

- > Cache-friendly, dense storage

  缓存友好，密集存储

- > Constant-time indexed element lookup

  索引元素查找为常量时间复杂度

- > Constant-time appending to the end of the array

  数组末尾追加元素为常量时间复杂度

> Those features are exactly why we used dynamic arrays all the time in jlox under the guise of Java's ArrayList class. Now that we're in C, we get to roll our own. If you're rusty on dynamic arrays, the idea is pretty simple. In addition to the array itself, we keep two numbers: the number of elements in the array we have allocated ("capacity") and how many of those allocated entries are actually in use ("count").

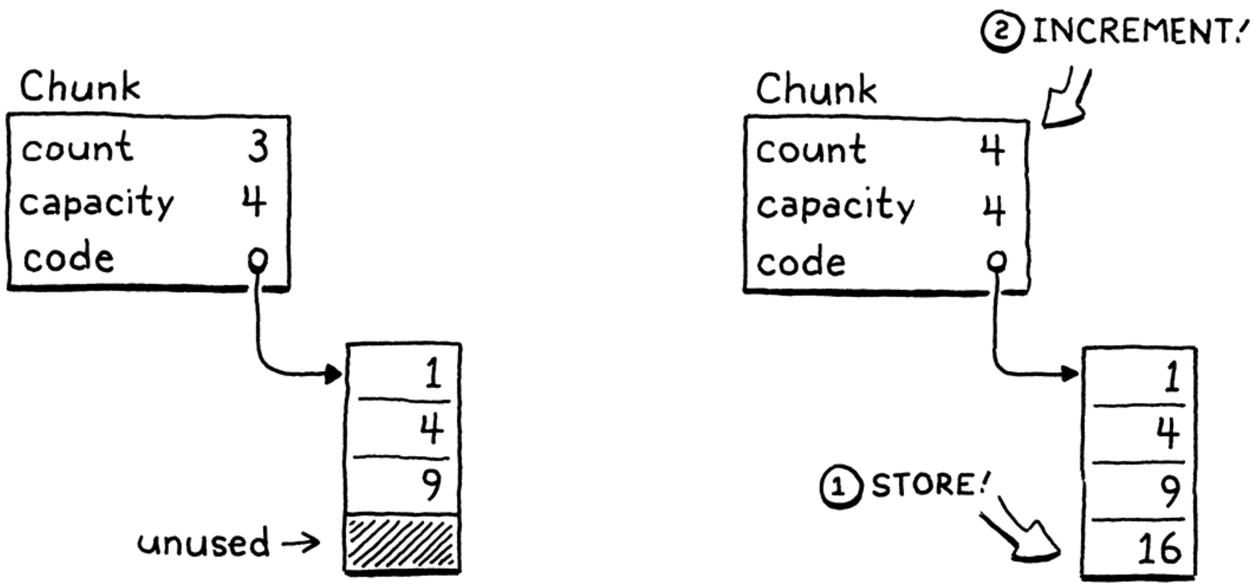
这些特性正是我们在jlox中以ArrayList类的名义一直使用动态数组的原因。现在我们在C语言中，可以推出我们自己的动态数组。如果你对动态数组不熟悉，其实这个想法非常简单。除了数组本身，我们还保留了两个数字：数组中已分配的元素数量（容量，capacity）和实际使用的已分配元数数量（计数，count）。

chunk.h，在结构体 Chunk 中添加代码：

```c
typedef struct {
  // 新增部分开始
  int count;
  int capacity;
  // 新增部分结束
  uint8_t* code;
} Chunk;
```

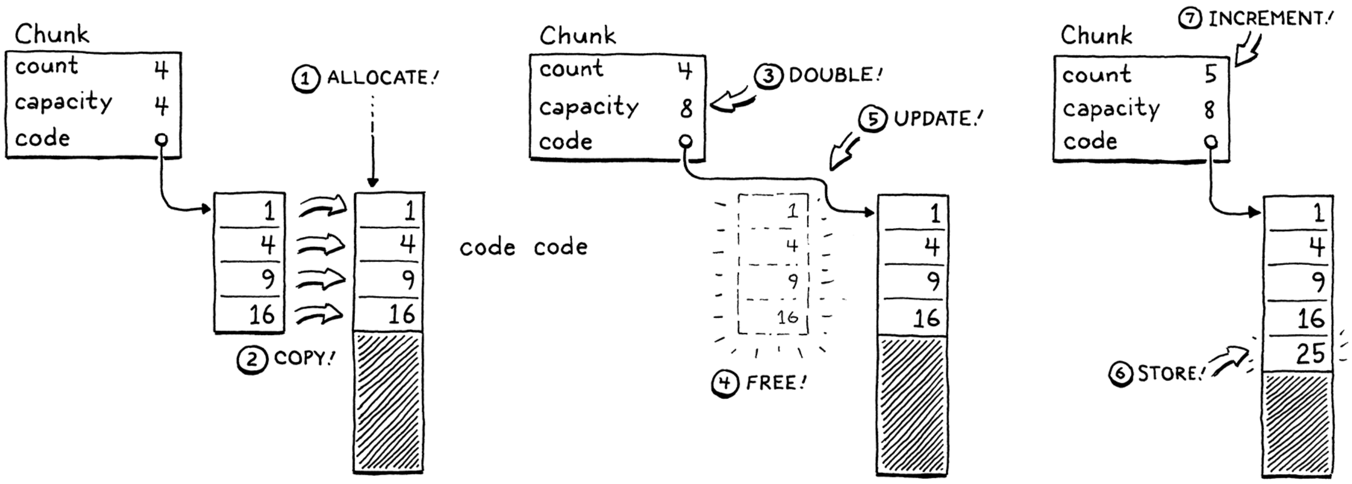
> When we add an element, if the count is less than the capacity, then there is already available space in the array. We store the new element right in there and bump the count.

当添加元素时，如果计数小于容量，那么数组中已有可用空间。我们将新元素直接存入其中，并修改计数值。



> If we have no spare capacity, then the process is a little more involved.

如果没有多余的容量，那么这个过程会稍微复杂一些。



1. > Allocate a new array with more capacity.

   分配一个容量更大的新数组^6。

2. > Copy the existing elements from the old array to the new one.

   将旧数组中的已有元素复制到新数组中。

3. > Store the new `capacity`.

   保存新的`capacity`。

4. > Delete the old array.

   删除旧数组。

5. > Update `code` to point to the new array.

   更新`code`指向新的数组。

6. > Store the element in the new array now that there is room.

现在有了空间，将元素存储在新数组中。

7. > Update the `count`.

更新`count`。

> We have our struct ready, so let's implement the functions to work with it. C doesn't have constructors, so we declare a function to initialize a new chunk.

我们的结构体已经就绪，现在我们来实现和它相关的函数。C语言没有构造函数，所以我们声明一个函数来初始化一个新的块。

chunk.h，在结构体Chunk后添加：

```
} Chunk;
// 新增部分开始
void initChunk(Chunk* chunk);
// 新增部分结束
#endif
```

> And implement it thusly:

并这样实现它：

chunk.c，创建新文件：

```
#include <stdlib.h>

#include "chunk.h"

void initChunk(Chunk* chunk) {
  chunk->count = 0;
  chunk->capacity = 0;
  chunk->code = NULL;
}
```

> The dynamic array starts off completely empty. We don't even allocate a raw array yet. To append a byte to the end of the chunk, we use a new function.

动态数组一开始是完全空的。我们甚至还没有分配原始数组。要将一个字节追加到块的末尾，我们使用一个新函数。

chunk.h，在initChunk()方法后添加：

```
void initChunk(Chunk* chunk);
// 新增部分开始
void writeChunk(Chunk* chunk, uint8_t byte);
// 新增部分结束
#endif
```

> This is where the interesting work happens.

这就是有趣的地方。

chunk.c，在 initChunk()方法后添加：

```
void writeChunk(Chunk* chunk, uint8_t byte) {
  if (chunk->capacity < chunk->count + 1) {
    int oldCapacity = chunk->capacity;
    chunk->capacity = GROW_CAPACITY(oldCapacity);
    chunk->code = GROW_ARRAY(uint8_t, chunk->code,
        oldCapacity, chunk->capacity);
  }

  chunk->code[chunk->count] = byte;
  chunk->count++;
}
```

> The first thing we need to do is see if the current array already has capacity for the new byte. If it doesn't, then we first need to grow the array to make room. (We also hit this case on the very first write when the array is `NULL` and `capacity` is 0.)

我们需要做的第一件事是查看当前数组是否已经有容纳新字节的容量。如果没有，那么我们首先需要扩充数组以腾出空间（当我们第一个写入时，数组为`NULL`并且`capacity`为0，也会遇到这种情况）

> To grow the array, first we figure out the new capacity and grow the array to that size. Both of those lower-level memory operations are defined in a new module.

要扩充数组，首先我们要算出新容量，然后将数组容量扩充到该大小。这两种低级别的内存操作都在一个新模块中定义。

chunk.c，添加代码：

```
#include "chunk.h"
// 新增部分开始
#include "memory.h"
// 新增部分结束
void initChunk(Chunk* chunk) {
```

> This is enough to get us started.

这就足够我们开始后面的事情了。

memory.h，创建新文件：

```
#ifndef clox_memory_h
#define clox_memory_h

#include "common.h"

#define GROW_CAPACITY(capacity) \
    ((capacity) < 8 ? 8 : (capacity) * 2)

#endif
```

> This macro calculates a new capacity based on a given current capacity. In order to get the performance we want, the important part is that it *scales* based on the old size. We grow by a factor of two, which is pretty typical. 1.5× is another common choice.

这个宏会根据给定的当前容量计算出新的容量。为了获得我们想要的性能，重要的部分就是基于旧容量大小进行扩展。我们以2的系数增长，这是一个典型的取值。1.5是另外一个常见的选择。

> We also handle when the current capacity is zero. In that case, we jump straight to eight elements instead of starting at one. That avoids a little extra memory churn when the array is very small, at the expense of wasting a few bytes on very small chunks.

我们还会处理当前容量为0的情况。在这种情况下，我们的容量直接跳到8，而不是从1开始^7。这就避免了在数组非常小的时候出现额外的内存波动，代价是在非常小的块中浪费几个字节。

> Once we know the desired capacity, we create or grow the array to that size using `GROW_ARRAY()`.

一旦我们知道了所需的容量，就可以使用`GROW_ARRAY()`创建或扩充数组到该大小。

memory.h，添加代码：

```
#define GROW_CAPACITY(capacity) ((capacity) < 8 ? 8 : (capacity) * 2)
// 新增部分开始
#define GROW_ARRAY(type, pointer, oldCount, newCount) \
    (type*)reallocate(pointer, sizeof(type) * (oldCount), \
        sizeof(type) * (newCount))

void* reallocate(void* pointer, size_t oldSize, size_t newSize);
// 新增部分结束
#endif
```

> This macro pretties up a function call to `reallocate()` where the real work happens. The macro itself takes care of getting the size of the array's element type and casting the resulting `void*` back to a pointer of the right type.

这个宏简化了对`reallocate()`函数的调用，真正的工作就是在其中完成的。宏本身负责获取数组元素类型的大小，并将生成的`void*`转换成正确类型的指针。

> This `reallocate()` function is the single function we'll use for all dynamic memory management in clox—allocating memory, freeing it, and changing the size of an existing allocation. Routing all of those

> operations through a single function will be important later when we add a garbage collector that needs to keep track of how much memory is in use.

这个`reallocate()`函数是我们将在clox中用于所有动态内存管理的唯一函数——分配内存，释放内存以及改变现有分配的大小。当我们稍后添加一个需要跟踪内存使用情况的垃圾收集器时，通过单个函数路由所有这些操作是很重要的。

> The two size arguments passed to `reallocate()` control which operation to perform:

传递给`reallocate()` 函数的两个大小参数控制了要执行的操作：

oldSize	newSize	Operation
0	Non-zero	Allocate new block. 分配新块
Non-zero	0	Free allocation. 释放已分配内存
Non-zero	Smaller than `oldSize`	Shrink existing allocation. 收缩已分配内存
Non-zero	Larger than `oldSize`	Grow existing allocation. 增加已分配内存

> That sounds like a lot of cases to handle, but here’s the implementation:

看起来好像有很多情况需要处理，但下面是其实现：

memory.c，创建新文件：

```
#include <stdlib.h>

#include "memory.h"

void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
  if (newSize == 0) {
    free(pointer);
    return NULL;
  }

  void* result = realloc(pointer, newSize);
  return result;
}
```

> When `newSize` is zero, we handle the deallocation case ourselves by calling `free()`. Otherwise, we rely on the C standard library’s `realloc()` function. That function conveniently supports the other three aspects of our policy. When `oldSize` is zero, `realloc()` is equivalent to calling `malloc()`.

当`newSize`为0时，我们通过调用`free()`来自己处理回收的情况。其它情况下，我们依赖于C标准库的`realloc()`函数。该函数可以方便地支持我们策略中的其它三个场景。当`oldSize`为0时，`realloc()` 等同于调用`malloc()`。

> The interesting cases are when both `oldSize` and `newSize` are not zero. Those tell `realloc()` to resize the previously allocated block. If the new size is smaller than the existing block of memory, it

> simply updates the size of the block and returns the same pointer you gave it. If the new size is larger, it attempts to grow the existing block of memory.

有趣的情况是当`oldSize`和`newSize`都不为0时。它们会告诉`realloc()`要调整之前分配的块的大小。如果新的大小小于现有的内存块，它就只是更新块的大小，并返回传入的指针。如果新块大小更大，它就会尝试增长现有的内存块^8。

> It can do that only if the memory after that block isn't already in use. If there isn't room to grow the block, `realloc()` instead allocates a *new* block of memory of the desired size, copies over the old bytes, frees the old block, and then returns a pointer to the new block. Remember, that's exactly the behavior we want for our dynamic array.

只有在该块之后的内存未被使用的情况下，才能这样做。如果没有空间支持块的增长，`realloc()`会分配一个所需大小的*新*的内存块，复制旧的字节，释放旧内存块，然后返回一个指向新内存块的指针。记住，这正是我们的动态数组想要的行为。

> Because computers are finite lumps of matter and not the perfect mathematical abstractions computer science theory would have us believe, allocation can fail if there isn't enough memory and `realloc()` will return `NULL`. We should handle that.

因为计算机是有限的物质块，而不是计算机科学理论所认为的完美的数学抽象，如果没有足够的内存，分配就会失败，`reealloc()`会返回`NULL`。我们应该解决这个问题。

memory.c，在 reallocate()方法中添加：

```
  void* result = realloc(pointer, newSize);
  // 新增部分开始
  if (result == NULL) exit(1);
  // 新增部分结束
  return result;
```

> There's not really anything *useful* that our VM can do if it can't get the memory it needs, but we at least detect that and abort the process immediately instead of returning a `NULL` pointer and letting it go off the rails later.

如果我们的VM不能得到它所需要的内存，那就做不了什么有用的事情，但我们至少可以检测这一点，并立即中止进程，而不是返回一个`NULL`指针，然后让程序运行偏离轨道。

> OK, we can create new chunks and write instructions to them. Are we done? Nope! We're in C now, remember, we have to manage memory ourselves, like in Ye Olden Times, and that means *freeing* it too.

好了，我们可以创建新的块并向其中写入指令。我们完成了吗？不！要记住，我们现在是在C语言中，我们必须自己管理内存，就像在《Ye Olden Times》中那样，这意味着我们也要*释放*内存。

chunk.h，在 initChunk()方法后添加：

```
void initChunk(Chunk* chunk);
// 新增部分开始
```

```
void freeChunk(Chunk* chunk);
// 新增部分结束
void writeChunk(Chunk* chunk, uint8_t byte);
```

实现为:

chunk.c，在 initChunk()方法后添加：

```
void freeChunk(Chunk* chunk) {
  FREE_ARRAY(uint8_t, chunk->code, chunk->capacity);
  initChunk(chunk);
}
```

> We deallocate all of the memory and then call `initChunk()` to zero out the fields leaving the chunk in a well-defined empty state. To free the memory, we add one more macro.

我们释放所有的内存，然后调用`initChunk()`将字段清零，使字节码块处于一个定义明确的空状态。为了释放内存，我们再添加一个宏。

memory.h，添加代码：

```
#define GROW_ARRAY(type, pointer, oldCount, newCount) \
    (type*)reallocate(pointer, sizeof(type) * (oldCount), \
        sizeof(type) * (newCount))
// 新增部分开始
#define FREE_ARRAY(type, pointer, oldCount) \
    reallocate(pointer, sizeof(type) * (oldCount), 0)
// 新增部分结束
void* reallocate(void* pointer, size_t oldSize, size_t newSize);
```

> Like `GROW_ARRAY()`, this is a wrapper around a call to `reallocate()`. This one frees the memory by passing in zero for the new size. I know, this is a lot of boring low-level stuff. Don't worry, we'll get a lot of use out of these in later chapters and will get to program at a higher level. Before we can do that, though, we gotta lay our own foundation.

与`GROW_ARRAY()`类似，这是对`reallocate()`调用的包装。这个函数通过传入0作为新的内存块大小，来释放内存。我知道，这是一堆无聊的低级别代码。别担心，在后面的章节中，我们会大量使用这些内容。但在此之前，我们必须先打好自己的基础。

> ## 14.4Disassembling Chunks

14.4 反汇编字节码块

> Now we have a little module for creating chunks of bytecode. Let's try it out by hand-building a sample chunk.

现在我们有一个创建字节码块的小模块。让我们手动构建一个样例字节码块来测试一下。

main.c，在 main()方法中添加：

```
int main(int argc, const char* argv[]) {
  // 新增部分开始
  Chunk chunk;
  initChunk(&chunk);
  writeChunk(&chunk, OP_RETURN);
  freeChunk(&chunk);
  // 新增部分结束
  return 0;
```

> Don’t forget the include.

不要忘了include。

main.c，添加代码：

```
#include "common.h"
// 新增部分开始
#include "chunk.h"
// 新增部分结束
int main(int argc, const char* argv[]) {
```

> Run that and give it a try. Did it work? Uh . . . who knows? All we’ve done is push some bytes around in memory. We have no human-friendly way to see what’s actually inside that chunk we made.

试着运行一下，它起作用了吗？额......谁知道呢。我们所做的只是在内存中存入一些字节。我们没有友好的方法来查看我们制作的字节码块中到底有什么。

> To fix this, we’re going to create a **disassembler**. An **assembler** is an old-school program that takes a file containing human-readable mnemonic names for CPU instructions like “ADD” and “MULT” and translates them to their binary machine code equivalent. A *dis*assembler goes in the other direction—given a blob of machine code, it spits out a textual listing of the instructions.

为了解决这个问题，我们要创建一个**反汇编程序。汇编程序**是一个老式程序，它接收一个文件，该文件中包含CPU指令（如 "ADD "和 "MULT"）的可读助记符名称，并将它们翻译成等价的二进制机器代码。反汇编程序则相反——给定一串机器码，它会返回指令的文本列表。

> We’ll implement something similar. Given a chunk, it will print out all of the instructions in it. A Lox *user* won’t use this, but we Lox *maintainers* will certainly benefit since it gives us a window into the interpreter’s internal representation of code.

我们将实现一个类似的模块。给定一个字节码块，它将打印出其中所有的指令。Lox用户不会使用它，但我们这些Lox的维护者肯定会从中受益，因为它给我们提供了一个了解解释器内部代码表示的窗口。

> In `main()`, after we create the chunk, we pass it to the disassembler.

在`main()`中，我们创建字节码块后，将其传入反汇编器。

main.c，在 main()方法中添加：

```
  initChunk(&chunk);
  writeChunk(&chunk, OP_RETURN);
  // 新增部分开始
  disassembleChunk(&chunk, "test chunk");
  // 新增部分结束
  freeChunk(&chunk);
```

> Again, we whip up yet another module.

我们又创建了另一个模块。

main.c，添加代码：

```
#include "chunk.h"
// 新增部分开始
#include "debug.h"
// 新增部分结束
int main(int argc, const char* argv[]) {
```

> Here’s that header:

下面是这个头文件：

debug.h，创建新文件：

```
#ifndef clox_debug_h
#define clox_debug_h

#include "chunk.h"

void disassembleChunk(Chunk* chunk, const char* name);
int disassembleInstruction(Chunk* chunk, int offset);

#endif
```

> In `main()`, we call `disassembleChunk()` to disassemble all of the instructions in the entire chunk. That’s implemented in terms of the other function, which just disassembles a single instruction. It shows up here in the header because we’ll call it from the VM in later chapters.

在`main()`方法中，我们调用`disassembleChunk()`来反汇编整个字节码块中的所有指令。这是用另一个函数实现的，该函数只反汇编一条指令。因为我们将在后面的章节中从VM中调用它，所以将它添加到头文件中。

> Here’s a start at the implementation file:

下面是简单的实现文件：

debug.c，创建新文件：

```c
#include <stdio.h>

#include "debug.h"

void disassembleChunk(Chunk* chunk, const char* name) {
  printf("== %s ==\n", name);

  for (int offset = 0; offset < chunk->count;) {
    offset = disassembleInstruction(chunk, offset);
  }
}
```

> To disassemble a chunk, we print a little header (so we can tell *which* chunk we're looking at) and then crank through the bytecode, disassembling each instruction. The way we iterate through the code is a little odd. Instead of incrementing `offset` in the loop, we let `disassembleInstruction()` do it for us. When we call that function, after disassembling the instruction at the given offset, it returns the offset of the *next* instruction. This is because, as we'll see later, instructions can have different sizes.

要反汇编一个字节码块，我们首先打印一个小标题（这样我们就知道正在看哪个字节码块），然后通过字节码反汇编每个指令。我们遍历代码的方式有点奇怪。我们没有在循环中增加`offset`，而是让`disassembleInstruction()`为我们做这个。当我们调用该函数时，在对给定偏移量的位置反汇编指令后，会返回 *下一条*指令的偏移量。这是因为，我们后面也会看到，指令可以有不同的大小。

> The core of the "debug" module is this function:

"debug"模块的核心是这个函数：

debug.c，在disassembleChunk()方法后添加：

```c
int disassembleInstruction(Chunk* chunk, int offset) {
  printf("%04d ", offset);

  uint8_t instruction = chunk->code[offset];
  switch (instruction) {
    case OP_RETURN:
      return simpleInstruction("OP_RETURN", offset);
    default:
      printf("Unknown opcode %d\n", instruction);
      return offset + 1;
  }
}
```

> First, it prints the byte offset of the given instruction—that tells us where in the chunk this instruction is. This will be a helpful signpost when we start doing control flow and jumping around in the bytecode.

首先，它会打印给定指令的字节偏移量——这能告诉我们当前指令在字节码块中的位置。当我们在字节码中实现控制流和跳转时，这将是一个有用的路标。

> Next, it reads a single byte from the bytecode at the given offset. That’s our opcode. We switch on that. For each kind of instruction, we dispatch to a little utility function for displaying it. On the off chance that the given byte doesn’t look like an instruction at all—a bug in our compiler—we print that too. For the one instruction we do have, `OP_RETURN`, the display function is:

接下来，它从字节码中的给定偏移量处读取一个字节。这也就是我们的操作码。我们根据该值做switch操作。对于每一种指令，我们都分派给一个小的工具函数来展示它。如果给定的字节看起来根本不像一条指令——这是我们编译器的一个错误——我们也要打印出来。对于我们目前仅有的一条指令`OP_RETURN`，对应的展示函数是：

debug.c，在 disassembleChunk()方法后添加：

```
static int simpleInstruction(const char* name, int offset) {
  printf("%s\n", name);
  return offset + 1;
}
```

> There isn’t much to a return instruction, so all it does is print the name of the opcode, then return the next byte offset past this instruction. Other instructions will have more going on.

return指令的内容不多，所以它所做的只是打印操作码的名称，然后返回该指令后的下一个字节偏移量。其它指令会有更多的内容。

> If we run our nascent interpreter now, it actually prints something:

如果我们现在运行我们的新解释器，它实际上会打印出来：

```
== test chunk ==
0000 OP_RETURN
```

> It worked! This is sort of the “Hello, world!” of our code representation. We can create a chunk, write an instruction to it, and then extract that instruction back out. Our encoding and decoding of the binary bytecode is working.

成功了！这有点像我们代码表示中的“Hello, world!”。我们可以创建一个字节码块，向其中写入一条指令，然后将该指令提取出来。我们对二进制字节码的编码和解码工作正常。

> ## 14.5Constants

14.5 常量

> Now that we have a rudimentary chunk structure working, let’s start making it more useful. We can store *code* in chunks, but what about *data*? Many values the interpreter works with are created at runtime as the result of operations.

现在我们有了一个基本的块结构，我们来让它变得更有用。我们可以在块中存储*代码*，但是*数据*呢？解释器中使用的很多值都是在运行时作为操作的结果创建的。

```
1 + 2;
```

> The value 3 appears nowhere in the code here. However, the literals 1 and 2 do. To compile that statement to bytecode, we need some sort of instruction that means "produce a constant" and those literal values need to get stored in the chunk somewhere. In jlox, the Expr.Literal AST node held the value. We need a different solution now that we don't have a syntax tree.

这里的代码中没有出现3这个值。但是，字面量1和2出现了。为了将该语句编译成字节码，我们需要某种指令，其含义是"生成一个常量"，而这些字母值需要存储在字节码块中的某个地方。在jlox中，Expr.Literal 这个AST节点中保存了这些值。因为我们没有语法树，现在我们需要一个不同的解决方案。

> ## 14.5.1 Representing values

14.5.1 表示值

> We won't be *running* any code in this chapter, but since constants have a foot in both the static and dynamic worlds of our interpreter, they force us to start thinking at least a little bit about how our VM should represent values.

在本章中我们不会运行任何代码，但是由于常量在解释器的静态和动态世界中都有涉足，这会迫使我们开始思考我们的虚拟机中应该如何表示数值。

> For now, we're going to start as simple as possible—we'll support only double-precision, floating-point numbers. This will obviously expand over time, so we'll set up a new module to give ourselves room to grow.

现在，我们尽可能从最简单的开始——只支持双精度浮点数。这种表示形式显然会逐渐扩大，所以我们将建立一个新的模块，给自己留出扩展的空间。

value.h，创建新文件：

```
#ifndef clox_value_h
#define clox_value_h

#include "common.h"

typedef double Value;

#endif
```

> This typedef abstracts how Lox values are concretely represented in C. That way, we can change that representation without needing to go back and fix existing code that passes around values.

这个类型定义抽象了Lox值在C语言中的具体表示方式。这样，我们就可以直接改变表示方法，而不需要回去修改现有的传递值的代码。

> Back to the question of where to store constants in a chunk. For small fixed-size values like integers, many instruction sets store the value directly in the code stream right after the opcode. These are called **immediate instructions** because the bits for the value are immediately after the opcode.

回到在字节码块中存储常量的问题。对于像整数这种固定大小的值，许多指令集直接将值存储在操作码之后的代码流中。这些指令被称为**即时指令**，因为值的比特位紧跟在操作码之后。

> That doesn't work well for large or variable-sized constants like strings. In a native compiler to machine code, those bigger constants get stored in a separate "constant data" region in the binary executable. Then, the instruction to load a constant has an address or offset pointing to where the value is stored in that section.

对于字符串这种较大的或可变大小的常量来说，这并不适用。在本地编译器的机器码中，这些较大的常量会存储在二进制可执行文件中的一个单独的"常量数据"区域。然后，加载常量的指令会有一个地址和偏移量，指向该值在区域中存储的位置。

> Most virtual machines do something similar. For example, the Java Virtual Machine associates a **constant pool** with each compiled class. That sounds good enough for clox to me. Each chunk will carry with it a list of the values that appear as literals in the program. To keep things simpler, we'll put *all* constants in there, even simple integers.

大多数虚拟机都会做类似的事。例如，Java虚拟机将常量池与每个编译后的类关联起来。我认为，这对于clox来说已经足够了。每个字节码块都会携带一个在程序中以字面量形式出现的值的列表。为简单起见，我们会把所有的常量都放进去，甚至包括简单的整数^9。

> ## 14.5.2Value arrays

14.5.2 值数组

> The constant pool is an array of values. The instruction to load a constant looks up the value by index in that array. As with our bytecode array, the compiler doesn't know how big the array needs to be ahead of time. So, again, we need a dynamic one. Since C doesn't have generic data structures, we'll write another dynamic array data structure, this time for Value.

常量池是一个值的数组。加载常量的指令根据数组中的索引查找该数组中的值。与字节码数组一样，编译器也无法提前知道这个数组需要多大。因此，我们需要一个动态数组。由于C语言没有通用数据结构，我们将编写另一个动态数组数据结构，这次存储的是Value。

value.h ：

```
typedef double Value;
// 新增部分开始
typedef struct {
  int capacity;
  int count;
  Value* values;
} ValueArray;
```

```
// 新增部分结束
#endif
```

> As with the bytecode array in Chunk, this struct wraps a pointer to an array along with its allocated capacity and the number of elements in use. We also need the same three functions to work with value arrays.

与Chunk中的字节码数组一样，这个结构体包装了一个指向数组的指针，以及其分配的容量和已使用元素的数量。我们也需要相同的三个函数来处理值数组。

value.h，在结构体 ValueArray后添加：

```
} ValueArray;
// 新增部分开始
void initValueArray(ValueArray* array);
void writeValueArray(ValueArray* array, Value value);
void freeValueArray(ValueArray* array);
// 新增部分结束
#endif
```

> The implementations will probably give you déjà vu. First, to create a new one:

对应的实现可能会让你有似曾相识的感觉。首先，创建一个新文件：

value.c，创建一个新文件：

```
#include <stdio.h>

#include "memory.h"
#include "value.h"

void initValueArray(ValueArray* array) {
  array->values = NULL;
  array->capacity = 0;
  array->count = 0;
}
```

> Once we have an initialized array, we can start adding values to it.

一旦我们有了初始化的数组，我们就可以开始向其中添加值。

value.c，在 initValueArray()方法后添加：

```
void writeValueArray(ValueArray* array, Value value) {
  if (array->capacity < array->count + 1) {
    int oldCapacity = array->capacity;
    array->capacity = GROW_CAPACITY(oldCapacity);
    array->values = GROW_ARRAY(Value, array->values,
                               oldCapacity, array->capacity);
  }

  array->values[array->count] = value;
  array->count++;
}
```

> The memory-management macros we wrote earlier do let us reuse some of the logic from the code array, so this isn't too bad. Finally, to release all memory used by the array:

我们之前写的内存管理宏确实让我们重用了代码数组中的一些逻辑，所以这并不是太糟糕。最后，释放数组所使用的所有内存：

value.c，在 writeValueArray()方法后添加：

```
void freeValueArray(ValueArray* array) {
  FREE_ARRAY(Value, array->values, array->capacity);
  initValueArray(array);
}
```

> Now that we have growable arrays of values, we can add one to Chunk to store the chunk's constants.

现在我们有了可增长的值数组，我们可以向Chunk中添加一个来保存字节码块中的常量值。

chunk.h，在结构体 Chunk 中添加：

```
  uint8_t* code;
  // 新增部分开始
  ValueArray constants;
  // 新增部分结束
} Chunk;
```

> Don't forget the include.

不要忘记include。

chunk.h，添加代码：

```
#include "common.h"
// 新增部分开始
#include "value.h"
// 新增部分结束
typedef enum {
```

> Ah, C, and its Stone Age modularity story. Where were we? Right. When we initialize a new chunk, we initialize its constant list too.

初始化新的字节码块时，我们也要初始化其常量值列表。

chunk.c，在 initChunk()方法中添加：

```
    chunk->code = NULL;
    // 新增部分开始
    initValueArray(&chunk->constants);
    // 新增部分结束
  }
```

> Likewise, we free the constants when we free the chunk.

同样地，我们在释放字节码块时，也需要释放常量值。

chunk.c，在 freeChunk()方法中添加：

```
    FREE_ARRAY(uint8_t, chunk->code, chunk->capacity);
    // 新增部分开始
    freeValueArray(&chunk->constants);
    // 新增部分结束
    initChunk(chunk);
```

> Next, we define a convenience method to add a new constant to the chunk. Our yet-to-be-written compiler could write to the constant array inside Chunk directly—it's not like C has private fields or anything—but it's a little nicer to add an explicit function.

接下来，我们定义一个便捷的方法来向字节码块中添加一个新常量。我们尚未编写的编译器可以在Chunk内部直接把常量值写入常量数组——它不像C语言那样有私有字段之类的东西——但是添加一个显式函数显然会更好一些。

chunk.h，在 writeChunk()方法后添加：

```
  void writeChunk(Chunk* chunk, uint8_t byte);
  // 新增部分开始
  int addConstant(Chunk* chunk, Value value);
  // 新增部分结束
  #endif
```

> Then we implement it.

然后我们实现它。

chunk.c，在 writeChunk()方法后添加：

```c
int addConstant(Chunk* chunk, Value value) {
  writeValueArray(&chunk->constants, value);
  return chunk->constants.count - 1;
}
```

> After we add the constant, we return the index where the constant was appended so that we can locate that same constant later.

在添加常量之后，我们返回追加常量的索引，以便后续可以定位到相同的常量。

> 14.5.3Constant instructions

14.5.3 常量指令

> We can *store* constants in chunks, but we also need to *execute* them. In a piece of code like:

我们可以将常量存储在字节码块中，但是我们也需要执行它们。在如下这段代码中：

```
print 1;
print 2;
```

> The compiled chunk needs to not only contain the values 1 and 2, but know *when* to produce them so that they are printed in the right order. Thus, we need an instruction that produces a particular constant.

编译后的字节码块不仅需要包含数值1和2，还需要知道何时生成它们，以便按照正确的顺序打印它们。因此，我们需要一种产生特定常数的指令。

chunk.h，在枚举OpCode中添加：

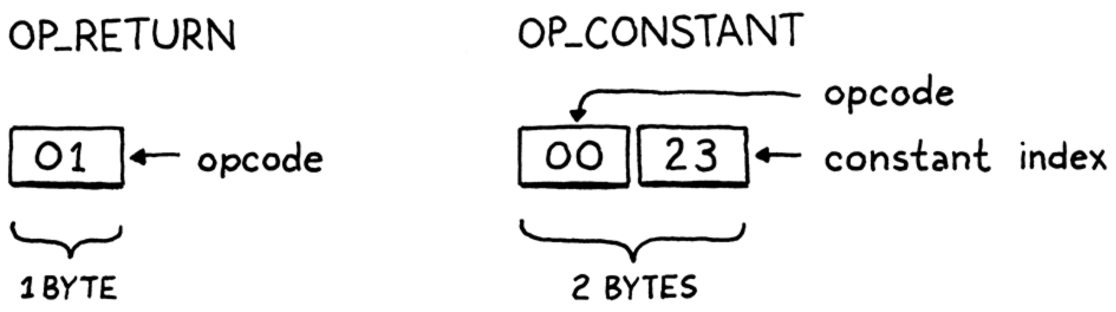
```c
typedef enum {
  // 新增部分开始
  OP_CONSTANT,
  // 新增部分结束
  OP_RETURN,
```

> When the VM executes a constant instruction, it "loads" the constant for use. This new instruction is a little more complex than `OP_RETURN`. In the above example, we load two different constants. A single bare opcode isn't enough to know *which* constant to load.

当VM执行常量指令时，它会"加载"常量以供使用^10。这个新指令比`OP_RETURN`要更复杂一些。在上面的例子中，我们加载了两个不同的常量。一个简单的操作码不足以知道要加载哪个常量。

> To handle cases like this, our bytecode—like most others—allows instructions to have **operands**. These are stored as binary data immediately after the opcode in the instruction stream and let us parameterize what the instruction does.

为了处理这样的情况，我们的字节码像大多数其它字节码一样，允许指令有**操作数**^11。这些操作数以二进制数据的形式存储在指令流的操作码之后，让我们对指令的操作进行参数化。



> Each opcode determines how many operand bytes it has and what they mean. For example, a simple operation like "return" may have no operands, where an instruction for "load local variable" needs an operand to identify which variable to load. Each time we add a new opcode to clox, we specify what its operands look like—its **instruction format**.

每个操作码会定义它有多少操作数以及各自的含义。例如，一个像"return"这样简单的操作可能没有操作数，而一个"加载局部变量"的指令需要一个操作数来确定要加载哪个变量。每次我们向clox添加一个新的操作码时，我们都会指定它的操作数是什么样子的——即它的**指令格式**。

> In this case, `OP_CONSTANT` takes a single byte operand that specifies which constant to load from the chunk's constant array. Since we don't have a compiler yet, we "hand-compile" an instruction in our test chunk.

在这种情况下，`OP_CONSTANT`会接受一个单字节的操作数，该操作数指定从块的常量数组中加载哪个常量。由于我们还没有编译器，所以我们在测试字节码块中"手动编译"一个指令。

main.c，在 main()方法中添加：

```
  initChunk(&chunk);
  // 新增部分开始
  int constant = addConstant(&chunk, 1.2);
  writeChunk(&chunk, OP_CONSTANT);
  writeChunk(&chunk, constant);
  // 新增部分结束
  writeChunk(&chunk, OP_RETURN);
```

> We add the constant value itself to the chunk's constant pool. That returns the index of the constant in the array. Then we write the constant instruction, starting with its opcode. After that, we write the one-byte constant index operand. Note that `writeChunk()` can write opcodes or operands. It's all raw bytes as far as that function is concerned.

我们将常量值添加到字节码块的常量池中。这会返回常量在数组中的索引。然后我们写常量操作指令，从操作码开始。之后，我们写入一字节的常量索引操作数。注意， `writeChunk()` 可以写操作码或操作数。对于该函数而言，它们都是原始字节。

> If we try to run this now, the disassembler is going to yell at us because it doesn't know how to decode the new instruction. Let's fix that.

如果我们现在尝试运行上面的代码，反汇编器会遇到问题，因为它不知道如何解码新指令。让我们来修复这个问题。

debug.c，在 disassembleInstruction()方法中添加：

```
  switch (instruction) {
    // 新增部分开始
    case OP_CONSTANT:
      return constantInstruction("OP_CONSTANT", chunk, offset);
    // 新增部分结束
    case OP_RETURN:
```

> This instruction has a different instruction format, so we write a new helper function to disassemble it.

这条指令的格式有所不同，所以我们编写一个新的辅助函数来对其反汇编。

debug.c，在 disassembleChunk()方法后添加：

```
static int constantInstruction(const char* name, Chunk* chunk,
                               int offset) {
  uint8_t constant = chunk->code[offset + 1];
  printf("%-16s %4d '", name, constant);
  printValue(chunk->constants.values[constant]);
  printf("'\n");
}
```

> There's more going on here. As with `OP_RETURN`, we print out the name of the opcode. Then we pull out the constant index from the subsequent byte in the chunk. We print that index, but that isn't super useful to us human readers. So we also look up the actual constant value—since constants *are* known at compile time after all—and display the value itself too.

这里要做的事情更多一些。与`OP_ETURN`一样，我们会打印出操作码的名称。然后，我们从该字节码块的后续字节中获取常量索引。我们打印出这个索引值，但是这对于我们人类读者来说并不十分有用。所以，我们也要查找实际的常量值——因为常量毕竟是在编译时就知道的——并将这个值也展示出来。

> This requires some way to print a clox Value. That function will live in the "value" module, so we include that.

这就需要一些方法来打印clox中的一个Value。这个函数放在"value"模块中，所以我们要将其include。

debug.c，新增代码：

```
#include "debug.h"
// 新增部分开始
#include "value.h"
// 新增部分结束
void disassembleChunk(Chunk* chunk, const char* name) {
```

> Over in that header, we declare:

在这个头文件中，我们声明：

value.h，在 freeValueArray()方法后添加：

```
void freeValueArray(ValueArray* array);
// 新增部分开始
void printValue(Value value);
// 新增部分结束
#endif
```

> And here’s an implementation:

下面是对应的实现：

value.c，在 freeValueArray()方法后添加：

```
void printValue(Value value) {
  printf("%g", value);
}
```

> Magnificent, right? As you can imagine, this is going to get more complex once we add dynamic typing to Lox and have values of different types.

很壮观，是吧？你可以想象，一旦我们在Lox中加入动态类型，并且包含了不同类型的值，这部分将会变得更加复杂。

> Back in `constantInstruction()`, the only remaining piece is the return value.

回到`constantInstruction()`中，唯一剩下的部分就是返回值。

debug.c，在 constantInstruction()方法中添加：

```
  printf("'\n");
  // 新增部分开始
  return offset + 2;
  // 新增部分结束
}
```

> Remember that `disassembleInstruction()` also returns a number to tell the caller the offset of the beginning of the *next* instruction. Where `OP_RETURN` was only a single byte, `OP_CONSTANT` is two—one for the opcode and one for the operand.

记住，`disassembleInstruction()`也会返回一个数字，告诉调用方 *下一条*指令的起始位置的偏移量。`OP_RETURN`只有一个字节，而`OP_CONSTANT`有两个字节——一个是操作码，一个是操作数。

14.6 Line Information

14.6 行信息

> Chunks contain almost all of the information that the runtime needs from the user's source code. It's kind of crazy to think that we can reduce all of the different AST classes that we created in jlox down to an array of bytes and an array of constants. There's only one piece of data we're missing. We need it, even though the user hopes to never see it.

字节码块中几乎包含了运行时需要从用户源代码中获取的所有信息。想到我们可以把jlox中不同的AST类减少到一个字节数组和一个常量数组，这实在有一点疯狂。我们只缺少一个数据。我们需要它，尽管用户希望永远不会看到它。

> When a runtime error occurs, we show the user the line number of the offending source code. In jlox, those numbers live in tokens, which we in turn store in the AST nodes. We need a different solution for clox now that we've ditched syntax trees in favor of bytecode. Given any bytecode instruction, we need to be able to determine the line of the user's source program that it was compiled from.

当运行时错误发生时，我们会向用户显示出错的源代码的行号。在jlox中，这些数字保存在词法标记中，而我们又将词法标记存储在AST节点中。既然我们已经抛弃了语法树而采用了字节码，我们就需要为clox提供不同的解决方案。对于任何字节码指令，我们需要能够确定它是从用户源代码的哪一行编译出来的。

> There are a lot of clever ways we could encode this. I took the absolute simplest approach I could come up with, even though it's embarrassingly inefficient with memory. In the chunk, we store a separate array of integers that parallels the bytecode. Each number in the array is the line number for the corresponding byte in the bytecode. When a runtime error occurs, we look up the line number at the same index as the current instruction's offset in the code array.

我们有很多聪明的方法可以对此进行编码。我采取了我能想到的绝对最简单的方法，尽管这种方法的内存效率低得令人发指^12。在字节码块中，我们存储一个单独的整数数组，该数组与字节码平级。数组中的每个数字都是字节码中对应字节所在的行号。当发生运行时错误时，我们根据当前指令在代码数组中的偏移量查找对应的行号。

> To implement this, we add another array to Chunk.

为了实现这一点，我们向Chunk中添加另一个数组。

chunk.h，在结构体 Chunk 中添加：

```
  uint8_t* code;
  // 新增部分开始
  int* lines;
  // 新增部分结束
  ValueArray constants;
```

> Since it exactly parallels the bytecode array, we don't need a separate count or capacity. Every time we touch the code array, we make a corresponding change to the line number array, starting with initialization.

由于它与字节码数组完全平行，我们不需要单独的计数值和容量值。每次我们访问代码数组时，也会对行号数组做相应的修改，从初始化开始。

chunk.c，在 initChunk()方法中添加：

```
  chunk->code = NULL;
  // 新增部分开始
  chunk->lines = NULL;
  // 新增部分结束
  initValueArray(&chunk->constants);
```

> And likewise deallocation:

回收也是类似的：

chunk.c，在 freeChunk()中添加：

```
  FREE_ARRAY(uint8_t, chunk->code, chunk->capacity);
  // 新增部分开始
  FREE_ARRAY(int, chunk->lines, chunk->capacity);
  // 新增部分结束
  freeValueArray(&chunk->constants);
```

> When we write a byte of code to the chunk, we need to know what source line it came from, so we add an extra parameter in the declaration of `writeChunk()`.

当我们向块中写入一个代码字节时，我们需要知道它来自哪个源代码行，所以我们在`writeChunk()`的声明中添加一个额外的参数。

chunk.h，在 writeChunk()函数中替换一行：

```
void freeChunk(Chunk* chunk);
// 替换部分开始
void writeChunk(Chunk* chunk, uint8_t byte, int line);
// 替换部分结束
int addConstant(Chunk* chunk, Value value);
```

> And in the implementation:

然后在实现中修改：

chunk.c，在 writeChunk()函数中替换一行：

```
// 替换部分开始
void writeChunk(Chunk* chunk, uint8_t byte, int line) {
```

```
// 替换部分结束
  if (chunk->capacity < chunk->count + 1) {
```

> When we allocate or grow the code array, we do the same for the line info too.

当我们分配或扩展代码数组时，我们也要对行信息进行相同的处理。

chunk.c，在 writeChunk()方法中添加：

```
    chunk->code = GROW_ARRAY(uint8_t, chunk->code,
        oldCapacity, chunk->capacity);
    // 新增部分开始
    chunk->lines = GROW_ARRAY(int, chunk->lines,
        oldCapacity, chunk->capacity);
    // 新增部分结束
  }
```

> Finally, we store the line number in the array.

最后，我们在数组中保存行信息。

chunk.c，在 writeChunk()方法中添加：

```
  chunk->code[chunk->count] = byte;
  // 新增部分开始
  chunk->lines[chunk->count] = line;
  // 新增部分结束
  chunk->count++;
```

> ### 14.6.1Disassembling line information

14.6.1 反汇编行信息

> Alright, let's try this out with our little, uh, artisanal chunk. First, since we added a new parameter to `writeChunk()`, we need to fix those calls to pass in some—arbitrary at this point—line number.

好吧，让我们手动编译一个小的字节码块测试一下。首先，由于我们向`writeChunk()`添加了一个新参数，我们需要修改一下该方法的调用，向其中添加一些行号（这里可以随意选择行号值）。

> *main.c*，在 *main*()方法中替换四行：

```
  int constant = addConstant(&chunk, 1.2);
  // 替换部分开始
  writeChunk(&chunk, OP_CONSTANT, 123);
  writeChunk(&chunk, constant, 123);

  writeChunk(&chunk, OP_RETURN, 123);
  // 替换部分结束
  disassembleChunk(&chunk, "test chunk");
```

> Once we have a real front end, of course, the compiler will track the current line as it parses and pass that in.

当然，一旦我们有了真正的前端，编译器会在解析时跟踪当前行，并将其传入字节码中。

> Now that we have line information for every instruction, let’s put it to good use. In our disassembler, it’s helpful to show which source line each instruction was compiled from. That gives us a way to map back to the original code when we’re trying to figure out what some blob of bytecode is supposed to do. After printing the offset of the instruction—the number of bytes from the beginning of the chunk—we show its source line.

现在我们有了每条指令的行信息，让我们好好利用它吧。在我们的反汇编程序中，展示每条指令是由哪一行源代码编译出来的是很有帮助的。当我们试图弄清楚某些字节码应该做什么时，这给我们提供了一种方法来映射回原始代码。在打印了指令的偏移量之后——从字节码块起点到当前指令的字节数——我们也展示它在源代码中的行号。

debug.c，在 disassembleInstruction()方法中添加：

```
int disassembleInstruction(Chunk* chunk, int offset) {
  printf("%04d ", offset);
  // 新增部分开始
  if (offset > 0 &&
      chunk->lines[offset] == chunk->lines[offset - 1]) {
    printf("   | ");
  } else {
    printf("%4d ", chunk->lines[offset]);
  }
  // 新增部分结束
  uint8_t instruction = chunk->code[offset];
```

> Bytecode instructions tend to be pretty fine-grained. A single line of source code often compiles to a whole sequence of instructions. To make that more visually clear, we show a `|` for any instruction that comes from the same source line as the preceding one. The resulting output for our handwritten chunk looks like:

字节码指令往往是非常细粒度的。一行源代码往往可以编译成一个完整的指令序列。为了更直观地说明这一点，我们在与前一条指令来自同一源码行的指令前面显示一个“|”。我们的手写字节码块的输出结果如下所示：

```
== test chunk ==
0000  123 OP_CONSTANT         0 '1.2'
0002    | OP_RETURN
```

> We have a three-byte chunk. The first two bytes are a constant instruction that loads 1.2 from the chunk’s constant pool. The first byte is the `OP_CONSTANT` opcode and the second is the index in the

> constant pool. The third byte (at offset 2) is a single-byte return instruction.

我们有一个三字节的块。前两个字节是一个常量指令，从该块的常量池中加载1.2。第一个字节是`OP_CONSTANT`字节码，第二个是在常量池中的索引。第三个字节（偏移量为2）是一个单字节的返回指令。

> In the remaining chapters, we will flesh this out with lots more kinds of instructions. But the basic structure is here, and we have everything we need now to completely represent an executable piece of code at runtime in our virtual machine. Remember that whole family of AST classes we defined in jlox? In clox, we've reduced that down to three arrays: bytes of code, constant values, and line information for debugging.

在接下来的章节中，我们将用更多种类的指令来充实这个结构。但是基本结构已经在这里了，我们现在拥有了所需要的一切，可以在虚拟机运行时完全表示一段可执行的代码。还记得我们在jlox中定义的整个AST类族吗？在clox中，我们把它减少到了三个数组：代码字节数组，常量值数组，以及用于调试的行信息。

> This reduction is a key reason why our new interpreter will be faster than jlox. You can think of bytecode as a sort of compact serialization of the AST, highly optimized for how the interpreter will deserialize it in the order it needs as it executes. In the next chapter, we will see how the virtual machine does exactly that.

这种减少是我们的新解释器比jlox更快的一个关键原因。你可以把字节码看作是AST的一种紧凑的序列化，并且解释器在执行时按照需要对其反序列化的方式进行了高度优化。在下一章中，我们将会看到虚拟机是如何做到这一点的。

^5:最早的字节码格式之一是p-code，是为Niklaus Wirth的Pascal语言开发的。你可能会认为一个运行在15MHz的PDP-11无法承担模拟虚拟机的开销。但在当时，计算机正处于寒武纪大爆发时期，每天都有新的架构出现。跟上最新的芯片要比从某个芯片中压榨出最大性能更有价值。这就是为什么p-code中的"p"指的不是"Pascal"而是"可移植性Portable"。 ^6:增长数组时会复制现有元素，使得追加元素的复杂度看起来像是O(n)，而不是O(1)。但是，你只需要在某些追加操作中执行这个操作步骤。大多数时候，已有多余的容量，所以不需要复制。要理解这一点，我们需要进行摊销分析。这表明，只要我们把数组大小增加到当前大小的倍数，当我们把一系列追加操作的成本平均化时，每次追加都是O(1)。 ^7:我在这本书中选择了数字8，有些随意。大多数动态数组实现都有一个这样的最小阈值。挑选这个值的正确方法是根据实际使用情况进行分析，看看那个常数能在额外增长和浪费的空间之间做出最佳的性能权衡。 ^8:既然我们传入的只是一个指向内存第一个字节的裸指针，那么"更新"块的大小意味着什么呢？在内部，内存分配器为堆分配的每个内存块都维护了额外的簿记信息，包括它的大小。给定一个指向先前分配的内存的指针，它就可以找到这个簿记信息，为了能干净地释放内存，这是必需的。`realloc()`所更新的正是这个表示大小的元数据。许多`malloc()`的实现将分配的大小存储在返回地址之前的内存中。 ^9:除了需要两种常量指令（一种用于即时值，一种用于常量表中的常量）之外，即时指令还要求我们考虑对齐、填充和字节顺序的问题。如果你尝试在一个奇数地址填充一个4字节的整数，有些架构中会出错。 ^10:我这里对于"加载"或"产生"一个常量的含义含糊其辞，因为我们还没有学到虚拟机在运行时是如何执行的代码的。关于这一点，你必须等到（或者直接跳到）下一章。 ^11:字节码指令的操作数与传递给算术操作符的操作数不同。当我们讲到表达式时，你会看到算术操作数的值是被单独跟踪的。指令操作数是一个较低层次的概念，它可以修改字节码指令本身的行为方式。 ^12: 这种脑残的编码至少做对了一件事：它将行信息保存一个单独的数组中，而不是将其编入字节码本身中。由于行信息只在运行时出现错误时才使用，我们不希望它在指令之间占用CPU缓存中的宝贵空间，而且解释器在跳过行数获取它所关心的操作码和操作数时，会造成更多的缓存丢失。

> ## CHALLENGES

习题

1. > Our encoding of line information is hilariously wasteful of memory. Given that a series of instructions often correspond to the same source line, a natural solution is something akin to run-length encoding of the line numbers.

   我们对行信息的编码非常浪费内存。鉴于一系列指令通常对应于同一源代码行，一个自然的解决方案是对行号进行类似游程编码的操作。

   > Devise an encoding that compresses the line information for a series of instructions on the same line. Change `writeChunk()` to write this compressed form, and implement a `getLine()` function that, given the index of an instruction, determines the line where the instruction occurs.

   设计一个编码方式，压缩同一行上一系列指令的行信息。修改`writeChunk()`以写入该压缩形式，并实现一个`getLine()`函数，给定一条指令的索引，确定该指令所在的行。

   > *Hint: It's not necessary for `getLine()` to be particularly efficient. Since it is called only when a runtime error occurs, it is well off the critical path where performance matters.*

   提示：`getLine()`不一定要特别高效。因为它只在出现运行时错误时才被调用，所以在它并不是影响性能的关键因素。

2. > Because `OP_CONSTANT` uses only a single byte for its operand, a chunk may only contain up to 256 different constants. That's small enough that people writing real-world code will hit that limit. We could use two or more bytes to store the operand, but that makes *every* constant instruction take up more space. Most chunks won't need that many unique constants, so that wastes space and sacrifices some locality in the common case to support the rare case.

   因为`OP_CONSTANT`只使用一个字节作为操作数，所以一个块最多只能包含256个不同的常数。这已经够小了，用户在编写真正的代码时很容易会遇到这个限制。我们可以使用两个或更多字节来存储操作数，但这会使*每*个常量指令占用更多的空间。大多数字节码块都不需要那么多独特的常量，所以这就浪费了空间，并牺牲了一些常规情况下的局部性来支持罕见场景。

   > To balance those two competing aims, many instruction sets feature multiple instructions that perform the same operation but with operands of different sizes. Leave our existing one-byte `OP_CONSTANT` instruction alone, and define a second `OP_CONSTANT_LONG` instruction. It stores the operand as a 24-bit number, which should be plenty.

   为了平衡这两个相互冲突的目标，许多指令集具有多个执行相同操作但操作数大小不同的指令。保留现有的使用一个字节的`OP_CONSTANT`指令，并定义一个新的`OP_CONSTANT_LONG`指令。它将操作数存储为24位的数字，这应该就足够了。

   > Implement this function:

   实现该函数：

   ```c
   void writeConstant(Chunk* chunk, Value value, int line) {
     // Implement me...
   }
   ```

> It adds `value` to `chunk`'s constant array and then writes an appropriate instruction to load the constant. Also add support to the disassembler for `OP_CONSTANT_LONG` instructions.

它向`chunk`的常量数组中添加`value`，然后写一条合适的指令来加载常量。同时在反汇编程序中增加对`OP_CONSTANT_LONG`指令的支持。

> Defining two instructions seems to be the best of both worlds. What sacrifices, if any, does it force on us?

定义两条指令似乎是两全其美的办法。它会迫使我们做出什么牺牲呢（如果有的话）？

3. > Our `reallocate()` function relies on the C standard library for dynamic memory allocation and freeing. `malloc()` and `free()` aren't magic. Find a couple of open source implementations of them and explain how they work. How do they keep track of which bytes are allocated and which are free? What is required to allocate a block of memory? Free it? How do they make that efficient? What do they do about fragmentation?

   我们的`reallocate()`函数依赖于C标准库进行动态内存分配和释放。`malloc()` 和 `free()` 并不神奇。找几个它们的开源实现，并解释它们是如何工作的。它们如何跟踪哪些字节被分配，哪些被释放？分配一个内存块需要什么？释放的时候呢？它们如何实现高效？它们如何处理碎片化内存？

   > *Hardcore mode:* Implement `reallocate()` without calling `realloc()`, `malloc()`, or `free()`. You are allowed to call `malloc()` *once*, at the beginning of the interpreter's execution, to allocate a single big block of memory, which your `reallocate()` function has access to. It parcels out blobs of memory from that single region, your own personal heap. It's your job to define how it does that.

   硬核模式：在不调用`realloc()`, `malloc()`, 和 `free()`的前提下，实现`reallocate()`。你可以在解释器开始执行时调用一次`malloc()`，来分配一个大的内存块，你的`reallocate()`函数能够访问这个内存块。它可以从这个区域（你自己的私人堆内存）中分配内存块。你的工作就是定义如何做到这一点。

> ## DESIGN NOTE: TEST YOUR LANGUAGE

设计笔记：测试你的语言

> We're almost halfway through the book and one thing we haven't talked about is *testing* your language implementation. That's not because testing isn't important. I can't possibly stress enough how vital it is to have a good, comprehensive test suite for your language.
>
> I wrote a test suite for Lox (which you are welcome to use on your own Lox implementation) before I wrote a single word of this book. Those tests found countless bugs in my implementations.
>
> Tests are important in all software, but they're even more important for a programming language for at least a couple of reasons:
>
> - **Users expect their programming languages to be rock solid.** We are so used to mature, stable compilers and interpreters that "It's your code, not the compiler" is an ingrained part of software culture. If there are bugs in your language implementation, users will go through the full five stages of grief before they can figure out what's going on, and you don't want to put them through all that.

> - **A language implementation is a deeply interconnected piece of software.** Some codebases are broad and shallow. If the file loading code is broken in your text editor, it—hopefully!—won't cause failures in the text rendering on screen. Language implementations are narrower and deeper, especially the core of the interpreter that handles the language's actual semantics. That makes it easy for subtle bugs to creep in caused by weird interactions between various parts of the system. It takes good tests to flush those out.
> - **The input to a language implementation is, by design, combinatorial.** There are an infinite number of possible programs a user could write, and your implementation needs to run them all correctly. You obviously can't test that exhaustively, but you need to work hard to cover as much of the input space as you can.
> - **Language implementations are often complex, constantly changing, and full of optimizations.** That leads to gnarly code with lots of dark corners where bugs can hide.
>
> All of that means you're gonna want a lot of tests. But *what* tests? Projects I've seen focus mostly on end-to-end "language tests". Each test is a program written in the language along with the output or errors it is expected to produce. Then you have a test runner that pushes the test program through your language implementation and validates that it does what it's supposed to. Writing your tests in the language itself has a few nice advantages:
>
> - The tests aren't coupled to any particular API or internal architecture decisions of the implementation. This frees you to reorganize or rewrite parts of your interpreter or compiler without needing to update a slew of tests.
> - You can use the same tests for multiple implementations of the language.
> - Tests can often be terse and easy to read and maintain since they are simply scripts in your language.
>
> It's not all rosy, though:
>
> - End-to-end tests help you determine *if* there is a bug, but not *where* the bug is. It can be harder to figure out where the erroneous code in the implementation is because all the test tells you is that the right output didn't appear.
> - It can be a chore to craft a valid program that tickles some obscure corner of the implementation. This is particularly true for highly optimized compilers where you may need to write convoluted code to ensure that you end up on just the right optimization path where a bug may be hiding.
> - The overhead can be high to fire up the interpreter, parse, compile, and run each test script. With a big suite of tests—which you *do* want, remember—that can mean a lot of time spent waiting for the tests to finish running.
>
> I could go on, but I don't want this to turn into a sermon. Also, I don't pretend to be an expert on *how* to test languages. I just want you to internalize how important it is *that* you test yours. Seriously. Test your language. You'll thank me for it.

我们的书已经过半了，有一件事我们还没有谈及，那就是*测试*你的语言实现。这并不是因为测试不重要。语言实现有一个好的、全面的套件是多么重要，我怎么强调都不为过。

在我写本书之前，我为Lox写了一个测试套件（你也可以在自己的Lox实现中使用它）。这些测试在我的语言实现中发现了无数的bug。

测试在所有软件中都很重要，但对于编程语言来说，测试甚至更重要，至少有以下几个原因：

- **用户希望他们的编程语言能够坚如磐石**。我们已经习惯了成熟的编译器、解释器，以至于"是你的代码（出错了），而不是编译器"成为软件文化中根深蒂固的一部分。如果你的语言实现中有错误，用户需要经历全部五个痛苦的阶段才能弄清楚发生了什么，而你并不想让他们经历这一切。
- **语言的实现是一个紧密相连的软件**。有些代码库既广泛又浮浅。如果你的文本编辑器中的文件加载代码被破坏了，它不会导致屏幕上的文本渲染失败（希望如此）。语言的实现则更狭窄和深入，特别是处理语言实际语义的解释器核心部分。这使得系统的各个部分之间奇怪的交互会造成微妙的错误。这就需要好的测试来清除这些问题。
- **从设计上来说，语言实现的输入是组合性的**。用户可以写出无限多的程序，而你的实现需要能够正确地运行这些程序。您显然不能进行详尽地测试，但需要努力覆盖尽可能多的输入空间。
- **语言的实现通常是复杂的、不断变化的，而且充满了优化**。这就导致了粗糙代码中有很多隐藏错误的黑暗角落。

所有这些都意味着你需要做大量的测试。但是什么测试呢？我见过的项目主要集中在端到端的"语言测试"上。每个测试都是一段用该语言编写的程序，以及它预期产生的输出或错误。然后，你还需要一个测试运行器，将这些测试程序输入到你的语言实现中，并验证它是否按照预期执行。用语言本身编写测试有一些很好的优势：

- 测试不与任何特定的API或语言实现的内部结构相耦合。这样你可以重新组织或重写解释器或编译器的一部分，而不需要更新大量的测试。
- 你可以对该语言的多种实现使用相同的测试。
- 测试通常是简洁的，易于阅读和维护，因为它们只是语言写就的简单脚本。

不过，这并不全是好事：

- 端到端测试可以帮助你确定是否存在错误，但不能确认错误在哪里。在语言实现中找出错误代码的位置可能更加困难，因为测试只能告诉你没有出现正确的输出。
- 要编写一个有效的程序来测试实现中一些不太明显的角落，可能是一件比较麻烦的事。对于高度优化的编译器来说尤其如此，你可能需要编写复杂的代码，以确保最终能够到达正确的优化路径，以测试其中可能隐藏的错误。
- 启动解释器、解析、编译和运行每个测试脚本的开销可能很高。对于一个大的测试套件来说，（如果你确实需要的话，请记住）这可能意味着需要花费很多时间来等待测试的完成。

我可以继续说下去，但是我不希望这变成一场说教。此外，我并不想假装自己是语言测试专家。我只是想让你在内心深处明白，测试你的语言是多么重要。我是认真的。测试你的语言。你会为此感谢我的。

15.虚拟机 A Virtual Machine

> Magicians protect their secrets not because the secrets are large and important, but because they are so small and trivial. The wonderful effects created on stage are often the result of a secret so absurd that the magician would be embarrassed to admit that that was how it was done.
>
> ——Christopher Priest, *The Prestige*

魔术师们之所以保护他们的秘密，并不是因为秘密很大、很重要，而是它们是如此小而微不足道。在舞台上创造出的奇妙效果往往源自于一个荒谬的小秘密，以至于魔术师都不好意思承认这是如何完成的。

> We've spent a lot of time talking about how to represent a program as a sequence of bytecode instructions, but it feels like learning biology using only stuffed, dead animals. We know what instructions are in theory, but we've never seen them in action, so it's hard to really understand what

> they *do*. It would be hard to write a compiler that outputs bytecode when we don't have a good understanding of how that bytecode behaves.

我们已经花了很多时间讨论如何将程序表示为字节码指令序列，但是这感觉像是只用填充的死动物来学习生物学。我们知道理论上的指令是什么，但我们在实际操作中从未见过，因此很难真正理解指令的*作用*。如果我们不能很好地理解字节码的行为方式，就很难编写输出字节码的编译器。

> So, before we go and build the front end of our new interpreter, we will begin with the back end—the virtual machine that executes instructions. It breathes life into the bytecode. Watching the instructions prance around gives us a clearer picture of how a compiler might translate the user's source code into a series of them.

因此，在构建新解释器的前端之前，我们先从后端开始——执行指令的虚拟机。它为字节码注入了生命。通过观察这些指令的运行，我们可以更清楚地了解编译器如何将用户的源代码转换成一系列的指令。

> ## 15.1An Instruction Execution Machine

15.1 指令执行机器

> The virtual machine is one part of our interpreter's internal architecture. You hand it a chunk of code—literally a Chunk—and it runs it. The code and data structures for the VM reside in a new module.

虚拟机是我们解释器内部结构的一部分。你把一个代码块交给它，它就会运行这块代码。VM的代码和数据结构放在一个新的模块中。

vm.h，创建新文件：

```
#ifndef clox_vm_h
#define clox_vm_h

#include "chunk.h"

typedef struct {
  Chunk* chunk;
} VM;

void initVM();
void freeVM();

#endif
```

> As usual, we start simple. The VM will gradually acquire a whole pile of state it needs to keep track of, so we define a struct now to stuff that all in. Currently, all we store is the chunk that it executes.

跟之前一样，我们从简单的部分开始。VM会逐步获取到一大堆它需要跟踪的状态，所以我们现在定义一个结构，把这些状态都塞进去。目前，我们只存储它执行的代码块。

> Like we do with most of the data structures we create, we also define functions to create and tear down a VM. Here's the implementation:

与我们创建的大多数数据结构类似，我们也会定义用来创建和释放虚拟机的函数。下面是其对应实现：

vm.c，创建新文件：

```
#include "common.h"
#include "vm.h"

VM vm;

void initVM() {
}

void freeVM() {
}
```

> OK, calling those functions “implementations” is a stretch. We don’t have any interesting state to initialize or free yet, so the functions are empty. Trust me, we’ll get there.

好吧，把这些函数称为"实现"有点牵强了。我们目前还没有任何感兴趣的状态需要初始化或释放，所以这些函数是空的。相信我，我们终会实现它的。

> The slightly more interesting line here is that declaration of `vm`. This module is eventually going to have a slew of functions and it would be a chore to pass around a pointer to the VM to all of them. Instead, we declare a single global VM object. We need only one anyway, and this keeps the code in the book a little lighter on the page.

这里稍微有趣的一行是`vm`的声明。这个模块最终会有一系列的函数，如果要将一个指向VM的指针传递给所有的函数，那就太麻烦了。相反，我们声明了一个全局VM对象。反正我们只需要一个虚拟机对象，这样可以让本书中的代码在页面上更轻便^1。

> Before we start pumping fun code into our VM, let’s go ahead and wire it up to the interpreter’s main entrypoint.

在我们开始向虚拟机中添加有效代码之前，我们先将其连接到解释器的主入口点。

main.c，在 main()方法中新增代码：

```
int main(int argc, const char* argv[]) {
  // 新增部分开始
  initVM();
  // 新增部分结束
  Chunk chunk;
```

> We spin up the VM when the interpreter first starts. Then when we’re about to exit, we wind it down.

当解释器第一次启动时，我们也启动虚拟机。然后当我们要退出时，我们将其关闭。

main.c，在 main()方法中添加代码：

```
  disassembleChunk(&chunk, "test chunk");
  // 新增部分开始
  freeVM();
  // 新增部分结束
  freeChunk(&chunk);
```

> One last ceremonial obligation:

最后一项仪式性任务：

main.c，添加代码：

```
#include "debug.h"
// 新增部分开始
#include "vm.h"
// 新增部分结束
int main(int argc, const char* argv[]) {
```

> Now when you run clox, it starts up the VM before it creates that hand-authored chunk from the last chapter. The VM is ready and waiting, so let's teach it to do something.

现在如果你运行clox，它会先启动虚拟机，再创建上一章中的手写代码块。虚拟机已经就绪了，我们来教它一些事情吧。

> ## 15.1.1Executing instructions

15.1.1 执行指令

> The VM springs into action when we command it to interpret a chunk of bytecode.

当我们命令VM解释一个字节码块时，它就会开始启动了。

main.c，在 main()方法中添加代码：

```
  disassembleChunk(&chunk, "test chunk");
  // 新增部分开始
  interpret(&chunk);
  // 新增部分结束
  freeVM();
```

> This function is the main entrypoint into the VM. It's declared like so:

这个函数是进入VM的主要入口。它的声明如下：

vm.h，在 freeVM()方法后添加：

```
void freeVM();
// 新增部分开始
InterpretResult interpret(Chunk* chunk);
// 新增部分结束
#endif
```

> The VM runs the chunk and then responds with a value from this enum:

VM会运行字节码块，然后返回下面枚举中的一个值作为响应：

vm.h，在结构体VM后添加：

```
} VM;
// 新增部分开始
typedef enum {
  INTERPRET_OK,
  INTERPRET_COMPILE_ERROR,
  INTERPRET_RUNTIME_ERROR
} InterpretResult;
// 新增部分结束
void initVM();
void freeVM();
```

> We aren't using the result yet, but when we have a compiler that reports static errors and a VM that detects runtime errors, the interpreter will use this to know how to set the exit code of the process.

我们现在还不会使用这个结果，但是当我们有一个报告静态错误的编译器和检测运行时错误的VM时，解释器会通过它来知道如何设置进程的退出代码。

> We're inching towards some actual implementation.

我们正逐步走向一些真正的实现。

vm.c，在freeVM()方法后添加：

```
InterpretResult interpret(Chunk* chunk) {
  vm.chunk = chunk;
  vm.ip = vm.chunk->code;
  return run();
}
```

> First, we store the chunk being executed in the VM. Then we call `run()`, an internal helper function that actually runs the bytecode instructions. Between those two parts is an intriguing line. What is this `ip` business?

首先，我们在虚拟机中存储正在执行的块。然后我们调用`run()`，这是一个内部辅助函数，实际运行字节码指令。在这两部分之间，有一条耐人寻味的线。这个`ip`作用是什么？

> As the VM works its way through the bytecode, it keeps track of where it is—the location of the instruction currently being executed. We don't use a local variable inside `run()` for this because eventually other functions will need to access it. Instead, we store it as a field in VM.

当虚拟机运行字节码时，它会记录它在哪里——即当前执行的指令所在的位置。我们没有在`run()`方法中使用局部变量来进行记录，因为最终其它函数也会访问该值。相对地，我们将其作为一个字段存储在VM中^2。

vm.h，在结构体VM中添加代码：

```
typedef struct {
  Chunk* chunk;
  // 新增部分开始
  uint8_t* ip;
  // 新增部分结束
} VM;
```

> Its type is a byte pointer. We use an actual real C pointer pointing right into the middle of the bytecode array instead of something like an integer index because it's faster to dereference a pointer than look up an element in an array by index.

它的类型是一个字节指针。我们使用一个真正的C指针指向字节码数组的中间，而不是使用类似整数索引这种方式，这是因为对指针的引用比通过索引查找数组中的一个元素要更快。

> The name "IP" is traditional, and—unlike many traditional names in CS—actually makes sense: it's an **instruction pointer**. Almost every instruction set in the world, real and virtual, has a register or variable like this.

"IP"这个名字很传统，而且与CS中的很多传统名称不同的是，它是有实际意义的：它是一个指令指针。几乎世界上所有的指令集，不管是真实的还是虚拟的，都有一个类似的寄存器或变量^3。

> We initialize `ip` by pointing it at the first byte of code in the chunk. We haven't executed that instruction yet, so `ip` points to the instruction *about to be executed*. This will be true during the entire time the VM is running: the IP always points to the next instruction, not the one currently being handled.

我们通过将`ip`指向块中的第一个字节码来对其初始化。我们还没有执行该指令，所以`ip`指向*即将执行*的指令。在虚拟机执行的整个过程中都是如此：IP总是指向下一条指令，而不是当前正在处理的指令。

> The real fun happens in `run`().

真正有趣的部分在`run()`中。

vm.c，在 freeVM()方法后添加：

```
static InterpretResult run() {
#define READ_BYTE() (*vm.ip++)

  for (;;) {
    uint8_t instruction;
    switch (instruction = READ_BYTE()) {
      case OP_RETURN: {
        return INTERPRET_OK;
      }
    }
  }

#undef READ_BYTE
}
```

> This is the single most important function in all of clox, by far. When the interpreter executes a user's program, it will spend something like 90% of its time inside `run()`. It is the beating heart of the VM.

到目前为止，这是clox中最重要的一个函数。当解释器执行用户的程序时，它有大约90%的时间是在`run()`中。它是虚拟机跳动的心脏。

> Despite that dramatic intro, it's conceptually pretty simple. We have an outer loop that goes and goes. Each turn through that loop, we read and execute a single bytecode instruction.

尽管这个介绍很戏剧性，但从概念上来说很简单。我们有一个不断进行的外层循环。每次循环中，我们会读取并执行一条字节码指令。

> To process an instruction, we first figure out what kind of instruction we're dealing with. The `READ_BYTE` macro reads the byte currently pointed at by `ip` and then advances the instruction pointer. The first byte of any instruction is the opcode. Given a numeric opcode, we need to get to the right C code that implements that instruction's semantics. This process is called **decoding** or **dispatching** the instruction.

为了处理一条指令，我们首先需要弄清楚要处理的是哪种指令。`READ_BYTE`这个宏会读取`ip`当前指向字节，然后推进指令指针^4。任何指令的第一个字节都是操作码。给定一个操作码，我们需要找到实现该指令语义的正确的C代码。这个过程被称为**解码**或指令**分派**。

> We do that process for every single instruction, every single time one is executed, so this is the most performance critical part of the entire virtual machine. Programming language lore is filled with clever techniques to do bytecode dispatch efficiently, going all the way back to the early days of computers.

每一条指令，每一次执行时，我们都会进行这个过程，所以这是整个虚拟机性能最关键的部分。编程语言的传说中充满了高效进行字节码分派的各种奇技淫巧^5，一直可以追溯到计算机的早期。

> Alas, the fastest solutions require either non-standard extensions to C, or handwritten assembly code. For clox, we'll keep it simple. Just like our disassembler, we have a single giant `switch` statement with a case for each opcode. The body of each case implements that opcode's behavior.

可惜的是，最快的解决方案要么需要对C进行非标准的扩展，要么需要手写汇编代码。对于clox，我们要保持简单。就像我们的反汇编程序一样，我们写一个巨大的`switch`语句，其中每个case对应一个操作码。每个case代码体实现了操作码的行为。

> So far, we handle only a single instruction, `OP_RETURN`, and the only thing it does is exit the loop entirely. Eventually, that instruction will be used to return from the current Lox function, but we don't have functions yet, so we'll repurpose it temporarily to end the execution.

到目前为止，我们只处理了一条指令，`OP_RETURN`，而它做的唯一的事情就是完全退出循环。最终，该指令将被用于从当前的Lox函数返回，但是我们目前还没有函数，所以我们暂时用它来结束代码执行。

> Let’s go ahead and support our one other instruction.

让我们继续支持另一个指令。

vm.c，在 run()方法中增加代码：

```
    switch (instruction = READ_BYTE()) {
      // 新增部分开始
      case OP_CONSTANT: {
        Value constant = READ_CONSTANT();
        printValue(constant);
        printf("\n");
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> We don’t have enough machinery in place yet to do anything useful with a constant. For now, we’ll just print it out so we interpreter hackers can see what’s going on inside our VM. That call to `printf()` necessitates an include.

我们还没有足够的机制来使用常量做任何有用的事。现在，我们只是把它打印出来，这样我们这些解释器黑客就可以看到我们的VM内部发生了什么。调用`printf()`方法需要进行include。

vm.c，在文件顶部添加：

```
// 新增部分开始
#include <stdio.h>
// 新增部分结束
#include "common.h"
```

> We also have a new macro to define.

我们还需要定义一个新的宏。

vm.c，在 run()方法中添加代码：

```
#define READ_BYTE() (*vm.ip++)
// 新增部分开始
#define READ_CONSTANT() (vm.chunk->constants.values[READ_BYTE()])
// 新增部分结束
  for (;;) {
```

> READ_CONSTANT() reads the next byte from the bytecode, treats the resulting number as an index, and looks up the corresponding Value in the chunk's constant table. In later chapters, we'll add a few more instructions with operands that refer to constants, so we're setting up this helper macro now.

READ_CONTANT()从字节码中读取下一个字节，将得到的数字作为索引，并在代码块的常量表中查找相应的Value。在后面的章节中，我们将添加一些操作数指向常量的指令，所以我们现在要设置这个辅助宏。

Like the previous READ_BYTE macro, READ_CONSTANT is only used inside run(). To make that scoping more explicit, the macro definitions themselves are confined to that function. We define them at the beginning and —because we care—undefine them at the end.

与之前的READ_BYTE宏类似，READ_CONSTANT只会在run()方法中使用。为了使作用域更明确，宏定义本身要被限制在该函数中。我们在开始时定义了它们，然后因为我们比较关心，在结束时取消它们的定义^6。

vm.c，在 run()方法中添加：

```
#undef READ_BYTE
// 新增部分开始
#undef READ_CONSTANT
// 新增部分结束
}
```

> ## 15.1.2 Execution tracing

15.1.2 执行跟踪

> If you run clox now, it executes the chunk we hand-authored in the last chapter and spits out 1.2 to your terminal. We can see that it's working, but that's only because our implementation of OP_CONSTANT has temporary code to log the value. Once that instruction is doing what it's supposed to do and plumbing that constant along to other operations that want to consume it, the VM will become a black box. That makes our lives as VM implementers harder.

如果现在运行clox，它会执行我们在上一章中手工编写的字节码块，并向终端输出1.2。我们可以看到它在工作，但这是因为我们在OP_CONSTANT的实现中，使用临时代码记录了这个值。一旦该指令执行了它应做的操作，并将取得的常量传递给其它想要使用该常量的操作，虚拟机就会变成一个黑盒子。这使得我们作为虚拟机实现者的工作更加艰难。

> To help ourselves out, now is a good time to add some diagnostic logging to the VM like we did with chunks themselves. In fact, we'll even reuse the same code. We don't want this logging enabled all the time—it's just for us VM hackers, not Lox users—so first we create a flag to hide it behind.

为了帮助我们自己解脱这种困境，现在是给虚拟机添加一些诊断性日志的好时机，就像我们对代码块本身所做的那样。事实上，我们甚至会重用相同的代码。我们不希望一直启用这个日志——它只针对我们这些虚拟机开发者，而不是Lox用户——所以我们首先创建一个标志来隐藏它。

common.h，新增代码：

```
#include <stdint.h>
// 新增部分开始
#define DEBUG_TRACE_EXECUTION
// 新增部分结束
#endif
```

> When this flag is defined, the VM disassembles and prints each instruction right before executing it. Where our previous disassembler walked an entire chunk once, statically, this disassembles instructions dynamically, on the fly.

定义了这个标志之后，虚拟机在执行每条指令之前都会反汇编并将其打印出来。我们之前的反汇编程序只是静态地遍历一次整个字节码块，而这个反编译程序则是动态地、即时地对指令进行反汇编。

vm.c，在 run()方法中新增代码：

```
  for (;;) {
// 新增部分开始
#ifdef DEBUG_TRACE_EXECUTION
    disassembleInstruction(vm.chunk,
                           (int)(vm.ip - vm.chunk->code));
#endif
// 新增部分结束
    uint8_t instruction;
```

> Since `disassembleInstruction()` takes an integer byte *offset* and we store the current instruction reference as a direct pointer, we first do a little pointer math to convert `ip` back to a relative offset from the beginning of the bytecode. Then we disassemble the instruction that begins at that byte.

由于 `disassembleInstruction()` 方法接收一个整数`offset`作为字节偏移量，而我们将当前指令引用存储为一个直接指针，所以我们首先要做一个小小的指针运算，将`ip`转换成从字节码开始的相对偏移量。然后，我们对从该字节开始的指令进行反汇编。

> As ever, we need to bring in the declaration of the function before we can call it.

跟之前一样，我们需要在调用函数之前先引入函数的声明。

vm.c，新增代码：

```
#include "common.h"
// 新增部分开始
#include "debug.h"
// 新增部分结束
#include "vm.h"
```

> I know this code isn't super impressive so far—it's literally a switch statement wrapped in a `for` loop but, believe it or not, this is one of the two major components of our VM. With this, we can imperatively execute instructions. Its simplicity is a virtue—the less work it does, the faster it can do it.

> Contrast this with all of the complexity and overhead we had in jlox with the Visitor pattern for walking the AST.

我知道这段代码到目前为止还不是很令人印象深刻——它实际上只是一个封装在`for`循环中的switch语句，但信不信由你，这就是我们虚拟机的两个主要组成部分之一。有了它，我们就可以命令式地执行指令。它的简单是一种优点——它做的工作越少，就能做得越快。作为对照，可以回想一下我们在jlox中使用Visitor模式遍历AST的复杂度和开销。

15.2 A Value Stack Manipulator

15.2 一个值栈操作器

> In addition to imperative side effects, Lox has expressions that produce, modify, and consume values. Thus, our compiled bytecode needs a way to shuttle values around between the different instructions that need them. For example:

除了命令式的副作用外，Lox还有产生、修改和使用值的表达式。因此，我们编译的字节码还需要一种方法在需要值的不同指令之间传递它们。例如：

```
print 3 - 2;
```

> We obviously need instructions for the constants 3 and 2, the `print` statement, and the subtraction. But how does the subtraction instruction know that 3 is the minuend and 2 is the subtrahend? How does the print instruction know to print the result of that?

显然我们需要常数3和2、`print`语句和减法对应的指令。但是减法指令如何知道3是被减数而2是减数呢？打印指令怎么知道要打印计算结果的呢？

> To put a finer point on it, look at this thing right here:

为了说得更清楚一点，看看下面的代码：

```
fun echo(n) {
  print n;
  return n;
}

print echo(echo(1) + echo(2)) + echo(echo(4) + echo(5));
```

> I wrapped each subexpression in a call to `echo()` that prints and returns its argument. That side effect means we can see the exact order of operations.

我将每个子表达式都包装在对`echo()`的调用中，这个调用会打印并返回其参数。这个副作用意味着我们可以看到操作的确切顺序。

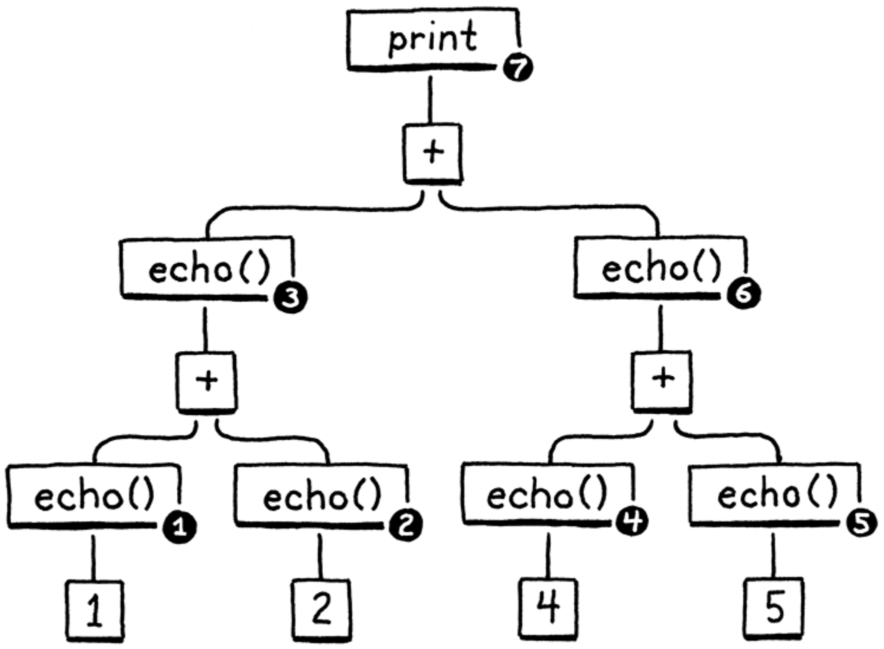
> Don't worry about the VM for a minute. Think about just the semantics of Lox itself. The operands to an arithmetic operator obviously need to be evaluated before we can perform the operation itself. (It's

> pretty hard to add `a + b` if you don’t know what `a` and `b` are.) Also, when we implemented expressions in jlox, we decided that the left operand must be evaluated before the right.

暂时不要担心虚拟机的问题。只考虑Lox本身的语义。算术运算符的操作数显然需要在执行运算操作之前求值（如果你不知道`a`和`b`是什么，就很难计算`a+b`）。另外，当我们在jlox中实现表达式时，我们决定了左操作数必须在右操作数之前进行求值^7。

> Here is the syntax tree for the `print` statement:

下面是`print`语句的语法树：



> Given left-to-right evaluation, and the way the expressions are nested, any correct Lox implementation *must* print these numbers in this order:

确定了从左到右的求值顺序，以及表达式嵌套方式，任何一个正确的Lox实现都*必须*按照下面的顺序打印这些数字：

```
1  // from echo(1)
2  // from echo(2)
3  // from echo(1 + 2)
4  // from echo(4)
5  // from echo(5)
9  // from echo(4 + 5)
12 // from print 3 + 9
```

> Our old jlox interpreter accomplishes this by recursively traversing the AST. It does a postorder traversal. First it recurses down the left operand branch, then the right operand, then finally it evaluates the node itself.

我们的老式jlox解释器通过递归遍历AST来实现这一点。其中使用的是后序遍历。首先，它向下递归左操作数分支，然后是右操作数分支，最后计算节点本身。

> After evaluating the left operand, jlox needs to store that result somewhere temporarily while it's busy traversing down through the right operand tree. We use a local variable in Java for that. Our recursive tree-walk interpreter creates a unique Java call frame for each node being evaluated, so we could have as many of these local variables as we needed.

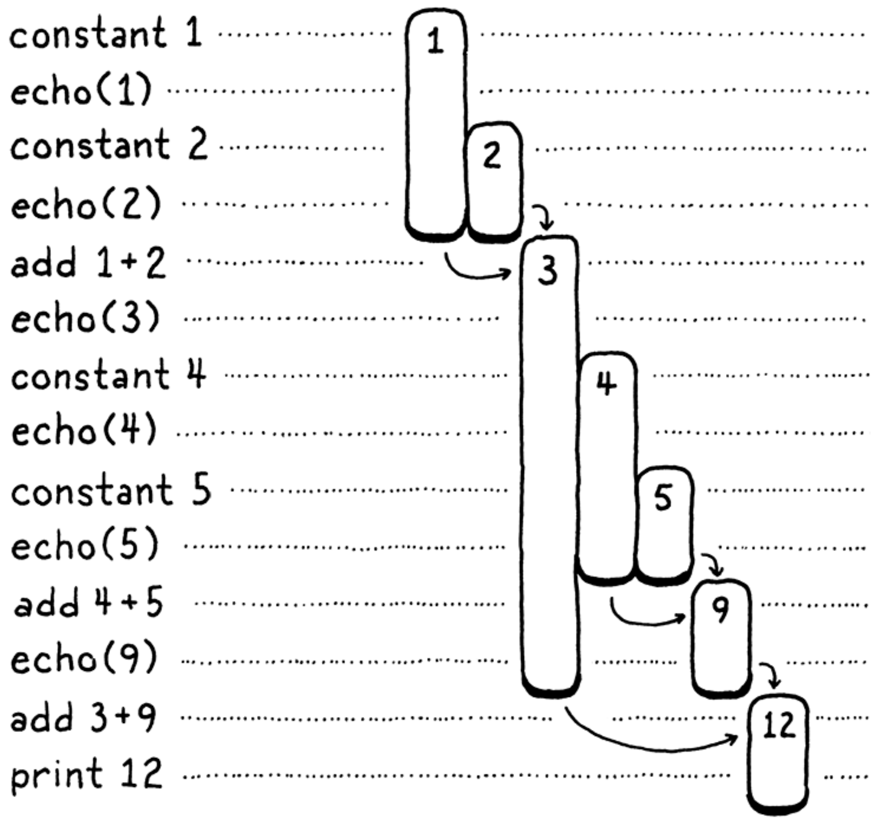
在对左操作数求值之后，jlox需要将结果临时保存在某个地方，然后再向下遍历右操作数。我们使用Java中的一个局部变量来实现。我们的递归树遍历解释器会为每个正在求值的节点创建一个单独的Java调用帧，所以我们可以根据需要维护很多这样的局部变量。

> In clox, our `run()` function is not recursive—the nested expression tree is flattened out into a linear series of instructions. We don't have the luxury of using C local variables, so how and where should we store these temporary values? You can probably guess already, but I want to really drill into this because it's an aspect of programming that we take for granted, but we rarely learn *why* computers are architected this way.

在clox中，我们的`run()`函数不是递归的——嵌套的表达式被展开成一系列线性指令。我们没有办法使用C语言的局部变量，那我们应该如何存储这些临时值呢？你可能已经猜到了，但我想真正深入研究这个问题，因为这是编程中我们习以为常的一个方面，但我们很少了解为什么计算机是这样架构的。

> Let's do a weird exercise. We'll walk through the execution of the above program a step at a time:

让我们做一个奇怪的练习。我们来一步一步地遍历上述程序的执行过程：



> On the left are the steps of code. On the right are the values we're tracking. Each bar represents a number. It starts when the value is first produced—either a constant or the result of an addition. The length of the bar tracks when a previously produced value needs to be kept around, and it ends when that value finally gets consumed by an operation.

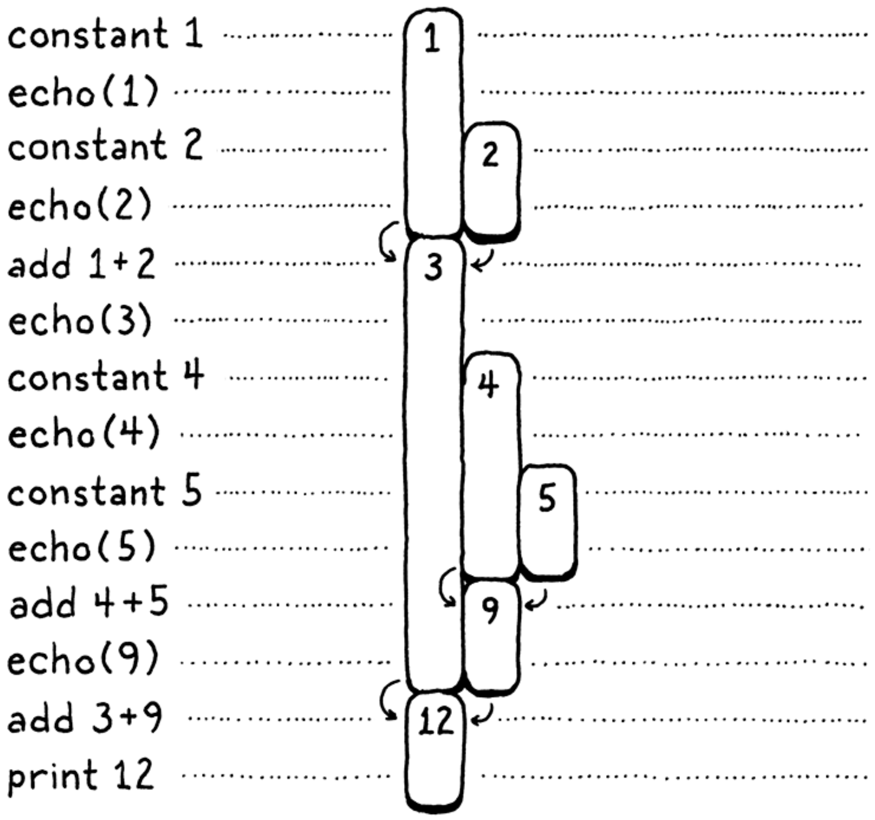
左边是代码的执行步骤。右边是我们要追踪的值。每条杠代表一个数字。起点是数值产生时——要么是一个常数，要么是一个加法计算结果；杠的长度表示之前产生的值需要保留的时间；当该值最终被某个操作消费后，杠就到终点了。

> As you step through, you see values appear and then later get eaten. The longest-lived ones are the values produced from the left-hand side of an addition. Those stick around while we work through the right-hand operand expression.

随着你不断执行，你会看到一些数值出现，然后被消费掉。寿命最长的是加法左侧产生的值。当我们在处理右边的操作数表达式时，这些值会一直存在。

> In the above diagram, I gave each unique number its own visual column. Let's be a little more parsimonious. Once a number is consumed, we allow its column to be reused for another later value. In other words, we take all of those gaps up there and fill them in, pushing in numbers from the right:

在上图中，我为每个数字提供了单独的可视化列。让我们更简洁一些。一旦一个数字被消费了，我们就允许它的列被其它值重用。换句话说，我们将数字从右向左推入，把上面的空隙都填上：



> There's some interesting stuff going on here. When we shift everything over, each number still manages to stay in a single column for its entire life. Also, there are no gaps left. In other words, whenever a number appears earlier than another, then it will live at least as long as that second one. The first number to appear is the last to be consumed. Hmm . . . last-in, first-out . . . why, that's a stack!

这里有一些有趣的事情发生了。当我们把所有数字都移动以后，每个数字在整个生命周期中仍然能保持在一列。此外，也没有留下任何空隙。换句话说，只要一个数字比另一个数字出现得早，那么它的寿命至少和第二个数字一样长。第一个出现的数字是最后一个消费掉的，嗯……后进先出……哎呀，这是一个栈！

> In the second diagram, each time we introduce a number, we push it onto the stack from the right. When numbers are consumed, they are always popped off from rightmost to left.

在第二张图中，每次我们生成一个数字时，都会从右边将它压入栈。当数字被消费时，它们也是从右向左进行弹出。

> Since the temporary values we need to track naturally have stack-like behavior, our VM will use a stack to manage them. When an instruction "produces" a value, it pushes it onto the stack. When it needs to consume one or more values, it gets them by popping them off the stack.

由于我们需要跟踪的临时值天然具有类似栈的行为，我们的虚拟机将使用栈来管理它们。当一条指令"生成"一个值时，它会把这个值压入栈中。当它需要消费一个或多个值时，通过从栈中弹出数据来获得这些值。

> ### 15.2.1 The VM's Stack

15.2.1 虚拟机的栈

> Maybe this doesn't seem like a revelation, but I *love* stack-based VMs. When you first see a magic trick, it feels like something actually magical. But then you learn how it works—usually some mechanical gimmick or misdirection—and the sense of wonder evaporates. There are a couple of ideas in computer science where even after I pulled them apart and learned all the ins and outs, some of the initial sparkle remained. Stack-based VMs are one of those.

也许这看起来不像是什么新发现，但我喜欢基于栈的虚拟机。当你第一次看到一个魔术时，你会觉得它真的很神奇。但是当你了解到它是如何工作的——通常是一些机械式花招或误导——惊奇的感觉就消失了。在计算机科学中，有一些理念，即使我把它们拆开并了解了所有的来龙去脉之后，最初的闪光点仍然存在。基于堆栈的虚拟机就是其中之一^8。

> As you'll see in this chapter, executing instructions in a stack-based VM is dead simple. In later chapters, you'll also discover that compiling a source language to a stack-based instruction set is a piece of cake. And yet, this architecture is fast enough to be used by production language implementations. It almost feels like cheating at the programming language game.

你在本章中将会看到，在基于堆栈的虚拟机中执行指令是非常简单的。在后面的章节中，你还会发现，将源语言编译成基于栈的指令集是小菜一碟。但是，这种架构的速度快到足以在产生式语言的实现中使用。这感觉就像是在编程语言游戏中作弊^9。

> Alrighty, it's codin' time! Here's the stack:

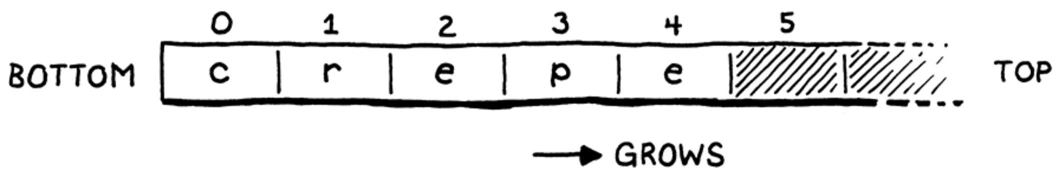
好了，编码时间到！下面是栈：

vm.h，在结构体VM中添加代码：

```
typedef struct {
  Chunk* chunk;
  uint8_t* ip;
  // 新增部分开始
  Value stack[STACK_MAX];
  Value* stackTop;
  // 新增部分结束
} VM;
```

> We implement the stack semantics ourselves on top of a raw C array. The bottom of the stack—the first value pushed and the last to be popped—is at element zero in the array, and later pushed values follow it. If we push the letters of "crepe"—my favorite stackable breakfast item—onto the stack, in order, the resulting C array looks like this:

我们在一个原生的C数组上自己实现了栈语义。栈的底部——第一个推入的值和最后一个被弹出的值——位于数组中的零号位置，后面推入的值跟在它后面。如果我们把"crepe"几个字母按顺序推入栈中，得到的C数组看起来像这样：

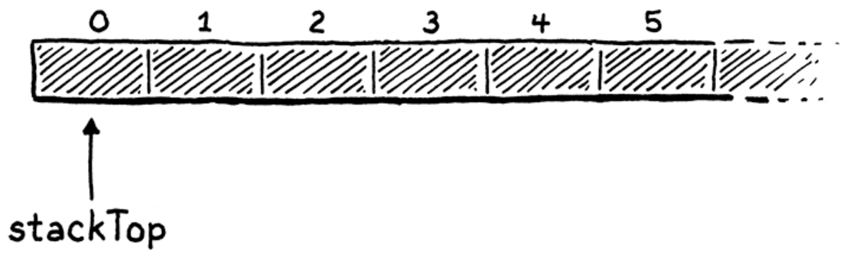


> Since the stack grows and shrinks as values are pushed and popped, we need to track where the top of the stack is in the array. As with `ip`, we use a direct pointer instead of an integer index since it's faster to dereference the pointer than calculate the offset from the index each time we need it.

由于栈会随着值的压入和弹出而伸缩，我们需要跟踪栈的顶部在数组中的位置。和`ip`一样，我们使用一个直接指针而不是整数索引，因为每次我们需要使用它时，解引用比计算索引的偏移量更快。

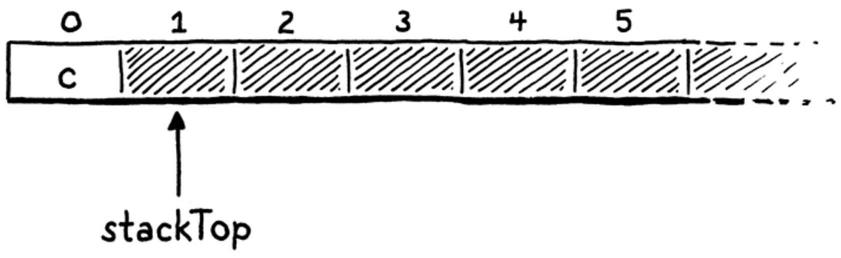
> The pointer points at the array element just *past* the element containing the top value on the stack. That seems a little odd, but almost every implementation does this. It means we can indicate that the stack is empty by pointing at element zero in the array.

指针指向数组中栈顶元素的下一个元素位置，这看起来有点奇怪，但几乎每个实现都会这样做。这意味着我们可以通过指向数组中的零号元素来表示栈是空的。



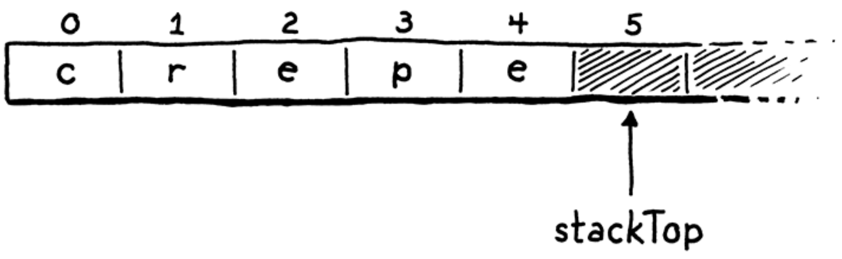
> If we pointed to the top element, then for an empty stack we'd need to point at element -1. That's undefined in C. As we push values onto the stack . . .

如果我们指向栈顶元素，那么对于空栈，我们就需要指向-1位置的元素^10。这在C语言中是没有定义的。当我们把值压入栈时：



> . . . `stackTop` always points just past the last item.

`stackTop`一直会超过栈中的最后一个元素。



> I remember it like this: `stackTop` points to where the next value to be pushed will go. The maximum number of values we can store on the stack (for now, at least) is:

我是这样记的：`stackTop`指向下一个值要被压入的位置。我们在栈中可以存储的值的最大数量（至少目前是这样）为：

vm.h，添加代码：

```
#include "chunk.h"
// 新增部分开始
#define STACK_MAX 256
// 新增部分结束
typedef struct {
```

> Giving our VM a fixed stack size means it's possible for some sequence of instructions to push too many values and run out of stack space—the classic "stack overflow". We could grow the stack dynamically as needed, but for now we'll keep it simple. Since VM uses Value, we need to include its declaration.

给我们的虚拟机一个固定的栈大小，意味着某些指令系列可能会压入太多的值并耗尽栈空间——典型的"堆栈溢出"。我们可以根据需要动态地增加栈，但是现在我们还是保持简单。因为VM中会使用Value，我们需要包含它的声明。

vm.h，添加代码：

```
#include "chunk.h"
// 新增部分开始
#include "value.h"
// 新增部分结束
#define STACK_MAX 256
```

> Now that VM has some interesting state, we get to initialize it.

现在，虚拟机中有了一些有趣的状态，我们要对它进行初始化。

vm.c，在 initVM() 中添加代码：

```
void initVM() {
  // 新增部分开始
  resetStack();
  // 新增部分结束
}
```

> That uses this helper function:

其中使用了这个辅助函数：

vm.c，在变量 vm后添加：

```
static void resetStack() {
  vm.stackTop = vm.stack;
}
```

> Since the stack array is declared directly inline in the VM struct, we don't need to allocate it. We don't even need to clear the unused cells in the array—we simply won't access them until after values have been stored in them. The only initialization we need is to set `stackTop` to point to the beginning of the array to indicate that the stack is empty.

因为栈数组是直接在VM结构体中内联声明的，所以我们不需要为其分配空间。我们甚至不需要清除数组中不使用的单元——我们只有在值存入之后才会访问它们。我们需要的唯一的初始化操作就是将`stackTop`指向数组的起始位置，以表明栈是空的。

> The stack protocol supports two operations:

栈协议支持两种操作：

vm.h，在 interpret()方法后添加：

```
InterpretResult interpret(Chunk* chunk);
// 新增部分开始
void push(Value value);
Value pop();
// 新增部分结束
#endif
```

> You can push a new value onto the top of the stack, and you can pop the most recently pushed value back off. Here's the first function:

你可以把一个新值压入栈顶，你也可以把最近压入的值弹出。下面是第一个函数：

vm.c，在 freeVM()方法后添加：

```
void push(Value value) {
  *vm.stackTop = value;
  vm.stackTop++;
}
```

> If you're rusty on your C pointer syntax and operations, this is a good warm-up. The first line stores `value` in the array element at the top of the stack. Remember, `stackTop` points just *past* the last used element, at the next available one. This stores the value in that slot. Then we increment the pointer itself to point to the next unused slot in the array now that the previous slot is occupied.

如果你对C指针的语法和操作感到生疏，这是一个很好的熟悉的机会。第一行在栈顶的数组元素中存储`value`。记住，`stackTop`刚刚跳过上次使用的元素，即下一个可用的元素。这里把值存储在该元素槽中。接着，因为上一个槽被占用了，我们增加指针本身，指向数组中下一个未使用的槽。

> Popping is the mirror image.

弹出正好是压入的镜像操作。

vm.c，在 push()方法后添加代码：

```
Value pop() {
  vm.stackTop--;
  return *vm.stackTop;
}
```

> First, we move the stack pointer *back* to get to the most recent used slot in the array. Then we look up the value at that index and return it. We don't need to explicitly "remove" it from the array—moving `stackTop` down is enough to mark that slot as no longer in use.

首先，我们将栈指针回退到数组中最近使用的槽。然后，我们查找该索引处的值并将其返回。我们不需要显式地将其从数组中“移除”——将`stackTop`下移就足以将该槽标记为不再使用了。

> ### 15.2.2 Stack tracing

15.2.2 栈跟踪

> We have a working stack, but it's hard to *see* that it's working. When we start implementing more complex instructions and compiling and running larger pieces of code, we'll end up with a lot of values crammed into that array. It would make our lives as VM hackers easier if we had some visibility into the stack.

我们有了一个工作的栈，但是很难看出它在工作。当我们开始实现更复杂的指令，编译和运行更大的代码片段时，最终会在这个数组中塞入很多值。如果我们对栈有一定的可见性，那么作为虚拟机开发者，我们就会更轻松。

> To that end, whenever we're tracing execution, we'll also show the current contents of the stack before we interpret each instruction.

为此，每当我们追踪执行情况时，我们也会在解释每条指令之前展示栈中的当前内容。

vm.c，在 run()方法中添加代码：

```
#ifdef DEBUG_TRACE_EXECUTION
// 新增部分开始
```

```
    printf("          ");
    for (Value* slot = vm.stack; slot < vm.stackTop; slot++) {
      printf("[ ");
      printValue(*slot);
      printf(" ]");
    }
    printf("\n");
// 新增部分结束
    disassembleInstruction(vm.chunk,
```

> We loop, printing each value in the array, starting at the first (bottom of the stack) and ending when we reach the top. This lets us observe the effect of each instruction on the stack. The output is pretty verbose, but it's useful when we're surgically extracting a nasty bug from the bowels of the interpreter.

我们循环打印数组中的每个值，从第一个值开始（栈底），到栈顶结束。这样我们可以观察到每条指令对栈的影响。这个输出会相当冗长，但是从我们在解释器中遇到令人讨厌的错误时，这就会很有用了。

> Stack in hand, let's revisit our two instructions. First up:

堆栈在手，让我们重新审视一下目前的两条指令。首先是：

vm.c，在 run()方法中替换两行：

```
      case OP_CONSTANT: {
        Value constant = READ_CONSTANT();
        // 新增部分开始
        push(constant);
        // 新增部分结束
        break;
```

> In the last chapter, I was hand-wavey about how the `OP_CONSTANT` instruction "loads" a constant. Now that we have a stack you know what it means to actually produce a value: it gets pushed onto the stack.

在上一节中，我粗略介绍了`OP_CONSTANT`指令是如何"加载"一个常量的。现在我们有了一个堆栈，你就知道产生一个值实际上意味着什么：将它压入栈。

vm.c，在 run()方法中添加代码：

```
      case OP_RETURN: {
        // 新增部分开始
        printValue(pop());
        printf("\n");
        // 新增部分结束
        return INTERPRET_OK;
```

> Then we make `OP_RETURN` pop the stack and print the top value before exiting. When we add support for real functions to clox, we'll change this code. But, for now, it gives us a way to get the VM executing

> simple instruction sequences and displaying the result.

接下来，我们让`OP_RETURN`在退出之前弹出栈顶值并打印。等到我们在clox中添加对真正的函数的支持时，我们将会修改这段代码。但是，目前来看，我们可以使用这种方法让VM执行简单的指令序列并显示结果。

15.3 An Arithmetic Calculator

15.3 数学计算器

> The heart and soul of our VM are in place now. The bytecode loop dispatches and executes instructions. The stack grows and shrinks as values flow through it. The two halves work, but it’s hard to get a feel for how cleverly they interact with only the two rudimentary instructions we have so far. So let’s teach our interpreter to do arithmetic.

我们的虚拟机的核心和灵魂现在都已经就位了。字节码循环分派和执行指令。栈堆随着数值的流动而增长和收缩。这两部分都在工作，但仅凭我们目前的两条基本指令，很难感受到它们如何巧妙地互动。所以让我们教解释器如何做算术。

We’ll start with the simplest arithmetic operation, unary negation.

我们从最简单的算术运算开始，即一元取负。

```
var a = 1.2;
print -a; // -1.2.
```

> The prefix `-` operator takes one operand, the value to negate. It produces a single result. We aren’t fussing with a parser yet, but we can add the bytecode instruction that the above syntax will compile to.

前缀的`-`运算符接受一个操作数，也就是要取负的值。它只产生一个结果。我们还没有对解析器进行处理，但可以添加上述语法编译后对应的字节码指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CONSTANT,
  // 新增部分开始
  OP_NEGATE,
  // 新增部分结束
  OP_RETURN,
```

> We execute it like so:

我们这样执行它：

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_NEGATE:   push(-pop()); break;
      // 新增部分结束
      case OP_RETURN: {
```

> The instruction needs a value to operate on, which it gets by popping from the stack. It negates that, then pushes the result back on for later instructions to use. Doesn’t get much easier than that. We can disassemble it too.

该指令需要操作一个值，该值通过弹出栈获得。它对该值取负，然后把结果重新压入栈，以便后面的指令使用。没有什么比这更简单的了。我们也可以对其反汇编：

debug.c，在 disassembleInstruction()方法中添加代码：

```
    case OP_CONSTANT:
      return constantInstruction("OP_CONSTANT", chunk, offset);
    // 新增部分开始
    case OP_NEGATE:
      return simpleInstruction("OP_NEGATE", offset);
    // 新增部分结束
    case OP_RETURN:
```

> And we can try it out in our test chunk.

我们可以在测试代码中试一试。

main.c，在 main()方法中添加代码：

```
  writeChunk(&chunk, constant, 123);
  // 新增部分开始
  writeChunk(&chunk, OP_NEGATE, 123);
  // 新增部分结束
  writeChunk(&chunk, OP_RETURN, 123);
```

> After loading the constant, but before returning, we execute the negate instruction. That replaces the constant on the stack with its negation. Then the return instruction prints that out:

在加载常量之后，返回之前，我们会执行取负指令。这条指令会将栈中的常量替换为其对应的负值。然后返回指令会打印出：

```
-1.2
```

> Magical!

神奇！

> ### 15.3.1 Binary operators

15.3.1 二元操作符

> OK, unary operators aren't *that* impressive. We still only ever have a single value on the stack. To really see some depth, we need binary operators. Lox has four binary arithmetic operators: addition, subtraction, multiplication, and division. We'll go ahead and implement them all at the same time.

好吧，一元运算符并没有那么令人印象深刻。我们的栈中仍然只有一个值。要真正看到一些深度，我们需要二元运算符。Lox中有四个二进制算术运算符：加、减、乘、除。我们接下来会同时实现它们。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CONSTANT,
  // 新增部分开始
  OP_ADD,
  OP_SUBTRACT,
  OP_MULTIPLY,
  OP_DIVIDE,
  // 新增部分结束
  OP_NEGATE,
```

> Back in the bytecode loop, they are executed like this:

回到字节码循环中，它们是这样执行的：

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_ADD:      BINARY_OP(+); break;
      case OP_SUBTRACT: BINARY_OP(-); break;
      case OP_MULTIPLY: BINARY_OP(*); break;
      case OP_DIVIDE:   BINARY_OP(/); break;
      // 新增部分结束
      case OP_NEGATE:   push(-pop()); break;
```

> The only difference between these four instructions is which underlying C operator they ultimately use to combine the two operands. Surrounding that core arithmetic expression is some boilerplate code to pull values off the stack and push the result. When we later add dynamic typing, that boilerplate will grow. To avoid repeating that code four times, I wrapped it up in a macro.

这四条指令之间唯一的区别是，它们最终使用哪一个底层C运算符来组合两个操作数。围绕这个核心算术表达式的是一些模板代码，用于从栈中获取数值，并将结果结果压入栈中。等我们后面添加动态类型时，这些模板代码会增加。为了避免这些代码重复出现四次，我将它包装在一个宏中。

vm.c，在 run()方法中添加代码：

```
#define READ_CONSTANT() (vm.chunk->constants.values[READ_BYTE()])
// 新增部分开始
#define BINARY_OP(op) \
    do { \
      double b = pop(); \
      double a = pop(); \
      push(a op b); \
    } while (false)
// 新增部分结束
  for (;;) {
```

> I admit this is a fairly adventurous use of the C preprocessor. I hesitated to do this, but you'll be glad in later chapters when we need to add the type checking for each operand and stuff. It would be a chore to walk you through the same code four times.

我承认这是对C预处理器的一次相当大胆的使用^11。我曾犹豫过要不要这么做，但在后面的章节中，等到我们需要为每个操作数和其它内容添加类型检查时，你就会高兴的。如果把相同的代码遍历四遍就太麻烦了。

> If you aren't familiar with the trick already, that outer `do while` loop probably looks really weird. This macro needs to expand to a series of statements. To be careful macro authors, we want to ensure those statements all end up in the same scope when the macro is expanded. Imagine if you defined:

如果你对这个技巧还不熟悉，那么外层的`do while`循环可能看起来非常奇怪。这个宏需要扩展为一系列语句。作为一个谨慎的宏作者，我们要确保当宏展开时，这些语句都在同一个作用域内。想象一下，如果你定义了：

```
#define WAKE_UP() makeCoffee(); drinkCoffee();
```

> And then used it like:

然后这样使用它：

```
if (morning) WAKE_UP();
```

> The intent is to execute both statements of the macro body only if `morning` is true. But it expands to:

其本意是在`morning`为true时执行这两个语句。但是宏展开结果为：

```
if (morning) makeCoffee(); drinkCoffee();;
```

> Oops. The `if` attaches only to the *first* statement. You might think you could fix this using a block.

哎呀。`if`只关联了第一条语句。您可能认为可以用代码块解决这个问题。

```
#define WAKE_UP() { makeCoffee(); drinkCoffee(); }
```

> That's better, but you still risk:

这样好一点，但还是有风险：

```
if (morning)
  WAKE_UP();
else
  sleepIn();
```

> Now you get a compile error on the `else` because of that trailing `;` after the macro's block. Using a `do while` loop in the macro looks funny, but it gives you a way to contain multiple statements inside a block that *also* permits a semicolon at the end.

现在你会在`else`子句遇到编译错误，因为在宏代码块后面有个`;`。在宏中使用`do while`循环看起来很滑稽，但它提供了一种方法，可以在一个代码块中包含多个语句，并且允许在末尾使用分号。

> Where were we? Right, so what the body of that macro does is straightforward. A binary operator takes two operands, so it pops twice. It performs the operation on those two values and then pushes the result.

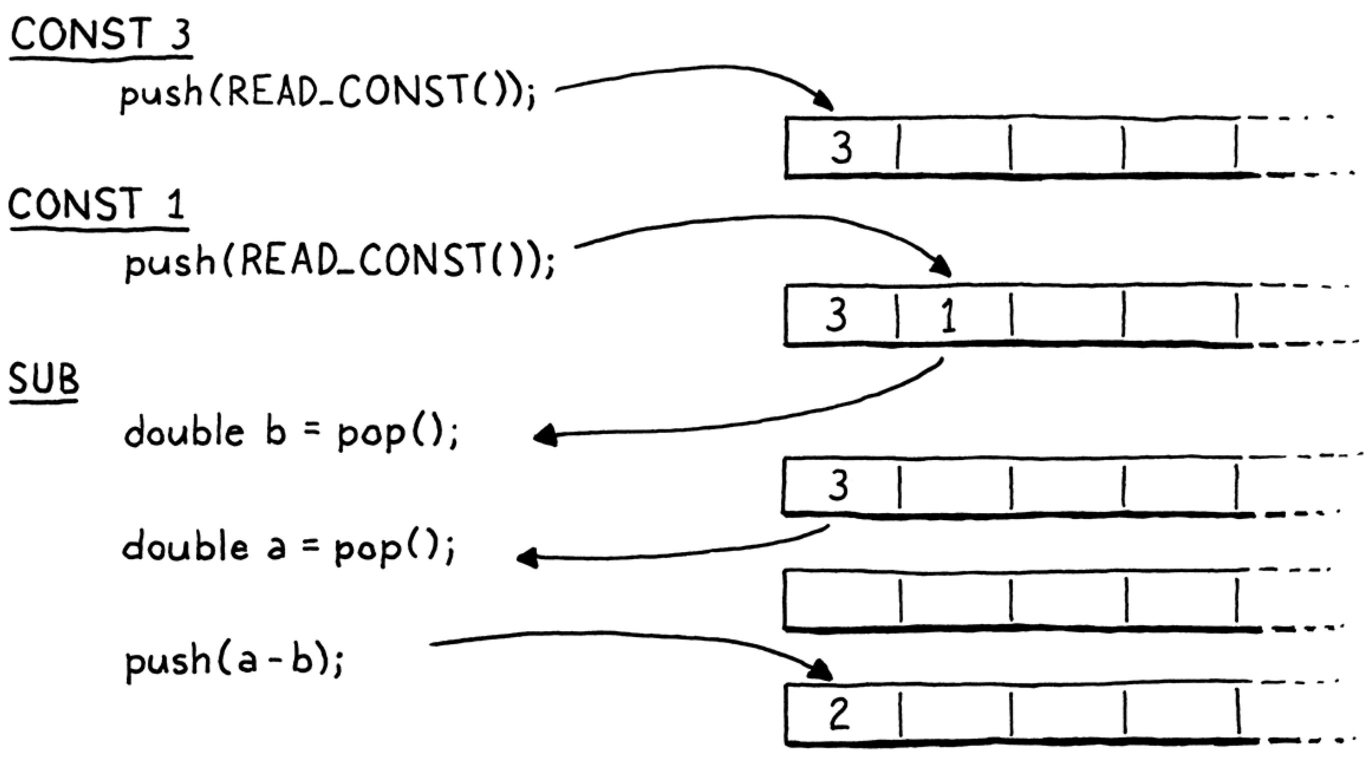
我们说到哪里了？对了，这个宏的主体所做的事情很直接。一个二元运算符接受两个操作数，因此会弹出栈两次，对这两个值执行操作，然后将结果压入栈。

> Pay close attention to the *order* of the two pops. Note that we assign the first popped operand to `b`, not `a`. It looks backwards. When the operands themselves are calculated, the left is evaluated first, then the right. That means the left operand gets pushed before the right operand. So the right operand will be on top of the stack. Thus, the first value we pop is `b`.

请密切注意这两次弹出栈的顺序。注意，我们将第一个弹出的操作数赋值给`b`，而不是`a`。在对操作数求值时，先计算左操作数，再计算右操作数。这意味着左操作数会在右操作数之前被压入栈，所以右侧的操作数在栈顶。因此，我们弹出的第一个值属于`b`。

> For example, if we compile `3 - 1`, the data flow between the instructions looks like so:

举例来说，如果我们编译`3-1`，指令之间的数据流看起来是这样的：



> As we did with the other macros inside `run()`, we clean up after ourselves at the end of the function.

正如我们在`run()`内的其它宏中做的那样，我们在函数结束时自行清理。

vm.c，在run()方法中添加代码：

```
#undef READ_CONSTANT
// 新增部分开始
#undef BINARY_OP
// 新增部分结束
}
```

> Last is disassembler support.

最后是反汇编器的支持。

debug.c，在 disassembleInstruction()方法中添加代码：

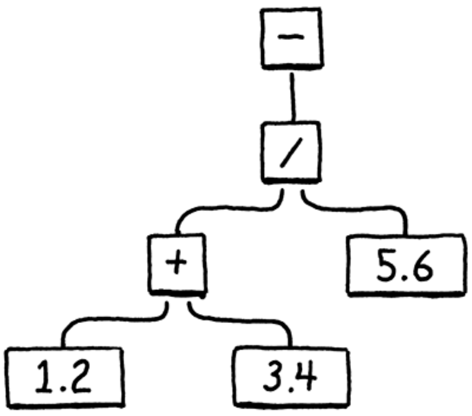
```
    case OP_CONSTANT:
      return constantInstruction("OP_CONSTANT", chunk, offset);
    // 新增部分开始
    case OP_ADD:
      return simpleInstruction("OP_ADD", offset);
    case OP_SUBTRACT:
      return simpleInstruction("OP_SUBTRACT", offset);
    case OP_MULTIPLY:
      return simpleInstruction("OP_MULTIPLY", offset);
    case OP_DIVIDE:
      return simpleInstruction("OP_DIVIDE", offset);
      // 新增部分结束
      case OP_NEGATE:
```

> The arithmetic instruction formats are simple, like `OP_RETURN`. Even though the arithmetic *operators* take operands—which are found on the stack—the arithmetic *bytecode instructions* do not.

算术指令的格式很简单，类似于`OP_RETURN`。即使算术运算符需要操作数（从堆栈中获取），算术的*字节码指令*也不需要。

> Let's put some of our new instructions through their paces by evaluating a larger expression:

我们通过计算一个更大的表达式来检验一些新指令。



> Building on our existing example chunk, here's the additional instructions we need to hand-compile that AST to bytecode.

在我们现有的示例代码块基础上，下面是我们将AST手动编译为字节码后需要添加的指令。

main.c，在main()方法中添加代码：

```
  int constant = addConstant(&chunk, 1.2);
  writeChunk(&chunk, OP_CONSTANT, 123);
  writeChunk(&chunk, constant, 123);
  // 新增部分开始
  constant = addConstant(&chunk, 3.4);
  writeChunk(&chunk, OP_CONSTANT, 123);
  writeChunk(&chunk, constant, 123);

  writeChunk(&chunk, OP_ADD, 123);

  constant = addConstant(&chunk, 5.6);
  writeChunk(&chunk, OP_CONSTANT, 123);
  writeChunk(&chunk, constant, 123);

  writeChunk(&chunk, OP_DIVIDE, 123);
  // 新增部分结束
  writeChunk(&chunk, OP_NEGATE, 123);

  writeChunk(&chunk, OP_RETURN, 123);
```

> The addition goes first. The instruction for the left constant, 1.2, is already there, so we add another for 3.4. Then we add those two using `OP_ADD`, leaving it on the stack. That covers the left side of the division. Next we push the 5.6, and divide the result of the addition by it. Finally, we negate the result of that.

首先进行加法运算。左边的常数1.2的指令已经存在了，所以我们再加一条3.4的指令。然后我们用`OP_ADD`把这两个值加起来，将结果压入堆栈中。这样就完成了除法的左操作数。接下来，我们压入5.6，并用加法的结果除以它。最后，我们对结果取负。

> Note how the output of the `OP_ADD` implicitly flows into being an operand of `OP_DIVIDE` without either instruction being directly coupled to each other. That's the magic of the stack. It lets us freely compose instructions without them needing any complexity or awareness of the data flow. The stack acts like a shared workspace that they all read from and write to.

注意，`OP_ADD`的输出如何隐式地变成了`OP_DIVIDE`的一个操作数，而这两条指令都没有直接耦合在一起。这就是堆栈的魔力。他让我们可以自由地编写指令，而无需任何复杂性或对于数据流的感知。堆栈就像一个共享工作区，它们都可以从中读取和写入。

> In this tiny example chunk, the stack still only gets two values tall, but when we start compiling Lox source to bytecode, we'll have chunks that use much more of the stack. In the meantime, try playing around with this hand-authored chunk to calculate different nested arithmetic expressions and see how values flow through the instructions and stack.

在这个小示例中，堆栈仍然只有两个值，但当我们开始将Lox源代码编译为字节码时，我们的代码块将使用更多的堆栈。同时，你可以试着用这个手工编写的字节码块来计算不同的嵌套算术表达式，看看数值是如何在指令和栈中流动的。

> You may as well get it out of your system now. This is the last chunk we'll build by hand. When we next revisit bytecode, we will be writing a compiler to generate it for us.

你不妨现在就把这块代码从系统中拿出来。这是我们手工构建的最后一个字节码块。当我们下次使用字节码时，我们将编写一个编译器来生成。

^3: x86、x64和CLR称其为 "IP"。68k、PowerPC、ARM、p-code和JVM称它为 "PC"，意为程序计数器。

^5: 如果你想了解其中一些技术，可以搜索“direct threaded code”、“jump table” 和 “computed goto”。^6: 显示地取消这些宏定义，可能会显得毫无必要，但C语言往往会惩罚粗心的用户，而C语言的预处理器更是如此。 ^7: 我们可以不指定计算顺序，让每个语言实现自行决定。这就为优化编译器重新排列算术表达式以提高效率留下了余地，即使是在操作数有明显副作用的情况下也是如此。C和Scheme没有指定求值顺序。Java规定了从左到右进行求值，就跟我们在Lox中所做的一样。
我认为指定这样的内容通常对用户更好。当表达式没有按照用户的直觉顺序进行求值时——可能在不同的实现中会有不同的顺序——要想弄清楚发生了什么，可能是非常痛苦的。 ^8: 堆——数据结构，不是内存管理——是另一个。还有Vaughan Pratt自顶向下的运算符优先级解析方案，我们会在适当的时候学习。 ^9: 稍微说明一下：基于堆栈的解释器并不是银弹。它们通常是够用的，但是JVM、CLR和JavaScript的现代化实现中都使用了复杂的即时编译管道，在动态中生成*更快的*本地代码。 ^10: 聪明的读者，你可能会问，那如果栈满了怎么办？C标准比您领先一步。C语言中允许数组指针正好指向数组末尾的下一个位置。 ^11: 你之前知道可以把操作符作为参数传递给宏吗？现在你知道了。预处理器并不关心操作符是不是C语言中的类，在它看来，这一切都只是文本符号。我知道，你已经感受到滥用预处理器的诱惑了，不是吗？

CHALLENGES

习题

1. > What bytecode instruction sequences would you generate for the following expressions:

   你会为以下表达式生成什么样的*字节码*指令序列：

   ```
   1 * 2 + 3
   1 + 2 * 3
   3 - 2 - 1
   1 + 2 * 3 - 4 / -5
   ```

   > (Remember that Lox does not have a syntax for negative number literals, so the `-5` is negating the number 5.)

   （请记得，Lox语法中没有负数字面量，所以`-5`是对数字5取负）

2. > If we really wanted a minimal instruction set, we could eliminate either `OP_NEGATE` or `OP_SUBTRACT`. Show the bytecode instruction sequence you would generate for:

   如果我们真的想要一个最小指令集，我们可以取消`OP_NEGATE`或`OP_SUBTRACT`。请写出你为下面的表达式生成的字节码指令序列：

   ```
   4 - 3 * -2
   ```

   > First, without using `OP_NEGATE`. Then, without using `OP_SUBTRACT`.
   >
   > Given the above, do you think it makes sense to have both instructions? Why or why not? Are there any other redundant instructions you would consider including?

   首先是不能使用 `OP_NEGATE`。然后，试一下不使用`OP_SUBTRACT`。

   综上所述，你认为同时拥有这两条指令有意义吗？为什么呢？还有没有其它指令可以考虑加入？

3. > Our VM's stack has a fixed size, and we don't check if pushing a value overflows it. This means the wrong series of instructions could cause our interpreter to crash or go into undefined behavior. Avoid that by dynamically growing the stack as needed.
   >
   > What are the costs and benefits of doing so?

   我们虚拟机的堆栈有一个固定大小，而且我们不会检查压入一个值是否会溢出。这意味着错误的指令序列可能会导致我们的解释器崩溃或进入未定义的行为。通过根据需求动态增长堆栈来避免这种情况。

   这样做的代价和好处是什么？

4. > To interpret `OP_NEGATE`, we pop the operand, negate the value, and then push the result. That's a simple implementation, but it increments and decrements `stackTop` unnecessarily, since the stack ends up the same height in the end. It might be faster to simply negate the value in place

> on the stack and leave `stackTop` alone. Try that and see if you can measure a performance difference.
>
> Are there other instructions where you can do a similar optimization?

为了解释`OP_NEGATE`，我们弹出操作数，对值取负，然后将结果压入栈。这是一个简单的实现，但它对`stackTop`进行了不必要的增减操作，因为栈最终的高度是相同的。简单地对栈中的值取负而不处理`stackTop`可能会更快。试一下，看看你是否能测出性能差异。

是否有其它指令可以做类似的优化？

DESIGN NOTE: REGISTER-BASED BYTECODE

设计笔记：基于寄存器的字节码

> For the remainder of this book, we'll meticulously implement an interpreter around a stack-based bytecode instruction set. There's another family of bytecode architectures out there—*register-based*. Despite the name, these bytecode instructions aren't quite as difficult to work with as the registers in an actual chip like x64. With real hardware registers, you usually have only a handful for the entire program, so you spend a lot of effort trying to use them efficiently and shuttling stuff in and out of them.
>
> In a register-based VM, you still have a stack. Temporary values still get pushed onto it and popped when no longer needed. The main difference is that instructions can read their inputs from anywhere in the stack and can store their outputs into specific stack slots.
>
> Take this little Lox script:
>
> ```
> var a = 1;
> var b = 2;
> var c = a + b;
> ```
>
> In our stack-based VM, the last statement will get compiled to something like:
>
> ```
> load <a>  // Read local variable a and push onto stack.
> load <b>  // Read local variable b and push onto stack.
> add       // Pop two values, add, push result.
> store <c> // Pop value and store in local variable c.
> ```
>
> (Don't worry if you don't fully understand the load and store instructions yet. We'll go over them in much greater detail when we implement variables.) We have four separate instructions. That means four times through the bytecode interpret loop, four instructions to decode and dispatch. It's at least seven bytes of code—four for the opcodes and another three for the operands identifying which locals to load and store. Three pushes and three pops. A lot of work!

> In a register-based instruction set, instructions can read from and store directly into local variables. The bytecode for the last statement above looks like:
>
> ```
> add <a> <b> <c> // Read values from a and b, add, store in c.
> ```
>
> The add instruction is bigger—it has three instruction operands that define where in the stack it reads its inputs from and writes the result to. But since local variables live on the stack, it can read directly from `a` and `b` and then store the result right into `c`.
>
> There's only a single instruction to decode and dispatch, and the whole thing fits in four bytes. Decoding is more complex because of the additional operands, but it's still a net win. There's no pushing and popping or other stack manipulation.
>
> The main implementation of Lua used to be stack-based. For Lua 5.0, the implementers switched to a register instruction set and noted a speed improvement. The amount of improvement, naturally, depends heavily on the details of the language semantics, specific instruction set, and compiler sophistication, but that should get your attention.
>
> The Lua dev team—Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo—wrote a *fantastic* paper on this, one of my all time favorite computer science papers, "The Implementation of Lua 5.0" (PDF).
>
> That raises the obvious question of why I'm going to spend the rest of the book doing a stack-based bytecode. Register VMs are neat, but they are quite a bit harder to write a compiler for. For what is likely to be your very first compiler, I wanted to stick with an instruction set that's easy to generate and easy to execute. Stack-based bytecode is marvelously simple.
>
> It's also *much* better known in the literature and the community. Even though you may eventually move to something more advanced, it's a good common ground to share with the rest of your language hacker peers.

在本书的其余部分，我们将围绕基于堆栈的字节码指令集精心实现一个解释器。此外还有另一种字节码架构——基于寄存器。尽管名称如此，但这些字节码指令并不像x64这样的真实芯片中的寄存器那样难以操作。对于真正的硬件寄存器，整个程序通常只用少数几个，所以你要花很多精力来有效地使用它们，并把数据存入或取出。（基于寄存器的字节码更接近于SPARC芯片支持的寄存器窗口）

在一个基于寄存器的虚拟机中，仍然有一个栈。临时值还是被压入栈中，当不再需要时再被弹出。主要的区别是，指令可以从栈的任意位置读取它们的输入值，并可以将它们的输出值存储到任一指定的槽中。

以Lox脚本为例：

```
var a = 1;
var b = 2;
var c = a + b;
```

在我们基于堆栈的虚拟机中，最后一条指令的编译结果类似于：

```
load <a>  // 读取局部变量a，并将其压入栈
load <b>  // 读取局部变量b，并将其压入栈
add       // 弹出两个值，相加，将结果压入栈
store <c> // 弹出值，并存入局部变量c
```

（如果你还没有完全理解加载load和存储store指令，也不用担心。我们会在实现变量时详细地讨论它们）我们有四条独立的指令，这意味着会有四次字节码解释循环，四条指令需要解码和调度。这至少包含7个字节的代码——四个字节是操作码，另外三个是操作数，用于标识要加载和存储哪些局部变量。三次入栈，三次出栈，工作量很大！

在基于寄存器的指令集中，指令可以直接对局部变量进行读取和存储。上面最后一条语句的字节码如下所示：

```
add <a> <b> <c> // 从a和b中读取值，相加，并存储到c中
```

add指令比之前更大——有三个指令操作数，定义了从堆栈的哪个位置读取输入，并将结果写入哪个位置。但由于局部变量在堆栈中，它可以直接从`a`和`b`中读取数据，如何将结果存入`c`中。

只有一条指令需要解码和调度，整个程序只需要四个字节。由于有了额外的操作数，解码变得更加复杂，但相比之下它仍然是更优秀的。没有压入和弹出或其它堆栈操作。

Lua的实现曾经是基于堆栈的。到了Lua 5.0，实现切换到了寄存器指令集，并注意到速度有所提高。当然，提高的幅度很大程度上取决于语言语义的细节、特定指令集和编译器复杂性，但这应该引起你的注意。

这就引出了一个显而易见的问题：我为什么要在本书的剩余部分做一个基于堆栈的字节码。寄存器虚拟机是很好的，但要为它们编写编译器却相当困难。考虑到这可能是你写的第一个编译器，我想坚持使用一个易于生成和易于执行的指令集。基于堆栈的字节码是非常简单的。

它的文献和社区中也更广为人知。即使你最终可能会转向更高级的东西，这也是一个你可以与其他语言开发者分享的很好的共同点。

16.按需扫描 Scanning on Demand

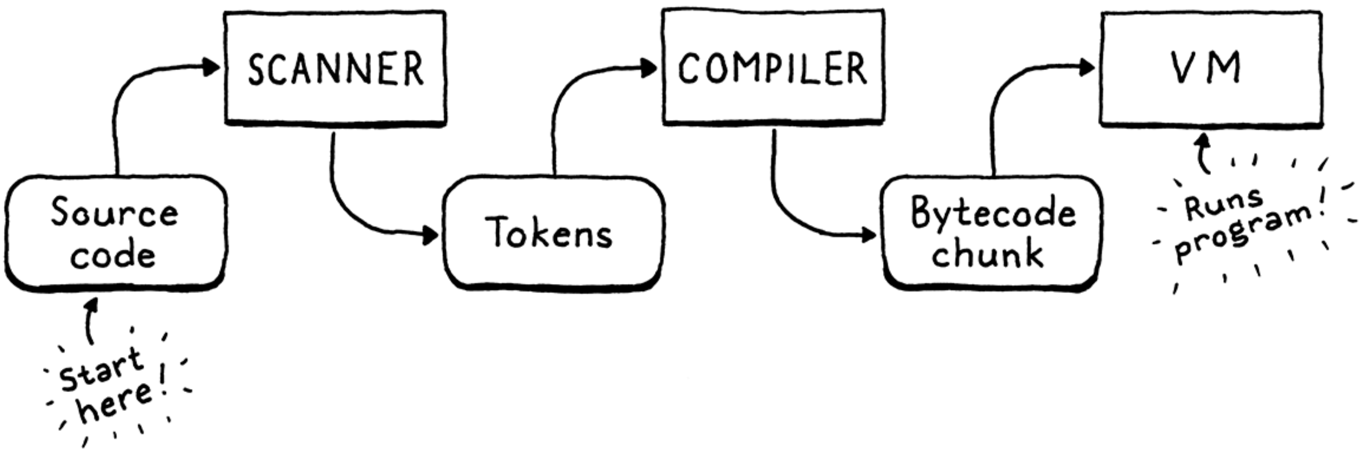
> Literature is idiosyncratic arrangements in horizontal lines in only twenty-six phonetic symbols, ten Arabic numbers, and about eight punctuation marks.
>
> —— Kurt Vonnegut, *Like Shaking Hands With God: A Conversation about Writing*

文学就是26个字母、10个阿拉伯数字和大概8个标点符号的水平排列。（冯尼古特《像与上帝握手：关于写作的谈话》）

> Our second interpreter, clox, has three phases—scanner, compiler, and virtual machine. A data structure joins each pair of phases. Tokens flow from scanner to compiler, and chunks of bytecode from compiler to VM. We began our implementation near the end with chunks and the VM. Now, we're going to hop back to the beginning and build a scanner that makes tokens. In the next chapter, we'll tie the two ends together with our bytecode compiler.

我们的第二个解释器clox分为三个阶段——扫描器、编译器和虚拟机。每两个阶段之间有一个数据结构进行衔接。词法标识从扫描器流入编译器，字节码块从编译器流向虚拟机。我们是从尾部开始先实现了字节码块和虚

拟机。现在，我们要回到起点，构建一个生成词法标识的扫描器。在下一章中，我们将用字节码编译器将这两部分连接起来。



> I'll admit, this is not the most exciting chapter in the book. With two implementations of the same language, there's bound to be some redundancy. I did sneak in a few interesting differences compared to jlox's scanner. Read on to see what they are.

我承认，这并不是书中最精彩的一章。对于同一种语言的两个实现，肯定会有一些冗余。与jlox的扫描器相比，我确实添加了一些有趣的差异点。往下读，看看它们是什么。

16.1 Spinning Up the Interpreter

> Now that we're building the front end, we can get clox running like a real interpreter. No more hand-authored chunks of bytecode. It's time for a REPL and script loading. Tear out most of the code in `main()` and replace it with:

现在我们正在构建前端，我们可以让clox像一个真正的解释器一样运行。不需要再手动编写字节码块。现在是时候实现REPL和脚本加载了。删除`main()`方法中的大部分代码，替换成：

main.c，在 main()方法中替换26 行：

```c
int main(int argc, const char* argv[]) {
  initVM();
  // 替换部分开始
  if (argc == 1) {
    repl();
  } else if (argc == 2) {
    runFile(argv[1]);
  } else {
    fprintf(stderr, "Usage: clox [path]\n");
    exit(64);
  }

  freeVM();
  // 替换部分结束
  return 0;
}
```

> If you pass no arguments to the executable, you are dropped into the REPL. A single command line argument is understood to be the path to a script to run.

如果你没有向可执行文件传递任何参数，就会进入REPL。如果传入一个参数，就将其当做要运行的脚本的路径^1。

> We'll need a few system headers, so let's get them all out of the way.

我们需要一些系统头文件，所以把它们都列出来。

main.c，在文件顶部添加：

```
// 新增部分开始
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// 新增部分结束
#include "common.h"
```

> Next, we get the REPL up and REPL-ing.

接下来，我们启动REPL并运行。

main.c，添加代码：

```
#include "vm.h"
// 新增部分开始
static void repl() {
  char line[1024];
  for (;;) {
    printf("> ");

    if (!fgets(line, sizeof(line), stdin)) {
      printf("\n");
      break;
    }

    interpret(line);
  }
}
// 新增部分结束
```

> A quality REPL handles input that spans multiple lines gracefully and doesn't have a hardcoded line length limit. This REPL here is a little more, ahem, austere, but it's fine for our purposes.

一个高质量的REPL可以优雅地处理多行的输入，并且没有硬编码的行长度限制。这里的REPL有点……简朴，但足以满足我们的需求。

> The real work happens in `interpret()`. We'll get to that soon, but first let's take care of loading scripts.

真正的工作发生在`interpret()`中。我们很快会讲到这个问题，但首先让我们来看看如何加载脚本。

main.c，在repl()方法后添加：

```
// 新增部分开始
static void runFile(const char* path) {
  char* source = readFile(path);
  InterpretResult result = interpret(source);
  free(source);

  if (result == INTERPRET_COMPILE_ERROR) exit(65);
  if (result == INTERPRET_RUNTIME_ERROR) exit(70);
}
// 新增部分结束
```

> We read the file and execute the resulting string of Lox source code. Then, based on the result of that, we set the exit code appropriately because we're scrupulous tool builders and care about little details like that.

我们读取文件并执行生成的Lox源码字符串。然后，根据其结果，我们适当地设置退出码，因为我们是严谨的工具制作者，并且关心这样的小细节。

> We also need to free the source code string because `readFile()` dynamically allocates it and passes ownership to its caller. That function looks like this:

我们还需要释放源代码字符串，因为`readFile()`会动态地分配内存，并将所有权传递给它的调用者^2。这个函数看起来像这样：

main.c，在 repl()方法后添加代码：

```
static char* readFile(const char* path) {
  FILE* file = fopen(path, "rb");

  fseek(file, 0L, SEEK_END);
  size_t fileSize = ftell(file);
  rewind(file);

  char* buffer = (char*)malloc(fileSize + 1);
  size_t bytesRead = fread(buffer, sizeof(char), fileSize, file);
  buffer[bytesRead] = '\0';

  fclose(file);
  return buffer;
}
```

> Like a lot of C code, it takes more effort than it seems like it should, especially for a language expressly designed for operating systems. The difficult part is that we want to allocate a big enough string to read the whole file, but we don't know how big the file is until we've read it.

像很多C语言代码一样，它所花费的精力比看起来要多，尤其是对于一门专为操作系统而设计的语言而言。困难的地方在于，我们想分配一个足以读取整个文件的字符串，但是我们在读取文件之前并不知道它有多大。

> The code here is the classic trick to solve that. We open the file, but before reading it, we seek to the very end using `fseek()`. Then we call `ftell()` which tells us how many bytes we are from the start of the file. Since we seeked (sought?) to the end, that's the size. We rewind back to the beginning, allocate a string of that size, and read the whole file in a single batch.

这里的代码是解决这个问题的经典技巧。我们打开文件，但是在读之前，先通过`fseek()`寻找到文件的最末端。接下来我们调用`ftell()`，它会告诉我们里文件起始点有多少字节。既然我们定位到了最末端，那它就是文件大小^3。我们退回到起始位置，分配一个相同大小的字符串，然后一次性读取整个文件。

> So we're done, right? Not quite. These function calls, like most calls in the C standard library, can fail. If this were Java, the failures would be thrown as exceptions and automatically unwind the stack so we wouldn't *really* need to handle them. In C, if we don't check for them, they silently get ignored.

这样就完成了吗？不完全是。这些函数调用，像C语言标准库中的大多数调用一样，可能会失败。如果是在Java中，这些失败会被当做异常抛出，并自动清除堆栈，所以我们实际上并不需要处理它们。但在C语言中，如果我们不检查，它们就会被忽略。

> This isn't really a book on good C programming practice, but I hate to encourage bad style, so let's go ahead and handle the errors. It's good for us, like eating our vegetables or flossing.

这并不是一本关于良好C语言编程实践的书，但我讨厌鼓励糟糕的编程风格，所以让我们继续处理这些错误。这对我们有好处，就像吃蔬菜或使用牙线清洁牙齿一样。

> Fortunately, we don't need to do anything particularly clever if a failure occurs. If we can't correctly read the user's script, all we can really do is tell the user and exit the interpreter gracefully. First of all, we might fail to open the file.

幸运地是，如果发生故障，我们不需要特别聪明的做法。如果我们不能正确地读取用户的脚本，我们真正能做的就是告诉用户并优雅地退出解释器。首先，我们可能无法打开文件。

main.c，在readFile()方法中添加代码：

```
  FILE* file = fopen(path, "rb");
  // 新增部分开始
  if (file == NULL) {
    fprintf(stderr, "Could not open file \"%s\".\n", path);
    exit(74);
  }
  // 新增部分结束
  fseek(file, 0L, SEEK_END);
```

> This can happen if the file doesn't exist or the user doesn't have access to it. It's pretty common—people mistype paths all the time.

如果文件不存在或用户没有访问权限，就会发生这种情况。这是很常见的——人们经常会输入错误的路径。

> This failure is much rarer:

下面这种错误要少见得多：

main.c，在readFile()方法中添加代码：

```
char* buffer = (char*)malloc(fileSize + 1);
// 新增部分开始
if (buffer == NULL) {
  fprintf(stderr, "Not enough memory to read \"%s\".\n", path);
  exit(74);
}
// 新增部分结束
size_t bytesRead = fread(buffer, sizeof(char), fileSize, file);
```

> If we can't even allocate enough memory to read the Lox script, the user's probably got bigger problems to worry about, but we should do our best to at least let them know.

如果我们甚至不能分配足够的内存来读取Lox脚本，那么用户可能会有更大的问题需要担心，但我们至少应该尽最大努力让他们知道。

> Finally, the read itself may fail.

最后，读取本身可能会失败。

main.c，在readFile()方法中添加代码：

```
size_t bytesRead = fread(buffer, sizeof(char), fileSize, file);
// 新增部分开始
if (bytesRead < fileSize) {
  fprintf(stderr, "Could not read file \"%s\".\n", path);
  exit(74);
}
// 新增部分结束
buffer[bytesRead] = '\0';
```

> This is also unlikely. Actually, the calls to `fseek()`, `ftell()`, and `rewind()` could theoretically fail too, but let's not go too far off in the weeds, shall we?

这也是不大可能发生的。实际上，`fseek()`, `ftell()`, 和`rewind()` 的调用在理论上也可能会失败，但是我们不要太过深入，好吗？

> ### 16.1.1 Opening the compilation pipeline

16.1.1 开启编译管道

> We've got ourselves a string of Lox source code, so now we're ready to set up a pipeline to scan, compile, and execute it. It's driven by `interpret()`. Right now, that function runs our old hardcoded test chunk. Let's change it to something closer to its final incarnation.

我们已经得到了Lox源代码字符串，所以现在我们准备建立一个管道来扫描、编译和执行它。管道是由`interpret()`驱动的。现在，该函数运行的是旧的硬编码测试字节码块。我们来把它改成更接近其最终形态的东西。

vm.h，函数interpret()中替换1行：

```
void freeVM();
// 替换部分开始
InterpretResult interpret(const char* source);
// 替换部分结束
void push(Value value);
```

> Where before we passed in a Chunk, now we pass in the string of source code. Here's the new implementation:

以前我们传入一个字节码块，现在我们传入的是源代码的字符串。下面是新的实现：

vm.c，函数interpret()中替换4行：

```
// 替换部分开始
InterpretResult interpret(const char* source) {
  compile(source);
  return INTERPRET_OK;
//  替换部分结束
}
```

> We won't build the actual *compiler* yet in this chapter, but we can start laying out its structure. It lives in a new module.

在本章中，我们还不会构建真正的*编译器*，但我们可以开始布局它的结构。它存在于一个新的模块中。

vm.c，添加代码：

```
#include "common.h"
// 新增部分开始
#include "compiler.h"
// 新增部分结束
#include "debug.h"
```

> For now, the one function in it is declared like so:

目前，其中有一个函数声明如下：

compiler.h，创建新文件：

```
#ifndef clox_compiler_h
#define clox_compiler_h

void compile(const char* source);

#endif
```

> That signature will change, but it gets us going.

这个签名以后会变，但现在足以让我们继续工作。

> The first phase of compilation is scanning—the thing we're doing in this chapter—so right now all the compiler does is set that up.

编译的第一阶段是扫描——即我们在本章中要做的事情——所以现在编译器所做的就是设置扫描。

compiler.c，创建新文件：

```
#include <stdio.h>

#include "common.h"
#include "compiler.h"
#include "scanner.h"

void compile(const char* source) {
  initScanner(source);
}
```

> This will also grow in later chapters, naturally.

当然，这在后面的章节中也会继续扩展。

> ## 16.1.2 The scanner scans

16.1.2 扫描器扫描

> There are still a few more feet of scaffolding to stand up before we can start writing useful code. First, a new header:

在我们开始编写实际有用的代码之前，还有一些脚手架需要先搭建起来。首先，是一个新的头文件：

scanner.h，创建新文件：

```
#ifndef clox_scanner_h
#define clox_scanner_h
```

```
void initScanner(const char* source);

#endif
```

> And its corresponding implementation:

还有其对应的实现：

scanner.c，创建新文件：

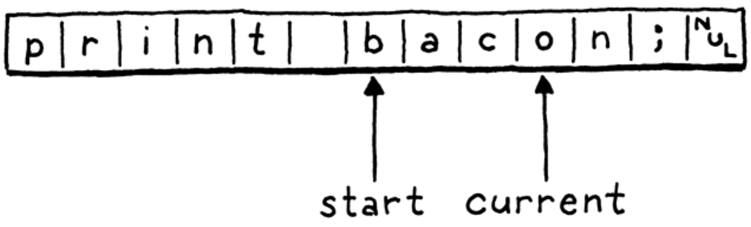
```
#include <stdio.h>
#include <string.h>

#include "common.h"
#include "scanner.h"

typedef struct {
  const char* start;
  const char* current;
  int line;
} Scanner;

Scanner scanner;
```

> As our scanner chews through the user's source code, it tracks how far it's gone. Like we did with the VM, we wrap that state in a struct and then create a single top-level module variable of that type so we don't have to pass it around all of the various functions.

当我们的扫描器一点点处理用户的源代码时，它会跟踪自己已经走了多远。就像我们在虚拟机中所做的那样，我们将状态封装在一个结构体中，然后创建一个该类型的顶层模块变量，这样就不必在所有的函数之间传递它。

> There are surprisingly few fields. The `start` pointer marks the beginning of the current lexeme being scanned, and `current` points to the current character being looked at.

这里的字段少得惊人。`start`指针标识正在被扫描的词素的起点，而`current`指针指向当前正在查看的字符。



> We have a `line` field to track what line the current lexeme is on for error reporting. That's it! We don't even keep a pointer to the beginning of the source code string. The scanner works its way through the code once and is done after that.

我们还有一个`line`字段，用于跟踪当前词素在哪一行，以便进行错误报告。就是这样！我们甚至没有保留指向源代码字符串起点的指针。扫描器只处理一遍代码，然后就结束了。

> Since we have some state, we should initialize it.

因为我们有一些状态，我们还应该初始化它。

scanner.c，在变量scanner后添加代码：

```
void initScanner(const char* source) {
  scanner.start = source;
  scanner.current = source;
  scanner.line = 1;
}
```

> We start at the very first character on the very first line, like a runner crouched at the starting line.

我们从第一行的第一个字符开始，就像一个运动员蹲在起跑线上。

16.2 A Token at a Time

16.2 一次一个标识

> In jlox, when the starting gun went off, the scanner raced ahead and eagerly scanned the whole program, returning a list of tokens. This would be a challenge in clox. We'd need some sort of growable array or list to store the tokens in. We'd need to manage allocating and freeing the tokens, and the collection itself. That's a lot of code, and a lot of memory churn.

在jlox中，当发令枪响起时，扫描器飞快地前进，急切地扫描整个程序，并返回一个词法标识序列。这在clox中有点困难。我们需要某种可增长的数组或列表来存储标识。我们需要管理标识的分配和释放，以及集合本身。这需要大量的代码和大量的内存。

> At any point in time, the compiler needs only one or two tokens—remember our grammar requires only a single token of lookahead—so we don't need to keep them *all* around at the same time. Instead, the simplest solution is to not scan a token until the compiler needs one. When the scanner provides one, it returns the token by value. It doesn't need to dynamically allocate anything—it can just pass tokens around on the C stack.

在任何时间点，编译器只需要一个或两个词法标识——记住我们的语法只需要前瞻一个词法标识——所以我们不需要同时保留它们。相反，最简单的解决方案是在编译器需要标识的时候再去扫描。当扫描器提供一个标识时，它按值返回标识。它不需要动态分配任何东西——只需要在C栈上传递词法标识即可。

> Unfortunately, we don't have a compiler yet that can ask the scanner for tokens, so the scanner will just sit there doing nothing. To kick it into action, we'll write some temporary code to drive it.

不巧的是，我们还没有可以向扫描器请求词法标识的编译器，所以扫描器只能干等着什么也不做。为了让它工作起来，我们要编写一些临时代码来驱动它^4。

compiler.c，在compile()方法中添加代码：

```
  initScanner(source);
  // 新增部分开始
```

```
    int line = -1;
    for (;;) {
      Token token = scanToken();
      if (token.line != line) {
        printf("%4d ", token.line);
        line = token.line;
      } else {
        printf("   | ");
      }
      printf("%2d '%.*s'\n", token.type, token.length, token.start);

      if (token.type == TOKEN_EOF) break;
    }
    // 新增部分结束
  }
```

> This loops indefinitely. Each turn through the loop, it scans one token and prints it. When it reaches a special “end of file” token or an error, it stops. For example, if we run the interpreter on this program:

这个循环是无限的。每循环一次，它就会扫描一个词法标识并打印出来。当它遇到特殊的“文件结束”标识或错误时，就会停止。例如，如果我们对下面的程序运行解释器：

```
print 1 + 2;
```

> It prints out:

就会打印出：

```
   1 31 'print'
   | 21 '1'
   |  7 '+'
   | 21 '2'
   |  8 ';'
   2 39 ''
```

> The first column is the line number, the second is the numeric value of the token type, and then finally the lexeme. That last empty lexeme on line 2 is the EOF token.

第一列是行号，第二列是标识类型的数值，最后是词素。第2行中最后一个空词素就是EOF标识。

> The goal for the rest of the chapter is to make that blob of code work by implementing this key function:

本章其余部分的目标就是通过实现下面这个关键函数，使这块代码能正常工作：

scanner.h，在initScanner()方法后添加：

```
void initScanner(const char* source);
// 新增部分开始
Token scanToken();
// 新增部分结束
#endif
```

> Each call scans and returns the next token in the source code. A token looks like this:

该函数的每次调用都会扫描并返回源代码中的下一个词法标识。一个词法标识结构如下：

scanner.h，添加代码：

```
#define clox_scanner_h
// 新增部分开始
typedef struct {
  TokenType type;
  const char* start;
  int length;
  int line;
} Token;
// 新增部分结束
void initScanner(const char* source);
```

> It's pretty similar to jlox's Token class. We have an enum identifying what type of token it is—number, identifier, `+` operator, etc. The enum is virtually identical to the one in jlox, so let's just hammer out the whole thing.

它和jlox中的Token类很相似。我们用一个枚举来标记它是什么类型的词法标识——数字、标识符、`+`运算符等等。这个枚举与jlox中的枚举几乎完全相同，所以我们直接来敲定整个事情。

scanner.h，添加代码：

```
#ifndef clox_scanner_h
#define clox_scanner_h
// 新增部分开始
typedef enum {
  // Single-character tokens. 单字符词法
  TOKEN_LEFT_PAREN, TOKEN_RIGHT_PAREN,
  TOKEN_LEFT_BRACE, TOKEN_RIGHT_BRACE,
  TOKEN_COMMA, TOKEN_DOT, TOKEN_MINUS, TOKEN_PLUS,
  TOKEN_SEMICOLON, TOKEN_SLASH, TOKEN_STAR,
  // One or two character tokens. 一或两字符词法
  TOKEN_BANG, TOKEN_BANG_EQUAL,
  TOKEN_EQUAL, TOKEN_EQUAL_EQUAL,
  TOKEN_GREATER, TOKEN_GREATER_EQUAL,
  TOKEN_LESS, TOKEN_LESS_EQUAL,
  // Literals. 字面量
  TOKEN_IDENTIFIER, TOKEN_STRING, TOKEN_NUMBER,
  // Keywords. 关键字
  TOKEN_AND, TOKEN_CLASS, TOKEN_ELSE, TOKEN_FALSE,
  TOKEN_FOR, TOKEN_FUN, TOKEN_IF, TOKEN_NIL, TOKEN_OR,
  TOKEN_PRINT, TOKEN_RETURN, TOKEN_SUPER, TOKEN_THIS,
  TOKEN_TRUE, TOKEN_VAR, TOKEN_WHILE,

  TOKEN_ERROR, TOKEN_EOF
} TokenType;

// 新增部分结束
typedef struct {
```

> Aside from prefixing all the names with `TOKEN_` (since C tosses enum names in the top-level namespace) the only difference is that extra `TOKEN_ERROR` type. What's that about?

除了在所有名称前都加上`TOKEN_`前缀（因为C语言会将枚举名称抛出到顶层命名空间）之外，唯一的区别就是多了一个`TOKEN_ERROR`类型。那是什么呢？

> There are only a couple of errors that get detected during scanning: unterminated strings and unrecognized characters. In jlox, the scanner reports those itself. In clox, the scanner produces a synthetic "error" token for that error and passes it over to the compiler. This way, the compiler knows an error occurred and can kick off error recovery before reporting it.

在扫描过程中只会检测到几种错误：未终止的字符串和无法识别的字符。在jlox中，扫描器会自己报告这些错误。在clox中，扫描器会针对这些错误生成一个合成的"错误"标识，并将其传递给编译器。这样一来，编译器就知道发生了一个错误，并可以在报告错误之前启动错误恢复。

> The novel part in clox's Token type is how it represents the lexeme. In jlox, each Token stored the lexeme as its own separate little Java string. If we did that for clox, we'd have to figure out how to manage the memory for those strings. That's especially hard since we pass tokens by value—multiple tokens could point to the same lexeme string. Ownership gets weird.

在clox的Token类型中，新颖之处在于它如何表示一个词素。在jlox中，每个Token将词素保存到其单独的Java字符串中。如果我们在clox中也这样做，我们就必须想办法管理这些字符串的内存。这非常困难，因为我们是通过值传递词法标识的——多个标识可能指向相同的词素字符串。所有权会变得混乱。

> Instead, we use the original source string as our character store. We represent a lexeme by a pointer to its first character and the number of characters it contains. This means we don't need to worry about managing memory for lexemes at all and we can freely copy tokens around. As long as the main source code string outlives all of the tokens, everything works fine.

相反，我们将原始的源码字符串作为我们的字符存储。我们用指向第一个字符的指针和其中包含的字符数来表示一个词素。这意味着我们完全不需要担心管理词素的内存，而且我们可以自由地复制词法标识。只要主源码字符串的寿命超过所有词法标识，一切都可以正常工作^5。

> ### 16.2.1 Scanning tokens

16.2.1 扫描标识

> We're ready to scan some tokens. We'll work our way up to the complete implementation, starting with this:

我们已经准备好扫描一些标识了。我们将从下面的代码开始，逐步达成完整的实现：

scanner.c，在initScanner()方法后添加代码：

```
Token scanToken() {
  scanner.start = scanner.current;

  if (isAtEnd()) return makeToken(TOKEN_EOF);

  return errorToken("Unexpected character.");
}
```

> Since each call to this function scans a complete token, we know we are at the beginning of a new token when we enter the function. Thus, we set `scanner.start` to point to the current character so we remember where the lexeme we're about to scan starts.

由于对该函数的每次调用都会扫描一个完整的词法标识，所以当我们进入该函数时，就知道我们正处于一个新词法标识的开始处。因此，我们将`scanner.start`设置为指向当前字符，这样我们就能记住我们将要扫描的词素的开始位置。

> Then we check to see if we've reached the end of the source code. If so, we return an EOF token and stop. This is a sentinel value that signals to the compiler to stop asking for more tokens.

然后检查是否已达到源代码的结尾。如果是，我们返回一个EOF标识并停止。这是一个标记值，它向编译器发出信号，停止请求更多标记。

> If we aren't at the end, we do some . . . stuff . . . to scan the next token. But we haven't written that code yet. We'll get to that soon. If that code doesn't successfully scan and return a token, then we reach the end of the function. That must mean we're at a character that the scanner can't recognize, so we return an error token for that.

如果我们没有达到结尾，我们会做一些......事情......来扫描下一个标识。但我们还没有写这些代码。我们很快就会讲到。如果这段代码没有成功扫描并返回一个词法标识，那么我们就到达了函数的终点。这肯定意味着我们遇到了一个扫描器无法识别的字符，所以我们为此返回一个错误标识。

> This function relies on a couple of helpers, most of which are familiar from jlox. First up:

这个函数依赖于几个辅助函数，其中大部分都是在jlox中已熟悉的。首先是：

scanner.c，在initScanner()方法后添加代码：

```
static bool isAtEnd() {
  return *scanner.current == '\0';
}
```

> We require the source string to be a good null-terminated C string. If the current character is the null byte, then we've reached the end.

我们要求源字符串是一个良好的以null结尾的C字符串。如果当前字符是null字节，那我们就到达了终点。

> To create a token, we have this constructor-like function:

要创建一个标识，我们还需要这个类似于构造函数的函数：

scanner.c，在isAtEnd()方法后添加代码：

```
static Token makeToken(TokenType type) {
  Token token;
  token.type = type;
  token.start = scanner.start;
  token.length = (int)(scanner.current - scanner.start);
  token.line = scanner.line;
  return token;
}
```

> It uses the scanner's `start` and `current` pointers to capture the token's lexeme. It sets a couple of other obvious fields then returns the token. It has a sister function for returning error tokens.

其中使用扫描器的`start`和`current`指针来捕获标识的词素。它还设置了其它几个明显的字段，如何返回标识。它还有一个用于返回错误标识的姊妹函数。

scanner.c，在makeToken()方法后添加代码：

```
static Token errorToken(const char* message) {
  Token token;
  token.type = TOKEN_ERROR;
  token.start = message;
  token.length = (int)strlen(message);
  token.line = scanner.line;
  return token;
}
```

> The only difference is that the "lexeme" points to the error message string instead of pointing into the user's source code. Again, we need to ensure that the error message sticks around long enough for the compiler to read it. In practice, we only ever call this function with C string literals. Those are constant and eternal, so we're fine.

唯一的区别在于，"词素"指向错误信息字符串而不是用户的源代码。同样，我们需要确保错误信息能保持足够长的时间，以便编译器能够读取它。在实践中，我们只会用C语言的字符串字面量来调用这个函数。它们是恒定不变的，所以我们不会有问题。

> What we have now is basically a working scanner for a language with an empty lexical grammar. Since the grammar has no productions, every character is an error. That's not exactly a fun language to

> program in, so let's fill in the rules.

我们现在所拥有的是一个基本可用的扫描器，用于扫描空语法语言。因为语法没有产生式，所以每个字符都是一个错误。这并不是一种有趣的编程语言，所以让我们把规则填进去。

> ## 16.3 A Lexical Grammar for Lox

16.3 Lox语法

> The simplest tokens are only a single character. We recognize those like so:

最简单的词法标识只有一个字符。我们这样来识别它们：

scanner.c，在scanToken()方法中添加代码：

```
  if (isAtEnd()) return makeToken(TOKEN_EOF);
  // 新增部分开始
  char c = advance();

  switch (c) {
    case '(': return makeToken(TOKEN_LEFT_PAREN);
    case ')': return makeToken(TOKEN_RIGHT_PAREN);
    case '{': return makeToken(TOKEN_LEFT_BRACE);
    case '}': return makeToken(TOKEN_RIGHT_BRACE);
    case ';': return makeToken(TOKEN_SEMICOLON);
    case ',': return makeToken(TOKEN_COMMA);
    case '.': return makeToken(TOKEN_DOT);
    case '-': return makeToken(TOKEN_MINUS);
    case '+': return makeToken(TOKEN_PLUS);
    case '/': return makeToken(TOKEN_SLASH);
    case '*': return makeToken(TOKEN_STAR);
  }
  // 新增部分结束
  return errorToken("Unexpected character.");
```

> We read the next character from the source code, and then do a straightforward switch to see if it matches any of Lox's one-character lexemes. To read the next character, we use a new helper which consumes the current character and returns it.

我们从源代码中读取下一个字符，然后做一个简单的switch判断，看它是否与Lox中的某个单字符词素相匹配。为了读取下一个字符，我们使用一个新的辅助函数，它会消费当前字符并将其返回。

scanner.c，在isAtEnd()方法后添加代码：

```
static char advance() {
  scanner.current++;
  return scanner.current[-1];
}
```

> Next up are the two-character punctuation tokens like `!=` and `>=`. Each of these also has a corresponding single-character token. That means that when we see a character like `!`, we don't know if we're in a `!` token or a `!=` until we look at the next character too. We handle those like so:

接下来是两个字符的符号，如`!=`和`>=`，其中每一个都包含对应的单字符标识。这意味着，当我们看到一个像`!`这样的字符时，我们只有看到下一个字符，才能确认当前是`!`标识还是`!=`标识。我们是这样处理的：

scanner.c in *scanToken*()

```
      case '*': return makeToken(TOKEN_STAR);
      // 新增部分开始
      case '!':
        return makeToken(
            match('=') ? TOKEN_BANG_EQUAL : TOKEN_BANG);
      case '=':
        return makeToken(
            match('=') ? TOKEN_EQUAL_EQUAL : TOKEN_EQUAL);
      case '<':
        return makeToken(
            match('=') ? TOKEN_LESS_EQUAL : TOKEN_LESS);
      case '>':
        return makeToken(
            match('=') ? TOKEN_GREATER_EQUAL : TOKEN_GREATER);
      // 新增部分结束
    }
```

> After consuming the first character, we look for an `=`. If found, we consume it and return the corresponding two-character token. Otherwise, we leave the current character alone (so it can be part of the *next* token) and return the appropriate one-character token.

在消费第一个字符之后，我们会尝试寻找一个`=`。如果找到了，我们就消费它并返回对应的双字符标识。否则，我们就不处理当前字符（这样它就是下一个标识的一部分）并返回相应的单字符标识。

> That logic for conditionally consuming the second character lives here:

这个有条件地消费第二个字符的逻辑如下：

scanner.c，在advance()方法后添加：

```
static bool match(char expected) {
  if (isAtEnd()) return false;
  if (*scanner.current != expected) return false;
  scanner.current++;
  return true;
}
```

> If the current character is the desired one, we advance and return `true`. Otherwise, we return `false` to indicate it wasn't matched.

如果当前字符是所需的字符，则指针前进并返回`true`。否则，我们返回`false`表示没有匹配。

> Now our scanner supports all of the punctuation-like tokens. Before we get to the longer ones, let's take a little side trip to handle characters that aren't part of a token at all.

现在我们的扫描器支持所有类似标点符号的标识。在我们处理更长的字符之前，我们先来处理一下那些根本不属于标识的字符。

> ### 16.3.1 Whitespace

16.3.1 空白字符

> Our scanner needs to handle spaces, tabs, and newlines, but those characters don't become part of any token's lexeme. We could check for those inside the main character switch in `scanToken()` but it gets a little tricky to ensure that the function still correctly finds the next token *after* the whitespace when you call it. We'd have to wrap the whole body of the function in a loop or something.

我们的扫描器需要处理空格、制表符和换行符，但是这些字符不会成为任何标识词素的一部分。我们可以在`scanToken()`中的主要的字符switch语句中检查这些字符，但要想确保当你调用该函数时，它仍然能正确地找到空白字符后的下一个标识，这就有点棘手了。我们必须将整个函数封装在一个循环或其它东西中。

> Instead, before starting the token, we shunt off to a separate function.

相应地，在开始扫描标识之前，我们切换到一个单独的函数。

scanner.c，在scanToken()方法中添加代码：

```
Token scanToken() {
  // 新增部分开始
  skipWhitespace();
  // 新增部分结束
  scanner.start = scanner.current;
```

> This advances the scanner past any leading whitespace. After this call returns, we know the very next character is a meaningful one (or we're at the end of the source code).

这将使扫描器跳过所有的前置空白字符。在这个调用返回后，我们知道下一个字符是一个有意义的字符（或者我们到达了源代码的末尾）。

scanner.c，在errorToken()方法后添加：

```
static void skipWhitespace() {
  for (;;) {
    char c = peek();
    switch (c) {
      case ' ':
      case '\r':
      case '\t':
        advance();
```

```
          break;
        default:
          return;
      }
    }
  }
```

> It's sort of a separate mini-scanner. It loops, consuming every whitespace character it encounters. We need to be careful that it does *not* consume any *non*-whitespace characters. To support that, we use this:

这有点像一个独立的微型扫描器。它循环，消费遇到的每一个空白字符。我们需要注意的是，它*不会*消耗任何非空白字符。为了支持这一点，我们使用下面的函数：

scanner.c，在 advance()方法后添加代码：

```
static char peek() {
  return *scanner.current;
}
```

> This simply returns the current character, but doesn't consume it. The previous code handles all the whitespace characters except for newlines.

这只是简单地返回当前字符，但并不消费它。前面的代码已经处理了除换行符外的所有空白字符。

scanner.c，在skipWhitespace()方法内添加代码：

```
          break;
        // 新增部分开始
        case '\n':
          scanner.line++;
          advance();
          break;
        // 新增部分结束
        default:
          return;
```

> When we consume one of those, we also bump the current line number.

当我们消费换行符时，也会增加当前行数。

> ### 16.3.2 Comments

16.3.2 注释

> Comments aren't technically "whitespace", if you want to get all precise with your terminology, but as far as Lox is concerned, they may as well be, so we skip those too.

如果你想用精确的术语，那注释在技术上来说不是“空白字符”，但就Lox目前而言，它们也可以是，所以我们也跳过它们。

scanner.c，在skipWhitespace()函数内添加代码：

```
      break;
    // 新增部分开始
    case '/':
      if (peekNext() == '/') {
        // A comment goes until the end of the line.
        while (peek() != '\n' && !isAtEnd()) advance();
      } else {
        return;
      }
      break;
    // 新增部分结束
    default:
      return;
```

> Comments start with `//` in Lox, so as with `!=` and friends, we need a second character of lookahead. However, with `!=`, we still wanted to consume the `!` even if the `=` wasn't found. Comments are different. If we don't find a second `/`, then `skipWhitespace()` needs to not consume the *first* slash either.

Lox中的注释以`//`开头，因此与`!=`类似，我们需要前瞻第二个字符。然而，在处理`!=`时，即使没有找到`=`，也仍然希望消费`!`。注释是不同的。如果我们没有找到第二个`/`，那么`skipWhitespace()`也不需要消费第一个斜杠。

> To handle that, we add:

为此，我们添加以下函数：

scanner.c，在peek()方法后添加代码：

```
static char peekNext() {
  if (isAtEnd()) return '\0';
  return scanner.current[1];
}
```

> This is like `peek()` but for one character past the current one. If the current character and the next one are both `/`, we consume them and then any other characters until the next newline or the end of the source code.

这就像`peek()`一样，但是是针对当前字符之后的一个字符。如果当前字符和下一个字符都是`/`，则消费它们，然后再消费其它字符，直到遇见下一个换行符或源代码结束。

> We use `peek()` to check for the newline but not consume it. That way, the newline will be the current character on the next turn of the outer loop in `skipWhitespace()` and we'll recognize it and increment `scanner.line`.

我们使用`peek()`来检查换行符，但是不消费它。这样一来，换行符将成为`skipWhitespace()`外部下一轮循环中的当前字符，我们就能识别它并增加`scanner.line`。

> ### 16.3.3 Literal tokens

16.3.3 字面量标识

> Number and string tokens are special because they have a runtime value associated with them. We'll start with strings because they are easy to recognize—they always begin with a double quote.

数字和字符串标识比较特殊，因为它们有一个与之关联的运行时值。我们会从字符串开始，因为它们很容易识别——总是以双引号开始。

scanner.c，在 scanToken()方法中添加代码：

```
          match('=') ? TOKEN_GREATER_EQUAL : TOKEN_GREATER);
    // 新增部分开始
    case '"': return string();
    // 新增部分结束
  }
```

> That calls a new function.

它会调用一个新函数：

scanner.c，在 skipWhitespace()方法后添加代码：

```
static Token string() {
  while (peek() != '"' && !isAtEnd()) {
    if (peek() == '\n') scanner.line++;
    advance();
  }

  if (isAtEnd()) return errorToken("Unterminated string.");

  // The closing quote.
  advance();
  return makeToken(TOKEN_STRING);
}
```

> Similar to jlox, we consume characters until we reach the closing quote. We also track newlines inside the string literal. (Lox supports multi-line strings.) And, as ever, we gracefully handle running out of source code before we find the end quote.

与jlox类似，我们消费字符，直到遇见右引号。我们也会追踪字符串字面量中的换行符（Lox支持多行字符串）。并且，与之前一样，我们会优雅地处理在找到结束引号之前源代码耗尽的问题。

> The main change here in clox is something that’s *not* present. Again, it relates to memory management. In jlox, the Token class had a field of type Object to store the runtime value converted from the literal token’s lexeme.

clox中的主要变化是一些不存在的东西。同样，这与内存管理有关。在jlox中，Token类有一个Object类型的字段，用于存储从字面量词素转换而来的运行时值。

> Implementing that in C would require a lot of work. We’d need some sort of union and type tag to tell whether the token contains a string or double value. If it’s a string, we’d need to manage the memory for the string’s character array somehow.

在C语言中实现这一点需要大量的工作。我们需要某种union和type标签来告诉我们标识中是否包含字符串或浮点数。如果是字符串，我们还需要以某种方式管理字符串中字符数组的内存。

> Instead of adding that complexity to the scanner, we defer converting the literal lexeme to a runtime value until later. In clox, tokens only store the lexeme—the character sequence exactly as it appears in the user’s source code. Later in the compiler, we’ll convert that lexeme to a runtime value right when we are ready to store it in the chunk’s constant table.

我们没有给扫描器增加这种复杂性，我们把字面量词素转换为运行值的工作推迟到以后。在clox中，词法标识只存储词素——即用户源代码中出现的字符序列。稍后在编译器中，当我们准备将其存储在字节码块中的常量表中时，我们会将词素转换为运行时值^6。

> Next up, numbers. Instead of adding a switch case for each of the ten digits that can start a number, we handle them here:

接下来是数字。我们没有为可能作为数字开头的10个数位各添加对应的switch分支，而是使用如下方式处理：

scanner.c，在scanToken()方法中添加代码：

```
  char c = advance();
  // 新增部分开始
  if (isDigit(c)) return number();
  // 新增部分结束
  switch (c) {
```

> That uses this obvious utility function:

这里使用了下面这个明显的工具函数：

scanner.c，在initScanner()方法后添加代码：

```
static bool isDigit(char c) {
  return c >= '0' && c <= '9';
}
```

> We finish scanning the number using this:

我们使用下面的函数完成扫描数字的工作：

scanner.c，在skipWhitespace()方法后添加代码：

```
static Token number() {
  while (isDigit(peek())) advance();

  // Look for a fractional part.
  if (peek() == '.' && isDigit(peekNext())) {
    // Consume the ".".
    advance();

    while (isDigit(peek())) advance();
  }

  return makeToken(TOKEN_NUMBER);
}
```

> It's virtually identical to jlox's version except, again, we don't convert the lexeme to a double yet.

它与jlox版本几乎是相同的，只是我们还没有将词素转换为浮点数。

> ## 16.4 Identifiers and Keywords

16.4 标识符和关键字

> The last batch of tokens are identifiers, both user-defined and reserved. This section should be fun—the way we recognize keywords in clox is quite different from how we did it in jlox, and touches on some important data structures.

最后一批词法是标识符，包括用户定义的和保留字。这一部分应该很有趣——我们在clox中识别关键字的方式与我们在jlox中的方式完全不同，而且涉及到一些重要的数据结构。

> First, though, we have to scan the lexeme. Names start with a letter or underscore.

不过，首先我们需要扫描词素。名称以字母或下划线开头。

scanner.c，在scanToken()方法中添加代码：

```
  char c = advance();
  // 新增部分开始
  if (isAlpha(c)) return identifier();
  // 新增部分结束
  if (isDigit(c)) return number();
```

> We recognize those using this:

我们使用这个方法识别这些标识符：

scanner.c，在initScanner()方法后添加代码：

```
static bool isAlpha(char c) {
  return (c >= 'a' && c <= 'z') ||
         (c >= 'A' && c <= 'Z') ||
          c == '_';
}
```

> Once we've found an identifier, we scan the rest of it here:

一旦我们发现一个标识符，我们就通过下面的方法扫描其余部分：

scanner.c，在skipWhitespace()方法后添加代码：

```
static Token identifier() {
  while (isAlpha(peek()) || isDigit(peek())) advance();
  return makeToken(identifierType());
}
```

> After the first letter, we allow digits too, and we keep consuming alphanumerics until we run out of them. Then we produce a token with the proper type. Determining that "proper" type is the unique part of this chapter.

在第一个字母之后，我们也允许使用数字，并且我们会一直消费字母数字，直到消费完为止。然后我们生成一个具有适当类型的词法标识。确定"适当"类型是本章的特点部分。

scanner.c，在skipWhitespace()方法后添加代码：

```
static TokenType identifierType() {
  return TOKEN_IDENTIFIER;
}
```

> Okay, I guess that's not very exciting yet. That's what it looks like if we have no reserved words at all. How should we go about recognizing keywords? In jlox, we stuffed them all in a Java Map and looked them up by name. We don't have any sort of hash table structure in clox, at least not yet.

好吧，我想这还不算很令人兴奋。如果我们没有保留字，那就是这个样子了。我们应该如何去识别关键字呢？在jlox中，我们将其都塞入一个Java Map中，然后按名称查找它们。在clox中，我们没有任何类型的哈希表结构，至少现在还没有。

> A hash table would be overkill anyway. To look up a string in a hash table, we need to walk the string to calculate its hash code, find the corresponding bucket in the hash table, and then do a character-by-character equality comparison on any string it happens to find there.

无论如何，哈希表都是冗余的。要在哈希表中查找一个字符串，我们需要遍历该字符串以计算其哈希码，在哈希表中找到对应的桶，然后对其中的所有字符串逐个字符进行相等比较^7。

> Let's say we've scanned the identifier "gorgonzola". How much work *should* we need to do to tell if that's a reserved word? Well, no Lox keyword starts with "g", so looking at the first character is enough to definitively answer no. That's a lot simpler than a hash table lookup.

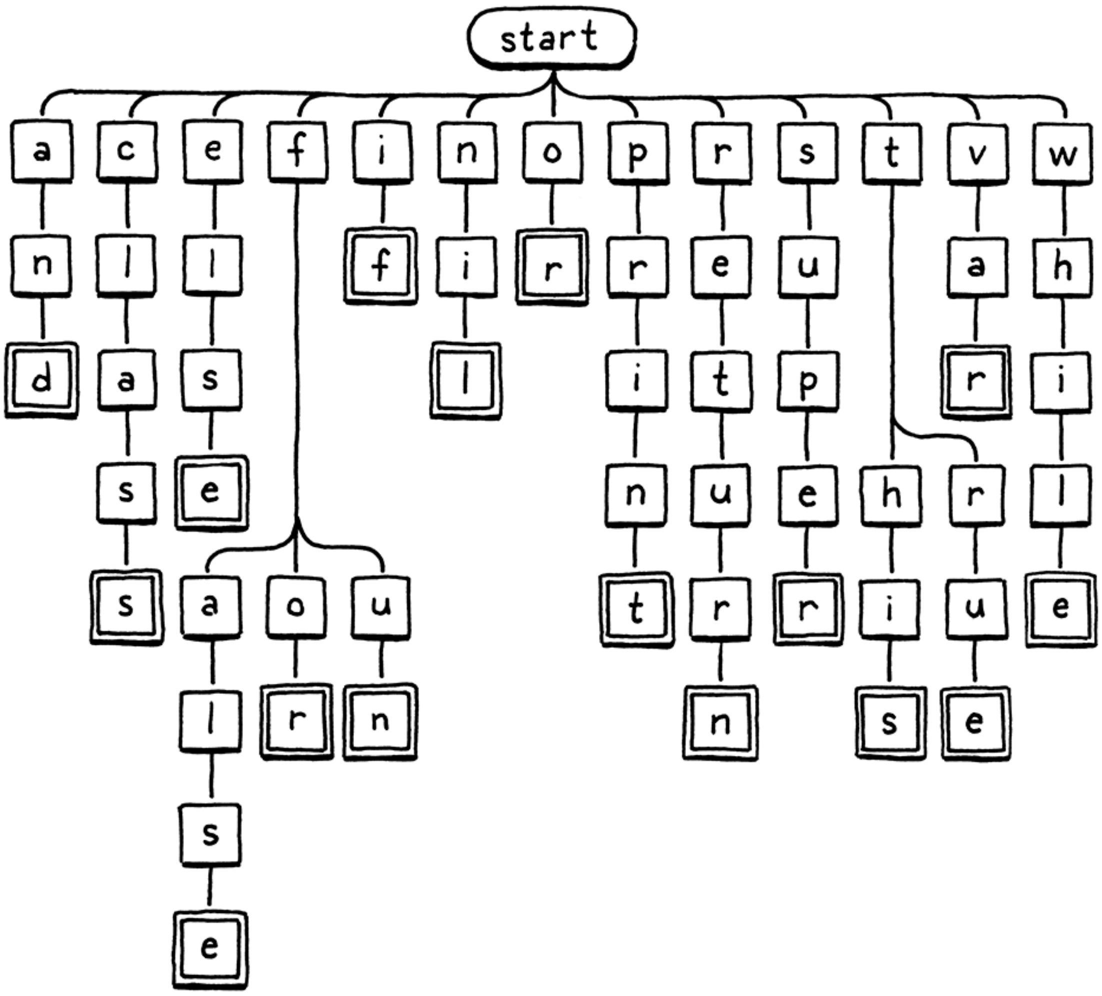
假定我们已经扫描到了标识符"gorgonzola"。我们需要做多少工作来判断这是否是一个保留字？好吧，没有Lox关键字是以"g"开头的，所以看第一个字符就足以明确地回答不是。这比哈希表查询要简单的多。

> What about "cardigan"? We do have a keyword in Lox that starts with "c": "class". But the second character in "cardigan", "a", rules that out. What about "forest"? Since "for" is a keyword, we have to go farther in the string before we can establish that we don't have a reserved word. But, in most cases, only a character or two is enough to tell we've got a user-defined name on our hands. We should be able to recognize that and fail fast.

那"cardigan"呢？我们在Lox中确实有一个以"c"开头的关键字："class"。但是"cardigan"中的第二个字符"a"就排除了这种情况。那"forest"呢？因为"for"是一个关键字，我们必须在字符串中继续遍历，才能确定这不是一个保留字。但是，在大多数情况下，只有一两个字符就足以告诉我们现在处理的是一个用户定义的名称。我们应该能够意识到这一点，并快速失败。

> Here's a visual representation of that branching character-inspection logic:

下面是这个分支字符检查逻辑的一个可视化表示^8：



> We start at the root node. If there is a child node whose letter matches the first character in the lexeme, we move to that node. Then repeat for the next letter in the lexeme and so on. If at any point the next

> letter in the lexeme doesn't match a child node, then the identifier must not be a keyword and we stop. If we reach a double-lined box, and we're at the last character of the lexeme, then we found a keyword.

我们从根节点开始。如果有一个子节点的字母与词素中的第一个字符相匹配，我们就移动到该节点上。然后对词素中的下一个字母重复此操作，以此类推。如果在任意节点上，词素的下一个字符没有匹配到子节点，那么该标识符一定不是一个关键字，我们就停止。如果我们到达了一个双线框，并且我们在词素的最后一个字符处，那么我们就找到了一个关键字。

> ### 16.4.1 Tries and state machines

16.4.1 字典树和状态机

> This tree diagram is an example of a thing called a **trie**. A trie stores a set of strings. Most other data structures for storing strings contain the raw character arrays and then wrap them inside some larger construct that helps you search faster. A trie is different. Nowhere in the trie will you find a whole string.

这个树状图是**trie**^9（字典树）的一个例子。字典树会存储一组字符串。大多数其它用于存储字符串的数据结构都包含原始字符数组，然后将它们封装在一些更大的结果中，以帮助你更快地搜索。字典树则不同，在其中你找不到一个完整的字符串。

> Instead, each string the trie "contains" is represented as a *path* through the tree of character nodes, as in our traversal above. Nodes that match the last character in a string have a special marker—the double lined boxes in the illustration. That way, if your trie contains, say, "banquet" and "ban", you are able to tell that it does *not* contain "banque"—the "e" node won't have that marker, while the "n" and "t" nodes will.

相应地，字典树中"包含"的每个字符串被表示为通过字符树中节点的路径，就像上面的遍历一样。用于匹配字符串中最后一个字符的节点中有一个特殊的标记——插图中的双线框。这样一来，假定你的字典树中包含"banquet"和"ban"，你就能知道它不包括"banque"——"e"节点没有这个标记，而"n"和"t"节点中有。

> Tries are a special case of an even more fundamental data structure: a **deterministic finite automaton** (**DFA**). You might also know these by other names: **finite state machine**, or just **state machine**. State machines are rad. They end up useful in everything from game programming to implementing networking protocols.

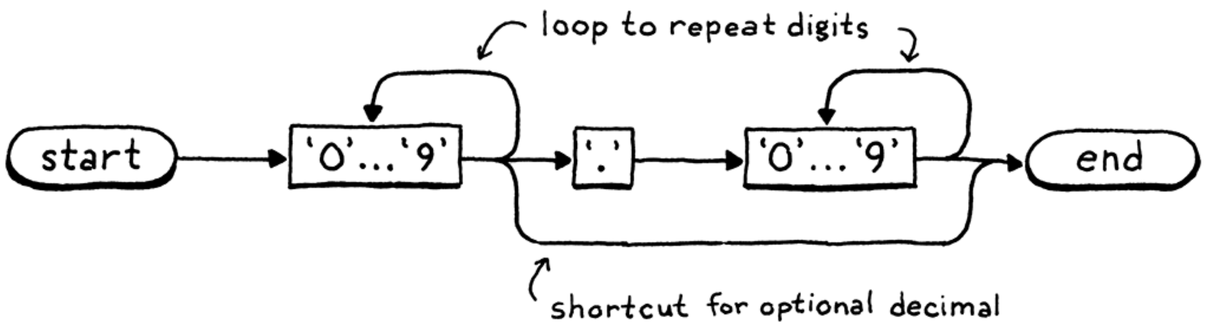
字典树是一种更基本的数据结构的特殊情况：确定性有限状态机（**deterministic finite automaton**， **DFA**）。你可能还知道它的其它名字：**有限状态机**，或就叫**状态机**。状态机是非常重要的，从游戏编程到实现网络协议的一切方面都很有用。

> In a DFA, you have a set of *states* with *transitions* between them, forming a graph. At any point in time, the machine is "in" exactly one state. It gets to other states by following transitions. When you use a DFA for lexical analysis, each transition is a character that gets matched from the string. Each state represents a set of allowed characters.

在DFA中，你有一组*状态*，它们之间有*转换*，形成一个图。在任何时间点，机器都"处于"其中一个状态。它通过转换过渡到其它状态。当你使用DFA进行词法分析时，每个转换都是从字符串中匹配到的一个字符。每个状态代表一组允许的字符。

> Our keyword tree is exactly a DFA that recognizes Lox keywords. But DFAs are more powerful than simple trees because they can be arbitrary *graphs*. Transitions can form cycles between states. That lets you recognize arbitrarily long strings. For example, here's a DFA that recognizes number literals:

我们的关键字树正是一个能够识别Lox关键字的DFA。但是DFA比简单的树更强大，因为它们可以是任意的图。转换可以在状态之间形成循环。这让你可以识别任意长的字符串。举例来说，下面是一个可以识别数字字面量的DFA^10：



> I've collapsed the nodes for the ten digits together to keep it more readable, but the basic process works the same—you work through the path, entering nodes whenever you consume a corresponding character in the lexeme. If we were so inclined, we could construct one big giant DFA that does *all* of the lexical analysis for Lox, a single state machine that recognizes and spits out all of the tokens we need.

我把十个数位的节点折叠在一起，以使其更易于阅读，但是基本的过程是相同的——遍历路径，每当你消费词素中的一个字符，就进入对应节点。如果我们愿意的话，可以构建一个巨大的DFA来完成Lox的所有词法分析，用一个状态机来识别并输出我们需要的所有词法标识。

> However, crafting that mega-DFA by hand would be challenging. That's why Lex was created. You give it a simple textual description of your lexical grammar—a bunch of regular expressions—and it automatically generates a DFA for you and produces a pile of C code that implements it.

然而，手工完成这种巨型DFA是一个巨大的挑战。这就是Lex诞生的原因。你给它一个关于语法的简单文本描述——一堆正则表达式——它就会自动为你生成一个DFA，并生成一堆实现它的C代码^11。

This is also how most regular expression engines in programming languages and text editors work under the hood. They take your regex string and convert it to a DFA, which they then use to match strings.

If you want to learn the algorithm to convert a regular expression into a DFA, the dragon book has you covered.

> We won't go down that road. We already have a perfectly serviceable hand-rolled scanner. We just need a tiny trie for recognizing keywords. How should we map that to code?

我们就不走这条路了。我们已经有了一个完全可用的简单扫描器。我们只需要一个很小的字典树来识别关键字。我们应该如何将其映射到代码中？

> The absolute simplest solution is to use a switch statement for each node with cases for each branch. We'll start with the root node and handle the easy keywords.

最简单的解决方案是对每个节点使用一个switch语句，每个分支是一个case。我们从根节点开始，处理简单的关键字^12。

scanner.c，在identifierType()方法中添加代码：

```
static TokenType identifierType() {
  // 新增部分开始
  switch (scanner.start[0]) {
    case 'a': return checkKeyword(1, 2, "nd", TOKEN_AND);
    case 'c': return checkKeyword(1, 4, "lass", TOKEN_CLASS);
    case 'e': return checkKeyword(1, 3, "lse", TOKEN_ELSE);
    case 'i': return checkKeyword(1, 1, "f", TOKEN_IF);
    case 'n': return checkKeyword(1, 2, "il", TOKEN_NIL);
    case 'o': return checkKeyword(1, 1, "r", TOKEN_OR);
    case 'p': return checkKeyword(1, 4, "rint", TOKEN_PRINT);
    case 'r': return checkKeyword(1, 5, "eturn", TOKEN_RETURN);
    case 's': return checkKeyword(1, 4, "uper", TOKEN_SUPER);
    case 'v': return checkKeyword(1, 2, "ar", TOKEN_VAR);
    case 'w': return checkKeyword(1, 4, "hile", TOKEN_WHILE);
  }
  // 新增部分结束
  return TOKEN_IDENTIFIER;
```

> These are the initial letters that correspond to a single keyword. If we see an “s”, the only keyword the identifier could possibly be is `super`. It might not be, though, so we still need to check the rest of the letters too. In the tree diagram, this is basically that straight path hanging off the “s”.

这些是对应于单个关键字的首字母。如果我们看到一个“s”，那么这个标识符唯一可能的关键字就是`super`。但也可能不是，所以我们仍然需要检查其余的字母。在树状图中，这基本上就是挂在“s”上的一条直线路径。

> We won’t roll a switch for each of those nodes. Instead, we have a utility function that tests the rest of a potential keyword’s lexeme.

我们不会为每个节点都增加一个switch语句。相反，我们有一个工具函数来测试潜在关键字词素的剩余部分。

scanner.c，在skipWhitespace()方法后添加代码：

```
static TokenType checkKeyword(int start, int length,
    const char* rest, TokenType type) {
  if (scanner.current - scanner.start == start + length &&
      memcmp(scanner.start + start, rest, length) == 0) {
    return type;
  }

  return TOKEN_IDENTIFIER;
}
```

> We use this for all of the unbranching paths in the tree. Once we’ve found a prefix that could only be one possible reserved word, we need to verify two things. The lexeme must be exactly as long as the keyword. If the first letter is “s”, the lexeme could still be “sup” or “superb”. And the remaining characters must match exactly—“supar” isn’t good enough.

我们将此用于树中的所有无分支路径。一旦我们发现一个前缀，其只有可能是一种保留字，我们需要验证两件事。词素必须与关键字一样长。如果第一个字母是“s”，词素仍然可以是“sup”或“superb”。剩下的字符必须完全匹配——“supar”就不够好。

> If we do have the right number of characters, and they're the ones we want, then it's a keyword, and we return the associated token type. Otherwise, it must be a normal identifier.

如果我们字符数量确实正确，并且它们是我们想要的字符，那这就是一个关键字，我们返回相关的标识类型。否则，它必然是一个普通的标识符。

> We have a couple of keywords where the tree branches again after the first letter. If the lexeme starts with “f”, it could be `false`, `for`, or `fun`. So we add another switch for the branches coming off the “f” node.

我们有几个关键字是在第一个字母之后又有树的分支。如果词素以“f”开头，它可能是`false`、`for`或`fun`。因此我们在“f”节点下的分支中添加一个switch语句。

scanner.c，在identifierType()方法中添加代码：

```
    case 'e': return checkKeyword(1, 3, "lse", TOKEN_ELSE);
    // 新增部分开始
    case 'f':
      if (scanner.current - scanner.start > 1) {
        switch (scanner.start[1]) {
          case 'a': return checkKeyword(2, 3, "lse", TOKEN_FALSE);
          case 'o': return checkKeyword(2, 1, "r", TOKEN_FOR);
          case 'u': return checkKeyword(2, 1, "n", TOKEN_FUN);
        }
      }
      break;
    // 新增部分结束
    case 'i': return checkKeyword(1, 1, "f", TOKEN_IF);
```

> Before we switch, we need to check that there even *is* a second letter. “f” by itself is a valid identifier too, after all. The other letter that branches is “t”.

在我们进入switch语句之前，需要先检查是否有第二个字母。毕竟，“f”本身也是一个有效的标识符。另外一个需要分支的字母是“t”。

scanner.c，在identifierType()方法中添加代码：

```
    case 's': return checkKeyword(1, 4, "uper", TOKEN_SUPER);
    // 新增部分开始
    case 't':
      if (scanner.current - scanner.start > 1) {
        switch (scanner.start[1]) {
          case 'h': return checkKeyword(2, 2, "is", TOKEN_THIS);
          case 'r': return checkKeyword(2, 2, "ue", TOKEN_TRUE);
        }
      }
      break;
    // 新增部分结束
    case 'v': return checkKeyword(1, 2, "ar", TOKEN_VAR);
```

> That's it. A couple of nested `switch` statements. Not only is this code short, but it's very, very fast. It does the minimum amount of work required to detect a keyword, and bails out as soon as it can tell the identifier will not be a reserved one.

就是这样。几个嵌套的`switch`语句。这段代码不仅短，而且非常非常快。它只做了检测一个关键字所需的最少的工作，而且一旦知道这个标识符不是一个保留字，就会直接结束^13。

> And with that, our scanner is complete.

这样一来，我们的扫描器就完整了。

^9: "Trie"是CS中最令人困惑的名字之一。Edward Fredkin从"检索（retrieval）"中把这个词提取出来，这意味着它的读音应该像"tree"。但是，已经有一个非常重要的数据结构发音为"tree"，而trie只是一个特例。所以如果你谈论这些东西时，没人能分辨出你在说哪一个。因此，现在人们经常把它读作"try"，以免头痛。 ^10: 这种风格的图被称为语法图或**铁路图**。后者的名字是因为它看起来像火车的调度场。
在Backus-Naur范式出现之前，这是记录语言语法的主要方式之一。如今，我们大多使用文本，但一种*文本语言*的官方规范依赖于*图像*，这一点很令人高兴。 ^11: 这也是大多数编程语言和文本编辑器中的正则表达式引擎的工作原理。它们获取你的正则表达式字符串并将其转换为DFA，然后使用DFA来匹配字符串。
如果你想学习将正则表达式转换为DFA的算法，龙书中已经为你提供了答案。 ^12: 简单并不意味着愚蠢。V8也采用了同样的方法，而它是目前世界上最复杂、最快的语言实现之一。 ^13: 我们有时会陷入这样的误区：任务性能来自于复杂的数据结构、多级缓存和其它花哨的优化。但是，很多时候所需要的就是做更少的工作，而我经常发现，编写最简单的代码就足以完成这些工作。

> ## CHALLENGES

习题

1. > Many newer languages support **string interpolation**. Inside a string literal, you have some sort of special delimiters—most commonly `${` at the beginning and `}` at the end. Between those delimiters, any expression can appear. When the string literal is executed, the inner expression is evaluated, converted to a string, and then merged with the surrounding string literal.

   许多较新的语言都支持字符串插值。在字符串字面量中，有一些特殊的分隔符——最常见的是以`${`开头以`}`结尾。在这些分隔符之间，可以出现任何表达式。当字符串字面量被执行时，内部表达式也会求值，转换为字符串，然后与周围的字符串字面量合并。

   > For example, if Lox supported string interpolation, then this . . .

   举例来说，如果Lox支持字符串插值，那么下面的代码......

   ```
   var drink = "Tea";
   var steep = 4;
   var cool = 2;
   print "${drink} will be ready in ${steep + cool} minutes.";
   ```

> . . . would print:

将会输出：

```
Tea will be ready in 6 minutes.
```

> What token types would you define to implement a scanner for string interpolation? What sequence of tokens would you emit for the above string literal?

你会定义什么标识类型来实现支持字符串插值的扫描器？对于上面的字符串，你会生成什么样的标识序列？

> What tokens would you emit for:

下面的字符串会产生哪些标识：

```
"Nested ${"interpolation?! Are you ${"mad?!"}"}"
```

> Consider looking at other language implementations that support interpolation to see how they handle it.

可以考虑看看其它支持插值的语言实现，看它们是如何处理的。

2. > Several languages use angle brackets for generics and also have a `>>` right shift operator. This led to a classic problem in early versions of C++:

   有些语言使用尖括号来表示泛型，也有右移操作符`>>`。这就导致了C++早期版本中的一个经典问题：

   ```
   vector<vector<string>> nestedVectors;
   ```

   > This would produce a compile error because the `>>` was lexed to a single right shift token, not two `>` tokens. Users were forced to avoid this by putting a space between the closing angle brackets.

   这将产生一个编译错误，因为`>>`被词法识别为一个右移符号，而不是两个`>`标识。用户不得不在右侧的两个尖括号之间增加一个空格来避免这种情况。

   > Later versions of C++ are smarter and can handle the above code. Java and C# never had the problem. How do those languages specify and implement this?

   后续的C++版本更加智能，可以处理上述代码。Java和C#从未出现过这个问题。这些语言是如何规定和实现这一点的呢？

3. > Many languages, especially later in their evolution, define “contextual keywords”. These are identifiers that act like reserved words in some contexts but can be normal user-defined

> identifiers in others.

许多语言，尤其是在其发展的后期，都定义了“上下文式关键字”。这些标识符在某些情况下类似于保留字，但在其它上下文中可以是普通的用户定义的标识符。

> For example, `await` is a keyword inside an `async` method in C#, but in other methods, you can use `await` as your own identifier.

例如，在C#中，`await`在`async`方法中是一个关键字，但在其它方法中，你可以使用`await`作为自己的标识符。

> Name a few contextual keywords from other languages, and the context where they are meaningful. What are the pros and cons of having contextual keywords? How would you implement them in your language’s front end if you needed to?

说出几个来自其它语言中的上下文关键字，以及它们在哪些情况下是有意义的。拥有上下文关键字的优点和缺点是什么？如果需要的话，你要如何在语言的前端中实现它们？

17.编译表达式 Compiling Expressions

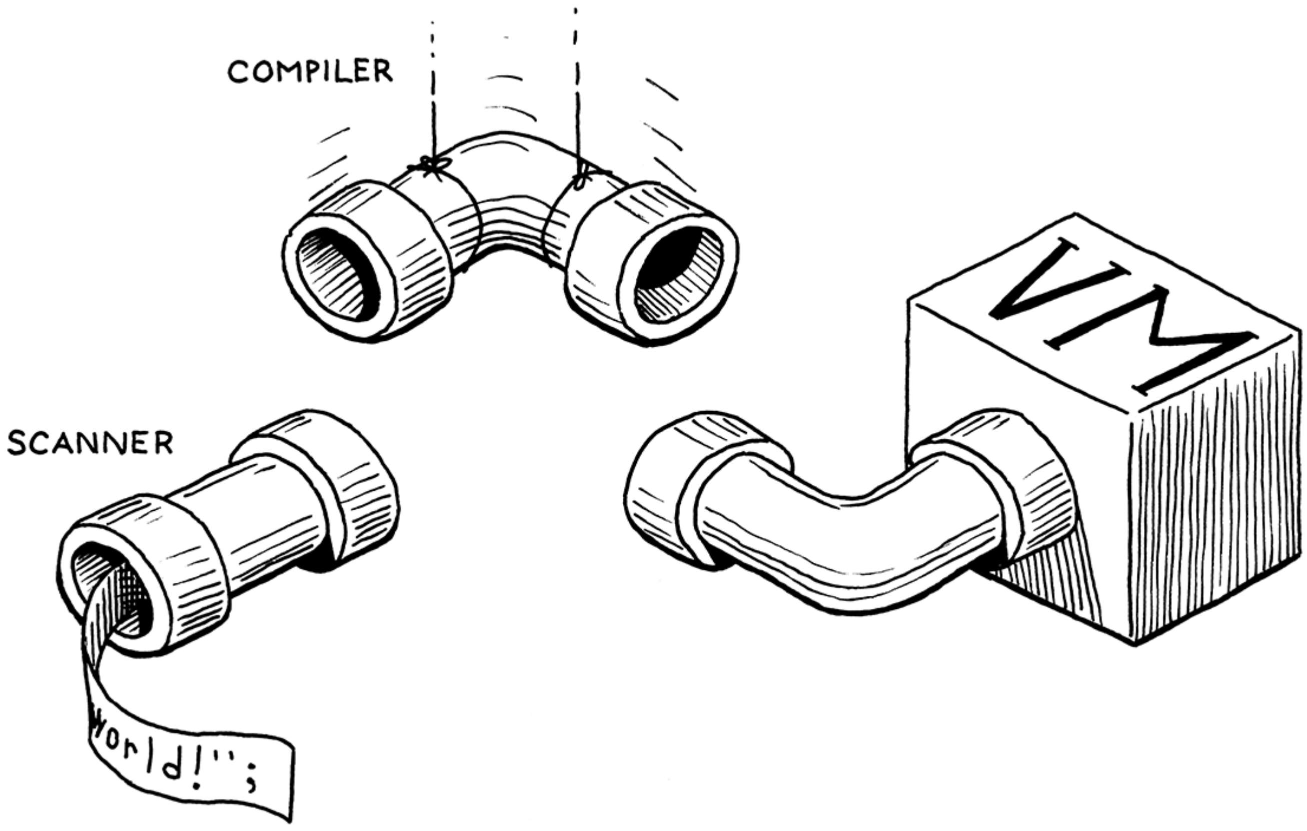
> In the middle of the journey of our life I found myself within a dark woods where the straight way was lost.
>
> —— Dante Alighieri, *Inferno*

方吾生之半路，恍余处乎幽林，失正轨而迷误。（但丁，《地狱》）

【译者注：这里引用的是大名鼎鼎的《神曲》，所以我也直接引用了钱稻孙先生的译文】

> This chapter is exciting for not one, not two, but *three* reasons. First, it provides the final segment of our VM’s execution pipeline. Once in place, we can plumb the user’s source code from scanning all the way through to executing it.

这一章令人激动，原因不止一个，也不止两个，而是三个。首先，它补齐了虚拟机执行管道的最后一段。一旦到位，我们就可以处理用户的源代码，从扫描一直到执行。



> Second, we get to write an actual, honest-to-God *compiler*. It parses source code and outputs a low-level series of binary instructions. Sure, it's bytecode and not some chip's native instruction set, but it's way closer to the metal than jlox was. We're about to be real language hackers.

第二，我们要编写一个真正的*编译器*。它会解析源代码并输出低级的二进制指令序列。当然，它是字节码，而不是某个芯片的原生指令集，但它比jlox更接近于硬件。我们即将成为真正的语言黑客了。

> Third and finally, I get to show you one of my absolute favorite algorithms: Vaughan Pratt's "top-down operator precedence parsing". It's the most elegant way I know to parse expressions. It gracefully handles prefix operators, postfix, infix, *mixfix*, any kind of *-fix* you got. It deals with precedence and associativity without breaking a sweat. I love it.

第三，也是最后一个，我可以向你们展示我们最喜欢的算法之一：Vaughan Pratt的"自顶向下算符优先解析"。这是我所知道的解析表达的最优雅的方法。它可以优雅地处理前缀、后缀、中缀、多元运算符，以及任何类型的运算符。它能处理优先级和结合性，而且毫不费力。我喜欢它。

> As usual, before we get to the fun stuff, we've got some preliminaries to work through. You have to eat your vegetables before you get dessert. First, let's ditch that temporary scaffolding we wrote for testing the scanner and replace it with something more useful.

与往常一样，在我们开始真正有趣的工作之前，还有一些准备工作需要做。在得到甜点之前，你得先吃点蔬菜。首先，让我们抛弃我们为测试扫描器而编写的临时脚手架，用更有效的东西来替换它。

vm.c，在interpret() 方法中替换2 行：

```
InterpretResult interpret(const char* source) {
  // 替换部分开始
  Chunk chunk;
  initChunk(&chunk);
  if (!compile(source, &chunk)) {
    freeChunk(&chunk);
    return INTERPRET_COMPILE_ERROR;
  }

  vm.chunk = &chunk;
  vm.ip = vm.chunk->code;

  InterpretResult result = run();

  freeChunk(&chunk);
  return result;
  // 替换部分结束
}
```

> We create a new empty chunk and pass it over to the compiler. The compiler will take the user's program and fill up the chunk with bytecode. At least, that's what it will do if the program doesn't have any compile errors. If it does encounter an error, `compile()` returns `false` and we discard the unusable chunk.

我们创建一个新的空字节码块，并将其传递给编译器。编译器会获取用户的程序，并将字节码填充到该块中。至少在程序没有任何编译错误的情况下，它就会这么做。如果遇到错误，`compile()`方法会返回`false`，我们就会丢弃不可用的字节码块。

> Otherwise, we send the completed chunk over to the VM to be executed. When the VM finishes, we free the chunk and we're done. As you can see, the signature to `compile()` is different now.

否则，我们将完整的字节码块发送到虚拟机中去执行。当虚拟机完成后，我们会释放该字节码块，这样就完成了。如你所见，现在`compile()`的签名已经不同了。

compiler.h，替换一行代码：

```
#define clox_compiler_h
// 替换部分开始
#include "vm.h"

bool compile(const char* source, Chunk* chunk);
// 替换部分结束
#endif
```

> We pass in the chunk where the compiler will write the code, and then `compile()` returns whether or not compilation succeeded. We make the same change to the signature in the implementation.

我们将字节码块传入，而编译器会向其中写入代码，如何`compile()`返回编译是否成功。我们在实现方法中对签名进行相同的修改。

compiler.c，在compile()方法中替换1行：

```
#include "scanner.h"
// 替换部分开始
bool compile(const char* source, Chunk* chunk) {
// 替换部分结束
  initScanner(source);
```

> That call to `initScanner()` is the only line that survives this chapter. Rip out the temporary code we wrote to test the scanner and replace it with these three lines:

对`initScanner()`的调用是本章中唯一保留下来的代码行。删除我们为测试扫描器而编写的临时代码，将其替换为以下三行：

compiler.c，在compile()方法中替换13行：

```
  initScanner(source);
  // 替换部分开始
  advance();
  expression();
  consume(TOKEN_EOF, "Expect end of expression.");
  // 替换部分结束
}
```

> The call to `advance()` "primes the pump" on the scanner. We'll see what it does soon. Then we parse a single expression. We aren't going to do statements yet, so that's the only subset of the grammar we support. We'll revisit this when we add statements in a few chapters. After we compile the expression, we should be at the end of the source code, so we check for the sentinel EOF token.

对`advance()`的调用会在扫描器上"启动泵"。我们很快会看到它的作用。然后我们解析一个表达式。我们还不打算处理语句，所以表达式是我们支持的唯一的语法子集。等到我们在后面的章节中添加语句时，会重新审视这个问题。在编译表达式之后，我们应该处于源代码的末尾，所以我们要检查EOF标识。

> We're going to spend the rest of the chapter making this function work, especially that little `expression()` call. Normally, we'd dive right into that function definition and work our way through the implementation from top to bottom.

我们将用本章的剩余时间让这个函数工作起来。尤其是那个小小的`expression()`调用。通常情况下，我们会直接进入函数定义，并从上到下地进行实现。

> This chapter is different. Pratt's parsing technique is remarkably simple once you have it all loaded in your head, but it's a little tricky to break into bite-sized pieces. It's recursive, of course, which is part of the problem. But it also relies on a big table of data. As we build up the algorithm, that table grows additional columns.

这一章则不同。Pratt的解析技术，你一旦理解了就非常简单，但是要把它分解成小块就比较麻烦了^1。当然，它是递归的，这也是问题的一部分。但它也依赖于一个很大的数据表。等我们构建算法时，这个表格会增加更多的列。

> I don’t want to revisit 40-something lines of code each time we extend the table. So we’re going to work our way into the core of the parser from the outside and cover all of the surrounding bits before we get to the juicy center. This will require a little more patience and mental scratch space than most chapters, but it’s the best I could do.

我不想在每次扩展表时都要重新查看40多行代码。因此，我们要从外部进入解析器的核心，并在进入有趣的中心之前覆盖其外围的所有部分。与大多数章节相比，这将需要更多的耐心和思考空间，但这是我能做到的最好的了。

17.1 Single-Pass Compilation

17.1 单遍编译

> A compiler has roughly two jobs. It parses the user’s source code to understand what it means. Then it takes that knowledge and outputs low-level instructions that produce the same semantics. Many languages split those two roles into two separate passes in the implementation. A parser produces an AST—just like jlox does—and then a code generator traverses the AST and outputs target code.

一个编译器大约有两项工作^2。它会解析用户的源代码以理解其含义。然后，它利用这些知识并输出产生相同语义的低级指令。许多语言在实现中将这两个角色分成两遍独立的执行部分。一个解析器生成AST——就像jlox那样——还有一个代码生成器遍历AST并输出目标代码。

> In clox, we’re taking an old-school approach and merging these two passes into one. Back in the day, language hackers did this because computers literally didn’t have enough memory to store an entire source file’s AST. We’re doing it because it keeps our compiler simpler, which is a real asset when programming in C.

在clox中，我们采用了一种老派的方法，将这两遍处理合而为一。在过去，语言黑客们这样做是因为计算机没有足够的内存来存储整个源文件的AST。我们这样做是因为它使我们的编译器更简单，这是用C语言编程时的真正优势。

> Single-pass compilers like we’re going to build don’t work well for all languages. Since the compiler has only a peephole view into the user’s program while generating code, the language must be designed such that you don’t need much surrounding context to understand a piece of syntax. Fortunately, tiny, dynamically typed Lox is well-suited to that.

像我们要构建的单遍编译器并不是对所有语言都有效。因为编译器在生产代码时只能“管窥”用户的程序，所以语言必须设计成不需要太多外围的上下文环境就能理解一段语法。幸运的是，微小的、动态类型的Lox非常适合这种情况。

> What this means in practical terms is that our “compiler” C module has functionality you’ll recognize from jlox for parsing—consuming tokens, matching expected token types, etc. And it also has functions for code gen—emitting bytecode and adding constants to the destination chunk. (And it means I’ll use “parsing” and “compiling” interchangeably throughout this and later chapters.)

在实践中，这意味着我们的“编译器”C模块具有你在jlox中认识到的解析功能——消费标识，匹配期望的标识类型，等等。而且它还具有代码生成的功能——生成字节码和向目标块中添加常量。（这也意味着我会在本章和后面的章节中交替使用“解析”和“编译”。）

> We'll build the parsing and code generation halves first. Then we'll stitch them together with the code in the middle that uses Pratt's technique to parse Lox's particular grammar and output the right bytecode.

我们首先分别构建解析和代码生成两个部分。然后，我们会用中间代码将它们缝合在一起，该代码使用Pratt的技术来解析Lox 的语法并输出正确的字节码。

17.2 Parsing Tokens

17.2 解析标识

> First up, the front half of the compiler. This function's name should sound familiar.

首先是编译器的前半部分。这个函数的名字听起来应该很熟悉。 *compiler.c，添加代码：*

```
#include "scanner.h"
// 新增部分开始
static void advance() {
  parser.previous = parser.current;

  for (;;) {
    parser.current = scanToken();
    if (parser.current.type != TOKEN_ERROR) break;

    errorAtCurrent(parser.current.start);
  }
}
// 新增部分结束
```

> Just like in jlox, it steps forward through the token stream. It asks the scanner for the next token and stores it for later use. Before doing that, it takes the old `current` token and stashes that in a `previous` field. That will come in handy later so that we can get at the lexeme after we match a token.

就像在jlox中一样，该函数向前通过标识流。它会向扫描器请求下一个词法标识，并将其存储起来以供后面使用。在此之前，它会获取旧的`current`标识，并将其存储在`previous`字段中。这在以后会派上用场，让我们可以在匹配到标识之后获得词素。

> The code to read the next token is wrapped in a loop. Remember, clox's scanner doesn't report lexical errors. Instead, it creates special *error tokens* and leaves it up to the parser to report them. We do that here.

读取下一个标识的代码被包在一个循环中。记住，clox的扫描器不会报告词法错误。相反地，它创建了一个特殊的*错误标识*，让解析器来报告这些错误。我们这里就是这样做的。

> We keep looping, reading tokens and reporting the errors, until we hit a non-error one or reach the end. That way, the rest of the parser sees only valid tokens. The current and previous token are stored in this struct:

我们不断地循环，读取标识并报告错误，直到遇到一个没有错误的标识或者到达标识流终点。这样一来，解析器的其它部分只能看到有效的标记。当前和之前的标记被存储在下面的结构体中：

compiler.c，新增代码：

```
#include "scanner.h"
// 新增部分开始
typedef struct {
  Token current;
  Token previous;
} Parser;

Parser parser;
// 新增部分结束
static void advance() {
```

> Like we did in other modules, we have a single global variable of this struct type so we don't need to pass the state around from function to function in the compiler.

就像我们在其它模块中所做的那样，我们维护一个这种结构体类型的单一全局变量，所以我们不需要在编译器中将状态从一个函数传递到另一个函数。

> ### 17.2.1 Handling syntax errors

17.2.1 处理语法错误

> If the scanner hands us an error token, we need to actually tell the user. That happens using this:

如果扫描器交给我们一个错误标识，我们必须明确地告诉用户。这就需要使用下面的语句：

compiler.c，在变量parser后添加代码：

```
static void errorAtCurrent(const char* message) {
  errorAt(&parser.current, message);
}
```

> We pull the location out of the current token in order to tell the user where the error occurred and forward it to `errorAt()`. More often, we'll report an error at the location of the token we just consumed, so we give the shorter name to this other function:

我们从当前标识中提取位置信息，以便告诉用户错误发生在哪里，并将其转发给`errorAt()`。更常见的情况是，我们会在刚刚消费的令牌的位置报告一个错误，所以我们给另一个函数取了一个更短的名字：

compiler.c，在变量parser后添加代码：

```
static void error(const char* message) {
  errorAt(&parser.previous, message);
}
```

> The actual work happens here:

实际的工作发生在这里：

compiler.c，在变量parser后添加代码：

```
static void errorAt(Token* token, const char* message) {
  fprintf(stderr, "[line %d] Error", token->line);

  if (token->type == TOKEN_EOF) {
    fprintf(stderr, " at end");
  } else if (token->type == TOKEN_ERROR) {
    // Nothing.
  } else {
    fprintf(stderr, " at '%.*s'", token->length, token->start);
  }

  fprintf(stderr, ": %s\n", message);
  parser.hadError = true;
}
```

> First, we print where the error occurred. We try to show the lexeme if it's human-readable. Then we print the error message itself. After that, we set this `hadError` flag. That records whether any errors occurred during compilation. This field also lives in the parser struct.

首先，我们打印出错误发生的位置。如果词素是人类可读的，我们就尽量显示词素。然后我们打印错误信息。之后，我们设置这个`hadError`标志。该标志记录了编译过程中是否有任何错误发生。这个字段也存在于解析器结构体中。

compiler.c，在结构体Parser中添加代码：

```
  Token previous;
  // 新增部分开始
  bool hadError;
  // 新增部分结束
} Parser;
```

> Earlier I said that `compile()` should return `false` if an error occurred. Now we can make it do that.

前面我说过，如果发生错误，`compile()`应该返回`false`。现在我们可以这样做：

compiler.c，在compile()函数中添加代码：

```
  consume(TOKEN_EOF, "Expect end of expression.");
  // 新增部分开始
  return !parser.hadError;
  // 新增部分结束
}
```

> I've got another flag to introduce for error handling. We want to avoid error cascades. If the user has a mistake in their code and the parser gets confused about where it is in the grammar, we don't want it to spew out a whole pile of meaningless knock-on errors after the first one.

我还要引入另一个用于错误处理的标志。我们想要避免错误的级联效应。如果用户在他们的代码中犯了一个错误，而解析器又不理解它在语法中的含义，我们不希望解析器在第一个错误之后，又抛出一大堆无意义的连带错误。

> We fixed that in jlox using panic mode error recovery. In the Java interpreter, we threw an exception to unwind out of all of the parser code to a point where we could skip tokens and resynchronize. We don't have exceptions in C. Instead, we'll do a little smoke and mirrors. We add a flag to track whether we're currently in panic mode.

我们在jlox中使用紧急模式错误恢复来解决这个问题。在Java解释器中，我们抛出一个异常，跳出解析器代码直到可以跳过标识并重新同步。我们在C语言中没有异常^3。相反，我们会做一些欺骗性行为。我们添加一个标志来跟踪当前是否在紧急模式中。

compiler.c，在结构体Parser中添加代码：

```
  bool hadError;
  // 新增部分开始
  bool panicMode;
  // 新增部分结束
} Parser;
```

> When an error occurs, we set it.

当出现错误时，我们为其赋值。

compiler.c，在errorAt()方法中添加代码：

```
static void errorAt(Token* token, const char* message) {
  // 新增部分开始
  parser.panicMode = true;
  // 新增部分结束
  fprintf(stderr, "[line %d] Error", token->line);
```

> After that, we go ahead and keep compiling as normal as if the error never occurred. The bytecode will never get executed, so it's harmless to keep on trucking. The trick is that while the panic mode flag is set, we simply suppress any other errors that get detected.

之后，我们继续进行编译，就像错误从未发生过一样。字节码永远不会被执行，所以继续运行也是无害的。诀窍在于，虽然设置了紧急模式标志，但我们只是简单地屏蔽了检测到的其它错误。

compiler.c，在errorAt()方法中添加代码：

```
static void errorAt(Token* token, const char* message) {
  // 新增部分开始
  if (parser.panicMode) return;
  // 新增部分结束
  parser.panicMode = true;
```

There's a good chance the parser will go off in the weeds, but the user won't know because the errors all get swallowed. Panic mode ends when the parser reaches a synchronization point. For Lox, we chose statement boundaries, so when we later add those to our compiler, we'll clear the flag there.

解析器很有可能会崩溃，但是用户不会知道，因为错误都会被吞掉。当解析器到达一个同步点时，紧急模式就结束了。对于Lox，我们选择了语句作为边界，所以当我们稍后将语句添加到编译器时，将会清除该标志。

> These new fields need to be initialized.

这些新字段需要被初始化。

compiler.c，在compile()方法中添加代码：

```
  initScanner(source);
  // 新增部分开始
  parser.hadError = false;
  parser.panicMode = false;
  // 新增部分结束
  advance();
```

> And to display the errors, we need a standard header.

为了展示这些错误，我们需要一个标准的头文件。

compiler.c，添加代码：

```
#include <stdio.h>
// 新增部分开始
#include <stdlib.h>
// 新增部分结束
#include "common.h"
```

> There's one last parsing function, another old friend from jlox.

还有最后一个解析函数，是jlox中的另一个老朋友。

compiler.c，在advance()方法后添加代码：

```
static void consume(TokenType type, const char* message) {
  if (parser.current.type == type) {
    advance();
```

```
      return;
    }

    errorAtCurrent(message);
  }
```

> It's similar to `advance()` in that it reads the next token. But it also validates that the token has an expected type. If not, it reports an error. This function is the foundation of most syntax errors in the compiler.

它类似于`advance()`，都是读取下一个标识。但它也会验证标识是否具有预期的类型。如果不是，则报告错误。这个函数是编译器中大多数语法错误的基础。

> OK, that's enough on the front end for now.

好了，关于前端的介绍就到此为止。

17.3 Emitting Bytecode

17.3 发出字节码

> After we parse and understand a piece of the user's program, the next step is to translate that to a series of bytecode instructions. It starts with the easiest possible step: appending a single byte to the chunk.

在我们解析并理解了用户的一段程序之后，下一步是将其转换为一系列字节码指令。它从最简单的步骤开始：向块中追加一个字节。

compiler.c，在consume()方法后添加代码：

```
  static void emitByte(uint8_t byte) {
    writeChunk(currentChunk(), byte, parser.previous.line);
  }
```

> It's hard to believe great things will flow through such a simple function. It writes the given byte, which may be an opcode or an operand to an instruction. It sends in the previous token's line information so that runtime errors are associated with that line.

很难相信伟大的东西会流经这样一个简单的函数。它将给定的字节写入一个指令，该字节可以是操作码或操作数。它会发送前一个标记的行信息，以便将运行时错误与该行关联起来。

> The chunk that we're writing gets passed into `compile()`, but it needs to make its way to `emitByte()`. To do that, we rely on this intermediary function:

我们正在写入的字节码块被传递给`compile()`，但是它也需要进入`emitByte()`中。要做到这一点，我们依靠这个中间函数：

compiler.c，在变量parser后添加代码：

```
Parser parser;
// 新增部分开始
Chunk* compilingChunk;

static Chunk* currentChunk() {
  return compilingChunk;
}
// 新增部分结束
static void errorAt(Token* token, const char* message) {
```

> Right now, the chunk pointer is stored in a module-level variable like we store other global state. Later, when we start compiling user-defined functions, the notion of "current chunk" gets more complicated. To avoid having to go back and change a lot of code, I encapsulate that logic in the `currentChunk()` function.

现在，chunk指针存储在一个模块级变量中，就像我们存储其它全局状态一样。以后，当我们开始编译用户定义的函数时，"当前块"的概念会变得更加复杂。为了避免到时候需要回头修改大量代码，我把这个逻辑封装在`currentChunk()`函数中。

> We initialize this new module variable before we write any bytecode:

在写入任何字节码之前，我们先初始化这个新的模块变量：

compiler.c，在compile()方法中添加代码：

```
bool compile(const char* source, Chunk* chunk) {
  initScanner(source);
  // 新增部分开始
  compilingChunk = chunk;
  // 新增部分结束
  parser.hadError = false;
```

> Then, at the very end, when we're done compiling the chunk, we wrap things up.

然后，在最后，当我们编译完字节码块后，对全部内容做个了结。

compiler.c，在compile()方法中添加代码：

```
  consume(TOKEN_EOF, "Expect end of expression.");
  // 新增部分开始
  endCompiler();
  // 新增部分结束
  return !parser.hadError;
```

> That calls this:

会调用下面的函数：

compiler.c，在emitByte()方法后添加代码：

```
static void endCompiler() {
  emitReturn();
}
```

> In this chapter, our VM deals only with expressions. When you run clox, it will parse, compile, and execute a single expression, then print the result. To print that value, we are temporarily using the `OP_RETURN` instruction. So we have the compiler add one of those to the end of the chunk.

在本章中，我们的虚拟机只处理表达式。当你运行clox时，它会解析、编译并执行一个表达式，然后打印结果。为了打印这个值，我们暂时使用`OP_RETURN`指令。我们让编译器在块的模块添加一条这样的指令。

compiler.c，在emitByte()方法后添加代码：

```
static void emitReturn() {
  emitByte(OP_RETURN);
}
```

> While we're here in the back end we may as well make our lives easier.

既然已经在编写后端，不妨让我们的工作更轻松一点。

compiler.c，在emitByte()方法后添加代码：

```
static void emitBytes(uint8_t byte1, uint8_t byte2) {
  emitByte(byte1);
  emitByte(byte2);
}
```

> Over time, we'll have enough cases where we need to write an opcode followed by a one-byte operand that it's worth defining this convenience function.

随着时间的推移，我们将遇到很多的情况中需要写一个操作码，后面跟一个单字节的操作数，因此值得定义这个便利的函数。

> ## 17.4 Parsing Prefix Expressions

17.4 解析前缀表达式

> We've assembled our parsing and code generation utility functions. The missing piece is the code in the middle that connects those together.

我们已经组装了解析和生成代码的工具函数。缺失的部分就是将它们连接在一起的的中间代码。

> The only step in `compile()` that we have left to implement is this function:

`compile()`中唯一还未实现的步骤就是这个函数：

compiler.c，在endCompiler()方法后添加代码：

```
static void expression() {
  // What goes here?
}
```

> We aren't ready to implement every kind of expression in Lox yet. Heck, we don't even have Booleans. For this chapter, we're only going to worry about four:

我们还没有准备好在Lox中实现每一种表达式。见鬼，我们甚至还没有布尔值。在本章中，我们只考虑四个问题：

> - Number literals: `123`
> - Parentheses for grouping: `(123)`
> - Unary negation: `-123`
> - The Four Horsemen of the Arithmetic: `+`, `-`, `*`, `/`

- 数值字面量：`123`
- 用于分组的括号：`(123)`
- 一元取负：`-123`
- 算术运算四骑士：`+`、`-`、`*`、`/`

> As we work through the functions to compile each of those kinds of expressions, we'll also assemble the requirements for the table-driven parser that calls them.

当我们通过函数编译每种类型的表达式时，我们也会对调用这些表达式的表格驱动的解析器的要求进行汇总。

> ### 17.4.1 Parsers for tokens

17.4.1 标识解析器

> For now, let's focus on the Lox expressions that are each only a single token. In this chapter, that's just number literals, but there will be more later. Here's how we can compile them:

现在，让我们把注意力集中在那些只由单个 token 组成的Lox表达式上。在本章中，这只包括数值字面量，但后面会有更多。下面是我们如何编译它们：

> We map each token type to a different kind of expression. We define a function for each expression that outputs the appropriate bytecode. Then we build an array of function pointers. The indexes in the array correspond to the `TokenType` enum values, and the function at each index is the code to compile an expression of that token type.

我们将每种标识类型映射到不同类型的表达式。我们为每个表达式定义一个函数，该函数会输出对应的字节码。然后我们构建一个函数指针的数组。数组中的索引对应于`TokenType`枚举值，每个索引处的函数是编译该标识类型的表达式的代码。

> To compile number literals, we store a pointer to the following function at the `TOKEN_NUMBER` index in the array.

为了编译数值字面量，我们在数组的`TOKEN_NUMBER`索引处存储一个指向下面函数的指针，

compiler.c，在endCompiler()方法后添加代码：

```
static void number() {
  double value = strtod(parser.previous.start, NULL);
  emitConstant(value);
}
```

> We assume the token for the number literal has already been consumed and is stored in `previous`. We take that lexeme and use the C standard library to convert it to a double value. Then we generate the code to load that value using this function:

我们假定数值字面量标识已经被消耗了，并被存储在`previous`中。我们获取该词素，并使用C标准库将其转换为一个double值。然后我们用下面的函数生成加载该double值的字节码：

compiler.c，在emitReturn()方法后添加代码：

```
static void emitConstant(Value value) {
  emitBytes(OP_CONSTANT, makeConstant(value));
}
```

> First, we add the value to the constant table, then we emit an `OP_CONSTANT` instruction that pushes it onto the stack at runtime. To insert an entry in the constant table, we rely on:

首先，我们将值添加到常量表中，然后我们发出一条`OP_CONSTANT`指令，在运行时将其压入栈中。要在常量表中插入一条数据，我们需要依赖：

compiler.c，在emitReturn()方法后添加代码：

```
static uint8_t makeConstant(Value value) {
  int constant = addConstant(currentChunk(), value);
  if (constant > UINT8_MAX) {
    error("Too many constants in one chunk.");
    return 0;
  }

  return (uint8_t)constant;
}
```

> Most of the work happens in `addConstant()`, which we defined back in an earlier chapter. That adds the given value to the end of the chunk's constant table and returns its index. The new function's job is mostly to make sure we don't have too many constants. Since the `OP_CONSTANT` instruction uses a single byte for the index operand, we can store and load only up to 256 constants in a chunk.

大部分的工作发生在`addConstant()`中，我们在前面的章节中定义了这个函数。它将给定的值添加到字节码块的常量表的末尾，并返回其索引。这个新函数的工作主要是确保我们没有太多常量。由于`OP_CONSTANT`指令使用单个字节来索引操作数，所以我们在一个块中最多只能存储和加载256个常量^4。

> That's basically all it takes. Provided there is some suitable code that consumes a `TOKEN_NUMBER` token, looks up `number()` in the function pointer array, and then calls it, we can now compile number literals to bytecode.

这基本就是所有的事情了。只要有了这些合适的代码，能够消耗一个`TOKEN_NUMBER`标识，在函数指针数组中查找`number()`方法，然后调用它，我们现在就可以将数值字面量编译为字节码。

> ### 17.4.2 Parentheses for grouping

17.4.2 括号分组

> Our as-yet-imaginary array of parsing function pointers would be great if every expression was only a single token long. Alas, most are longer. However, many expressions *start* with a particular token. We call these *prefix* expressions. For example, when we're parsing an expression and the current token is `(`, we know we must be looking at a parenthesized grouping expression.

如果每个表达式只有一个标识，那我们这个尚未成型的解析函数指针数组就很好处理了。不幸的是，大多数表达式都比较长。然而，许多表达式以一个特定的标识*开始*。我们称之为*前缀*表达式。举例来说，当我们解析一个表达式，而当前标识是`(`，我们就知道当前处理的一定是一个带括号的分组表达式。

> It turns out our function pointer array handles those too. The parsing function for an expression type can consume any additional tokens that it wants to, just like in a regular recursive descent parser. Here's how parentheses work:

事实证明，我们的函数指针数组也能处理这些。一个表达式类型的解析函数可以消耗任何它需要的标识，就像在常规的递归下降解析器中一样。下面是小括号的工作原理：

compiler.c，在endCompiler()方法后添加代码：

```
static void grouping() {
  expression();
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after expression.");
}
```

> Again, we assume the initial (has already been consumed. We recursively call back into `expression()` to compile the expression between the parentheses, then parse the closing) at the end.

同样，我们假定初始的(已经被消耗了。我们递归地^5调用`expression()`来编译括号之间的表达式，然后解析结尾的)。

> As far as the back end is concerned, there's literally nothing to a grouping expression. Its sole function is syntactic—it lets you insert a lower-precedence expression where a higher precedence is expected. Thus, it has no runtime semantics on its own and therefore doesn't emit any bytecode. The inner call to `expression()` takes care of generating bytecode for the expression inside the parentheses.

就后端而言，分组表达式实际上没有任何意义。它的唯一功能是语法上的——它允许你在需要高优先级的地方插入一个低优先级的表达式。因此，它本身没有运行时语法，也就不会发出任何字节码。对`expression()`的内部调用负责为括号内的表达式生成字节码。

> ### 17.4.3 Unary negation

17.4.3 一元取负

> Unary minus is also a prefix expression, so it works with our model too.

一元减号也是一个前缀表达式，因此也适用于我们的模型。

compiler.c，在number()方法后添加代码：

```
static void unary() {
  TokenType operatorType = parser.previous.type;

  // Compile the operand.
  expression();

  // Emit the operator instruction.
  switch (operatorType) {
    case TOKEN_MINUS: emitByte(OP_NEGATE); break;
    default: return; // Unreachable.
  }
}
```

> The leading - token has been consumed and is sitting in `parser.previous`. We grab the token type from that to note which unary operator we're dealing with. It's unnecessary right now, but this will make more sense when we use this same function to compile the ! operator in the next chapter.

前导的-标识已经被消耗掉了，并被放在`parser.previous`中。我们从中获取标识类型，以了解当前正在处理的是哪个一元运算符。现在还没必要这样做，但当下一章中我们使用这个函数来编译!时，这将会更有意义。

> As in `grouping()`, we recursively call `expression()` to compile the operand. After that, we emit the bytecode to perform the negation. It might seem a little weird to write the negate instruction *after* its operand's bytecode since the `-` appears on the left, but think about it in terms of order of execution:

就像在`grouping()`中一样，我们会递归地调用`expression()`来编译操作数。之后，我们发出字节码执行取负运算。因为`-`出现在左边，将取负指令放在其操作数的*后面*似乎有点奇怪，但是从执行顺序的角度来考虑：

> 1. We evaluate the operand first which leaves its value on the stack.
> 2. Then we pop that value, negate it, and push the result.

1. 首先计算操作数，并将其值留在堆栈中。
2. 然后弹出该值，对其取负，并将结果压入栈中。

> So the `OP_NEGATE` instruction should be emitted last. This is part of the compiler's job—parsing the program in the order it appears in the source code and rearranging it into the order that execution happens.

所以`OP_NEGATE`指令应该是最后发出的^6。这也是编译器工作的一部分——按照源代码中的顺序对程序进行解析，并按照执行的顺序对其重新排序。

> There is one problem with this code, though. The `expression()` function it calls will parse any expression for the operand, regardless of precedence. Once we add binary operators and other syntax, that will do the wrong thing. Consider:

不过，这段代码有一个问题。它所调用的`expression()`函数会解析操作数中的任何表达式，而不考虑优先级。一旦我们加入二元运算符和其它语法，就会出错。考虑一下：

```
-a.b + c;
```

> Here, the operand to `-` should be just the `a.b` expression, not the entire `a.b + c`. But if `unary()` calls `expression()`, the latter will happily chew through all of the remaining code including the `+`. It will erroneously treat the `-` as lower precedence than the `+`.

在这里`-`的操作数应该只是`a.b`表达式，而不是整个`a.b+c`。但如果`unary()`调用`expression()`，后者会愉快地处理包括`+`在内的所有剩余代码。它会错误地把`-`视为比`+`的优先级低。

> When parsing the operand to unary `-`, we need to compile only expressions at a certain precedence level or higher. In jlox's recursive descent parser we accomplished that by calling into the parsing method for the lowest-precedence expression we wanted to allow (in this case, `call()`). Each method for parsing a specific expression also parsed any expressions of higher precedence too, so that included the rest of the precedence table.

当解析一元`-`的操作数时，只需要编译具有某一优先级或更高优先级的表达式。在jlox的递归下降解析器中，我们通过调用我们想要允许的最低优先级的表达式的解析方法（在本例中是`call()`）来实现这一点。每个解析特定表达式的方法也会解析任何优先级更高的表达式，也就是包括优先级表的其余部分。

> The parsing functions like `number()` and `unary()` here in clox are different. Each only parses exactly one type of expression. They don't cascade to include higher-precedence expression types too. We need a different solution, and it looks like this:

clox中的`number()`和`unary()`这样的解析函数是不同的。每个函数只解析一种类型的表达式。它们不会级联处理更高优先级的表达式类型。我们需要一个不同的解决方案，看起来是这样的：

compiler.c，在unary()方法后添加代码：

```c
static void parsePrecedence(Precedence precedence) {
  // What goes here?
}
```

> This function—once we implement it—starts at the current token and parses any expression at the given precedence level or higher. We have some other setup to get through before we can write the body of this function, but you can probably guess that it will use that table of parsing function pointers I've been talking about. For now, don't worry too much about how it works. In order to take the "precedence" as a parameter, we define it numerically.

这个函数（一旦实现）从当前的标识开始，解析给定优先级或更高优先级的任何表达式。在编写这个函数的主体之前，我们还有一些其它的设置要完成，但你可能也猜得到，它会使用我一直在谈论的解析函数指针列表。现在，还不用太担心它的如何工作的。为了把"优先级"作为一个参数，我们用数值来定义它。

compiler.c，在结构体Parser后添加代码：

```c
} Parser;
// 新增部分开始
typedef enum {
  PREC_NONE,
  PREC_ASSIGNMENT,  // =
  PREC_OR,          // or
  PREC_AND,         // and
  PREC_EQUALITY,    // == !=
  PREC_COMPARISON,  // < > <= >=
  PREC_TERM,        // + -
  PREC_FACTOR,      // * /
  PREC_UNARY,       // ! -
  PREC_CALL,        // . ()
  PREC_PRIMARY
} Precedence;
// 新增部分结束
Parser parser;
```

> These are all of Lox's precedence levels in order from lowest to highest. Since C implicitly gives successively larger numbers for enums, this means that `PREC_CALL` is numerically larger than `PREC_UNARY`. For example, say the compiler is sitting on a chunk of code like:

这些是Lox中的所有优先级，按照从低到高的顺序排列。由于C语言会隐式地为枚举赋值连续递增的数字，这就意味着`PREC_CALL`在数值上比`PREC_UNARY`要大。举例来说，假设编译器正在处理这样的代码：

```
-a.b + c
```

> If we call `parsePrecedence(PREC_ASSIGNMENT)`, then it will parse the entire expression because `+` has higher precedence than assignment. If instead we call `parsePrecedence(PREC_UNARY)`, it will compile the `-a.b` and stop there. It doesn't keep going through the `+` because the addition has lower precedence than unary operators.

如果我们调用`parsePrecedence(PREC_ASSIGNMENT)`，那么它就会解析整个表达式，因为`+`的优先级高于赋值。如果我们调用`parsePrecedence(PREC_UNARY)`，它就会编译`-a.b`并停止。它不会径直解析`+`，因为加法的优先级比一元取负运算符要低。

> With this function in hand, it's a snap to fill in the missing body for `expression()`.

有了这个函数，我们就可以轻松地填充`expression()`的缺失部分。

compiler.c，在expression()方法中替换1行：

```
static void expression() {
  // 替换部分开始
  parsePrecedence(PREC_ASSIGNMENT);
  // 替换部分结束
}
```

> We simply parse the lowest precedence level, which subsumes all of the higher-precedence expressions too. Now, to compile the operand for a unary expression, we call this new function and limit it to the appropriate level:

我们只需要解析最低优先级，它也包含了所有更高优先级的表达式。现在，为了编译一元表达式的操作数，我们调用这个新函数并将其限制在适当的优先级：

compiler.c，在unary()方法中替换1行：

```
  // Compile the operand.
  // 替换部分开始
  parsePrecedence(PREC_UNARY);
  // 替换部分结束
  // Emit the operator instruction.
```

> We use the unary operator's own `PREC_UNARY` precedence to permit nested unary expressions like `!!doubleNegative`. Since unary operators have pretty high precedence, that correctly excludes things like binary operators. Speaking of which . . .

我们使用一元运算符本身的`PREC_UNARY`优先级来允许嵌套的一元表达式，如`!!doubleNegative`。因为一元运算符的优先级很高，所以正确地排除了二元运算符之类的东西。说到这一点......

> ## 17.5 Parsing Infix Expressions

17.5 解析中缀表达式

> Binary operators are different from the previous expressions because they are *infix*. With the other expressions, we know what we are parsing from the very first token. With infix expressions, we don't know we're in the middle of a binary operator until *after* we've parsed its left operand and then stumbled onto the operator token in the middle.

二元运算符与之前的表达式不同，因为它们是中缀的。对于其它表达式，我们从第一个标识就知道我们在解析什么，对于中缀表达式，只有在解析了左操作数并发现了中间的运算符时，才知道自己正在处理二元运算符。

> Here's an example:

下面是一个例子：

```
1 + 2
```

> Let's walk through trying to compile it with what we know so far:

让我们用目前已知的逻辑走一遍，试着编译一下它：

> 1. We call `expression()`. That in turn calls `parsePrecedence(PREC_ASSIGNMENT)`.
> 2. That function (once we implement it) sees the leading number token and recognizes it is parsing a number literal. It hands off control to `number()`.
> 3. `number()` creates a constant, emits an `OP_CONSTANT`, and returns back to `parsePrecedence()`.

1. 我们调用`expression()`，它会进一步调用`parsePrecedence(PREC_ASSIGNMENT)`
2. 该函数（一旦实现后）会看到前面的数字标识，并意识到正在解析一个数值字面量。它将控制权交给`number()`。
3. `number()`创建一个常数，发出一个`OP_CONSTANT`指令，然后返回到`parsePrecedence()`

> Now what? The call to `parsePrecedence()` should consume the entire addition expression, so it needs to keep going somehow. Fortunately, the parser is right where we need it to be. Now that we've compiled the leading number expression, the next token is `+`. That's the exact token that `parsePrecedence()` needs to detect that we're in the middle of an infix expression and to realize that the expression we already compiled is actually an operand to that.

现在怎么办？对 `parsePrecedence()`的调用应该要消费整个加法表达式，所以它需要以某种方式继续进行解析。幸运的是，解析器就在我们需要它的地方。现在我们已经编译了前面的数字表达式，下一个标识就是`+`。这正是`parsePrecedence()`用于判断我们是否在处理中缀表达式所需的标识，并意识到我们已经编译的表达式实际上是中缀表达式的操作数。

> So this hypothetical array of function pointers doesn't just list functions to parse expressions that start with a given token. Instead, it's a *table* of function pointers. One column associates prefix parser functions with token types. The second column associates infix parser functions with token types.

所以，这个假定的函数指针数组，不只是列出用于解析以指定标识开头的表达式的函数。相反，这个一个函数指针的*表格*。一列将前缀解析函数与标识类型关联起来，第二列将中缀解析函数与标识类型相关联。

> The function we will use as the infix parser for `TOKEN_PLUS`, `TOKEN_MINUS`, `TOKEN_STAR`, and `TOKEN_SLASH` is this:

我们将使用下面的函数作为`TOKEN_PLUS`，`TOKEN_MINUS`，`TOKEN_STAR`和`TOKEN_SLASH` 的中缀解析函数：

compiler.c，在endCompiler()方法后添加代码：

```
static void binary() {
  TokenType operatorType = parser.previous.type;
  ParseRule* rule = getRule(operatorType);
  parsePrecedence((Precedence)(rule->precedence + 1));

  switch (operatorType) {
    case TOKEN_PLUS:          emitByte(OP_ADD); break;
    case TOKEN_MINUS:         emitByte(OP_SUBTRACT); break;
    case TOKEN_STAR:          emitByte(OP_MULTIPLY); break;
    case TOKEN_SLASH:         emitByte(OP_DIVIDE); break;
    default: return; // Unreachable.
  }
}
```

> When a prefix parser function is called, the leading token has already been consumed. An infix parser function is even more *in medias res*—the entire left-hand operand expression has already been compiled and the subsequent infix operator consumed.

当前缀解析函数被调用时，前缀标识已经被消耗了。中缀解析函数被调用时，情况更进一步——整个左操作数已经被编译，而随后的中缀操作符也已经被消耗掉。

> The fact that the left operand gets compiled first works out fine. It means at runtime, that code gets executed first. When it runs, the value it produces will end up on the stack. That’s right where the infix operator is going to need it.

首先左操作数已经被编译的事实是很好的。这意味着在运行时，其代码已经被执行了。当它运行时，它产生的值最终进入栈中。而这正是中缀操作符需要它的地方。

> Then we come here to `binary()` to handle the rest of the arithmetic operators. This function compiles the right operand, much like how `unary()` compiles its own trailing operand. Finally, it emits the bytecode instruction that performs the binary operation.

然后我们使用`binary()`来处理算术操作符的其余部分。这个函数会编译右边的操作数，就像`unary()`编译自己的尾操作数那样。最后，它会发出执行对应二元运算的字节码指令。

> When run, the VM will execute the left and right operand code, in that order, leaving their values on the stack. Then it executes the instruction for the operator. That pops the two values, computes the operation, and pushes the result.

当运行时，虚拟机会按顺序执行左、右操作数的代码，将它们的值留在栈上。然后它会执行操作符的指令。这时，会从栈中弹出这两个值，计算结果，并将结果推入栈中。

> The code that probably caught your eye here is that `getRule()` line. When we parse the right-hand operand, we again need to worry about precedence. Take an expression like:

这里可能会引起你注意的代码是`getRule()`这一行。当我们解析右操作数时，我们又一次需要考虑优先级的问题。以下面这个表达式为例：

```
2 * 3 + 4
```

> When we parse the right operand of the `*` expression, we need to just capture `3`, and not `3 + 4`, because `+` is lower precedence than `*`. We could define a separate function for each binary operator. Each would call `parsePrecedence()` and pass in the correct precedence level for its operand.

当我们解析`*`表达式的右操作数时，我们只需要获取`3`，而不是`3+4`，因为`+`的优先级比`*`低。我们可以为每个二元运算符定义一个单独的函数。每个函数都会调用 `parsePrecedence()` 并传入正确的优先级来解析其操作数。

> But that’s kind of tedious. Each binary operator’s right-hand operand precedence is one level higher than its own. We can look that up dynamically with this `getRule()` thing we’ll get to soon. Using that, we call `parsePrecedence()` with one level higher than this operator’s level.

但这有点乏味。每个二元运算符的右操作数的优先级都比自己高一级^7。我们可以通过`getRule()`动态地查找，我们很快就会讲到。有了它，我们就可以使用比当前运算符高一级的优先级来调用`parsePrecedence()`。

> This way, we can use a single `binary()` function for all binary operators even though they have different precedences.

这样，我们就可以对所有的二元运算符使用同一个`binary()`函数，即使它们的优先级各不相同。

17.6 A Pratt Parser

17.6 Pratt解析器

> We now have all of the pieces and parts of the compiler laid out. We have a function for each grammar production: `number()`, `grouping()`, `unary()`, and `binary()`. We still need to implement `parsePrecedence()`, and `getRule()`. We also know we need a table that, given a token type, lets us find

现在我们已经排列好了编译器的所有部分。对于每个语法生成式都有对应的函数：`number()`, `grouping()`, `unary()` 和 `binary()`。我们仍然需要实现`parsePrecedence()`和`getRule()`。我们还知道，我们需要一个表格，给定一个标识类型，可以从中找到：

> - the function to compile a prefix expression starting with a token of that type,
> - the function to compile an infix expression whose left operand is followed by a token of that type, and
> - the precedence of an infix expression that uses that token as an operator.

- 编译以该类型标识为起点的前缀表达式的函数
- 编译一个左操作数后跟该类型标识的中缀表达式的函数，以及
- 使用该标识作为操作符的中缀表达式的优先级^8

> We wrap these three properties in a little struct which represents a single row in the parser table.

我们将这三个属性封装在一个小结构体中^9，该结构体表示解析器表格中的一行。

compiler.c，在枚举Precedence后添加代码：

```
} Precedence;
// 新增部分开始
typedef struct {
  ParseFn prefix;
  ParseFn infix;
  Precedence precedence;
} ParseRule;
// 新增部分结束
Parser parser;
```

> That ParseFn type is a simple typedef for a function type that takes no arguments and returns nothing.

这个ParseFn类型是一个简单的函数类型定义，这类函数不需要任何参数且不返回任何内容。

compiler.c，在枚举 Precedence后添加代码：

```
} Precedence;
// 新增部分开始
typedef void (*ParseFn)();
// 新增部分结束
typedef struct {
```

> The table that drives our whole parser is an array of ParseRules. We've been talking about it forever, and finally you get to see it.

驱动整个解析器的表格是一个ParserRule的数组。我们讨论了这么久，现在你终于可以看到它了^10。

compiler.c，在unary()方法后添加代码：

```
ParseRule rules[] = {
  [TOKEN_LEFT_PAREN]    = {grouping, NULL,   PREC_NONE},
  [TOKEN_RIGHT_PAREN]   = {NULL,     NULL,   PREC_NONE},
  [TOKEN_LEFT_BRACE]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_RIGHT_BRACE]   = {NULL,     NULL,   PREC_NONE},
  [TOKEN_COMMA]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_DOT]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_MINUS]         = {unary,    binary, PREC_TERM},
  [TOKEN_PLUS]          = {NULL,     binary, PREC_TERM},
  [TOKEN_SEMICOLON]     = {NULL,     NULL,   PREC_NONE},
  [TOKEN_SLASH]         = {NULL,     binary, PREC_FACTOR},
  [TOKEN_STAR]          = {NULL,     binary, PREC_FACTOR},
  [TOKEN_BANG]          = {NULL,     NULL,   PREC_NONE},
  [TOKEN_BANG_EQUAL]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_EQUAL]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_EQUAL_EQUAL]   = {NULL,     NULL,   PREC_NONE},
  [TOKEN_GREATER]       = {NULL,     NULL,   PREC_NONE},
  [TOKEN_GREATER_EQUAL] = {NULL,     NULL,   PREC_NONE},
  [TOKEN_LESS]          = {NULL,     NULL,   PREC_NONE},
  [TOKEN_LESS_EQUAL]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_IDENTIFIER]    = {NULL,     NULL,   PREC_NONE},
  [TOKEN_STRING]        = {NULL,     NULL,   PREC_NONE},
  [TOKEN_NUMBER]        = {number,   NULL,   PREC_NONE},
  [TOKEN_AND]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_CLASS]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_ELSE]          = {NULL,     NULL,   PREC_NONE},
  [TOKEN_FALSE]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_FOR]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_FUN]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_IF]            = {NULL,     NULL,   PREC_NONE},
  [TOKEN_NIL]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_OR]            = {NULL,     NULL,   PREC_NONE},
  [TOKEN_PRINT]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_RETURN]        = {NULL,     NULL,   PREC_NONE},
  [TOKEN_SUPER]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_THIS]          = {NULL,     NULL,   PREC_NONE},
  [TOKEN_TRUE]          = {NULL,     NULL,   PREC_NONE},
  [TOKEN_VAR]           = {NULL,     NULL,   PREC_NONE},
  [TOKEN_WHILE]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_ERROR]         = {NULL,     NULL,   PREC_NONE},
  [TOKEN_EOF]           = {NULL,     NULL,   PREC_NONE},
};
```

> You can see how `grouping` and `unary` are slotted into the prefix parser column for their respective token types. In the next column, `binary` is wired up to the four arithmetic infix operators. Those infix operators also have their precedences set in the last column.

你可以看到`grouping`和`unary`是如何被插入到它们各自标识类型对应的前缀解析器列中的。在下一列中，`binary`被连接到四个算术中缀操作符上。这些中缀操作符的优先级也设置在最后一列。

> Aside from those, the rest of the table is full of `NULL` and `PREC_NONE`. Most of those empty cells are because there is no expression associated with those tokens. You can't start an expression with, say, `else`, and `}` would make for a pretty confusing infix operator.

除此之外，表格的其余部分都是`NULL`和`PREC_NONE`。这些空的单元格中大部分是因为没有与这些标识相关联的表达式。比如说，你不能用`else`作为表达式开头，而`}`如果作为中缀操作符也会变得很混乱。

> But, also, we haven't filled in the entire grammar yet. In later chapters, as we add new expression types, some of these slots will get functions in them. One of the things I like about this approach to parsing is that it makes it very easy to see which tokens are in use by the grammar and which are available.

但是，我们还没有填入整个语法。在后面的章节中，当我们添加新的表达式类型时，其中一些槽会插入函数。我喜欢这种解析方法的一点是，它使我们很容易看到哪些标识被语法使用，以及哪些标识是可用的。

Now that we have the table, we are finally ready to write the code that uses it. This is where our Pratt parser comes to life. The easiest function to define is `getRule()`.

我们现在有了这个表格，终于准备好编写使用它的代码了。这就是我们的Pratt解析器发挥作用的地方。最容易定义的函数是`getRule()`。

compiler.c，在parsePrecedence()方法后添加代码：

```
static ParseRule* getRule(TokenType type) {
  return &rules[type];
}
```

> It simply returns the rule at the given index. It's called by `binary()` to look up the precedence of the current operator. This function exists solely to handle a declaration cycle in the C code. `binary()` is defined *before* the rules table so that the table can store a pointer to it. That means the body of `binary()` cannot access the table directly.

它只是简单地返回指定索引处的规则。`binary()`调用该函数来查询当前操作符的优先级。这个函数的存在只是为了处理C代码中的声明循环。`binary()`在规则表之前定义，以便规则表中可以存储指向它的指针。这也就意味着`binary()`的函数体不能直接访问表格。

> Instead, we wrap the lookup in a function. That lets us forward declare `getRule()` before the definition of `binary()`, and then *define* `getRule()` after the table. We'll need a couple of other forward declarations to handle the fact that our grammar is recursive, so let's get them all out of the way.

相反地，我们将查询封装在一个函数中。这样我们可以在`binary()`函数定义之前声明`getRule()`，然后在表格之后*定义*`getRule()`。我们还需要一些其它的前置声明来处理语法的递归，所以让我们一次性全部理出来。

compiler.c，在endCompiler()方法后添加代码：

```
  emitReturn();
}
// 新增部分开始
static void expression();
static ParseRule* getRule(TokenType type);
static void parsePrecedence(Precedence precedence);
// 新增部分结束
static void binary() {
```

> If you're following along and implementing clox yourself, pay close attention to the little annotations that tell you where to put these code snippets. Don't worry, though, if you get it wrong, the C compiler will be happy to tell you.

如果你正在跟随本文实现自己的clox，请密切注意那些告诉你代码片段应该加在哪里的小注释。不过不用担心，如果你弄错了，C编译器会很乐意告诉你。

> ### 17.6.1 Parsing with precedence

17.6.1 带优先级解析

> Now we're getting to the fun stuff. The maestro that orchestrates all of the parsing functions we've defined is `parsePrecedence()`. Let's start with parsing prefix expressions.

现在，我们要开始做有趣的事情了。我们定义的所有解析函数的协调者是 `parsePrecedence()`。让我们从解析前缀表达式开始。

compiler.c，在parsePrecedence()方法中替换一行：

```
static void parsePrecedence(Precedence precedence) {
  // 替换部分开始
  advance();
  ParseFn prefixRule = getRule(parser.previous.type)->prefix;
  if (prefixRule == NULL) {
    error("Expect expression.");
    return;
  }

  prefixRule();
  // 替换部分结束
}
```

> We read the next token and look up the corresponding ParseRule. If there is no prefix parser, then the token must be a syntax error. We report that and return to the caller.

我们读取下一个标识并查找对应的ParseRule。如果没有前缀解析器，那么这个标识一定是语法错误。我们会报告这个错误并返回给调用方。

> Otherwise, we call that prefix parse function and let it do its thing. That prefix parser compiles the rest of the prefix expression, consuming any other tokens it needs, and returns back here. Infix expressions are where it gets interesting since precedence comes into play. The implementation is remarkably simple.

否则，我们就调用前缀解析函数，让它做自己的事情。该前缀解析器会编译表达式的其余部分，消耗它需要的任何其它标识，然后返回这里。中缀表达式是比较有趣的地方，因为优先级开始发挥作用了。这个实现非常简单。 *compiler.c，在parsePrecedence()方法中添加代码：*

```
  prefixRule();
  // 新增部分开始
  while (precedence <= getRule(parser.current.type)->precedence) {
    advance();
    ParseFn infixRule = getRule(parser.previous.type)->infix;
    infixRule();
  }
  // 新增部分结束
}
```

> That's the whole thing. Really. Here's how the entire function works: At the beginning of `parsePrecedence()`, we look up a prefix parser for the current token. The first token is *always* going

> to belong to some kind of prefix expression, by definition. It may turn out to be nested as an operand inside one or more infix expressions, but as you read the code from left to right, the first token you hit always belongs to a prefix expression.

这就是全部内容了，真的。下面是整个函数的工作原理：在`parsePrecedence()`的开头，我们会为当前标识查找对应的前缀解析器。根据定义，第一个标识*总是*属于某种前缀表达式。它可能作为一个操作数嵌套在一个或多个中缀表达式中，但是当你从左到右阅读代码时，你碰到的第一个标识总是属于一个前缀表达式。

> After parsing that, which may consume more tokens, the prefix expression is done. Now we look for an infix parser for the next token. If we find one, it means the prefix expression we already compiled might be an operand for it. But only if the call to `parsePrecedence()` has a `precedence` that is low enough to permit that infix operator.

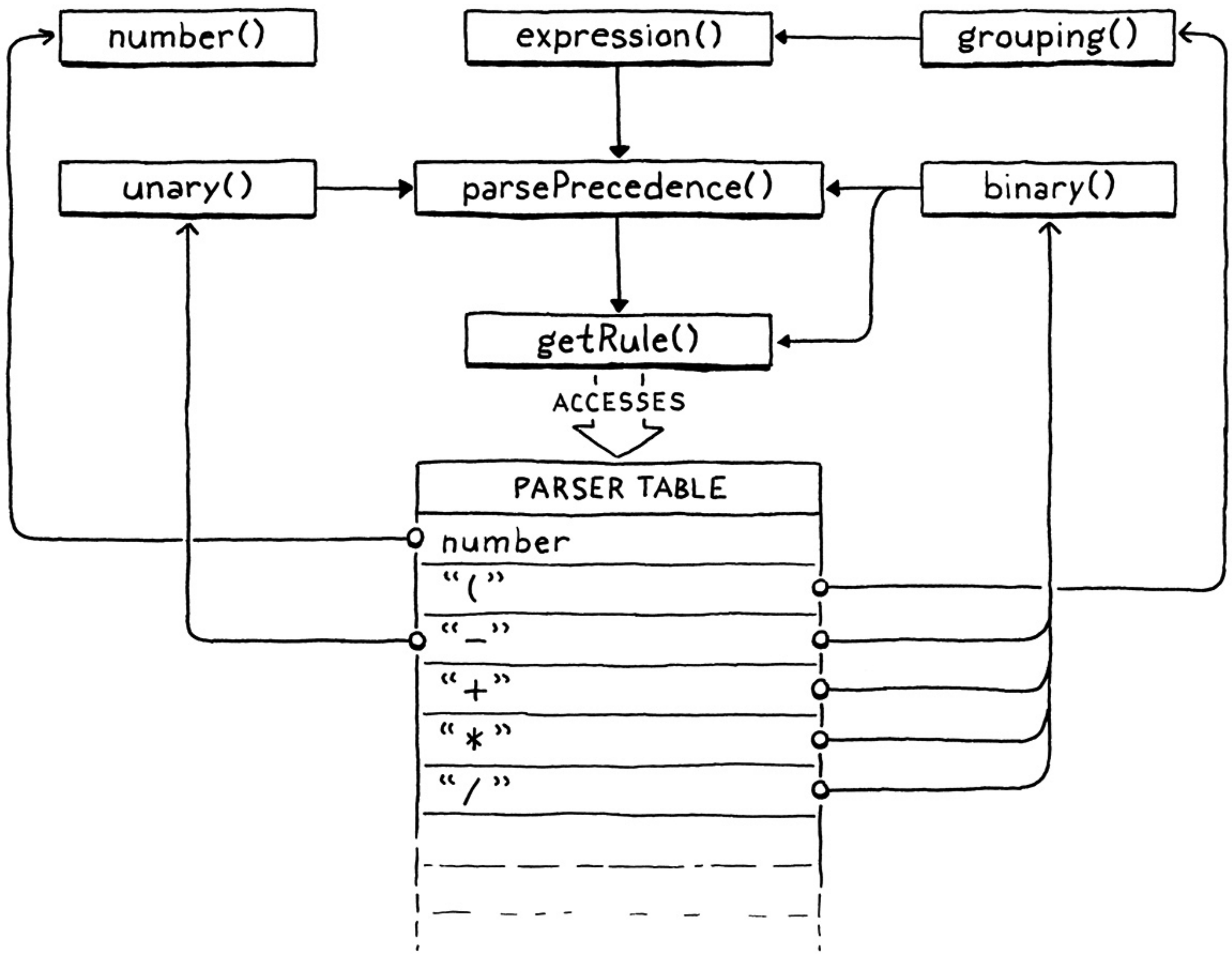
解析之后（可能会消耗更多的标识），前缀表达式就完成了。现在我们要为下一个标识寻找一个中缀解析器。如果我们找到了，就意味着我们刚刚编译的前缀表达式可能是它的一个操作数。但前提是调用`parsePrecedence()` 时传入的`precedence`允许该中缀操作符。

> If the next token is too low precedence, or isn't an infix operator at all, we're done. We've parsed as much expression as we can. Otherwise, we consume the operator and hand off control to the infix parser we found. It consumes whatever other tokens it needs (usually the right operand) and returns back to `parsePrecedence()`. Then we loop back around and see if the *next* token is also a valid infix operator that can take the entire preceding expression as its operand. We keep looping like that, crunching through infix operators and their operands until we hit a token that isn't an infix operator or is too low precedence and stop.

如果下一个标识的优先级太低，或者根本不是一个中缀操作符，我们就结束了。我们已经尽可能多地解析了表达式。否则，我们就消耗操作符，并将控制权移交给我们发现的中缀解析器。它会消耗所需要的其它标识（通常是右操作数）并返回到`parsePrecedence()`。然后我们再次循环，并查看下一个标识符是否也是一个有效的中缀操作符，且该操作符可以把前面的整个表达式作为其操作数。我们就这样一直循环下去，直到遇见一个不是中缀操作符或优先级太低的标识，然后停止。

> That's a lot of prose, but if you really want to mind meld with Vaughan Pratt and fully understand the algorithm, step through the parser in your debugger as it works through some expressions. Maybe a picture will help. There's only a handful of functions, but they are marvelously intertwined:

这是一篇冗长的文章，但是如果你真的想与Vaughan Pratt心意相通，完全理解这个算法，你可以让解析器处理一些表达式，然后在调试器中逐步查看解析器。也许图片会有帮助，只有少数几个函数，但它们奇妙地交织在一起^11。



> Later, we'll need to tweak the code in this chapter to handle assignment. But, otherwise, what we wrote covers all of our expression compiling needs for the rest of the book. We'll plug additional parsing functions into the table when we add new kinds of expressions, but `parsePrecedence()` is complete.

稍后，我们在处理赋值的时候需要调整本章中的代码。但是，除此之外，我们所写的内容涵盖了本书中其余部分所有表达式编译的需求。在添加新的表达式类型时，我们会在表格中插入额外的解析函数，但是 `parsePrecedence()` 是完整的。

17.7 Dumping Chunks

17.7 转储字节码块

> While we're here in the core of our compiler, we should put in some instrumentation. To help debug the generated bytecode, we'll add support for dumping the chunk once the compiler finishes. We had some temporary logging earlier when we hand-authored the chunk. Now we'll put in some real code so that we can enable it whenever we want.

既然我们已经进入了编译器的核心，我们就应该加入一些工具。为了有助于调试生成的字节码，我们会增加对编译器完成后转储字节码块的支持。在之前我们手工编写字节码块时，进行了一些临时的日志记录。现在，我们要填入一些实际的代码，以便我们可以随时启用它。

> Since this isn't for end users, we hide it behind a flag.

因为这不是为终端用户准备的，所以我们把它隐藏在一个标志后面。

common.h，添加代码：

```
#include <stdint.h>
// 新增部分开始
#define DEBUG_PRINT_CODE
// 新增部分结束
#define DEBUG_TRACE_EXECUTION
```

> When that flag is defined, we use our existing “debug” module to print out the chunk’s bytecode.

当这个标志被定义后，我们使用现有的“debug”模块打印出块中的字节码。

compiler.c，在endCompiler()方法中添加代码：

```
  emitReturn();
// 新增部分开始
#ifdef DEBUG_PRINT_CODE
  if (!parser.hadError) {
    disassembleChunk(currentChunk(), "code");
  }
#endif
// 新增部分结束
}
```

> We do this only if the code was free of errors. After a syntax error, the compiler keeps on going but it’s in kind of a weird state and might produce broken code. That’s harmless because it won’t get executed, but we’ll just confuse ourselves if we try to read it.

只有在代码没有错误的情况下，我们才会这样做。在出现语法错误后，编译器会继续运行，但它会处于一种奇怪的状态，可能会产生错误的代码。不过这是无害的，因为它不会被执行，但如果我们试图阅读它，只会把我们弄糊涂。

> Finally, to access `disassembleChunk()`, we need to include its header.

最后，为了访问`disassembleChunk()`，我们需要包含它的头文件。

compiler.c，添加代码：

```
#include "scanner.h"
// 新增部分开始
#ifdef DEBUG_PRINT_CODE
#include "debug.h"
#endif
// 新增部分结束
typedef struct {
```

> We made it! This was the last major section to install in our VM's compilation and execution pipeline. Our interpreter doesn't *look* like much, but inside it is scanning, parsing, compiling to bytecode, and executing.

我们成功了！这是我们的虚拟机的编译和执行管道中需要安装的最后一个主要部分。我们的解释器*看起来*不大，但它内部有扫描、解析、编译字节码并执行。

> Fire up the VM and type in an expression. If we did everything right, it should calculate and print the result. We now have a very over-engineered arithmetic calculator. We have a lot of language features to add in the coming chapters, but the foundation is in place.

启动虚拟机，输入一个表达式。如果我们所有操作都正确，它应该会计算并打印结果。我们现在有了一个过度设计的算术计算器。在接下来的章节中，我们还好添加很多语言特性，但是基础已经准备好了。

^1: 如果你对这一章不感兴趣，而你又希望从另一个角度了解这些概念，我写过一篇文章讲授了同样的算法，但使用了Java和面向对象的风格："Pratt Parsing: Expression Parsing Made Easy" ^2: 事实上，大多数复杂的优化编译器都不止两遍执行过程。不仅要确定需要进行哪些优化，还要确定如何安排它们的顺序——因为优化往往以复杂的方式相互作用——这是介于"开放的研究领域"和"黑暗的艺术"之间的问题。 ^3: 有`setjmp()`和`longjmp()`，但我不想使用它们。这些使我们很容易泄漏内存、忘记维护不变量，或者说寝食难安。 ^4: 确实，这个限制是很低的。如果这是一个完整的语言实现，我们应该添加另一个指令，比如`OP_CONSTANT_16`，将索引存储为两字节的操作数，这样就可以在需要时处理更多的常量。支持这个指令的代码不是特别有启发性，所以我在clox中省略了它，但你会希望你的虚拟机能够扩展成更大的程序。 ^5: Pratt解析器不是递归下降解析器，但它仍然是递归的。这是意料之中的，因为语法本身是递归的。 ^6: 在操作数之后发出`OP_NEGATE`确实意味着写入字节码时的当前标识不是`-`标识。但这并不重要，除了我们使用标识中的行号与指令相关联。这意味着，如果你有一个多行的取负表达式，比如

```
print -
  true;
```

那么运行时错误会报告在错误的代码行上。这里，它将在第2行显示错误，而`-`是在第一行。一个更稳健的方法是在编译器操作数之前存储标识中的行号，然后将其传递给`emitByte()`，当我想在本书中尽量保持简单。 ^7: 我们对右操作数使用高一级的优先级，因为二元操作符是左结合的。给出一系列相同的运算符，如：
`1+2+3+4`
我们想这样解析它：
`((1+2)+3)+4`
因此，当解析第一个`+`的右侧操作数时，我们希望消耗`2`，但不消耗其余部分，所以我们使用比`+`高一个优先级的操作数。但如果我们的操作符是右结合的，这就错了。考虑一下：
`a=b=c=d`
因为赋值是右结合的，我们希望将其解析为：
`a=(b=(c=d))`
为了实现这一点，我们会使用与当前操作符*相同*的优先级来调用`parsePrecedence()`。 ^8: 我们不需要跟踪以指定标识开头的前缀表达式的优先级，因为Lox中的所有前缀操作符都有相同的优先级。 ^9: C语言中函数指针类型的语法非常糟糕，所以我总是把它隐藏在类型定义之后。我理解这种语法背后的意图——整个"声明反映使用"之类的——但我认为这是一个失败的语法实验。 ^10: 现在明白我所说的"不想每次需要新列时都重新审视这个表格"是什么意思了吧？这就是个野兽。也许你没有见过C语言数组字面量中的`[TOKEN_DOT]=`语法，这是C99指定的初始化器语法。这比手动计算数组索引要清楚得多。 ^11: 箭头连接一个函数与其直接调用的另一个函数， 箭头连接表格中的指针与解析函数。

CHALLENGES

习题

1. > To really understand the parser, you need to see how execution threads through the interesting parsing functions—`parsePrecedence()` and the parser functions stored in the table. Take this (strange) expression:

   要真正理解解析器，你需要查看执行线程如何通过有趣的解析函数——`parsePrecedence()`和表格中的解析器函数。以这个（奇怪的）表达式为例：

   ```
   (-1 + 2) * 3 - -4
   ```

   > Write a trace of how those functions are called. Show the order they are called, which calls which, and the arguments passed to them.

   写一下关于这些函数如何被调用的追踪信息。显示它们被调用的顺序，哪个调用哪个，以及传递给它们的参数。

2. > The ParseRule row for `TOKEN_MINUS` has both prefix and infix function pointers. That's because `-` is both a prefix operator (unary negation) and an infix one (subtraction).
   >
   > In the full Lox language, what other tokens can be used in both prefix and infix positions? What about in C or in another language of your choice?

   `TOKEN_MINUS`的ParseRule行同时具有前缀和中缀函数指针。这是因为`-`既是前缀操作符（一元取负），也是一个中缀操作符（减法）。

   在完整的Lox语言中，还有哪些标识可以同时用于前缀和中缀位置？在C语言或你选择的其它语言中呢？

3. > You might be wondering about complex "mixfix" expressions that have more than two operands separated by tokens. C's conditional or "ternary" operator, `?:`, is a widely known one.
   >
   > Add support for that operator to the compiler. You don't have to generate any bytecode, just show how you would hook it up to the parser and handle the operands.

   你可能会好奇负责的"多元"表达式，他有两个以上的操作数，操作数之间由标识分开。C语言中的条件运算符或"三元"运算符`?:`就是一个广为人知的多元操作符。

   向编译器中添加对该运算符的支持。你不需要生成任何字节码，只需要展示如何将其连接到解析器中并处理操作数。

DESIGN NOTE: IT'S JUST PARSING

设计笔记：只是解析

> I'm going to make a claim here that will be unpopular with some compiler and language people. It's OK if you don't agree. Personally, I learn more from strongly stated opinions that I disagree with than I do from several pages of qualifiers and equivocation. My claim is that *parsing doesn't matter*.

> Over the years, many programming language people, especially in academia, have gotten *really* into parsers and taken them very seriously. Initially, it was the compiler folks who got into compiler-compilers, LALR, and other stuff like that. The first half of the dragon book is a long love letter to the wonders of parser generators.
>
> All of us suffer from the vice of “when all you have is a hammer, everything looks like a nail”, but perhaps none so visibly as compiler people. You wouldn’t believe the breadth of software problems that miraculously seem to require a new little language in their solution as soon as you ask a compiler hacker for help.
>
> Yacc and other compiler-compilers are the most delightfully recursive example. “Wow, writing compilers is a chore. I know, let’s write a compiler to write our compiler for us.”
>
> For the record, I don’t claim immunity to this affliction.
>
> Later, the functional programming folks got into parser combinators, packrat parsers, and other sorts of things. Because, obviously, if you give a functional programmer a problem, the first thing they’ll do is whip out a pocketful of higher-order functions.
>
> Over in math and algorithm analysis land, there is a long legacy of research into proving time and memory usage for various parsing techniques, transforming parsing problems into other problems and back, and assigning complexity classes to different grammars.
>
> At one level, this stuff is important. If you’re implementing a language, you want some assurance that your parser won’t go exponential and take 7,000 years to parse a weird edge case in the grammar. Parser theory gives you that bound. As an intellectual exercise, learning about parsing techniques is also fun and rewarding.
>
> But if your goal is just to implement a language and get it in front of users, almost all of that stuff doesn’t matter. It’s really easy to get worked up by the enthusiasm of the people who *are* into it and think that your front end *needs* some whiz-bang generated combinator-parser-factory thing. I’ve seen people burn tons of time writing and rewriting their parser using whatever today’s hot library or technique is.
>
> That’s time that doesn’t add any value to your user’s life. If you’re just trying to get your parser done, pick one of the bog-standard techniques, use it, and move on. Recursive descent, Pratt parsing, and the popular parser generators like ANTLR or Bison are all fine.
>
> Take the extra time you saved not rewriting your parsing code and spend it improving the compile error messages your compiler shows users. Good error handling and reporting is more valuable to users than almost anything else you can put time into in the front end.

我在这里要提出一个主张，这个主张可能不被一些编译器和语言人士所欢迎。如果你不同意也没关系。就我个人而言，比起几页的限定词和含糊其辞，从那些我不同意的强烈的观点中学习到的东西更多。我的主张是，解析并不重要。

多年来，许多从事编程语言的人，尤其是在学术界，确实是真正地深入了解析器，并且非常认真地对待它们^12。最初，是编译器研究者，他们深入研究编译器的编译器、LALR，以及其它类似的东西。龙书的前半部分就是写给对解析器生成器好奇的人的一封长信。

后来，函数式编程人员开始研究解析器组合子、packrat解析器和其它类型的东西。原因很明显，如果你给函数式程序员提出一个问题，他们要做的第一件事就是拿出一堆高阶函数。

在数学和算法分析领域，长期以来一直在研究证明各种解析技术的时间和内存使用情况，将解析问题转换为其它问题，并为不同的语法进行复杂性分类。

在某种程度上，这些东西很重要。如果你正在实现一门语言，你希望能够确保你的解析器复杂度不会是指数级，不会花费7000年时间来解析语法中的一个奇怪的边界情况。解析器理论给了你这种约束。作为一项智力练习，学习解析技术也是很有趣和有意义的。

但是，如果你的目标只是实现一门语言并将其送到用户面前，那么几乎所有这些都不重要了。你很容易被那些对语言感兴趣的人们的热情所感染，认为你的前端*需要*一些快速生成的解析器组合子工厂之类的东西。我见过人们花费大量的时间，使用当下最热门的库或技术，编写或重写他们的解析器。

这些时间并不会给用户的生活带来任何价值。如果你只是想完成解析器，那么可以选择一个普通的标准技术，使用它，然后继续前进。递归下降法，Pratt解析和流行的解析器生成器（如ANTLR或Bison）都很不错。

把你不用重写解析代码而节省下来的额外时间，花在改进编译器向用户显示的编译错误信息上。对用户来说，良好的错误处理和报告比你在语言前端投入时间所做的几乎任何事情都更有价值。

18.值类型 Types of Values

> When you are a Bear of Very Little Brain, and you Think of Things, you find sometimes that a Thing which seemed very Thingish inside you is quite different when it gets out into the open and has other people looking at it.
>
> —— A. A. Milne, *Winnie-the-Pooh*

你要是一只脑子很小的熊，当你想事情的时候，你会发现，有时在你心里看起来很像回事的事情，当它展示出来，让别人看着它的时候，就完全不同了。（A. A.米尔恩，《小熊维尼》）

> The past few chapters were huge, packed full of complex techniques and pages of code. In this chapter, there's only one new concept to learn and a scattering of straightforward code. You've earned a respite.

前面的几章篇幅很长，充满了复杂的技术和一页又一页的代码。在本章中，只需要学习一个新概念和一些简单的代码。你获得了喘息的机会。

> Lox is dynamically typed. A single variable can hold a Boolean, number, or string at different points in time. At least, that's the idea. Right now, in clox, all values are numbers. By the end of the chapter, it will also support Booleans and `nil`. While those aren't super interesting, they force us to figure out how our value representation can dynamically handle different types.

Lox是动态类型的^1。一个变量可以在不同的时间点持有布尔值、数字或字符串。至少，我们的想法是如此。现在，在clox中，所有的值都是数字。到本章结束时，它还将支持布尔值和`nil`。虽然这些不是特别有趣，但它们迫使我们弄清楚值表示如何动态地处理不同类型。

> ## 18.1 Tagged Unions

18.1 带标签联合体

> The nice thing about working in C is that we can build our data structures from the raw bits up. The bad thing is that we *have* to do that. C doesn't give you much for free at compile time and even less at runtime. As far as C is concerned, the universe is an undifferentiated array of bytes. It's up to us to decide how many of those bytes to use and what they mean.

使用C语言工作的好处是，我们可以从最基础的比特位开始构建数据结构。坏处是，我们必须这样做。C语言在编译时并没有提供多少免费的东西，在运行时就更少了。对C语言来说，宇宙是一个无差别的字节数组。由我们来决定使用多少个字节以及它们的含义。

> In order to choose a value representation, we need to answer two key questions:

为了选择一种值的表示形式，我们需要先回答两个关键问题：

1. > **How do we represent the type of a value?** If you try to, say, multiply a number by `true`, we need to detect that error at runtime and report it. In order to do that, we need to be able to tell what a value's type is.

   我们如何表示一个值的类型？ 比如说，如果你将一个数字乘以`true`，我们需要在运行时检测到这个错误并报告它。为此，我们需要知道值的类型是什么？

2. > **How do we store the value itself?** We need to not only be able to tell that three is a number, but that it's different from the number four. I know, seems obvious, right? But we're operating at a level where it's good to spell these things out.

   我们如何存储该值本身？ 我们不仅要能分辨出3是一个数字，还要能分辨出它与4是不同的。我知道，这是显而易见的对吧？但是在我们所讨论的层面，最好把这些事情说清楚。

> Since we're not just designing this language but building it ourselves, when answering these two questions we also have to keep in mind the implementer's eternal quest: to do it *efficiently*.

因为我们不仅仅是设计这门语言，还要自己构建它，所以在回答这两个问题时，我们还必须牢记实现者们永恒的追求：*高效*地完成它。

> Language hackers over the years have come up with a variety of clever ways to pack the above information into as few bits as possible. For now, we'll start with the simplest, classic solution: a **tagged union**. A value contains two parts: a type "tag", and a payload for the actual value. To store the value's type, we define an enum for each kind of value the VM supports.

多年来，语言黑客们想出了各种巧妙的方法，将上述信息打包成尽可能少的比特。现在，我们将从最简单、最经典的解决方案开始：**带标签的联合体**。一个值包含两个部分：一个类型"标签"，和一个实际值的有效载荷。为了存储值的类型，我们要为虚拟机支持的每一种值定义一个枚举^2。

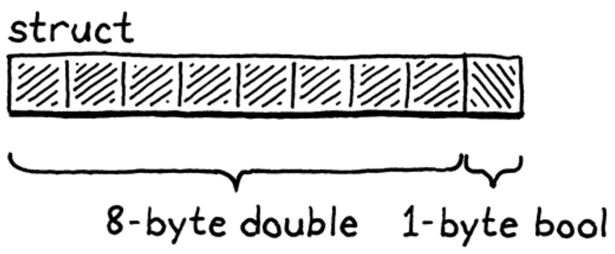
value.h，添加代码：

```
#include "common.h"
// 新增部分开始
typedef enum {
  VAL_BOOL,
  VAL_NIL,
  VAL_NUMBER,
} ValueType;
```

```c
  // 新增部分结束
  typedef double Value;
```

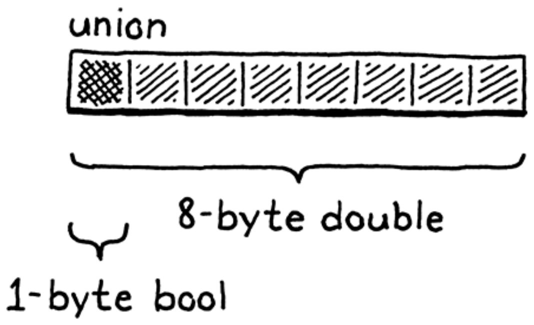
> For now, we have only a couple of cases, but this will grow as we add strings, functions, and classes to clox. In addition to the type, we also need to store the data for the value—the `double` for a number, `true` or `false` for a Boolean. We could define a struct with fields for each possible type.

现在，我们只有这几种情况，但随着我们向clox中添加字符串、函数和类，这里也会越来越多。除了类型之外，我们还需要存储值的数据——数字是`double`值，Boolean是`true`或`false`。我们可以定义一个结构体，其中包含每种可能的类型所对应的字段。



> But this is a waste of memory. A value can't simultaneously be both a number and a Boolean. So at any point in time, only one of those fields will be used. C lets you optimize this by defining a union. A union looks like a struct except that all of its fields overlap in memory.

但这是对内存的一种浪费。一个值不可能同时是数字和布尔值。所以在任何时候，这些字段中只有一个会被使用。C语言中允许定义联合体来优化这一点。联合体看起来很像是结构体，区别在于其中的所有字段在内存中是重叠的。



> The size of a union is the size of its largest field. Since the fields all reuse the same bits, you have to be very careful when working with them. If you store data using one field and then access it using another, you will reinterpret what the underlying bits mean.

联合体的大小就是其最大字段的大小。由于这些字段都复用相同的比特位，你在使用它们时必须要非常小心。如果你使用一个字段存储数据，然后用另一个字段访问数据，那你需要重新解释底层比特位的含义^3。

> As the name "tagged union" implies, our new value representation combines these two parts into a single struct.

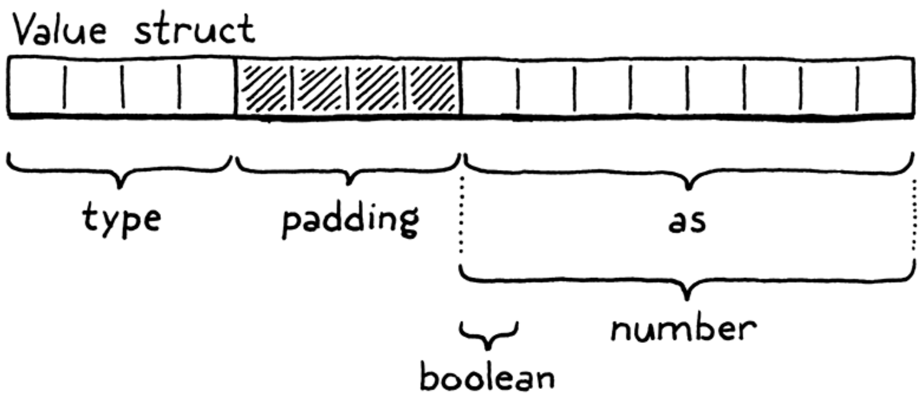
顾名思义，"带标签的联合体"说明，我们新的值表示形式中将这两部分合并成一个结构体。

value.h，在枚举ValueType后替换一行：

```
} ValueType;
// 替换部分开始
typedef struct {
  ValueType type;
  union {
    bool boolean;
    double number;
  } as;
} Value;
// 替换部分结束
typedef struct {
```

> There's a field for the type tag, and then a second field containing the union of all of the underlying values. On a 64-bit machine with a typical C compiler, the layout looks like this:

有一个字段用作类型标签，然后是第二个字段，一个包含所有底层值的联合体^4。在使用典型的C语言编译器的64位机器上，布局看起来如下：



> The four-byte type tag comes first, then the union. Most architectures prefer values be aligned to their size. Since the union field contains an eight-byte double, the compiler adds four bytes of padding after the type field to keep that double on the nearest eight-byte boundary. That means we're effectively spending eight bytes on the type tag, which only needs to represent a number between zero and three. We could stuff the enum in a smaller size, but all that would do is increase the padding.

首先是4字节的类型标签，然后是联合体。大多数体系结构都喜欢将值与它们的字长对齐。由于联合体字段中包含一个8字节的double值，所以编译器在类型字段后添加了4个字节的填充，以使该double值保持在最近的8字节边界上。这意味着我们实际在类型标签上花费了8个字节，而它只需要表示0到3之间的数字。我们可以把枚举放在一个占位更少的变量中，但这样做只会增加填充量^5。

> So our Values are 16 bytes, which seems a little large. We'll improve it later. In the meantime, they're still small enough to store on the C stack and pass around by value. Lox's semantics allow that because the only types we support so far are **immutable**. If we pass a copy of a Value containing the number three to some function, we don't need to worry about the caller seeing modifications to the value. You can't "modify" three. It's three forever.

所以我们的Value是16个字节，这似乎有点大。我们稍后会改进它。同时，它们也足够小，可以存储在C语言的堆栈中，并按值传递。Lox的语义允许这样做，因为到目前为止我们只支持**不可变**类型。如果我们把一个包含数

字3的Value的副本传递给某个函数，我们不需要担心调用者会看到对该值的修改。你不能"修改"3，它永远都是3。

18.2 Lox Values and C Values

18.2 Lox值和C值

> That's our new value representation, but we aren't done. Right now, the rest of clox assumes Value is an alias for `double`. We have code that does a straight C cast from one to the other. That code is all broken now. So sad.

这就是我们新的值表示形式，但是我们还没有做完。现在，clox的其余部分都假定了Value是`double`的别名。我们有一些代码是直接用C语言将一个值转换为另一个值。这些代码现在都被破坏了，好伤心。

> With our new representation, a Value can *contain* a double, but it's not *equivalent* to it. There is a mandatory conversion step to get from one to the other. We need to go through the code and insert those conversions to get clox working again.

在我们新的表示形式中，Value可以*包含*一个double值，但它并不等同于double类型。有一个强制性的转换步骤可以实现从一个值到另一个值的转换。我们需要遍历代码并插入这些转换步骤，以使clox重新工作。

> We'll implement these conversions as a handful of macros, one for each type and operation. First, to promote a native C value to a clox Value:

我们会用少量的宏来实现这些转换，每个宏对应一个类型和操作。首先，将原生的C值转换为clox Value：

value.h，在结构体Value后添加代码：

```
} Value;
// 新增部分开始
#define BOOL_VAL(value)   ((Value){VAL_BOOL, {.boolean = value}})
#define NIL_VAL           ((Value){VAL_NIL, {.number = 0}})
#define NUMBER_VAL(value) ((Value){VAL_NUMBER, {.number = value}})
// 新增部分结束
typedef struct {
```

> Each one of these takes a C value of the appropriate type and produces a Value that has the correct type tag and contains the underlying value. This hoists statically typed values up into clox's dynamically typed universe. In order to *do* anything with a Value, though, we need to unpack it and get the C value back out.

其中每个宏都接收一个适当类型的C值，并生成一个Value，其具有正确类型标签并包含底层的值。这就把静态类型的值提升到了clox的动态类型的世界。但是为了能对Value做任何操作，我们需要将其拆包并取出对应的C值^6。

value.h，在结构体Value后添加代码：

```
} Value;
// 新增部分开始
```

```
#define AS_BOOL(value)    ((value).as.boolean)
#define AS_NUMBER(value)  ((value).as.number)
// 新增部分结束
#define BOOL_VAL(value)   ((Value){VAL_BOOL, {.boolean = value}})
```

> These macros go in the opposite direction. Given a Value of the right type, they unwrap it and return the corresponding raw C value. The “right type” part is important! These macros directly access the union fields. If we were to do something like:

这些宏的作用是反方向的。给定一个正确类型的Value，它们会将其解包并返回对应的原始C值。“正确类型”很重要！这些宏会直接访问联合体字段。如果我们要这样做：

```
Value value = BOOL_VAL(true);
double number = AS_NUMBER(value);
```

> Then we may open a smoldering portal to the Shadow Realm. It’s not safe to use any of the `AS_` macros unless we know the Value contains the appropriate type. To that end, we define a last few macros to check a Value’s type.

那我们可能会打开一个通往暗影王国的阴燃之门。除非我们知道Value包含适当的类型，否则使用任何的`AS_`宏都是不安全的。为此，我们定义最后几个宏来检查Value的类型。

value.h，在结构体Value后添加代码：

```
} Value;
// 新增部分开始
#define IS_BOOL(value)    ((value).type == VAL_BOOL)
#define IS_NIL(value)     ((value).type == VAL_NIL)
#define IS_NUMBER(value)  ((value).type == VAL_NUMBER)
// 新增部分结束
#define AS_BOOL(value)    ((value).as.boolean)
```

> These macros return `true` if the Value has that type. Any time we call one of the `AS_` macros, we need to guard it behind a call to one of these first. With these eight macros, we can now safely shuttle data between Lox’s dynamic world and C’s static one.

如果Value具有对应类型，这些宏会返回`true`。每当我们调用一个`AS_`宏时，我们都需要保证首先调用了这些宏。有了这8个宏，我们现在可以安全地在Lox的动态世界和C的静态世界之间传输数据了。

> ## 18.3 Dynamically Typed Numbers

18.3 动态类型数字

> We’ve got our value representation and the tools to convert to and from it. All that’s left to get clox running again is to grind through the code and fix every place where data moves across that boundary. This is one of those sections of the book that isn’t exactly mind-blowing, but I promised I’d show you every single line of code, so here we are.

我们已经有了值的表示形式和转换的工具。要想让clox重新运行起来，剩下的工作就是仔细检查代码，修复每个数据跨边界传递的地方。这是本书中不太让人兴奋的章节之一，但我保证会给你展示每一行代码，所以我们开始吧。

> The first values we create are the constants generated when we compile number literals. After we convert the lexeme to a C double, we simply wrap it in a Value before storing it in the constant table.

我们创建的第一个值是在编译数值字面量时生成的常量。在我们将词素转换为C语言的double之后，我们简单地将其包装在一个Value中，然后再存储到常量表中。

compiler.c，在number()函数中替换一行：

```
    double value = strtod(parser.previous.start, NULL);
    // 替换部分开始
    emitConstant(NUMBER_VAL(value));
    // 替换部分结束
  }
```

> Over in the runtime, we have a function to print values.

在运行时，我们有一个函数来打印值。

value.c，在printValue()方法中替换一行：

```
  void printValue(Value value) {
   // 替换部分开始
   printf("%g", AS_NUMBER(value));
   // 替换部分结束
  }
```

> Right before we send the Value to `printf()`, we unwrap it and extract the double value. We'll revisit this function shortly to add the other types, but let's get our existing code working first.

在我们将Value发送给`printf()`之前，我们将其拆装并提取出double值。我们很快会重新回顾这个函数并添加其它类型，但是我们先让现有的代码工作起来。

> ### 18.3.1 Unary negation and runtime errors

18.3.1 一元取负与运行时错误

> The next simplest operation is unary negation. It pops a value off the stack, negates it, and pushes the result. Now that we have other types of values, we can't assume the operand is a number anymore. The user could just as well do:

接下来最简单的操作是一元取负。它会从栈中弹出一个值，对其取负，并将结果压入栈。现在我们有了其它类型的值，我们不能再假设操作数是一个数字。用户也可以这样做：

```
print -false; // Uh...
```

> We need to handle that gracefully, which means it's time for *runtime errors*. Before performing an operation that requires a certain type, we need to make sure the Value *is* that type.

我们需要优雅地处理这个问题，这意味着是时候讨论运行时错误了。在执行需要特定类型的操作之前，我们需要确保Value是该类型。

> For unary negation, the check looks like this:

对于一元取负来说，检查是这样的：

vm.c，在run()方法中替换一行：

```
      case OP_DIVIDE:   BINARY_OP(/); break;
      // 替换部分开始
      case OP_NEGATE:
        if (!IS_NUMBER(peek(0))) {
          runtimeError("Operand must be a number.");
          return INTERPRET_RUNTIME_ERROR;
        }
        push(NUMBER_VAL(-AS_NUMBER(pop())));
        break;
      // 替换部分结束
      case OP_RETURN: {
```

> First, we check to see if the Value on top of the stack is a number. If it's not, we report the runtime error and stop the interpreter. Otherwise, we keep going. Only after this validation do we unwrap the operand, negate it, wrap the result and push it.

首先，我们检查栈顶的Value是否是一个数字。如果不是，则报告运行时错误并停止解释器^7。否则，我们就继续运行。只有在验证之后，我们才会拆装操作数，取负，将结果封装并压入栈。

> To access the Value, we use a new little function.

为了访问Value，我们使用一个新的小函数。

vm.c，在pop()方法后添加代码：

```
static Value peek(int distance) {
  return vm.stackTop[-1 - distance];
}
```

> It returns a Value from the stack but doesn't pop it. The `distance` argument is how far down from the top of the stack to look: zero is the top, one is one slot down, etc.

它从堆栈中返回一个Value，但是并不弹出它^8。`distance`参数是指要从堆栈顶部向下看多远：0是栈顶，1是下一个槽，以此类推。

> We report the runtime error using a new function that we’ll get a lot of mileage out of over the remainder of the book.

我们使用一个新函数来报告运行时错误，在本书的剩余部分，我们会从中得到很多的好处。

vm.c，在resetStack()方法后添加代码：

```
static void runtimeError(const char* format, ...) {
  va_list args;
  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
  fputs("\n", stderr);

  size_t instruction = vm.ip - vm.chunk->code - 1;
  int line = vm.chunk->lines[instruction];
  fprintf(stderr, "[line %d] in script\n", line);
  resetStack();
}
```

> You’ve certainly *called* variadic functions—ones that take a varying number of arguments—in C before: `printf()` is one. But you may not have *defined* your own. This book isn’t a C tutorial, so I’ll skim over it here, but basically the `...` and `va_list` stuff let us pass an arbitrary number of arguments to `runtimeError()`. It forwards those on to `vfprintf()`, which is the flavor of `printf()` that takes an explicit `va_list`.

你以前肯定在C语言中调用过变参函数——接受不同数量参数的函数：`printf()`就是其中之一。但你可能还没*定义*过自己的变参函数。这本书不是C语言教程^9，所以我在这里略过了，但是基本上是`...`和`va_list`让我们可以向`runtimeError()`传递任意数量的参数。它将这些参数转发给`vfprintf()`，这是`printf()`的一个变体，需要一个显式地`va_list`。

> Callers can pass a format string to `runtimeError()` followed by a number of arguments, just like they can when calling `printf()` directly. `runtimeError()` then formats and prints those arguments. We won’t take advantage of that in this chapter, but later chapters will produce formatted runtime error messages that contain other data.

调用者可以向`runtimeError()`传入一个格式化字符串，后跟一些参数，就像他们直接调用`printf()`一样。然后`runtimeError()`格式化并打印这些参数。在本章中我们不会利用这一点，但后面的章节中将生成包含其它数据的格式化运行时错误信息。

> After we show the hopefully helpful error message, we tell the user which line of their code was being executed when the error occurred. Since we left the tokens behind in the compiler, we look up the line in the debug information compiled into the chunk. If our compiler did its job right, that corresponds to the line of source code that the bytecode was compiled from.

在显示了希望有帮助的错误信息之后，我们还会告诉用户，当错误发生时正在执行代码中的哪一行^10。因为我们在编译器中留下了标识，所以我们可以从编译到字节码块中的调试信息中查找行号。如果我们的编译器正确完成了它的工作，就能对应到字节码被编译出来的那一行源代码。

> We look into the chunk's debug line array using the current bytecode instruction index *minus one*. That's because the interpreter advances past each instruction before executing it. So, at the point that we call `runtimeError()`, the failed instruction is the previous one.

我们使用当前字节码指令索引减1来查看字节码块的调试行数组。这是因为解释器在之前每条指令之前都会向前推进。所以，当我们调用 `runtimeError()`，失败的指令就是前一条。

Just showing the immediate line where the error occurred doesn't provide much context. Better would be a full stack trace. But we don't even have functions to call yet, so there is no call stack to trace.

> In order to use `va_list` and the macros for working with it, we need to bring in a standard header.

为了使用`va_list`和相关的宏，我们需要引入一个标准头文件。

vm.c，在文件顶部添加代码：

```
// 新增部分开始
#include <stdarg.h>
// 新增部分结束
#include <stdio.h>
```

> With this, our VM can not only do the right thing when we negate numbers (like it used to before we broke it), but it also gracefully handles erroneous attempts to negate other types (which we don't have yet, but still).

有了它，我们的虚拟机不仅可以在对数字取负时正确执行（原本就会这样做），而且还可以优雅地处理对其它类型取负的错误尝试（目前还没有，但仍然存在）。

> ### 18.3.2 Binary arithmetic operators

18.3.2 二元数字运算符

> We have our runtime error machinery in place now, so fixing the binary operators is easier even though they're more complex. We support four binary operators today: `+`, `-`, `*`, and `/`. The only difference between them is which underlying C operator they use. To minimize redundant code between the four operators, we wrapped up the commonality in a big preprocessor macro that takes the operator token as a parameter.

我们现在已经有了运行时错误机制，所以修复二元运算符更容易，尽管它们更复杂。现在我们支持四种二元运算符：`+`、`-`、`*`和`/`。它们之间唯一的区别就是使用的是哪种底层C运算符。为了尽量减少这四个运算符之间的冗余代码，我们将它们的共性封装在一个大的预处理宏中，该宏以运算符标识作为参数。

> That macro seemed like overkill a few chapters ago, but we get the benefit from it today. It lets us add the necessary type checking and conversions in one place.

这个宏在前几章中似乎是多余的，但现在我们却从中受益。它让我们可以在某个地方添加必要的类型检查和转换。 *vm.c，在run()方法中替换6行：*

```
#define READ_CONSTANT() (vm.chunk>constants.values[READ_BYTE()])
// 替换部分开始
#define BINARY_OP(valueType, op) \
    do { \
      if (!IS_NUMBER(peek(0)) || !IS_NUMBER(peek(1))) { \
        runtimeError("Operands must be numbers."); \
        return INTERPRET_RUNTIME_ERROR; \
      } \
      double b = AS_NUMBER(pop()); \
      double a = AS_NUMBER(pop()); \
      push(valueType(a op b)); \
    } while (false)
 // 替换部分结束
  for (;;) {
```

> Yeah, I realize that's a monster of a macro. It's not what I'd normally consider good C practice, but let's roll with it. The changes are similar to what we did for unary negate. First, we check that the two operands are both numbers. If either isn't, we report a runtime error and yank the ejection seat lever.

是的，我知道这是一个巨大的宏。这不是我通常认为的好的C语言实践，但我们还是用它吧。这些调整与我们对一元取负所做的相似。首先，我们检查两个操作数是否都是数字。如果其中一个不是，我们就报告一个运行时错误，并拉下弹射座椅手柄。

> If the operands are fine, we pop them both and unwrap them. Then we apply the given operator, wrap the result, and push it back on the stack. Note that we don't wrap the result by directly using `NUMBER_VAL()`. Instead, the wrapper to use is passed in as a macro parameter. For our existing arithmetic operators, the result is a number, so we pass in the `NUMBER_VAL` macro.

如果操作数都没有问题，我们就把它们都弹出栈并进行拆装。然后我们应用给定的运算符，包装结果并将其压回栈中。注意，我们没有直接使用`NUMBER_VAL()`来包装结果。相反，我们要使用的包装器是作为宏参数传入的。对于我们现有的数字运算符来说，结果是一个数字，所以我们传入`NUMBER_VAL`宏^11。

vm.c，在run()方法中替换4行：

```
      }
      // 替换部分开始
      case OP_ADD:      BINARY_OP(NUMBER_VAL, +); break;
      case OP_SUBTRACT: BINARY_OP(NUMBER_VAL, -); break;
      case OP_MULTIPLY: BINARY_OP(NUMBER_VAL, *); break;
      case OP_DIVIDE:   BINARY_OP(NUMBER_VAL, /); break;
      // 替换部分结束
      case OP_NEGATE:
```

> Soon, I'll show you why we made the wrapping macro an argument.

很快，我就会告诉你为什么我们要将包装宏作为参数。

18.4 Two New Types

18.4 两个新类型

> All of our existing clox code is back in working order. Finally, it’s time to add some new types. We’ve got a running numeric calculator that now does a number of pointless paranoid runtime type checks. We can represent other types internally, but there’s no way for a user’s program to ever create a Value of one of those types.

我们现有的所有clox代码都恢复正常工作了。最后，是时候添加一些新类型了。我们有一个正在运行的数字计算器，它现在做了一些毫无意义的偏执的运行时类型检查。我们可以在内部表示其它类型，但用户的程序无法创建这些类型的Value。

> Not until now, that is. We’ll start by adding compiler support for the three new literals: `true`, `false`, and `nil`. They’re all pretty simple, so we’ll do all three in a single batch.

现在还不能。首先，我们向编译器添加对三个新字面量的支持：`true`、`false`、`nil`。它们都很简单，所以我们一次性完成这三个。

> With number literals, we had to deal with the fact that there are billions of possible numeric values. We attended to that by storing the literal’s value in the chunk’s constant table and emitting a bytecode instruction that simply loaded that constant. We could do the same thing for the new types. We’d store, say, `true`, in the constant table, and use an `OP_CONSTANT` to read it out.

对于数字字面量，我们要面对这样一个事实：有数十亿个可能的数字值。为此，我们将字面量的值保存在字节码块的常量表中，并生成一个加载该常量的字节码指令。我们可以对这些新类型做同样的事。我们在常量表中存储值，比如`true`，并使用`OP_CONSTANT`来读取它。

> But given that there are literally (heh) only three possible values we need to worry about with these new types, it’s gratuitous—and slow!—to waste a two-byte instruction and a constant table entry on them. Instead, we’ll define three dedicated instructions to push each of these literals on the stack.

但是考虑到这些新类型实际上只有三种可能的值，这样做是没有必要的——而且速度很慢！——浪费了一个两字节的指令和常量表中的一个项。相反，我们会定义三个专用指令^12来将这些字面量压入栈中。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CONSTANT,
  // 新增部分开始
  OP_NIL,
  OP_TRUE,
  OP_FALSE,
  // 新增部分结束
  OP_ADD,
```

> Our scanner already treats `true`, `false`, and `nil` as keywords, so we can skip right to the parser. With our table-based Pratt parser, we just need to slot parser functions into the rows associated with those

> keyword token types. We'll use the same function in all three slots. Here:

我们的扫描器已经将`true`、`false`和`nil`视为关键字，所以我们可以直接调到解析器。对于我们这个基于表格的Pratt解析器，只需要将解析器函数插入到与这些关键字标识类型相对应的行中。我们会在三个槽中使用相同的函数。这里：

compiler.c，替换一行：

```
  [TOKEN_ELSE]          = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_FALSE]         = {literal,  NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_FOR]           = {NULL,     NULL,   PREC_NONE},
```

这里:

compiler.c，替换一行：

```
  [TOKEN_THIS]          = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_TRUE]          = {literal,  NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_VAR]           = {NULL,     NULL,   PREC_NONE},
```

还有这里:

compiler.c，替换一行：

```
  [TOKEN_IF]            = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_NIL]           = {literal,  NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_OR]            = {NULL,     NULL,   PREC_NONE},
```

> When the parser encounters `false`, `nil`, or `true`, in prefix position, it calls this new parser function:

当解析器在前缀位置遇到`false`、`nil`或 `true`时，它会调用这个新的解析器函数：

compiler.c，在binary()方法后添加代码：

```
static void literal() {
  switch (parser.previous.type) {
    case TOKEN_FALSE: emitByte(OP_FALSE); break;
    case TOKEN_NIL: emitByte(OP_NIL); break;
    case TOKEN_TRUE: emitByte(OP_TRUE); break;
    default: return; // Unreachable.
    }
  }
```

> Since `parsePrecedence()` has already consumed the keyword token, all we need to do is output the proper instruction. We figure that out based on the type of token we parsed. Our front end can now compile Boolean and nil literals to bytecode. Moving down the execution pipeline, we reach the interpreter.

因为`parsePrecedence()`已经消耗了关键字标识，我们需要做的就是输出正确的指令。我们根据解析出的标识的类型来确定指令。我们的前端现在可以将布尔值和`nil`字面量编译为字节码。沿着执行管道向下移动，我们就到了解释器。

vm.c，在run()方法中添加代码：

```
        case OP_CONSTANT: {
          Value constant = READ_CONSTANT();
          push(constant);
          break;
        }
        // 新增部分开始
        case OP_NIL: push(NIL_VAL); break;
        case OP_TRUE: push(BOOL_VAL(true)); break;
        case OP_FALSE: push(BOOL_VAL(false)); break;
        // 新增部分结束
        case OP_ADD:      BINARY_OP(NUMBER_VAL, +); break;
```

> This is pretty self-explanatory. Each instruction summons the appropriate value and pushes it onto the stack. We shouldn't forget our disassembler either.

这一点是不言而喻的。每条指令都会召唤出相应的值并将其压入堆栈。我们也不能忘记反汇编程序。

debug.c，在disassembleInstruction()方法中添加代码：

```
      case OP_CONSTANT:
        return constantInstruction("OP_CONSTANT", chunk, offset);
      // 新增部分开始
      case OP_NIL:
        return simpleInstruction("OP_NIL", offset);
      case OP_TRUE:
        return simpleInstruction("OP_TRUE", offset);
      case OP_FALSE:
        return simpleInstruction("OP_FALSE", offset);
      // 新增部分结束
      case OP_ADD:
```

> With this in place, we can run this Earth-shattering program:

有了这些，我们就可以运行这个惊天动地的程序：

```
true
```

> Except that when the interpreter tries to print the result, it blows up. We need to extend `printValue()` to handle the new types too:

只是当解释器试图打印结果时，就崩溃了。我们也需要扩展`printValue()`来处理新类型：

value.c，在printValue()方法中替换1行：

```
void printValue(Value value) {
  // 替换部分开始
  switch (value.type) {
    case VAL_BOOL:
      printf(AS_BOOL(value) ? "true" : "false");
      break;
    case VAL_NIL: printf("nil"); break;
    case VAL_NUMBER: printf("%g", AS_NUMBER(value)); break;
  }
  // 替换部分结束
}
```

> There we go! Now we have some new types. They just aren't very useful yet. Aside from the literals, you can't really *do* anything with them. It will be a while before `nil` comes into play, but we can start putting Booleans to work in the logical operators.

我们继续！现在我们有了一些新的类型，只是它们目前还不是很有用。除了字面量之外，你无法真正对其做任何事。还需要一段时间`nil`才会发挥作用，但我们可以先让布尔值在逻辑运算符中发挥作用。

> ### 18.4.1 Logical not and falsiness

18.4.1 逻辑非和falsiness

> The simplest logical operator is our old exclamatory friend unary not.

最简单的逻辑运算符是我们充满感叹意味的老朋友一元取非。

```
print !true; // "false"
```

> This new operation gets a new instruction.

这个新操作会有一条新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_DIVIDE,
  // 新增部分开始
  OP_NOT,
  // 新增部分结束
  OP_NEGATE,
```

> We can reuse the `unary()` parser function we wrote for unary negation to compile a not expression. We just need to slot it into the parsing table.

我们可以重用为一元取负所写的解析函数来编译一个逻辑非表达式。我们只需要将其插入到解析表格中。

compiler.c，替换一行：

```
  [TOKEN_STAR]          = {NULL,     binary, PREC_FACTOR},
  // 替换部分开始
  [TOKEN_BANG]          = {unary,    NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_BANG_EQUAL]    = {NULL,     NULL,   PREC_NONE},
```

> Because I knew we were going to do this, the `unary()` function already has a switch on the token type to figure out which bytecode instruction to output. We merely add another case.

因为我之前已知道我们要这样做，`unary()`函数已经有了关于标识类型的switch语句，来判断要输出哪个字节码指令。我们只需要增加一个分支即可。

compiler.c，在unary()方法中添加代码：

```
  switch (operatorType) {
    // 新增部分开始
    case TOKEN_BANG: emitByte(OP_NOT); break;
    // 新增部分结束
    case TOKEN_MINUS: emitByte(OP_NEGATE); break;
    default: return; // Unreachable.
  }
```

> That's it for the front end. Let's head over to the VM and conjure this instruction into life.

前端就这样了。让我们去虚拟机那里，并将这个指令变成现实。

vm.c，在run()方法中添加代码：

```
      case OP_DIVIDE:   BINARY_OP(NUMBER_VAL, /); break;
      // 新增部分开始
      case OP_NOT:
        push(BOOL_VAL(isFalsey(pop())));
        break;
      // 新增部分结束
      case OP_NEGATE:
```

> Like our previous unary operator, it pops the one operand, performs the operation, and pushes the result. And, as we did there, we have to worry about dynamic typing. Taking the logical not of `true` is easy, but there's nothing preventing an unruly programmer from writing something like this:

跟之前的一元运算符一样，它会弹出一个操作数，执行操作，并将结果压入栈中。正如我们所做的那样，我们必须考虑动态类型。对`true`进行逻辑取非很容易，但没什么能阻止一个不守规矩的程序员写出这样的东西：

```
print !nil;
```

> For unary minus, we made it an error to negate anything that isn't a number. But Lox, like most scripting languages, is more permissive when it comes to `!` and other contexts where a Boolean is expected. The rule for how other types are handled is called "falsiness", and we implement it here:

对于一元取负，我们把对任何非数字的东西进行取负^13当作一个错误。但是Lox，像大多数脚本语言一样，在涉及到`!`和其它期望出现布尔值的情况下，是比较宽容的。处理其它类型的规则被称为"falsiness"，我们在这里实现它：

vm.c，在peek()方法后添加代码：

```
static bool isFalsey(Value value) {
  return IS_NIL(value) || (IS_BOOL(value) && !AS_BOOL(value));
}
```

> Lox follows Ruby in that `nil` and `false` are falsey and every other value behaves like `true`. We've got a new instruction we can generate, so we also need to be able to *un*generate it in the disassembler.

Lox遵循Ruby的规定，`nil`和`false`是假的，其它的值都表现为`true`。我们已经有了一条可以生成的新指令，所以我们也需要能够在反汇编程序中反生成它。

debug.c，在disassembleInstruction()方法中添加代码：

```
    case OP_DIVIDE:
      return simpleInstruction("OP_DIVIDE", offset);
    // 新增部分开始
    case OP_NOT:
      return simpleInstruction("OP_NOT", offset);
    // 新增部分结束
    case OP_NEGATE:
```

> ## 18.4.2 Equality and comparison operators

18.4.2 相等与比较运算符

> That wasn't too bad. Let's keep the momentum going and knock out the equality and comparison operators too: `==`, `!=`, `<`, `>`, `<=`, and `>=`. That covers all of the operators that return Boolean results

> except the logical operators and and or. Since those need to short-circuit (basically do a little control flow) we aren't ready for them yet.

还不算太糟。让我们继续保持这种势头，搞定相等与比较运算符：`==`，`!=`，`<`，`>`，`<=`和`>=`。这涵盖了所有会返回布尔值的运算符，除了逻辑运算符`and`和`or`。因为这些运算符需要短路计算（基本上是做一个小小的控制流），我们还没准备好。

> Here are the new instructions for those operators:

下面是这些运算符对应的新指令：

chunk.h，在枚举OpCode中添加代码：

```
  OP_FALSE,
  // 新增部分开始
  OP_EQUAL,
  OP_GREATER,
  OP_LESS,
  // 新增部分结束
  OP_ADD,
```

> Wait, only three? What about `!=`, `<=`, and `>=`? We could create instructions for those too. Honestly, the VM would execute faster if we did, so we *should* do that if the goal is performance.

等一下，只有三个？那`!=`、`<=`和`>=`呢？我们也可以为它们创建指令。老实说，如果我们这样做，虚拟机的执行速度会更快。所以如果我们的目标是追求性能，那就*应该*这样做。

> But my main goal is to teach you about bytecode compilers. I want you to start internalizing the idea that the bytecode instructions don't need to closely follow the user's source code. The VM has total freedom to use whatever instruction set and code sequences it wants as long as they have the right user-visible behavior.

但我的主要目标是教你有关字节码编译器的知识。我想要你开始内化一个想法：字节码指令不需要紧跟用户的源代码。虚拟机可以完全自由地使用它想要的任何指令集和代码序列，只要它们有正确的用户可见的行为。

> The expression `a != b` has the same semantics as `!(a == b)`, so the compiler is free to compile the former as if it were the latter. Instead of a dedicated `OP_NOT_EQUAL` instruction, it can output an `OP_EQUAL` followed by an `OP_NOT`. Likewise, `a <= b` is the same as `!(a > b)` and `a >= b` is `!(a < b)`. Thus, we only need three new instructions.

表达式`a!=b`与`!(a==b)`具有相同的语义^14，所以编译器可以自由地编译前者，就好像它是后者一样。它可以输出一条`OP_EQUAL`指令，之后是一条`OP_NOT`，而不是一条专用的`OP_NOT_EQUAL`指令。同样地，`a<=b`与`!(a>b)`相同，而`a>=b`与`!(a<b)`相同，所以我们只需要三条新指令。

> Over in the parser, though, we do have six new operators to slot into the parse table. We use the same `binary()` parser function from before. Here's the row for `!=`:

不过，在解析器中，我们确实有6个新的操作符要加入到解析表中。我们使用与之前相同的`binary()`解析函数。下面是`!=`对应的行：

compiler.c，替换1行：

```
  [TOKEN_BANG]          = {unary,    NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_BANG_EQUAL]    = {NULL,     binary, PREC_EQUALITY},
  // 替换部分结束
  [TOKEN_EQUAL]         = {NULL,     NULL,   PREC_NONE},
```

> The remaining five operators are a little farther down in the table.

其余五个运算符在表的最下方。

compiler.c，替换5行：

```
  [TOKEN_EQUAL]         = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_EQUAL_EQUAL]   = {NULL,     binary, PREC_EQUALITY},
  [TOKEN_GREATER]       = {NULL,     binary, PREC_COMPARISON},
  [TOKEN_GREATER_EQUAL] = {NULL,     binary, PREC_COMPARISON},
  [TOKEN_LESS]          = {NULL,     binary, PREC_COMPARISON},
  [TOKEN_LESS_EQUAL]    = {NULL,     binary, PREC_COMPARISON},
  // 替换部分结束
  [TOKEN_IDENTIFIER]    = {NULL,     NULL,   PREC_NONE},
```

> Inside `binary()` we already have a switch to generate the right bytecode for each token type. We add cases for the six new operators.

在`binary()`中，我们已经有了一个switch语句，为每种标识类型生成正确的字节码。我们为这六个新运算符添加分支。

compiler.c，在binary()方法内添加代码：

```
  switch (operatorType) {
    // 新增部分开始
    case TOKEN_BANG_EQUAL:    emitBytes(OP_EQUAL, OP_NOT); break;
    case TOKEN_EQUAL_EQUAL:   emitByte(OP_EQUAL); break;
    case TOKEN_GREATER:       emitByte(OP_GREATER); break;
    case TOKEN_GREATER_EQUAL: emitBytes(OP_LESS, OP_NOT); break;
    case TOKEN_LESS:          emitByte(OP_LESS); break;
    case TOKEN_LESS_EQUAL:    emitBytes(OP_GREATER, OP_NOT); break;
    // 新增部分结束
    case TOKEN_PLUS:          emitByte(OP_ADD); break;
```

> The `==`, `<`, and `>` operators output a single instruction. The others output a pair of instructions, one to evalute the inverse operation, and then an `OP_NOT` to flip the result. Six operators for the price of three instructions!

`==`、`<`和`>` 运算符输出单个指令。其它运算符则输出一对指令，一条用于计算逆运算，然后用`OP_NOT`来反转结果。仅仅使用三种指令就表达出了六种运算符的效果！

> That means over in the VM, our job is simpler. Equality is the most general operation.

这意味着在虚拟机中，我们的工作更简单了。相等是最普遍的操作。

vm.c，在run()方法中添加代码：

```
        case OP_FALSE: push(BOOL_VAL(false)); break;
        // 新增部分开始
        case OP_EQUAL: {
          Value b = pop();
          Value a = pop();
          push(BOOL_VAL(valuesEqual(a, b)));
          break;
        }
        // 新增部分结束
        case OP_ADD:      BINARY_OP(NUMBER_VAL, +); break;
```

> You can evaluate `==` on any pair of objects, even objects of different types. There's enough complexity that it makes sense to shunt that logic over to a separate function. That function always returns a C `bool`, so we can safely wrap the result in a `BOOL_VAL`. The function relates to Values, so it lives over in the "value" module.

你可以对任意一对对象执行`==`，即使这些对象是不同类型的。这有足够的复杂性，所以有必要把这个逻辑分流到一个单独的函数中。这个函数会一个C语言的`bool`值，所以我们可以安全地把结果包装在一个`BOLL_VAL`中。这个函数与Value有关，所以它位于"value"模块中。

value.h，在结构体ValueArray后添加代码：

```
} ValueArray;
// 新增部分开始
bool valuesEqual(Value a, Value b);
// 新增部分结束
void initValueArray(ValueArray* array);
```

> And here's the implementation:

下面是实现：

value.c，在printValue()方法后添加代码：

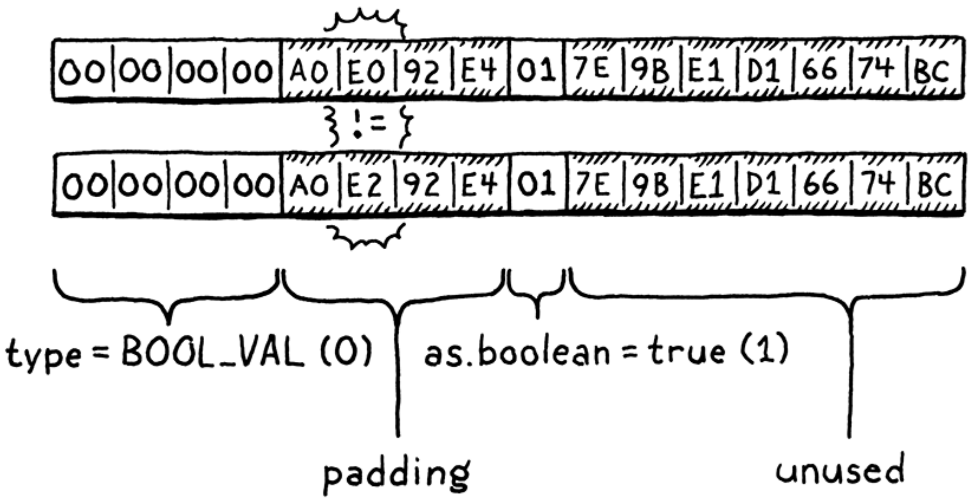
```
bool valuesEqual(Value a, Value b) {
  if (a.type != b.type) return false;
  switch (a.type) {
    case VAL_BOOL:   return AS_BOOL(a) == AS_BOOL(b);
    case VAL_NIL:    return true;
    case VAL_NUMBER: return AS_NUMBER(a) == AS_NUMBER(b);
    default:         return false; // Unreachable.
  }
}
```

> First, we check the types. If the Values have different types, they are definitely not equal. Otherwise, we unwrap the two Values and compare them directly.

首先，我们检查类型。如果两个Value的类型不同，它们肯定不相等^15。否则，我们就把这两个Value拆装并直接进行比较。

> For each value type, we have a separate case that handles comparing the value itself. Given how similar the cases are, you might wonder why we can't simply `memcmp()` the two Value structs and be done with it. The problem is that because of padding and different-sized union fields, a Value contains unused bits. C gives no guarantee about what is in those, so it's possible that two equal Values actually differ in memory that isn't used.

对于每一种值类型，我们都有一个单独的case分支来处理值本身的比较。考虑到这些分支的相似性，你可能会想，为什么我们不能简单地对两个Value结构体进行`memcmp()`，然后就可以了。问题在于，因为填充以及联合体字段的大小不同，Value中会包含无用的比特位。C语言不能保证这些值是什么，所以两个相同的Value在未使用的内存中可能是完全不同的。


00 00 00 00 A0 E0 92 E4 01 7E 9B E1 D1 66 74 BC
!=
00 00 00 00 A0 E2 92 E4 01 7E 9B E1 D1 66 74 BC
type = BOOL_VAL (0)
as.boolean = true (1)
padding
unused

> (You wouldn't believe how much pain I went through before learning this fact.)

(你无法想象在了解这个事实之前我经历了多少痛苦。)

> Anyway, as we add more types to clox, this function will grow new cases. For now, these three are sufficient. The other comparison operators are easier since they work only on numbers.

总之，随着我们向clox中添加更多的类型，这个函数也会增加更多的case分支。就目前而言，这三个已经足够了。其它的比较运算符更简单，因为它们只处理数字。

vm.c，在run()方法中添加代码：

```
        push(BOOL_VAL(valuesEqual(a, b)));
        break;
      }
      // 新增部分开始
      case OP_GREATER:  BINARY_OP(BOOL_VAL, >); break;
      case OP_LESS:     BINARY_OP(BOOL_VAL, <); break;
      // 新增部分结束
      case OP_ADD:      BINARY_OP(NUMBER_VAL, +); break;
```

> We already extended the BINARY_OP macro to handle operators that return non-numeric types. Now we get to use that. We pass in BOOL_VAL since the result value type is Boolean. Otherwise, it's no different from plus or minus.

我们已经扩展了BINARY_OP宏，来处理返回非数字类型的运算符。现在我们要用到它了。因为结果值类型是布尔型，所以我们传入BOOL_VAL。除此之外，这与加减运算没有区别。

> As always, the coda to today's aria is disassembling the new instructions.

与往常一样，今天的咏叹调的尾声是对新指令进行反汇编。

debug.c，在disassembleInstruction()方法中添加代码：

```
      case OP_FALSE:
        return simpleInstruction("OP_FALSE", offset);
      // 新增部分开始
      case OP_EQUAL:
        return simpleInstruction("OP_EQUAL", offset);
      case OP_GREATER:
        return simpleInstruction("OP_GREATER", offset);
      case OP_LESS:
        return simpleInstruction("OP_LESS", offset);
      // 新增部分结束
      case OP_ADD:
```

> With that, our numeric calculator has become something closer to a general expression evaluator. Fire up clox and type in:

这样一来，我们的数字计算器就变得更接近于一个通用的表达式求值器。启动clox并输入：

```
!(5 - 4 > 3 * 2 == !nil)
```

> OK, I'll admit that's maybe not the most *useful* expression, but we're making progress. We have one missing built-in type with its own literal form: strings. Those are much more complex because strings can vary in size. That tiny difference turns out to have implications so large that we give strings their very own chapter.

好吧，我承认这可能不是最*有用的*表达式，但我们正在取得进展。我们还缺少一种自带字面量形式的内置类型：字符串。它们要复杂得多，因为字符串的大小可以不同。这个微小的差异会产生巨大的影响，以至于我们给字符串单独开了一章。

^14: `a<=b`总是与`!(a>b)`相同吗？根据IEEE 754标准，当操作数为NaN时，所有的比较运算符都返回假。这意味着`NaN <= 1`是假的，`NaN > 1`也是假的。但我们的脱糖操作假定了后者是前者的非值。
在本书中，我们不必纠结于此，但这些细节在你的真正的语言实现中会很重要。 ^15: 有些语言支持"隐式转换"，如果某个类型的值可以转换为另一个类型，那么这两种类型的值就可以被认为是相等的。举例来说，在JavaScript中，数字0等同于字符串"0"。这种松散性导致JS增加了一个单独的"严格相等"运算符，`===`。
PHP认为字符串"1"和"01"是等价的，因为两者都可以转换成等价的数字，但是最根本的原因在于PHP是由Lovecraftian(译者注：洛夫克拉夫特，克苏鲁之父，可见作者对PHP怨念颇深)的邪神设计的，目的是摧毁人类心智。
大多数具有单独的整数和浮点数类型的动态类型语言认为，如果数值相同，则不同数字类型的值是相等的（所以说，1.0等于1），但即便是这种看似无害的便利，如果一不小心也会让人吃不消。

CHALLENGES

习题

1. > We could reduce our binary operators even further than we did here. Which other instructions can you eliminate, and how would the compiler cope with their absence?

   我们可以进一步简化二元操作符。还有哪些指令可以取消，编译器如何应对这些指令的缺失？

2. > Conversely, we can improve the speed of our bytecode VM by adding more specific instructions that correspond to higher-level operations. What instructions would you define to speed up the kind of user code we added support for in this chapter?

   相反，我们可以通过添加更多对应于高级操作的专用指令来提高字节码虚拟机的速度。你会定义什么指令来加速我们在本章中添加的那种用户代码？

19.字符串 Strings

> "Ah? A small aversion to menial labor?" The doctor cocked an eyebrow. "Understandable, but misplaced. One should treasure those hum-drum tasks that keep the body occupied but leave the mind and heart unfettered."
>
> ——Tad Williams, *The Dragonbone Chair*

"啊？对琐碎的劳动有点反感？"医生挑了挑眉毛，"可以理解，但这是错误的。一个人应该珍惜那些让身体忙碌，但让思想和心灵不受束缚的琐碎工作。"（泰德-威廉姆斯，《龙骨椅》）

> Our little VM can represent three types of values right now: numbers, Booleans, and `nil`. Those types have two important things in common: they're immutable and they're small. Numbers are the largest, and they still fit into two 64-bit words. That's a small enough price that we can afford to pay it for all values, even Booleans and nils which don't need that much space.

我们的小虚拟机现在可以表示三种类型的值：数字，布尔值和`nil`。这些类型有两个重要的共同点：它们是不可变的，它们很小。数字是最大的，而它仍可以被2个64比特的字容纳。这是一个足够小的代价，我们可以为所有值都支付这个代价，即使是不需要那么多空间的布尔值和nil。

> Strings, unfortunately, are not so petite. There's no maximum length for a string. Even if we were to artificially cap it at some contrived limit like 255 characters, that's still too much memory to spend on

> every single value.

不幸的是，字符串就没有这么小了。一个字符串没有最大的长度，即使我们人为地将其限制在255个字符^1，这对于每个单独的值来说仍然花费了太多的内存。

> We need a way to support values whose sizes vary, sometimes greatly. This is exactly what dynamic allocation on the heap is designed for. We can allocate as many bytes as we need. We get back a pointer that we'll use to keep track of the value as it flows through the VM.

我们需要一种方法来支持那些大小变化（有时变化很大）的值。这正是堆上动态分配的设计目的。我们可以根据需要分配任意多的字节。我们会得到一个指针，当值在虚拟机中流动时，我们会用该指针来跟踪它。

19.1 Values and Objects

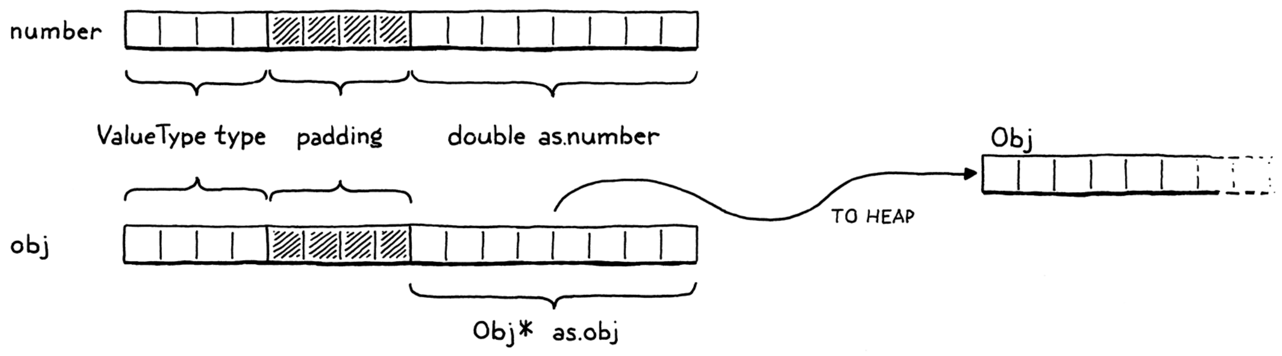
19.1 值与对象

> Using the heap for larger, variable-sized values and the stack for smaller, atomic ones leads to a two-level representation. Every Lox value that you can store in a variable or return from an expression will be a Value. For small, fixed-size types like numbers, the payload is stored directly inside the Value struct itself.

将堆用于较大的、可变大小的值，将栈用于较小的、原子性的值，这就导致了两级表示形式。每个可以存储在变量中或从表达式返回的Lox值都是一个Value。对于小的、固定大小的类型（如数字），有效载荷直接存储在Value结构本身。

> If the object is larger, its data lives on the heap. Then the Value's payload is a *pointer* to that blob of memory. We'll eventually have a handful of heap-allocated types in clox: strings, instances, functions, you get the idea. Each type has its own unique data, but there is also state they all share that our future garbage collector will use to manage their memory.

如果对象比较大，它的数据就驻留在堆中。那么Value的有效载荷就是指向那块内存的一个指针。我们最终会在clox中拥有一些堆分配的类型：字符串、实例、函数，你懂的。每个类型都有自己独特的数据，但它们也有共同的状态，我们未来的垃圾收集器会用这些状态来管理它们的内存。



> We'll call this common representation "Obj". Each Lox value whose state lives on the heap is an Obj. We can thus use a single new ValueType case to refer to all heap-allocated types.

我们将这个共同的表示形式称为"Obj"^2。每个状态位于堆上的Lox值都是一个Obj。因此，我们可以使用一个新的ValueType来指代所有堆分配的类型。

value.h，在枚举ValueType中添加代码：

```
  VAL_NUMBER,
  // 新增部分开始
  VAL_OBJ
  // 新增部分结束
} ValueType;
```

> When a Value's type is `VAL_OBJ`, the payload is a pointer to the heap memory, so we add another case to the union for that.

当Value的类型是`VAL_OBJ`时，有效载荷是一个指向堆内存的指针，因此我们在联合体中为其添加另一种情况。

value.h，在结构体Value中添加代码：

```
    double number;
    // 新增部分开始
    Obj* obj;
    // 新增部分结束
  } as;
```

> As we did with the other value types, we crank out a couple of helpful macros for working with Obj values.

正如我们对其它值类型所做的那样，我们提供了几个有用的宏来处理Obj值。

value.h，在结构体Value后添加代码：

```
#define IS_NUMBER(value)  ((value).type == VAL_NUMBER)
// 新增部分开始
#define IS_OBJ(value)     ((value).type == VAL_OBJ)
// 新增部分结束
#define AS_BOOL(value)    ((value).as.boolean)
```

> This evaluates to `true` if the given Value is an Obj. If so, we can use this:

如果给定的Value是一个Obj，则该值计算结果为`true`。如果这样，我们可以使用这个：

value.h，添加代码：

```
#define IS_OBJ(value)     ((value).type == VAL_OBJ)
// 新增部分开始
#define AS_OBJ(value)     ((value).as.obj)
// 新增部分结束
#define AS_BOOL(value)    ((value).as.boolean)
```

> It extracts the Obj pointer from the value. We can also go the other way.

它会从值中提取Obj指针。我们也可以反其道而行之。

value.h，添加代码：

```
#define NUMBER_VAL(value) ((Value){VAL_NUMBER, {.number = value}})
// 新增部分开始
#define OBJ_VAL(object)   ((Value){VAL_OBJ, {.obj = (Obj*)object}})
// 新增部分结束
typedef struct {
```

> This takes a bare Obj pointer and wraps it in a full Value.

该方法会接受一个Obj指针，并将其包装成一个完整的Value。

19.2 Struct Inheritance

19.2 结构体继承

> Every heap-allocated value is an Obj, but Objs are not all the same. For strings, we need the array of characters. When we get to instances, they will need their data fields. A function object will need its chunk of bytecode. How do we handle different payloads and sizes? We can't use another union like we did for Value since the sizes are all over the place.

每个堆分配的值都是一个Obj，但Obj并不都是一样的。对于字符串，我们需要字符数组。等我们有了实例，它们需要自己的数据字段。一个函数对象需要的是其字节码块。我们如何处理不同的有效载荷和大小？我们不能像Value那样使用另一个联合体，因为这些大小各不相同。

> Instead, we'll use another technique. It's been around for ages, to the point that the C specification carves out specific support for it, but I don't know that it has a canonical name. It's an example of *type punning*, but that term is too broad. In the absence of any better ideas, I'll call it **struct inheritance**, because it relies on structs and roughly follows how single-inheritance of state works in object-oriented languages.

相对地，我们会使用另一种技术。它已经存在了很久，以至于C语言规范为它提供了明确的支持，但我不知道它是否有一个正式的名字。这是一个类型双关的例子，但这个术语太宽泛了。鉴于没有更好的想法，我将其称为**结构体继承**，因为它依赖于结构体，并大致遵循了面向对象语言中状态的单继承工作方式。

> Like a tagged union, each Obj starts with a tag field that identifies what kind of object it is—string, instance, etc. Following that are the payload fields. Instead of a union with cases for each type, each type is its own separate struct. The tricky part is how to treat these structs uniformly since C has no concept of inheritance or polymorphism. I'll explain that soon, but first lets get the preliminary stuff out of the way.

与带标签的联合体一样，每个Obj开头都是一个标签字段，用于识别它是什么类型的对象——字符串、实例，等等。接下来是有效载荷字段。每种类型都有自己单独的结构，而不是各类型结构的联合体。棘手的部分是如何统一处理这些结构，因为C没有继承或多态的概念。我很快就会对此进行解释，但是首先让我们先弄清楚一些基本的东西。

> The name “Obj” itself refers to a struct that contains the state shared across all object types. It’s sort of like the “base class” for objects. Because of some cyclic dependencies between values and objects, we forward-declare it in the “value” module.

“Obj”这个名称本身指的是一个结构体，它包含所有对象类型共享的状态。它有点像对象的“基类”。由于值和对象之间存在一些循环依赖关系，我们在“value”模块中对其进行前置声明。

value.h，添加代码：

```
#include "common.h"
// 新增部分开始
typedef struct Obj Obj;
// 新增部分结束
typedef enum {
```

> And the actual definition is in a new module.

实际的定义是在一个新的模块中。

object.h，创建新文件：

```
#ifndef clox_object_h
#define clox_object_h

#include "common.h"
#include "value.h"

struct Obj {
  ObjType type;
};

#endif
```

> Right now, it contains only the type tag. Shortly, we’ll add some other bookkeeping information for memory management. The type enum is this:

现在，它只包含一个类型标记。不久之后，我们将为内存管理添加一些其它的簿记信息。类型枚举如下：

object.h，添加代码：

```
#include "value.h"
// 新增部分开始
typedef enum {
  OBJ_STRING,
} ObjType;
// 新增部分结束
struct Obj {
```

> Obviously, that will be more useful in later chapters after we add more heap-allocated types. Since we'll be accessing these tag types frequently, it's worth making a little macro that extracts the object type tag from a given Value.

显然，等我们在后面的章节中添加了更多的堆分配类型之后，这个枚举会更有用。因为我们会经常访问这些标记类型，所以有必要编写一个宏，从给定的Value中提取对象类型标签。

object.h，添加代码：

```
#include "value.h"
// 新增部分开始
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分结束
typedef enum {
```

> That's our foundation.

这是我们的基础。

> Now, let's build strings on top of it. The payload for strings is defined in a separate struct. Again, we need to forward-declare it.

现在，让我们在其上建立字符串。字符串的有效载荷定义在一个单独的结构体中。同样，我们需要对其进行前置声明。

value.h，添加代码：

```
typedef struct Obj Obj;
// 新增部分开始
typedef struct ObjString ObjString;
// 新增部分结束
typedef enum {
```

> The definition lives alongside Obj.

这个定义与Obj是并列的。

object.h，在结构体Obj后添加代码：

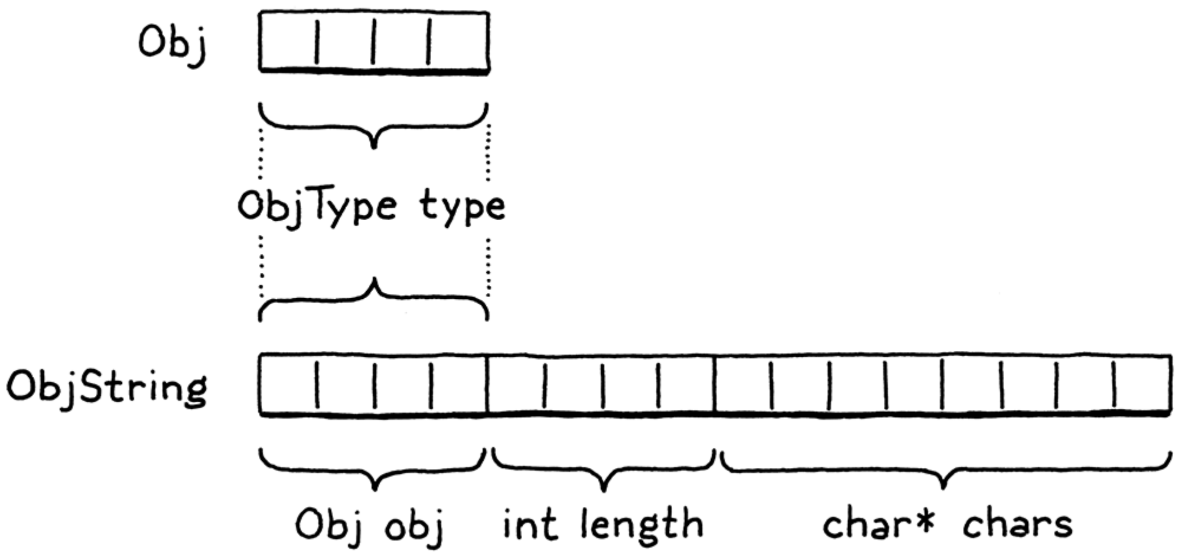
```
};
// 新增部分开始
struct ObjString {
  Obj obj;
  int length;
  char* chars;
};
// 新增部分结束
#endif
```

> A string object contains an array of characters. Those are stored in a separate, heap-allocated array so that we set aside only as much room as needed for each string. We also store the number of bytes in the array. This isn't strictly necessary but lets us tell how much memory is allocated for the string without walking the character array to find the null terminator.

字符串对象中包含一个字符数组。这些字符存储在一个单独的、由堆分配的数组中，这样我们就可以按需为每个字符串留出空间。我们还会保存数组中的字节数。这并不是严格必需的，但可以让我们迅速知道为字符串分配了多少内存，而不需要遍历字符数组寻找空结束符。

> Because ObjString is an Obj, it also needs the state all Objs share. It accomplishes that by having its first field be an Obj. C specifies that struct fields are arranged in memory in the order that they are declared. Also, when you nest structs, the inner struct's fields are expanded right in place. So the memory for Obj and for ObjString looks like this:

因为ObjString是一个Obj，它也需要所有Obj共有的状态。它通过将第一个字段置为Obj来实现这一点。C语言规定，结构体的字段在内存中是按照它们的声明顺序排列的。此外，当结构体嵌套时，内部结构体的字段会在适当的位置展开。所以Obj和ObjString的内存看起来是这样的：



> Note how the first bytes of ObjString exactly line up with Obj. This is not a coincidence—C mandates it. This is designed to enable a clever pattern: You can take a pointer to a struct and safely convert it to a pointer to its first field and back.

注意ObjString的第一个字节是如何与Obj精确对齐的。这并非巧合——是C语言强制要求的^3。这是为实现一个巧妙的模式而设计的：你可以接受一个指向结构体的指针，并安全地将其转换为指向其第一个字段的指针，反之亦可。

> Given an `ObjString*`, you can safely cast it to `Obj*` and then access the `type` field from it. Every ObjString "is" an Obj in the OOP sense of "is". When we later add other object types, each struct will have an Obj as its first field. Any code that wants to work with all objects can treat them as base `Obj*` and ignore any other fields that may happen to follow.

给定一个`ObjString*`，你可以安全地将其转换为`Obj*`，然后访问其中的`type`字段。每个ObjString"是"一个Obj，这里的"是"指OOP意义上的"是"。等我们稍后添加其它对象类型时，每个结构体都会有一个Obj作为其第一

个字段。任何代码若想要面向所有对象，都可以把它们当做基础的`Obj*`，并忽略后面可能出现的任何其它字段。

> You can go in the other direction too. Given an `Obj*`, you can “downcast” it to an `ObjString*`. Of course, you need to ensure that the `Obj*` pointer you have does point to the `obj` field of an actual ObjString. Otherwise, you are unsafely reinterpreting random bits of memory. To detect that such a cast is safe, we add another macro.

你也能反向操作。给定一个`Obj*`，你可以将其“向下转换”为一个`ObjString*`。当然，你需要确保你的`Obj*`指针确实指向一个实际的ObjString中的`obj`字段。否则，你就会不安全地重新解释内存中的随机比特位。为了检测这种类型转换是否安全，我们再添加另一个宏。

object.h，添加代码：

```
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分开始
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分结束
typedef enum {
```

> It takes a Value, not a raw `Obj*` because most code in the VM works with Values. It relies on this inline function:

它接受一个Value，而不是原始的`Obj*`，因为虚拟机中的大多数代码都使用Value。它依赖于这个内联函数：

object.h，在结构体ObjString后添加代码：

```
};
// 新增部分开始
static inline bool isObjType(Value value, ObjType type) {
  return IS_OBJ(value) && AS_OBJ(value)->type == type;
}
// 新增部分结束
#endif
```

> Pop quiz: Why not just put the body of this function right in the macro? What’s different about this one compared to the others? Right, it’s because the body uses `value` twice. A macro is expanded by inserting the argument *expression* every place the parameter name appears in the body. If a macro uses a parameter more than once, that expression gets evaluated multiple times.

突击测试：为什么不直接把这个函数体放在宏中？与其它函数相比，这个函数有什么不同？对，这是因为函数体使用了两次`value`。宏的展开方式是在主体中形参名称出现的每个地方插入实参*表达式*。如果一个宏中使用某个参数超过一次，则该表达式就会被求值多次。

> That’s bad if the expression has side effects. If we put the body of `isObjType()` into the macro definition and then you did, say,

> ```
> IS_STRING(POP())
> ```
>
> then it would pop two values off the stack! Using a function fixes that.

如果这个表达式有副作用，那就不好了。如果我们把`isObjType()`的主体放到宏的定义中，假设你这么使用

```
IS_STRING(POP())
```

那么它就会从堆栈中弹出两个值！使用函数可以解决这个问题。

> As long as we ensure that we set the type tag correctly whenever we create an Obj of some type, this macro will tell us when it's safe to cast a value to a specific object type. We can do that using these:

只要我们确保在创建某种类型的Obj时正确设置了类型标签，这个宏就会告诉我们何时将一个值转换为特定的对象类型是安全的。我们可以用下面这些函数来做转换：

object.h，添加代码：

```
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分开始
#define AS_STRING(value)       ((ObjString*)AS_OBJ(value))
#define AS_CSTRING(value)      (((ObjString*)AS_OBJ(value))->chars)
// 新增部分结束
typedef enum {
```

> These two macros take a Value that is expected to contain a pointer to a valid ObjString on the heap. The first one returns the `ObjString*` pointer. The second one steps through that to return the character array itself, since that's often what we'll end up needing.

这两个宏会接受一个Value，其中应当包含一个指向堆上的有效ObjString指针。第一个函数返回 `ObjString*` 指针。第二个函数更进一步返回了字符数组本身，因为这往往是我们最终需要的。

19.3 Strings

19.3 字符串

> OK, our VM can now represent string values. It's time to add strings to the language itself. As usual, we begin in the front end. The lexer already tokenizes string literals, so it's the parser's turn.

好了，我们的虚拟机现在可以表示字符串值了。现在是时候向语言本身添加字符串了。像往常一样，我们从前端开始。词法解析器已经将字符串字面量标识化了，所以现在轮到解析器了。

compiler.c，替换1行：

```
  [TOKEN_IDENTIFIER]    = {NULL,     NULL,   PREC_NONE},
  // 替换一行开始
```

```
  [TOKEN_STRING]        = {string,   NULL,   PREC_NONE},
  // 替换一行结束
  [TOKEN_NUMBER]        = {number,   NULL,   PREC_NONE},
```

> When the parser hits a string token, it calls this parse function:

当解析器遇到一个字符串标识时，会调用这个解析函数：

compiler.c，在number()方法后添加代码：

```
static void string() {
  emitConstant(OBJ_VAL(copyString(parser.previous.start + 1,
                                  parser.previous.length - 2)));
}
```

> This takes the string's characters directly from the lexeme. The `+ 1` and `- 2` parts trim the leading and trailing quotation marks. It then creates a string object, wraps it in a Value, and stuffs it into the constant table.

这里直接从词素中获取字符串的字符^4。`+1`和`-2`部分去除了开头和结尾的引号。然后，它创建了一个字符串对象，将其包装为一个Value，并塞入常量表中。

> To create the string, we use `copyString()`, which is declared in `object.h`.

为了创建字符串，我们使用了在`object.h`中声明的`copyString()`。

object.h，在结构体ObjString后添加代码：

```
};
// 新增部分开始
ObjString* copyString(const char* chars, int length);
// 新增部分结束
static inline bool isObjType(Value value, ObjType type) {
```

> The compiler module needs to include that.

编译器模块需要引入它。

compiler.h，添加代码：

```
#define clox_compiler_h
// 新增部分开始
#include "object.h"
// 新增部分结束
#include "vm.h"
```

> Our “object” module gets an implementation file where we define the new function.

我们的“object”模块有了一个实现文件，我们在其中定义新函数。

object.c，创建新文件：

```
#include <stdio.h>
#include <string.h>

#include "memory.h"
#include "object.h"
#include "value.h"
#include "vm.h"

ObjString* copyString(const char* chars, int length) {
  char* heapChars = ALLOCATE(char, length + 1);
  memcpy(heapChars, chars, length);
  heapChars[length] = '\0';
  return allocateString(heapChars, length);
}
```

> First, we allocate a new array on the heap, just big enough for the string’s characters and the trailing terminator, using this low-level macro that allocates an array with a given element type and count:

首先，我们在堆上分配一个新数组，其大小刚好可以容纳字符串中的字符和末尾的结束符，使用这个底层宏来分配一个具有给定元素类型和数量的数组：

memory.h，添加代码：

```
#include "common.h"
// 新增部分开始
#define ALLOCATE(type, count) \
    (type*)reallocate(NULL, 0, sizeof(type) * (count))
// 新增部分结束
#define GROW_CAPACITY(capacity) \
```

> Once we have the array, we copy over the characters from the lexeme and terminate it.

有了数组以后，就把词素中的字符复制过来并终止^5。

> You might wonder why the ObjString can’t just point back to the original characters in the source string. Some ObjStrings will be created dynamically at runtime as a result of string operations like concatenation. Those strings obviously need to dynamically allocate memory for the characters, which means the string needs to *free* that memory when it’s no longer needed.

你可能想知道为什么ObjString不能直接执行源字符串中的原始字符。由于连接等字符串操作，一些ObjString会在运行时被动态创建。这些字符串显然需要为字符动态分配内存，这也意味着该字符串不再需要这些内存时，要*释放*它们。

> If we had an ObjString for a string literal, and tried to free its character array that pointed into the original source code string, bad things would happen. So, for literals, we preemptively copy the characters over to the heap. This way, every ObjString reliably owns its character array and can free it.

如果我们有一个ObjString存储字符串字面量，并且试图释放其中指向原始的源代码字符串的字符数组，糟糕的事情就会发生。因此，对于字面量，我们预先将字符复制到堆中。这样一来，每个ObjString都能可靠地拥有自己的字符数组，并可以释放它。

> The real work of creating a string object happens in this function:

创建字符串对象的真正工作发生在这个函数中：

object.c，添加代码：

```
#include "vm.h"
// 新增部分开始
static ObjString* allocateString(char* chars, int length) {
  ObjString* string = ALLOCATE_OBJ(ObjString, OBJ_STRING);
  string->length = length;
  string->chars = chars;
  return string;
}
// 新增部分结束
```

> It creates a new ObjString on the heap and then initializes its fields. It's sort of like a constructor in an OOP language. As such, it first calls the "base class" constructor to initialize the Obj state, using a new macro.

它在堆上创建一个新的ObjString，然后初始化其字段。这有点像OOP语言中的构建函数。因此，它首先调用"基类"的构造函数来初始化Obj状态，使用了一个新的宏。

object.c，添加代码：

```
#include "vm.h"
// 新增部分开始
#define ALLOCATE_OBJ(type, objectType) \
    (type*)allocateObject(sizeof(type), objectType)
// 新增部分结束
static ObjString* allocateString(char* chars, int length) {
```

> Like the previous macro, this exists mainly to avoid the need to redundantly cast a `void*` back to the desired type. The actual functionality is here:

跟前面的宏一样，这个宏^6的存在主要是为了避免重复地将`void*`转换回期望的类型。实际的功能在这里：

object.c，添加代码：

```
#define ALLOCATE_OBJ(type, objectType) \
    (type*)allocateObject(sizeof(type), objectType)
// 新增部分开始
static Obj* allocateObject(size_t size, ObjType type) {
  Obj* object = (Obj*)reallocate(NULL, 0, size);
  object->type = type;
  return object;
}
// 新增部分结束
static ObjString* allocateString(char* chars, int length) {
```

> It allocates an object of the given size on the heap. Note that the size is *not* just the size of Obj itself. The caller passes in the number of bytes so that there is room for the extra payload fields needed by the specific object type being created.

它在堆上分配了一个给定大小的对象。注意，这个大小*不仅仅*是Obj本身的大小。调用者传入字节数，以便为被创建的对象类型留出额外的载荷字段所需的空间。

> Then it initializes the Obj state—right now, that's just the type tag. This function returns to `allocateString()`, which finishes initializing the ObjString fields. *Voilà*, we can compile and execute string literals.

然后它初始化Obj状态——现在这只是个类型标签。这个函数会返回到`allocateString()`，它来完成对ObjString字段的初始化。就是这样，我们可以编译和执行字符串字面量了。

19.4 Operations on Strings

19.4 字符串操作

> Our fancy strings are there, but they don't do much of anything yet. A good first step is to make the existing print code not barf on the new value type.

我们的花哨的字符串已经就位了，但是它们还没有发挥什么作用。一个好的第一步是使现有的打印代码不要排斥新的值类型。

value.c，在printValue()方法中添加代码：

```
    case VAL_NUMBER: printf("%g", AS_NUMBER(value)); break;
    // 新增部分开始
    case VAL_OBJ: printObject(value); break;
    // 新增部分结束
  }
```

> If the value is a heap-allocated object, it defers to a helper function over in the "object" module.

如果该值是一个堆分配的对象，它会调用"object"模块中的一个辅助函数。

object.h，在copyString()方法后添加代码：

```
ObjString* copyString(const char* chars, int length);
// 新增部分开始
void printObject(Value value);
// 新增部分结束
static inline bool isObjType(Value value, ObjType type) {
```

> The implementation looks like this:

对应的实现的这样的：

object.c，在copyString()方法后添加代码：

```
void printObject(Value value) {
  switch (OBJ_TYPE(value)) {
    case OBJ_STRING:
      printf("%s", AS_CSTRING(value));
      break;
  }
}
```

> We have only a single object type now, but this function will sprout additional switch cases in later chapters. For string objects, it simply prints the character array as a C string.

我们现在只有一个对象类型，但是这个函数在后续的章节中会出现更多case分支。对于字符串对象，只是简单地将字符数组作为C字符串打印出来^7。

> The equality operators also need to gracefully handle strings. Consider:

相等运算符也需要优雅地处理字符串。考虑一下：

```
"string" == "string"
```

> These are two separate string literals. The compiler will make two separate calls to `copyString()`, create two distinct ObjString objects and store them as two constants in the chunk. They are different objects in the heap. But our users (and thus we) expect strings to have value equality. The above expression should evaluate to `true`. That requires a little special support.

这是两个独立的字符串字面量。编译器会对`copyString()`进行两次单独的调用，创建两个不同的ObjString对象，并将它们作为两个常量存储在字节码块中。它们是堆中的不同对象。但是我们的用户（也就是我们）希望字符串的值是相等的。上面的表达式计算结果应该是`true`。这需要一点特殊的支持。

value.c，在valuesEqual()中添加代码：

```
    case VAL_NUMBER: return AS_NUMBER(a) == AS_NUMBER(b);
    // 新增部分开始
    case VAL_OBJ: {
      ObjString* aString = AS_STRING(a);
      ObjString* bString = AS_STRING(b);
      return aString->length == bString->length &&
          memcmp(aString->chars, bString->chars,
                 aString->length) == 0;
    }
    // 新增部分结束
    default:         return false; // Unreachable.
```

> If the two values are both strings, then they are equal if their character arrays contain the same characters, regardless of whether they are two separate objects or the exact same one. This does mean that string equality is slower than equality on other types since it has to walk the whole string. We'll revise that later, but this gives us the right semantics for now.

如果两个值都是字符串，那么当它们的字符数组中包含相同的字符时，它们就是相等的，不管它们是两个独立的对象还是完全相同的一个对象。这确实意味着字符串相等比其它类型的相等要慢，因为它必须遍历整个字符串。我们稍后会对此进行修改，但目前这为我们提供了正确的语义。

> Finally, in order to use `memcmp()` and the new stuff in the "object" module, we need a couple of includes. Here:

最后，为了使用`memcmp()`和"object"模块中的新内容，我们需要一些引入。这里：

value.c，添加代码：

```
#include <stdio.h>
// 新增部分开始
#include <string.h>
// 新增部分结束
#include "memory.h"
```

> And here:

还有这里：

value.c，添加代码：

```
#include <string.h>
// 新增部分开始
#include "object.h"
// 新增部分结束
#include "memory.h"
```

> ### 19.4.1 Concatenation

19.4.1 连接

> Full-grown languages provide lots of operations for working with strings—access to individual characters, the string's length, changing case, splitting, joining, searching, etc. When you implement your language, you'll likely want all that. But for this book, we keep things *very* minimal.

成熟的语言都提供了很多处理字符串的操作——访问单个字符、字符串长度、改变大小写、分割、连接、搜索等。当你实现自己的语言时，你可能会想要所有这些。但是在本书中，我们还是让事情保持简单。

> The only interesting operation we support on strings is `+`. If you use that operator on two string objects, it produces a new string that's a concatenation of the two operands. Since Lox is dynamically typed, we can't tell which behavior is needed at compile time because we don't know the types of the operands until runtime. Thus, the `OP_ADD` instruction dynamically inspects the operands and chooses the right operation.

我们对字符串支持的唯一有趣的操作是`+`。如果你在两个字符串对象上使用这个操作符，它会产生一个新的字符串，是两个操作数的连接。由于Lox是动态类型的，因此我们在编译时无法判断需要哪种行为，因为我们在运行时才知道操作数的类型。因此，`OP_ADD`指令会动态地检查操作数，并选择正确的操作。

vm.c，在run()方法中替换1行：

```
      case OP_LESS:     BINARY_OP(BOOL_VAL, <); break;
      // 替换部分开始
      case OP_ADD: {
        if (IS_STRING(peek(0)) && IS_STRING(peek(1))) {
          concatenate();
        } else if (IS_NUMBER(peek(0)) && IS_NUMBER(peek(1))) {
          double b = AS_NUMBER(pop());
          double a = AS_NUMBER(pop());
          push(NUMBER_VAL(a + b));
        } else {
          runtimeError(
              "Operands must be two numbers or two strings.");
          return INTERPRET_RUNTIME_ERROR;
        }
        break;
      }
      // 替换部分结束
      case OP_SUBTRACT: BINARY_OP(NUMBER_VAL, -); break;
```

> If both operands are strings, it concatenates. If they're both numbers, it adds them. Any other combination of operand types is a runtime error.

如果两个操作数都是字符串，则连接。如果都是数字，则相加。任何其它操作数类型的组合都是一个运行时错误^8。

> To concatenate strings, we define a new function.

为了连接字符串，我们定义一个新函数。

vm.c，在isFalsey()方法后添加代码：

```
static void concatenate() {
  ObjString* b = AS_STRING(pop());
  ObjString* a = AS_STRING(pop());

  int length = a->length + b->length;
  char* chars = ALLOCATE(char, length + 1);
  memcpy(chars, a->chars, a->length);
  memcpy(chars + a->length, b->chars, b->length);
  chars[length] = '\0';

  ObjString* result = takeString(chars, length);
  push(OBJ_VAL(result));
}
```

> It's pretty verbose, as C code that works with strings tends to be. First, we calculate the length of the result string based on the lengths of the operands. We allocate a character array for the result and then copy the two halves in. As always, we carefully ensure the string is terminated.

这是相当繁琐的，因为处理字符串的C语言代码往往是这样。首先，我们根据操作数的长度计算结果字符串的长度。我们为结果分配一个字符数组，然后将两个部分复制进去。与往常一样，我们要小心地确保这个字符串被终止了。

> In order to call `memcpy()`, the VM needs an include.

为了调用`memcpy()`，虚拟机需要引入头文件。

vm.c，添加代码：

```
#include <stdio.h>
// 新增部分开始
#include <string.h>
// 新增部分结束
#include "common.h"
```

> Finally, we produce an ObjString to contain those characters. This time we use a new function, `takeString()`.

最后，我们生成一个ObjString来包含这些字符。这次我们使用一个新函数`takeString()`。

object.h，在结构体ObjString后添加代码：

```
};
// 新增部分开始
ObjString* takeString(char* chars, int length);
// 新增部分结束
ObjString* copyString(const char* chars, int length);
```

> The implementation looks like this:

其实现如下：

object.c，在allocateString()方法后添加代码：

```
ObjString* takeString(char* chars, int length) {
  return allocateString(chars, length);
}
```

> The previous `copyString()` function assumes it *cannot* take ownership of the characters you pass in. Instead, it conservatively creates a copy of the characters on the heap that the ObjString can own. That's the right thing for string literals where the passed-in characters are in the middle of the source string.

前面的`copyString()`函数假定它*不能*拥有传入的字符的所有权。相对地，它保守地在堆上创建了一个ObjString可以拥有的字符的副本。对于传入的字符位于源字符串中间的字面量来说，这样做是正确的。

> But, for concatenation, we've already dynamically allocated a character array on the heap. Making another copy of that would be redundant (and would mean `concatenate()` has to remember to free its copy). Instead, this function claims ownership of the string you give it.

但是，对于连接，我们已经在堆上动态地分配了一个字符数组。再做一个副本是多余的（而且意味着`concatenate()`必须记得释放它的副本）。相反，这个函数要求拥有传入字符串的所有权。

> As usual, stitching this functionality together requires a couple of includes.

通常，将这个功能拼接在一起需要引入一些头文件。

vm.c，添加代码：

```
#include "debug.h"
// 新增部分开始
#include "object.h"
#include "memory.h"
// 新增部分结束
#include "vm.h"
```

19.5 Freeing Objects

19.5 释放对象

> Behold this innocuous-seeming expression:

看看这个看似无害的表达式：

```
"st" + "ri" + "ng"
```

> When the compiler chews through this, it allocates an ObjString for each of those three string literals and stores them in the chunk's constant table and generates this bytecode:

当编译器在处理这个表达式时，会为这三个字符串字面量分别分配一个ObjString，将它们存储到字节码块的常量表中^9，并生成这个字节码：

```
0000    OP_CONSTANT          0 "st"
0002    OP_CONSTANT          1 "ri"
0004    OP_ADD
0005    OP_CONSTANT          2 "ng"
0007    OP_ADD
0008    OP_RETURN
```

> The first two instructions push `"st"` and `"ri"` onto the stack. Then the `OP_ADD` pops those and concatenates them. That dynamically allocates a new `"stri"` string on the heap. The VM pushes that and then pushes the `"ng"` constant. The last `OP_ADD` pops `"stri"` and `"ng"`, concatenates them, and pushes the result: `"string"`. Great, that's what we expect.

前两条指令将`"st"`和`"ri"`压入栈中。然后`OP_ADD`将它们弹出并连接。这会在堆上动态分配一个新的`"stri"`字符串。虚拟机将它压入栈中，然后压入`"ng"`常量。最后一个`OP_ADD`会弹出`"stri"`和`"ng"`，将它们连接起来，并将结果`"string"`压入栈。很好，这就是我们所期望的。

> But, wait. What happened to that `"stri"` string? We dynamically allocated it, then the VM discarded it after concatenating it with `"ng"`. We popped it from the stack and no longer have a reference to it, but we never freed its memory. We've got ourselves a classic memory leak.

但是，请等一下。那个`"stri"`字符串怎么样了？我们动态分配了它，然后虚拟机在将其与`"ng"`连接后丢弃了它。我们把它从栈中弹出，不再持有对它的引用，但是我们从未释放它的内存。我们遇到了典型的内存泄露。

> Of course, it's perfectly fine for the *Lox program* to forget about intermediate strings and not worry about freeing them. Lox automatically manages memory on the user's behalf. The responsibility to manage memory doesn't *disappear*. Instead, it falls on our shoulders as VM implementers.

当然，Lox程序完全可以忘记中间的字符串，也不必担心释放它们。Lox代表用户自动管理内存。管理内存的责任并没有*消失*，相反，它落到了我们这些虚拟机实现者的肩上。

> The full solution is a garbage collector that reclaims unused memory while the program is running. We've got some other stuff to get in place before we're ready to tackle that project. Until then, we are living on borrowed time. The longer we wait to add the collector, the harder it is to do.

完整的解决方案是一个垃圾回收器，在程序运行时回收不使用的内存。在我们准备着手那个项目之前，还有一些其它的事情要做^10。在那之前，我们只是侥幸运行。我们等待添加收集器的时间越长，它就越难做。

> Today, we should at least do the bare minimum: avoid *leaking* memory by making sure the VM can still find every allocated object even if the Lox program itself no longer references them. There are many sophisticated techniques that advanced memory managers use to allocate and track memory for objects. We're going to take the simplest practical approach.

今天我们至少应该做到最基本的一点：确保虚拟机可以找到每一个分配的对象，即使Lox程序本身不再引用它们，从而避免*泄露*内存。高级内存管理程序会使用很多复杂的技术来分配和跟踪对象的内存。我们将采取最简单的实用方法。

> We'll create a linked list that stores every Obj. The VM can traverse that list to find every single object that has been allocated on the heap, whether or not the user's program or the VM's stack still has a reference to it.

我们会创建一个链表存储每个Obj。虚拟机可以遍历这个列表，找到在堆上分配的每一个对象，无论用户的程序或虚拟机的堆栈是否仍然有对它的引用。

> We could define a separate linked list node struct but then we'd have to allocate those too. Instead, we'll use an **intrusive list**—the Obj struct itself will be the linked list node. Each Obj gets a pointer to the next Obj in the chain.

我们可以定义一个单独的链表节点结构体，但那样我们也必须分配这些节点。相反，我们会使用**侵入式列表**——Obj结构体本身将作为链表节点。每个Obj都有一个指向链中下一个Obj的指针。

object.h，在结构体Obj中添加代码：

```
struct Obj {
  ObjType type;
  // 新增部分开始
  struct Obj* next;
  // 新增部分结束
};
```

> The VM stores a pointer to the head of the list.

VM存储一个指向表头的指针。

vm.h，在结构体VM中添加代码：

```
  Value* stackTop;
  // 新增部分开始
  Obj* objects;
  // 新增部分结束
} VM;
```

> When we first initialize the VM, there are no allocated objects.

当我们第一次初始化VM时，没有分配的对象。

vm.c，在initVM()方法中添加代码：

```
  resetStack();
  // 新增部分开始
  vm.objects = NULL;
  // 新增部分结束
}
```

> Every time we allocate an Obj, we insert it in the list.

每当我们分配一个Obj时，就将其插入到列表中。

object.c，在allocateObject()方法中添加代码：

```
  object->type = type;
  // 新增部分开始
  object->next = vm.objects;
  vm.objects = object;
  // 新增部分结束
  return object;
```

> Since this is a singly linked list, the easiest place to insert it is as the head. That way, we don’t need to also store a pointer to the tail and keep it updated.

由于这是一个单链表，所以最容易插入的地方是头部。这样，我们就不需要同时存储一个指向尾部的指针并保持对其更新。

> The “object” module is directly using the global `vm` variable from the “vm” module, so we need to expose that externally.

“object”模块直接使用了“vm”模块的`vm`变量，所以我们需要将该变量公开到外部。

vm.h，在枚举InterpretResult后添加代码：

```
} InterpretResult;
// 新增部分开始
extern VM vm;
// 新增部分结束
void initVM();
```

> Eventually, the garbage collector will free memory while the VM is still running. But, even then, there will usually be unused objects still lingering in memory when the user’s program completes. The VM should free those too.

最终，垃圾收集器会在虚拟机仍在运行时释放内存。但是，即便如此，当用户的程序完成时，通常仍会有未使用的对象驻留在内存中。VM也应该释放这些对象。

> There’s no sophisticated logic for that. Once the program is done, we can free *every* object. We can and should implement that now.

这方面没有什么复杂的逻辑。一旦程序完成，我们就可以释放*每个*对象。我们现在可以也应该实现它。

vm.c，在freeVM()方法中添加代码：

```
void freeVM() {
  // 新增部分开始
  freeObjects();
  // 新增部分结束
}
```

> That empty function we defined way back when finally does something! It calls this:

我们早先定义的空函数终于有了用武之地！它调用了这个方法：

memory.h，在reallocate()方法后添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize);
// 新增部分开始
void freeObjects();
// 新增部分结束
#endif
```

> Here’s how we free the objects:

下面是释放对象的方法：

memory.c，在reallocate()后添加代码：

```
void freeObjects() {
  Obj* object = vm.objects;
  while (object != NULL) {
    Obj* next = object->next;
    freeObject(object);
    object = next;
  }
}
```

> This is a CS 101 textbook implementation of walking a linked list and freeing its nodes. For each node, we call:

这是CS 101教科书中关于遍历链表并释放其节点的实现。对于每个节点，我们调用：

memory.c，在reallocate()方法后添加代码：

```
static void freeObject(Obj* object) {
  switch (object->type) {
    case OBJ_STRING: {
      ObjString* string = (ObjString*)object;
      FREE_ARRAY(char, string->chars, string->length + 1);
      FREE(ObjString, object);
```

```
        break;
      }
    }
  }
```

> We aren't only freeing the Obj itself. Since some object types also allocate other memory that they own, we also need a little type-specific code to handle each object type's special needs. Here, that means we free the character array and then free the ObjString. Those both use one last memory management macro.

我们不仅释放了Obj本身。因为有些对象类型还分配了它们所拥有的其它内存，我们还需要一些特定于类型的代码来处理每种对象类型的特殊需求。在这里，这意味着我们释放字符数组，然后释放ObjString。它们都使用了最后一个内存管理宏。

memory.h，添加代码：

```
    (type*)reallocate(NULL, 0, sizeof(type) * (count))
// 新增部分开始
#define FREE(type, pointer) reallocate(pointer, sizeof(type), 0)
// 新增部分结束
#define GROW_CAPACITY(capacity) \
```

> It's a tiny wrapper around `reallocate()` that "resizes" an allocation down to zero bytes.

这是围绕`reallocate()`的一个小包装^11，可以将分配的内存“调整”为零字节。

Using `reallocate()` to free memory might seem pointless. Why not just call `free()`? Later, this will help the VM track how much memory is still being used. If all allocation and freeing goes through `reallocate()`, it's easy to keep a running count of the number of bytes of allocated memory.

> As usual, we need an include to wire everything together.

像往常一样，我们需要一个include将所有东西连接起来

memory.h，添加代码：

```
#include "common.h"
// 新增部分开始
#include "object.h"
// 新增部分结束
#define ALLOCATE(type, count) \
```

> Then in the implementation file:

然后是实现文件：

memory.c，添加代码：

```
#include "memory.h"
// 新增部分开始
#include "vm.h"
// 新增部分结束
void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
```

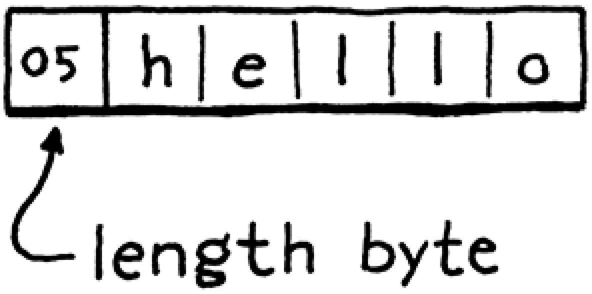
> With this, our VM no longer leaks memory. Like a good C program, it cleans up its mess before exiting. But it doesn't free any objects while the VM is running. Later, when it's possible to write longer-running Lox programs, the VM will eat more and more memory as it goes, not relinquishing a single byte until the entire program is done.

这样一来，我们的虚拟机就不会再泄露内存了。像一个好的C程序一样，它会在退出之前进行清理。但在虚拟机运行时，它不会释放任何对象。稍后，当可以编写长时间运行的Lox程序时，虚拟机在运行过程中会消耗越来越多的内存，在整个程序完成之前不会释放任何一个字节。

We won't address that until we've added a real garbage collector, but this is a big step. We now have the infrastructure to support a variety of different kinds of dynamically allocated objects. And we've used that to add strings to clox, one of the most used types in most programming languages. Strings in turn enable us to build another fundamental data type, especially in dynamic languages: the venerable hash table. But that's for the next chapter . . .

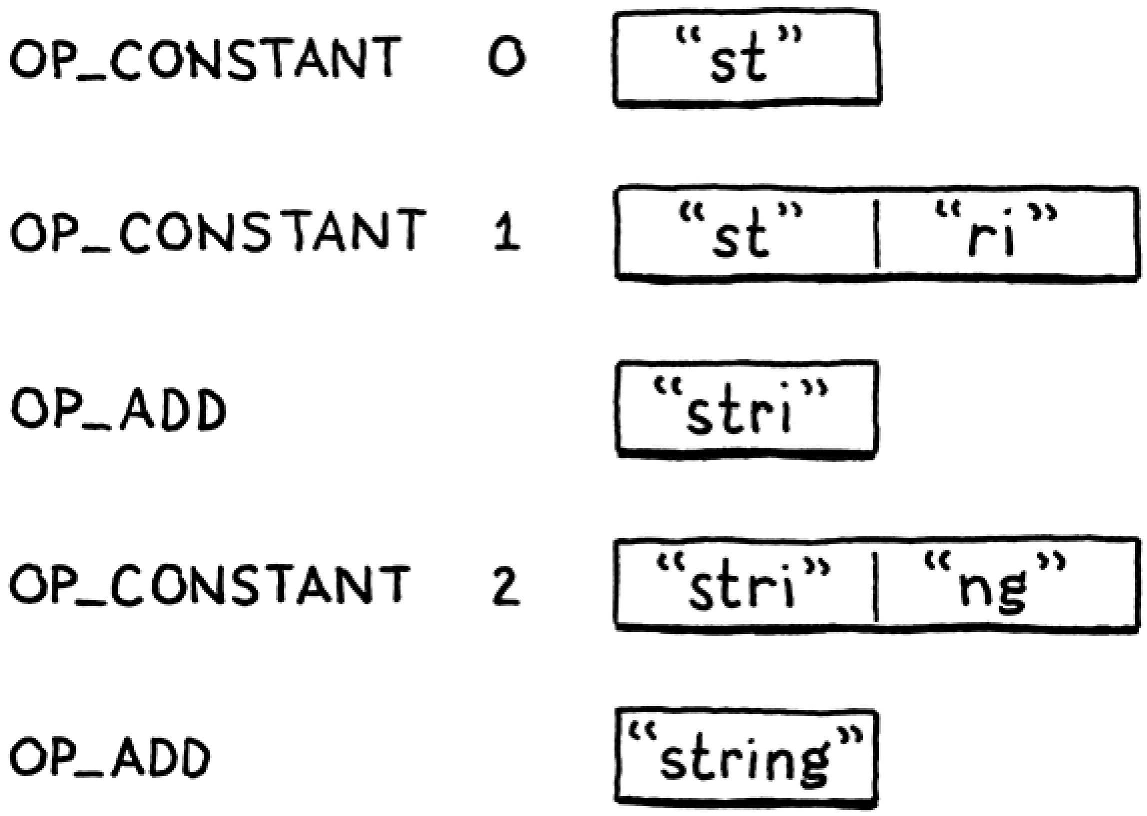
在添加真正的垃圾收集器之前，我们不会解决这个问题，但这是一个很大的进步。我们现在拥有了支持各种不同类型的动态分配对象的基础设施。我们利用这一点在clox中加入了字符串，这是大多数编程语言中最常用的类型之一。字符串反过来又使我们能够构建另一种基本的数据类型，尤其是在动态语言中：古老的哈希表。但这是下一章的内容了......

^1: UCSD Pascal，Pascal最早的实现之一，就有这个确切的限制。Pascal字符串开头是长度值，而不是像C语言那样用一个终止的空字符表示字符串的结束。因为UCSD只使用一个字节来存储长度，所以字符串不能超过255



个字符。 ^2: 当然，"Obj"是"对象（object）"的简称。 ^3: 语言规范中的关键部分是：
$ 6.7.2.1 13
在一个结构体对象中，非位域成员和位域所在的单元的地址按照它们被声明的顺序递增。一个指向结构对象的指针，经过适当转换后，指向其第一个成员（如果该成员是一个位域，则指向其所在的单元），反之亦然。在结构对象中可以有未命名的填充，但不允许在其开头。 ^4: 如果Lox支持像`\n`这样的字符串转义序列，我们会在这里对其进行转换。既然不支持，我们就可以原封不动地接受这些字符。 ^5: 我们需要自己终止字符串，因为词素指向整个源字符串中的一个字符范围，并且没有终止符。
由于ObjString明确存储了长度，我们*可以*让字符数组不终止，但是在结尾处添加一个终止符只花费一个字节，并且可以让我们将字符数组传递给期望带终止符的C标准库函数。 ^6: 我承认这一章涉及了大量的辅助函数和宏。我试图让代码保持良好的分解，但这导致了一些分散的小函数。等我们以后重用它们时，将会得到回报。
^7: 我说过，终止字符串会有用的。 ^8: 这比大多数语言都要保守。在其它语言中，如果一个操作数是字符串，另一个操作数可以是任何类型，在连接这两个操作数之前会隐式地转换为字符串。

我认为这是一个很好的特性，但是需要为每种类型编写冗长的“转换为字符串”的代码，所以我在Lox中没有支持它。 ^9: 下面是每条指令执行后的堆栈：



^10: 我见过很多人在实现看语言的大部分内容之后，才试图开始实现GC。对于在开发语言时通常会运行的那种玩具程序，实际上不会在程序结束之前耗尽内存，所以在需要GC之前，你可以开发出很多的特性。
但是，这低估了以后添加垃圾收集器的难度。收集器必须确保它能够找到每一点仍在使用的内存，这样它就不会收集活跃数据。一个语言的实现可以在数百个地方存储对某个对象的引用。如果你不能找到所有这些地方，你就会遇到噩梦般的漏洞。
我曾见过一些语言实现因为后来的GC太困难而夭折。如果你的语言需要GC，请尽快实现它。它是涉及整个代码库的横切关注点。 ^11: 使用`reallocate()`来释放内存似乎毫无意义。为什么不直接调用`free()`呢？稍后，这将帮助虚拟机跟踪仍在使用的内存数量。如果所有的分配和释放都通过`reallocate()`进行，那么就很容易对已分配的内存字节数进行记录。

习题

1. > Each string requires two separate dynamic allocations—one for the ObjString and a second for the character array. Accessing the characters from a value requires two pointer indirections, which can be bad for performance. A more efficient solution relies on a technique called **flexible array members**. Use that to store the ObjString and its character array in a single contiguous allocation.

   每个字符串都需要两次单独的动态分配——一个是ObjString，另一个是字符数组。从一个值中访问字符需要两个指针间接访问，这对性能是不利的。一个更有效的解决方案是依靠一种名为**灵活数组成员**的技术。用该方法将ObjString和它的字符数据存储在一个连续分配的内存中。

2. > When we create the ObjString for each string literal, we copy the characters onto the heap. That way, when the string is later freed, we know it is safe to free the characters too.

> This is a simpler approach but wastes some memory, which might be a problem on very constrained devices. Instead, we could keep track of which ObjStrings own their character array and which are "constant strings" that just point back to the original source string or some other non-freeable location. Add support for this.

当我们为每个字符串字面量创建ObjString时，会将字符复制到堆中。这样，当字符串后来被释放时，我们知道释放这些字符也是安全的。

这是一个简单但是会浪费一下内存的方法，这在非常受限的设备上可能是一个问题。相反，我们可以追踪哪些ObjString拥有自己的字符数组，哪些是"常量字符串"，只是指向原始的源字符串或其它不可释放的位置。添加对此的支持。

3. > If Lox was your language, what would you have it do when a user tries to use `+` with one string operand and the other some other type? Justify your choice. What do other languages do?

   如果Lox是你的语言，当用户试图用一个字符串操作数使用`+`，而另一个操作数是其它类型时，你会让它做什么？证明你的选择是正确的，其它的语言是怎么做的？

DESIGN NOTE: STRING ENCODING

设计笔记：字符串编码

> In this book, I try not to shy away from the gnarly problems you'll run into in a real language implementation. We might not always use the most *sophisticated* solution—it's an intro book after all—but I don't think it's honest to pretend the problem doesn't exist at all. However, I did skirt around one really nasty conundrum: deciding how to represent strings.
>
> There are two facets to a string encoding:
>
> - **What is a single "character" in a string?** How many different values are there and what do they represent? The first widely adopted standard answer to this was ASCII. It gave you 127 different character values and specified what they were. It was great . . . if you only ever cared about English. While it has weird, mostly forgotten characters like "record separator" and "synchronous idle", it doesn't have a single umlaut, acute, or grave. It can't represent "jalapeño", "naïve", "Gruyère", or "Mötley Crüe".
>
>   Next came Unicode. Initially, it supported 16,384 different characters (**code points**), which fit nicely in 16 bits with a couple of bits to spare. Later that grew and grew, and now there are well over 100,000 different code points including such vital instruments of human communication as 💩 (Unicode Character 'PILE OF POO', `U+1F4A9`).
>
>   Even that long list of code points is not enough to represent each possible visible glyph a language might support. To handle that, Unicode also has **combining characters** that modify a preceding code point. For example, "a" followed by the combining character "¨" gives you "ä". (To make things more confusing Unicode *also* has a single code point that looks like "ä".)
>
>   If a user accesses the fourth "character" in "naïve", do they expect to get back "v" or "¨"? The former means they are thinking of each code point and its combining character as a single unit

—what Unicode calls an **extended grapheme cluster**—the latter means they are thinking in individual code points. Which do your users expect?

- **How is a single unit represented in memory?** Most systems using ASCII gave a single byte to each character and left the high bit unused. Unicode has a handful of common encodings. UTF-16 packs most code points into 16 bits. That was great when every code point fit in that size. When that overflowed, they added *surrogate pairs* that use multiple 16-bit code units to represent a single code point. UTF-32 is the next evolution of UTF-16—it gives a full 32 bits to each and every code point.

  UTF-8 is more complex than either of those. It uses a variable number of bytes to encode a code point. Lower-valued code points fit in fewer bytes. Since each character may occupy a different number of bytes, you can't directly index into the string to find a specific code point. If you want, say, the 10th code point, you don't know how many bytes into the string that is without walking and decoding all of the preceding ones.

Choosing a character representation and encoding involves fundamental trade-offs. Like many things in engineering, there's no perfect solution:

- ASCII is memory efficient and fast, but it kicks non-Latin languages to the side.
- UTF-32 is fast and supports the whole Unicode range, but wastes a lot of memory given that most code points do tend to be in the lower range of values, where a full 32 bits aren't needed.
- UTF-8 is memory efficient and supports the whole Unicode range, but its variable-length encoding makes it slow to access arbitrary code points.
- UTF-16 is worse than all of them—an ugly consequence of Unicode outgrowing its earlier 16-bit range. It's less memory efficient than UTF-8 but is still a variable-length encoding thanks to surrogate pairs. Avoid it if you can. Alas, if your language needs to run on or interoperate with the browser, the JVM, or the CLR, you might be stuck with it, since those all use UTF-16 for their strings and you don't want to have to convert every time you pass a string to the underlying system.

One option is to take the maximal approach and do the "rightest" thing. Support all the Unicode code points. Internally, select an encoding for each string based on its contents—use ASCII if every code point fits in a byte, UTF-16 if there are no surrogate pairs, etc. Provide APIs to let users iterate over both code points and extended grapheme clusters.

This covers all your bases but is really complex. It's a lot to implement, debug, and optimize. When serializing strings or interoperating with other systems, you have to deal with all of the encodings. Users need to understand the two indexing APIs and know which to use when. This is the approach that newer, big languages tend to take—like Raku and Swift.

A simpler compromise is to always encode using UTF-8 and only expose an API that works with code points. For users that want to work with grapheme clusters, let them use a third-party library for that. This is less Latin-centric than ASCII but not much more complex. You lose fast direct indexing by code point, but you can usually live without that or afford to make it *O(n)* instead of *O(1)*.

If I were designing a big workhorse language for people writing large applications, I'd probably go with the maximal approach. For my little embedded scripting language Wren, I went with UTF-8 and code points.

在本书中，我尽量不回避你在真正的语言实现中会遇到的棘手问题。我们也许并不总是使用最复杂的解决方案——毕竟这只是一本入门书——但我认为假装问题根本不存在是不诚实的。但是，我们确实绕过了一个非常棘手的难题：决定如何表示字符串。

字符串编码有两个方面：

- 什么是字符串中的一个“字符”？有多少个不同的值，它们代表什么？第一个被广泛采用的标准答案是ASCII。它给出了127个不同的字符值，并指明了它们是什么。这太棒了……如果你只关心英语的话。虽然它包含有像“记录分隔符”和“同步空闲”这样奇怪的、几乎被遗忘的字符，但它没有元音变音、锐音或钝音。它无法表示 “jalapeño”， “naïve”， “Gruyère”或 “Mötley Crüe”。

  接下来是Unicode。最初，它支持16384个不同的字符（码点），这非常适合在16比特位中使用，还有几位是多余的。后来，这个数字不断增加，现在已经有了超过100,000个不同的码点，包括诸如💩（Unicode字符 "PILE OF POO"， U+1F4A9）等人类交流的重要工具。

  即使是这么长的码点列表，也不足以表示一种语言可能支持的每个可见字形。为了处理这个问题，Unicode还有一些组合字符，可以修改前面的码点。例如，“a”后面跟组合字符“¨”，就可以得到“ä”。（为了使事情更混乱，Unicode也有一个看起来像“ä”的单一码点）

  如果用户访问“naïve”中的第四个“字符”，他们希望得到的是“v”还是“¨”？前者意味着他们把每个码点及其组合符看着一个单元——Unicode称之为扩展的字母簇，后者意味着它们以单独的码点来思考。你的用户期望的是哪一种？

- 单一单元在内存中是如何表示的？大多数使用ASCII的系统给每个字符分配一个字节，高位不使用。Unicode有几种常见的编码方式。UTF-16将大多数码点打包成16比特。当每个码点都在这个范围内时，是很好的。当码点溢出时，它们增加了*代理对*，使用多个16比特码来表示一个码点。UTF-32是UTF-16的进一步演变，它为每个码点都提供了完整的32比特。

  UTF-8比这两个都要复杂。它使用可变数量的字节来对码点编码。低值的码点适合于较少的字节。因为每个字符可能占用不同数量的字节，你不能直接在字符串中索引到特定的码点。如果你想要访问，比如说，第10个码点，在不遍历和解码前面所有码点的情况下，你不知道它在字符串中占用多少个字节。

选择字符表示形式和编码涉及到基本的权衡。就像工程领域的许多事情一样，没有完美的解决方案：

【关于这个问题有多难的一个例子就是Python 。从Python 2到3的漫长转变之所以令人痛苦，主要是因为它围绕字符串编码的变化】

- ASCII内存效率高，速度快，但它把非拉丁语系的语言踢到了一边。
- UTF-32速度快，并且支持整个Unicode范围，但考虑到大多数码点往往都位于较低的值范围内，不需要完整的32比特，所以浪费了大量的内存。
- UTF-8的内存效率高，支持整个Unicode范围，但是它的可变长度编码使得在访问任意码点时速度很慢。
- UTF-16比所有这些都糟糕——这是Unicode超出其早期16比特范围的丑陋后果。它的内存效率低于UTF-8，但由于代理对的存在，它仍然是一种可变长度的编码。尽量避免使用它。唉，如果你的语言需要在浏览器、JVM或CLR上运行或与之交互，你也许就只能用它了，因为这些系统的字符串都使用UTF-16，而你并不想每次向底层系统传递字符串时都进行转换。

一种选择是采取最大限度的方法，做“最正确”的事情。支持所有的Unicode码点。在内部，根据每个字符串的内容选择编码——如果每个码点都在一个字节内，就使用ASCII；如果没有代理对，就使用UTF-16，等等。提供API，让用户对码点和扩展字母簇进行遍历。

这涵盖了所有的基础，但真的很复杂。需要实现、调试和优化的东西很多。当序列化字符串或与其它系统进行交互时，你必须处理所有的编码。用户需要理解这两种索引API，并知道何时使用哪一种。这是较新的大型语言倾向于采取的方法，比如Raku和Swift。

一种更简单的折衷办法是始终使用UTF-8编码，并且只暴露与码点相关的API。对于想要处理字母簇的用户，让他们使用第三方库来处理。这不像ASCII那样以拉丁语为中心，但也没有多复杂，虽然会失去通过码点快速直接索引的能力，但通常没有索引也可以，或者可以将索引改为O(n)而不是O(1)。

如果我要为编写大型应用程序的人设计一种大型工作语言，我可能会采用最大的方法。至于我的小型嵌入式脚本语言Wren，我采用了UTF-8和码点。

20.哈希表 Hash Tables

> Hash, x. There is no definition for this word—nobody knows what hash is.
>
> ——Ambrose Bierce, *The Unabridged Devil's Dictionary*

哈希，未知。这个词没有定义——没人知道哈希是什么。（安布罗斯·比尔斯，《无删减魔鬼词典》）

> Before we can add variables to our burgeoning virtual machine, we need some way to look up a value given a variable's name. Later, when we add classes, we'll also need a way to store fields on instances. The perfect data structure for these problems and others is a hash table.

在向这个发展迅速的虚拟机中添加变量之前，我们需要某种方法来根据给定的变量名称查询变量值。稍后，等到我们添加类时，也需要某种方法来存储实例中的字段。对于这些问题和其它问题，完美的数据结构就是哈希表。

> You probably already know what a hash table is, even if you don't know it by that name. If you're a Java programmer, you call them "HashMaps". C# and Python users call them "dictionaries". In C++, it's an "unordered map". "Objects" in JavaScript and "tables" in Lua are hash tables under the hood, which is what gives them their flexibility.

你可能已经知道什么是哈希表了，即使你不知道它的名字。如果你是Java程序员，你把它们称为"HashMap"。C#和Python用户则称它们为"字典"。在C++中，它是"无序映射"。JavaScript中的"对象"和Lua中的"表"本质上都是哈希表，这赋予了它们灵活性。

> A hash table, whatever your language calls it, associates a set of **keys** with a set of **values**. Each key/value pair is an **entry** in the table. Given a key, you can look up its corresponding value. You can add new key/value pairs and remove entries by key. If you add a new value for an existing key, it replaces the previous entry.

哈希表（无论你的语言中怎么称呼它）是将一组**键**和一组**值**关联起来。每个键/值对是表中的一个**条目**。给定一个键，可以查找它对应的值。你可以按键添加新的键/值对或删除条目。如果你为已有的键添加新值，它就会替换原先的条目。

> Hash tables appear in so many languages because they are incredibly powerful. Much of this power comes from one metric: given a key, a hash table returns the corresponding value in constant time, *regardless of how many keys are in the hash table*.

哈希表之所以出现在这么多的语言中，是因为它们非常强大。这种强大的能力主要来自于一个指标：给定一个键，哈希表会在常量时间^1内返回对应的值，*不管哈希表中有多少键*。

> That's pretty remarkable when you think about it. Imagine you've got a big stack of business cards and I ask you to find a certain person. The bigger the pile is, the longer it will take. Even if the pile is nicely sorted and you've got the manual dexterity to do a binary search by hand, you're still talking *O(log n)*. But with a hash table, it takes the same time to find that business card when the stack has ten cards as when it has a million.

仔细想想，这是非常了不起的。想象一下，你有一大堆名片，我让你去找出某个人。这堆名片越大，花的时间就越长。即使这堆名片被很好地排序，而且你有足够的能力来手动进行二分查找，你的复杂度仍然是O(log n)。但是对于哈希表来说，无论这摞名片有10张还是100万张，你找到那张特定名片所需的时间都是一样的。

20.1 An Array of Buckets

20.1 桶数组

> A complete, fast hash table has a couple of moving parts. I'll introduce them one at a time by working through a couple of toy problems and their solutions. Eventually, we'll build up to a data structure that can associate any set of names with their values.

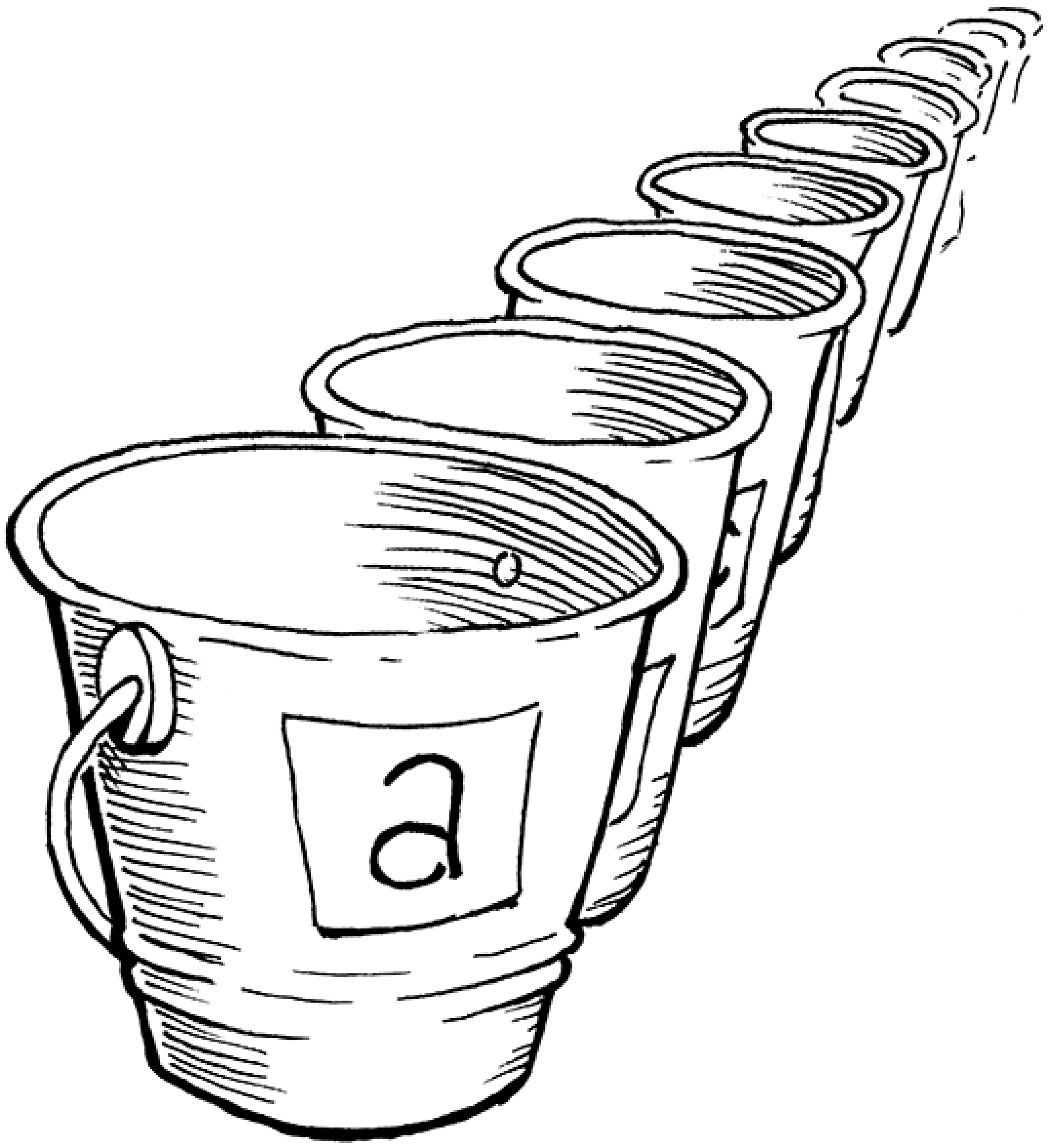
一个完整、快速的哈希表需要一些组件。我会通过几个小问题及其解决方案来逐一介绍它们。最终，我们将构建一个数据结构，可以将任何一组名称和它们的值关联起来。

> For now, imagine if Lox was a *lot* more restricted in variable names. What if a variable's name could only be a single lowercase letter. How could we very efficiently represent a set of variable names and their values?

现在，假定Lox在变量名称上有*更多*的限制。如果一个变量的名称只能是一个小写字母该怎么办^2？我们如何高效地表示一组变量名和它们的值？

> With only 26 possible variables (27 if you consider underscore a "letter", I guess), the answer is easy. Declare a fixed-size array with 26 elements. We'll follow tradition and call each element a **bucket**. Each represents a variable with `a` starting at index zero. If there's a value in the array at some letter's index, then that key is present with that value. Otherwise, the bucket is empty and that key/value pair isn't in the data structure.

由于只有26个可能的变量（如果你认为下划线是一个"字母"，我猜是27个），答案很简单。声明一个具有26个元素的固定大小的数组。我们遵循传统，将每个元素称为一个**桶（bucket）**。每个元素代表一个变量，`a`从索引下标0开始。如果数组中某个字母对应的索引位置有值，那么这个键就与该值相对应。否则的话，桶是空的，该键/值对在数据结构中不存在。



> Memory usage is great—just a single, reasonably sized array. There's some waste from the empty buckets, but it's not huge. There's no overhead for node pointers, padding, or other stuff you'd get with something like a linked list or tree.

这个方案的内存占用情况很好——只是一个大小合理的数组。空桶会有一些浪费，但不是很大。没有节点指针、填充以及其它类似于链表或树的开销。

> Performance is even better. Given a variable name—its character—you can subtract the ASCII value of `a` and use the result to index directly into the array. Then you can either look up the existing value or store a new value directly into that slot. It doesn't get much faster than that.

性能就更好了。给定一个变量名——它的字符——你可以减去`a`的ASCII值，并使用结果直接在数组中进行索引。然后，你可以查找已有的值或将新值存储到对应的槽中。没有比这更快的了。

> This is sort of our Platonic ideal data structure. Lightning fast, dead simple, and compact in memory. As we add support for more complex keys, we'll have to make some concessions, but this is what we're aiming for. Even once you add in hash functions, dynamic resizing, and collision resolution, this is still the core of every hash table out there—a contiguous array of buckets that you index directly into.

这是一种柏拉图式的理想数据结构。快如闪电，非常简单，而且内存紧凑。当我们进一步支持更复杂的键时，就必须作出一些让步，但这仍是我们的目标所在。即使加入了哈希函数、动态调整大小和冲突解决，这仍然是每个哈希表的核心——一个可以直接索引到的连续桶数组。

> ### 20.1.1 Load factor and wrapped keys

20.1.1 负载因子和封装键

> Confining Lox to single-letter variables would make our job as implementers easier, but it’s probably no fun programming in a language that gives you only 26 storage locations. What if we loosened it a little and allowed variables up to eight characters long?

将Lox限制为单字母变量，会使我们作为实现者的工作更容易，但在一种只提供26个存储位置的语言中编程可能没有什么乐趣。如果我们稍微放宽限制，允许变量的长度到8个字符呢^3？

> That’s small enough that we can pack all eight characters into a 64-bit integer and easily turn the string into a number. We can then use it as an array index. Or, at least, we could if we could somehow allocate a 295,148 *petabyte* array. Memory’s gotten cheaper over time, but not quite *that* cheap. Even if we could make an array that big, it would be heinously wasteful. Almost every bucket would be empty unless users started writing way bigger Lox programs than we’ve anticipated.

它足够小，我们可以将所有8个字符打包成一个64比特的整数，轻松地将字符串变成一个数字。然后我们可以把它作为数组索引。至少，如果我们能够以某种方式分配295,148 PB的数组，也是可以的。随着时间的推移，内存越来越便宜了，但还没那么便宜。即便我们可以创建这么大的数组，也会造成严重的浪费。除非用户会编写比我们的预期大得多的Lox程序，否则几乎每个桶都是空的。

> Even though our variable keys cover the full 64-bit numeric range, we clearly don’t need an array that large. Instead, we allocate an array with more than enough capacity for the entries we need, but not unreasonably large. We map the full 64-bit keys down to that smaller range by taking the value modulo the size of the array. Doing that essentially folds the larger numeric range onto itself until it fits the smaller range of array elements.

尽管我们的变量键覆盖了整个64位数字范围，但我们显然不需要那么大的数组。相反地，我们会分配一个数组，它的容量足以容纳我们需要的条目，但又不会大得不合理。通过对数组的大小进行取模，我们将完整的64位键值映射到较小的范围。这样做本质上是将较大的数值范围不断折叠，直到适合较小的数组元素范围。

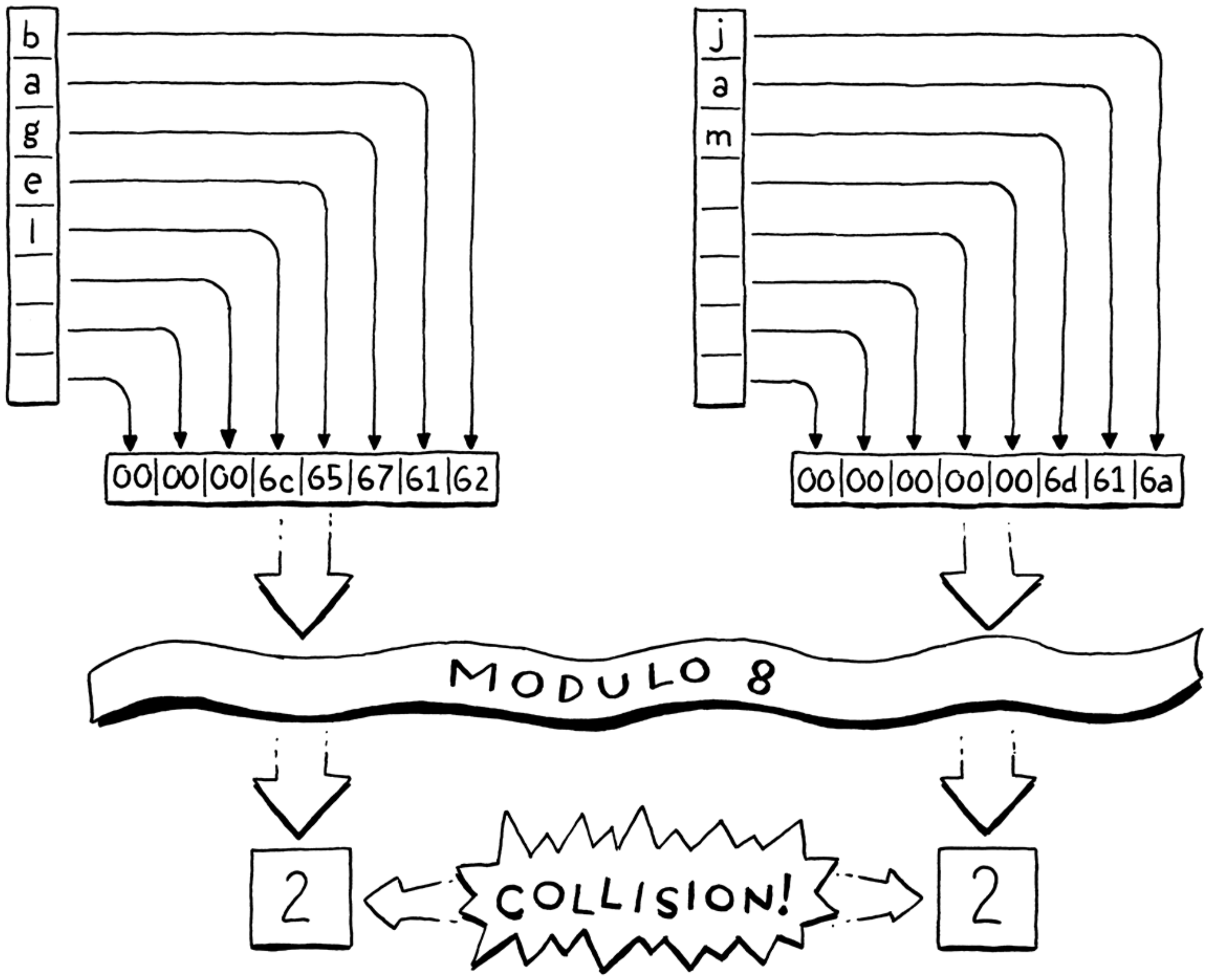
> For example, say we want to store “bagel”. We allocate an array with eight elements, plenty enough to store it and more later. We treat the key string as a 64-bit integer. On a little-endian machine like Intel, packing those characters into a 64-bit word puts the first letter, “b” (ASCII value 98), in the least-significant byte. We take that integer modulo the array size (8) to fit it in the bounds and get a bucket index, 2. Then we store the value there as usual.

例如，假设我们想要存储“bagel”。我们分配一个有8个元素的数组^4，足够存储它，之后还可以存储更多。我们把键字符串当作一个64位整数。在Intel这样的小端机器上，将这些字符打包成一个64位的字时，会将第一个字母“b”（ASCII值 98）放在最低的有效字节中。我们将这个整数与数组大小（8）取模以适应边界，并得到桶索引2。然后我们像往常一样，将值存储在那里。

> Using the array size as a modulus lets us map the key’s numeric range down to fit an array of any size. We can thus control the number of buckets independently of the key range. That solves our waste problem, but introduces a new one. Any two variables whose key number has the same remainder

> when divided by the array size will end up in the same bucket. Keys can **collide**. For example, if we try to add "jam", it also ends up in bucket 2.

使用数组的大小作为模数，可以让我们将键的数值范围向下适配到任意大小的数组。因此，我们可以独立于键的范围来控制桶的数量。这就解决了我们的浪费问题，但是也引入了一个新的问题。任意两个变量，如果它们的键值除以数组大小时有相同的余数，最后都会被放在同一个桶中。键会发生**冲突**。举例来说，如果我们尝试添加"jam"，它也会出现在2号桶中。



> We have some control over this by tuning the array size. The bigger the array, the fewer the indexes that get mapped to the same bucket and the fewer the collisions that are likely to occur. Hash table implementers track this collision likelihood by measuring the table's **load factor**. It's defined as the number of entries divided by the number of buckets. So a hash table with five entries and an array of 16 elements has a load factor of 0.3125. The higher the load factor, the greater the chance of collisions.

我们可以通过调整数组的大小来控制这个问题。数组越大，映射到同一个桶的索引就越少，可能发生的冲突也就越少。哈希表实现者评估这种冲突的可能性的方式就是计算表的**负载因子**。它的定义是条目的数量除以桶的数量。因此，一个包含5个条目和16个元素的数组的哈希表，其负载系数为0.3125。负载因子越大，发生冲突的可能性就越大。

> One way we mitigate collisions is by resizing the array. Just like the dynamic arrays we implemented earlier, we reallocate and grow the hash table's array as it fills up. Unlike a regular dynamic array,

> though, we won't wait until the array is *full*. Instead, we pick a desired load factor and grow the array when it goes over that.

减少冲突的一种方法是调整数组的大小。就像我们前面实现的动态数组一样，我们在哈希表的数组被填满时，重新分配并扩大该数组。但与常规的动态数组不同的是，我们不会等到数组填满。相反，我们选择一个理想的负载因子，当数组的负载因子超过该值时，我们就扩大数组。

> ## 20.2 Collision Resolution

20.2 冲突解决

> Even with a very low load factor, collisions can still occur. The *birthday paradox* tells us that as the number of entries in the hash table increases, the chance of collision increases very quickly. We can pick a large array size to reduce that, but it's a losing game. Say we wanted to store a hundred items in a hash table. To keep the chance of collision below a still-pretty-high 10%, we need an array with at least 47,015 elements. To get the chance below 1% requires an array with 492,555 elements, over 4,000 empty buckets for each one in use.

即使负载因子很低，仍可能发生碰撞。生日悖论告诉我们，随着哈希表中条目数量的增加，碰撞的概率会很快增加。我们可以选择一个很大的数组规模来减少这种情况，但这是注定失败的。假设我们想在哈希表中存储100个条目，要想使碰撞几率保持在10%以下，我们需要一个至少有47,105个元素的数组。要想使碰撞几率低于1%，需要一个有492,555个元素的数组，每使用一个元素就需要超过4000个空桶。

> A low load factor can make collisions rarer, but the *pigeonhole principle* tells us we can never eliminate them entirely. If you've got five pet pigeons and four holes to put them in, at least one hole is going to end up with more than one pigeon. With 18,446,744,073,709,551,616 different variable names, any reasonably sized array can potentially end up with multiple keys in the same bucket.

低负载因子可以使冲突变少，但是鸽笼原理告诉我们，我们永远无法完全消除冲突。如果你有5只宠物鸽，有4个洞来放它们，至少有一个洞最终会有不止一个鸽子。既然有18,446,744,073,709,551,616个不同的变量名，任何大小合理的数组都有可能在同一个桶中出现多个键。

> Thus we still have to handle collisions gracefully when they occur. Users don't like it when their programming language can look up variables correctly only *most* of the time.

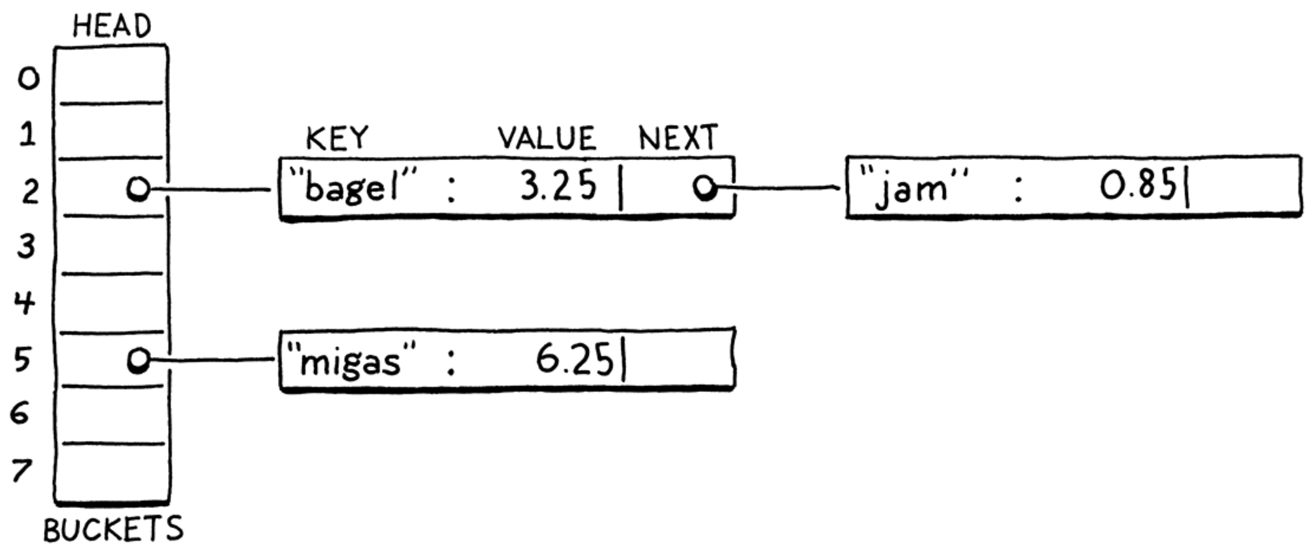
因此，当冲突发生时，我们仍然需要优雅地处理它们。用户并不喜欢他们的编程语言只在*大多数*情况下能正确地查找变量。

> ### 20.2.1 Separate chaining

20.2.1 拉链法

> Techniques for resolving collisions fall into two broad categories. The first is **separate chaining**. Instead of each bucket containing a single entry, we let it contain a collection of them. In the classic implementation, each bucket points to a linked list of entries. To look up an entry, you find its bucket and then walk the list until you find an entry with the matching key.

解决冲突的技术可以分为两大类。第一类是**拉链法**。每个桶中不再包含一个条目，而是包含条目的集合。在经典的实现中，每个桶都指向一个条目的链表。要查找一个条目，你要先找到它的桶，然后遍历列表，直到找到包含匹配键的条目。



> In catastrophically bad cases where every entry collides in the same bucket, the data structure degrades into a single unsorted linked list with *O(n)* lookup. In practice, it's easy to avoid that by controlling the load factor and how entries get scattered across buckets. In typical separate-chained hash tables, it's rare for a bucket to have more than one or two entries.

在最坏的情况下，每个条目都碰撞到同一个桶中，数据结构会退化成一个无序链表，查询复杂度为*O(n)*。在实践中，通过控制负载因子和条目在桶中的分散方式，可以很容易地避免这种情况。在典型的拉链哈希表中，一个桶中很少会有超过一个或两个条目的情况。

> Separate chaining is conceptually simple—it's literally an array of linked lists. Most operations are straightforward to implement, even deletion which, as we'll see, can be a pain. But it's not a great fit for modern CPUs. It has a lot of overhead from pointers and tends to scatter little linked list nodes around in memory which isn't great for cache usage.

拉链法在概念上很简单——它实际上就是一个链表数组。大多数操作实现都可以直接实现，甚至是删除（正如我们将看到的，这可能会很麻烦）。但它并不适合现代的CPU。它有很多指针带来的开销^5，并且倾向于在内存中分散的小的链表节点，这对缓存的使用不是很好。

> ### 20.2.2 Open addressing

20.2.2 开放地址法

> The other technique is called **open addressing** or (confusingly) **closed hashing**. With this technique, all entries live directly in the bucket array, with one entry per bucket. If two entries collide in the same bucket, we find a different empty bucket to use instead.

另一种技术称为**开放地址**或（令人困惑的）**封闭哈希**^6。使用这种技术时，所有的条目都直接存储在桶数组中，每个桶有一个条目。如果两个条目在同一个桶中发生冲突，我们会找一个其它的空桶来代替。

> Storing all entries in a single, big, contiguous array is great for keeping the memory representation simple and fast. But it makes all of the operations on the hash table more complex. When inserting an entry, its bucket may be full, sending us to look at another bucket. That bucket itself may be occupied and so on. This process of finding an available bucket is called **probing**, and the order that you examine buckets is a **probe sequence**.

将所有条目存储在一个单一的、大的、连续的数组中，对于保持内存表示方式的简单和快速是非常好的。但它使得哈希表上的所有操作变得非常复杂。当插入一个条目时，它的桶可能已经满了，这就会让我们去查看另一个桶。而那个桶本身可能也被占用了，等等。这个查找可用存储桶的过程被称为**探测**，而检查存储桶的顺序是**探测序列**。

> There are a number of algorithms for determining which buckets to probe and how to decide which entry goes in which bucket. There's been a ton of research here because even slight tweaks can have a large performance impact. And, on a data structure as heavily used as hash tables, that performance impact touches a very large number of real-world programs across a range of hardware capabilities.

有很多算法^7可以用来确定要探测哪些桶，以及如何决定哪个条目要放在哪个桶中。这方面有大量的研究，因为即使是轻微的调整也会对性能产生很大的影响。而且，对于像哈希表这样大量使用的数据结构来说，这种性能影响涉及到跨一系列硬件功能的大量实际的程序。

> As usual in this book, we'll pick the simplest one that gets the job done efficiently. That's good old **linear probing**. When looking for an entry, we look in the first bucket its key maps to. If it's not in there, we look in the very next element in the array, and so on. If we reach the end, we wrap back around to the beginning.

依照本书的惯例，我们会选择最简单的方法来有效地完成工作。这就是良好的老式**线性探测法**。当查找一个条目时，我们先在它的键映射的桶中查找。如果它不在里面，我们就在数组的下一个元素中查找，以此类推。如果我们到了数组终点，就绕回到起点。

> The good thing about linear probing is that it's cache friendly. Since you walk the array directly in memory order, it keeps the CPU's cache lines full and happy. The bad thing is that it's prone to **clustering**. If you have a lot of entries with numerically similar key values, you can end up with a lot of colliding, overflowing buckets right next to each other.

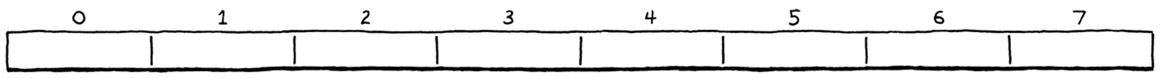
线性探测的好处是它对缓存友好。因为你是直接按照内存顺序遍历数组，所以它可以保持CPU缓存行完整且正常。坏处是，它容易**聚集**。如果你有很多具有相似键值的条目，那最终可能会产生许多相互紧挨的冲突、溢出的桶。

> Compared to separate chaining, open addressing can be harder to wrap your head around. I think of open addressing as similar to separate chaining except that the "list" of nodes is threaded through the bucket array itself. Instead of storing the links between them in pointers, the connections are calculated implicitly by the order that you look through the buckets.

与拉链法相比，开放地址法可能更难理解。我认为开放地址法与拉链法是类似的，区别在于"列表"中的节点是通过桶数组本身进行的。它们之间的链接并没有存储在指针中，而是通过查看桶的顺序隐式计算的。

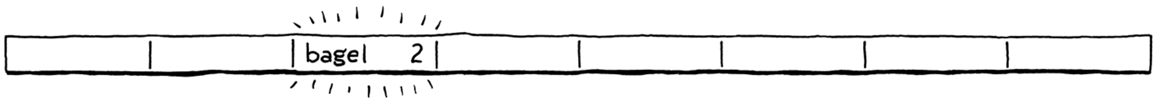
> The tricky part is that more than one of these implicit lists may be interleaved together. Let's walk through an example that covers all the interesting cases. We'll ignore values for now and just worry about a set of keys. We start with an empty array of 8 buckets.

棘手的部分是，这些隐式的列表中可能会有多个交错在一起。让我们通过一个例子，涵盖所有有意思的情况。我们现在先不考虑值，只关心一组键。首先从一个包含8个桶的数组开始。



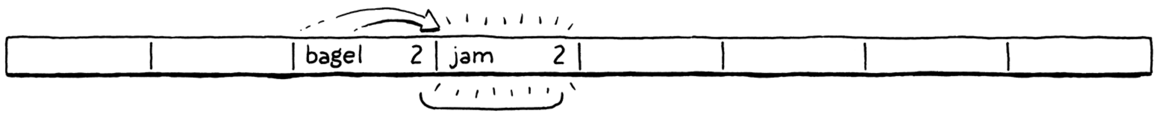
> We decide to insert "bagel". The first letter, "b" (ASCII value 98), modulo the array size (8) puts it in bucket 2.

我们决定插入"bagel"。第一个字母"b"（ASCII值是98），对数组大小（8）取模后，将其放入2号桶中。



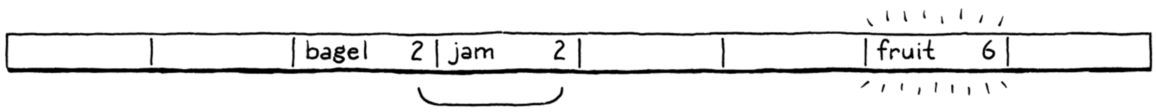
> Next, we insert "jam". That also wants to go in bucket 2 (106 mod 8 = 2), but that bucket's taken. We keep probing to the next bucket. It's empty, so we put it there.

接下来，我们插入"jam"。它也应该放在2号桶中（106 mod 8 = 2），但是这个桶已经被占用了。我们继续探测下一个桶。它是空的，所以我们把它放入其中。



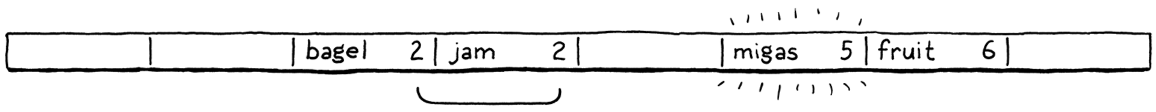
> We insert "fruit", which happily lands in bucket 6.

我们插入"fruit"，它愉快地落在6号桶中。



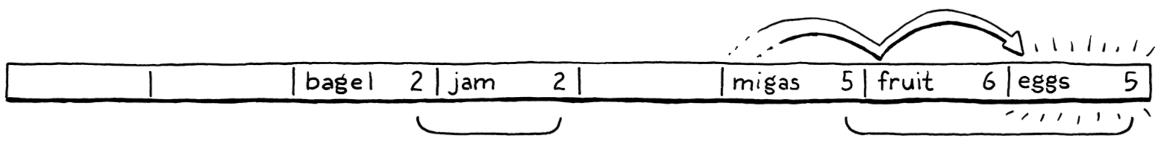
> Likewise, "migas" can go in its preferred bucket 5.

同样，"migas"可以放在其首选的5号桶中。



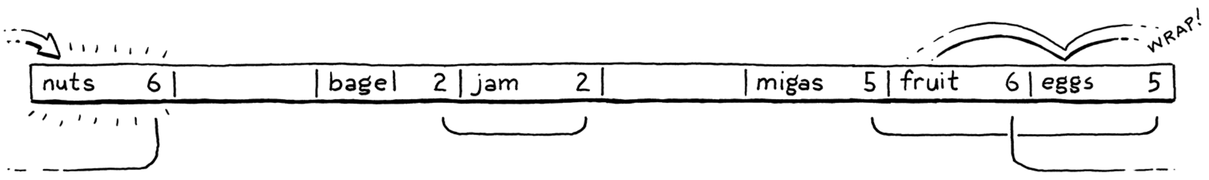
> When we try to insert "eggs", it also wants to be in bucket 5. That's full, so we skip to 6. Bucket 6 is also full. Note that the entry in there is *not* part of the same probe sequence. "Fruit" is in its preferred bucket, 6. So the 5 and 6 sequences have collided and are interleaved. We skip over that and finally put "eggs" in bucket 7.

当我们尝试插入"eggs"时，它也想放在5号桶中。满了，我们跳到6号桶。6号桶也满了。请注意，其中的条目并*不是*当前探测序列的一部分。"Fruit" 在其首选的6号桶中。因此，这里是5号和6号序列发生了碰撞，并交错在一起。我们跳过这个，最后把"eggs"放在7号桶中。



> We run into a similar problem with "nuts". It can't land in 6 like it wants to. Nor can it go into 7. So we keep going. But we've reached the end of the array, so we wrap back around to 0 and put it there.

我们在"nuts"上遇到了同样的问题。它不能按预期进入6号桶中，也不能进入7号桶中。所以我们继续前进，但是我们已经到了数组的末端，所以我们回到0并将其放入0号桶。



> In practice, the interleaving turns out to not be much of a problem. Even in separate chaining, we need to walk the list to check each entry's key because multiple keys can reduce to the same bucket. With open addressing, we need to do that same check, and that also covers the case where you are stepping over entries that "belong" to a different original bucket.

在实践中，这种交错并不是什么大问题。即使是在拉链法中，我们也需要遍历列表来检查每个条目的键，因为多个键会落入同一个桶中。使用开放地址法，我们需要做同样的检查，这也涵盖了你需要遍历"属于"不同原始桶的条目的情况。

20.3 Hash Functions

20.3 哈希函数

> We can now build ourselves a reasonably efficient table for storing variable names up to eight characters long, but that limitation is still annoying. In order to relax the last constraint, we need a way to take a string of any length and convert it to a fixed-size integer.

现在，我们可以为自己构建一个相当有效的表，来存储长度不超过8字符的变量名，但这个限制仍然令人讨厌。为了放宽最后一个限制，我们需要一种方法，将任意长度的字符串转换成固定大小的整数。

> Finally, we get to the "hash" part of "hash table". A **hash function** takes some larger blob of data and "hashes" it to produce a fixed-size integer **hash code** whose value depends on all of the bits of the original data. A good hash function has three main goals:

终于，我们来到了"哈希表"的"哈希"部分。哈希函数接受一些更大的数据块，并将其"哈希"生成一个固定大小的整数**哈希码**，该值取决于原始数据的每一个比特。一个好的哈希函数有三个主要目标^8：

- > **It must be *deterministic*.** The same input must always hash to the same number. If the same variable ends up in different buckets at different points in time, it's gonna get really hard to find it.

  它必须是*确定性*的。相同的输入必须总是哈希到相同的数字。如果同一个变量在不同的时间点出现在不同的桶中，那就很难找到它了。

- > **It must be *uniform*.** Given a typical set of inputs, it should produce a wide and evenly distributed range of output numbers, with as few clumps or patterns as possible. We want it to scatter values across the whole numeric range to minimize collisions and clustering.

  它必须是*均匀*的。给定一组典型的输入，它应该产生一个广泛而均匀分布的输出数字范围，尽可能少地出现簇或模式。我们希望它能在整个数字范围内分散数值，以尽量减少碰撞和聚类^9。

- > **It must be *fast*.** Every operation on the hash table requires us to hash the key first. If hashing is slow, it can potentially cancel out the speed of the underlying array storage.

  它必须是*快速*的。对哈希表的每个操作都需要我们首先对键进行哈希。如果哈希计算很慢，就有可能会抵消底层数组存储的速度优势。

> There is a veritable pile of hash functions out there. Some are old and optimized for architectures no one uses anymore. Some are designed to be fast, others cryptographically secure. Some take advantage of vector instructions and cache sizes for specific chips, others aim to maximize portability.

这里有一堆名副其实的哈希函数。有些是旧的，并且针对已经不再使用的架构进行了优化。有些是为了快速设计的，有些则是加密安全的。有的利用了特定芯片的矢量指令和缓存大小，有的则旨在最大限度地提高可移植性。

> There are people out there for whom designing and evaluating hash functions is, like, their *jam*. I admire them, but I'm not mathematically astute enough to *be* one. So for clox, I picked a simple, well-worn hash function called FNV-1a that's served me fine over the years. Consider trying out different ones in your code and see if they make a difference.

有些人把设计和计算哈希函数当作自己的工作。我很佩服他们，但我在数学上还不够精明，不足以成为其中一员。所以对于clox来说。我选择了一个简单、常用的哈希函数FNV-1a，多年来它一直为我所用。可以考虑在你的代码中尝试不同的方法，看看它们是否有什么不同。

> OK, that's a quick run through of buckets, load factors, open addressing, collision resolution, and hash functions. That's an awful lot of text and not a lot of real code. Don't worry if it still seems vague. Once we're done coding it up, it will all click into place.

好了，我们快速浏览了桶、负载因子、开放地址法、冲突解决和哈希函数。这里有非常多的文字，但没有多少真正的代码。如果它看起来仍然很模糊，不要担心。一旦我们完成了编码，一切都会全部就位。

20.4 Building a Hash Table

20.4 构建哈希表

> The great thing about hash tables compared to other classic techniques like balanced search trees is that the actual data structure is so simple. Ours goes into a new module.

与平衡搜索树等其它经典技术相比，哈希表的好处在于，它实际的数据结构非常简单。我们进入一个新的模块。

table.h，创建新文件：

```
#ifndef clox_table_h
#define clox_table_h

#include "common.h"
#include "value.h"

typedef struct {
  int count;
  int capacity;
  Entry* entries;
} Table;

#endif
```

> A hash table is an array of entries. As in our dynamic array earlier, we keep track of both the allocated size of the array (`capacity`) and the number of key/value pairs currently stored in it (`count`). The ratio of count to capacity is exactly the load factor of the hash table.

哈希表是一个条目数组。就像前面的动态数组一样，我们既要跟踪数组的分配大小（容量，`capacity`）和当前存储在其中的键/值对数量（计数，`count`）。数量与容量的比值正是哈希表的负载因子。

> Each entry is one of these:

每个条目都是这样的：

table.h，添加代码：

```
#include "value.h"
// 新增部分开始
typedef struct {
  ObjString* key;
  Value value;
} Entry;
// 新增部分结束
typedef struct {
```

> It's a simple key/value pair. Since the key is always a string, we store the ObjString pointer directly instead of wrapping it in a Value. It's a little faster and smaller this way.

这是一个简单的键/值对。因为键总是一个字符串^10，我们直接存储ObjString指针，而不是将其包装在Value中。这样做速度更快，体积更小。

> To create a new, empty hash table, we declare a constructor-like function.

为了创建一个新的、空的哈希表，我们声明一个类似构造器的函数。

table.h，在结构体Table后添加代码：

```
} Table;
// 新增部分开始
void initTable(Table* table);
// 新增部分结束
#endif
```

> We need a new implementation file to define that. While we're at it, let's get all of the pesky includes out of the way.

我们需要一个新的实现文件来定义它。既然说到这里，让我们把所有讨厌的依赖文件都搞定。

table.c，创建新文件：

```
#include <stdlib.h>
#include <string.h>

#include "memory.h"
#include "object.h"
#include "table.h"
#include "value.h"

void initTable(Table* table) {
  table->count = 0;
  table->capacity = 0;
  table->entries = NULL;
}
```

> As in our dynamic value array type, a hash table initially starts with zero capacity and a `NULL` array. We don't allocate anything until needed. Assuming we do eventually allocate something, we need to be able to free it too.

就像动态值数组类型一样，哈希表最初以容量0和`NULL`数组开始。等到需要的时候我们才会分配一些东西。假设我们最终分配了什么，我们也需要能够释放它。

table.h，在initTable()方法后添加代码：

```
void initTable(Table* table);
// 新增部分开始
void freeTable(Table* table);
// 新增部分结束
#endif
```

> And its glorious implementation:

以及其绚丽的实现：

table.c，在initTable()方法后添加代码：

```
void freeTable(Table* table) {
  FREE_ARRAY(Entry, table->entries, table->capacity);
  initTable(table);
}
```

> Again, it looks just like a dynamic array. In fact, you can think of a hash table as basically a dynamic array with a really strange policy for inserting items. We don't need to check for `NULL` here since `FREE_ARRAY()` already handles that gracefully.

同样，它看起来就像一个动态数组。实际上，你基本可以把哈希表看作是一个动态数组，它具有一个非常奇怪的插入条目策略。在这里我们不需要检查`NULL`，因为`FREE_ARRAY()`已经优雅地处理了这个问题。

> 20.4.1 Hashing strings

20.4.1 哈希字符串

> Before we can start putting entries in the table, we need to, well, hash them. To ensure that the entries get distributed uniformly throughout the array, we want a good hash function that looks at all of the bits of the key string. If it looked at, say, only the first few characters, then a series of strings that all shared the same prefix would end up colliding in the same bucket.

在我们开始向表中加入条目之前，我们需要，嗯，哈希它们。为了确保条目均匀分布在整个数组中，我们需要一个能考虑键字符串所有比特位的好的哈希函数。如果它只着眼于前几个字符，那么共享相同前缀的一系列字符串最终会碰撞在同一个桶中。

> On the other hand, walking the entire string to calculate the hash is kind of slow. We'd lose some of the performance benefit of the hash table if we had to walk the string every time we looked for a key in the table. So we'll do the obvious thing: cache it.

另一方面，遍历整个字符串来计算哈希值是有点慢的。如果我们每次在哈希表中查找键时都要遍历字符串，就会失去哈希表的一些性能优势。所以我们要做一件显而易见的事：缓存它。

> Over in the "object" module in ObjString, we add:

在"object"模块的ObjString中，添加：

object.h，在结构体ObjString中添加代码：

```
  char* chars;
  // 新增部分开始
  uint32_t hash;
  // 新增部分结束
};
```

> Each ObjString stores the hash code for its string. Since strings are immutable in Lox, we can calculate the hash code once up front and be certain that it will never get invalidated. Caching it eagerly makes a kind of sense: allocating the string and copying its characters over is already an *O(n)* operation, so it's a good time to also do the *O(n)* calculation of the string's hash.

每个ObjString 会存储其字符串的哈希码。由于字符串在Lox中是不可变的，所以我们可以预先计算一次哈希代码，并确保它永远不会失效。提前缓存它是有道理的：分配字符串并复制其字符已然是一个O(n)的操作了，所以这是一个很好的时机来执行字符串哈希的O(n)计算。

> Whenever we call the internal function to allocate a string, we pass in its hash code.

每当我们调用内部函数来分配字符串时，我们就会传入其哈希码。

object.c，在allocateString()方法中替换1行：

```
// 替换部分开始
static ObjString* allocateString(char* chars, int length,
                                                   uint32_t hash) {
  // 替换部分结束
  ObjString* string = ALLOCATE_OBJ(ObjString, OBJ_STRING);
```

> That function simply stores the hash in the struct.

该函数只是将哈希值存储在结构体中

object.c，在allocateString()方法中添加代码：

```
  string->chars = chars;
  // 新增部分开始
  string->hash = hash;
  // 新增部分结束
  return string;
}
```

> The fun happens over at the callers. `allocateString()` is called from two places: the function that copies a string and the one that takes ownership of an existing dynamically allocated string. We'll start with the first.

有趣的部分是在调用者中。`allocateString()`方法在两个地方被调用：复制字符串的函数和获取现有动态分配字符串所有权的函数。我们从第一个开始。

object.c，在copyString()方法中添加代码：

```
ObjString* copyString(const char* chars, int length) {
  // 新增部分开始
  uint32_t hash = hashString(chars, length);
  // 新增部分结束
  char* heapChars = ALLOCATE(char, length + 1);
```

> No magic here. We calculate the hash code and then pass it along.

没有魔法。我们计算哈希码，然后把它传递出去。

object.c，在copyString()方法中替换1行：

```
  memcpy(heapChars, chars, length);
  heapChars[length] = '\0';
  // 替换部分开始
  return allocateString(heapChars, length, hash);
  // 替换部分结束
}
```

> The other string function is similar.

另一个字符串函数是类似的。

object.c，在takeString()方法中替换1行：

```
ObjString* takeString(char* chars, int length) {
  // 替换部分开始
  uint32_t hash = hashString(chars, length);
  return allocateString(chars, length, hash);
  // 替换部分结束
}
```

> The interesting code is over here:

有趣的代码在这里：

object.c，在allocateString()方法后添加代码：

```
static uint32_t hashString(const char* key, int length) {
  uint32_t hash = 2166136261u;
  for (int i = 0; i < length; i++) {
    hash ^= (uint8_t)key[i];
    hash *= 16777619;
  }
  return hash;
}
```

> This is the actual bona fide “hash function” in clox. The algorithm is called “FNV-1a”, and is the shortest decent hash function I know. Brevity is certainly a virtue in a book that aims to show you every line of code.

这就是clox中真正的“哈希函数”。该算法被称为“FNV-1a”，是我所知道的最短的正统哈希函数。对于一本旨在向您展示每一行代码的书来说，简洁无疑是一种美德。

> The basic idea is pretty simple, and many hash functions follow the same pattern. You start with some initial hash value, usually a constant with certain carefully chosen mathematical properties. Then you walk the data to be hashed. For each byte (or sometimes word), you mix the bits into the hash value somehow, and then scramble the resulting bits around some.

基本思想非常简单，许多哈希函数都遵循同样的模式。从一些初始哈希值开始，通常是一个带有某些精心选择的数学特性的常量。然后遍历需要哈希的数据。对于每个字节（有些是每个字），以某种方式将比特与哈希值混合，然后将结果比特进行一些扰乱。

> What it means to “mix” and “scramble” can get pretty sophisticated. Ultimately, though, the basic goal is *uniformity*—we want the resulting hash values to be as widely scattered around the numeric range as possible to avoid collisions and clustering.

“混合”和“扰乱”的含义可以变得相当复杂。不过，最终的基本目标是均匀——我们希望得到的哈希值尽可能广泛地分散在数组范围内，以避免碰撞和聚集。

> 20.4.2 Inserting entries

20.4.2 插入条目

> Now that string objects know their hash code, we can start putting them into hash tables.

现在字符串对象已经知道了它们的哈希码，我们可以开始将它们放入哈希表了。

table.h，在freeTable()方法后添加代码：

```
void freeTable(Table* table);
// 新增部分开始
bool tableSet(Table* table, ObjString* key, Value value);
// 新增部分结束
#endif
```

> This function adds the given key/value pair to the given hash table. If an entry for that key is already present, the new value overwrites the old value. The function returns `true` if a new entry was added. Here's the implementation:

这个函数将给定的键/值对添加到给定的哈希表中。如果该键的条目已存在，新值将覆盖旧值。如果添加了新条目，则该函数返回`true`。下面是实现：

table.c，在freeTable()方法后添加代码：

```
bool tableSet(Table* table, ObjString* key, Value value) {
  Entry* entry = findEntry(table->entries, table->capacity, key);
  bool isNewKey = entry->key == NULL;
  if (isNewKey) table->count++;

  entry->key = key;
  entry->value = value;
  return isNewKey;
}
```

> Most of the interesting logic is in `findEntry()` which we'll get to soon. That function's job is to take a key and figure out which bucket in the array it should go in. It returns a pointer to that bucket—the address of the Entry in the array.

大部分有趣的逻辑都在`findEntry()`中，我们很快就会讲到。该函数的作用是接受一个键，并找到它应该放在数组中的哪个桶里。它会返回一个指向该桶的指针——数组中Entry的地址。

> Once we have a bucket, inserting is straightforward. We update the hash table's size, taking care to not increase the count if we overwrote the value for an already-present key. Then we copy the key and value into the corresponding fields in the Entry.

一旦有了桶，插入就很简单了。我们更新哈希表的大小，如果我们覆盖了一个已经存在的键的值，注意不要增加计数。然后，我们将键和值复制到Entry中的对应字段中。

> We're missing a little something here, though. We haven't actually allocated the Entry array yet. Oops! Before we can insert anything, we need to make sure we have an array, and that it's big enough.

不过，我们这里还少了点什么，我们还没有分配Entry数组。糟糕！在我们向其中插入数据之前，需要确保已经有一个数组，而且足够大。

table.c，在tableSet()方法中添加代码：

```
bool tableSet(Table* table, ObjString* key, Value value) {
  // 新增部分开始
  if (table->count + 1 > table->capacity * TABLE_MAX_LOAD) {
    int capacity = GROW_CAPACITY(table->capacity);
    adjustCapacity(table, capacity);
  }
  // 新增部分结束
  Entry* entry = findEntry(table->entries, table->capacity, key);
```

> This is similar to the code we wrote a while back for growing a dynamic array. If we don't have enough capacity to insert an item, we reallocate and grow the array. The `GROW_CAPACITY()` macro takes an existing capacity and grows it by a multiple to ensure that we get amortized constant performance over a series of inserts.

这与我们之前为扩展动态数组所写的代码相似。如果没有足够的容量插入条目，我们就重新分配和扩展数组。`GROW_CAPACITY()`宏会接受现有容量，并将其增长一倍，以确保在一系列插入操作中得到摊销的常数性能。

> The interesting difference here is that `TABLE_MAX_LOAD` constant.

这里一个有趣的区别是`TABLE_MAX_LOAD`常量。

table.c，添加代码：

```
#include "value.h"
// 新增部分开始
#define TABLE_MAX_LOAD 0.75
// 新增部分结束
void initTable(Table* table) {
```

> This is how we manage the table's load factor. We don't grow when the capacity is completely full. Instead, we grow the array before then, when the array becomes at least 75% full.

这就是我们管理表负载因子的方式。我们不会在容量全满的时候才进行扩展。相反，当数组达到75%满时^11，我们会提前扩展数组。

> We'll get to the implementation of `adjustCapacity()` soon. First, let's look at that `findEntry()` function you've been wondering about.

我们很快就会讨论`adjustCapacity()`的实现。首先，我们看看你一直很好奇的`findEntry()`函数。

table.c，在freeTable()方法后添加代码：

```
static Entry* findEntry(Entry* entries, int capacity,
                        ObjString* key) {
  uint32_t index = key->hash % capacity;
  for (;;) {
    Entry* entry = &entries[index];
    if (entry->key == key || entry->key == NULL) {
      return entry;
    }

    index = (index + 1) % capacity;
  }
}
```

> This function is the real core of the hash table. It's responsible for taking a key and an array of buckets, and figuring out which bucket the entry belongs in. This function is also where linear probing and collision handling come into play. We'll use `findEntry()` both to look up existing entries in the hash table and to decide where to insert new ones.

这个函数是哈希表的真正核心。它负责接受一个键和一个桶数组，并计算出该条目属于哪个桶。这个函数也是线性探测和冲突处理发挥作用的地方。我们在查询哈希表中的现有条目以及决定在哪里插入新条目时，都会使用`findEntry()`方法。

> For all that, there isn't much to it. First, we use modulo to map the key's hash code to an index within the array's bounds. That gives us a bucket index where, ideally, we'll be able to find or place the entry.

尽管如此，其中也没什么特别的。首先，我们通过取余操作将键的哈希码映射为数值边界内的一个索引值。这就给了我们一个桶索引，理想情况下，我们可以在这里找到或放置条目。

> There are a few cases to check for:

有几种情况需要检查：

- > If the key for the Entry at that array index is `NULL`, then the bucket is empty. If we're using `findEntry()` to look up something in the hash table, this means it isn't there. If we're using it to insert, it means we've found a place to add the new entry.

  如果数组索引处的Entry的键为`NULL`，则表示桶为空。如果我们使用`findEntry()`在哈希表中查找东西，这意味着它不存在。如果我们用来插入，这表明我们找到了一个可以插入新条目的地方。

- > If the key in the bucket is equal to the key we're looking for, then that key is already present in the table. If we're doing a lookup, that's good—we've found the key we seek. If we're doing an insert, this means we'll be replacing the value for that key instead of adding a new entry.

  如果桶中的键等于我们要找的键^12，那么这个键已经存在于表中了。如果我们在做查找，这很好——我们已经找到了要查找的键。如果我们在做插入，这意味着我们要替换该键的值，而不是添加一个新条目。

- > Otherwise, the bucket has an entry in it, but with a different key. This is a collision. In that case, we start probing. That's what that `for` loop does. We start at the bucket where the entry would ideally go. If that bucket is empty or has the same key, we're done. Otherwise, we advance to the

> next element—this is the *linear* part of "linear probing"—and check there. If we go past the end of the array, that second modulo operator wraps us back around to the beginning.

否则，就是桶中有一个条目，但具有不同的键。这就是一个冲突。在这种情况下，我们要开始探测。这也就是`for`循环所做的。我们从条目理想的存放位置开始。如果这个桶是空的或者有相同的键，我们就完成了。否则，我们就前进到下一个元素——这就是"线性探测"的*线性*部分——并进行检查。如果我们超过了数组的末端，第二个模运算符就会把我们重新带回起点。

> We exit the loop when we find either an empty bucket or a bucket with the same key as the one we're looking for. You might be wondering about an infinite loop. What if we collide with *every* bucket? Fortunately, that can't happen thanks to our load factor. Because we grow the array as soon as it gets close to being full, we know there will always be empty buckets.

当我们找到空桶或者与我们要找的桶具有相同键的桶时，我们就退出循环。你可能会考虑无限循环的问题。如果我们与所有的桶都冲突怎么办？幸运的是，因为负载因子的原因，这种情况不会发生。因为一旦数组接近满。我们就会扩展数组，所以我们知道总是会有空桶。

> We return directly from within the loop, yielding a pointer to the found Entry so the caller can either insert something into it or read from it. Way back in `tableSet()`, the function that first kicked this off, we store the new entry in that returned bucket and we're done.

我们会从循环中直接返回，得到一个指向找到的Entry的指针，这样调用方就可以向其中插入内容或从中读取内容。回到`tableSet()`——最先调用它的函数，我们将新条目存储到返回的桶中，然后就完成了。

> ### 20.4.3 Allocating and resizing

20.4.3 分配和调整

> Before we can put entries in the hash table, we do need a place to actually store them. We need to allocate an array of buckets. That happens in this function:

在我们将条目放入哈希表之前，我们确实需要一个地方来实际存储它们。我们需要分配一个桶数组。发生在这个函数中：

table.c，在findEntry()方法后添加代码：

```
static void adjustCapacity(Table* table, int capacity) {
  Entry* entries = ALLOCATE(Entry, capacity);
  for (int i = 0; i < capacity; i++) {
    entries[i].key = NULL;
    entries[i].value = NIL_VAL;
  }

  table->entries = entries;
  table->capacity = capacity;
}
```

> We create a bucket array with `capacity` entries. After we allocate the array, we initialize every element to be an empty bucket and then store the array (and its capacity) in the hash table's main struct. This

> code is fine for when we insert the very first entry into the table, and we require the first allocation of the array. But what about when we already have one and we need to grow it?

我们创建一个包含`capacity`个条目的桶数组。分配完数组后，我们将每个元素初始化为空桶，然后将数组（及其容量）存储到哈希表的主结构体中。当我们将第一个条目插入表中时，这段代码是没有问题的，而且我们需要对数组进行第一次分配。但如果我们已经有了一个数组，并且需要增加它的容量时，怎么办？

> Back when we were doing a dynamic array, we could just use `realloc()` and let the C standard library copy everything over. That doesn't work for a hash table. Remember that to choose the bucket for each entry, we take its hash key *modulo the array size*. That means that when the array size changes, entries may end up in different buckets.

在我们做动态数组时，我们只需使用`realloc()`，让C标准库把所有内容都复制过来。这对哈希表是行不通的。请记住，为了给每个条目选择存储桶，我们要用其哈希键与数组大小*取模*。这意味着，当数组大小发生变化时，条目可能会出现在不同的桶中。

> Those new buckets may have new collisions that we need to deal with. So the simplest way to get every entry where it belongs is to rebuild the table from scratch by re-inserting every entry into the new empty array.

这些新的桶可能会出现新的冲突，我们需要处理这些冲突。因此，获取每个条目所属位置的最简单的方法是从头重新构建哈希表，将每个条目都重新插入到新的空数组中。

table.c，在adjustCapacity()方法中添加代码：

```
    entries[i].value = NIL_VAL;
  }
  // 新增部分开始
  for (int i = 0; i < table->capacity; i++) {
    Entry* entry = &table->entries[i];
    if (entry->key == NULL) continue;

    Entry* dest = findEntry(entries, capacity, entry->key);
    dest->key = entry->key;
    dest->value = entry->value;
  }
  // 新增部分结束
  table->entries = entries;
```

We walk through the old array front to back. Any time we find a non-empty bucket, we insert that entry into the new array. We use `findEntry()`, passing in the *new* array instead of the one currently stored in the Table. (This is why `findEntry()` takes a pointer directly to an Entry array and not the whole `Table` struct. That way, we can pass the new array and capacity before we've stored those in the struct.)

我们从前到后遍历旧数组。只要发现一个非空的桶，我们就把这个条目插入到新的数组中。我们使用`findEntry()`，传入的是*新*数组，而不是当前存储在哈希表中的旧数组。（这就是为什么`findEntry()`接受一个直接指向Entry数组的指针，而不是整个`Table`结构体。这样，我们就可以在将新数组和容量存储到结构体之前传递这些数据）

> After that's done, we can release the memory for the old array.

完成之后，我们就可以释放旧数组的内存。

table.c，在adjustCapacity()方法中添加代码：

```
    dest->value = entry->value;
  }
  // 新增部分开始
  FREE_ARRAY(Entry, table->entries, table->capacity);
  // 新增部分结束
  table->entries = entries;
```

> With that, we have a hash table that we can stuff as many entries into as we like. It handles overwriting existing keys and growing itself as needed to maintain the desired load capacity.

这样，我们就有了一个哈希表，我们可以随心所欲地向其中塞入很多条目。它可以处理覆盖现有键，以及按需扩展自身，以保持所需的负载容量。

> While we're at it, let's also define a helper function for copying all of the entries of one hash table into another.

既然如此，我们也来定义一个辅助函数，将一个哈希表的所有条目复制到另一个哈希表中。

table.h，在tableSet()方法后添加代码：

```
bool tableSet(Table* table, ObjString* key, Value value);
// 新增部分开始
void tableAddAll(Table* from, Table* to);
// 新增部分结束
#endif
```

> We won't need this until much later when we support method inheritance, but we may as well implement it now while we've got all the hash table stuff fresh in our minds.

等到很久之后我们要支持方法继承时，才会需要这个能力，但是我们不妨现在就实现它，趁着我们还记得哈希表的一切内容。

table.c，在tableSet()方法后添加代码：

```
void tableAddAll(Table* from, Table* to) {
  for (int i = 0; i < from->capacity; i++) {
    Entry* entry = &from->entries[i];
    if (entry->key != NULL) {
      tableSet(to, entry->key, entry->value);
    }
  }
}
```

> There's not much to say about this. It walks the bucket array of the source hash table. Whenever it finds a non-empty bucket, it adds the entry to the destination hash table using the `tableSet()` function we recently defined.

这没什么可说的。它会遍历源哈希表的桶数组。只要发现一个非空的桶，就使用我们刚定义的`tableSet()`函数将条目添加到目标哈希表中。

> 20.4.4 Retrieving values

20.4.4 检索值

> Now that our hash table contains some stuff, let's start pulling things back out. Given a key, we can look up the corresponding value, if there is one, with this function:

现在，我们的哈希表中已经包含了一些东西，让我们开始把它们取出来。给定一个键，如果有对应的条目，我们可以用这个函数来查找对应的值：

table.h，在freeTable()方法后添加代码：

```
void freeTable(Table* table);
// 新增部分开始
bool tableGet(Table* table, ObjString* key, Value* value);
// 新增部分结束
bool tableSet(Table* table, ObjString* key, Value value);
```

> You pass in a table and a key. If it finds an entry with that key, it returns `true`, otherwise it returns `false`. If the entry exists, the `value` output parameter points to the resulting value.

传入一个表和一个键。如果它找到一个带有该键的条目，则返回`true`，否则返回`false`。如果该条目存在，输出的`value`参数会指向结果值。

> Since `findEntry()` already does the hard work, the implementation isn't bad.

因为`findEntry()`已经完成了这项艰苦的工作，所以实现起来并不难。

table.c，在findEntry()方法后添加代码：

```
bool tableGet(Table* table, ObjString* key, Value* value) {
  if (table->count == 0) return false;

  Entry* entry = findEntry(table->entries, table->capacity, key);
  if (entry->key == NULL) return false;

  *value = entry->value;
  return true;
}
```

> If the table is completely empty, we definitely won't find the entry, so we check for that first. This isn't just an optimization—it also ensures that we don't try to access the bucket array when the array is `NULL`. Otherwise, we let `findEntry()` work its magic. That returns a pointer to a bucket. If the bucket is empty, which we detect by seeing if the key is `NULL`, then we didn't find an Entry with our key. If `findEntry()` does return a non-empty Entry, then that's our match. We take the Entry's value and copy it to the output parameter so the caller can get it. Piece of cake.

如果表完全是空的，我们肯定找不到这个条目，所以我们先检查一下。这不仅仅是一种优化——它还确保当数组为`NULL`时，我们不会试图访问桶数组。其它情况下，我们就让`findEntry()`发挥它的魔力。这将返回一个指向桶的指针。如果桶是空的（我们通过查看键是否为`NULL`来检测），那么我们就没有找到包含对应键的Entry。如果`findEntry()`确实返回了一个非空的Entry，那么它就是我们的匹配项。我们获取Entry的值并将其复制到输出参数中，这样调用方就可以得到该值。小菜一碟。

> ### 20.4.5 Deleting entries

20.4.5 删除条目

> There is one more fundamental operation a full-featured hash table needs to support: removing an entry. This seems pretty obvious, if you can add things, you should be able to *un*-add them, right? But you'd be surprised how many tutorials on hash tables omit this.

全功能的哈希表还需要支持一个更基本的操作：删除一个条目。这看起来很明显，如果你能添加东西，你应该就能删除它，对吗？但你会惊讶于有多少关于哈希表的教程省略了这一点。

> I could have taken that route too. In fact, we use deletion in clox only in a tiny edge case in the VM. But if you want to actually understand how to completely implement a hash table, this feels important. I can sympathize with their desire to overlook it. As we'll see, deleting from a hash table that uses open addressing is tricky.

我也可以走这条路。事实上，我们在clox中只在VM的一个很小的边缘情况下会使用删除。但如果你想真正理解如何完全实现一个哈希表，这一点很重要。我能理解它们忽略这一点的想法。正如我们将看到的，从使用开放地址法的哈希表中删除数据是很棘手的^13。

> At least the declaration is simple.

至少声明是简单的。

table.h，在tableSet()方法后添加代码：

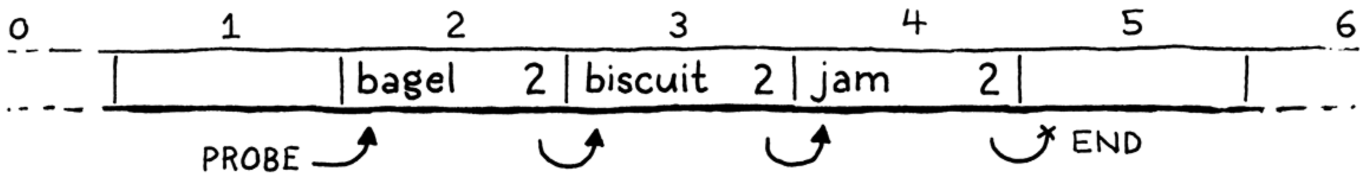
```
bool tableSet(Table* table, ObjString* key, Value value);
// 新增部分开始
bool tableDelete(Table* table, ObjString* key);
// 新增部分结束
void tableAddAll(Table* from, Table* to);
```

> The obvious approach is to mirror insertion. Use `findEntry()` to look up the entry's bucket. Then clear out the bucket. Done!

一个明显的方法就是插入的镜像操作。使用`findEntry()`找到条目的桶，然后把桶清空。完成了！

> In cases where there are no collisions, that works fine. But if a collision has occurred, then the bucket where the entry lives may be part of one or more implicit probe sequences. For example, here's a hash table containing three keys all with the same preferred bucket, 2:

在没有冲突的情况下，这样做没问题。但是如果发生了冲突，那么条目所在的桶可能是一个或多个隐式探测序列的一部分。举例来说，下面是一个哈希表，包含三个键，它们有着相同的首选桶，2：



> Remember that when we're walking a probe sequence to find an entry, we know we've reached the end of a sequence and that the entry isn't present when we hit an empty bucket. It's like the probe sequence is a list of entries and an empty entry terminates that list.

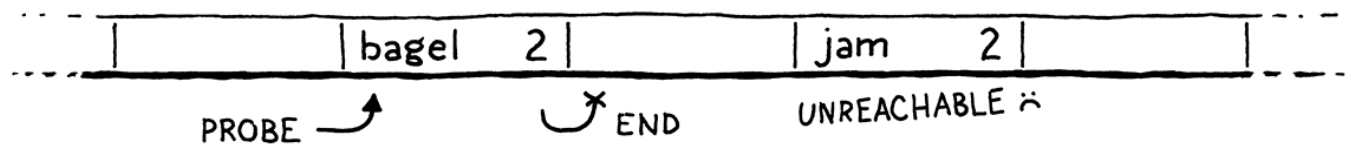
请记住，我们在遍历探测序列来查找一个条目时，如果碰到一个空桶，我们就知道已经到达了序列的末端而且该条目不存在。这就好像探测序列是一个条目的列表，而空条目终止了这个列表。

> If we delete "biscuit" by simply clearing the Entry, then we break that probe sequence in the middle, leaving the trailing entries orphaned and unreachable. Sort of like removing a node from a linked list without relinking the pointer from the previous node to the next one.

如果我们通过简单地清除Entry来删除"biscuit"，那么我们就会中断探测序列，让后面的条目变得孤立、不可访问。这有点像是从链表中删除了一个节点，而没有把指针从上一个节点重新链接到下一个节点。

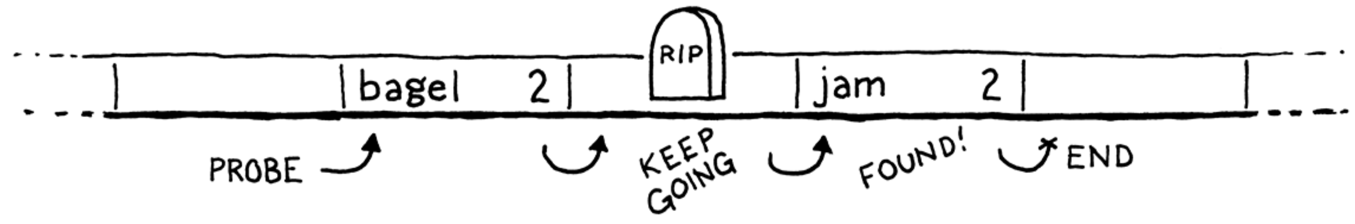
> If we later try to look for "jam", we'd start at "bagel", stop at the next empty Entry, and never find it.

如果我们后面尝试查找"jam"，我们会从"bagel"开始，在下一个空条目处停止，并且永远找不到它。



> To solve this, most implementations use a trick called **tombstones**. Instead of clearing the entry on deletion, we replace it with a special sentinel entry called a "tombstone". When we are following a probe sequence during a lookup, and we hit a tombstone, we *don't* treat it like an empty slot and stop iterating. Instead, we keep going so that deleting an entry doesn't break any implicit collision chains and we can still find entries after it.

为了解决这个问题，大多数实现都使用了一个叫作**墓碑**的技巧。我们不会在删除时清除条目，而是将其替换为一个特殊的哨兵条目，称为"墓碑"^14。当我们在查找过程中顺着探测序列遍历时，如果遇到墓碑，我们不会把它当作是空槽而停止遍历。相反，我们会继续前进，这样删除一个条目不会破坏任何隐式冲突链，我们仍然可以找到它之后的条目。



> The code looks like this:

这段代码看起来像这样：

table.c，在tableSet()方法后添加代码：

```
bool tableDelete(Table* table, ObjString* key) {
  if (table->count == 0) return false;

  // Find the entry.
  Entry* entry = findEntry(table->entries, table->capacity, key);
  if (entry->key == NULL) return false;

  // Place a tombstone in the entry.
  entry->key = NULL;
  entry->value = BOOL_VAL(true);
  return true;
}
```

> First, we find the bucket containing the entry we want to delete. (If we don’t find it, there’s nothing to delete, so we bail out.) We replace the entry with a tombstone. In clox, we use a `NULL` key and a `true` value to represent that, but any representation that can’t be confused with an empty bucket or a valid entry works.

首先，我们找到包含待删除条目的桶（如果我们没有找到，就没有什么可删除的，所以我们退出）。我们将该条目替换为墓碑。在clox中，我们使用`NULL`键和`true`值来表示，但任何不会与空桶或有效条目相混淆的表示形式都是可行的。

> That’s all we need to do to delete an entry. Simple and fast. But all of the other operations need to correctly handle tombstones too. A tombstone is a sort of “half” entry. It has some of the characteristics of a present entry, and some of the characteristics of an empty one.

这就是删除一个条目所需要做的全部工作。简单而快速。但是所有其它操作也需要正确处理墓碑。墓碑有点像“半个”条目。它既有当前条目的一些特征，也有空条目的一些特征。

> When we are following a probe sequence during a lookup, and we hit a tombstone, we note it and keep going.

当我们在查询中遵循探测序列时，如果遇到了墓碑，我们会记录它并继续前进。

table.c，在findEntry()方法中替换3行：

```
  for (;;) {
    Entry* entry = &entries[index];
    // 替换部分开始
    if (entry->key == NULL) {
      if (IS_NIL(entry->value)) {
        // Empty entry.
        return tombstone != NULL ? tombstone : entry;
      } else {
        // We found a tombstone.
        if (tombstone == NULL) tombstone = entry;
      }
    } else if (entry->key == key) {
      // We found the key.
      return entry;
    }
    // 替换部分结束
    index = (index + 1) % capacity;
```

> The first time we pass a tombstone, we store it in this local variable:

第一次通过一个墓碑条目时，我们将它存储在这个局部变量中：

table.c，在findEntry()方法中添加代码：

```
  uint32_t index = key->hash % capacity;
  // 新增部分开始
  Entry* tombstone = NULL;
  // 新增部分结束
  for (;;) {
```

> If we reach a truly empty entry, then the key isn't present. In that case, if we have passed a tombstone, we return its bucket instead of the later empty one. If we're calling `findEntry()` in order to insert a node, that lets us treat the tombstone bucket as empty and reuse it for the new entry.

如果我们遇到一个真正的空条目，那么这个键就不存在。在这种情况下，若我们经过了一个墓碑，就返回它的桶，而不是返回后面的空桶。如果我们为了插入一个节点而调用`findEntry()`，这时我们可以把墓碑桶视为空桶，并重用它来存储新条目。

> Reusing tombstone slots automatically like this helps reduce the number of tombstones wasting space in the bucket array. In typical use cases where there is a mixture of insertions and deletions, the number of tombstones grows for a while and then tends to stabilize.

像这样自动重用墓碑槽，有助于减少墓碑在桶数组中浪费的空间。在插入与删除混合使用的典型用例中，墓碑的数量会增长一段时间，然后趋于稳定。

> Even so, there's no guarantee that a large number of deletes won't cause the array to be full of tombstones. In the very worst case, we could end up with *no* empty buckets. That would be bad because, remember, the only thing preventing an infinite loop in `findEntry()` is the assumption that we'll eventually hit an empty bucket.

即便如此，也不能保证大量的删除操作不会导致数组中满是墓碑。在最坏的情况下，我们最终可能没有空桶。这是很糟糕的。因为，请记住，唯一能阻止`findEntry()`中无限循环的原因是假设我们最终会命中一个空桶。

> So we need to be thoughtful about how tombstones interact with the table's load factor and resizing. The key question is, when calculating the load factor, should we treat tombstones like full buckets or empty ones?

所以我们需要仔细考虑墓碑如何与表的负载因子和大小调整进行互动。关键问题是，在计算负载因子时，我们应该把墓碑当作满桶还是空桶？

> 20.4.6 Counting tombstones

20.4.6 墓碑计数

> If we treat tombstones like full buckets, then we may end up with a bigger array than we probably need because it artificially inflates the load factor. There are tombstones we could reuse, but they aren't treated as unused so we end up growing the array prematurely.

如果我们把墓碑当作满桶，那么我们最终得到的数组可能会比我们需要的更大，因为它人为地抬高了负载因子。有些墓碑我们可以重复使用，但是它们没有被视为未使用，所以我们最终会过早地扩展数组。

> But if we treat tombstones like empty buckets and *don't* include them in the load factor, then we run the risk of ending up with *no* actual empty buckets to terminate a lookup. An infinite loop is a much worse problem than a few extra array slots, so for load factor, we consider tombstones to be full buckets.

但是，如果我们把墓碑当作空桶，并且*不*将它们计入负载因子中，那么我们就有可能*没有*真正的空桶来终止查找。无限循环比几个多余的数组槽要糟糕得多，所以对于负载因子，我们把墓碑看作是满桶。

> That's why we don't reduce the count when deleting an entry in the previous code. The count is no longer the number of entries in the hash table, it's the number of entries plus tombstones. That implies that we increment the count during insertion only if the new entry goes into an entirely empty bucket.

这就是我们在前面的代码中删除条目却不减少计数的原因。这个计数不再是哈希表中的条目数，而是条目数加上墓碑数。这意味着，只有当新条目进入一个完全空的桶中时，才会在插入操作中增加计数。

table.c，在tableSet()方法中替换1行：

```
  bool isNewKey = entry->key == NULL;
  // 替换部分开始
  if (isNewKey && IS_NIL(entry->value)) table->count++;
  // 替换部分结束
  entry->key = key;
```

> If we are replacing a tombstone with a new entry, the bucket has already been accounted for and the count doesn't change.

如果我们用新条目替换墓碑，因为这个桶已经被统计过了，所以计数不会改变。

> When we resize the array, we allocate a new array and re-insert all of the existing entries into it. During that process, we *don't* copy the tombstones over. They don't add any value since we're rebuilding the probe sequences anyway, and would just slow down lookups. That means we need to recalculate the count since it may change during a resize. So we clear it out:

当我们调整数组的大小时，我们会分配一个新数组，并重新插入所有的现存条目。在这个过程中，我们不会把墓碑复制过来。因为无论如何我们都要重新构建探测序列，它们不会增加任何价值，而且只会减慢查找速度。

这意味着我们需要重算计数，因为它可能会在调整大小的期间发生变化。所以我们把它清除掉：

table.c，在adjustCapacity()方法中添加代码：

```
  }
  // 新增部分开始
  table->count = 0;
  // 新增部分结束
  for (int i = 0; i < table->capacity; i++) {
```

> Then each time we find a non-tombstone entry, we increment it.

然后，每当我们找到一个非墓碑的条目，就给它加1。

table.c，在adjustCapacity()方法中添加代码：

```
    dest->value = entry->value;
    // 新增部分开始
    table->count++;
    // 新增部分结束
  }
```

> This means that when we grow the capacity, we may end up with *fewer* entries in the resulting larger array because all of the tombstones get discarded. That’s a little wasteful, but not a huge practical problem.

这意味着，当我们增加容量时，最终在更大的数组中的条目可能会*更少*，因为所有的墓碑都被丢弃了。这有点浪费，但不是一个严重的实际问题。

> I find it interesting that much of the work to support deleting entries is in `findEntry()` and `adjustCapacity()`. The actual delete logic is quite simple and fast. In practice, deletions tend to be rare, so you’d expect a hash table to do as much work as it can in the delete function and leave the other functions alone to keep them faster. With our tombstone approach, deletes are fast, but lookups get penalized.

我发现一个有趣的现象，支持删除条目的大部分工作都是在`findEntry()`和`adjustCapacity()`中完成的。实际的删除逻辑是相当简单和快速的。在实践中，删除操作往往是少见的，所以你会希望哈希表在删除函数中完成尽可能多的工作，而让其它函数保持更快的速度。使用我们的墓碑方案，删除是快速的，但查找会受到影响。

> I did a little benchmarking to test this out in a few different deletion scenarios. I was surprised to discover that tombstones did end up being faster overall compared to doing all the work during deletion to reinsert the affected entries.

我做了一个小小的基准测试，在一些不同的删除场景中验证这一点。我惊讶地发现，与在删除过程中做所有的工作来重新插入受影响的条目相比，墓碑方案最终确实更快。

> But if you think about it, it's not that the tombstone approach pushes the work of fully deleting an entry to other operations, it's more that it makes deleting *lazy*. At first, it does the minimal work to turn the entry into a tombstone. That can cause a penalty when later lookups have to skip over it. But it also allows that tombstone bucket to be reused by a later insert too. That reuse is a very efficient way to avoid the cost of rearranging all of the following affected entries. You basically recycle a node in the chain of probed entries. It's a neat trick.

但如果你仔细想想，墓碑方案并不是将删除的工作完全推给了其它操作，它更像是让删除*延迟*了。首先，它只做了很少的工作，把条目变成墓碑。当以后的查找不得不跳过它时，这可能会造成损失。但是，它也允许墓碑桶被后续的插入操作重用。这种重用是一种非常有效的方法，可以避免重新安排后续受影响的条目的成本。你基本上是回收了探测条目链中的一个节点。这是一个很巧妙的技巧。

20.5 String Interning

20.5 字符串驻留

> We've got ourselves a hash table that mostly works, though it has a critical flaw in its center. Also, we aren't using it for anything yet. It's time to address both of those and, in the process, learn a classic technique used by interpreters.

我们已经有了一个基本可用的哈希表，尽管它的中心有一个严重的缺陷。此外，我们还没有用它做任何事情。现在是时候解决这两个问题了，在这个过程中，我们要学习解释器中使用的一个经典技术。

> The reason the hash table doesn't totally work is that when `findEntry()` checks to see if an existing key matches the one it's looking for, it uses `==` to compare two strings for equality. That only returns true if the two keys are the exact same string in memory. Two separate strings with the same characters should be considered equal, but aren't.

哈希表不能完全工作的原因在于，当`findEntry()`检查一个现有的键是否与要查找的键相匹配时，它使用了`==`来比较两个字符串是否相等。只有当两个键在内存中是完全相同的字符串时，才会返回true。两个具有相同字符的单独的字符串也应该被认为是相等的，但是并没有。

> Remember, back when we added strings in the last chapter, we added explicit support to compare the strings character-by-character in order to get true value equality. We could do that in `findEntry()`, but that's slow.

还记得吗？在上一章我们添加字符串时，也显式支持了逐字符比较字符串，以实现真正的值相等。我们在`findEntry()`中也可以这样做，但这很慢^15。

> Instead, we'll use a technique called **string interning**. The core problem is that it's possible to have different strings in memory with the same characters. Those need to behave like equivalent values even though they are distinct objects. They're essentially duplicates, and we have to compare all of their bytes to detect that.

相反，我们将使用一种叫作**字符串驻留**的技术，核心问题是，在内存中不同的字符串可能包含相同的字符。尽管它们是不同的对象，它们的行为也需要像等效值一样。它们本质上是相同的，而我们必须比较它们所有的字节来检查这一点。

> String interning is a process of deduplication. We create a collection of "interned" strings. Any string in that collection is guaranteed to be textually distinct from all others. When you intern a string, you look

> for a matching string in the collection. If found, you use that original one. Otherwise, the string you have is unique, so you add it to the collection.

字符串驻留是一个数据去重的过程^16。我们创建一个“驻留”字符串的集合。该集合中的任何字符串都保证与其它字符串在文本上不相同。当你要驻留一个字符串时，首先从集合中查找匹配的字符串，如果找到了，就使用原来的那个。否则，说明你持有的字符串是唯一的，所以你将其添加到集合中。

> In this way, you know that each sequence of characters is represented by only one string in memory. This makes value equality trivial. If two strings point to the same address in memory, they are obviously the same string and must be equal. And, because we know strings are unique, if two strings point to different addresses, they must be distinct strings.

通过这种方式，你知道每个字符序列在内存中只由一个字符串表示。这使得值相等变得很简单。如果两个字符串在内存中指向相同的地址，它们显然是同一个字符串，并且必须相等。而且，因为我们知道字符串是唯一的，如果两个字符串指向不同的地址，它们一定是不同的。

> Thus, pointer equality exactly matches value equality. Which in turn means that our existing `==` in `findEntry()` does the right thing. Or, at least, it will once we intern all the strings. In order to reliably deduplicate all strings, the VM needs to be able to find every string that's created. We do that by giving it a hash table to store them all.

因此，指针相等与值相等完全匹配。这反过来又意味着，我们在`findEntry()`中使用的`==`做了正确的事情。或者说，至少在我们实现字符串驻留之后，它是对的。为了可靠地去重所有字符串，虚拟机需要能够找到创建的每个字符串。我们用一个哈希表存储这些字符串，从而实现这一点。

vm.h，在结构体VM中添加代码：

```
  Value* stackTop;
  // 新增部分开始
  Table strings;
  // 新增部分结束
  Obj* objects;
```

> As usual, we need an include.

像往常一样，我们需要引入头文件。

vm.h，添加代码：

```
#include "chunk.h"
// 新增部分开始
#include "table.h"
// 新增部分结束
#include "value.h"
```

> When we spin up a new VM, the string table is empty.

当我们启动一个新的虚拟机时，字符串表是空的。

vm.c，在initVM()方法中添加代码：

```
  vm.objects = NULL;
  // 新增部分开始
  initTable(&vm.strings);
  // 新增部分结束
}
```

> And when we shut down the VM, we clean up any resources used by the table.

而当我们关闭虚拟机时，我们要清理该表使用的所有资源。

vm.c，在freeVM()方法中添加代码：

```
void freeVM() {
  // 新增部分开始
  freeTable(&vm.strings);
  // 新增部分结束
  freeObjects();
```

> Some languages have a separate type or an explicit step to intern a string. For clox, we'll automatically intern every one. That means whenever we create a new unique string, we add it to the table.

一些语言中有单独的类型或显式步骤来驻留字符串。对于clox，我们会自动驻留每个字符串。这意味着，每当我们创建了一个新的唯一字符串，就将其添加到表中。

object.c，在allocateString()方法中添加代码：

```
  string->hash = hash;
  // 新增部分开始
  tableSet(&vm.strings, string, NIL_VAL);
  // 新增部分结束
  return string;
```

> We're using the table more like a hash *set* than a hash *table*. The keys are the strings and those are all we care about, so we just use `nil` for the values.

我们使用这个表的方式更像是哈希*集合*而不是哈希*表*。键是字符串，而我们只关心这些，所以我们用`nil`作为值。

> This gets a string into the table assuming that it's unique, but we need to actually check for duplication before we get here. We do that in the two higher-level functions that call `allocateString()`. Here's one:

假定一个字符串是唯一的，这就会把它放入表中，但在此之前，我们需要实际检查字符串是否有重复。我们在两个调用`allocateString()`的高级函数中做到了这一点。这里有一个：

object.c，在copyString()方法中添加代码：

```
  uint32_t hash = hashString(chars, length);
  // 新增部分开始
  ObjString* interned = tableFindString(&vm.strings, chars, length,
                                        hash);
  if (interned != NULL) return interned;
  // 新增部分结束
  char* heapChars = ALLOCATE(char, length + 1);
```

> When copying a string into a new LoxString, we look it up in the string table first. If we find it, instead of “copying”, we just return a reference to that string. Otherwise, we fall through, allocate a new string, and store it in the string table.

当把一个字符串复制到新的LoxString中时，我们首先在字符串表中查找它。如果找到了，我们就不“复制”，而是直接返回该字符串的引用。如果没有找到，我们就是落空了，则分配一个新字符串，并将其存储到字符串表中。

> Taking ownership of a string is a little different.

获取字符串的所有权有点不同。

object.c，在takeString()方法中添加代码：

```
  uint32_t hash = hashString(chars, length);
  // 新增部分开始
  ObjString* interned = tableFindString(&vm.strings, chars, length,
                                        hash);
  if (interned != NULL) {
    FREE_ARRAY(char, chars, length + 1);
    return interned;
  }
  // 新增部分结束
  return allocateString(chars, length, hash);
```

> Again, we look up the string in the string table first. If we find it, before we return it, we free the memory for the string that was passed in. Since ownership is being passed to this function and we no longer need the duplicate string, it’s up to us to free it.

同样，我们首先在字符串表中查找该字符串。如果找到了，在返回它之前，我们释放传入的字符串的内存。因为所有权被传递给了这个函数，我们不再需要这个重复的字符串，所以由我们释放它。

> Before we get to the new function we need to write, there’s one more include.

在开始编写新函数之前，还需要引入一个头文件。

object.c，添加代码：

```
#include "object.h"
// 新增部分开始
#include "table.h"
// 新增部分结束
#include "value.h"
```

> To look for a string in the table, we can't use the normal `tableGet()` function because that calls `findEntry()`, which has the exact problem with duplicate strings that we're trying to fix right now. Instead, we use this new function:

要在表中查找字符串，我们不能使用普通的`tableGet()`函数，因为它调用了`findEntry()`，这正是我们现在试图解决的重复字符串的问题。相反地，我们使用这个新函数：

table.h，在tableAddAll()方法后添加代码：

```
void tableAddAll(Table* from, Table* to);
// 新增部分开始
ObjString* tableFindString(Table* table, const char* chars,
                           int length, uint32_t hash);
// 新增部分结束
#endif
```

> The implementation looks like so:

其实现如下：

table.c，在tableAddAll()方法后添加代码：

```
ObjString* tableFindString(Table* table, const char* chars,
                           int length, uint32_t hash) {
  if (table->count == 0) return NULL;

  uint32_t index = hash % table->capacity;
  for (;;) {
    Entry* entry = &table->entries[index];
    if (entry->key == NULL) {
      // Stop if we find an empty non-tombstone entry.
      if (IS_NIL(entry->value)) return NULL;
    } else if (entry->key->length == length &&
        entry->key->hash == hash &&
        memcmp(entry->key->chars, chars, length) == 0) {
      // We found it.
      return entry->key;
    }

    index = (index + 1) % table->capacity;
  }
}
```

> It appears we have copy-pasted `findEntry()`. There is a lot of redundancy, but also a couple of key differences. First, we pass in the raw character array of the key we're looking for instead of an ObjString. At the point that we call this, we haven't created an ObjString yet.

看起来我们是复制粘贴了`findEntry()`。这里确实有很多冗余，但也有几个关键的区别。首先，我们传入的是我们要查找的键的原始字符数组，而不是ObjString。在我们调用这个方法时，还没有创建ObjString。

> Second, when checking to see if we found the key, we look at the actual strings. We first see if they have matching lengths and hashes. Those are quick to check and if they aren't equal, the strings definitely aren't the same.

其次，在检查是否找到键时，我们要看一下实际的字符串。我们首先看看它们的长度和哈希值是否匹配。这些都是快速检查，如果它们不相等，那些字符串肯定不一样。

> If there is a hash collision, we do an actual character-by-character string comparison. This is the one place in the VM where we actually test strings for textual equality. We do it here to deduplicate strings and then the rest of the VM can take for granted that any two strings at different addresses in memory must have different contents.

如果存在哈希冲突，我们就进行实际的逐字符的字符串比较。这是虚拟机中我们真正测试字符串是否相等的一个地方。我们在这里这样做是为了对字符串去重，然后虚拟机的其它部分可以想当然地认为，内存中不同地址的任意两个字符串一定有着不同的内容。

> In fact, now that we've interned all the strings, we can take advantage of it in the bytecode interpreter. When a user does `==` on two objects that happen to be strings, we don't need to test the characters any more.

事实上，既然我们已经驻留了所有的字符串，我们就可以在字节码解释器中利用这一优势。当用户对两个字符串对象进行`==`时，我们不需要再检查字符了。

value.c，在valuesEqual()方法中替换7行：

```
    case VAL_NUMBER: return AS_NUMBER(a) == AS_NUMBER(b);
    // 替换部分开始
    case VAL_OBJ:    return AS_OBJ(a) == AS_OBJ(b);
    // 替换部分结束
    default:         return false; // Unreachable.
```

> We've added a little overhead when creating strings to intern them. But in return, at runtime, the equality operator on strings is much faster. With that, we have a full-featured hash table ready for us to use for tracking variables, instances, or any other key-value pairs that might show up.

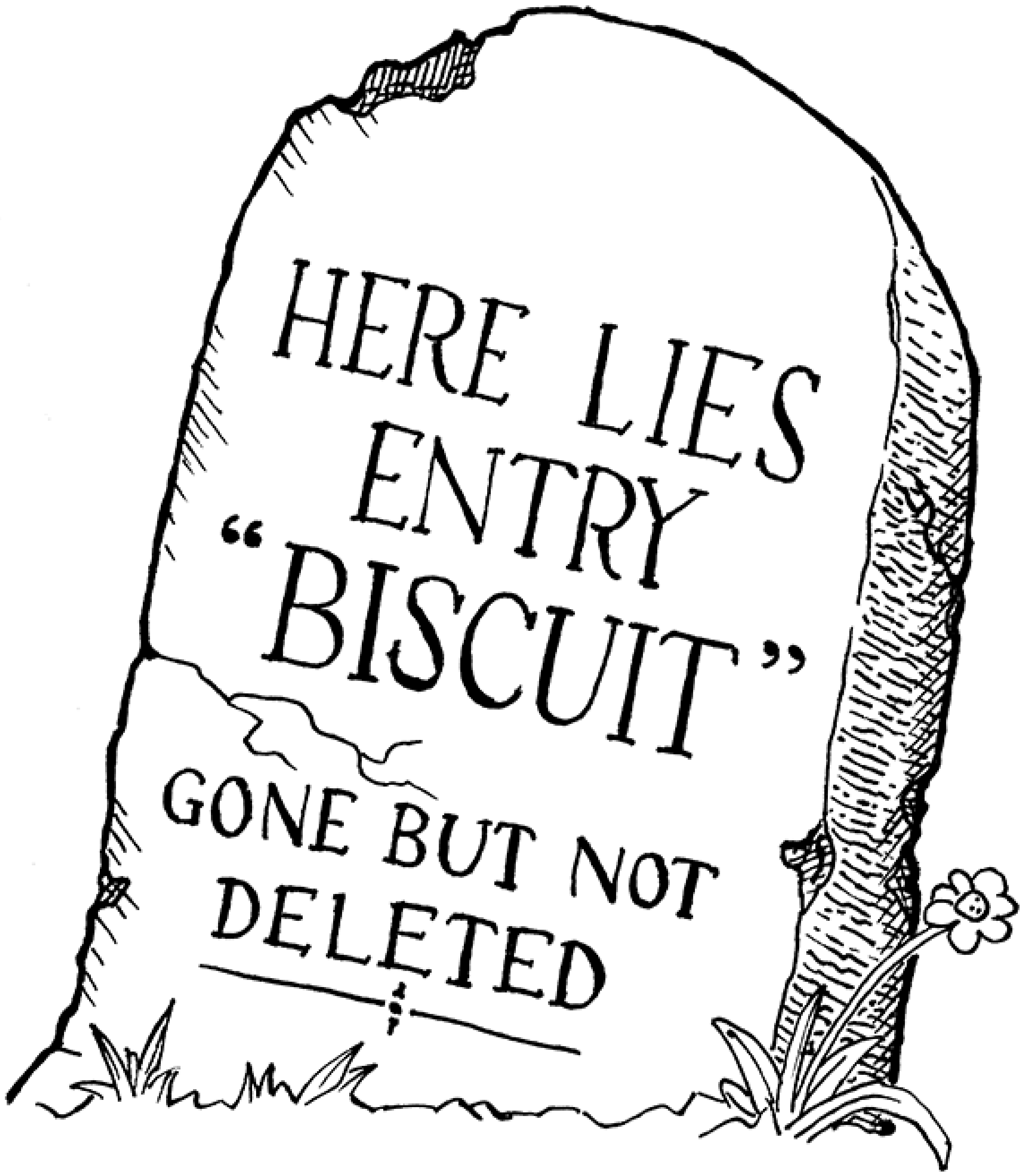
在创建字符串时，我们增加了一点开销来进行驻留。但作为回报，在运行时，字符串的相等操作符要快得多。这样，我们就有了一个全功能的哈希表，可以用来跟踪变量、实例或其它可能出现的任何键值对。

> We also sped up testing strings for equality. This is nice for when the user does `==` on strings. But it's even more critical in a dynamically typed language like Lox where method calls and instance fields are

> looked up by name at runtime. If testing a string for equality is slow, then that means looking up a method by name is slow. And if *that's* slow in your object-oriented language, then *everything* is slow.

我们还加快了测试字符串是否相等的速度。这对于用户在字符串上的==操作是很好的。但在Lox这样的动态类型语言中，这一点更为关键，因为在这种语言中，方法调用和实例属性都是在运行时根据名称查找的。如果测试字符串是否相等是很慢的，那就意味着按名称查找方法也很慢。在面向对象的语言中，如果这一点很慢，那么一切都会变得很慢。

^7: 如果你想了解更多（你应该了解，因为其中一些真的很酷），可以看看“双重哈希(double hashing)”、“布谷鸟哈希(cuckoo hashing)”以及“罗宾汉哈希(Robin Hood hashing)”。 ^8: 哈希函数也被用于密码学。在该领域中，“好”有一个更严格的定义，以避免暴露有关被哈希的数据的细节。值得庆幸的是，我们在本书中不需要担心这些问题。 ^9: 哈希表最初的名称之一是“散列表”，因为它会获取条目并将其分散到整个数组中。“哈希”这个词来自于这样的想法：哈希函数将输入数据分割开来，然后将其组合成一堆，从所有这些比特位中得出一个数字。 ^10: 在clox中，我们只需要支持字符串类型的键。处理其它类型的键不会增加太多复杂性。只要你能比较两个对象是否相等，并把它们简化为比特序列，就很容易将它们用作哈希键。 ^11: 理想的最大负载因子根据哈希函数、冲突处理策略和你将会看到的典型键集而变化。由于像Lox这样的玩具语言没有“真实世界”的数据集，所以很难对其进行优化，所以我随意地选择了75%。当你构建自己的哈希表时，请对其进行基准测试和调整。 ^12: 看起来我们在用==判断两个字符串是否相等。这行不通，对吧？相同的字符串可能会在内存的不同地方有两个副本。不要害怕，聪明的读者。我们会进一步解决这个问题。而且，奇怪的是，是一个哈希表提供了我们需要的工具。 ^13: 使用拉链法时，删除条目就像从链表中删除一个节点一样容易。 ^14:



^15: 在实践中，我们会首先比较两个字符串的哈希码。这样可以快速检测到几乎所有不同的字符串——如果不能，它就不是一个很好的哈希函数。但是，当两个哈希值相同时，我们仍然需要比较字符，以确保没有在不同的字符串上出现哈希冲突。 ^16: 我猜想"intern"是"internal（内部）"的缩写。我认为这个想法是，语言的运行时保留了这些字符串的"内部"集合，而其它字符串可以由用户创建并漂浮在内存中。当你要驻留一个字符串时，你要求运行时将该字符串添加到该内部集合，并返回一个指向该字符串的指针。
不同语言在字符串驻留程度以及对用户的暴露方式上有所不同。Lua会驻留*所有*字符串，这也是clox要做的事情。Lisp、Scheme、Smalltalk、Ruby和其他语言都有一个单独的类似字符串的类型"symbol（符号）"，它是隐式驻留的。（这就是为什么他们说Ruby中的符号"更快"）Java默认会驻留常量字符串，并提供一个API让你显式地驻留传入的任何字符串。

习题

1. > In clox, we happen to only need keys that are strings, so the hash table we built is hardcoded for that key type. If we exposed hash tables to Lox users as a first-class collection, it would be useful to support different kinds of keys.

> Add support for keys of the other primitive types: numbers, Booleans, and `nil`. Later, clox will support user-defined classes. If we want to support keys that are instances of those classes, what kind of complexity does that add?

在clox中，我们碰巧只需要字符串类型的键，所以我们构建的哈希表是针对这种键类型硬编码的。如果我们将哈希表作为一级集合暴露给Lox用户，那么支持不同类型的键就会很有用。

添加对其它基本类型键的支持：数字、布尔值和`nil`。稍后，clox会支持用户定义的类。如果我们想支持那些类的实例作为键，那会增加什么样的复杂性呢？

2. > Hash tables have a lot of knobs you can tweak that affect their performance. You decide whether to use separate chaining or open addressing. Depending on which fork in that road you take, you can tune how many entries are stored in each node, or the probing strategy you use. You control the hash function, load factor, and growth rate.
   >
   > All of this variety wasn't created just to give CS doctoral candidates something to publish theses on: each has its uses in the many varied domains and hardware scenarios where hashing comes into play. Look up a few hash table implementations in different open source systems, research the choices they made, and try to figure out why they did things that way.

   哈希表中有很多你可以调整的旋钮，它们会影响哈希表的性能。你可以决定使用拉链法还是开放地址法。根据你采取的方式，你可以调整每个节点中存储的条目数量，或者是使用的探测策略。你可以控制哈希函数、负载因子和增长率。

   所有这些变化不仅仅是为了给CS博士发表论文（至少这不是它们被创造的唯一原因，是否是主要原因还有待商榷）：在哈希表能发挥作用的许多不同领域和硬件场景中，每一种都有其用途。在不同的开源系统中查找一些哈希表的实现，研究他们所做的选择，并尝试弄清楚他们为什么这样做。

3. > Benchmarking a hash table is notoriously difficult. A hash table implementation may perform well with some keysets and poorly with others. It may work well at small sizes but degrade as it grows, or vice versa. It may choke when deletions are common, but fly when they aren't. Creating benchmarks that accurately represent how your users will use the hash table is a challenge.
   >
   > Write a handful of different benchmark programs to validate our hash table implementation. How does the performance vary between them? Why did you choose the specific test cases you chose?

   对哈希表进行基准测试是出了名的困难。一个哈希表的实现可能在某些键集上表现良好，而在其它键集上则表现不佳。它可能是规模较小时工作得很好，但随着规则扩展会退化，或者正好反过来。当删除操作很多时，它可能被卡住，但删除不常见时，它可能会飞起来。创建能够准确代表用户使用哈希表方式的基准是一项挑战。

   编写一些不同的基准程序来验证我们的哈希表实现。它们之间的表现有什么不同？你为什么选择这些测试用例？

21.全局变量 Global Variables

> If only there could be an invention that bottled up a memory, like scent. And it never faded, and it never got stale. And then, when one wanted it, the bottle could be uncorked, and it would be like living

> the moment all over again.
>
> —— Daphne du Maurier, *Rebecca*

如果有一种发明能把一段记忆装进瓶子里就好了，像香味一样。它永远不会褪色，也不会变质。然后，当一个人想要的时候，可以打开瓶塞，就像重新活在那个时刻一样。（达芙妮-杜穆里埃，《蝴蝶梦》）

> The previous chapter was a long exploration of one big, deep, fundamental computer science data structure. Heavy on theory and concept. There may have been some discussion of big-O notation and algorithms. This chapter has fewer intellectual pretensions. There are no large ideas to learn. Instead, it's a handful of straightforward engineering tasks. Once we've completed them, our virtual machine will support variables.

上一章对一个大的、深入的、基本的计算机科学数据结构进行了长时间的探索。偏重理论和概念。可能有一些关于大O符号和算法的讨论。这一章没有那么多知识分子的自吹自擂。没有什么伟大的思想需要学习。相反，它是一些简单的工程任务。一旦我们完成了这些任务，我们的虚拟机就可以支持变量。

> Actually, it will support only *global* variables. Locals are coming in the next chapter. In jlox, we managed to cram them both into a single chapter because we used the same implementation technique for all variables. We built a chain of environments, one for each scope, all the way up to the top. That was a simple, clean way to learn how to manage state.

事实上，它将只支持*全局*变量。局部变量将在下一章中支持。在jlox中，我们设法将它们塞进了一个章节，因为我们对所有变量都使用了相同的实现技术。我们建立了一个环境链，每个作用域都有一个，一直到顶部作用域。这是学习如何管理状态的一种简单、干净的方法。

> But it's also *slow*. Allocating a new hash table each time you enter a block or call a function is not the road to a fast VM. Given how much code is concerned with using variables, if variables go slow, everything goes slow. For clox, we'll improve that by using a much more efficient strategy for local variables, but globals aren't as easily optimized.

但它也很慢。每次进入一个代码块或调用一个函数时，都要分配一个新的哈希表，这不是通往快速虚拟机的道路。鉴于很多代码都与使用变量有关，如果变量操作缓慢，一切都会变慢。对于clox，我们会通过对局部变量使用更有效的策略来改善这一点，但全局变量不那么容易优化^1。

This is a common meta-strategy in sophisticated language implementations. Often, the same language feature will have multiple implementation techniques, each tuned for different use patterns. For example, JavaScript VMs often have a faster representation for objects that are used more like instances of classes compared to other objects whose set of properties is more freely modified. C and C++ compilers usually have a variety of ways to compile `switch` statements based on the number of cases and how densely packed the case values are.

> A quick refresher on Lox semantics: Global variables in Lox are "late bound", or resolved dynamically. This means you can compile a chunk of code that refers to a global variable before it's defined. As long as the code doesn't *execute* before the definition happens, everything is fine. In practice, that means you can refer to later variables inside the body of functions.

快速复习一下Lox语义：Lox中的全局变量是"后期绑定"的，或者说是动态解析的。这意味着，你可以在全局变量被定义之前，编译引用它的一大块代码。只要代码在定义发生之前没有执行，就没有问题。在实践中，这意味着你可以在函数的主体中引用后面的变量。

```
fun showVariable() {
  print global;
}

var global = "after";
showVariable();
```

> Code like this might seem odd, but it’s handy for defining mutually recursive functions. It also plays nicer with the REPL. You can write a little function in one line, then define the variable it uses in the next.

这样的代码可能看起来很奇怪，但它对于定义相互递归的函数很方便。它与REPL的配合也更好。你可以在一行中编写一个小函数，然后在下一行中定义它使用的变量。

> Local variables work differently. Since a local variable’s declaration *always* occurs before it is used, the VM can resolve them at compile time, even in a simple single-pass compiler. That will let us use a smarter representation for locals. But that’s for the next chapter. Right now, let’s just worry about globals.

局部变量的工作方式不同。因为局部变量的声明总是发生在使用之前，虚拟机可以在编译时解析它们，即使是在简单的单遍编译器中。这让我们可以为局部变量使用更聪明的表示形式。但这是下一章的内容。现在，我们只考虑全局变量。

21.1 Statements

21.1 语句

> Variables come into being using variable declarations, which means now is also the time to add support for statements to our compiler. If you recall, Lox splits statements into two categories. “Declarations” are those statements that bind a new name to a value. The other kinds of statements—control flow, print, etc.—are just called “statements”. We disallow declarations directly inside control flow statements, like this:

变量是通过变量声明产生的，这意味着现在是时候向编译器中添加对语句的支持了。如果你还记得的话，Lox将语句分为两类。“声明”是那些将一个新名称与值绑定的语句。其它类型的语句——控制流、打印等——只被称为“语句”。我们不允许在控制流语句中直接使用声明，像这样：

```
if (monday) var croissant = "yes"; // Error.
```

> Allowing it would raise confusing questions around the scope of the variable. So, like other languages, we prohibit it syntactically by having a separate grammar rule for the subset of statements that *are* allowed inside a control flow body.

允许这种做法会引发围绕变量作用域的令人困惑的问题。因此，像其它语言一样，对于允许出现在控制流主体内的语句子集，我们制定单独的语法规则，从而禁止这种做法。

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | returnStmt
               | whileStmt
               | block ;
```

> Then we use a separate rule for the top level of a script and inside a block.

然后，我们为脚本的顶层和代码块内部使用单独的规则。

```
declaration    → classDecl
               | funDecl
               | varDecl
               | statement ;
```

> The `declaration` rule contains the statements that declare names, and also includes `statement` so that all statement types are allowed. Since `block` itself is in `statement`, you can put declarations inside a control flow construct by nesting them inside a block.

`declaration`包含声明名称的语句，也包含`statement`规则，这样所有的语句类型都是允许的。因为`block`本身就在`statement`中，你可以通过将声明嵌套在代码块中的方式将它们放在控制流结构中^2。

> In this chapter, we'll cover only a couple of statements and one declaration.

在本章中，我们只讨论几个语句和一个声明。

```
statement      → exprStmt
               | printStmt ;

declaration    → varDecl
               | statement ;
```

> Up to now, our VM considered a "program" to be a single expression since that's all we could parse and compile. In a full Lox implementation, a program is a sequence of declarations. We're ready to support that now.

到目前为止，我们的虚拟机都认为"程序"是一个表达式，因为我们只能解析和编译一条表达式。在完整的Lox实现中，程序是一连串的声明。我们现在已经准备要支持它了。

compiler.c，在compile()方法中替换2行：

```
  advance();
  // 替换部分开始
  while (!match(TOKEN_EOF)) {
    declaration();
  }
  // 替换部分结束
  endCompiler();
```

> We keep compiling declarations until we hit the end of the source file. We compile a single declaration using this:

我们会一直编译声明语句，直到到达源文件的结尾。我们用这个方法来编译一条声明语句：

compiler.c，在expression()方法后添加代码：

```
static void declaration() {
  statement();
}
```

> We'll get to variable declarations later in the chapter, so for now, we simply forward to `statement()`.

我们将在本章后面讨论变量声明，所以现在，我们直接使用`statement()`。

compiler.c，在declaration()方法后添加代码：

```
static void statement() {
  if (match(TOKEN_PRINT)) {
    printStatement();
  }
}
```

> Blocks can contain declarations, and control flow statements can contain other statements. That means these two functions will eventually be recursive. We may as well write out the forward declarations now.

代码块可以包含声明，而控制流语句可以包含其它语句。这意味着这两个函数最终是递归的。我们不妨现在就把前置声明写出来。

compiler.c，在expression()方法后添加代码：

```
static void expression();
// 新增部分开始
static void statement();
static void declaration();
// 新增部分结束
static ParseRule* getRule(TokenType type);
```

> ### 21.1.1 Print statements

21.1.1 Print语句

> We have two statement types to support in this chapter. Let's start with `print` statements, which begin, naturally enough, with a `print` token. We detect that using this helper function:

在本章中，我们有两种语句类型需要支持。我们从`print`语句开始，它自然是以`print`标识开头的。我们使用这个辅助函数来检测：

compiler.c，在consume()方法后添加代码：

```
static bool match(TokenType type) {
  if (!check(type)) return false;
  advance();
  return true;
}
```

> You may recognize it from jlox. If the current token has the given type, we consume the token and return `true`. Otherwise we leave the token alone and return `false`. This helper function is implemented in terms of this other helper:

你可能看出它是从jlox来的。如果当前的标识是指定类型，我们就消耗该标识并返回`true`。否则，我们就不处理该标识并返回`false`。这个辅助函数是通过另一个辅助函数实现的：

compiler.c，在consume()方法后添加代码：

```
static bool check(TokenType type) {
  return parser.current.type == type;
}
```

> The `check()` function returns `true` if the current token has the given type. It seems a little silly to wrap this in a function, but we'll use it more later, and I think short verb-named functions like this make the parser easier to read.

如果当前标识符合给定的类型，`check()`函数返回`true`。将它封装在一个函数中似乎有点傻，但我们以后会更多地使用它，而且我们认为像这样简短的动词命名的函数使解析器更容易阅读^3。

> If we did match the `print` token, then we compile the rest of the statement here:

如果我们确实匹配到了`print`标识，那么我们在下面这个方法中编译该语句的剩余部分：

compiler.c，在expression()方法后添加代码：

```
static void printStatement() {
  expression();
  consume(TOKEN_SEMICOLON, "Expect ';' after value.");
  emitByte(OP_PRINT);
}
```

> A `print` statement evaluates an expression and prints the result, so we first parse and compile that expression. The grammar expects a semicolon after that, so we consume it. Finally, we emit a new instruction to print the result.

`print`语句会对表达式求值并打印出结果，所以我们首先解析并编译这个表达式。语法要求在表达式之后有一个分号，所以我们消耗一个分号标识。最后，我们生成一条新指令来打印结果。

chunk.h，在枚举OpCode中添加代码：

```
  OP_NEGATE,
  // 新增部分开始
  OP_PRINT,
  // 新增部分结束
  OP_RETURN,
```

> At runtime, we execute this instruction like so:

在运行时，我们这样执行这条指令：

vm.c，在run()方法中添加代码：

```
        break;
      // 新增部分开始
      case OP_PRINT: {
        printValue(pop());
        printf("\n");
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> When the interpreter reaches this instruction, it has already executed the code for the expression, leaving the result value on top of the stack. Now we simply pop and print it.

当解释器到达这条指令时，它已经执行了表达式的代码，将结果值留在了栈顶。现在我们只需要弹出该值并打印。

> Note that we don't push anything else after that. This is a key difference between expressions and statements in the VM. Every bytecode instruction has a **stack effect** that describes how the instruction modifies the stack. For example, `OP_ADD` pops two values and pushes one, leaving the stack one element smaller than before.

请注意，在此之后我们不会再向栈中压入任何内容。这是虚拟机中表达式和语句之间的一个关键区别。每个字节码指令都有**堆栈效应**，这个值用于描述指令如何修改堆栈内容。例如，`OP_ADD`会弹出两个值并压入一个值，使得栈中比之前少了一个元素^4。

> You can sum the stack effects of a series of instructions to get their total effect. When you add the stack effects of the series of instructions compiled from any complete expression, it will total one. Each expression leaves one result value on the stack.

你可以把一系列指令的堆栈效应相加，得到它们的总体效应。如果把从任何一个完整的表达式中编译得到的一系列指令的堆栈效应相加，其总数是1。每个表达式会在栈中留下一个结果值。

> The bytecode for an entire statement has a total stack effect of zero. Since a statement produces no values, it ultimately leaves the stack unchanged, though it of course uses the stack while it's doing its thing. This is important because when we get to control flow and looping, a program might execute a long series of statements. If each statement grew or shrank the stack, it might eventually overflow or underflow.

整个语句对应字节码的总堆栈效应为0。因为语句不产生任何值，所以它最终会保持堆栈不变，尽管它在执行自己的操作时难免会使用堆栈。这一点很重要，因为等我们涉及到控制流和循环时，一个程序可能会执行一长串的语句。如果每条语句都增加或减少堆栈，最终就可能会溢出或下溢。

> While we're in the interpreter loop, we should delete a bit of code.

在解释器循环中，我们应该删除一些代码。

vm.c，在run()方法中替换2行：

```
      case OP_RETURN: {
        // 替换部分开始
        // Exit interpreter.
        // 替换部分结束
        return INTERPRET_OK;
```

> When the VM only compiled and evaluated a single expression, we had some temporary code in `OP_RETURN` to output the value. Now that we have statements and `print`, we don't need that anymore. We're one step closer to the complete implementation of clox.

当虚拟机只编译和计算一条表达式时，我们在`OP_RETURN`中使用一些临时代码来输出值。现在我们已经有了语句和`print`，就不再需要这些了。我们离clox的完全实现又近了一步^5。

> As usual, a new instruction needs support in the disassembler.

像往常一样，一条新指令需要反汇编程序的支持。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_NEGATE", offset);
    // 新增部分开始
    case OP_PRINT:
      return simpleInstruction("OP_PRINT", offset);
    // 新增部分结束
    case OP_RETURN:
```

> That's our `print` statement. If you want, give it a whirl:

这就是我们的`print`语句。如果你愿意，可以试一试：

```
print 1 + 2;
print 3 * 4;
```

> Exciting! OK, maybe not thrilling, but we can build scripts that contain as many statements as we want now, which feels like progress.

令人兴奋！好吧，也许没有那么激动人心，但是我们现在可以构建包含任意多语句的脚本，这感觉是一种进步。

> 21.1.2 Expression statements

21.1.2 表达式语句

> Wait until you see the next statement. If we *don't* see a `print` keyword, then we must be looking at an expression statement.

等待，直到你看到下一条语句。如果没有看到`print`关键字，那么我们看到的一定是一条表达式语句。

compiler.c，在statement()方法中添加代码：

```
    printStatement();
  // 新增部分开始
  } else {
    expressionStatement();
  // 新增部分结束
  }
```

> It's parsed like so:

它是这样解析的：

compiler.c，在expression()方法后添加代码：

```
static void expressionStatement() {
  expression();
  consume(TOKEN_SEMICOLON, "Expect ';' after expression.");
  emitByte(OP_POP);
}
```

> An "expression statement" is simply an expression followed by a semicolon. They're how you write an expression in a context where a statement is expected. Usually, it's so that you can call a function or evaluate an assignment for its side effect, like this:

“表达式语句”就是一个表达式后面跟着一个分号。这是在需要语句的上下文中写表达式的方式。通常来说，这样你就可以调用函数或执行赋值操作以触发其副作用，像这样：

```
brunch = "quiche";
eat(brunch);
```

> Semantically, an expression statement evaluates the expression and discards the result. The compiler directly encodes that behavior. It compiles the expression, and then emits an `OP_POP` instruction.

从语义上说，表达式语句会对表达式求值并丢弃结果。编译器直接对这种行为进行编码。它会编译表达式，然后生成一条`OP_POP`指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_FALSE,
  // 新增部分开始
  OP_POP,
  // 新增部分结束
  OP_EQUAL,
```

> As the name implies, that instruction pops the top value off the stack and forgets it.

顾名思义，该指令会弹出栈顶的值并将其遗弃。

vm.c，在run()方法中添加代码：

```
      case OP_FALSE: push(BOOL_VAL(false)); break;
      // 新增部分开始
      case OP_POP: pop(); break;
      // 新增部分结束
      case OP_EQUAL: {
```

> We can disassemble it too.

我们也可以对它进行反汇编。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_FALSE", offset);
    // 新增部分开始
    case OP_POP:
      return simpleInstruction("OP_POP", offset);
    // 新增部分结束
    case OP_EQUAL:
```

> Expression statements aren't very useful yet since we can't create any expressions that have side effects, but they'll be essential when we add functions later. The majority of statements in real-world code in languages like C are expression statements.

表达式语句现在还不是很有用，因为我们无法创建任何有副作用的表达式，但等我们后面添加函数时，它们将是必不可少的。在像C这样的真正语言中，大部分语句都是表达式语句^6。

> ## 21.1.3 Error synchronization

21.1.3 错误同步

> While we're getting this initial work done in the compiler, we can tie off a loose end we left several chapters back. Like jlox, clox uses panic mode error recovery to minimize the number of cascaded compile errors that it reports. The compiler exits panic mode when it reaches a synchronization point. For Lox, we chose statement boundaries as that point. Now that we have statements, we can implement synchronization.

当我们在编译器中完成这些初始化工作时，我们可以把前几章遗留的一个小尾巴处理一下。与jlox一样，clox也使用了恐慌模式下的错误恢复来减少它所报告的级联编译错误。当编译器到达同步点时，就退出恐慌模式。对于Lox来说，我们选择语句边界作为同步点。现在我们有了语句，就可以实现同步了。

compiler.c，在declaration()方法中添加代码：

```
    statement();
    // 新增部分开始
    if (parser.panicMode) synchronize();
    // 新增部分结束
  }
```

> If we hit a compile error while parsing the previous statement, we enter panic mode. When that happens, after the statement we start synchronizing.

如果我们在解析前一条语句时遇到编译错误，我们就会进入恐慌模式。当这种情况发生时，我们会在这条语句之后开始同步。

compiler.c，在printStatement()方法后添加代码：

```
  static void synchronize() {
    parser.panicMode = false;

    while (parser.current.type != TOKEN_EOF) {
      if (parser.previous.type == TOKEN_SEMICOLON) return;
      switch (parser.current.type) {
        case TOKEN_CLASS:
        case TOKEN_FUN:
        case TOKEN_VAR:
        case TOKEN_FOR:
        case TOKEN_IF:
        case TOKEN_WHILE:
```

```
      case TOKEN_PRINT:
      case TOKEN_RETURN:
        return;

      default:
        ; // Do nothing.
    }

    advance();
  }
}
```

> We skip tokens indiscriminately until we reach something that looks like a statement boundary. We recognize the boundary by looking for a preceding token that can end a statement, like a semicolon. Or we'll look for a subsequent token that begins a statement, usually one of the control flow or declaration keywords.

我们会不分青红皂白地跳过标识，直到我们到达一个看起来像是语句边界的位置。我们识别边界的方式包括，查找可以结束一条语句的前驱标识，如分号；或者我们可以查找能够开始一条语句的后续标识，通常是控制流或声明语句的关键字之一。

> ## 21.2 Variable Declarations

21.2 变量声明

> Merely being able to *print* doesn't win your language any prizes at the programming language fair, so let's move on to something a little more ambitious and get variables going. There are three operations we need to support:

仅仅能够*打印*并不能为你的语言在编程语言博览会上赢得任何奖项，所以让我们继续做一些更有野心的事，让变量发挥作用。我们需要支持三种操作：

- > Declaring a new variable using a `var` statement.

  使用`var`语句声明一个新变量

- > Accessing the value of a variable using an identifier expression.

  使用标识符表达式访问一个变量的值

- > Storing a new value in an existing variable using an assignment expression.

  使用赋值表达式将一个新的值存储在现有的变量中

> We can't do either of the last two until we have some variables, so we start with declarations.

等我们有了变量以后，才能做后面两件事，所以我们从声明开始。

compiler.c，在declaration()方法中替换1行：

```
static void declaration() {
  // 替换部分开始
  if (match(TOKEN_VAR)) {
    varDeclaration();
  } else {
    statement();
  }
  // 替换部分结束
  if (parser.panicMode) synchronize();
```

> The placeholder parsing function we sketched out for the declaration grammar rule has an actual production now. If we match a `var` token, we jump here:

我们为声明语法规则建立的占位解析函数现在已经有了实际的生成式。如果我们匹配到一个`var`标识，就跳转到这里：

compiler.c，在expression()方法后添加代码：

```
static void varDeclaration() {
  uint8_t global = parseVariable("Expect variable name.");

  if (match(TOKEN_EQUAL)) {
    expression();
  } else {
    emitByte(OP_NIL);
  }
  consume(TOKEN_SEMICOLON,
          "Expect ';' after variable declaration.");

  defineVariable(global);
}
```

> The keyword is followed by the variable name. That's compiled by `parseVariable()`, which we'll get to in a second. Then we look for an `=` followed by an initializer expression. If the user doesn't initialize the variable, the compiler implicitly initializes it to `nil` by emitting an `OP_NIL` instruction. Either way, we expect the statement to be terminated with a semicolon.

关键字后面跟着变量名。它是由`parseVariable()`编译的，我们马上就会讲到。然后我们会寻找一个`=`，后跟初始化表达式。如果用户没有初始化变量，编译器会生成`OP_NIL`指令隐式地将其初始化为`nil`^7。无论哪种方式，我们都希望语句以分号结束。

> There are two new functions here for working with variables and identifiers. Here is the first:

这里有两个新函数用于处理变量和标识符。下面是第一个：

compiler.c，在parsePrecedence()方法后添加代码：

```
static void parsePrecedence(Precedence precedence);
// 新增部分开始
static uint8_t parseVariable(const char* errorMessage) {
  consume(TOKEN_IDENTIFIER, errorMessage);
  return identifierConstant(&parser.previous);
}
// 新增部分结束
```

> It requires the next token to be an identifier, which it consumes and sends here:

它要求下一个标识是一个标识符，它会消耗该标识并发送到这里：

compiler.c，在parsePrecedence()方法后添加代码：

```
static void parsePrecedence(Precedence precedence);
// 新增部分开始
static uint8_t identifierConstant(Token* name) {
  return makeConstant(OBJ_VAL(copyString(name->start,
                                         name->length)));
}
// 新增部分结束
```

> This function takes the given token and adds its lexeme to the chunk's constant table as a string. It then returns the index of that constant in the constant table.

这个函数接受给定的标识，并将其词素作为一个字符串添加到字节码块的常量表中。然后，它会返回该常量在常量表中的索引。

> Global variables are looked up *by name* at runtime. That means the VM—the bytecode interpreter loop—needs access to the name. A whole string is too big to stuff into the bytecode stream as an operand. Instead, we store the string in the constant table and the instruction then refers to the name by its index in the table.

全局变量在运行时是按*名称*查找的。这意味着虚拟机（字节码解释器循环）需要访问该名称。整个字符串太大，不能作为操作数塞进字节码流中。相反，我们将字符串存储到常量表中，然后指令通过该名称在表中的索引来引用它。

> This function returns that index all the way to `varDeclaration()` which later hands it over to here:

这个函数会将索引一直返回给`varDeclaration()`，随后又将其传递到这里：

compiler.c，在parseVariable()方法后添加代码：

```
static void defineVariable(uint8_t global) {
  emitBytes(OP_DEFINE_GLOBAL, global);
}
```

> This outputs the bytecode instruction that defines the new variable and stores its initial value. The index of the variable's name in the constant table is the instruction's operand. As usual in a stack-based VM, we emit this instruction last. At runtime, we execute the code for the variable's initializer first. That leaves the value on the stack. Then this instruction takes that value and stores it away for later.

它会输出字节码指令，用于定义新变量并存储其初始化值。变量名在常量表中的索引是该指令的操作数。在基于堆栈的虚拟机中，我们通常是最后发出这条指令。在运行时，我们首先执行变量初始化器的代码，将值留在栈中。然后这条指令会获取该值并保存起来，以供日后使用^8。

> Over in the runtime, we begin with this new instruction:

在运行时，我们从这条新指令开始：

chunk.h，在枚举OpCode中添加代码：

```
  OP_POP,
  // 新增部分开始
  OP_DEFINE_GLOBAL,
  // 新增部分结束
  OP_EQUAL,
```

> Thanks to our handy-dandy hash table, the implementation isn't too hard.

多亏了我们方便的哈希表，实现起来并不太难。

vm.c，在run()方法中添加代码：

```
      case OP_POP: pop(); break;
      // 新增部分开始
      case OP_DEFINE_GLOBAL: {
        ObjString* name = READ_STRING();
        tableSet(&vm.globals, name, peek(0));
        pop();
        break;
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> We get the name of the variable from the constant table. Then we take the value from the top of the stack and store it in a hash table with that name as the key.

我们从常量表中获取变量的名称，然后我们从栈顶获取值，并以该名称为键将其存储在哈希表中^9。

> This code doesn't check to see if the key is already in the table. Lox is pretty lax with global variables and lets you redefine them without error. That's useful in a REPL session, so the VM supports that by simply overwriting the value if the key happens to already be in the hash table.

这段代码并没有检查键是否已经在表中。Lox对全局变量的处理非常宽松，允许你重新定义它们而且不会出错。这在REPL会话中很有用，如果键恰好已经在哈希表中，虚拟机通过简单地覆盖值来支持这一点。

> There’s another little helper macro:

还有另一个小的辅助宏：

vm.c，在run()方法中添加代码：

```
#define READ_CONSTANT() (vm.chunk->constants.values[READ_BYTE()])
// 新增部分开始
#define READ_STRING() AS_STRING(READ_CONSTANT())
// 新增部分结束
#define BINARY_OP(valueType, op) \
```

> It reads a one-byte operand from the bytecode chunk. It treats that as an index into the chunk’s constant table and returns the string at that index. It doesn’t check that the value *is* a string—it just indiscriminately casts it. That’s safe because the compiler never emits an instruction that refers to a non-string constant.

它从字节码块中读取一个1字节的操作数。它将其视为字节码块的常量表的索引，并返回该索引处的字符串。它不检查该值是否是字符串——它只是不加区分地进行类型转换。这是安全的，因为编译器永远不会发出引用非字符串常量的指令。

> Because we care about lexical hygiene, we also undefine this macro at the end of the interpret function.

因为我们关心词法卫生，所以在解释器函数的末尾也取消了这个宏的定义。

vm.c，在run()方法中添加代码：

```
#undef READ_CONSTANT
// 新增部分开始
#undef READ_STRING
// 新增部分结束
#undef BINARY_OP
```

> I keep saying “the hash table”, but we don’t actually have one yet. We need a place to store these globals. Since we want them to persist as long as clox is running, we store them right in the VM.

我一直在说“哈希表”，但实际上我们还没有哈希表。我们需要一个地方来存储这些全局变量。因为我们希望它们在clox运行期间一直存在，所以我们将它们之间存储在虚拟机中。

vm.h，在结构体VM中添加代码：

```
  Value* stackTop;
  // 新增部分开始
  Table globals;
  // 新增部分结束
  Table strings;
```

> As we did with the string table, we need to initialize the hash table to a valid state when the VM boots up.

正如我们对字符串表所做的那样，我们需要在虚拟机启动时将哈希表初始化为有效状态。

vm.c，在initVM()方法中添加代码：

```
  vm.objects = NULL;
  // 新增部分开始
  initTable(&vm.globals);
  // 新增部分结束
  initTable(&vm.strings);
```

> And we tear it down when we exit.

当我们退出时，就将其删掉^10。

vm.c，在freeVM()方法中添加代码：

```
void freeVM() {
  // 新增部分开始
  freeTable(&vm.globals);
  // 新增部分结束
  freeTable(&vm.strings);
```

> As usual, we want to be able to disassemble the new instruction too.

跟往常一样，我们也希望能够对新指令进行反汇编。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_POP", offset);
    // 新增部分开始
    case OP_DEFINE_GLOBAL:
      return constantInstruction("OP_DEFINE_GLOBAL", chunk,
                                 offset);
    // 新增部分结束
    case OP_EQUAL:
```

> And with that, we can define global variables. Not that users can *tell* that they've done so, because they can't actually *use* them. So let's fix that next.

有了这个，我们就可以定义全局变量了。但用户并不能说他们可以定义全局变量，因为他们实际上还不能使用这些变量。所以，接下来我们解决这个问题。

21.3 Reading Variables

21.3 读取变量

> As in every programming language ever, we access a variable’s value using its name. We hook up identifier tokens to the expression parser here:

像所有编程语言中一样，我们使用变量的名称来访问它的值。我们在这里将标识符和表达式解析器进行挂钩：

compiler.c，替换1行：

```
  [TOKEN_LESS_EQUAL]    = {NULL,     binary, PREC_COMPARISON},
  // 替换部分开始
  [TOKEN_IDENTIFIER]    = {variable, NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_STRING]        = {string,   NULL,   PREC_NONE},
```

> That calls this new parser function:

这里调用了这个新解析器函数：

compiler.c，在string()方法后添加代码：

```
static void variable() {
  namedVariable(parser.previous);
}
```

> Like with declarations, there are a couple of tiny helper functions that seem pointless now but will become more useful in later chapters. I promise.

和声明一样，这里有几个小的辅助函数，现在看起来毫无意义，但在后面的章节中会变得更加有用。我保证。

compiler.c，在string()方法后添加代码：

```
static void namedVariable(Token name) {
  uint8_t arg = identifierConstant(&name);
  emitBytes(OP_GET_GLOBAL, arg);
}
```

> This calls the same `identifierConstant()` function from before to take the given identifier token and add its lexeme to the chunk’s constant table as a string. All that remains is to emit an instruction that loads the global variable with that name. Here’s the instruction:

这里会调用与之前相同的`identifierConstant()`函数，以获取给定的标识符标识，并将其词素作为字符串添加到字节码块的常量表中。剩下的工作就是生成一条指令，加载具有该名称的全局变量。下面是这个指令：

chunk.h，在枚举OpCode中添加代码：

```
  OP_POP,
  // 新增部分开始
  OP_GET_GLOBAL,
  // 新增部分结束
  OP_DEFINE_GLOBAL,
```

> Over in the interpreter, the implementation mirrors `OP_DEFINE_GLOBAL`.

在解释器中，它的实现是`OP_DEFINE_GLOBAL`的镜像操作。

vm.c，在run()方法中添加代码：

```
      case OP_POP: pop(); break;
      // 新增部分开始
      case OP_GET_GLOBAL: {
        ObjString* name = READ_STRING();
        Value value;
        if (!tableGet(&vm.globals, name, &value)) {
          runtimeError("Undefined variable '%s'.", name->chars);
          return INTERPRET_RUNTIME_ERROR;
        }
        push(value);
        break;
      }
      // 新增部分结束
      case OP_DEFINE_GLOBAL: {
```

> We pull the constant table index from the instruction’s operand and get the variable name. Then we use that as a key to look up the variable’s value in the globals hash table.

我们从指令操作数中提取常量表索引并获得变量名称。然后我们使用它作为键，在全局变量哈希表中查找变量的值。

> If the key isn’t present in the hash table, it means that global variable has never been defined. That’s a runtime error in Lox, so we report it and exit the interpreter loop if that happens. Otherwise, we take the value and push it onto the stack.

如果该键不在哈希表中，就意味着这个全局变量从未被定义过。这在Lox中是运行时错误，所以如果发生这种情况，我们要报告错误并退出解释器循环。否则，我们获取该值并将其压入栈中。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_POP", offset);
    // 新增部分开始
    case OP_GET_GLOBAL:
      return constantInstruction("OP_GET_GLOBAL", chunk, offset);
    // 新增部分结束
    case OP_DEFINE_GLOBAL:
```

> A little bit of disassembling, and we're done. Our interpreter is now able to run code like this:

稍微反汇编一下，就完成了。我们的解释器现在可以运行这样的代码了：

```
var beverage = "cafe au lait";
var breakfast = "beignets with " + beverage;
print breakfast;
```

> There's only one operation left.

只剩一个操作了。

> ## 21.4 Assignment

21.4 赋值

> Throughout this book, I've tried to keep you on a fairly safe and easy path. I don't avoid hard *problems*, but I try to not make the *solutions* more complex than they need to be. Alas, other design choices in our bytecode compiler make assignment annoying to implement.

在这本书中，我一直试图让你走在一条相对安全和简单的道路上。我并不回避困难的*问题*，但是我尽量不让解决方案过于复杂。可惜的是，我们的字节码编译器中的其它设计选择使得赋值的实现变得很麻烦^11。

> Our bytecode VM uses a single-pass compiler. It parses and generates bytecode on the fly without any intermediate AST. As soon as it recognizes a piece of syntax, it emits code for it. Assignment doesn't naturally fit that. Consider:

我们的字节码虚拟机使用的是单遍编译器。它在不需要任何中间AST的情况下，动态地解析并生成字节码。一旦它识别出某个语法，它就会生成对应的字节码。赋值操作天然不符合这一点。请考虑一下：

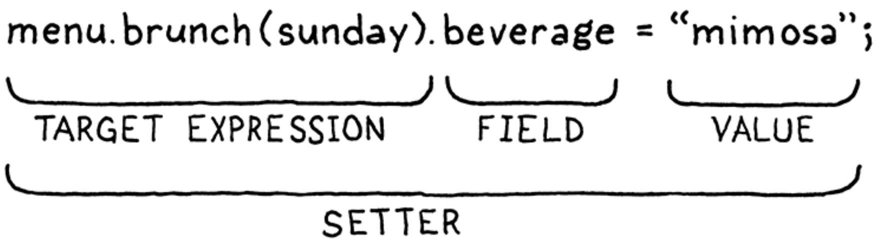
```
menu.brunch(sunday).beverage = "mimosa";
```

> In this code, the parser doesn't realize `menu.brunch(sunday).beverage` is the target of an assignment and not a normal expression until it reaches `=`, many tokens after the first `menu`. By then, the compiler has already emitted bytecode for the whole thing.

在这段代码中，直到解析器遇见`=`（第一个`menu`之后很多个标识），它才能意识到`menu.brunch(sunday).beverage`是赋值操作的目标，而不是常规的表达式。到那时，编译器已经为整个代码生成字节码了。

> The problem is not as dire as it might seem, though. Look at how the parser sees that example:

不过，这个问题并不像看上去那么可怕。看看解析器是如何处理这个例子的：



> Even though the `.beverage` part must not be compiled as a get expression, everything to the left of the `.` is an expression, with the normal expression semantics. The `menu.brunch(sunday)` part can be compiled and executed as usual.

尽管`.beverage`部分无法被编译为一个get表达式，`.`左侧的其它部分是一个表达式，有着正常的表达式语义。`menu.brunch(sunday)`部分可以像往常一样编译和执行。

> Fortunately for us, the only semantic differences on the left side of an assignment appear at the very right-most end of the tokens, immediately preceding the `=`. Even though the receiver of a setter may be an arbitrarily long expression, the part whose behavior differs from a get expression is only the trailing identifier, which is right before the `=`. We don't need much lookahead to realize `beverage` should be compiled as a set expression and not a getter.

幸运的是，赋值语句左侧部分唯一的语义差异在于其最右侧的标识，紧挨着`=`之前。尽管setter的接收方可能是一个任意长的表达式，但与get表达式不同的部分在于尾部的标识符，它就在`=`之前。我们不需要太多的前瞻就可以意识到`beverage`应该被编译为set表达式而不是getter。

> Variables are even easier since they are just a single bare identifier before an `=`. The idea then is that right *before* compiling an expression that can also be used as an assignment target, we look for a subsequent `=` token. If we see one, we compile it as an assignment or setter instead of a variable access or getter.

变量就更简单了，因为它们在`=`之前就是一个简单的标识符。那么我们的想法是，在编译一个也可以作为赋值目标的表达式*之前*，我们会寻找随后的`=`标识。如果我们看到了，那表明我们将其一个赋值表达式或setter来编译，而不是变量访问或getter。

> We don't have setters to worry about yet, so all we need to handle are variables.

我们还不需要考虑setter，所以我们需要处理的就是变量。

compiler.c，在namedVariable()方法中替换1行：

```
  uint8_t arg = identifierConstant(&name);
  // 替换部分开始
  if (match(TOKEN_EQUAL)) {
    expression();
    emitBytes(OP_SET_GLOBAL, arg);
  } else {
    emitBytes(OP_GET_GLOBAL, arg);
  }
  // 替换部分结束
}
```

> In the parse function for identifier expressions, we look for an equals sign after the identifier. If we find one, instead of emitting code for a variable access, we compile the assigned value and then emit an assignment instruction.

在标识符表达式的解析函数中，我们会查找标识符后面的等号。如果找到了，我们就不会生成变量访问的代码，我们会编译所赋的值，然后生成一个赋值指令。

> That’s the last instruction we need to add in this chapter.

这就是我们在本章中需要添加的最后一条指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_DEFINE_GLOBAL,
  // 新增部分开始
  OP_SET_GLOBAL,
  // 新增部分结束
  OP_EQUAL,
```

> As you’d expect, its runtime behavior is similar to defining a new variable.

如你所想，它的运行时行为类似于定义一个新变量。

vm.c，在run()方法中添加代码^12 *：*

```
      }
      // 新增部分开始
      case OP_SET_GLOBAL: {
        ObjString* name = READ_STRING();
        if (tableSet(&vm.globals, name, peek(0))) {
          tableDelete(&vm.globals, name);
          runtimeError("Undefined variable '%s'.", name->chars);
          return INTERPRET_RUNTIME_ERROR;
        }
        break;
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> The main difference is what happens when the key doesn’t already exist in the globals hash table. If the variable hasn’t been defined yet, it’s a runtime error to try to assign to it. Lox doesn’t do implicit variable declaration.

主要的区别在于，当键在全局变量哈希表中不存在时会发生什么。如果这个变量还没有定义，对其进行赋值就是一个运行时错误。Lox不做隐式的变量声明。

> The other difference is that setting a variable doesn’t pop the value off the stack. Remember, assignment is an expression, so it needs to leave that value there in case the assignment is nested inside some larger expression.

另一个区别是，设置变量并不会从栈中弹出值。记住，赋值是一个表达式，所以它需要把这个值保留在那里，以防赋值嵌套在某个更大的表达式中。

> Add a dash of disassembly:

加一点反汇编代码：

debug.c，在disassembleInstruction()方法中添加代码：

```
      return constantInstruction("OP_DEFINE_GLOBAL", chunk,
                                 offset);
    // 新增部分开始
    case OP_SET_GLOBAL:
      return constantInstruction("OP_SET_GLOBAL", chunk, offset);
    // 新增部分结束
    case OP_EQUAL:
```

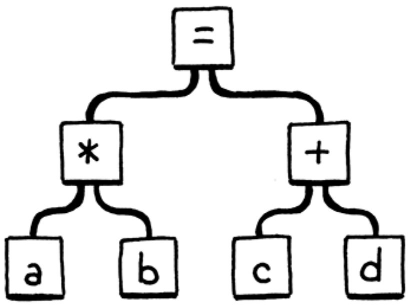
> So we’re done, right? Well . . . not quite. We’ve made a mistake! Take a gander at:

我们已经完成了，是吗？嗯……不完全是。我们犯了一个错误！看一下这个：

```
a * b = c + d;
```

> According to Lox’s grammar, `=` has the lowest precedence, so this should be parsed roughly like:

根据Lox语法，`=`的优先级最低，所以这大致应该解析为：



> Obviously, `a * b` isn’t a valid assignment target, so this should be a syntax error. But here’s what our parser does:

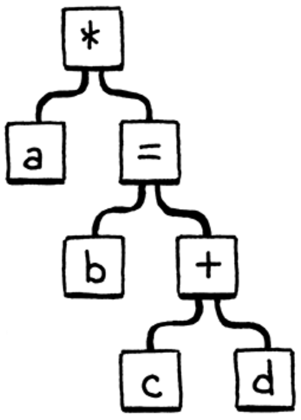
显然，`a*b`不是一个有效的赋值目标^13，所以这应该是一个语法错误。但我们的解析器是这样的：

> 1. First, `parsePrecedence()` parses `a` using the `variable()` prefix parser.
> 2. After that, it enters the infix parsing loop.
> 3. It reaches the `*` and calls `binary()`.
> 4. That recursively calls `parsePrecedence()` to parse the right-hand operand.
> 5. That calls `variable()` again for parsing `b`.
> 6. Inside that call to `variable()`, it looks for a trailing `=`. It sees one and thus parses the rest of the line as an assignment.

1. 首先，parsePrecedence()使用variable()前缀解析器解析a。
2. 之后，会进入中缀解析循环。
3. 达到*，并调用binary()。
4. 递归地调用parsePrecedence()解析右操作数。
5. 再次调用variable()解析b。
6. 在对variable()的调用中，会查找尾部的=。它看到了，因此会将本行的其余部分解析为一个赋值表达式。

> In other words, the parser sees the above code like:

换句话说，解析器将上面的代码看作：



> We've messed up the precedence handling because variable() doesn't take into account the precedence of the surrounding expression that contains the variable. If the variable happens to be the right-hand side of an infix operator, or the operand of a unary operator, then that containing expression is too high precedence to permit the =.

我们搞砸了优先级处理，因为variable()没有考虑包含变量的外围表达式的优先级。如果变量恰好是中缀操作符的右操作数，或者是一元操作符的操作数，那么这个包含表达式的优先级太高，不允许使用=。

> To fix this, variable() should look for and consume the = only if it's in the context of a low-precedence expression. The code that knows the current precedence is, logically enough, parsePrecedence(). The variable() function doesn't need to know the actual level. It just cares that the precedence is low enough to allow assignment, so we pass that fact in as a Boolean.

为了解决这个问题，variable()应该只在低优先级表达式的上下文中寻找并使用=。从逻辑上讲，知道当前优先级的代码是parsePrecedence()。variable()函数不需要知道实际的级别。它只关心优先级是否低到允许赋值表达式，所以我们把这个情况以布尔值传入。

compiler.c，在parsePrecedence()方法中替换1行：

```
    error("Expect expression.");
    return;
  }
  // 替换部分开始
  bool canAssign = precedence <= PREC_ASSIGNMENT;
  prefixRule(canAssign);
  // 替换部分结束
  while (precedence <= getRule(parser.current.type)->precedence) {
```

> Since assignment is the lowest-precedence expression, the only time we allow an assignment is when parsing an assignment expression or top-level expression like in an expression statement. That flag makes its way to the parser function here:

因为赋值是最低优先级的表达式，只有在解析赋值表达式或如表达式语句等顶层表达式时，才允许出现赋值。这个标志会被传入这个解析器函数：

compiler.c，在variable()函数中替换3行：

```
static void variable(bool canAssign) {
  namedVariable(parser.previous, canAssign);
}
```

> Which passes it through a new parameter:

通过一个新参数透传该值：

compiler.c，在namedVariable()方法中替换1行：

```
// 替换部分开始
static void namedVariable(Token name, bool canAssign) {
  // 替换部分结束
  uint8_t arg = identifierConstant(&name);
```

> And then finally uses it here:

最后在这里使用它：

```
  uint8_t arg = identifierConstant(&name);
```

compiler.c，在namedVariable()方法中替换1行：

```
  uint8_t arg = identifierConstant(&name);
  // 替换部分开始
  if (canAssign && match(TOKEN_EQUAL)) {
  // 替换部分结束
    expression();
```

> That's a lot of plumbing to get literally one bit of data to the right place in the compiler, but arrived it has. If the variable is nested inside some expression with higher precedence, `canAssign` will be `false` and this will ignore the `=` even if there is one there. Then `namedVariable()` returns, and execution eventually makes its way back to `parsePrecedence()`.

为了把字面上的1比特数据送到编译器的正确位置需要做很多工作，但它已经到达了。如果变量嵌套在某个优先级更高的表达式中，`canAssign`将为`false`，即使有`=`也会被忽略。然后`namedVariable()`返回，执行最终返回到了`parsePrecedence()`。

> Then what? What does the compiler do with our broken example from before? Right now, `variable()` won't consume the `=`, so that will be the current token. The compiler returns back to `parsePrecedence()` from the `variable()` prefix parser and then tries to enter the infix parsing loop. There is no parsing function associated with `=`, so it skips that loop.

然后呢？编译器会对我们前面的负面例子做什么？现在，`variable()`不会消耗`=`，所以它将是当前的标识。编译器从`variable()`前缀解析器返回到`parsePrecedence()`，然后尝试进入中缀解析循环。没有与`=`相关的解析函数，因此也会跳过这个循环。

> Then `parsePrecedence()` silently returns back to the caller. That also isn't right. If the `=` doesn't get consumed as part of the expression, nothing else is going to consume it. It's an error and we should report it.

然后`parsePrecedence()`默默地返回到调用方。这也是不对的。如果`=`没有作为表达式的一部分被消耗，那么其它任何东西都不会消耗它。这是一个错误，我们应该报告它。

compiler.c，在parsePrecedence()方法中添加代码：

```
      infixRule();
    }
    // 新增部分开始
    if (canAssign && match(TOKEN_EQUAL)) {
      error("Invalid assignment target.");
    }
    // 新增部分结束
  }
```

> With that, the previous bad program correctly gets an error at compile time. OK, *now* are we done? Still not quite. See, we're passing an argument to one of the parse functions. But those functions are stored in a table of function pointers, so all of the parse functions need to have the same type. Even though most parse functions don't support being used as an assignment target—setters are the only other one—our friendly C compiler requires them *all* to accept the parameter.

这样，前面的错误程序在编译时就会正确地得到一个错误。好了，现在我们完成了吗？也不尽然。看，我们正向一个解析函数传递参数。但是这些函数是存储在一个函数指令表格中的，所以所有的解析函数需要具有相同的类型。尽管大多数解析函数都不支持被用作赋值目标——setter是唯一的一个^14——但我们这个友好的C编译器要求它们*都*接受相同的参数。

> So we're going to finish off this chapter with some grunt work. First, let's go ahead and pass the flag to the infix parse functions.

所以我们要做一些苦差事来结束这一章。首先，让我们继续前进，将标志传给中缀解析函数。

compiler.c，在parsePrecedence()方法中替换1行：

```
    ParseFn infixRule = getRule(parser.previous.type)->infix;
    // 替换部分开始
    infixRule(canAssign);
    // 替换部分结束
  }
```

> We'll need that for setters eventually. Then we'll fix the typedef for the function type.

我们最终会在setter中需要它。然后，我们要修复函数类型的类型定义。

compiler.c，在枚举Precedence后替换1行：

```
} Precedence;
// 替换部分开始
typedef void (*ParseFn)(bool canAssign);
// 替换部分结束
typedef struct {
```

> And some completely tedious code to accept this parameter in all of our existing parse functions. Here:

还有一些非常乏味的代码，为了在所有的现有解析函数中接受这个参数。这里：

compiler.c，在binary()方法中替换1行：

```
// 替换部分开始
static void binary(bool canAssign) {
// 替换部分结束
  TokenType operatorType = parser.previous.type;
```

这里:

compiler.c，在literal()方法中替换1行：

```
// 替换部分开始
static void literal(bool canAssign) {
// 替换部分结束
  switch (parser.previous.type) {
```

这里:

compiler.c，在grouping()方法中替换1行：

```
// 替换部分开始
static void grouping(bool canAssign) {
```

```
// 替换部分结束
  expression();
```

这里:

compiler.c，在number()方法中替换1行：

```
// 替换部分开始
static void number(bool canAssign) {
// 替换部分结束
  double value = strtod(parser.previous.start, NULL);
```

还有这里:

compiler.c，在string()方法中替换1行：

```
// 替换部分开始
static void string(bool canAssign) {
// 替换部分结束
  emitConstant(OBJ_VAL(copyString(parser.previous.start + 1,
```

最后:

compiler.c，在unary()方法中替换1行：

```
// 替换部分开始
static void unary(bool canAssign) {
// 替换部分结束
  TokenType operatorType = parser.previous.type;
```

> Phew! We’re back to a C program we can compile. Fire it up and now you can run this:

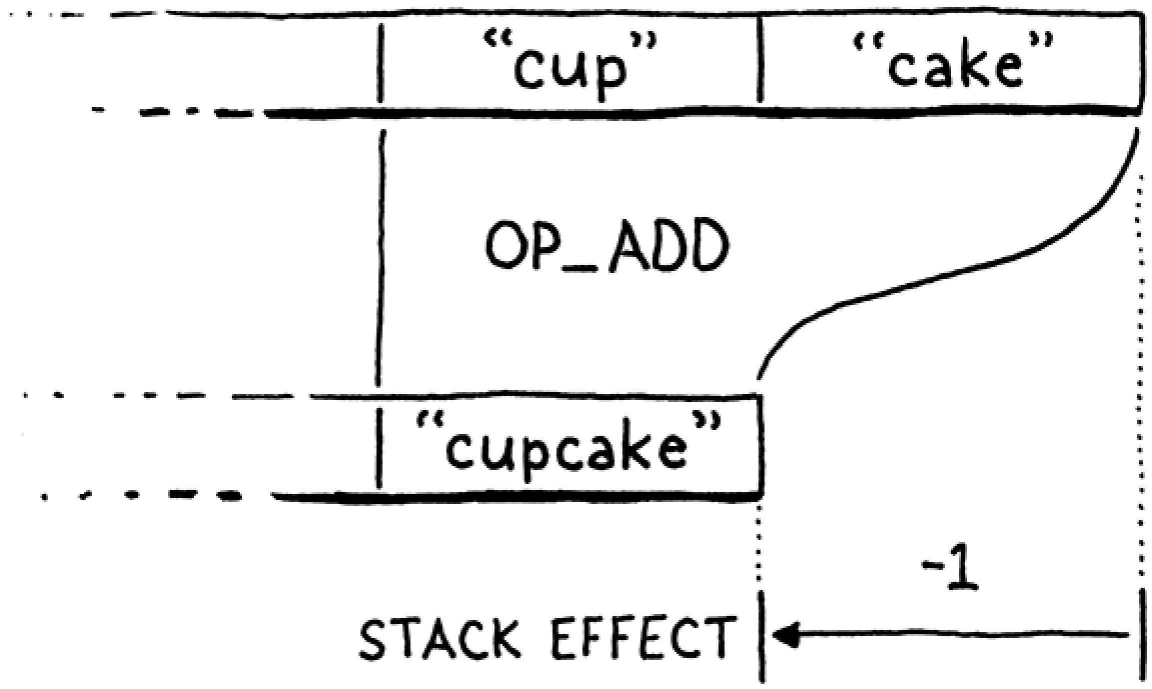
吁！我们又回到了可以编译的C程序。启动它，新增你可以运行这个：

```
var breakfast = "beignets";
var beverage = "cafe au lait";
breakfast = "beignets with " + beverage;

print breakfast;
```

> It’s starting to look like real code for an actual language!

它开始看起来像是实际语言的真正代码了！

^1: 这是复杂的语言实现中常见的元策略。通常情况下，同一种语言特性会有多种实现技术，每种技术都针对不同的使用模式进行了优化。举例来说，与属性集可以自由修改的其它对象相比，Java Script虚拟机通常对那些使用起来像类实例对象有着更快的表示形式。C和C++编译器通常由多种方法能够根据case分支数量和case值的密集程度来编译`switch`语句。 ^2: 代码块的作用有点像表达式中的括号。块可以让你把"低级别的"声明语句放在只允许"高级别的"非声明语句的地方。 ^3: 这听起来微不足道，但是非玩具型语言的手写解析器非常大。当你有数千行代码时，如果一个实用函数可以将两行代码简化为一行代码，并使结果更易于阅读，那它就很容易被接受。 ^4: `OP_ADD`执行过后堆栈会少一个元素，所以它的效应是`-1`：



^5: 不过，我们只是近了一步。等我们添加函数时，还会重新审视`OP_RETURN`。现在，它退出整个解释器的循环即可。^6: 据我统计，在本章末尾的`compiler.c`版本中，149条语句中有80条是表达式语句。 ^7: 基本上，编译器会对变量声明进行脱糖处理，如`var a;`变成`var a = nil;`，它为前者生成的代码和为后者生成的代码是相同的。 ^8: 我知道这里有一些函数现在看起来没什么意义。但是，随着我们增加更多与名称相关的语言特性，我们会从中获得更多的好处。函数和类声明都声明了新的变量，而变量表达式和赋值表达式会访问它们。 ^9: 请注意，直到将值添加到哈希表之后，我们才会弹出它。这确保了如果在将值添加到哈希表的过程中触发了垃圾回收，虚拟机仍然可以找到这个值。这显然是很可能的，因为哈希表在调整大小时需要动态分配。 ^10: 这个进程在退出时会释放所有的东西，但要求操作系统来收拾我们的烂摊子，总感觉很不体面。 ^11: 如果你还记得，在jlox中赋值是很容易的。 ^12: 对`tableSet()`的调用会将值存储在全局变量表中，即使该变量之前没有定义。这个问题在REPL会话中是用户可见的，因为即使报告了运行时错误，它仍然在运行。因此，我们也要注意从表中删除僵尸值。 ^13: 如果`a*b`是一个有效的赋值目标，这岂不是很疯狂？你可以想象一些类似代数的语言，试图以某种合理的方式划分所赋的值，并将其分配给`a`和`b`……这可能是一个很糟糕的主意。 ^14: 如果Lox有数组和下标操作符，如`array[index]`，那么中缀操作符`[`也能允许赋值，支持：`array[index] = value`。

习题

1. > The compiler adds a global variable's name to the constant table as a string every time an identifier is encountered. It creates a new constant each time, even if that variable name is already in a previous slot in the constant table. That's wasteful in cases where the same variable is referenced multiple times by the same function. That, in turn, increases the odds of filling up the constant table and running out of slots since we allow only 256 constants in a single chunk.

> Optimize this. How does your optimization affect the performance of the compiler compared to the runtime? Is this the right trade-off?

每次遇到标识符时，编译器都会将全局变量的名称作为字符串添加到常量表中。它每次都会创建一个新的常量，即使这个变量的名字已经在常量表中的前一个槽中存在。在同一个函数多次引用同一个变量的情况下，这是一种浪费。这反过来又增加了填满常量表的可能性，因为我们在一个字节码块中只允许有256个常量。

对此进行优化。与运行时相比，你的优化对编译器的性能有何影响？这是正确的取舍吗？

2. > Looking up a global variable by name in a hash table each time it is used is pretty slow, even with a good hash table. Can you come up with a more efficient way to store and access global variables without changing the semantics?

每次使用全局变量时，根据名称在哈希表中查找变量是很慢的，即使有一个很好的哈希表。你能否想出一种更有效的方法来存储和访问全局变量而不改变语义？

3. > When running in the REPL, a user might write a function that references an unknown global variable. Then, in the next line, they declare the variable. Lox should handle this gracefully by not reporting an "unknown variable" compile error when the function is first defined.
>
> But when a user runs a Lox *script*, the compiler has access to the full text of the entire program before any code is run. Consider this program:

当在REPL中运行时，用户可能会编写一个引用未知全局变量的函数。然后，在下一行中，他们声明了这个变量。Lox应该优雅地处理这个问题，在第一次定义函数时不报告"未知变量"的编译错误。

但是，当用户运行Lox脚本时，编译器可以在任何代码运行之前访问整个程序的全部文本。考虑一下这个程序：

```
fun useVar() {
  print oops;
}

var ooops = "too many o's!";
```

> Here, we can tell statically that `oops` will not be defined because there is *no* declaration of that global anywhere in the program. Note that `useVar()` is never called either, so even though the variable isn't defined, no runtime error will occur because it's never used either.
>
> We could report mistakes like this as compile errors, at least when running from a script. Do you think we should? Justify your answer. What do other scripting languages you know do?

这里，我们可以静态地告知用户`oops`不会被定义，因为在程序中没有任何地方对该全局变量进行了声明。请注意，`useVar()`也从未被调用，所以即使变量没有被定义，也不会发生运行时错误，因为它从未被使用。

我们可以将这样的错误报告为编译错误，至少在运行脚本时是这样。你认为我们应该这样做吗？请说明你的答案。你知道其它脚本语言是怎么做的吗？

22.局部变量 Local Variables

> And as imagination bodies forth The forms of things unknown, the poet's pen Turns them to shapes and gives to airy nothing A local habitation and a name.
>
> —— William Shakespeare, *A Midsummer Night's Dream*

随着想象力的不断涌现

未知事物的形式，诗人的笔

把它们变成形状，变成虚无

当地的居住地和名字。

（威廉·莎士比亚《仲夏夜之梦》）

> The last chapter introduced variables to clox, but only of the global variety. In this chapter, we'll extend that to support blocks, block scope, and local variables. In jlox, we managed to pack all of that and globals into one chapter. For clox, that's two chapters worth of work partially because, frankly, everything takes more effort in C.

上一章介绍了clox中的变量，但是只介绍了全局变量。在本章中，我们将进一步支持块、块作用域和局部变量。在jlox中，我们设法将所有这些内容和全局变量打包成一章。对于clox来说，这需要两章的工作量，坦率的说，部分原因是在C语言中一切都要花费更多的精力。

> But an even more important reason is that our approach to local variables will be quite different from how we implemented globals. Global variables are late bound in Lox. "Late" in this context means "resolved after compile time". That's good for keeping the compiler simple, but not great for performance. Local variables are one of the most-used parts of a language. If locals are slow, *everything* is slow. So we want a strategy for local variables that's as efficient as possible.

但更重要的原因是，我们处理局部变量的方法与我们实现全局变量的方法截然不同。全局变量在Lox中是后期绑定的。这里的"后期"是指"在编译后分析"。这有利于保持编译器的简单性，但不利于性能。局部变量是语言中最常用的部分之一。如果局部变量很慢，那么*一切*都是缓慢的。因此，对于局部变量，我们希望采取尽可能高效的策略^1。

> Fortunately, lexical scoping is here to help us. As the name implies, lexical scope means we can resolve a local variable just by looking at the text of the program—locals are *not* late bound. Any processing work we do in the compiler is work we *don't* have to do at runtime, so our implementation of local variables will lean heavily on the compiler.

幸运的是，词法作用域可以帮助我们。顾名思义，词法作用域意味着我们可以通过查看程序文本来解析局部变量——局部变量*不是*后期绑定的。我们在编译器中所做的任何处理工作都不必在运行时完成，因此局部变量的实现将在很大程度上依赖于编译器。

> ## 22.1 Representing Local Variables

22.1 表示局部变量

> The nice thing about hacking on a programming language in modern times is there's a long lineage of other languages to learn from. So how do C and Java manage their local variables? Why, on the stack,

> of course! They typically use the native stack mechanisms supported by the chip and OS. That's a little too low level for us, but inside the virtual world of clox, we have our own stack we can use.

在现代，实现一门编程语言的好处是，可以参考已经发展了很长时间的其它语言。那么，C和Java是如何管理它们的局部变量的呢？当然是在堆栈上！它们通常使用芯片和操作系统支持的本地堆栈机制。这对我们来说有点太底层了，但是在clox的虚拟世界中，我们有自己的堆栈可以使用。

> Right now, we only use it for holding on to **temporaries**—short-lived blobs of data that we need to remember while computing an expression. As long as we don't get in the way of those, we can stuff our local variables onto the stack too. This is great for performance. Allocating space for a new local requires only incrementing the `stackTop` pointer, and freeing is likewise a decrement. Accessing a variable from a known stack slot is an indexed array lookup.

现在，我们只使用它来保存**临时变量**——我们在计算表达式时需要记住的短期数据块。只要我们不妨碍这些数据，我们也可以把局部变量塞到栈中。这对性能很有帮助。为一个新的局部变量分配空间只需要递增`stackTop`指针，而释放也同样是递减的过程。从已知的栈槽访问变量是一种索引数组的查询。

> We do need to be careful, though. The VM expects the stack to behave like, well, a stack. We have to be OK with allocating new locals only on the top of the stack, and we have to accept that we can discard a local only when nothing is above it on the stack. Also, we need to make sure temporaries don't interfere.

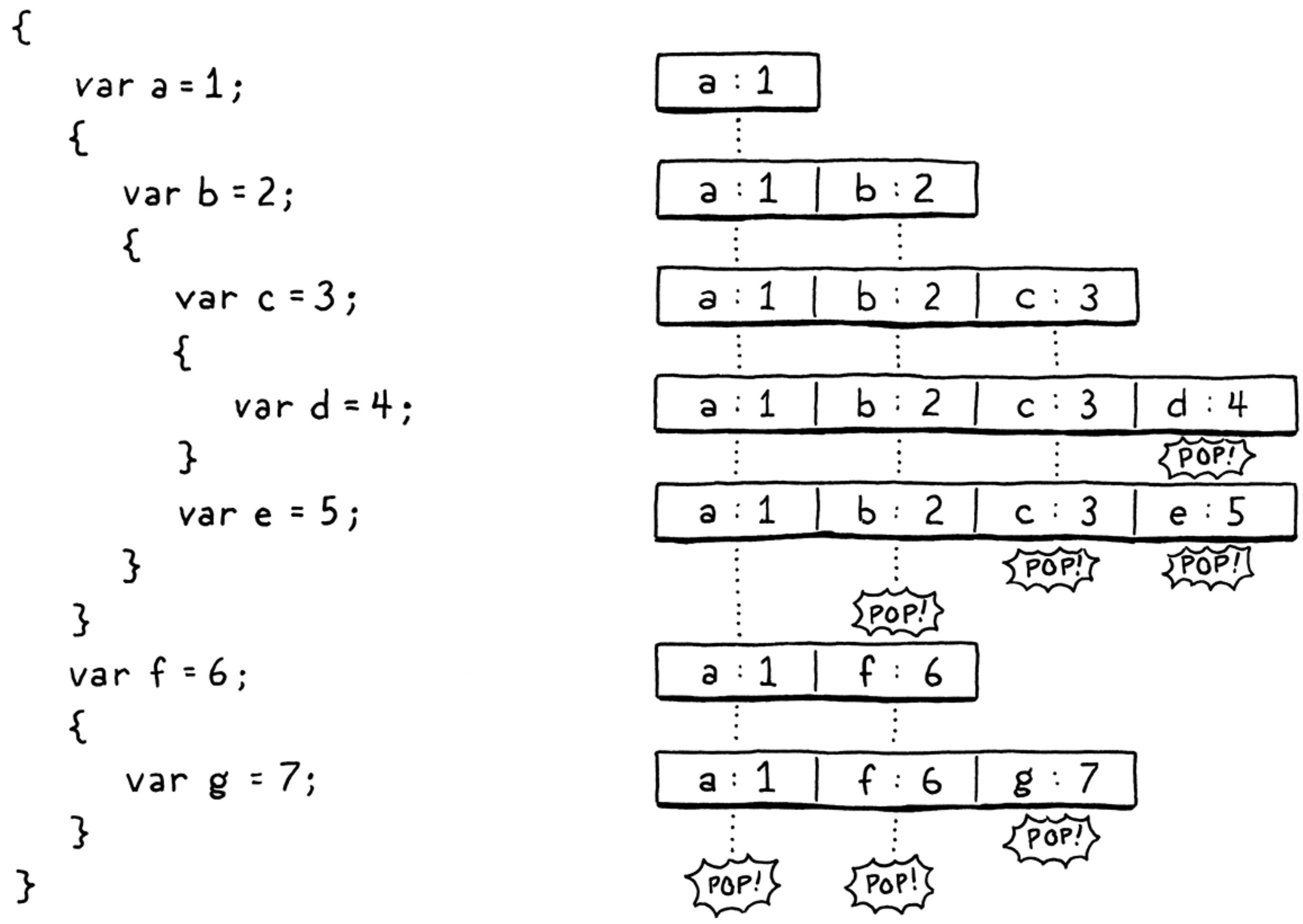
不过，我们确实需要小心。虚拟机希望栈的行为就像，嗯，一个栈。我们必须接受只能在栈顶分配新的局部变量，而且我们必须接受只有局部变量上方的栈槽没有数据时，才能丢弃该变量。此外，我们还需要保证临时变量不受干扰。

> Conveniently, the design of Lox is in harmony with these constraints. New locals are always created by declaration statements. Statements don't nest inside expressions, so there are never any temporaries on the stack when a statement begins executing. Blocks are strictly nested. When a block ends, it always takes the innermost, most recently declared locals with it. Since those are also the locals that came into scope last, they should be on top of the stack where we need them.

方便的是，Lox的设计与这些约束条件是一致的^2。新的局部变量总是通过声明语句创建的。语句不会嵌套在表达式内，所以当一个语句开始执行时，栈中没有任何临时变量。代码块是严格嵌套的。当一个块结束时，它总会带走最内部、最近声明的局部变量。因为这些也是最后进入作用域的局部变量，所以它们应该位于栈顶（我们期望它所在的位置）。

> Step through this example program and watch how the local variables come in and go out of scope:

逐步执行这段示例代码，查看局部变量是如何进入和离开作用域的：



> See how they fit a stack perfectly? It seems that the stack will work for storing locals at runtime. But we can go further than that. Not only do we know *that* they will be on the stack, but we can even pin down precisely *where* they will be on the stack. Since the compiler knows exactly which local variables are in scope at any point in time, it can effectively simulate the stack during compilation and note where in the stack each variable lives.

看到它们如何完美地适应堆栈了吗？看来，栈可以在运行时存储局部变量。但是我们可以更进一步。我们不仅知道它们会在栈上，而且我们甚至可以确定它们在栈上的精确位置。因为编译器确切地知道任何时间点上有哪些局部变量在作用域中，因此它可以在编译过程中有效地模拟堆栈，并注意每个变量在栈中的位置。

> We'll take advantage of this by using these stack offsets as operands for the bytecode instructions that read and store local variables. This makes working with locals deliciously fast—as simple as indexing into an array.

我们将利用这一点，对于读取和存储局部变量的字节码指令，把这些栈偏移量作为其操作数。这使得局部变量非常快——就像索引数组一样简单^3。

> There's a lot of state we need to track in the compiler to make this whole thing go, so let's get started there. In jlox, we used a linked chain of "environment" HashMaps to track which local variables were currently in scope. That's sort of the classic, schoolbook way of representing lexical scope. For clox, as usual, we're going a little closer to the metal. All of the state lives in a new struct.

我们需要在编译器中跟踪大量状态，以使整个程序运行起来，让我们就从那里开始。在jlox中，我们使用"环境"HashMap链来跟踪当前在作用域中的局部变量。这是一种经典的、教科书式的词法作用域表示方式。对于clox，像往常一样，我们更接近于硬件。所有的状态都保存了一个新的结构体中。

compiler.c，在结构体ParseRule后添加代码：

```
} ParseRule;
// 新增部分开始
typedef struct {
  Local locals[UINT8_COUNT];
  int localCount;
  int scopeDepth;
} Compiler;
// 新增部分结束
Parser parser;
```

> We have a simple, flat array of all locals that are in scope during each point in the compilation process. They are ordered in the array in the order that their declarations appear in the code. Since the instruction operand we'll use to encode a local is a single byte, our VM has a hard limit on the number of locals that can be in scope at once. That means we can also give the locals array a fixed size.

我们有一个简单、扁平的数组，其中包含了编译过程中每个时间点上处于作用域内的所有局部变量^4。它们在数组中的顺序与它们的声明在代码中出现的顺序相同。由于我们用来编码局部变量的指令操作数是一个字节，所以我们的虚拟机对同时处于作用域内的局部变量的数量有一个硬性限制。这意味着我们也可以给局部变量数组一个固定的大小。

common.h，添加代码：

```
#define DEBUG_TRACE_EXECUTION
// 新增部分开始
#define UINT8_COUNT (UINT8_MAX + 1)
// 新增部分结束
#endif
```

> Back in the Compiler struct, the `localCount` field tracks how many locals are in scope—how many of those array slots are in use. We also track the "scope depth". This is the number of blocks surrounding the current bit of code we're compiling.

回到Compiler结构体中，`localCount`字段记录了作用域中有多少局部变量——有多少个数组槽在使用。我们还会跟踪"作用域深度"。这指的是我们正在编译的当前代码外围的代码块数量。

> Our Java interpreter used a chain of maps to keep each block's variables separate from other blocks'. This time, we'll simply number variables with the level of nesting where they appear. Zero is the global scope, one is the first top-level block, two is inside that, you get the idea. We use this to track which block each local belongs to so that we know which locals to discard when a block ends.

我们的Java解释器使用了一个map链将每个块的变量与其它块分开。这一次，我们根据变量出现的嵌套级别对其进行编号。0是全局作用域，1是第一个顶层块，2是它内部的块，你懂的。我们用它来跟踪每个局部变量属于哪个块，这样当一个块结束时，我们就知道该删除哪些局部变量。

> Each local in the array is one of these:

数组中的每个局部变量都是这样的：

compiler.c，在结构体ParseRule后添加代码：

```
} ParseRule;
// 新增部分开始
typedef struct {
  Token name;
  int depth;
} Local;
// 新增部分结束
typedef struct {
```

> We store the name of the variable. When we're resolving an identifier, we compare the identifier's lexeme with each local's name to find a match. It's pretty hard to resolve a variable if you don't know its name. The `depth` field records the scope depth of the block where the local variable was declared. That's all the state we need for now.

我们存储变量的名称。当我们解析一个标识符时，会将标识符的词素与每个局部变量名称进行比较，以找到一个匹配项。如果你不知道变量的名称，就很难解析它。`depth`字段记录了声明局部变量的代码块的作用域深度。这就是我们现在需要的所有状态。

> This is a very different representation from what we had in jlox, but it still lets us answer all of the same questions our compiler needs to ask of the lexical environment. The next step is figuring out how the compiler *gets* at this state. If we were principled engineers, we'd give each function in the front end a parameter that accepts a pointer to a Compiler. We'd create a Compiler at the beginning and carefully thread it through each function call . . . but that would mean a lot of boring changes to the code we already wrote, so here's a global variable instead:

这与我们在jlox中使用的表示方式非常不同，但我们用它仍然可以回答编译器需要向词法环境提出的所有相同的问题。下一步是弄清楚编译器如何获取这个状态。如果我们是有原则的工程师，我们应该给前端的每个函数添加一个参数，接受一个指向Compiler的指针。我们在一开始就创建一个Compiler，并小心地在将它贯穿于每个函数的调用中……但这意味着要对我们已经写好的代码进行大量无聊的修改，所以这里用一个全局变量代替^5：

compiler.c，在变量parser后添加代码：

```
Parser parser;
// 新增部分开始
Compiler* current = NULL;
// 新增部分结束
Chunk* compilingChunk;
```

> Here's a little function to initialize the compiler:

下面是一个用于初始化编译器的小函数：

compiler.c，在emitConstant()方法后添加代码：

```
static void initCompiler(Compiler* compiler) {
  compiler->localCount = 0;
  compiler->scopeDepth = 0;
  current = compiler;
}
```

> When we first start up the VM, we call it to get everything into a clean state.

当我们第一次启动虚拟机时，我们会调用它使所有东西进入一个干净的状态。

compiler.c，在compile()方法中添加代码：

```
  initScanner(source);
  // 新增部分开始
  Compiler compiler;
  initCompiler(&compiler);
  // 新增部分结束
  compilingChunk = chunk;
```

> Our compiler has the data it needs, but not the operations on that data. There's no way to create and destroy scopes, or add and resolve variables. We'll add those as we need them. First, let's start building some language features.

我们的编译器有了它需要的数据，但还没有对这些数据的操作。没有办法创建或销毁作用域，添加和解析变量。我们会在需要的时候添加这些功能。首先，让我们开始构建一些语言特性。

22.2 Block Statements

22.2 块语句

> Before we can have any local variables, we need some local scopes. These come from two things: function bodies and blocks. Functions are a big chunk of work that we'll tackle in a later chapter, so for now we're only going to do blocks. As usual, we start with the syntax. The new grammar we'll introduce is:

在能够使用局部变量之前，我们需要一些局部作用域。它们来自于两方面：函数体和代码块。函数是一大块工作，我们在后面的章节中处理，因此现在我们只做块^6。和往常一样，我们从语法开始。我们要介绍的新语法是：

```
statement      → exprStmt
               | printStmt
               | block ;

block          → "{" declaration* "}" ;
```

> Blocks are a kind of statement, so the rule for them goes in the `statement` production. The corresponding code to compile one looks like this:

块是一种语句，所以它的规则是在`statement`生成式中。对应的编译代码如下：

compiler.c，在statement()方法中添加代码：

```
  if (match(TOKEN_PRINT)) {
    printStatement();
  // 新增部分开始
  } else if (match(TOKEN_LEFT_BRACE)) {
    beginScope();
    block();
    endScope();
  // 新增部分结束
  } else {
```

> After parsing the initial curly brace, we use this helper function to compile the rest of the block:

解析完开头的花括号之后，我们使用这个辅助函数^7来编译块的其余部分：

compiler.c，在expression()方法后添加代码：

```
static void block() {
  while (!check(TOKEN_RIGHT_BRACE) && !check(TOKEN_EOF)) {
    declaration();
  }

  consume(TOKEN_RIGHT_BRACE, "Expect '}' after block.");
}
```

> It keeps parsing declarations and statements until it hits the closing brace. As we do with any loop in the parser, we also check for the end of the token stream. This way, if there's a malformed program with a missing closing curly, the compiler doesn't get stuck in a loop.

它会一直解析声明和语句，直到遇见右括号。就像我们在解析器中的所有循环一样，我们也要检查标识流是否结束。这样一来，如果有一个格式不正确的程序缺少右括号，编译器也不会卡在循环里。

> Executing a block simply means executing the statements it contains, one after the other, so there isn't much to compiling them. The semantically interesting thing blocks do is create scopes. Before we compile the body of a block, we call this function to enter a new local scope:

执行代码块只是意味着一个接一个地执行其中包含的语句，所以不需要编译它们。从语义上讲，块所做的事就是创建作用域。在我们编译块的主体之前，我们会调用这个函数进入一个新的局部作用域：

compiler.c，在endCompiler()方法后添加代码：

```
static void beginScope() {
  current->scopeDepth++;
}
```

> In order to “create” a scope, all we do is increment the current depth. This is certainly much faster than jlox, which allocated an entire new HashMap for each one. Given `beginScope()`, you can probably guess what `endScope()` does.

为了“创建”一个作用域，我们所做的就是增加当前的深度。这当然比jlox快得多，因为jlox为每个作用域分配了全新的HashMap。有了`beginScope()`，你大概能猜到`endScope()`会做什么。

compiler.c，在beginScope()方法后添加代码：

```
static void endScope() {
  current->scopeDepth--;
}
```

> That’s it for blocks and scopes—more or less—so we’re ready to stuff some variables into them.

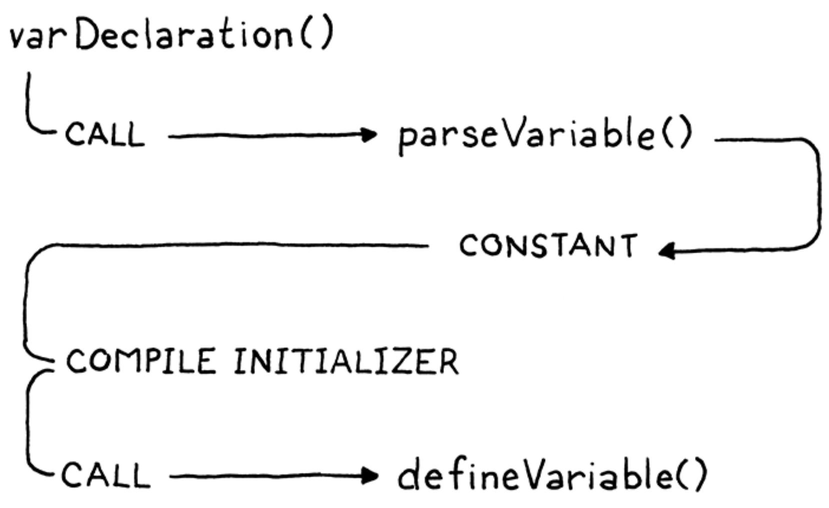
这就是块和作用域的全部内容——或多或少吧——现在我们准备在其中添加一些变量。

22.3 Declaring Local Variables

22.3 声明局部变量

> Usually we start with parsing here, but our compiler already supports parsing and compiling variable declarations. We’ve got `var` statements, identifier expressions and assignment in there now. It’s just that the compiler assumes all variables are global. So we don’t need any new parsing support, we just need to hook up the new scoping semantics to the existing code.

通常我们会从解析开始，但是我们的编译器已经支持了解析和编译变量声明。我们现在已经有了`var`语句、标识符表达式和赋值语句。只是编译器假设所有的变量都是全局变量。所以，我们不需要任何新的解析支持，我们只需要将新的作用域语义与已有的代码连接起来。



> Variable declaration parsing begins in `varDeclaration()` and relies on a couple of other functions. First, `parseVariable()` consumes the identifier token for the variable name, adds its lexeme to the chunk's constant table as a string, and then returns the constant table index where it was added. Then, after `varDeclaration()` compiles the initializer, it calls `defineVariable()` to emit the bytecode for storing the variable's value in the global variable hash table.

变量声明的解析从`varDeclaration()`开始，并依赖于其它几个函数。首先，`parseVariable()`会使用标识符标识作为变量名称，将其词素作为字符串添加到字节码块的常量表中，然后返回它的常量表索引。接着，在`varDeclaration()`编译完初始化表达式后，会调用`defineVariable()`生成字节码，将变量的值存储到全局变量哈希表中。

> Both of those helpers need a few changes to support local variables. In `parseVariable()`, we add:

这两个辅助函数都需要一些调整以支持局部变量。在`parseVariable()`中，我们添加：

compiler.c，在parseVariable()方法中添加代码：

```
  consume(TOKEN_IDENTIFIER, errorMessage);
  // 新增部分开始
  declareVariable();
  if (current->scopeDepth > 0) return 0;
  // 新增部分结束
  return identifierConstant(&parser.previous);
```

> First, we "declare" the variable. I'll get to what that means in a second. After that, we exit the function if we're in a local scope. At runtime, locals aren't looked up by name. There's no need to stuff the variable's name into the constant table, so if the declaration is inside a local scope, we return a dummy table index instead.

首先，我们"声明"这个变量。我一会儿会说到这是什么意思。之后，如果我们在局部作用域中，则退出函数。在运行时，不会通过名称查询局部变量。不需要将变量的名称放入常量表中，所以如果声明在局部作用域内，则返回一个假的表索引。

> Over in `defineVariable()`, we need to emit the code to store a local variable if we're in a local scope. It looks like this:

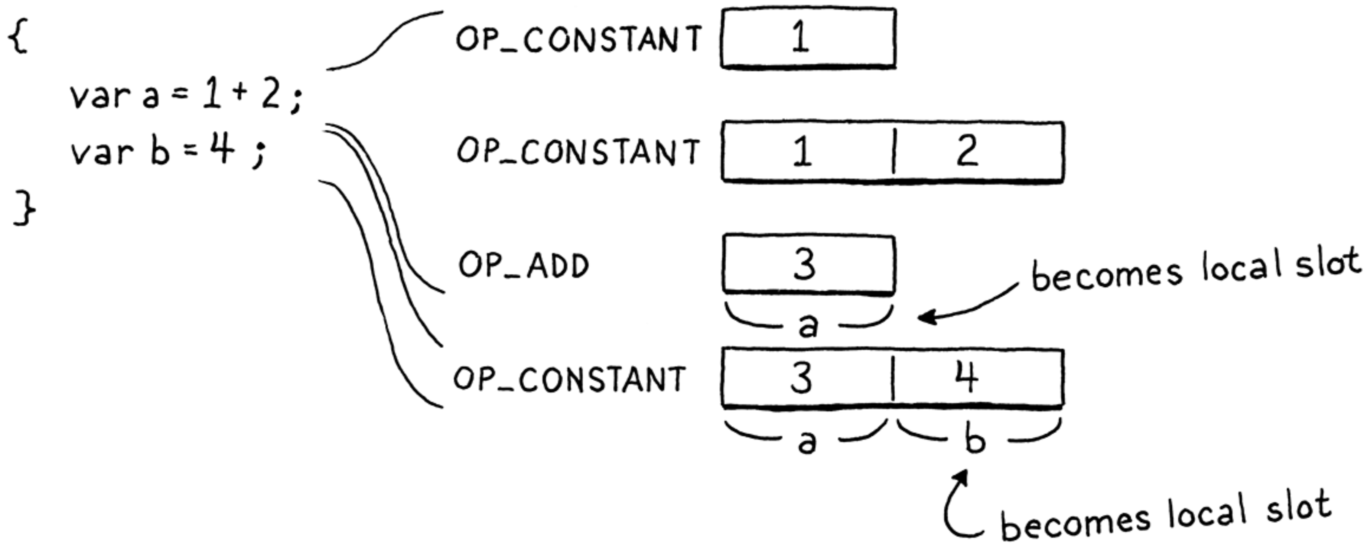
在`defineVariable()`中，如果处于局部作用域内，就需要生成一个字节码来存储局部变量。它看起来是这样的：

compiler.c，在defineVariable()方法中添加代码：

```
static void defineVariable(uint8_t global) {
  // 新增部分开始
  if (current->scopeDepth > 0) {
    return;
  }
  // 新增部分结束
  emitBytes(OP_DEFINE_GLOBAL, global);
```

> Wait, what? Yup. That's it. There is no code to create a local variable at runtime. Think about what state the VM is in. It has already executed the code for the variable's initializer (or the implicit `nil` if the user omitted an initializer), and that value is sitting right on top of the stack as the only remaining temporary. We also know that new locals are allocated at the top of the stack . . . right where that value already is. Thus, there's nothing to do. The temporary simply *becomes* the local variable. It doesn't get much more efficient than that.

等等，什么？是的，就是这样。没有代码会在运行时创建局部变量。想想虚拟机现在处于什么状态。它已经执行了变量初始化表达式的代码（如果用户省略了初始化，则是隐式的`nil`），并且该值作为唯一保留的临时变量位于栈顶。我们还知道，新的局部变量会被分配到栈顶……这个值已经在那里了。因此，没有什么可做的。临时变量直接*成为*局部变量。没有比这更有效的方法了。



> OK, so what's "declaring" about? Here's what that does:

好的，那"声明"是怎么回事呢？它的作用如下：

compiler.c，在identifierConstant()方法后添加代码：

```
static void declareVariable() {
  if (current->scopeDepth == 0) return;

  Token* name = &parser.previous;
  addLocal(*name);
}
```

> This is the point where the compiler records the existence of the variable. We only do this for locals, so if we're in the top-level global scope, we just bail out. Because global variables are late bound, the compiler doesn't keep track of which declarations for them it has seen.

在这里，编译器记录变量的存在。我们只对局部变量这样做，所以如果在顶层全局作用域中，就直接退出。因为全局变量是后期绑定的，所以编译器不会跟踪它所看到的关于全局变量的声明。

> But for local variables, the compiler does need to remember that the variable exists. That's what declaring it does—it adds it to the compiler's list of variables in the current scope. We implement that using another new function.

但是对于局部变量，编译器确实需要记住变量的存在。这就是声明的作用——将变量添加到编译器在当前作用域内的变量列表中。我们使用另一个新函数来实现这一点。

compiler.c，在identifierConstant()方法后添加代码：

```
static void addLocal(Token name) {
  Local* local = &current->locals[current->localCount++];
  local->name = name;
  local->depth = current->scopeDepth;
}
```

> This initializes the next available Local in the compiler's array of variables. It stores the variable's name and the depth of the scope that owns the variable.

这会初始化编译器变量数组中下一个可用的Local。它存储了变量的名称和持有变量的作用域的深度^8。

> Our implementation is fine for a correct Lox program, but what about invalid code? Let's aim to be robust. The first error to handle is not really the user's fault, but more a limitation of the VM. The instructions to work with local variables refer to them by slot index. That index is stored in a single-byte operand, which means the VM only supports up to 256 local variables in scope at one time.

我们的实现对于一个正确的Lox程序来说是没有问题的，但是对于无效的代码呢？我们还是以稳健为目标。第一个要处理的错误其实不是用户的错，而是虚拟机的限制。使用局部变量的指令通过槽的索引来引用变量。该索引存储在一个单字节操作数中，这意味着虚拟机一次最多只能支持256个局部变量。

> If we try to go over that, not only could we not refer to them at runtime, but the compiler would overwrite its own locals array, too. Let's prevent that.

如果我们试图超过这个范围，不仅不能在运行时引用变量，而且编译器也会覆盖自己的局部变量数组。我们要防止这种情况。

compiler.c，在addLocal()方法中添加代码：

```
static void addLocal(Token name) {
  // 新增部分开始
  if (current->localCount == UINT8_COUNT) {
    error("Too many local variables in function.");
    return;
  }
  // 新增部分结束
  Local* local = &current->locals[current->localCount++];
```

> The next case is trickier. Consider:

接下来的情况就有点棘手了。考虑一下：

```
{
  var a = "first";
  var a = "second";
}
```

> At the top level, Lox allows redeclaring a variable with the same name as a previous declaration because that's useful for the REPL. But inside a local scope, that's a pretty weird thing to do. It's likely to be a mistake, and many languages, including our own Lox, enshrine that assumption by making this an error.

在顶层，Lox允许使用与之前声明的变量相同的名称重新声明一个变量，因为这在REPL中很有用。但在局部作用域中，这就有些奇怪了。这很可能是一个误用，许多语言（包括我们的Lox）都把它作为一个错误^9。

> Note that the above program is different from this one:

请注意，上面的代码跟这个是不同的：

```
{
  var a = "outer";
  {
    var a = "inner";
  }
}
```

> It's OK to have two variables with the same name in *different* scopes, even when the scopes overlap such that both are visible at the same time. That's shadowing, and Lox does allow that. It's only an error to have two variables with the same name in the *same* local scope.

在*不同的*作用域中有两个同名变量是可以的，即使作用域重叠，以至于两个变量是同时可见的。这就是遮蔽，而Lox确实允许这样做。只有在*同一个*局部作用域中有两个同名的变量才是错误的。

> We detect that error like so:

我们这样检测这个错误^10：

compiler.c，在declareVariable()方法中添加代码：

```
  Token* name = &parser.previous;
  // 新增部分开始
  for (int i = current->localCount - 1; i >= 0; i--) {
    Local* local = &current->locals[i];
    if (local->depth != -1 && local->depth < current->scopeDepth) {
      break;
    }

    if (identifiersEqual(name, &local->name)) {
      error("Already a variable with this name in this scope.");
    }
  }
  // 新增部分结束
```

```
    addLocal(*name);
  }
```

> Local variables are appended to the array when they're declared, which means the current scope is always at the end of the array. When we declare a new variable, we start at the end and work backward, looking for an existing variable with the same name. If we find one in the current scope, we report the error. Otherwise, if we reach the beginning of the array or a variable owned by another scope, then we know we've checked all of the existing variables in the scope.

局部变量在声明时被追加到数组中，这意味着当前作用域始终位于数组的末端。当我们声明一个新的变量时，我们从末尾开始，反向查找具有相同名称的已有变量。如果是当前作用域中找到，我们就报告错误。此外，如果我们已经到达了数组开头或另一个作用域中的变量，我们就知道已经检查了当前作用域中的所有现有变量。

> To see if two identifiers are the same, we use this:

为了查看两个标识符是否相同，我们使用这个方法：

compiler.c，在identifierConstant()方法后添加代码：

```
  static bool identifiersEqual(Token* a, Token* b) {
    if (a->length != b->length) return false;
    return memcmp(a->start, b->start, a->length) == 0;
  }
```

> Since we know the lengths of both lexemes, we check that first. That will fail quickly for many non-equal strings. If the lengths are the same, we check the characters using `memcmp()`. To get to `memcmp()`, we need an include.

既然我们知道两个词素的长度，那我们首先检查它^11。对于很多不相等的字符串，在这一步就很快失败了。如果长度相同，我们就使用`memcmp()`检查字符。为了使用`memcmp()`，我们需要引入一下。

compiler.c，添加代码：

```
  #include <stdlib.h>
  // 新增部分开始
  #include <string.h>
  // 新增部分结束
  #include "common.h"
```

有了这个，我们就能创造出变量。但是，它们会停留在声明它们的作用域之外，像幽灵一样。当一个代码块结束时，我们需要让其中的变量安息。

compiler.c，在endScope()方法中添加代码：

```
    current->scopeDepth--;
    // 新增部分开始
    while (current->localCount > 0 &&
           current->locals[current->localCount - 1].depth >
              current->scopeDepth) {
      emitByte(OP_POP);
      current->localCount--;
    }
    // 新增部分结束
  }
```

> When we pop a scope, we walk backward through the local array looking for any variables declared at the scope depth we just left. We discard them by simply decrementing the length of the array.

当我们弹出一个作用域时，后向遍历局部变量数组，查找在刚刚离开的作用域深度上声明的所有变量。我们通过简单地递减数组长度来丢弃它们。

> There is a runtime component to this too. Local variables occupy slots on the stack. When a local variable goes out of scope, that slot is no longer needed and should be freed. So, for each variable that we discard, we also emit an `OP_POP` instruction to pop it from the stack.

这里也有一个运行时的因素。局部变量占用了堆栈中的槽位。当局部变量退出作用域时，这个槽就不再需要了，应该被释放。因此，对于我们丢弃的每一个变量，我们也要生成一条`OP_POP`指令，将其从栈中弹出^12。

> ## 22.4 Using Locals

22.4 使用局部变量

> We can now compile and execute local variable declarations. At runtime, their values are sitting where they should be on the stack. Let's start using them. We'll do both variable access and assignment at the same time since they touch the same functions in the compiler.

我们现在可以编译和执行局部变量的声明了。在运行时，它们的值就在栈中应在的位置上。让我们开始使用它们吧。我们会同时完成变量访问和赋值，因为它们在编译器中涉及相同的函数。

> We already have code for getting and setting global variables, and—like good little software engineers—we want to reuse as much of that existing code as we can. Something like this:

我们已经有了获取和设置全局变量的代码，而且像优秀的小软件工程师一样，我们希望尽可能多地重用现有的代码。就像这样：

compiler.c，在namedVariable()方法中替换1行：

```
  static void namedVariable(Token name, bool canAssign) {
    // 替换部分开始
    uint8_t getOp, setOp;
    int arg = resolveLocal(current, &name);
    if (arg != -1) {
      getOp = OP_GET_LOCAL;
      setOp = OP_SET_LOCAL;
    } else {
      arg = identifierConstant(&name);
      getOp = OP_GET_GLOBAL;
      setOp = OP_SET_GLOBAL;
    }
    // 替换部分结束
    if (canAssign && match(TOKEN_EQUAL)) {
```

> Instead of hardcoding the bytecode instructions emitted for variable access and assignment, we use a couple of C variables. First, we try to find a local variable with the given name. If we find one, we use the instructions for working with locals. Otherwise, we assume it's a global variable and use the existing bytecode instructions for globals.

我们不对变量访问和赋值对应的字节码指令进行硬编码，而是使用了一些C变量。首先，我们尝试查找具有给定名称的局部变量，如果我们找到了，就使用处理局部变量的指令。否则，我们就假定它是一个全局变量，并使用现有的处理全局变量的字节码。

> A little further down, we use those variables to emit the right instructions. For assignment:

再往下一点，我们使用这些变量来生成正确的指令。对于赋值：

compiler.c，在namedVariable()方法中替换1行：

```
    if (canAssign && match(TOKEN_EQUAL)) {
      expression();
      // 替换部分开始
      emitBytes(setOp, (uint8_t)arg);
      // 替换部分结束
    } else {
```

> And for access:

对于访问：

compiler.c，在namedVariable()方法中替换1行：

```
      emitBytes(setOp, (uint8_t)arg);
    } else {
      // 替换部分开始
      emitBytes(getOp, (uint8_t)arg);
      // 替换部分结束
    }
```

> The real heart of this chapter, the part where we resolve a local variable, is here:

本章的核心，也就是解析局部变量的部分，在这里：

compiler.c，在identifiersEqual()方法后添加代码：

```
static int resolveLocal(Compiler* compiler, Token* name) {
  for (int i = compiler->localCount - 1; i >= 0; i--) {
    Local* local = &compiler->locals[i];
    if (identifiersEqual(name, &local->name)) {
      return i;
    }
  }

  return -1;
}
```

> For all that, it's straightforward. We walk the list of locals that are currently in scope. If one has the same name as the identifier token, the identifier must refer to that variable. We've found it! We walk the array backward so that we find the *last* declared variable with the identifier. That ensures that inner local variables correctly shadow locals with the same name in surrounding scopes.

尽管如此，它还是很直截了当的。我们会遍历当前在作用域内的局部变量列表。如果有一个名称与标识符相同，则标识符一定指向该变量。我们已经找到了它！我们后向遍历数组，这样就能找到最后一个带有该标识符的已声明变量。这可以确保内部的局部变量能正确地遮蔽外围作用域中的同名变量。

> At runtime, we load and store locals using the stack slot index, so that's what the compiler needs to calculate after it resolves the variable. Whenever a variable is declared, we append it to the locals array in Compiler. That means the first local variable is at index zero, the next one is at index one, and so on. In other words, the locals array in the compiler has the *exact* same layout as the VM's stack will have at runtime. The variable's index in the locals array is the same as its stack slot. How convenient!

在运行时，我们使用栈中槽索引来加载和存储局部变量，因此编译器在解析变量之后需要计算索引。每当一个变量被声明，我们就将它追加到编译器的局部变量数组中。这意味着第一个局部变量在索引0的位置，下一个在索引1的位置，以此类推。换句话说，编译器中的局部变量数组的布局与虚拟机堆栈在运行时的布局完全相同。变量在局部变量数组中的索引与其在栈中的槽位相同。多么方便啊！

> If we make it through the whole array without finding a variable with the given name, it must not be a local. In that case, we return `-1` to signal that it wasn't found and should be assumed to be a global variable instead.

如果我们在整个数组中都没有找到具有指定名称的变量，那它肯定不是局部变量。在这种情况下，我们返回`-1`，表示没有找到，应该假定它是一个全局变量。

> ### 22.4.1 Interpreting local variables

22.4.1 解释局部变量

> Our compiler is emitting two new instructions, so let's get them working. First is loading a local variable:

我们的编译器发出了两条新指令，我们来让它们发挥作用。首先是加载一个局部变量：

chunk.h，在枚举OpCode中添加代码：

```
  OP_POP,
  // 新增部分开始
  OP_GET_LOCAL,
  // 新增部分结束
  OP_GET_GLOBAL,
```

> And its implementation:

还有其实现^13：

vm.c，在run()方法中添加代码：

```
      case OP_POP: pop(); break;
      // 新增部分开始
      case OP_GET_LOCAL: {
        uint8_t slot = READ_BYTE();
        push(vm.stack[slot]);
        break;
      }
      // 新增部分结束
      case OP_GET_GLOBAL: {
```

> It takes a single-byte operand for the stack slot where the local lives. It loads the value from that index and then pushes it on top of the stack where later instructions can find it.

它接受一个单字节操作数，用作局部变量所在的栈槽。它从索引处加载值，然后将其压入栈顶，在后面的指令可以找到它。

> Next is assignment:

接下来是赋值：

chunk.h，在枚举OpCode中添加代码：

```
  OP_GET_LOCAL,
  // 新增部分开始
  OP_SET_LOCAL,
  // 新增部分结束
  OP_GET_GLOBAL,
```

> You can probably predict the implementation.

你大概能预测到它的实现。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_SET_LOCAL: {
        uint8_t slot = READ_BYTE();
        vm.stack[slot] = peek(0);
        break;
      }
      // 新增部分结束
      case OP_GET_GLOBAL: {
```

> It takes the assigned value from the top of the stack and stores it in the stack slot corresponding to the local variable. Note that it doesn't pop the value from the stack. Remember, assignment is an expression, and every expression produces a value. The value of an assignment expression is the assigned value itself, so the VM just leaves the value on the stack.

它从栈顶获取所赋的值，然后存储到与局部变量对应的栈槽中。注意，它不会从栈中弹出值。请记住，赋值是一个表达式，而每个表达式都会产生一个值。赋值表达式的值就是所赋的值本身，所以虚拟机要把值留在栈上。

> Our disassembler is incomplete without support for these two new instructions.

如果不支持这两条新指令，我们的反汇编程序就不完整了。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_POP", offset);
    // 新增部分开始
    case OP_GET_LOCAL:
      return byteInstruction("OP_GET_LOCAL", chunk, offset);
    case OP_SET_LOCAL:
      return byteInstruction("OP_SET_LOCAL", chunk, offset);
    // 新增部分结束
    case OP_GET_GLOBAL:
```

> The compiler compiles local variables to direct slot access. The local variable's name never leaves the compiler to make it into the chunk at all. That's great for performance, but not so great for introspection. When we disassemble these instructions, we can't show the variable's name like we could with globals. Instead, we just show the slot number.

编译器将局部变量编译为直接的槽访问。局部变量的名称永远不会离开编译器，根本不可能进入字节码块。这对性能很好，但对内省(自我观察)来说就不那么好了。当我们反汇编这些指令时，我们不能像全局变量那样使用变量名称。相反，我们只显示槽号^14。

debug.c，在simpleInstruction()方法后添加代码：

```
static int byteInstruction(const char* name, Chunk* chunk,
                           int offset) {
  uint8_t slot = chunk->code[offset + 1];
```

```
    printf("%-16s %4d\n", name, slot);
    return offset + 2;
  }
```

22.4.2 Another scope edge case

22.4.2 另一种作用域边界情况

> We already sunk some time into handling a couple of weird edge cases around scopes. We made sure shadowing works correctly. We report an error if two variables in the same local scope have the same name. For reasons that aren't entirely clear to me, variable scoping seems to have a lot of these wrinkles. I've never seen a language where it feels completely elegant.

我们已经花了一些时间来处理部分关于作用域的奇怪的边界情况。我们确保变量遮蔽能正确工作。如果同一个局部作用域中的两个变量具有相同的名称，我们会报告错误。由于我并不完全清楚的原因，变量作用域似乎有很多这样的问题。我从来没有见过一种语言让人感觉绝对优雅^15。

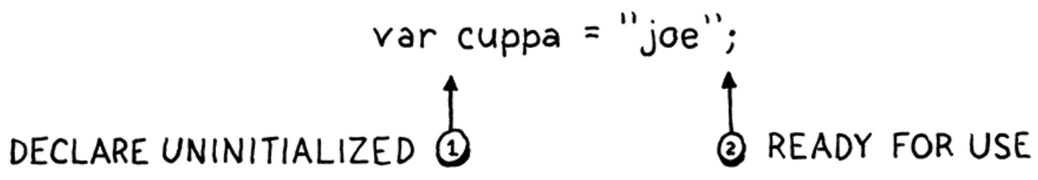
> We've got one more edge case to deal with before we end this chapter. Recall this strange beastie we first met in jlox's implementation of variable resolution:

在本章结束之前，我们还有一个边界情况需要处理。回顾一下我们第一次在jlox中实现变量解析时，遇到的这个奇怪的东西：

```
{
  var a = "outer";
  {
    var a = a;
  }
}
```

> We slayed it then by splitting a variable's declaration into two phases, and we'll do that again here:

我们当时通过将一个变量的声明拆分为两个阶段来解决这个问题，在这里我们也要这样做：



> As soon as the variable declaration begins—in other words, before its initializer—the name is declared in the current scope. The variable exists, but in a special "uninitialized" state. Then we compile the initializer. If at any point in that expression we resolve an identifier that points back to this variable, we'll see that it is not initialized yet and report an error. After we finish compiling the initializer, we mark the variable as initialized and ready for use.

一旦变量声明开始——换句话说，在它的初始化式之前——名称就会在当前作用域中声明。变量存在，但处于特殊的"未初始化"状态。然后我们编译初始化式。如果在表达式中的任何一个时间点，我们解析了一个指向该

变量的标识符，我们会发现它还没有初始化，并报告错误。在我们完成初始化表达式的编译之后，把变量标记为已初始化并可供使用。

> To implement this, when we declare a local, we need to indicate the "uninitialized" state somehow. We could add a new field to Local, but let's be a little more parsimonious with memory. Instead, we'll set the variable's scope depth to a special sentinel value, `-1`.

为了实现这一点，当声明一个局部变量时，我们需要以某种方式表明"未初始化"状态。我们可以在Local中添加一个新字段，但我们还是在内存方面更节省一些。相对地，我们将变量的作用域深度设置为一个特殊的哨兵值`-1`。

compiler.c，在addLocal()方法中替换1行：

```
    local->name = name;
    // 替换部分开始
    local->depth = -1;
    // 替换部分结束
  }
```

> Later, once the variable's initializer has been compiled, we mark it initialized.

稍后，一旦变量的初始化式编译完成，我们将其标记为已初始化。

compiler.c，在defineVariable()方法中添加代码：

```
    if (current->scopeDepth > 0) {
      // 新增部分开始
      markInitialized();
      // 新增部分结束
      return;
    }
```

> That is implemented like so:

实现如下：

compiler.c，在parseVariable()方法后添加代码：

```
  static void markInitialized() {
    current->locals[current->localCount - 1].depth =
        current->scopeDepth;
  }
```

> So this is *really* what "declaring" and "defining" a variable means in the compiler. "Declaring" is when the variable is added to the scope, and "defining" is when it becomes available for use.

所这就是编译器中“声明”和“定义”变量的真正含义。“声明”是指变量被添加到作用域中，而“定义”是变量可以被使用的时候。

> When we resolve a reference to a local variable, we check the scope depth to see if it's fully defined.

当解析指向局部变量的引用时，我们会检查作用域深度，看它是否被完全定义。

compiler.c，在resolveLocal()方法中添加代码：

```
    if (identifiersEqual(name, &local->name)) {
      // 新增部分开始
      if (local->depth == -1) {
        error("Can't read local variable in its own initializer.");
      }
      // 新增部分结束
      return i;
```

> If the variable has the sentinel depth, it must be a reference to a variable in its own initializer, and we report that as an error.

如果变量的深度是哨兵值，那这一定是在变量自身的初始化式中对该变量的引用，我们会将其报告为一个错误。

> That's it for this chapter! We added blocks, local variables, and real, honest-to-God lexical scoping. Given that we introduced an entirely different runtime representation for variables, we didn't have to write a lot of code. The implementation ended up being pretty clean and efficient.

这一章就讲到这里！我们添加了块、局部变量和真正的词法作用域。鉴于我们为变量引入了完全不同的运行时表示形式，我们不必编写很多代码。这个实现最终是相当干净和高效的。

> You'll notice that almost all of the code we wrote is in the compiler. Over in the runtime, it's just two little instructions. You'll see this as a continuing trend in clox compared to jlox. One of the biggest hammers in the optimizer's toolbox is pulling work forward into the compiler so that you don't have to do it at runtime. In this chapter, that meant resolving exactly which stack slot every local variable occupies. That way, at runtime, no lookup or resolution needs to happen.

你会注意到，我们写的几乎所有的代码都在编译器中。在运行时，只有两个小指令。你会看到，相比于jlox，这是clox中的一个持续的趋势^16。优化器工具箱中最大的锤子就是把工作提前到编译器中，这样你就不必在运行时做这些工作了。在本章中，这意味着要准确地解析每个局部变量占用的栈槽。这样，在运行时就不需要进行查找或解析。

^6: 仔细想想，“块”是个奇怪的名字。作为比喻来说，“块”通常意味着一个不可分割的小单元，但出于某种原因，Algol 60委员会决定用它来指代一个复合结构——一系列语句。我想，还有更糟的情况，Algol 58将`begin`和`end`称为“语句括号”。 ^7: 在后面编译函数体时，这个方法会派上用场。 ^8: 担心作为变量名称的字符串的生命周期吗？Local直接存储了标识符对应Token结构体的副本。Token存储了一个指向其词素中第一个字符的指针，以及词素的长度。该指针指向正在编译的脚本或REPL输入语句的源字符串。
只要这个字符串在整个编译过程中存在——你知道，它一定存在，我们正在编译它——那么所有指向它的标识都是正常的。 ^9: 有趣的是，Rust语言确实允许这样做，而且惯用代码也依赖于此。 ^10: 暂时先不用关心那个奇怪的`depth != -1`部分。我们稍后会讲到。 ^11: 如果我们能检查它们的哈希值，将是一个不错的小优化，

但标识不是完整的LoxString，所以我们还没有计算出它们的哈希值。 ^12: 当多个局部变量同时退出作用域时，你会得到一系列的`OP_POP`指令，这些指令会被逐个解释。你可以在你的Lox实现中添加一个简单的优化，那就是专门的`OP_POPN`指令，该指令接受一个操作数，作为弹出的槽位的数量，并一次性弹出所有槽位。 ^13: 把局部变量的值压到栈中似乎是多余的，因为它已经在栈中较低的某个位置了。问题是，其它字节码指令只能查找*栈顶*的数据。这也是我们的字节码指令集基于堆栈的主要表现。基于寄存器的字节码指令集避免了这种堆栈技巧，其代价是有着更多操作数的大型指令。 ^14: 如果我们想为虚拟机实现一个调试器，在编译器中擦除局部变量名称是一个真正的问题。当用户逐步执行代码时，他们希望看到局部变量的值按名称排列。为了支持这一点，我们需要输出一些额外的信息，以跟踪每个栈槽中的局部变量的名称。 ^15: 没有，即便Scheme也不是。 ^16: 你可以把静态类型看作是这种趋势的一个极端例子。静态类型语言将所有的类型分析和类型错误处理都在编译过程中进行了整理。这样，运行时就不必浪费时间来检查值是否具有适合其操作的类型。事实上，在一些静态类型语言（如C）中，你甚至不*知道*运行时的类型。编译器完全擦除值类型的任何表示，只留下空白的比特位。

习题

1. > Our simple local array makes it easy to calculate the stack slot of each local variable. But it means that when the compiler resolves a reference to a variable, we have to do a linear scan through the array.
   >
   > Come up with something more efficient. Do you think the additional complexity is worth it?

   我们这个简单的局部变量数组使得计算每个局部变量的栈槽很容易。但这意味着，当编译器解析一个变量的引用时，我们必须对数组进行线性扫描。

   想出一些更有效的方法。你认为这种额外的复杂性是否值得？

2. > How do other languages handle code like this:

   其它语言中如何处理这样的代码：

   ```
   var a = a;
   ```

   > What would you do if it was your language? Why?

   如果这是你的语言，你会怎么做？为什么？

3. > Many languages make a distinction between variables that can be reassigned and those that can't. In Java, the `final` modifier prevents you from assigning to a variable. In JavaScript, a variable declared with `let` can be assigned, but one declared using `const` can't. Swift treats `let` as single-assignment and uses `var` for assignable variables. Scala and Kotlin use `val` and `var`.
   >
   > Pick a keyword for a single-assignment variable form to add to Lox. Justify your choice, then implement it. An attempt to assign to a variable declared using your new keyword should cause a compile error.

   许多语言中，对可以重新赋值的变量与不能重新赋值的变量进行了区分。在Java中，`final`修饰符可以阻止你对变量进行赋值。在JavaScript中，用`let`声明的变量可以被赋值，但用`const`声明的变量不能被赋值。`Swift`将`let`视为单次赋值，并对可赋值变量使用`var`。`Scala`和`Kotlin`则使用`val`和`var`。

选一个关键字作为单次赋值变量的形式添加到Lox中。解释一下你的选择，然后实现它。试图赋值给一个用新关键字声明的变量应该会引起编译错误。

4. > Extend clox to allow more than 256 local variables to be in scope at a time.

   扩展Lox，允许作用域中同时有超过256个局部变量。

23.来回跳转 Jumping Back and Forth

> The order that our mind imagines is like a net, or like a ladder, built to attain something. But afterward you must throw the ladder away, because you discover that, even if it was useful, it was meaningless.
>
> ——Umberto Eco, *The Name of the Rose*

我们头脑中想象的秩序就像一张网，或者像一架梯子，用来达到某种目的。但事后你必须把梯子扔掉，因为你会发现，即使它有用，也毫无意义。（翁贝托·艾柯，《玫瑰之名》）

> It's taken a while to get here, but we're finally ready to add control flow to our virtual machine. In the tree-walk interpreter we built for jlox, we implemented Lox's control flow in terms of Java's. To execute a Lox `if` statement, we used a Java `if` statement to run the chosen branch. That works, but isn't entirely satisfying. By what magic does the *JVM itself* or a native CPU implement `if` statements? Now that we have our own bytecode VM to hack on, we can answer that.

虽然花了一些时间，但我们终于准备好向虚拟机中添加控制流了。在我们为jlox构建的树遍历解释器中，我们以Java的方式实现了控制流。为了执行Lox的`if`语句，我们使用Java的`if`语句来执行所选的分支。这是可行的，但并不是完全令人满意。JVM本身或原生CPU如何实现`if`语句呢？现在我们有了自己的字节码虚拟机，我们可以回答这个问题。

> When we talk about "control flow", what are we referring to? By "flow" we mean the way execution moves through the text of the program. Almost like there is a little robot inside the computer wandering through our code, executing bits and pieces here and there. Flow is the path that robot takes, and by *controlling* the robot, we drive which pieces of code it executes.

当我们谈论"控制流"时，我们指的是什么？我们所说的"流"是指执行过程在程序文本中的移动方式。就好像电脑里有一个小机器人在我们的代码里游荡，在这里或那里执行一些零零碎碎的片段。流就是机器人所走的路径，通过*控制*机器人，我们驱动它执行某些代码片段。

> In jlox, the robot's locus of attention—the *current* bit of code—was implicit based on which AST nodes were stored in various Java variables and what Java code we were in the middle of running. In clox, it is much more explicit. The VM's `ip` field stores the address of the current bytecode instruction. The value of that field is exactly "where we are" in the program.

在jlox中，机器人的关注点（*当前*代码位）是隐式的，它取决于哪些AST节点被存储在各种Java变量中，以及我们正在运行的Java代码是什么。在clox中，它要明确得多。VM的`ip`字段存储了当前字节码指令的地址。该字段的值正是我们在程序中的"位置"。

> Execution proceeds normally by incrementing the `ip`. But we can mutate that variable however we want to. In order to implement control flow, all that's necessary is to change the `ip` in more interesting ways. The simplest control flow construct is an `if` statement with no `else` clause:

执行操作通常是通过增加`ip`进行的。但是我们可以随意地改变这个变量。为了实现控制流，所需要做的就是以更有趣的方式改变`ip`。最简单的控制流结构是没有`else`子句的`if`语句：

```
if (condition) print("condition was truthy");
```

> The VM evaluates the bytecode for the condition expression. If the result is truthy, then it continues along and executes the `print` statement in the body. The interesting case is when the condition is falsey. When that happens, execution skips over the then branch and proceeds to the next statement.

虚拟机会计算条件表达式对应的字节码。如果结构是真，则继续执行主体中的`print`语句。有趣的是当条件为假的时候，这种情况下，执行会跳过then分支并执行下一条语句。

> To skip over a chunk of code, we simply set the `ip` field to the address of the bytecode instruction following that code. To *conditionally* skip over some code, we need an instruction that looks at the value on top of the stack. If it's falsey, it adds a given offset to the `ip` to jump over a range of instructions. Otherwise, it does nothing and lets execution proceed to the next instruction as usual.

要想跳过一大块代码，我们只需将`ip`字段设置为其后代码的字节码指令的地址。为了*有条件地*跳过一些代码，我们需要一条指令来查看栈顶的值。如果它是假，就在`ip`上增加一个给定的偏移量，跳过一系列指令。否则，它什么也不做，并照常执行下一条指令。

> When we compile to bytecode, the explicit nested block structure of the code evaporates, leaving only a flat series of instructions behind. Lox is a structured programming language, but clox bytecode isn't. The right—or wrong, depending on how you look at it—set of bytecode instructions could jump into the middle of a block, or from one scope into another.

当我们编译成字节码时，代码中显式的嵌套块结构就消失了，只留下一系列扁平的指令。Lox是一种结构化的编程语言，但clox字节码却不是。正确的（或者说错误的，取决于你怎么看待它）字节码指令集可以跳转到代码块的中间位置，或从一个作用域跳到另一个作用域。

> The VM will happily execute that, even if the result leaves the stack in an unknown, inconsistent state. So even though the bytecode is unstructured, we'll take care to ensure that our compiler only generates clean code that maintains the same structure and nesting that Lox itself does.

虚拟机会很高兴地执行这些指令，即使其结果会导致堆栈处于未知的、不一致的状态。因此，尽管字节码是非结构化的，我们也要确保编译成只生成与Lox本身保持相同结构和嵌套的干净代码。

> This is exactly how real CPUs behave. Even though we might program them using higher-level languages that mandate structured control flow, the compiler lowers that down to raw jumps. At the bottom, it turns out goto is the only real control flow.

这就是真正的CPU的行为方式。即使我们可能会使用高级语言对它们进行编程，这些语言能够规定格式化控制流，但编译器也会将其降级为原生跳转。在底层，事实证明goto是唯一真正的控制流。

> Anyway, I didn't mean to get all philosophical. The important bit is that if we have that one conditional jump instruction, that's enough to implement Lox's `if` statement, as long as it doesn't have an `else` clause. So let's go ahead and get started with that.

不管这么说，我并不是故意要搞得这么哲学化。重要的是，如果我们有一个条件跳转指令，就足以实现Lox的`if`语句了，只要它没有`else`子句。让我们开始吧。

23.1 If Statements

23.1 If语句

> This many chapters in, you know the drill. Any new feature starts in the front end and works its way through the pipeline. An `if` statement is, well, a statement, so that's where we hook it into the parser.

这么多章了，你知道该怎么做。任何新特性都是从前端开始的，如果沿着管道进行工作。`if`语句是一个，嗯，语句，所以我们通过语句将它连接到解析器。

compiler.c，在statement()语句中添加代码：

```
  if (match(TOKEN_PRINT)) {
    printStatement();
  // 新增部分开始
  } else if (match(TOKEN_IF)) {
    ifStatement();
  // 新增部分结束
  } else if (match(TOKEN_LEFT_BRACE)) {
```

> When we see an `if` keyword, we hand off compilation to this function:

如果我们看到`if`关键字，就把编译工作交给这个函数^1：

compiler.c，在expressionStatement()方法后添加代码：

```
static void ifStatement() {
  consume(TOKEN_LEFT_PAREN, "Expect '(' after 'if'.");
  expression();
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after condition.");

  int thenJump = emitJump(OP_JUMP_IF_FALSE);
  statement();

  patchJump(thenJump);
}
```

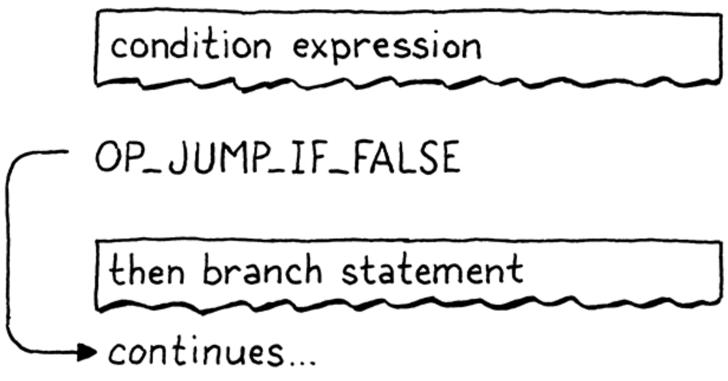
> First we compile the condition expression, bracketed by parentheses. At runtime, that will leave the condition value on top of the stack. We'll use that to determine whether to execute the then branch or skip it.

首先我们编译条件表达式（用小括号括起来）。在运行时，这会将条件值留在栈顶。我们将通过它来决定是执行then分支还是跳过它。

> Then we emit a new `OP_JUMP_IF_FALSE` instruction. It has an operand for how much to offset the `ip`—how many bytes of code to skip. If the condition is falsey, it adjusts the `ip` by that amount. Something

> like this:

然后我们生成一个新的`OP_JUMP_IF_ELSE`指令。这条指令有一个操作数，用来表示`ip`的偏移量——要跳过多少字节的代码。如果条件是假，它就按这个值调整`ip`，就像这样：

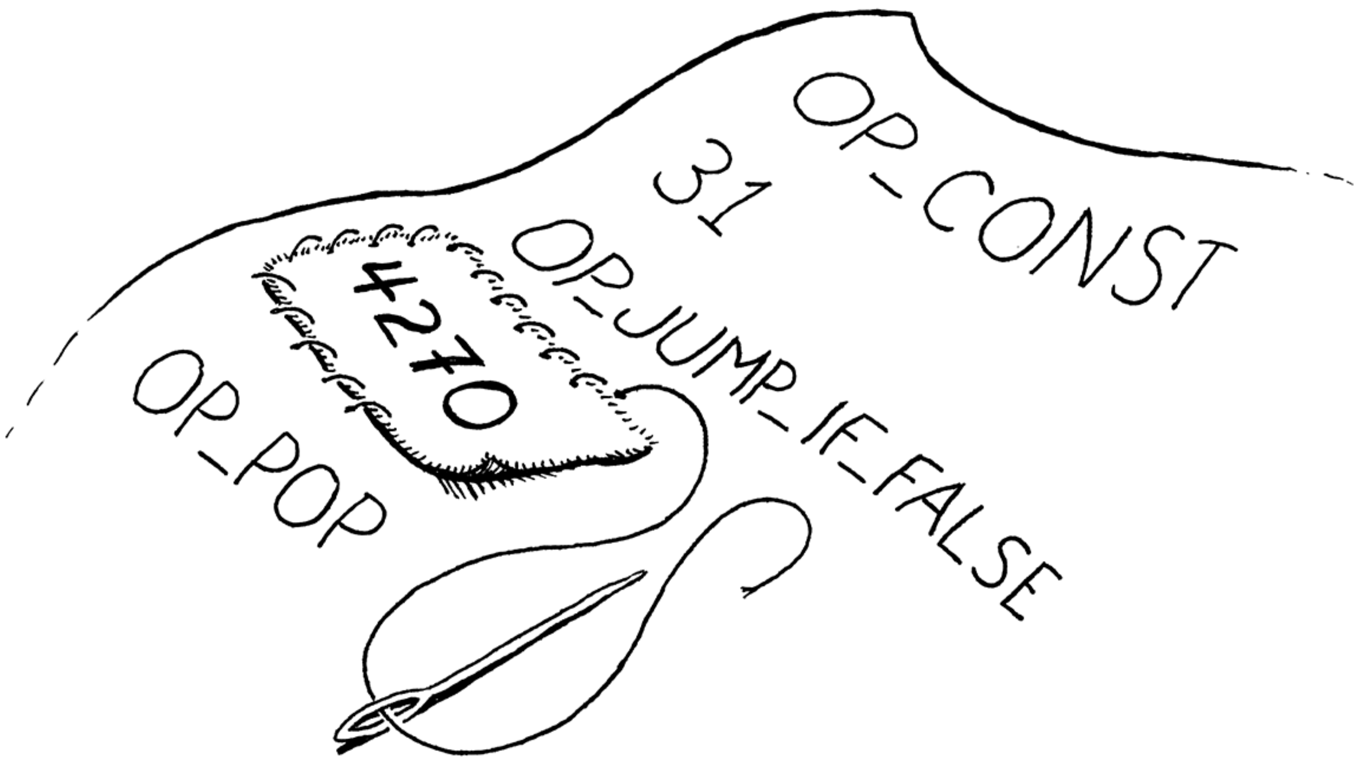


> But we have a problem. When we're writing the `OP_JUMP_IF_FALSE` instruction's operand, how do we know how far to jump? We haven't compiled the then branch yet, so we don't know how much bytecode it contains.

但我们有个问题。当我们写`OP_JUMP_IF_FALSE`指令的操作数时，我们怎么知道要跳多远？我们还没有编译then分支，所以我们不知道它包含多少字节码。

> To fix that, we use a classic trick called **backpatching**. We emit the jump instruction first with a placeholder offset operand. We keep track of where that half-finished instruction is. Next, we compile the then body. Once that's done, we know how far to jump. So we go back and replace that placeholder offset with the real one now that we can calculate it. Sort of like sewing a patch onto the existing fabric of the compiled code.

为了解决这个问题，我们使用了一个经典的技巧，叫作**回填（backpatching）**。我们首先生成跳转指令，并附上一个占位的偏移量操作数，我们跟踪这个半成品指令的位置。接下来，我们编译then主体。一旦完成，我们就知道要跳多远。所以我们回去将占位符替换为真正的偏移量，现在我们可以计算它了。这有点像在已编译代码的现有结构上打补丁。



> We encode this trick into two helper functions.

我们将这个技巧编码为两个辅助函数。

compiler.c，在emitBytes()方法后添加代码：

```
static int emitJump(uint8_t instruction) {
  emitByte(instruction);
  emitByte(0xff);
  emitByte(0xff);
  return currentChunk()->count - 2;
}
```

> The first emits a bytecode instruction and writes a placeholder operand for the jump offset. We pass in the opcode as an argument because later we'll have two different instructions that use this helper. We use two bytes for the jump offset operand. A 16-bit offset lets us jump over up to 65,535 bytes of code, which should be plenty for our needs.

第一个程序会生成一个字节码指令，并为跳转偏移量写入一个占位符操作数。我们把操作码作为参数传入，因为稍后我们会有两个不同的指令都使用这个辅助函数。我们使用两个字节作为跳转偏移量的操作数。一个16位的偏移量可以让我们跳转65535个字节的代码，这对于我们的需求来说应该足够了^2。

> The function returns the offset of the emitted instruction in the chunk. After compiling the then branch, we take that offset and pass it to this:

该函数会返回生成的指令在字节码块中的偏移量。编译完then分支后，我们将这个偏移量传递给这个函数：

compiler.c，在emitConstant()方法后添加代码：

```
static void patchJump(int offset) {
  // -2 to adjust for the bytecode for the jump offset itself.
  int jump = currentChunk()->count - offset - 2;

  if (jump > UINT16_MAX) {
    error("Too much code to jump over.");
  }

  currentChunk()->code[offset] = (jump >> 8) & 0xff;
  currentChunk()->code[offset + 1] = jump & 0xff;
}
```

> This goes back into the bytecode and replaces the operand at the given location with the calculated jump offset. We call `patchJump()` right before we emit the next instruction that we want the jump to land on, so it uses the current bytecode count to determine how far to jump. In the case of an `if` statement, that means right after we compile the then branch and before we compile the next statement.

这个函数会返回到字节码中，并将给定位置的操作数替换为计算出的跳转偏移量。我们在生成下一条希望跳转的指令之前调用`patchJump()`，因此会使用当前字节码计数来确定要跳转的距离。在`if`语句的情况下，就是在编译完then分支之后，并在编译下一个语句之前。

> That's all we need at compile time. Let's define the new instruction.

这就是在编译时需要做的。让我们来定义新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_PRINT,
  // 新增部分开始
  OP_JUMP_IF_FALSE,
  // 新增部分结束
  OP_RETURN,
```

> Over in the VM, we get it working like so:

在虚拟机中，我们让它这样工作：

vm.c，在run()方法中添加代码：

```
        break;
      }
      // 新增部分开始
      case OP_JUMP_IF_FALSE: {
        uint16_t offset = READ_SHORT();
        if (isFalsey(peek(0))) vm.ip += offset;
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> This is the first instruction we've added that takes a 16-bit operand. To read that from the chunk, we use a new macro.

这是我们添加的第一个需要16位操作数的指令。为了从字节码块中读出这个指令，需要使用一个新的宏。

vm.c，在run()方法中添加代码：

```
#define READ_CONSTANT() (vm.chunk->constants.values[READ_BYTE()])
// 新增部分开始
#define READ_SHORT() \
    (vm.ip += 2, (uint16_t)((vm.ip[-2] << 8) | vm.ip[-1]))
// 新增部分结束
#define READ_STRING() AS_STRING(READ_CONSTANT())
```

> It yanks the next two bytes from the chunk and builds a 16-bit unsigned integer out of them. As usual, we clean up our macro when we're done with it.

它从字节码块中抽取接下来的两个字节，并从中构建出一个16位无符号整数。和往常一样，当我们结束之后要清理宏。

vm.c，在run()方法中添加代码：

```
#undef READ_BYTE
// 新增部分开始
#undef READ_SHORT
// 新增部分结束
#undef READ_CONSTANT
```

> After reading the offset, we check the condition value on top of the stack. If it's falsey, we apply this jump offset to the `ip`. Otherwise, we leave the `ip` alone and execution will automatically proceed to the next instruction following the jump instruction.

读取偏移量之后，我们检查栈顶的条件值。如果是假，我们就将这个跳转偏移量应用到`ip`上。否则，我们就保持`ip`不变，执行会自动进入跳转指令的下一条指令。

> In the case where the condition is falsey, we don't need to do any other work. We've offset the `ip`, so when the outer instruction dispatch loop turns again, it will pick up execution at that new instruction, past all of the code in the then branch.

在条件为假的情况下，我们不需要做任何其它工作。我们已经移动了`ip`，所以当外部指令调度循环再次启动时，将会在新指令处执行，跳过了then分支的所有代码^3。

> Note that the jump instruction doesn't pop the condition value off the stack. So we aren't totally done here, since this leaves an extra value floating around on the stack. We'll clean that up soon. Ignoring

> that for the moment, we do have a working `if` statement in Lox now, with only one little instruction required to support it at runtime in the VM.

请注意，跳转指令并没有将条件值弹出栈。因此，我们在这里还没有全部完成，因为还在堆栈上留下了一个额外的值。我们很快就会把它清理掉。暂时先忽略这个问题，我们现在在Lox中已经有了可用的`if`语句，只需要一条小指令在虚拟机运行时支持它。

> ## 23.1.1 Else clauses

23.1.1 Else子句

> An `if` statement without support for `else` clauses is like Morticia Addams without Gomez. So, after we compile the then branch, we look for an `else` keyword. If we find one, we compile the else branch.

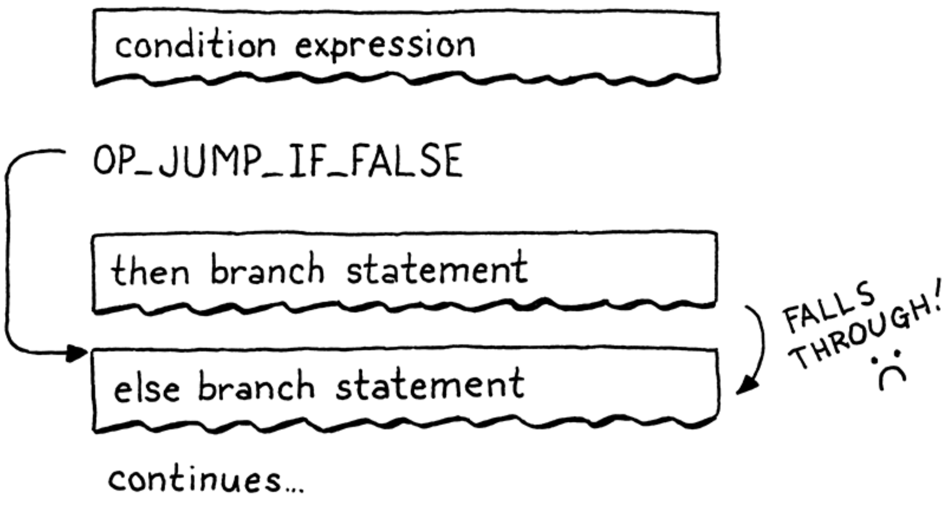
一个不支持`else`子句的`if`语句就像没有Gomez的Morticia Addams（《亚当斯一家》）。因此，在我们编译完then分支之后，我们要寻找`else`关键字。如果找到了，则编译else分支。

compiler.c，在ifStatement()方法中添加代码：

```
  patchJump(thenJump);
  // 新增部分开始
  if (match(TOKEN_ELSE)) statement();
  // 新增部分结束
}
```

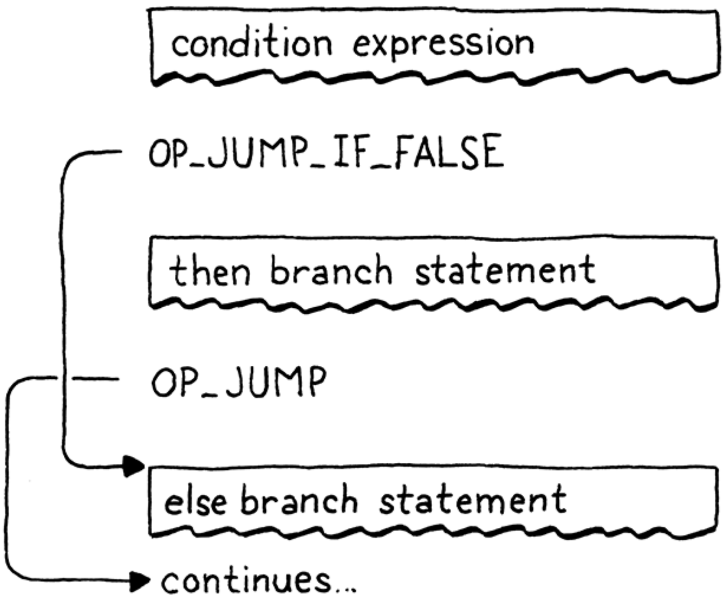
> When the condition is falsey, we'll jump over the then branch. If there's an else branch, the `ip` will land right at the beginning of its code. But that's not enough, though. Here's the flow that leads to:

当条件为假时，我们会跳过then分支。如果存在else分支，`ip`就会出现在其字节码的开头处。但这还不够。下面是对应的流：



> If the condition is truthy, we execute the then branch like we want. But after that, execution rolls right on through into the else branch. Oops! When the condition is true, after we run the then branch, we need to jump over the else branch. That way, in either case, we only execute a single branch, like this:

如果条件是真，则按照要求执行then分支。但在那之后，执行会直接转入到else分支。糟糕！当条件为真时，执行完then分支后，我们需要跳过else分支。这样，无论哪种情况，我们都只执行一个分支，像这样：



> To implement that, we need another jump from the end of the then branch.

为了实现这一点，我们需要从then分支的末端再进行一次跳转。

compiler.c，在ifStatement()方法中添加代码：

```
  statement();
  // 新增部分开始
  int elseJump = emitJump(OP_JUMP);
  // 新增部分结束
  patchJump(thenJump);
```

> We patch that offset after the end of the else body.

我们在else主体结束后修补这个偏移量。

compiler.c，在ifStatement()方法中添加代码：

```
  if (match(TOKEN_ELSE)) statement();
  // 新增部分开始
  patchJump(elseJump);
  // 新增部分结束
}
```

> After executing the then branch, this jumps to the next statement after the else branch. Unlike the other jump, this jump is unconditional. We always take it, so we need another instruction that expresses that.

在执行完then分支后，会跳转到else分支之后的下一条语句。与其它跳转不同，这个跳转是无条件的。我们一定会接受该跳转，所以我们需要另一条指令来表达它。

chunk.h，在枚举OpCode中添加代码：

```
  OP_PRINT,
  // 新增部分开始
  OP_JUMP,
  // 新增部分结束
  OP_JUMP_IF_FALSE,
```

> We interpret it like so:

我们这样来解释它：

vm.c，在run()方法中添加代码：

```
        break;
      }
      // 新增部分开始
      case OP_JUMP: {
        uint16_t offset = READ_SHORT();
        vm.ip += offset;
        break;
      }
      // 新增部分结束
      case OP_JUMP_IF_FALSE: {
```

> Nothing too surprising here—the only difference is that it doesn't check a condition and always applies the offset.

这里没有什么特别出人意料的——唯一的区别就是它不检查条件，并且一定会应用偏移量。

> We have then and else branches working now, so we're close. The last bit is to clean up that condition value we left on the stack. Remember, each statement is required to have zero stack effect—after the statement is finished executing, the stack should be as tall as it was before.

我们现在有了then和else分支，所以已经接近完成了。最后一点是清理我们遗留在栈上的条件值。请记住，每个语句都要求是0堆栈效应——在语句执行完毕后，堆栈应该与之前一样高。

> We could have the `OP_JUMP_IF_FALSE` instruction pop the condition itself, but soon we'll use that same instruction for the logical operators where we don't want the condition popped. Instead, we'll have the compiler emit a couple of explicit `OP_POP` instructions when compiling an `if` statement. We need to take care that every execution path through the generated code pops the condition.

我们可以让`OP_JUMP_IF_FALSE`指令自身弹出条件值，但很快我们会对不希望弹出条件值的逻辑运算符使用相同的指令。相对地，我们在编译`if`语句时，会让编译器生成几条显式的`OP_POP`指令，我们需要注意生成的代码中的每一条执行路径都要弹出条件值。

> When the condition is truthy, we pop it right before the code inside the then branch.

当条件为真时，我们会在进入then分支的代码前弹出该值。

compiler.c，在ifStatement()方法中添加代码：

```
  int thenJump = emitJump(OP_JUMP_IF_FALSE);
  // 新增部分开始
  emitByte(OP_POP);
  // 新增部分结束
  statement();
```

> Otherwise, we pop it at the beginning of the else branch.

否则，我们就在else分支的开头弹出它。

compiler.c，在ifStatement()方法中添加代码：

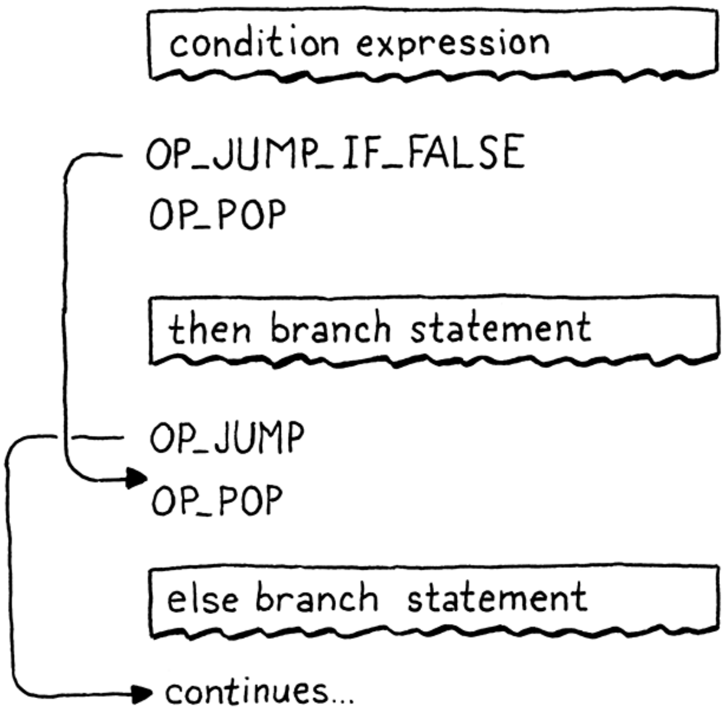
```
  patchJump(thenJump);
  // 新增部分开始
  emitByte(OP_POP);
  // 新增部分结束
  if (match(TOKEN_ELSE)) statement();
```

> This little instruction here also means that every `if` statement has an implicit else branch even if the user didn't write an `else` clause. In the case where they left it off, all the branch does is discard the condition value.

这里的这个小指令也意味着每个`if`语句都有一个隐含的else分支，即使用户没有写`else`子句。在用户没有写else子句的情况下，这个分支所做的就是丢弃条件值。

> The full correct flow looks like this:

完整正确的流看起来是这样的：



> If you trace through, you can see that it always executes a single branch and ensures the condition is popped first. All that remains is a little disassembler support.

如果你跟踪整个过程，可以看到它总是只执行一个分支，并确保条件值首先被弹出。剩下的就是一点反汇编程序的支持了。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return simpleInstruction("OP_PRINT", offset);
    // 新增部分开始
    case OP_JUMP:
      return jumpInstruction("OP_JUMP", 1, chunk, offset);
    case OP_JUMP_IF_FALSE:
      return jumpInstruction("OP_JUMP_IF_FALSE", 1, chunk, offset);
    // 新增部分结束
    case OP_RETURN:
```

> These two instructions have a new format with a 16-bit operand, so we add a new utility function to disassemble them.

这两条指令具有新格式，有着16位的操作数，因此我们添加了一个新的工具函数来反汇编它们。

debug.c，在byteInstruction()方法后添加代码：

```
static int jumpInstruction(const char* name, int sign,
                           Chunk* chunk, int offset) {
  uint16_t jump = (uint16_t)(chunk->code[offset + 1] << 8);
  jump |= chunk->code[offset + 2];
  printf("%-16s %4d -> %d\n", name, offset,
         offset + 3 + sign * jump);
  return offset + 3;
}
```

> There we go, that’s one complete control flow construct. If this were an ’80s movie, the montage music would kick in and the rest of the control flow syntax would take care of itself. Alas, the ’80s are long over, so we’ll have to grind it out ourselves.

就这样，这就是一个完整的控制流结构。如果这是一部80年代的电影，蒙太奇音乐就该响起了，剩下的控制流语法就会自行完成。唉，80年代已经过去很久了，所以我们得自己打磨了。

> ## 23.2 Logical Operators

23.2 逻辑运算符

> You probably remember this from jlox, but the logical operators `and` and `or` aren’t just another pair of binary operators like `+` and `-`. Because they short-circuit and may not evaluate their right operand depending on the value of the left one, they work more like control flow expressions.

你可能还记得jlox中的实现，但是逻辑运算符`and`和`or`并不仅仅是另一对像`+`和`-`一样的二元运算符。因为它们是短路的，根据左操作数的值，有可能不会对右操作数求值，它们的工作方式 更像是控制流表达式。

> They’re basically a little variation on an `if` statement with an `else` clause. The easiest way to explain them is to just show you the compiler code and the control flow it produces in the resulting bytecode. Starting with `and`, we hook it into the expression parsing table here:

它们基本上是带有`else`子句的`if`语句的小变体。解释它们的最简单的方法是向你展示编译器代码以及它在字节码中生成的控制流。从`and`开始，我们把它挂接到表达式解析表中：

compiler.c，替换1行：

```
  [TOKEN_NUMBER]        = {number,   NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_AND]           = {NULL,     and_,   PREC_AND},
  // 替换部分结束
  [TOKEN_CLASS]         = {NULL,     NULL,   PREC_NONE},
```

> That hands off to a new parser function.

这就交给了一个新的解析器函数。

compiler.c，在defineVariable()方法后添加代码：

```
static void and_(bool canAssign) {
  int endJump = emitJump(OP_JUMP_IF_FALSE);

  emitByte(OP_POP);
  parsePrecedence(PREC_AND);
```

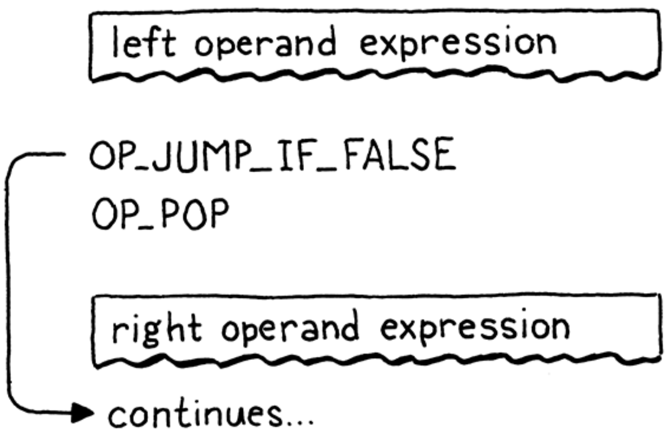
```
    patchJump(endJump);
  }
```

> At the point this is called, the left-hand side expression has already been compiled. That means at runtime, its value will be on top of the stack. If that value is falsey, then we know the entire and must be false, so we skip the right operand and leave the left-hand side value as the result of the entire expression. Otherwise, we discard the left-hand value and evaluate the right operand which becomes the result of the whole and expression.

在这个方法被调用时，左侧的表达式已经被编译了。这意味着，在运行时，它的值将会在栈顶。如果这个值为假，我们就知道整个and表达式的结果一定是假，所以我们跳过右边的操作数，将左边的值作为整个表达式的结果。否则，我们就丢弃左值，计算右操作数，并将它作为整个and表达式的结果。

> Those four lines of code right there produce exactly that. The flow looks like this:

这四行代码正是产生这样的结果。流程看起来像这样：



> Now you can see why OP_JUMP_IF_FALSE leaves the value on top of the stack. When the left-hand side of the and is falsey, that value sticks around to become the result of the entire expression.

现在你可以看到为什么OP_JUMP_IF_FALSE要将值留在栈顶。当and左侧的值为假时，这个值会保留下来，成为整个表达式的结果^4。

> ### 23.2.1 Logical or operator

23.2.1 逻辑or运算符

> The or operator is a little more complex. First we add it to the parse table.

or运算符有点复杂。首先，我们将它添加到解析表中。

compiler.c，替换1行：

```
  [TOKEN_NIL]           = {literal,  NULL,    PREC_NONE},
  // 替换部分开始
  [TOKEN_OR]            = {NULL,     or_,     PREC_OR},
  // 替换部分结束
  [TOKEN_PRINT]         = {NULL,     NULL,    PREC_NONE},
```

> When that parser consumes an infix or token, it calls this:

当解析器处理中缀or标识时，会调用这个：

compiler.c，在number()方法后添加代码：

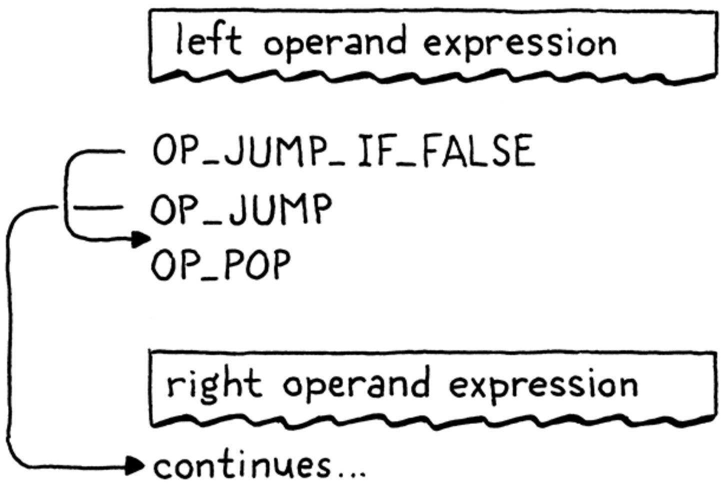
```
static void or_(bool canAssign) {
  int elseJump = emitJump(OP_JUMP_IF_FALSE);
  int endJump = emitJump(OP_JUMP);

  patchJump(elseJump);
  emitByte(OP_POP);

  parsePrecedence(PREC_OR);
  patchJump(endJump);
}
```

> In an or expression, if the left-hand side is *truthy*, then we skip over the right operand. Thus we need to jump when a value is truthy. We could add a separate instruction, but just to show how our compiler is free to map the language's semantics to whatever instruction sequence it wants, I implemented it in terms of the jump instructions we already have.

在or表达式中，如果左侧值为*真*，那么我们就跳过右侧的操作数。因此，当值为真时，我们需要跳过。我们可以添加一条单独的指令，但为了说明编译器如何自由地将语言的语义映射为它想要的任何指令序列，我会使用已有的跳转指令来实现它。

> When the left-hand side is falsey, it does a tiny jump over the next statement. That statement is an unconditional jump over the code for the right operand. This little dance effectively does a jump when the value is truthy. The flow looks like this:

当左侧值为假时，它会做一个小跳跃，跳过下一条语句。该语句会无条件跳过右侧操作数的代码。当值为真时，就会进行该跳转。流程看起来是这样的：



> If I'm honest with you, this isn't the best way to do this. There are more instructions to dispatch and more overhead. There's no good reason why or should be slower than and. But it is kind of fun to see

> that it’s possible to implement both operators without adding any new instructions. Forgive me my indulgences.

说实话，这并不是最好的方法。（这种方式中）需要调度的指令更多，开销也更大。没有充足的理由说明为什么`or`要比`and`慢。但是，可以在不增加任何新指令的前提下实现两个运算符，这是有趣的。请原谅我的放纵。

> OK, those are the three *branching* constructs in Lox. By that, I mean, these are the control flow features that only jump *forward* over code. Other languages often have some kind of multi-way branching statement like `switch` and maybe a conditional expression like `?:`, but Lox keeps it simple.

好了，这就是Lox中的三个分支结构。我的意思是，这些控制流特性只能在代码上*向前*跳转。其它语言中通常有某种多路分支语句，如`switch`，也许还有条件表达式`?:`，但Lox保持简单。

> ## 23.3 While Statements

23.3 While语句

> That takes us to the *looping* statements, which jump *backward* so that code can be executed more than once. Lox only has two loop constructs, `while` and `for`. A `while` loop is (much) simpler, so we start the party there.

这就将我们带到了*循环*语句，循环语句会向后跳转，使代码可以多次执行。Lox只有两种循环结构`while`和`for`。`while`循环要简单（得多），所以我们从这里开始。

compiler.c，在statement()方法中添加代码：

```
    ifStatement();
  // 新增部分开始
  } else if (match(TOKEN_WHILE)) {
    whileStatement();
  // 新增部分结束
  } else if (match(TOKEN_LEFT_BRACE)) {
```

> When we reach a `while` token, we call:

当我们遇到`while`标识时，调用：

compiler.c，在printStatement()方法后添加代码：

```
static void whileStatement() {
  consume(TOKEN_LEFT_PAREN, "Expect '(' after 'while'.");
  expression();
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after condition.");

  int exitJump = emitJump(OP_JUMP_IF_FALSE);
  emitByte(OP_POP);
  statement();

  patchJump(exitJump);
```

```
    emitByte(OP_POP);
  }
```

> Most of this mirrors `if` statements—we compile the condition expression, surrounded by mandatory parentheses. That's followed by a jump instruction that skips over the subsequent body statement if the condition is falsey.

大部分跟`if`语句相似——我们编译条件表达式（强制用括号括起来）。之后是一个跳转指令，如果条件为假，会跳过后续的主体语句。

> We patch the jump after compiling the body and take care to pop the condition value from the stack on either path. The only difference from an `if` statement is the loop. That looks like this:

我们在编译完主体之后对跳转指令进行修补，并注意在每个执行路径上都要弹出栈顶的条件值。与`if`语句的唯一区别就是循环^5。看起来像这样：

compiler.c，在whileStatement()方法中添加代码：

```
    statement();
    // 新增部分开始
    emitLoop(loopStart);
    // 新增部分结束
    patchJump(exitJump);
```

> After the body, we call this function to emit a "loop" instruction. That instruction needs to know how far back to jump. When jumping forward, we had to emit the instruction in two stages since we didn't know how far we were going to jump until after we emitted the jump instruction. We don't have that problem now. We've already compiled the point in code that we want to jump back to—it's right before the condition expression.

在主体之后，我们调用这个函数来生成一个"循环"指令。该指令需要知道往回跳多远。当向前跳时，我们必须分两个阶段发出指令，因为在发出跳跃指令前，我们不知道要跳多远。现在我们没有这个问题了。我们已经编译了要跳回去的代码位置——就在条件表达式之前。

> All we need to do is capture that location as we compile it.

我们所需要做的就是在编译时捕获这个位置。

compiler.c，在whileStatement()方法中添加代码：

```
  static void whileStatement() {
    // 新增部分开始
    int loopStart = currentChunk()->count;
    // 新增部分结束
    consume(TOKEN_LEFT_PAREN, "Expect '(' after 'while'.");
```

> After executing the body of a `while` loop, we jump all the way back to before the condition. That way, we re-evaluate the condition expression on each iteration. We store the chunk's current instruction count in `loopStart` to record the offset in the bytecode right before the condition expression we're about to compile. Then we pass that into this helper function:

在执行完`while`循环后，我们会一直跳到条件表达式之前。这样，我们就可以在每次迭代时都重新对条件表达式求值。我们在`loopStar`中存储字节码块中当前的指令数，作为我们即将编译的条件表达式在字节码中的偏移量。然后我们将该值传给这个辅助函数：

compiler.c，在emitBytes()方法后添加代码：

```c
static void emitLoop(int loopStart) {
  emitByte(OP_LOOP);

  int offset = currentChunk()->count - loopStart + 2;
  if (offset > UINT16_MAX) error("Loop body too large.");

  emitByte((offset >> 8) & 0xff);
  emitByte(offset & 0xff);
}
```

> It's a bit like `emitJump()` and `patchJump()` combined. It emits a new loop instruction, which unconditionally jumps *backwards* by a given offset. Like the jump instructions, after that we have a 16-bit operand. We calculate the offset from the instruction we're currently at to the `loopStart` point that we want to jump back to. The `+ 2` is to take into account the size of the `OP_LOOP` instruction's own operands which we also need to jump over.

这有点像`emitJump()`和`patchJump()` 的结合。它生成一条新的循环指令，该指令会无条件地*向回*跳转给定的偏移量。和跳转指令一样，其后还有一个16位的操作数。我们计算当前指令到我们想要跳回的`loopStart`之间的偏移量。`+2`是考虑到了`OP_LOOP`指令自身操作数的大小，这个操作数我们也需要跳过。

> From the VM's perspective, there really is no semantic difference between `OP_LOOP` and `OP_JUMP`. Both just add an offset to the `ip`. We could have used a single instruction for both and given it a signed offset operand. But I figured it was a little easier to sidestep the annoying bit twiddling required to manually pack a signed 16-bit integer into two bytes, and we've got the opcode space available, so why not use it?

从虚拟机的角度看，`OP_LOOP` 和`OP_JUMP`之间实际上没有语义上的区别。两者都只是在`ip`上加了一个偏移量。我们本可以用一条指令来处理这两者，并给该指令传入一个有符号的偏移量操作数。但我认为，这样做更容易避免手动将一个有符号的16位整数打包到两个字节所需要的烦人的位操作，况且我们有可用的操作码空间，为什么不使用呢？

> The new instruction is here:

新指令如下：

chunk.h，在枚举OpCode中添加代码：

```
  OP_JUMP_IF_FALSE,
  // 新增部分开始
  OP_LOOP,
  // 新增部分结束
  OP_RETURN,
```

> And in the VM, we implement it thusly:

在虚拟机中，我们这样实现它：

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_LOOP: {
        uint16_t offset = READ_SHORT();
        vm.ip -= offset;
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> The only difference from `OP_JUMP` is a subtraction instead of an addition. Disassembly is similar too.

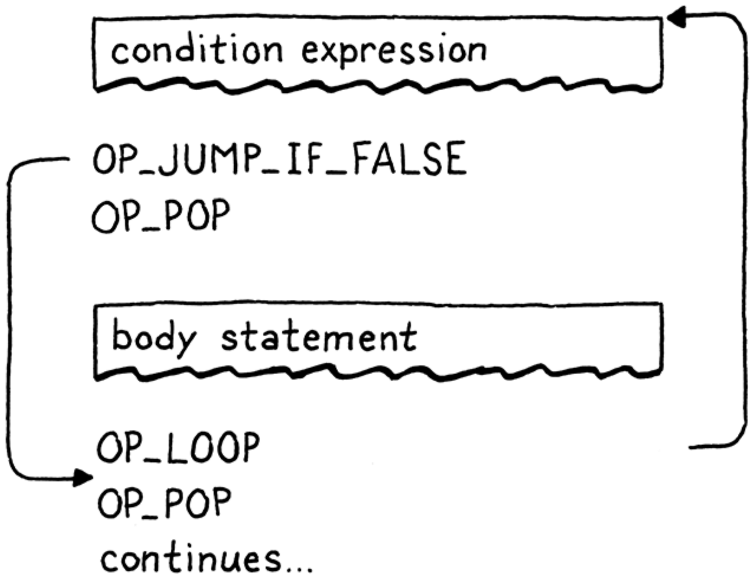
与`OP_JUMP`唯一的区别就是这里使用了减法而不是加法。反汇编也是相似的。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return jumpInstruction("OP_JUMP_IF_FALSE", 1, chunk, offset);
    // 新增部分开始
    case OP_LOOP:
      return jumpInstruction("OP_LOOP", -1, chunk, offset);
    // 新增部分结束
    case OP_RETURN:
```

> That's our `while` statement. It contains two jumps—a conditional forward one to escape the loop when the condition is not met, and an unconditional loop backward after we have executed the body. The flow looks like this:

这就是我们的`while`语句。它包含两个跳转——一个是有条件的前向跳转，用于在不满足条件的时候退出循环；另一个是在执行完主体代码后的无条件跳转。流程看起来如下：



23.4 For Statements

23.4 For语句

> The other looping statement in Lox is the venerable `for` loop, inherited from C. It's got a lot more going on with it compared to a `while` loop. It has three clauses, all of which are optional:

Lox中的另一个循环语句是古老的`for`循环，继承自C语言。与`while`循环相比，它有着更多的功能。它有三个子句，都是可选的：

> - The initializer can be a variable declaration or an expression. It runs once at the beginning of the statement.
> - The condition clause is an expression. Like in a `while` loop, we exit the loop when it evaluates to something falsey.
> - The increment expression runs once at the end of each loop iteration.

- 初始化器可以是一个变量声明或一个表达式。它会在整个语句的开头运行一次。
- 条件子句是一个表达式。就像`while`循环一样，如果其计算结果为假，就退出循环。
- 增量表达式在每次循环迭代结束时运行一次。

> In jlox, the parser desugared a `for` loop to a synthesized AST for a `while` loop with some extra stuff before it and at the end of the body. We'll do something similar, though we won't go through anything like an AST. Instead, our bytecode compiler will use the jump and loop instructions we already have.

在jlox中，解析器将`for`循环解构为一个`while`循环与其主体前后的一些额外内容的合成AST。我们会做一些类似的事情，不过我们不会使用AST之类的东西。相反，我们的字节码编译器将使用我们已有的跳转和循环指令。

> We'll work our way through the implementation a piece at a time, starting with the `for` keyword.

我们将从`for`关键字开始，逐步完成整个实现。

compiler.c，在statement()方法中添加代码：

```
    printStatement();
  // 新增部分开始
  } else if (match(TOKEN_FOR)) {
    forStatement();
  // 新增部分结束
  } else if (match(TOKEN_IF)) {
```

> It calls a helper function. If we only supported `for` loops with empty clauses like `for (;;)`, then we could implement it like this:

它会调用一个辅助函数。如果我们只支持`for(;;)`这样带有空子句的`for`循环，那么我们可以这样实现它：

compiler.c，在expressionStatement()方法后添加代码：

```
static void forStatement() {
  consume(TOKEN_LEFT_PAREN, "Expect '(' after 'for'.");
  consume(TOKEN_SEMICOLON, "Expect ';'.");

  int loopStart = currentChunk()->count;
  consume(TOKEN_SEMICOLON, "Expect ';'.");
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after for clauses.");

  statement();
  emitLoop(loopStart);
}
```

> There's a bunch of mandatory punctuation at the top. Then we compile the body. Like we did for `while` loops, we record the bytecode offset at the top of the body and emit a loop to jump back to that point after it. We've got a working implementation of infinite loops now.

首先是一堆强制性的标点符号。然后我们编译主体。与`while`循环一样，我们在主体的顶部记录字节码的偏移量，并在之后生成一个循环指令跳回该位置。现在我们已经有了一个无限循环的有效实现。

> ### 23.4.1 Initializer clause

23.4.1 初始化子句

> Now we'll add the first clause, the initializer. It executes only once, before the body, so compiling is straightforward.

现在我们要添加第一个子句，初始化器。它只在主体之前执行一次，因此编译很简单。

compiler.c，在forStatement()方法中替换1行：

```
  consume(TOKEN_LEFT_PAREN, "Expect '(' after 'for'.");
  // 替换部分开始
  if (match(TOKEN_SEMICOLON)) {
    // No initializer.
  } else if (match(TOKEN_VAR)) {
    varDeclaration();
  } else {
    expressionStatement();
  }
  // 替换部分结束
  int loopStart = currentChunk()->count;
```

> The syntax is a little complex since we allow either a variable declaration or an expression. We use the presence of the var keyword to tell which we have. For the expression case, we call expressionStatement() instead of expression(). That looks for a semicolon, which we need here too, and also emits an OP_POP instruction to discard the value. We don't want the initializer to leave anything on the stack.

语法有点复杂，因为我们允许出现变量声明或表达式。我们通过是否存在`var`关键字来判断是哪种类型。对于表达式，我们调用`expressionStatement()`而不是`expression()`。它会查找分号（我们这里也需要一个分号），并生成一个`OP_POP`指令来丢弃表达式的值。我们不希望初始化器在堆栈中留下任何东西。

> If a for statement declares a variable, that variable should be scoped to the loop body. We ensure that by wrapping the whole statement in a scope.

如果`for`语句声明了一个变量，那么该变量的作用域应该限制在循环体中。我们通过将整个语句包装在一个作用域中来确保这一点。

compiler.c，在forStatement()方法中添加代码：

```
  static void forStatement() {
    // 新增部分开始
    beginScope();
    // 新增部分结束
    consume(TOKEN_LEFT_PAREN, "Expect '(' after 'for'.");
```

> Then we close it at the end.

然后我们在结尾关闭这个作用域。

compiler.c，在forStatement()方法中添加代码：

```
    emitLoop(loopStart);
    // 新增部分开始
    endScope();
    // 新增部分结束
  }
```

> ## 23.4.2 Condition clause

23.4.2 条件子句

> Next, is the condition expression that can be used to exit the loop.

接下来，是可以用来退出循环的条件表达式。

compiler.c，在forStatement()方法中替换1行：

```
  int loopStart = currentChunk()->count;
  // 替换部分开始
  int exitJump = -1;
  if (!match(TOKEN_SEMICOLON)) {
    expression();
    consume(TOKEN_SEMICOLON, "Expect ';' after loop condition.");

    // Jump out of the loop if the condition is false.
    exitJump = emitJump(OP_JUMP_IF_FALSE);
    emitByte(OP_POP); // Condition.
  }
  // 替换部分结束
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after for clauses.");
```

> Since the clause is optional, we need to see if it's actually present. If the clause is omitted, the next token must be a semicolon, so we look for that to tell. If there isn't a semicolon, there must be a condition expression.

因为子句是可选的，我们需要查看它是否存在。如果子句被省略，下一个标识一定是分号，所以我们通过查找分号来进行判断。如果没有分号，就一定有一个条件表达式。

> In that case, we compile it. Then, just like with while, we emit a conditional jump that exits the loop if the condition is falsey. Since the jump leaves the value on the stack, we pop it before executing the body. That ensures we discard the value when the condition is true.

在这种情况下，我们对它进行编译。然后，就像`while`一样，我们生成一个条件跳转指令，如果条件为假则退出循环。因为跳转指令将值留在了栈上，我们在执行主体之前将值弹出。这样可以确保当条件值为真时，我们会丢弃这个值。

> After the loop body, we need to patch that jump.

在循环主体之后，我们需要修补跳转指令。

compiler.c，在forStatement()方法中添加代码：

```
  emitLoop(loopStart);
  // 新增部分开始
  if (exitJump != -1) {
    patchJump(exitJump);
    emitByte(OP_POP); // Condition.
  }
  // 新增部分结束
  endScope();
}
```

> We do this only when there is a condition clause. If there isn't, there's no jump to patch and no condition value on the stack to pop.

我们只在有条件子句的时候才会这样做。如果没有条件子句，就没有需要修补的跳转指令，堆栈中也没有条件值需要弹出。

> 23.4.3 Increment clause

23.4.3 增量子句

> I've saved the best for last, the increment clause. It's pretty convoluted. It appears textually before the body, but executes *after* it. If we parsed to an AST and generated code in a separate pass, we could simply traverse into and compile the `for` statement AST's body field before its increment clause.

我把非常复杂的增量子句部分留到最后。从文本上看，它出现在循环主体之前，但却是在主体*之后*执行。如果我们将其解析为AST，并在单独的处理过程中生成代码，就可以简单地遍历并编译`for`语句AST的主体字段，然后再编译其增量子句。

> Unfortunately, we can't compile the increment clause later, since our compiler only makes a single pass over the code. Instead, we'll *jump over* the increment, run the body, jump *back* up to the increment, run it, and then go to the next iteration.

不幸的是，我们不能稍后再编译增量子句，因为我们的编译器只对代码做了一次遍历。相对地，我们会*跳过*增量子句，运行主体，*跳回*增量子句，运行它，然后进入下一个迭代。

> I know, a little weird, but hey, it beats manually managing ASTs in memory in C, right? Here's the code:

我知道，这有点奇怪，但是，这总比在C语言中手动管理内存中的AST要好，对吗？代码如下：

compiler.c，在forStatement()方法中替换1行：

```
  }
  // 替换部分开始
  if (!match(TOKEN_RIGHT_PAREN)) {
    int bodyJump = emitJump(OP_JUMP);
    int incrementStart = currentChunk()->count;
    expression();
    emitByte(OP_POP);
    consume(TOKEN_RIGHT_PAREN, "Expect ')' after for clauses.");

    emitLoop(loopStart);
    loopStart = incrementStart;
    patchJump(bodyJump);
  }
  // 替换部分结束
  statement();
```

> Again, it's optional. Since this is the last clause, when omitted, the next token will be the closing parenthesis. When an increment is present, we need to compile it now, but it shouldn't execute yet. So,

> first, we emit an unconditional jump that hops over the increment clause’s code to the body of the loop.

同样，它也是可选的。因为这是最后一个子句，下一个标识是右括号。当存在增加子句时，我们需要立即编译它，但是它还不应该执行。因此，首先我们生成一个无条件跳转指令，该指令会跳过增量子句的代码进入循环体中。

> Next, we compile the increment expression itself. This is usually an assignment. Whatever it is, we only execute it for its side effect, so we also emit a pop to discard its value.

接下来，我们编译增量表达式本身。这通常是一个赋值语句。不管它是什么，我们执行它只是为了它的副作用，所以我们也生成一个弹出指令丢弃该值。

> The last part is a little tricky. First, we emit a loop instruction. This is the main loop that takes us back to the top of the `for` loop—right before the condition expression if there is one. That loop happens right after the increment, since the increment executes at the end of each loop iteration.

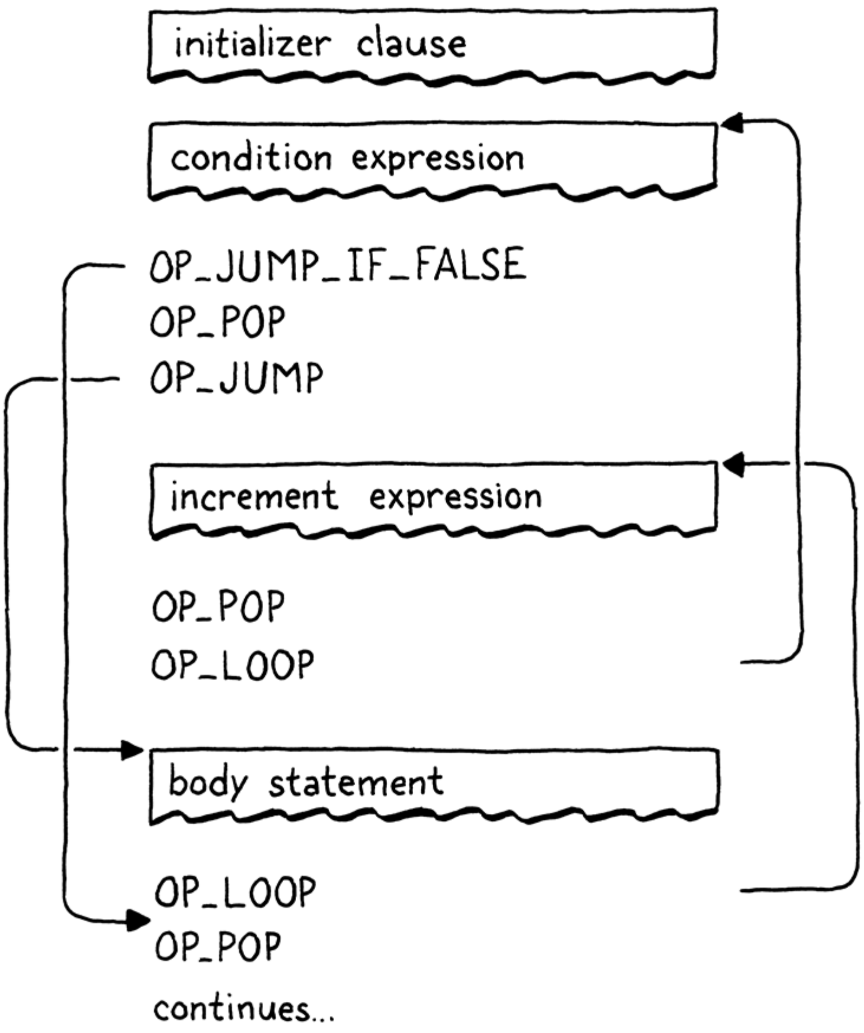
最后一部分有点棘手。首先，我们生成一个循环指令。这是主循环，会将我们带到`for`循环的顶部——如果有条件表达式的话，就回在它前面。这个循环发生在增量语句之后，因此增量语句是在每次循环迭代结束时执行的。

> Then we change `loopStart` to point to the offset where the increment expression begins. Later, when we emit the loop instruction after the body statement, this will cause it to jump up to the *increment* expression instead of the top of the loop like it does when there is no increment. This is how we weave the increment in to run after the body.

然后我们更改`loopStart`，指向增量表达式开始处的偏移量。之后，当我们在主体语句结束之后生成循环指令时，就会跳转到增量表达式，而不是像没有增量表达式时那样跳转到循环顶部。这就是我们如何在主体之后运行增量子句的办法。

> It’s convoluted, but it all works out. A complete loop with all the clauses compiles to a flow like this:

这很复杂，但一切都解决了。一个包含所有子句的完整循环会被编译为类似这样的流程：



> As with implementing `for` loops in jlox, we didn't need to touch the runtime. It all gets compiled down to primitive control flow operations the VM already supports. In this chapter, we've taken a big leap forward—clox is now Turing complete. We've also covered quite a bit of new syntax: three statements and two expression forms. Even so, it only took three new simple instructions. That's a pretty good effort-to-reward ratio for the architecture of our VM.

与jlox中实现`for`循环一样，我们不需要接触运行时。所有这些都被编译到虚拟机已经支持的原始控制流中。在这一章中，我们向前迈出了一大步——clox现在图灵完整了。我们还讨论了相当多的新语法：三种语句和两种表达式形式。即便如此，我们也只用了三个简单的新指令。对于我们的虚拟机架构来说，这是一个相当不错的努力-回报比。

^1: 你有没有注意到，`if`关键字后面的`(`实际上没有什么用处？如果没有它，语言也会很明确，而且容易解析，比如：
`if condition) print("looks weird");`
结尾的`)`是有用的，因为它将条件表达式和主体分隔开。有些语言使用`then`关键字来代替。但是开头的`(`没有任何作用。它之所以存在，是因为不匹配的括号在我们人类看来很糟糕。 ^2: 一些指令集中有单独的"长"跳转指令，这些指令会接受更大的操作数，当你需要跳转更远的距离时可以使用。 ^3: 我说过我们不会使用C的`if`语句来实现Lox的控制流，但我们在这里确实使用了`if`语句来决定是否偏移指令指针。但我们并没有真正使用C语言来实现控制流。如果我们想的话，可以用纯粹的算术做到同样的事情。假设我们有一个函数`falsey()`，它接受一个Lox Value，如果是假则返回1，否则返回0。那我们可以这样实现跳转指令：

```
case OP_JUMP_IF_FALSE: {
  uint16_t offset = READ_SHORT();
  vm.ip += falsey() * offset;
  break;
}
```

`falsey()`函数可能会使用一些控制流来处理不同的值类型，但这是该函数的实现细节，并不影响我们的虚拟机如何处理自己的控制流。 ^4: 我们的操作码范围中还有足够的空间，所以我们可以为隐式弹出值的条件跳转和不弹出值的条件跳转制定单独的指令。但我想尽量在书中保持简约。在你的字节码虚拟机中，值得探索添加更多的专用指令，看看它们是如何影响性能的。 ^5: 真的开始怀疑我对逻辑运算符使用相同的跳转指令的决定了。

习题

1. > In addition to `if` statements, most C-family languages have a multi-way `switch` statement. Add one to clox. The grammar is:

   除了`if`语句，大多数C家族语言都要一个多路`switch`语句。在clox中添加一个。语法如下：

   ```
   switchStmt     → "switch" "(" expression ")"
                    "{" switchCase* defaultCase? "}" ;
   switchCase     → "case" expression ":" statement* ;
   defaultCase    → "default" ":" statement* ;
   ```

   > To execute a `switch` statement, first evaluate the parenthesized switch value expression. Then walk the cases. For each case, evaluate its value expression. If the case value is equal to the switch value, execute the statements under the case and then exit the `switch` statement. Otherwise, try the next case. If no case matches and there is a `default` clause, execute its statements.
   >
   > To keep things simpler, we're omitting fallthrough and `break` statements. Each case automatically jumps to the end of the switch statement after its statements are done.

   为了执行`switch`语句，首先要计算括号内的switch值表达式。然后遍历分支。对于每个分支，计算其值表达式。如果case值等于switch值，就执行case下的语句，然后退出`switch`语句。否则，就尝试下一个case分支。如果没有匹配的分支，并且有`default`子句，就执行其中的语句。

   为了让事情更简单，我们省略了fall through和`break`语句。每个case子句在其语句完成后会自动跳转到switch语句的结尾。

2. > In jlox, we had a challenge to add support for `break` statements. This time, let's do `continue`:

   在jlox中，我们有一个习题是添加对`break`语句的支持。这一次，我们来做`continue`：

   ```
   continueStmt   → "continue" ";" ;
   ```

> A `continue` statement jumps directly to the top of the nearest enclosing loop, skipping the rest of the loop body. Inside a `for` loop, a `continue` jumps to the increment clause, if there is one. It's a compile-time error to have a `continue` statement not enclosed in a loop.
>
> Make sure to think about scope. What should happen to local variables declared inside the body of the loop or in blocks nested inside the loop when a `continue` is executed?

`continue`语句直接跳转到最内层的封闭循环的顶部，跳过循环体的其余部分。在`for`循环中，如果有增量子句，`continue`会跳到增量子句。如果`continue`子句没有被包含在循环中，则是一个编译时错误。

一定要考虑作用域问题。当执行`continue`语句时，在循环体内或嵌套在循环体中的代码块内声明的局部变量应该如何处理？

3. > Control flow constructs have been mostly unchanged since Algol 68. Language evolution since then has focused on making code more declarative and high level, so imperative control flow hasn't gotten much attention.
   >
   > For fun, try to invent a useful novel control flow feature for Lox. It can be a refinement of an existing form or something entirely new. In practice, it's hard to come up with something useful enough at this low expressiveness level to outweigh the cost of forcing a user to learn an unfamiliar notation and behavior, but it's a good chance to practice your design skills.

   自Algol 68以来，控制流结构基本没有变化。从那时起，语言的发展就专注于使代码更具有声明性和高层次，因此命令式控制流并没有得到太多的关注。

   为了好玩，可以试着为Lox发明一个有用的新的控制流功能。它可以是现有形式的改进，也可以是全新的东西。实践中，在这种较低的表达层次上，很难想出足够有用的东西来抵消迫使用户学习不熟悉的符号和行为的代价，但这是一个练习设计技能的好机会。

DESIGN NOTE: CONSIDERING GOTO HARMFUL

设计笔记：认为GOTO有害

> Discovering that all of our beautiful structured control flow in Lox is actually compiled to raw unstructured jumps is like the moment in Scooby Doo when the monster rips the mask off their face. It was goto all along! Except in this case, the monster is *under* the mask. We all know goto is evil. But . . . why?
>
> It is true that you can write outrageously unmaintainable code using goto. But I don't think most programmers around today have seen that first hand. It's been a long time since that style was common. These days, it's a boogie man we invoke in scary stories around the campfire.
>
> The reason we rarely confront that monster in person is because Edsger Dijkstra slayed it with his famous letter "Go To Statement Considered Harmful", published in *Communications of the ACM* (March, 1968). Debate around structured programming had been fierce for some time with adherents on both sides, but I think Dijkstra deserves the most credit for effectively ending it. Most new languages today have no unstructured jump statements.
>
> A one-and-a-half page letter that almost single-handedly destroyed a language feature must be pretty impressive stuff. If you haven't read it, I encourage you to do so. It's a seminal piece of computer

> science lore, one of our tribe's ancestral songs. Also, it's a nice, short bit of practice for reading academic CS writing, which is a useful skill to develop.
>
> That is, if you can get past Dijkstra's insufferable faux-modest self-aggrandizing writing style:
>
> > More recently I discovered why the use of the go to statement has such disastrous effects. . . . At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.
>
> Ah, yet another one of my many discoveries. I couldn't even be bothered to write it up until the clamoring masses begged me to.
>
> I've read it through a number of times, along with a few critiques, responses, and commentaries. I ended up with mixed feelings, at best. At a very high level, I'm with him. His general argument is something like this:
>
> 1. As programmers, we write programs—static text—but what we care about is the actual running program—its dynamic behavior.
> 2. We're better at reasoning about static things than dynamic things. (He doesn't provide any evidence to support this claim, but I accept it.)
> 3. Thus, the more we can make the dynamic execution of the program reflect its textual structure, the better.
>
> This is a good start. Drawing our attention to the separation between the code we write and the code as it runs inside the machine is an interesting insight. Then he tries to define a "correspondence" between program text and execution. For someone who spent literally his entire career advocating greater rigor in programming, his definition is pretty hand-wavey. He says:
>
> > Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?)
>
> Imagine it like this. You have two computers with the same program running on the exact same inputs—so totally deterministic. You pause one of them at an arbitrary point in its execution. What data would you need to send to the other computer to be able to stop it exactly as far along as the first one was?
>
> If your program allows only simple statements like assignment, it's easy. You just need to know the point after the last statement you executed. Basically a breakpoint, the `ip` in our VM, or the line number in an error message. Adding branching control flow like `if` and `switch` doesn't add any more to this. Even if the marker points inside a branch, we can still tell where we are.
>
> Once you add function calls, you need something more. You could have paused the first computer in the middle of a function, but that function may be called from multiple places. To pause the second machine at exactly the same point in *the entire program's* execution, you need to pause it on the *right* call to that function.

> So you need to know not just the current statement, but, for function calls that haven't returned yet, you need to know the locations of the callsites. In other words, a call stack, though I don't think that term existed when Dijkstra wrote this. Groovy.
>
> He notes that loops make things harder. If you pause in the middle of a loop body, you don't know how many iterations have run. So he says you also need to keep an iteration count. And, since loops can nest, you need a stack of those (presumably interleaved with the call stack pointers since you can be in loops in outer calls too).
>
> This is where it gets weird. So we're really building to something now, and you expect him to explain how goto breaks all of this. Instead, he just says:
>
> > The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress.
>
> He doesn't prove that this is hard, or say why. He just says it. He does say that one approach is unsatisfactory:
>
> > With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful.
>
> But . . . that's effectively what loop counters do, and he was fine with those. It's not like every loop is a simple "for every integer from 0 to 10" incrementing count. Many are `while` loops with complex conditionals.
>
> Taking an example close to home, consider the core bytecode execution loop at the heart of clox. Dijkstra argues that that loop is tractable because we can simply count how many times the loop has run to reason about its progress. But that loop runs once for each executed instruction in some user's compiled Lox program. Does knowing that it executed 6,201 bytecode instructions really tell us VM maintainers *anything* edifying about the state of the interpreter?
>
> In fact, this particular example points to a deeper truth. Böhm and Jacopini proved that *any* control flow using goto can be transformed into one using just sequencing, loops, and branches. Our bytecode interpreter loop is a living example of that proof: it implements the unstructured control flow of the clox bytecode instruction set without using any gotos itself.
>
> That seems to offer a counter-argument to Dijkstra's claim: you *can* define a correspondence for a program using gotos by transforming it to one that doesn't and then use the correspondence from that program, which—according to him—is acceptable because it uses only branches and loops.
>
> But, honestly, my argument here is also weak. I think both of us are basically doing pretend math and using fake logic to make what should be an empirical, human-centered argument. Dijkstra is right that some code using goto is really bad. Much of that could and should be turned into clearer code by using structured control flow.
>
> By eliminating goto completely from languages, you're definitely prevented from writing bad code using gotos. It may be that forcing users to use structured control flow and making it an uphill battle to write goto-like code using those constructs is a net win for all of our productivity.

> But I do wonder sometimes if we threw out the baby with the bathwater. In the absence of goto, we often resort to more complex structured patterns. The “switch inside a loop” is a classic one. Another is using a guard variable to exit out of a series of nested loops:
>
> ```
> // See if the matrix contains a zero.
> bool found = false;
> for (int x = 0; x < xSize; x++) {
>   for (int y = 0; y < ySize; y++) {
>     for (int z = 0; z < zSize; z++) {
>       if (matrix[x][y][z] == 0) {
>         printf("found");
>         found = true;
>         break;
>       }
>     }
>     if (found) break;
>   }
>   if (found) break;
> }
> ```
>
> Is that really better than:
>
> ```
> for (int x = 0; x < xSize; x++) {
>   for (int y = 0; y < ySize; y++) {
>     for (int z = 0; z < zSize; z++) {
>       if (matrix[x][y][z] == 0) {
>         printf("found");
>         goto done;
>       }
>     }
>   }
> }
> done:
> ```
>
> You could do this without `break` statements—themselves a limited goto-ish construct—by inserting `!found &&` at the beginning of the condition clause of each loop.
>
> I guess what I really don’t like is that we’re making language design and engineering decisions today based on fear. Few people today have any subtle understanding of the problems and benefits of goto. Instead, we just think it’s “considered harmful”. Personally, I’ve never found dogma a good starting place for quality creative work.

发现我们在Lox中的所有漂亮的结构化控制流实际上都被编译成原始的非结构化跳转，就像《Scooby Doo》中怪兽撕下脸上的面具一样。一直以来都是goto！只不过这一次，怪物藏在面具下。我们都知道goto是魔鬼。但是......为什么呢？

的确，你可以用goto编写极度难以维护的代码。但我认为现在的大多数程序员都没有亲身经历过这种情况。这种风格已经很久没有出现了。如今，它只是我们在篝火旁的恐怖故事里会提到的一个恶棍。

我们之所以很少亲自面对这个怪物，是因为Edsger Dijkstra用他那封著名的信件“Go To Statement Considered Harmful”杀死了它，这封信发表在《ACM通讯》(1968年3月刊)上。彼时围绕结构化编程的争论已经激烈了一段时间，双方都有支持者，但我认为Dijkstra最突出的贡献就是有效地结束了争论。今天的大多数新语言都没有非结构化的跳转语句。

一封一页半的信，几乎以一己之力摧毁了一种语言特性，这一定是相当令人印象深刻的东西。如果你还没有读过，我鼓励你读一下。它是计算机科学知识的开山之作，是我们部落的祖传歌曲之一。同时，它也是阅读学术性CS文章的一个很好的、简短的练习，这是一个很有用的技能。

【也就是说，你是否能克服Dijkstra那令人难以忍受的虚伪谦虚、自我吹嘘的写作风格：

> 最近，我发现了为什么goto语句是使用会产生灾难性的影响。……当时我并没有太重视这个发现；现在我把我的想法提交出来进行发表，是因为在最近关于这个问题的讨论中，有人敦促我这样做。

嗯，这是我众多发现中的又一项。我甚至懒得把它写下来，都是吵吵嚷嚷的群众求我写。】

我把它读了好几遍，还有一些批评、回复和评论。我最后的感受充其量是喜忧参半。在很高的层次上来说，我是支持他的。他的总体论点是这样的：

1. 作为程序员，我们编写程序——静态文本——但我们关心的是实际运行的程序——它的动态行为。
2. 相比之下，我们更擅长对静态事物进行推理，而不是动态事物。（他没有提供任何证据来支持这一说法，但我接受这个说法）
3. 因此，我们越能使程序的动态执行反映其文本结构，就越好。

这是一个良好的开端。让我们注意到编写的代码和机器内部运行的代码之间的分离是一个有趣的见解。然后，他试图在程序文本和执行之间定义一种“对应关系”。对于一个几乎在整个职业生涯中都倡导更严格的编程的人来说，他的定义是相当简单的。他说：

> 现在让我们考虑一下，如何能够描述一个过程的进展。（你可以用一种非常具体的方式来思考这个问题：假设一个过程，被看做是一系列操作的时间序列，在一个任意的操作之后停止，我们必须要固定哪些数据，才能重做整个过程，并达到完全相同的点）

想象一下这样的情况，你有两台计算机，在完全相同的输入上运行相同的程序，所以这是完全确定性的。在执行过程中，你可以在任意点暂停其中一个函数。你需要向另一台计算机发送什么数据才能让它完全像第一台那样暂停。

如果你的程序只允许像赋值这样的简单语句，这很容易。你只需要知道你执行的最后一条语句之后的那一个点。基本上就是一个断点，即我们虚拟机中的`ip`或错误信息中的行号。添加`if`和`switch`这样的分支控制流并不会改变什么。即时标记点指向分支内部，我们仍然可以知道我们在哪里。

一旦增加了函数调用，就需要更多的数据才行。你可以在函数中间位置暂停第一台计算机，但是该函数可能会从多个地方调用。要想在*整个程序执行*的同一时刻暂停第二台机器，你就需要在*正确*调用该函数的时机暂停它。

因此，你不仅需要知道当前的语句，而且，对于尚未返回的函数调用，你也需要知道调用点的位置。换句话说，就是调用堆栈，尽管我认为Dijkstra写这个的时候，这个术语还不存在。有趣。

他指出，循环使事情变得更难。如果你在循环体中间暂停，你就不知道运行了多少次迭代。所以他说你还需要记录迭代数。而且，由于循环可以嵌套，所以你需要一个堆栈（估计是与调用栈指针交错在一起，因为你也可能在外部调用的循环中）。

这就是奇怪的地方。所以，我们现在真的有了一些进展，你希望他解释goto是如何破坏这一切的。相反，他说：

> 无节制地使用goto语句会产生一个直接的后果，那就是很难找到一组有意义的坐标来描述进程的进展。

他没有证明这很困难，也没有说明原因。他就是这么说了一下。他确实说过有一种方法是无法令人满意的：

> 当然，有了goto语句，我们仍然可以通过一个计数器来唯一性地描述进程，计数器计算自程序启动以来所执行的操作的数量（即一种规范化的时钟）。困难的是，这样的坐标虽然是唯一的，但完全没有帮助。

但是……这就是循环计数器的作用，而且他对这些计数器很满意。并不是每个循环都是简单地“对于0到10的每个整数”的递增计数。许多是带有复杂条件的`while`循环。

举一个比较接近的例子，考虑一下clox中的核心字节码执行循环。Dijkstra认为这个循环很容易处理，因为我们可以简单地计算循环运行了多少次来推断它的进度。但是，对于某些用户编译的Lox程序中执行的每条指令，该循环都会运行一次。知道它执行了6201条字节码指令真的能告诉我们这些虚拟机维护者关于解释器状态的任何信息吗？

事实上，这个特殊的例子指出了一个更深层次的事实。Böhm和Jacopini证明，*任何*使用goto的控制流都可以转换为只使用排序、循环和分支的控制流。我们的字节码解释器核心循环就是一个活生生的例子：它实现了clox字节码指令集的非结构化控制流，而本身没有使用任何goto。

这似乎提供了一种反驳Dijkstra主张的论点：你可以为使用goto的程序定义一个对应关系，将其转换为不使用goto的程序，然后使用该程序的对应关系，根据他的说法，这是可接受的，因为它只使用了分支和循环。

但是，老实说，我的论点也很弱。我认为我们两个人基本上都在做假数学，用假逻辑来做一个应该是经验性的、以人为本的论证。Dijkstra是对的，一些使用goto的代码真的很糟糕。通过使用结构化控制流，其中的大部分内容可以也应该变成更清晰的代码。

从语言中消除goto，你肯定可以避免使用goto写出糟糕的代码。对我们所有的生产力来说，迫使用户使用结构化控制流，并使用这些结构写出类似goto的代码，可能是一场净胜。

但我有时会怀疑我们是否把孩子和洗澡水一起倒掉了。在没有goto的情况下，我们常常求助于更复杂的结构化模式。“循环中的分支”就是一个典型的例子。另一个例子是使用保护变量退出一系列的嵌套循环：

```
// See if the matrix contains a zero.
bool found = false;
for (int x = 0; x < xSize; x++) {
  for (int y = 0; y < ySize; y++) {
    for (int z = 0; z < zSize; z++) {
      if (matrix[x][y][z] == 0) {
        printf("found");
        found = true;
        break;
      }
    }
    if (found) break;
  }
  if (found) break;
}
```

【你可以在每个循环的条件子句的开头插入`!found &&`，而不需要使用`break`语句（它们本身就是一种有限的goto式结构）】

这真的比下面的形式好吗：

```
for (int x = 0; x < xSize; x++) {
  for (int y = 0; y < ySize; y++) {
    for (int z = 0; z < zSize; z++) {
      if (matrix[x][y][z] == 0) {
        printf("found");
        goto done;
      }
    }
  }
}
done:
```

我想我真正不喜欢的是，我们现在基于恐惧来进行语言设计和工程决策。如今，很少有人对goto的问题和好处有任何微妙的了解。相反，我们只是认为它“被认为是有害的”。就我个人而言，我从不觉得教条是高质量创造性工作的良好开端。

24.调用和函数 Calls and Functions

> Any problem in computer science can be solved with another level of indirection. Except for the problem of too many layers of indirection.
>
> ——David Wheeler

计算机科学中的任何问题都可以通过引入一个中间层来解决。除了中间层太多的问题。（David Wheeler）

> This chapter is a beast. I try to break features into bite-sized pieces, but sometimes you gotta swallow the whole meal. Our next task is functions. We could start with only function declarations, but that’s not very useful when you can’t call them. We could do calls, but there’s nothing to call. And all of the runtime support needed in the VM to support both of those isn’t very rewarding if it isn’t hooked up to anything you can see. So we’re going to do it all. It’s a lot, but we’ll feel good when we’re done.

这一章是一头猛兽。我试图把功能分解成小块，但有时候你不得不吞下整顿饭。我们的下一个任务是函数。我们可以只从函数声明开始，但是如果你不能调用它们，那就没什么用了。我们可以实现调用，但是也没什么可调用的。而且，为了实现这两个功能所需的所有运行时支持，如果不能与你能直观看到的东西相挂钩，就不是很有价值。所以我们都要做。虽然内容很多，但等我们完成时，我们会感觉很好。

> ## 24.1 Function Objects

24.1 函数对象

> The most interesting structural change in the VM is around the stack. We already *have* a stack for local variables and temporaries, so we’re partway there. But we have no notion of a *call* stack. Before we can make much progress, we’ll have to fix that. But first, let’s write some code. I always feel better once I

> start moving. We can’t do much without having some kind of representation for functions, so we’ll start there. From the VM’s perspective, what is a function?

虚拟机中最有趣的结构变化是围绕堆栈进行的。我们已经有了用于局部变量和临时变量的栈，所以我们已经完成了一半。但是我们还没有调用堆栈的概念。在我们取得更大进展之前，必须先解决这个问题。但首先，让我们编写一些代码。一旦开始行动，我就感觉好多了。如果没有函数的某种表示形式，我们就做不了太多事，所以我们先从这里开始。从虚拟机的角度来看，什么是函数？

> A function has a body that can be executed, so that means some bytecode. We could compile the entire program and all of its function declarations into one big monolithic Chunk. Each function would have a pointer to the first instruction of its code inside the Chunk.

函数有一个可以被执行的主体，也就是一些字节码。我们可以把整个程序和所有的函数声明编译成一个大的字节码块。每个函数都有一个指针指向其在字节码块中的第一条指令。

> This is roughly how compilation to native code works where you end up with one solid blob of machine code. But for our bytecode VM, we can do something a little higher level. I think a cleaner model is to give each function its own Chunk. We’ll want some other metadata too, so let’s go ahead and stuff it all in a struct now.

这大概就是编译为本地代码的工作原理，你最终得到的是一大堆机器码。但是对于我们的字节码虚拟机，我们可以做一些更高层次的事情。我认为一个更简洁的模型是给每个函数它自己的字节码块。我们还需要一些其它的元数据，所以我们现在来把它们塞进一个结构体中。

object.h，在结构体Obj后添加代码：

```
  struct Obj* next;
};
// 新增部分开始
typedef struct {
  Obj obj;
  int arity;
  Chunk chunk;
  ObjString* name;
} ObjFunction;
// 新增部分结束
struct ObjString {
```

> Functions are first class in Lox, so they need to be actual Lox objects. Thus ObjFunction has the same Obj header that all object types share. The `arity` field stores the number of parameters the function expects. Then, in addition to the chunk, we store the function’s name. That will be handy for reporting readable runtime errors.

函数是Lox中的一等公民，所以它们需要作为实际的Lox对象。因此，ObjFunction具有所有对象类型共享的Obj头。`arity`字段存储了函数所需要的参数数量。然后，除了字节码块，我们还需要存储函数名称。这有助于报告可读的运行时错误^1。

> This is the first time the “object” module has needed to reference Chunk, so we get an include.

这是“object”模块第一次需要引用Chunk，所以我们需要引入一下。

object.h，添加代码：

```
#include "common.h"
// 新增部分开始
#include "chunk.h"
// 新增部分结束
#include "value.h"
```

> Like we did with strings, we define some accessories to make Lox functions easier to work with in C. Sort of a poor man's object orientation. First, we'll declare a C function to create a new Lox function.

就像我们处理字符串一样，我们定义一些辅助程序，使Lox函数更容易在C语言中使用。有点像穷人版的面向对象。首先，我们会声明一个C函数来创建新Lox函数。

object.h，在结构体ObjString后添加代码：

```
  uint32_t hash;
};
// 新增部分开始
ObjFunction* newFunction();
// 新增部分结束
ObjString* takeString(char* chars, int length);
```

> The implementation is over here:

实现如下：

object.c，在allocateObject()方法后添加代码：

```
ObjFunction* newFunction() {
  ObjFunction* function = ALLOCATE_OBJ(ObjFunction, OBJ_FUNCTION);
  function->arity = 0;
  function->name = NULL;
  initChunk(&function->chunk);
  return function;
}
```

> We use our friend `ALLOCATE_OBJ()` to allocate memory and initialize the object's header so that the VM knows what type of object it is. Instead of passing in arguments to initialize the function like we did with ObjString, we set the function up in a sort of blank state—zero arity, no name, and no code. That will get filled in later after the function is created.

我们使用好朋友`ALLOCATE_OBJ()`来分配内存并初始化对象的头信息，以便虚拟机知道它是什么类型的对象。我们没有像对ObjString那样传入参数来初始化函数，而是将函数设置为一种空白状态——零参数、无名称、无代码。这里会在稍后创建函数后被填入数据。

> Since we have a new kind of object, we need a new object type in the enum.

因为有了一个新类型的对象，我们需要在枚举中添加一个新的对象类型。

object.h，在枚举ObjType中添加代码：

```
typedef enum {
  // 新增部分开始
  OBJ_FUNCTION,
  // 新增部分结束
  OBJ_STRING,
} ObjType;
```

> When we’re done with a function object, we must return the bits it borrowed back to the operating system.

当我们使用完一个函数对象后，必须将它借用的比特位返还给操作系统。

memory.c，在freeObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_FUNCTION: {
      ObjFunction* function = (ObjFunction*)object;
      freeChunk(&function->chunk);
      FREE(ObjFunction, object);
      break;
    }
    // 新增部分结束
    case OBJ_STRING: {
```

> This switch case is responsible for freeing the ObjFunction itself as well as any other memory it owns. Functions own their chunk, so we call Chunk’s destructor-like function.

这个switch语句负责释放ObjFunction本身以及它所占用的其它内存。函数拥有自己的字节码块，所以我们调用Chunk中类似析构器的函数^2。

> Lox lets you print any object, and functions are first-class objects, so we need to handle them too.

Lox允许你打印任何对象，而函数是一等对象，所以我们也需要处理它们。

object.c，在printObject()方法中添加代码：

```
  switch (OBJ_TYPE(value)) {
    // 新增部分开始
    case OBJ_FUNCTION:
      printFunction(AS_FUNCTION(value));
      break;
    // 新增部分结束
    case OBJ_STRING:
```

> This calls out to:

这就引出了：

object.c，在copyString()方法后添加代码：

```
static void printFunction(ObjFunction* function) {
  printf("<fn %s>", function->name->chars);
}
```

> Since a function knows its name, it may as well say it.

既然函数知道它的名称，那就应该说出来。

> Finally, we have a couple of macros for converting values to functions. First, make sure your value actually *is* a function.

最后，我们有几个宏用于将值转换为函数。首先，确保你的值实际上*是*一个函数。

object.h，添加代码：

```
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分开始
#define IS_FUNCTION(value)     isObjType(value, OBJ_FUNCTION)
// 新增部分结束
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
```

> Assuming that evaluates to true, you can then safely cast the Value to an ObjFunction pointer using this:

假设计算结果为真，你就可以使用这个方法将Value安全地转换为一个ObjFunction指针：

object.h，添加代码：

```
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分开始
#define AS_FUNCTION(value)     ((ObjFunction*)AS_OBJ(value))
// 新增部分结束
#define AS_STRING(value)       ((ObjString*)AS_OBJ(value))
```

> With that, our object model knows how to represent functions. I'm feeling warmed up now. You ready for something a little harder?

这样，我们的对象模型就知道如何表示函数了。我现在感觉已经热身了。你准备好来点更难的东西了吗？

24.2 Compiling to Function Objects

24.2 编译为函数对象

> Right now, our compiler assumes it is always compiling to one single chunk. With each function's code living in separate chunks, that gets more complex. When the compiler reaches a function declaration, it needs to emit code into the function's chunk when compiling its body. At the end of the function body, the compiler needs to return to the previous chunk it was working with.

现在，我们的编译器假定它总会编译到单个字节码块中。由于每个函数的代码都位于不同的字节码块，这就变得更加复杂了。当编译器碰到函数声明时，需要在编译函数主体时将代码写入函数自己的字节码块中。在函数主体的结尾，编译器需要返回到它之前正处理的前一个字节码块。

> That's fine for code inside function bodies, but what about code that isn't? The "top level" of a Lox program is also imperative code and we need a chunk to compile that into. We can simplify the compiler and VM by placing that top-level code inside an automatically defined function too. That way, the compiler is always within some kind of function body, and the VM always runs code by invoking a function. It's as if the entire program is wrapped inside an implicit `main()` function.

这对于函数主体内的代码来说很好，但是对于不在其中的代码呢？Lox程序的"顶层"也是命令式代码，而且我们需要一个字节码块来编译它。我们也可以将顶层代码放入一个自动定义的函数中，从而简化编译器和虚拟机的工作。这样一来，编译器总是在某种函数主体内，而虚拟机总是通过调用函数来运行代码。这就像整个程序被包裹在一个隐式的`main()`函数中一样^3。

> Before we get to user-defined functions, then, let's do the reorganization to support that implicit top-level function. It starts with the Compiler struct. Instead of pointing directly to a Chunk that the compiler writes to, it instead has a reference to the function object being built.

在我们讨论用户定义的函数之前，让我们先重新组织一下，支持隐式的顶层函数。这要从Compiler结构体开始。它不再直接指向编译器写入的Chunk，而是指向正在构建的函数对象的引用。

compiler.c，在结构体Compiler中添加代码：

```
typedef struct {
  // 新增部分开始
  ObjFunction* function;
  FunctionType type;
  // 新增部分结束
  Local locals[UINT8_COUNT];
```

> We also have a little FunctionType enum. This lets the compiler tell when it's compiling top-level code versus the body of a function. Most of the compiler doesn't care about this—that's why it's a useful abstraction—but in one or two places the distinction is meaningful. We'll get to one later.

我们也有一个小小的FunctionType枚举。这让编译器可以区分它在编译顶层代码还是函数主体。大多数编译器并不关心这一点——这就是为什么它是一个有用的抽象——但是在一两个地方，这种区分是有意义的。我们稍后会讲到其中一个。

compiler.c，在结构体Local后添加代码：

```
typedef enum {
  TYPE_FUNCTION,
  TYPE_SCRIPT
} FunctionType;
```

> Every place in the compiler that was writing to the Chunk now needs to go through that `function` pointer. Fortunately, many chapters ago, we encapsulated access to the chunk in the `currentChunk()` function. We only need to fix that and the rest of the compiler is happy.

编译器中所有写入Chunk的地方，现在都需要通过`function`指针。幸运的是，在很多章节之前，我们在`currentChunk()`函数中封装了对字节码块的访问。我们只需要修改它，编译器的其它部分就可以了^4。

compiler.c，在变量current后，替换5行：

```
Compiler* current = NULL;
// 替换部分开始
static Chunk* currentChunk() {
  return &current->function->chunk;
}
// 替换部分结束
static void errorAt(Token* token, const char* message) {
```

> The current chunk is always the chunk owned by the function we're in the middle of compiling. Next, we need to actually create that function. Previously, the VM passed a Chunk to the compiler which filled it with code. Instead, the compiler will create and return a function that contains the compiled top-level code—which is all we support right now—of the user's program.

当前的字节码块一定是我们正在编译的函数所拥有的块。接下来，我们需要实际创建该函数。之前，虚拟机将一个Chunk传递给编译器，编译器会将代码填入其中。现在取而代之的是，编译器创建并返回一个包含已编译顶层代码的函数——这就是我们目前所支持的。

> ### 24.2.1 Creating functions at compile time

24.2.1 编译时创建函数

> We start threading this through in `compile()`, which is the main entry point into the compiler.

我们在`compile()`中开始执行此操作，该方法是进入编译器的主要入口点。

compiler.c，在compile()方法中替换1行：

```
  Compiler compiler;
  // 替换部分开始
  initCompiler(&compiler, TYPE_SCRIPT);
  // 替换部分结束
  parser.hadError = false;
```

> There are a bunch of changes in how the compiler is initialized. First, we initialize the new Compiler fields.

在如何初始化编译器方面有很多改变。首先，我们初始化新的Compiler字段。

compiler.c，在函数initCompiler()中替换3行：

```
  // 替换部分开始
static void initCompiler(Compiler* compiler, FunctionType type) {
  compiler->function = NULL;
  compiler->type = type;
  // 替换部分结束
  compiler->localCount = 0;
```

> Then we allocate a new function object to compile into.

然后我们分配一个新的函数对象用于编译。

compiler.c，在initCompiler()方法中添加代码^5：

```
  compiler->scopeDepth = 0;
  // 新增部分开始
  compiler->function = newFunction();
  // 新增部分结束
  current = compiler;
```

> Creating an ObjFunction in the compiler might seem a little strange. A function object is the *runtime* representation of a function, but here we are creating it at compile time. The way to think of it is that a function is similar to a string or number literal. It forms a bridge between the compile time and runtime worlds. When we get to function *declarations*, those really *are* literals—they are a notation that produces values of a built-in type. So the compiler creates function objects during compilation. Then, at runtime, they are simply invoked.

在编译器中创建ObjFunction可能看起来有点奇怪。函数对象是一个函数的运行时表示，但这里我们是在编译时创建它。我们可以这样想：函数类似于一个字符串或数字字面量。它在编译时和运行时之间形成了一座桥梁。当我们碰到函数*声明*时，它们确实*是*字面量——它们是一种生成内置类型值的符号。因此，编译器在编译期间创建函数对象^6。然后，在运行时，它们被简单地调用。

> Here is another strange piece of code:

下面是另一段奇怪的代码：

compiler.c，在initCompiler()方法中添加代码：

```
  current = compiler;
  // 新增部分开始
  Local* local = &current->locals[current->localCount++];
  local->depth = 0;
  local->name.start = "";
  local->name.length = 0;
  // 新增部分结束
}
```

> Remember that the compiler's `locals` array keeps track of which stack slots are associated with which local variables or temporaries. From now on, the compiler implicitly claims stack slot zero for the VM's own internal use. We give it an empty name so that the user can't write an identifier that refers to it. I'll explain what this is about when it becomes useful.

请记住，编译器的`locals`数组记录了哪些栈槽与哪些局部变量或临时变量相关联。从现在开始，编译器隐式地要求栈槽0供虚拟机自己内部使用。我们给它一个空的名称，这样用户就不能向一个指向它的标识符写值。等它起作用时，我会解释这是怎么回事。

> That's the initialization side. We also need a couple of changes on the other end when we finish compiling some code.

这就是初始化这一边的工作。当我们完成一些代码的编译时，还需要在另一边做一些改变。

compiler.c，在函数endCompiler()中替换1行：

```
// 替换部分开始
static ObjFunction* endCompiler() {
// 替换部分结束
  emitReturn();
```

> Previously, when `interpret()` called into the compiler, it passed in a Chunk to be written to. Now that the compiler creates the function object itself, we return that function. We grab it from the current compiler here:

以前，当调用`interpret()`方法进入编译器时，会传入一个要写入的Chunk。现在，编译器自己创建了函数对象，我们返回该函数。我们从当前编译器中这样获取它：

compiler.c，在endCompiler()方法中添加代码：

```
  emitReturn();
  // 新增部分开始
  ObjFunction* function = current->function;
  // 新增部分结束
#ifdef DEBUG_PRINT_CODE
```

> And then return it to `compile()` like so:

然后这样将其返回给`compile()`：

compiler.c，在endCompiler()方法中添加代码：

```
#endif
  // 新增部分开始
  return function;
  // 新增部分结束
}
```

> Now is a good time to make another tweak in this function. Earlier, we added some diagnostic code to have the VM dump the disassembled bytecode so we could debug the compiler. We should fix that to keep working now that the generated chunk is wrapped in a function.

现在是对该函数进行另一个调整的好时机。之前，我们添加了一些诊断性代码，让虚拟机转储反汇编的字节码，以便我们可以调试编译器。现在生成的字节码块包含在一个函数中，我们要修复这些代码，使其继续工作。

compiler.c，在*endCompiler*()方法中替换1行：

```
#ifdef DEBUG_PRINT_CODE
  if (!parser.hadError) {
    // 替换部分开始
    disassembleChunk(currentChunk(), function->name != NULL
        ? function->name->chars : "<script>");
    // 替换部分结束
  }
#endif
```

> Notice the check in here to see if the function's name is NULL? User-defined functions have names, but the implicit function we create for the top-level code does not, and we need to handle that gracefully even in our own diagnostic code. Speaking of which:

注意到这里检查了函数名称是否为NULL吗？用户定义的函数有名称，但我们为顶层代码创建的隐式函数却没有，即使在我们自己的诊断代码中，我们也需要优雅地处理这个问题。说到这一点：

object.c，在printFunction()方法中添加代码：

```
static void printFunction(ObjFunction* function) {
  // 新增部分开始
  if (function->name == NULL) {
    printf("<script>");
    return;
  }
  // 新增部分结束
  printf("<fn %s>", function->name->chars);
```

> There's no way for a *user* to get a reference to the top-level function and try to print it, but our DEBUG_TRACE_EXECUTION diagnostic code that prints the entire stack can and does.

用户没有办法获取对顶层函数的引用并试图打印它，但我们用来打印整个堆栈的诊断代码DEBUG_TRACE_EXECUTION可以而且确实这样做了^7。

> Bumping up a level to compile(), we adjust its signature.

为了给compile()提升一级，我们调整其签名。

compiler.h，在函数compile()中替换1行：

```
#include "vm.h"
// 替换部分开始
ObjFunction* compile(const char* source);
// 替换部分结束
#endif
```

> Instead of taking a chunk, now it returns a function. Over in the implementation:

现在它不再接受字节码块，而是返回一个函数。在实现中：

compiler.c，在函数compile()中替换1行：

```
// 替换部分开始
ObjFunction* compile(const char* source) {
// 替换部分结束
  initScanner(source);
```

> Finally we get to some actual code. We change the very end of the function to this:

最后，我们得到了一些实际的代码。我们把方法的最后部分改成这样： *compiler.c，在compile()方法中替换2行：*

```
  while (!match(TOKEN_EOF)) {
    declaration();
  }
  // 替换部分开始
  ObjFunction* function = endCompiler();
  return parser.hadError ? NULL : function;
  // 替换部分结束
}
```

> We get the function object from the compiler. If there were no compile errors, we return it. Otherwise, we signal an error by returning NULL. This way, the VM doesn't try to execute a function that may contain invalid bytecode.

我们从编译器获取函数对象。如果没有编译错误，就返回它。否则，我们通过返回NULL表示错误。这样，虚拟机就不会试图执行可能包含无效字节码的函数。

> Eventually, we will update `interpret()` to handle the new declaration of `compile()`, but first we have some other changes to make.

最终，我们会更新`interpret()`来处理`compile()`的新声明，但首先我们要做一些其它的改变。

24.3 Call Frames

24.3 调用帧

> It's time for a big conceptual leap. Before we can implement function declarations and calls, we need to get the VM ready to handle them. There are two main problems we need to worry about:

是时候进行一次重大的概念性飞跃了。在我们实现函数声明和调用之前，需要让虚拟机准备好处理它们。我们需要考虑两个主要问题：

24.3.1 Allocating local variables

24.3.1 分配局部变量

> The compiler allocates stack slots for local variables. How should that work when the set of local variables in a program is distributed across multiple functions?

编译器为局部变量分配了堆栈槽。当程序中的局部变量集分布在多个函数中时，应该如何操作？

> One option would be to keep them totally separate. Each function would get its own dedicated set of slots in the VM stack that it would own forever, even when the function isn't being called. Each local variable in the entire program would have a bit of memory in the VM that it keeps to itself.

一种选择是将它们完全分开。每个函数在虚拟机堆栈中都有自己的一组专用槽，即使在函数没有被调用的情况下，它也会永远拥有这些槽。整个程序中的每个局部变量在虚拟机中都有自己保留的一小块内存^8。

> Believe it or not, early programming language implementations worked this way. The first Fortran compilers statically allocated memory for each variable. The obvious problem is that it's really inefficient. Most functions are not in the middle of being called at any point in time, so sitting on unused memory for them is wasteful.

信不信由你，早期的编程语言实现就是这样工作的。第一个Fortran编译器为每个变量静态地分配了内存。最显而易见的问题是效率很低。大多数函数不会随时都在被调用，所以一直占用未使用的内存是浪费的。

> The more fundamental problem, though, is recursion. With recursion, you can be "in" multiple calls to the same function at the same time. Each needs its own memory for its local variables. In jlox, we solved this by dynamically allocating memory for an environment each time a function was called or a block entered. In clox, we don't want that kind of performance cost on every function call.

不过，更根本的问题是递归。通过递归，你可以在同一时刻处于对同一个函数的多次调用"中"。每个函数的局部变量都需要自己的内存。在jlox中，我们通过在每次调用函数或进入代码块时为环境动态分配内存来解决这个问题^9。在clox中，我们不希望在每次调用时都付出这样的性能代价。

> Instead, our solution lies somewhere between Fortran's static allocation and jlox's dynamic approach. The value stack in the VM works on the observation that local variables and temporaries behave in a

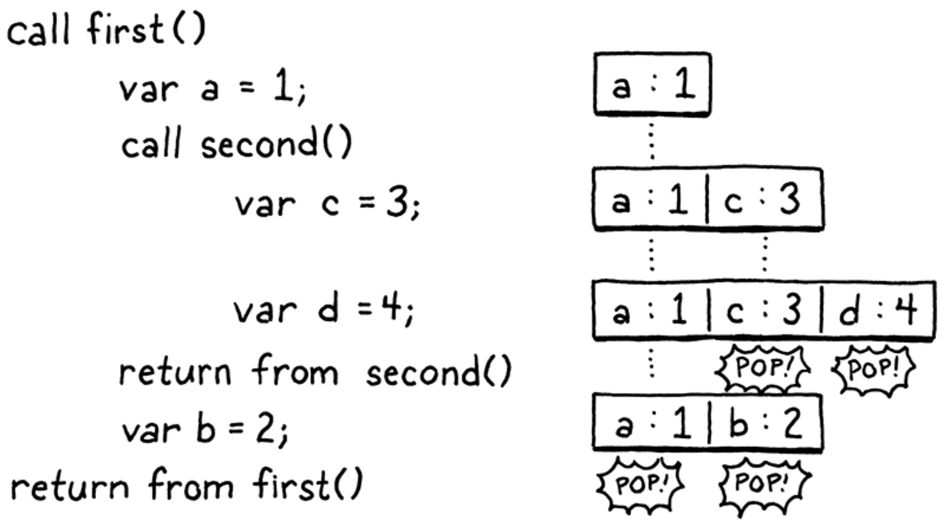
> last-in first-out fashion. Fortunately for us, that's still true even when you add function calls into the mix. Here's an example:

相反，我们的解决方案介于Fortran的静态分配和jlox的动态方法之间。虚拟机中的值栈的工作原理是：局部变量和临时变量的后进先出的行为模式。幸运的是，即使你把函数调用考虑在内，这仍然是正确的。这里有一个例子：

```
fun first() {
  var a = 1;
  second();
  var b = 2;
}

fun second() {
  var c = 3;
  var d = 4;
}

first();
```

> Step through the program and look at which variables are in memory at each point in time:

逐步执行程序，看看在每个时间点上内存中有哪些变量：



> As execution flows through the two calls, every local variable obeys the principle that any variable declared after it will be discarded before the first variable needs to be. This is true even across calls. We know we'll be done with c and d before we are done with a. It seems we should be able to allocate local variables on the VM's value stack.

在这两次调用的执行过程中，每个局部变量都遵循这样的原则：当某个变量需要被丢弃时，在它之后声明的任何变量都会被丢弃。甚至在不同的调用中也是如此。我们知道，在我们用完a之前，已经用完了c和d。看起来我们应该能够在虚拟机的值栈上分配局部变量。

> Ideally, we still determine *where* on the stack each variable will go at compile time. That keeps the bytecode instructions for working with variables simple and fast. In the above example, we could imagine doing so in a straightforward way, but that doesn't always work out. Consider:

理想情况下，我们仍然在编译时确定每个变量在栈中的位置。这使得处理变量的字节码指令变得简单而快速。在上面的例子中，我们可以想象^10以一种直接的方式这样做，但这并不总是可行的。考虑一下：

```
fun first() {
  var a = 1;
  second();
  var b = 2;
  second();
}

fun second() {
  var c = 3;
  var d = 4;
}

first();
```

> In the first call to `second()`, `c` and `d` would go into slots 1 and 2. But in the second call, we need to have made room for `b`, so `c` and `d` need to be in slots 2 and 3. Thus the compiler can’t pin down an exact slot for each local variable across function calls. But *within* a given function, the *relative* locations of each local variable are fixed. Variable `d` is always in the slot right after `c`. This is the key insight.

在对`second()`的第一次调用中，`c`和`d`将进入槽1和2。但在第二次调用中，我们需要为`b`腾出空间，所以`c`和`d`需要放在槽2和3里。因此，编译器不能在不同的函数调用中为每个局部变量指定一个确切的槽。但是在特定的函数中，每个局部变量的相对位置是固定的。变量`d`总是在变量`c`后面的槽里。这是关键的见解。

> When a function is called, we don’t know where the top of the stack will be because it can be called from different contexts. But, wherever that top happens to be, we do know where all of the function’s local variables will be relative to that starting point. So, like many problems, we solve our allocation problem with a level of indirection.

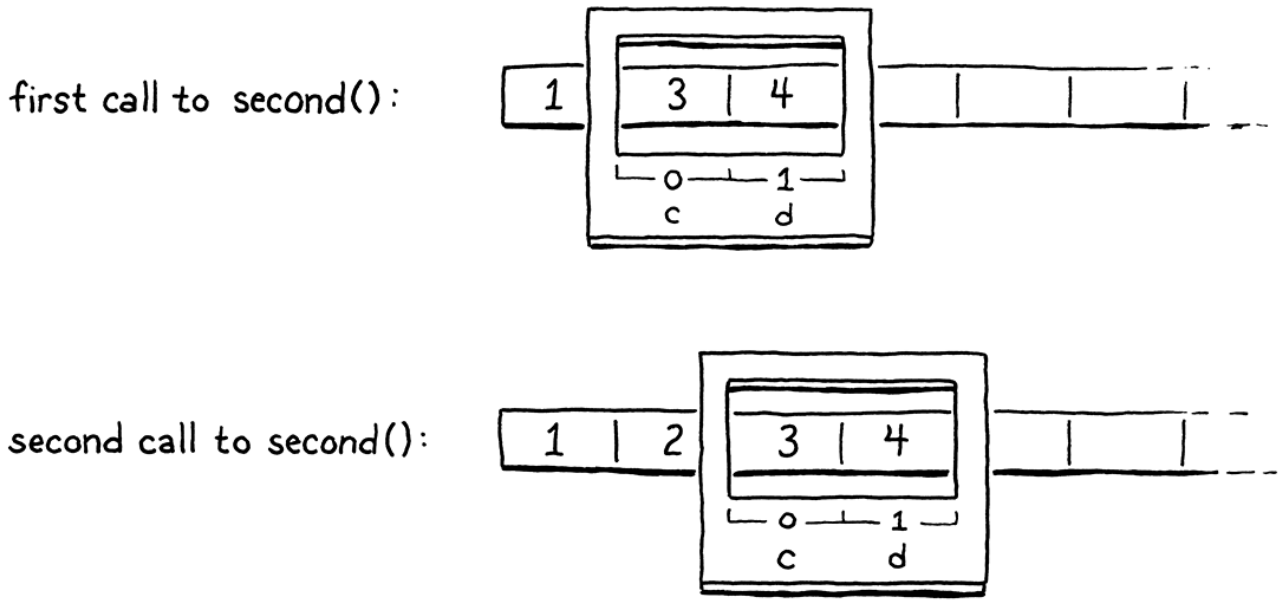
当函数被调用时，我们不知道栈顶在什么位置，因为它可以从不同的上下文中被调用。但是，无论栈顶在哪里，我们都知道该函数的所有局部变量相对于起始点的位置。因此，像很多问题一样，我们使用一个中间层来解决分配问题。

> At the beginning of each function call, the VM records the location of the first slot where that function’s own locals begin. The instructions for working with local variables access them by a slot index relative to that, instead of relative to the bottom of the stack like they do today. At compile time, we calculate those relative slots. At runtime, we convert that relative slot to an absolute stack index by adding the function call’s starting slot.

在每次函数调用开始时，虚拟机都会记录函数自身的局部变量开始的第一个槽的位置。使用局部变量的指令通过相对于该槽的索引来访问它们，而不是像现在这样使用相对于栈底的索引。在编译时，我们可以计算出这些相对槽位。在运行时，加上函数调用时的起始槽位，就能将相对位置转换为栈中的绝对索引。

> It’s as if the function gets a “window” or “frame” within the larger stack where it can store its locals. The position of the **call frame** is determined at runtime, but within and relative to that region, we know where to find things.

这就好像是函数在更大的堆栈中得到了一个“窗口”或“帧”，它可以在其中存储局部变量。**调用帧**的位置是在运行时确定的，但在该区域内部及其相对位置上，我们知道在哪里可以找到目标。



> The historical name for this recorded location where the function's locals start is a **frame pointer** because it points to the beginning of the function's call frame. Sometimes you hear **base pointer**, because it points to the base stack slot on top of which all of the function's variables live.

这个记录了函数局部变量开始的位置的历史名称是**帧指针**，因为它指向函数调用帧的开始处。有时你会听到**基指针**，因为它指向一个基本栈槽，函数的所有变量都在其之上。

> That's the first piece of data we need to track. Every time we call a function, the VM determines the first stack slot where that function's variables begin.

这是我们需要跟踪的第一块数据。每次我们调用函数时，虚拟机都会确定该函数变量开始的第一个栈槽。

> ## 24.3.2 Return addresses

24.3.2 返回地址

> Right now, the VM works its way through the instruction stream by incrementing the `ip` field. The only interesting behavior is around control flow instructions which offset the `ip` by larger amounts. *Calling* a function is pretty straightforward—simply set `ip` to point to the first instruction in that function's chunk. But what about when the function is done?

现在，虚拟机通过递增`ip`字段的方式在指令流中工作。唯一有趣的行为是关于控制流指令的，这些指令会以较大的数值对`ip`进行偏移。调用函数非常直接——将`ip`简单地设置为指向函数块中的第一条指令。但是等函数完成后怎么办？

> The VM needs to return back to the chunk where the function was called from and resume execution at the instruction immediately after the call. Thus, for each function call, we need to track where we jump back to when the call completes. This is called a **return address** because it's the address of the instruction that the VM returns to after the call.

虚拟机需要返回到调用函数的字节码块，并在调用之后立即恢复执行指令。因此，对于每个函数调用，在调用完成后，需要记录调用完成后需要跳回什么地方。这被称为**返回地址**，因为它是虚拟机在调用后返回的指令的

地址。

> Again, thanks to recursion, there may be multiple return addresses for a single function, so this is a property of each *invocation* and not the function itself.

同样，由于递归的存在，一个函数可能会对应多个返回地址，所以这是每个调用的属性，而不是函数本身的属性^11。

> 24.3.3 The call stack

24.3.3 调用栈

> So for each live function invocation—each call that hasn't returned yet—we need to track where on the stack that function's locals begin, and where the caller should resume. We'll put this, along with some other stuff, in a new struct.

因此，对于每个活动的函数执行（每个尚未返回的调用），我们需要跟踪该函数的局部变量在堆栈中的何处开始，以及调用方应该在何处恢复。我们会将这些信息以及其它一些数据放在新的结构体中。

vm.h，添加代码：

```
#define STACK_MAX 256
// 新增部分开始
typedef struct {
  ObjFunction* function;
  uint8_t* ip;
  Value* slots;
} CallFrame;
// 新增部分结束
typedef struct {
```

> A CallFrame represents a single ongoing function call. The `slots` field points into the VM's value stack at the first slot that this function can use. I gave it a plural name because—thanks to C's weird "pointers are sort of arrays" thing—we'll treat it like an array.

一个CallFrame代表一个正在进行的函数调用。`slots`字段指向虚拟机的值栈中该函数可以使用的第一个槽。我给它取了一个复数的名字是因为我们会把它当作一个数组来对待（感谢C语言中"指针是一种数组"这个奇怪的概念）。

> The implementation of return addresses is a little different from what I described above. Instead of storing the return address in the callee's frame, the caller stores its own `ip`. When we return from a function, the VM will jump to the `ip` of the caller's CallFrame and resume from there.

返回地址的实现与我上面的描述有所不同。调用者不是将返回地址存储在被调用者的帧中，而是将自己的`ip`存储起来。等到从函数中返回时，虚拟机会跳转到调用方的CallFrame的`ip`，并从那里继续执行。

> I also stuffed a pointer to the function being called in here. We'll use that to look up constants and for a few other things.

我还在这里塞了一个指向被调用函数的指针。我们会用它来查询常量和其它一些事情。

> Each time a function is called, we create one of these structs. We could dynamically allocate them on the heap, but that's slow. Function calls are a core operation, so they need to be as fast as possible. Fortunately, we can make the same observation we made for variables: function calls have stack semantics. If `first()` calls `second()`, the call to `second()` will complete before `first()` does.

每次函数被调用时，我们会创建一个这样的结构体。我们可以在堆上动态地分配它们，但那样会很慢。函数调用是核心操作，所以它们需要尽可能快。幸运的是，我们意识到它和变量很相似：函数调用具有堆栈语义。如果`first()`调用`second()`，对`second()`的调用将在`first()`之前完成^12。

> So over in the VM, we create an array of these CallFrame structs up front and treat it as a stack, like we do with the value array.

因此在虚拟机中，我们预先创建一个CallFrame结构体的数组，并将其作为堆栈对待，就像我们对值数组所做的那样。

vm.h，在结构体VM中替换2行：

```
typedef struct {
  // 替换部分开始
  CallFrame frames[FRAMES_MAX];
  int frameCount;
  // 替换部分结束
  Value stack[STACK_MAX];
```

> This array replaces the `chunk` and `ip` fields we used to have directly in the VM. Now each CallFrame has its own `ip` and its own pointer to the ObjFunction that it's executing. From there, we can get to the function's chunk.

这个数组取代了我们过去在VM中直接使用的`chunk`和`ip`字段。现在，每个CallFrame都有自己的`ip`和指向它正在执行的ObjFunction的指针。通过它们，我们可以得到函数的字节码块。

> The new `frameCount` field in the VM stores the current height of the CallFrame stack—the number of ongoing function calls. To keep clox simple, the array's capacity is fixed. This means, as in many language implementations, there is a maximum call depth we can handle. For clox, it's defined here:

VM中新的`frameCount`字段存储了CallFrame栈的当前高度——正在进行的函数调用的数量。为了使clox简单，数组的容量是固定的。这意味着，和许多语言的实现一样，存在一个我们可以处理的最大调用深度。对于clox，在这里定义它：

vm.h，替换1行：

```
#include "value.h"
// 替换部分开始
#define FRAMES_MAX 64
#define STACK_MAX (FRAMES_MAX * UINT8_COUNT)
// 替换部分结束
typedef struct {
```

> We also redefine the value stack's size in terms of that to make sure we have plenty of stack slots even in very deep call trees. When the VM starts up, the CallFrame stack is empty.

我们还以此重新定义了值栈的大小，以确保即使在很深的调用树中我们也有足够的栈槽^13。当虚拟机启动时，CallFrame栈是空的。

vm.c，在resetStack()方法中添加代码：

```
  vm.stackTop = vm.stack;
  // 新增部分开始
  vm.frameCount = 0;
  // 新增部分结束
}
```

> The "vm.h" header needs access to ObjFunction, so we add an include.

"vm.h"头文件需要访问ObjFunction，所以我们加一个引入。

vm.h，替换1行：

```
#define clox_vm_h
// 替换部分开始
#include "object.h"
// 替换部分结束
#include "table.h"
```

> Now we're ready to move over to the VM's implementation file. We've got some grunt work ahead of us. We've moved `ip` out of the VM struct and into CallFrame. We need to fix every line of code in the VM that touches `ip` to handle that. Also, the instructions that access local variables by stack slot need to be updated to do so relative to the current CallFrame's `slots` field.

现在我们准备转移到VM的实现文件中。我们还有很多艰巨的工作要做。我们已经将`ip`从VM结构体移到了CallFrame中。我们需要修改VM中使用了`ip`的每一行代码来解决这个问题。此外，需要更新根据栈槽访问局部变量的指令，使其相对于当前CallFrame的`slots`字段进行访问。

> We'll start at the top and plow through it.

我们从最上面开始，彻底解决这个问题。

vm.c，在run()方法中替换4行：

```
static InterpretResult run() {
  // 替换部分开始
  CallFrame* frame = &vm.frames[vm.frameCount - 1];

#define READ_BYTE() (*frame->ip++)

#define READ_SHORT() \
    (frame->ip += 2, \
    (uint16_t)((frame->ip[-2] << 8) | frame->ip[-1]))

#define READ_CONSTANT() \
    (frame->function->chunk.constants.values[READ_BYTE()])
// 替换部分结束
#define READ_STRING() AS_STRING(READ_CONSTANT())
```

> First, we store the current topmost CallFrame in a local variable inside the main bytecode execution function. Then we replace the bytecode access macros with versions that access `ip` through that variable.

首先，我们将当前最顶部的CallFrame存储在主字节码执行函数中的一个局部变量中。然后我们将字节码访问宏替换为通过该变量访问`ip`的版本^14。

> Now onto each instruction that needs a little tender loving care.

现在我们来看看每条需要温柔呵护的指令。

vm.c，在run()方法中替换1行：

```
      case OP_GET_LOCAL: {
        uint8_t slot = READ_BYTE();
        // 替换部分开始
        push(frame->slots[slot]);
        // 替换部分结束
        break;
```

> Previously, `OP_GET_LOCAL` read the given local slot directly from the VM's stack array, which meant it indexed the slot starting from the bottom of the stack. Now, it accesses the current frame's `slots` array, which means it accesses the given numbered slot relative to the beginning of that frame.

以前，`OP_GET_LOCAL`直接从虚拟机的栈数组中读取给定的局部变量槽，这意味着它是从栈底开始对槽进行索引。现在，它访问的是当前帧的`slots`数组，这意味着它是访问相对于该帧起始位置的给定编号的槽。

> Setting a local variable works the same way.

设置局部变量的方法也是如此。

vm.c，在run()方法中替换1行：

```
      case OP_SET_LOCAL: {
        uint8_t slot = READ_BYTE();
        // 替换部分开始
        frame->slots[slot] = peek(0);
        // 替换部分结束
        break;
```

> The jump instructions used to modify the VM’s `ip` field. Now, they do the same for the current frame’s `ip`.

跳转指令之前是修改VM的`ip`字段。现在，它会对当前帧的`ip`做相同的操作。

vm.c，在run()方法中替换1行：

```
      case OP_JUMP: {
        uint16_t offset = READ_SHORT();
        // 替换部分开始
        frame->ip += offset;
        // 替换部分结束
        break;
```

> Same with the conditional jump:

条件跳转也是如此：

vm.c，在run()方法中替换1行：

```
      case OP_JUMP_IF_FALSE: {
        uint16_t offset = READ_SHORT();
        // 替换部分开始
        if (isFalsey(peek(0))) frame->ip += offset;
        // 替换部分结束
        break;
```

> And our backward-jumping loop instruction:

还有向后跳转的循环指令：

vm.c，在run()方法中替换1行：

```
      case OP_LOOP: {
        uint16_t offset = READ_SHORT();
        // 替换部分开始
        frame->ip -= offset;
        // 替换部分结束
        break;
```

> We have some diagnostic code that prints each instruction as it executes to help us debug our VM. That needs to work with the new structure too.

我们还有一些诊断代码，可以在每条指令执行时将其打印出来，帮助我们调试虚拟机。这也需要能处理新的结构体。

vm.c，在run()方法中替换2行：

```
    printf("\n");
    // 替换部分开始
    disassembleInstruction(&frame->function->chunk,
        (int)(frame->ip - frame->function->chunk.code));
    // 替换部分结束
#endif
```

> Instead of passing in the VM’s `chunk` and `ip` fields, now we read from the current CallFrame.

现在我们从当前的CallFrame中读取数据，而不是传入VM的`chunk` 和`ip` 字段。

> You know, that wasn’t too bad, actually. Most instructions just use the macros so didn’t need to be touched. Next, we jump up a level to the code that calls `run()`.

其实，这不算太糟。大多数指令只是使用了宏，所以不需要修改。接下来，我们向上跳到调用`run()`的代码。

vm.c，在interpret() 方法中替换10行：

```
InterpretResult interpret(const char* source) {
  // 替换部分开始
  ObjFunction* function = compile(source);
  if (function == NULL) return INTERPRET_COMPILE_ERROR;

  push(OBJ_VAL(function));
  CallFrame* frame = &vm.frames[vm.frameCount++];
  frame->function = function;
  frame->ip = function->chunk.code;
  frame->slots = vm.stack;
  // 替换部分结束
  InterpretResult result = run();
```

> We finally get to wire up our earlier compiler changes to the back-end changes we just made. First, we pass the source code to the compiler. It returns us a new ObjFunction containing the compiled top-level code. If we get `NULL` back, it means there was some compile-time error which the compiler has already reported. In that case, we bail out since we can’t run anything.

我们终于可以将之前的编译器修改与我们刚刚做的后端更改联系起来。首先，我们将源代码传递给编译器。它返回给我们一个新的ObjFunction，其中包含编译好的顶层代码。如果我们得到的是`NULL`，这意味着存在一些编译时错误，编译器已经报告过了。在这种情况下，我们就退出，因为我们没有可以运行的代码。

> Otherwise, we store the function on the stack and prepare an initial CallFrame to execute its code. Now you can see why the compiler sets aside stack slot zero—that stores the function being called. In the new CallFrame, we point to the function, initialize its `ip` to point to the beginning of the function’s bytecode, and set up its stack window to start at the very bottom of the VM’s value stack.

否则，我们将函数存储在堆栈中，并准备一个初始CallFrame来执行其代码。现在你可以看到为什么编译器将栈槽0留出来——其中存储着正在被调用的函数。在新的CallFrame中，我们指向该函数，将`ip`初始化为函数字节码的起始位置，并将堆栈窗口设置为从VM值栈的最底部开始。

> This gets the interpreter ready to start executing code. After finishing, the VM used to free the hardcoded chunk. Now that the ObjFunction owns that code, we don't need to do that anymore, so the end of `interpret()` is simply this:

这样解释器就准备好开始执行代码了。完成后，虚拟机原本会释放硬编码的字节码块。现在ObjFunction持有那段代码，我们就不需要再这样做了，所以`interpret()`的结尾是这样的：

vm.c，在interpret()方法中替换4行：

```
  frame->slots = vm.stack;
  // 替换部分开始
  return run();
  // 替换部分结束
}
```

> The last piece of code referring to the old VM fields is `runtimeError()`. We'll revisit that later in the chapter, but for now let's change it to this:

最后一段引用旧的VM字段的代码是`runtimeError()`。我们会在本章后面重新讨论这个问题，但现在我们先将它改成这样：

vm.c，在runtimeError()方法中替换2行：

```
  fputs("\n", stderr);
  // 替换部分开始
  CallFrame* frame = &vm.frames[vm.frameCount - 1];
  size_t instruction = frame->ip - frame->function->chunk.code - 1;
  int line = frame->function->chunk.lines[instruction];
  // 替换部分结束
  fprintf(stderr, "[line %d] in script\n", line);
```

> Instead of reading the chunk and `ip` directly from the VM, it pulls those from the topmost CallFrame on the stack. That should get the function working again and behaving as it did before.

它不是直接从VM中读取字节码块和`ip`，而是从栈顶的CallFrame中获取这些信息。这应该能让函数重新工作，并且表现像以前一样。

> Assuming we did all of that correctly, we got clox back to a runnable state. Fire it up and it does . . . exactly what it did before. We haven't added any new features yet, so this is kind of a let down. But all of the infrastructure is there and ready for us now. Let's take advantage of it.

假如我们都正确执行了所有这些操作，就可以让clox回到可运行的状态。启动它，它就会......像以前一样。我们还没有添加任何新功能，所以这有点让人失望。但是所有的基础设施都已经就绪了。让我们好好利用它。

24.4 Function Declarations

24.4 函数声明

> Before we can do call expressions, we need something to call, so we'll do function declarations first. The fun starts with a keyword.

在我们确实可以调用表达式之前，首先需要一些可以用来调用的东西，所以我们首先要处理函数声明。一切从关键字开始。【译者注：作者这里使用了一个小小的双关，实在不好翻译】

compiler.c，在declaration()方法中替换1行：

```
static void declaration() {
  // 替换部分开始
  if (match(TOKEN_FUN)) {
    funDeclaration();
  } else if (match(TOKEN_VAR)) {
  // 替换部分结束
    varDeclaration();
```

> That passes control to here:

它将控制权传递到这里：

compiler.c，在block()方法后添加：

```
static void funDeclaration() {
  uint8_t global = parseVariable("Expect function name.");
  markInitialized();
  function(TYPE_FUNCTION);
  defineVariable(global);
}
```

> Functions are first-class values, and a function declaration simply creates and stores one in a newly declared variable. So we parse the name just like any other variable declaration. A function declaration at the top level will bind the function to a global variable. Inside a block or other function, a function declaration creates a local variable.

函数是一等公民，函数声明只是在新声明的变量中创建并存储一个函数。因此，我们像其它变量声明一样解析名称。顶层的函数声明会将函数绑定到一个全局变量。在代码块或其它函数内部，函数声明会创建一个局部变量。

> In an earlier chapter, I explained how variables get defined in two stages. This ensures you can't access a variable's value inside the variable's own initializer. That would be bad because the variable doesn't *have* a value yet.

在前面的章节中，我解释了变量是如何分两个阶段定义的。这确保了你不能在变量自己的初始化器中访问该变量的值。这很糟糕，因为变量还*没有*值。

> Functions don't suffer from this problem. It's safe for a function to refer to its own name inside its body. You can't *call* the function and execute the body until after it's fully defined, so you'll never see

> the variable in an uninitialized state. Practically speaking, it's useful to allow this in order to support recursive local functions.

函数不会遇到这个问题。函数在其主体内引用自己的名称是安全的。在函数被完全定义之后，你才能调用函数并执行函数体，所以你永远不会看到处于未初始化状态的变量。实际上，为了支持递归局部函数，允许这样做是很有用的。

> To make that work, we mark the function declaration's variable "initialized" as soon as we compile the name, before we compile the body. That way the name can be referenced inside the body without generating an error.

为此，在我们编译函数名称时（编译函数主体之前），就将函数声明的变量标记为"已初始化"。这样就可以在主体中引用该名称，而不会产生错误。

> We do need one check, though.

不过，我们确实需要做一个检查。

compiler.c，在markInitialized()方法中添加代码：

```
static void markInitialized() {
  // 新增部分开始
  if (current->scopeDepth == 0) return;
  // 新增部分结束
  current->locals[current->localCount - 1].depth =
```

> Before, we called `markInitialized()` only when we already knew we were in a local scope. Now, a top-level function declaration will also call this function. When that happens, there is no local variable to mark initialized—the function is bound to a global variable.

以前，只有在已经知道当前处于局部作用域中时，我们才会调用`markInitialized()`。现在，顶层的函数声明也会调用这个函数。当这种情况发生时，没有局部变量需要标记为已初始化——函数被绑定到了一个全局变量。

> Next, we compile the function itself—its parameter list and block body. For that, we use a separate helper function. That helper generates code that leaves the resulting function object on top of the stack. After that, we call `defineVariable()` to store that function back into the variable we declared for it.

接下来，我们编译函数本身——它的参数列表和代码块主体。为此，我们使用一个单独的辅助函数。该函数生成的代码会将生成的函数对象留在栈顶。之后，我们调用`defineVariable()`，将该函数存储到我们为其声明的变量中。

> I split out the code to compile the parameters and body because we'll reuse it later for parsing method declarations inside classes. Let's build it incrementally, starting with this:

我将编译参数和主体的代码分开，因为我们稍后会重用它来解析类中的方法声明。我们来逐步构建它，从这里开始：

compiler.c，在block()方法后添加代码^15：

```
static void function(FunctionType type) {
  Compiler compiler;
  initCompiler(&compiler, type);
  beginScope();

  consume(TOKEN_LEFT_PAREN, "Expect '(' after function name.");
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after parameters.");
  consume(TOKEN_LEFT_BRACE, "Expect '{' before function body.");
  block();

  ObjFunction* function = endCompiler();
  emitBytes(OP_CONSTANT, makeConstant(OBJ_VAL(function)));
}
```

> For now, we won’t worry about parameters. We parse an empty pair of parentheses followed by the body. The body starts with a left curly brace, which we parse here. Then we call our existing `block()` function, which knows how to compile the rest of a block including the closing brace.

现在，我们不需要考虑参数。我们解析一对空括号，然后是主体。主体以左大括号开始，我们在这里会解析它。然后我们调用现有的`block()`函数，该函数知道如何编译代码块的其余部分，包括结尾的右大括号。

> ### 24.4.1 A stack of compilers

24.4.1 编译器栈

> The interesting parts are the compiler stuff at the top and bottom. The Compiler struct stores data like which slots are owned by which local variables, how many blocks of nesting we’re currently in, etc. All of that is specific to a single function. But now the front end needs to handle compiling multiple functions nested within each other.

有趣的部分是顶部和底部的编译器。Compiler结构体存储的数据包括哪些栈槽被哪些局部变量拥有，目前处于多少层的嵌套块中，等等。所有这些都是针对单个函数的。但是现在，前端需要处理编译相互嵌套的多个函数的编译^16。

> The trick for managing that is to create a separate Compiler for each function being compiled. When we start compiling a function declaration, we create a new Compiler on the C stack and initialize it. `initCompiler()` sets that Compiler to be the current one. Then, as we compile the body, all of the functions that emit bytecode write to the chunk owned by the new Compiler’s function.

管理这个问题的诀窍是为每个正在编译的函数创建一个单独的Compiler。当我们开始编译函数声明时，会在C语言栈中创建一个新的Compiler并初始化它。`initCompiler()`将该Compiler设置为当前编译器。然后，在编译主体时，所有产生字节码的函数都写入新Compiler的函数所持有的字节码块。

> After we reach the end of the function’s block body, we call `endCompiler()`. That yields the newly compiled function object, which we store as a constant in the *surrounding* function’s constant table. But, wait, how do we get back to the surrounding function? We lost it when `initCompiler()` overwrote the current compiler pointer.

在我们到达函数主体块的末尾时，会调用`endCompiler()`。这就得到了新编译的函数对象，我们将其作为常量存储在*外围*函数的常量表中。但是，等等。我们怎样才能回到外围的函数中呢？在`initCompiler()`覆盖当前编译器指针时，我们把它丢了。

> We fix that by treating the series of nested Compiler structs as a stack. Unlike the Value and CallFrame stacks in the VM, we won't use an array. Instead, we use a linked list. Each Compiler points back to the Compiler for the function that encloses it, all the way back to the root Compiler for the top-level code.

我们通过将一系列嵌套的Compiler结构体视为一个栈来解决这个问题。与VM中的Value和CallFrame栈不同，我们不会使用数组。相反，我们使用链表。每个Compiler都指向包含它的函数的Compiler，一直到顶层代码的根Compiler。

compiler.c，在枚举FunctionType后替换1行：

```
} FunctionType;
// 替换部分开始
typedef struct Compiler {
  struct Compiler* enclosing;
  // 替换部分结束
  ObjFunction* function;
```

> Inside the Compiler struct, we can't reference the Compiler *typedef* since that declaration hasn't finished yet. Instead, we give a name to the struct itself and use that for the field's type. C is weird.

在Compiler结构体内部，我们不能引用Compiler*类型定义*，因为声明还没有结束。相反，我们要为结构体本身提供一个名称，并将其用作字段的类型。C语言真奇怪。

> When initializing a new Compiler, we capture the about-to-no-longer-be-current one in that pointer.

在初始化一个新的Compiler时，我们捕获即将更换的当前编译器。

compiler.c，在initCompiler()方法中添加代码：

```
static void initCompiler(Compiler* compiler, FunctionType type) {
  // 新增部分开始
  compiler->enclosing = current;
  // 新增部分结束
  compiler->function = NULL;
```

> Then when a Compiler finishes, it pops itself off the stack by restoring the previous compiler to be the new current one.

然后，当编译器完成时，将之前的编译器恢复为新的当前编译器，从而将自己从栈中弹出。

compiler.c，在endCompiler()方法中添加代码：

```
#endif
  // 新增部分开始
  current = current->enclosing;
  // 新增部分结束
  return function;
```

> Note that we don't even need to dynamically allocate the Compiler structs. Each is stored as a local variable in the C stack—either in `compile()` or `function()`. The linked list of Compilers threads through the C stack. The reason we can get an unbounded number of them is because our compiler uses recursive descent, so `function()` ends up calling itself recursively when you have nested function declarations.

请注意，我们甚至不需要动态地分配Compiler结构体。每个结构体都作为局部变量存储在C语言栈中——不是`compile()`就是`function()`。编译器链表在C语言栈中存在。我们之所以能得到无限多的编译器^17，是因为我们的编译器使用了递归下降，所以当有嵌套的函数声明时，`function()`最终会递归地调用自己。

> ## 24.4.2 Function parameters

24.4.2 函数参数

> Functions aren't very useful if you can't pass arguments to them, so let's do parameters next.

如果你不能向函数传递参数，那函数就不是很有用，所以接下来我们实现参数。

compiler.c，在function()方法中添加代码：

```
  consume(TOKEN_LEFT_PAREN, "Expect '(' after function name.");
  // 新增部分开始
  if (!check(TOKEN_RIGHT_PAREN)) {
    do {
      current->function->arity++;
      if (current->function->arity > 255) {
        errorAtCurrent("Can't have more than 255 parameters.");
      }
      uint8_t constant = parseVariable("Expect parameter name.");
      defineVariable(constant);
    } while (match(TOKEN_COMMA));
  }
  // 新增部分结束
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after parameters.");
```

> Semantically, a parameter is simply a local variable declared in the outermost lexical scope of the function body. We get to use the existing compiler support for declaring named local variables to parse and compile parameters. Unlike local variables, which have initializers, there's no code here to initialize the parameter's value. We'll see how they are initialized later when we do argument passing in function calls.

语义上讲，形参就是在函数体最外层的词法作用域中声明的一个局部变量。我们可以使用现有的编译器对声明命名局部变量的支持来解析和编译形参。与有初始化器的局部变量不同，这里没有代码来初始化形参的值。稍后在函数调用中传递参数时，我们会看到它们是如何初始化的。

> While we're at it, we note the function's arity by counting how many parameters we parse. The other piece of metadata we store with a function is its name. When compiling a function declaration, we call `initCompiler()` right after we parse the function's name. That means we can grab the name right then from the previous token.

在此过程中，我们通过计算所解析的参数数量来确定函数的元数。函数中存储的另一个元数据是它的名称。在编译函数声明时，我们在解析完函数名称之后，会立即调用`initCompiler()`。这意味着我们可以立即从上一个标识中获取名称。

compiler.c，在initCompiler()方法中添加代码：

```
  current = compiler;
  // 新增部分开始
  if (type != TYPE_SCRIPT) {current->function->name =
copyString(parser.previous.start, parser.previous.length);
  }
  // 新增部分结束
  Local* local = &current->locals[current->localCount++];
```

> Note that we're careful to create a copy of the name string. Remember, the lexeme points directly into the original source code string. That string may get freed once the code is finished compiling. The function object we create in the compiler outlives the compiler and persists until runtime. So it needs its own heap-allocated name string that it can keep around.

请注意，我们谨慎地创建了名称字符串的副本。请记住，词素直接指向了源代码字符串。一旦代码编译完成，该字符串就可能被释放。我们在编译器中创建的函数对象比编译器的寿命更长，并持续到运行时。所以它需要自己的堆分配的名称字符串，以便随时可用。

> Rad. Now we can compile function declarations, like this:

太棒了。现在我们可以编译函数声明了，像这样：

```
fun areWeHavingItYet() {
  print "Yes we are!";
}

print areWeHavingItYet;
```

> We just can't do anything useful with them.

只是我们还不能用它们来做任何有用的事情。

24.5 Function Calls

24.5 函数调用

> By the end of this section, we'll start to see some interesting behavior. The next step is calling functions. We don't usually think of it this way, but a function call expression is kind of an infix `(`

> operator. You have a high-precedence expression on the left for the thing being called—usually just a single identifier. Then the (in the middle, followed by the argument expressions separated by commas, and a final) to wrap it up at the end.

在本小节结束时，我们将开始看到一些有趣的行为。下一步是调用函数。我们通常不会这样想，但是函数调用表达式有点像是一个中缀(操作符。在左边有一个高优先级的表达式，表示被调用的内容——通常只是一个标识符。然后是中间的(，后跟由逗号分隔的参数表达式，最后是一个)把它包起来。

> That odd grammatical perspective explains how to hook the syntax into our parsing table.

这个奇怪的语法视角解释了如何将语法挂接到我们的解析表格中。

compiler.c，在unary()方法后添加，替换1行：

```
ParseRule rules[] = {
  // 替换部分开始
  [TOKEN_LEFT_PAREN]    = {grouping, call,   PREC_CALL},
  // 替换部分结束
  [TOKEN_RIGHT_PAREN]   = {NULL,     NULL,   PREC_NONE},
```

> When the parser encounters a left parenthesis following an expression, it dispatches to a new parser function.

当解析器遇到表达式后面的左括号时，会将其分派到一个新的解析器函数。

compiler.c，在binary()方法后添加代码：

```
static void call(bool canAssign) {
  uint8_t argCount = argumentList();
  emitBytes(OP_CALL, argCount);
}
```

> We've already consumed the (token, so next we compile the arguments using a separate `argumentList()` helper. That function returns the number of arguments it compiled. Each argument expression generates code that leaves its value on the stack in preparation for the call. After that, we emit a new `OP_CALL` instruction to invoke the function, using the argument count as an operand.

我们已经消费了(标识，所以接下来我们用一个单独的`argumentList()`辅助函数来编译参数。该函数会返回它所编译的参数的数量。每个参数表达式都会生成代码，将其值留在栈中，为调用做准备。之后，我们发出一条新的`OP_CALL`指令来调用该函数，将参数数量作为操作数。

> We compile the arguments using this friend:

我们使用这个助手来编译参数：

compiler.c，在defineVariable()方法后添加代码：

```
static uint8_t argumentList() {
  uint8_t argCount = 0;
  if (!check(TOKEN_RIGHT_PAREN)) {
    do {
      expression();
      argCount++;
    } while (match(TOKEN_COMMA));
  }
  consume(TOKEN_RIGHT_PAREN, "Expect ')' after arguments.");
  return argCount;
}
```

> That code should look familiar from jlox. We chew through arguments as long as we find commas after each expression. Once we run out, we consume the final closing parenthesis and we're done.

这段代码看起来跟jlox很相似。只要我们在每个表达式后面找到逗号，就会仔细分析函数。一旦运行完成，消耗最后的右括号，我们就完成了。

> Well, almost. Back in jlox, we added a compile-time check that you don't pass more than 255 arguments to a call. At the time, I said that was because clox would need a similar limit. Now you can see why—since we stuff the argument count into the bytecode as a single-byte operand, we can only go up to 255. We need to verify that in this compiler too.

嗯，大概就这样。在jlox中，我们添加了一个编译时检查，即一次调用传递的参数不超过255个。当时，我说这是因为clox需要类似的限制。现在你可以明白为什么了——因为我们把参数数量作为单字节操作数填充到字节码中，所以最多只能达到255。我们也需要在这个编译器中验证。

compiler.c，在argumentList()方法中添加代码：

```
      expression();
      // 新增部分开始
      if (argCount == 255) {
        error("Can't have more than 255 arguments.");
      }
      // 新增部分结束
      argCount++;
```

> That's the front end. Let's skip over to the back end, with a quick stop in the middle to declare the new instruction.

这就是前端。让我们跳到后端继续，不过要在中间快速暂停一下，声明一个新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_LOOP,
  // 新增部分开始
  OP_CALL,
  // 新增部分结束
  OP_RETURN,
```

> 24.5.1 Binding arguments to parameters

24.5.1 绑定形参与实参

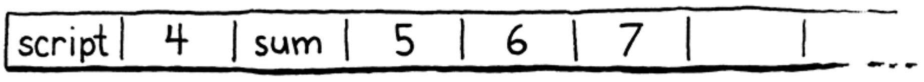
> Before we get to the implementation, we should think about what the stack looks like at the point of a call and what we need to do from there. When we reach the call instruction, we have already executed the expression for the function being called, followed by its arguments. Say our program looks like this:

在我们开始实现之前，应该考虑一下堆栈在调用时是什么样子的，以及我们需要从中做什么。当我们到达调用指令时，我们已经执行了被调用函数的表达式，后面是其参数。假设我们的程序是这样的：

```
fun sum(a, b, c) {
  return a + b + c;
}

print 4 + sum(5, 6, 7);
```

> If we pause the VM right on the `OP_CALL` instruction for that call to `sum()`, the stack looks like this:

如果我们在调用`sum()`的`OP_CALL`指令处暂停虚拟机，栈看起来是这样的：

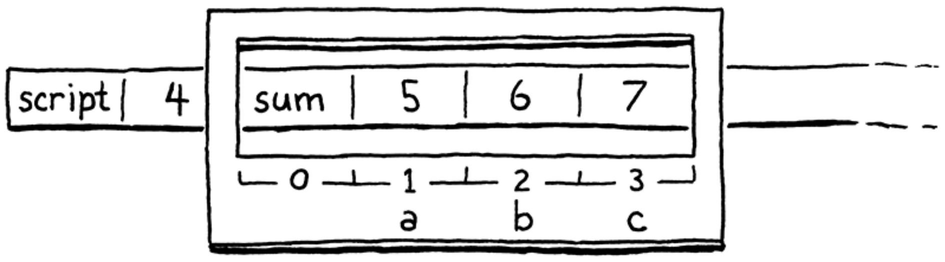


> Picture this from the perspective of `sum()` itself. When the compiler compiled `sum()`, it automatically allocated slot zero. Then, after that, it allocated local slots for the parameters `a`, `b`, and `c`, in order. To perform a call to `sum()`, we need a CallFrame initialized with the function being called and a region of stack slots that it can use. Then we need to collect the arguments passed to the function and get them into the corresponding slots for the parameters.

从`sum()`本身的角度来考虑这个问题。当编译器编译`sum()`时，它自动分配了槽位0。然后，它在该位置后为参数`a`、`b`、`c`依次分配了局部槽。为了执行对`sum()`的调用，我们需要一个通过被调用函数和可用栈槽区域初始化的CallFrame。然后我们需要收集传递给函数的参数，并将它们放入参数对应的槽中。

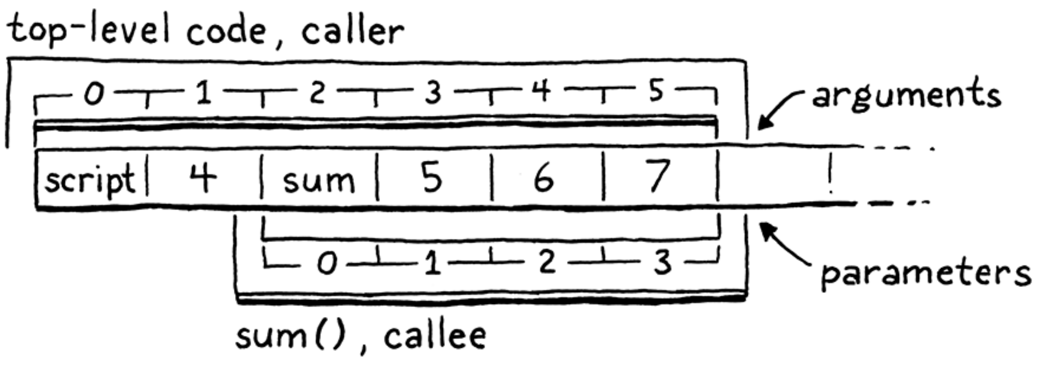
> When the VM starts executing the body of `sum()`, we want its stack window to look like this:

当VM开始执行`sum()`函数体时，我们需要栈窗口看起来像这样：



> Do you notice how the argument slots that the caller sets up and the parameter slots the callee needs are both in exactly the right order? How convenient! This is no coincidence. When I talked about each CallFrame having its own window into the stack, I never said those windows must be *disjoint*. There's nothing preventing us from overlapping them, like this:

你是否注意到，调用者设置的实参槽和被调用者需要的形参槽的顺序是完全匹配的？多么方便啊！这并非巧合。当我谈到每个CallFrame在栈中都有自己的窗口时，从未说过这些窗口一定是不相交的。没有什么能阻止我们将它们重叠起来，就像这样：



> The top of the caller's stack contains the function being called followed by the arguments in order. We know the caller doesn't have any other slots above those in use because any temporaries needed when evaluating argument expressions have been discarded by now. The bottom of the callee's stack overlaps so that the parameter slots exactly line up with where the argument values already live.

调用者栈的顶部包括被调用的函数，后面依次是参数。我们知道调用者在这些正在使用的槽位之上没有占用其它槽，因为在计算参数表达式时需要的所有临时变量都已经被丢弃了。被调用者栈的底部是重叠的，这样形参的槽位与已有的实参值的位置就完全一致^18。

> This means that we don't need to do *any* work to "bind an argument to a parameter". There's no copying values between slots or across environments. The arguments are already exactly where they need to be. It's hard to beat that for performance.

这意味着我们不需要做*任何*工作来"将形参绑定到实参"。不用在槽之间或跨环境复制值。这些实参已经在它们需要在的位置了。很难有比这更好的性能了。

> Time to implement the call instruction.

是时候来实现调用指令了。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_CALL: {
        int argCount = READ_BYTE();
        if (!callValue(peek(argCount), argCount)) {
          return INTERPRET_RUNTIME_ERROR;
        }
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> We need to know the function being called and the number of arguments passed to it. We get the latter from the instruction's operand. That also tells us where to find the function on the stack by counting past the argument slots from the top of the stack. We hand that data off to a separate `callValue()` function. If that returns `false`, it means the call caused some sort of runtime error. When that happens, we abort the interpreter.

我们需要知道被调用的函数以及传递给它的参数数量。我们从指令的操作数中得到后者。它还告诉我们，从栈顶向下跳过参数数量的槽位，就可以在栈中找到该函数。我们将这些数据传给一个单独的`callValue()`函数。如果函数返回`false`，意味着该调用引发了某种运行时错误。当这种情况发生时，我们中止解释器。

> If `callValue()` is successful, there will be a new frame on the CallFrame stack for the called function. The `run()` function has its own cached pointer to the current frame, so we need to update that.

如果`callValue()`成功，将会在CallFrame栈中为被调用函数创建一个新帧。`run()`函数有它自己缓存的指向当前帧的指针，所以我们需要更新它。

vm.c，在run()方法中添加代码：

```
          return INTERPRET_RUNTIME_ERROR;
        }
        // 新增部分开始
        frame = &vm.frames[vm.frameCount - 1];
        // 新增部分结束
        break;
```

> Since the bytecode dispatch loop reads from that `frame` variable, when the VM goes to execute the next instruction, it will read the `ip` from the newly called function's CallFrame and jump to its code. The work for executing that call begins here:

因为字节码调度循环会从`frame`变量中读取数据，当VM执行下一条指令时，它会从新的被调用函数CallFrame中读取`ip`，并跳转到其代码处。执行该调用的工作从这里开始：

vm.c，在peek()方法后添加代码^19：

```
static bool callValue(Value callee, int argCount) {
  if (IS_OBJ(callee)) {
    switch (OBJ_TYPE(callee)) {
      case OBJ_FUNCTION:
        return call(AS_FUNCTION(callee), argCount);
      default:
        break; // Non-callable object type.
    }
  }
  runtimeError("Can only call functions and classes.");
  return false;
}
```

> There's more going on here than just initializing a new CallFrame. Because Lox is dynamically typed, there's nothing to prevent a user from writing bad code like:

这里要做的不仅仅是初始化一个新的CallFrame，因为Lox是动态类型的，所以没有什么可以防止用户编写这样的糟糕代码：

```
var notAFunction = 123;
notAFunction();
```

> If that happens, the runtime needs to safely report an error and halt. So the first thing we do is check the type of the value that we're trying to call. If it's not a function, we error out. Otherwise, the actual call happens here:

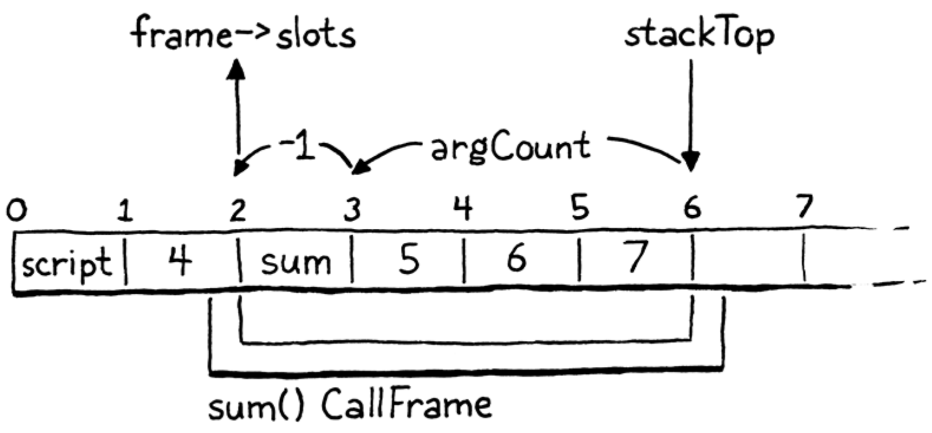
如果发生这种情况，运行时需要安全报告错误并停止。所以我们要做的第一件事就是检查我们要调用的值的类型。如果不是函数，我们就报错退出。否则，真正的调用就发生在这里：

vm.c，在peek()方法后添加代码：

```
static bool call(ObjFunction* function, int argCount) {
  CallFrame* frame = &vm.frames[vm.frameCount++];
  frame->function = function;
  frame->ip = function->chunk.code;
  frame->slots = vm.stackTop - argCount - 1;
  return true;
}
```

> This simply initializes the next CallFrame on the stack. It stores a pointer to the function being called and points the frame's `ip` to the beginning of the function's bytecode. Finally, it sets up the `slots` pointer to give the frame its window into the stack. The arithmetic there ensures that the arguments already on the stack line up with the function's parameters:

这里只是初始化了栈上的下一个CallFrame。其中存储了一个指向被调用函数的指针，并将调用帧的`ip`指向函数字节码的开始处。最后，它设置`slots`指针，告诉调用帧它在栈上的窗口位置。这里的算法可以确保栈中已存在的实参与函数的形参是对齐的。



> The funny little `- 1` is to account for stack slot zero which the compiler set aside for when we add methods later. The parameters start at slot one so we make the window start one slot earlier to align them with the arguments.

这个有趣的`-1`是为了处理栈槽0，编译器留出了这个槽，以便稍后添加方法时使用。形参从栈槽1开始，所以我们让窗口提前一个槽开始，以使它们与实参对齐。

> Before we move on, let’s add the new instruction to our disassembler.

在我们更进一步之前，让我们把新指令添加到反汇编程序中。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return jumpInstruction("OP_LOOP", -1, chunk, offset);
    // 新增部分开始
    case OP_CALL:
      return byteInstruction("OP_CALL", chunk, offset);
    // 新增部分结束
    case OP_RETURN:
```

> And one more quick side trip. Now that we have a handy function for initiating a CallFrame, we may as well use it to set up the first frame for executing the top-level code.

还有一个快速的小改动。现在我们有一个方便的函数用来初始化CallFrame，我们不妨用它来设置用于执行顶层代码的第一个帧。

vm.c，在interpret()方法中替换4行：

```
  push(OBJ_VAL(function));
  // 替换部分开始
  call(function, 0);
  // 替换部分结束
  return run();
```

> OK, now back to calls . . .

好了，现在回到调用……

> ### 24.5.2 Runtime error checking

24.5.2 运行时错误检查

> The overlapping stack windows work based on the assumption that a call passes exactly one argument for each of the function's parameters. But, again, because Lox ain't statically typed, a foolish user could pass too many or too few arguments. In Lox, we've defined that to be a runtime error, which we report like so:

重叠的栈窗口的工作基于这样一个假设：一次调用中正好为函数的每个形参传入一个实参。但是，同样的，由于Lox不是静态类型的，某个愚蠢的用户可以会传入太多或太少的参数。在Lox中，我们将其定义为运行时错误，并像这样报告：

vm.c，在call()方法中添加代码：

```
static bool call(ObjFunction* function, int argCount) {
  // 新增部分开始
  if (argCount != function->arity) {
    runtimeError("Expected %d arguments but got %d.",
        function->arity, argCount);
    return false;
  }
  // 新增部分结束
  CallFrame* frame = &vm.frames[vm.frameCount++];
```

> Pretty straightforward. This is why we store the arity of each function inside the ObjFunction for it.

非常简单直接。这就是为什么我们要在ObjFunction中存储每个函数的元数。

> There's another error we need to report that's less to do with the user's foolishness than our own. Because the CallFrame array has a fixed size, we need to ensure a deep call chain doesn't overflow it.

还有一个需要报告的错误，与其说是用户的愚蠢行为，不如说是我们自己的愚蠢行为。因为CallFrame数组具有固定的大小，我们需要确保一个深的调用链不会溢出。

vm.c，在call()方法中添加代码：

```
  }
  // 新增部分开始
  if (vm.frameCount == FRAMES_MAX) {
    runtimeError("Stack overflow.");
    return false;
  }
  // 新增部分结束
  CallFrame* frame = &vm.frames[vm.frameCount++];
```

> In practice, if a program gets anywhere close to this limit, there's most likely a bug in some runaway recursive code.

在实践中，如果一个程序接近这个极限，那么很可能在某些失控的递归代码中出现了错误。

> 24.5.3 Printing stack traces

24.5.3 打印栈跟踪记录

> While we're on the subject of runtime errors, let's spend a little time making them more useful. Stopping on a runtime error is important to prevent the VM from crashing and burning in some ill-defined way. But simply aborting doesn't help the user fix their code that *caused* that error.

既然我们在讨论运行时错误，那我们就花一点时间让它们变得更有用。在出现运行时错误时停止很重要，可以防止虚拟机以某种不明确的方式崩溃。但是简单的中止并不能帮助用户修复导致错误的代码。

> The classic tool to aid debugging runtime failures is a **stack trace**—a print out of each function that was still executing when the program died, and where the execution was at the point that it died. Now that we have a call stack and we've conveniently stored each function's name, we can show that entire stack when a runtime error disrupts the harmony of the user's existence. It looks like this:

帮助调试运行时故障的经典工具是**堆栈跟踪**——打印出程序死亡时仍在执行的每个函数，以及程序死亡时执行的位置。现在我们有了一个调度栈，并且方便地存储了每个函数的名称。当运行时错误破坏了用户的和谐时，我们可以显示整个堆栈。它看起来像这样：

vm.c，在runtimeError()方法中替换4行[^20]：

```
  fputs("\n", stderr);
  // 替换部分开始
  for (int i = vm.frameCount - 1; i >= 0; i--) {
    CallFrame* frame = &vm.frames[i];
    ObjFunction* function = frame->function;
    size_t instruction = frame->ip - function->chunk.code - 1;
    fprintf(stderr, "[line %d] in ",
            function->chunk.lines[instruction]);
    if (function->name == NULL) {
      fprintf(stderr, "script\n");
    } else {
      fprintf(stderr, "%s()\n", function->name->chars);
    }
  }
  // 替换部分结束
  resetStack();
}
```

> After printing the error message itself, we walk the call stack from top (the most recently called function) to bottom (the top-level code). For each frame, we find the line number that corresponds to the current `ip` inside that frame's function. Then we print that line number along with the function name.

在打印完错误信息本身之后，我们从顶部（最近调用的函数）到底部（顶层代码）遍历调用栈[^21]。对于每个调用帧，我们找到与该帧的函数内的当前`ip`相对应的行号。然后我们将该行号与函数名称一起打印出来。

> For example, if you run this broken program:

举例来说，如果你运行这个坏掉的程序：

```
fun a() { b(); }
fun b() { c(); }
fun c() {
  c("too", "many");
}

a();
```

> It prints out:

它会打印：

```
Expected 0 arguments but got 2.
[line 4] in c()
[line 2] in b()
[line 1] in a()
[line 7] in script
```

> That doesn't look too bad, does it?

看起来还不错，是吧？

> ### 24.5.4 Returning from functions

24.5.4 从函数中返回

> We're getting close. We can call functions, and the VM will execute them. But we can't *return* from them yet. We've had an `OP_RETURN` instruction for quite some time, but it's always had some kind of temporary code hanging out in it just to get us out of the bytecode loop. The time has arrived for a real implementation.

我们快完成了。我们可以调用函数，而虚拟机会执行它们。但是我们还不能从函数中返回。我们支持`OP_RETURN`指令已经有一段时间了，但其中一直有一些临时代码，只是为了让我们脱离字节码循环。现在是真正实现它的时候了。

vm.c，在run()方法中替换2行：

```
      case OP_RETURN: {
        // 替换部分开始
        Value result = pop();
        vm.frameCount--;
        if (vm.frameCount == 0) {
          pop();
          return INTERPRET_OK;
        }

        vm.stackTop = frame->slots;
        push(result);
        frame = &vm.frames[vm.frameCount - 1];
        break;
        // 替换部分结束
      }
```

> When a function returns a value, that value will be on top of the stack. We're about to discard the called function's entire stack window, so we pop that return value off and hang on to it. Then we discard the CallFrame for the returning function. If that was the very last CallFrame, it means we've finished executing the top-level code. The entire program is done, so we pop the main script function from the stack and then exit the interpreter.

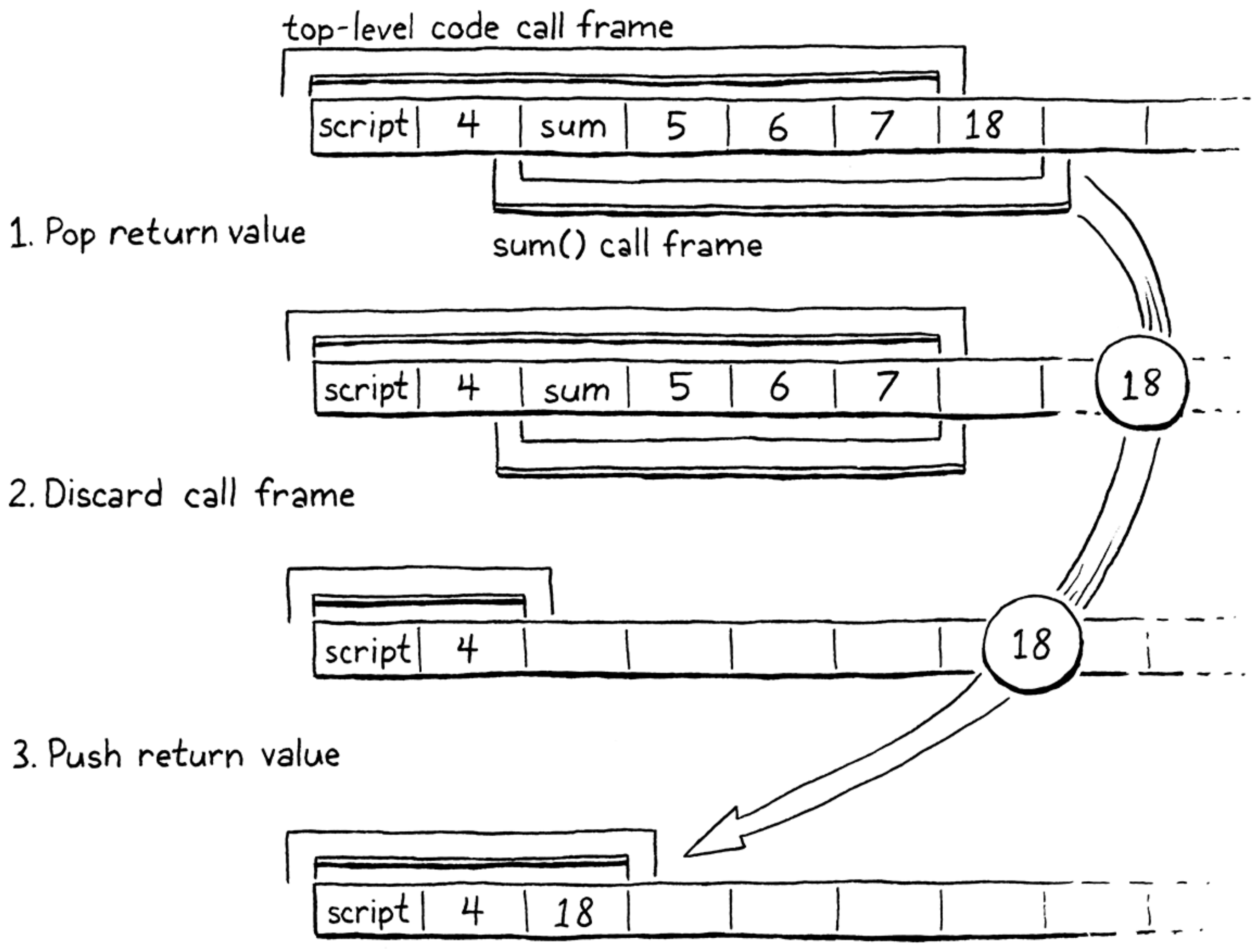
当函数返回一个值时，该值会在栈顶。我们将会丢弃被调用函数的整个堆栈窗口，因此我们将返回值弹出栈并保留它。然后我们丢弃CallFrame，从函数中返回。如果是最后一个CallFrame，这意味着我们已经完成了顶层代码的执行。整个程序已经完成，所以我们从堆栈中弹出主脚本函数，然后退出解释器。

> Otherwise, we discard all of the slots the callee was using for its parameters and local variables. That includes the same slots the caller used to pass the arguments. Now that the call is done, the caller doesn't need them anymore. This means the top of the stack ends up right at the beginning of the returning function's stack window.

否则，我们会丢弃所有被调用者用于存储参数和局部变量的栈槽，其中包括调用者用来传递实参的相同的槽。现在调用已经完成，调用者不再需要它们了。这意味着栈顶的结束位置正好在返回函数的栈窗口的开头。

> We push the return value back onto the stack at that new, lower location. Then we update the `run()` function's cached pointer to the current frame. Just like when we began a call, on the next iteration of the bytecode dispatch loop, the VM will read `ip` from that frame, and execution will jump back to the caller, right where it left off, immediately after the `OP_CALL` instruction.

我们把返回值压回堆栈，放在新的、较低的位置。然后我们更新`run`函数中缓存的指针，将其指向当前帧。就像我们开始调用一样，在字节码调度循环的下一次迭代中，VM会从该帧中读取`ip`，执行程序会跳回调用者，就在它离开的地方，紧挨着`OP_CALL`指令之后。



> Note that we assume here that the function *did* actually return a value, but a function can implicitly return by reaching the end of its body:

请注意，我们这里假设函数确实返回了一个值，但是函数可以在到达主体末尾时隐式返回：

```
fun noReturn() {
  print "Do stuff";
  // No return here.
}

print noReturn(); // ???
```

> We need to handle that correctly too. The language is specified to implicitly return `nil` in that case. To make that happen, we add this:

我们也需要正确地处理这个问题。在这种情况下，语言被指定为隐式返回`nil`。为了实现这一点，我们添加了以下内容：

compiler.c，在emitReturn()方法中添加代码：

```
static void emitReturn() {
  // 新增部分开始
  emitByte(OP_NIL);
  // 新增部分结束
  emitByte(OP_RETURN);
}
```

> The compiler calls `emitReturn()` to write the `OP_RETURN` instruction at the end of a function body. Now, before that, it emits an instruction to push `nil` onto the stack. And with that, we have working function calls! They can even take parameters! It almost looks like we know what we're doing here.

编译器调用`emitReturn()`，在函数体的末尾写入`OP_RETURN`指令。现在，在此之前，它会生成一条指令将`nil`压入栈中。这样，我们就有了可行的函数调用！它们甚至可以接受参数！看起来我们好像知道自己在做什么。

> ## 24.6 Return Statements

24.6 Return语句

> If you want a function that returns something other than the implicit `nil`, you need a `return` statement. Let's get that working.

如果你想让某个函数返回一些数据，而不是隐式的`nil`，你就需要一个`return`语句。我们来完成它。

compiler.c，在statement()方法中添加代码：

```
    ifStatement();
  // 新增部分开始
  } else if (match(TOKEN_RETURN)) {
    returnStatement();
  // 新增部分结束
  } else if (match(TOKEN_WHILE)) {
```

> When the compiler sees a `return` keyword, it goes here:

当编译器看到`return`关键字时，会进入这里：

compiler.c，在printStatement()方法后添加代码：

```
static void returnStatement() {
  if (match(TOKEN_SEMICOLON)) {
    emitReturn();
  } else {
    expression();
    consume(TOKEN_SEMICOLON, "Expect ';' after return value.");
    emitByte(OP_RETURN);
  }
}
```

> The return value expression is optional, so the parser looks for a semicolon token to tell if a value was provided. If there is no return value, the statement implicitly returns `nil`. We implement that by calling

> emitReturn(), which emits an OP_NIL instruction. Otherwise, we compile the return value expression and return it with an OP_RETURN instruction.

返回值表达式是可选的，因此解析器会寻找分号标识来判断是否提供了返回值。如果没有返回值，语句会隐式地返回nil。我们通过调用emitReturn()来实现，该函数会生成一个OP_NIL指令。否则，我们编译返回值表达式，并用OP_RETURN指令将其返回。

> This is the same OP_RETURN instruction we've already implemented—we don't need any new runtime code. This is quite a difference from jlox. There, we had to use exceptions to unwind the stack when a return statement was executed. That was because you could return from deep inside some nested blocks. Since jlox recursively walks the AST, that meant there were a bunch of Java method calls we needed to escape out of.

这与我们已经实现的OP_RETURN指令相同——我们不需要任何新的运行时代码。这与jlox有很大的不同。在jlox中，当执行return语句时，我们必须使用异常来跳出堆栈。这是因为你可以从某些嵌套的代码块深处返回。因为jlox递归地遍历AST。这意味着我们需要从一堆Java方法调用中退出。

> Our bytecode compiler flattens that all out. We do recursive descent during parsing, but at runtime, the VM's bytecode dispatch loop is completely flat. There is no recursion going on at the C level at all. So returning, even from within some nested blocks, is as straightforward as returning from the end of the function's body.

我们的字节码编译器把这些都扁平化了。我们在解析时进行递归下降，但在运行时，虚拟机的字节码调度循环是完全扁平的。在C语言级别上根本没有发生递归。因此，即使从一些嵌套代码块中返回，也和从函数体的末端返回一样简单。

> We're not totally done, though. The new return statement gives us a new compile error to worry about. Returns are useful for returning from functions but the top level of a Lox program is imperative code too. You shouldn't be able to return from there.

不过，我们还没有完全完成。新的return语句为我们带来了一个新的编译错误。return语句从函数中返回是很有用的，但是Lox程序的顶层代码也是命令式代码。你不能从那里返回[^22]。

```
return "What?!";
```

> We've specified that it's a compile error to have a return statement outside of any function, which we implement like so:

我们已经规定，在任何函数之外有return语句都是编译错误，我们这样实现：

compiler.c，在returnStatement()方法中添加代码：

```
static void returnStatement() {
  // 新增部分开始
  if (current->type == TYPE_SCRIPT) {
    error("Can't return from top-level code.");
  }
```

```
    // 新增部分结束
    if (match(TOKEN_SEMICOLON)) {
```

> This is one of the reasons we added that FunctionType enum to the compiler.

这是我们在编译器中添加FunctionType枚举的原因之一。

> ## 24.7 Native Functions

24.7 本地函数

> Our VM is getting more powerful. We've got functions, calls, parameters, returns. You can define lots of different functions that can call each other in interesting ways. But, ultimately, they can't really *do* anything. The only user-visible thing a Lox program can do, regardless of its complexity, is print. To add more capabilities, we need to expose them to the user.

我们的虚拟机越来越强大。我们已经支持了函数、调用、参数、返回。你可以定义许多不同的函数，它们可以以有趣的方式相互调用。但是，最终，它们什么都做不了。不管Lox程序有多复杂，它唯一能做的用户可见的事情就是打印。为了添加更多的功能，我们需要将函数暴露给用户。

> A programming language implementation reaches out and touches the material world through **native functions**. If you want to be able to write programs that check the time, read user input, or access the file system, we need to add native functions—callable from Lox but implemented in C—that expose those capabilities.

编程语言的实现通过**本地函数**向外延伸并接触物质世界。如果你想编写检查时间、读取用户输入或访问文件系统的程序，则需要添加本地函数——可以从Lox调用，但是使用C语言实现——来暴露这些能力。

> At the language level, Lox is fairly complete—it's got closures, classes, inheritance, and other fun stuff. One reason it feels like a toy language is because it has almost no native capabilities. We could turn it into a real language by adding a long list of them.

在语言层面，Lox是相当完整的——它支持闭包、类、继承和其它有趣的东西。它之所以给人一种玩具语言的感觉，是因为它几乎没有原生功能。我们可以通过添加一系列功能将其变成一种真正的语言。

> However, grinding through a pile of OS operations isn't actually very educational. Once you've seen how to bind one piece of C code to Lox, you get the idea. But you do need to see *one*, and even a single native function requires us to build out all the machinery for interfacing Lox with C. So we'll go through that and do all the hard work. Then, when that's done, we'll add one tiny native function just to prove that it works.

然而，辛辛苦苦地完成一堆操作系统的操作，实际上并没有什么教育意义。只要你看到如何将一段C代码与Lox绑定，你就会明白了。但你确实需要看到一个例子，即使只是一个本地函数，我们也需要构建将Lox与C语言对接的所有机制。所以我们将详细讨论这个问题并完成所有困难的工作。等这些工作完成之后，我们会添加一个小小的本地函数，以证明它是可行的。

> The reason we need new machinery is because, from the implementation's perspective, native functions are different from Lox functions. When they are called, they don't push a CallFrame, because there's no bytecode code for that frame to point to. They have no bytecode chunk. Instead, they somehow reference a piece of native C code.

我们需要新机制的原因是，从实现的角度来看，本地函数与Lox函数不同。当它们被调用时，它们不会压入一个CallFrame，因为没有这个帧要指向的字节码。它们没有字节码块。相反，它们会以某种方式引用一段本地C代码。

> We handle this in clox by defining native functions as an entirely different object type.

在clox中，我们通过将本地函数定义为一个完全不同的对象类型来处理这个问题。

object.h，在结构体ObjFunction后添加代码：

```
} ObjFunction;
// 新增部分开始
typedef Value (*NativeFn)(int argCount, Value* args);

typedef struct {
  Obj obj;
  NativeFn function;
} ObjNative;
// 新增部分结束
struct ObjString {
```

> The representation is simpler than ObjFunction—merely an Obj header and a pointer to the C function that implements the native behavior. The native function takes the argument count and a pointer to the first argument on the stack. It accesses the arguments through that pointer. Once it’s done, it returns the result value.

其表示形式比ObjFunction更简单——仅仅是一个Obj头和一个指向实现本地行为的C函数的指针。该本地函数接受参数数量和指向栈中第一个参数的指针。它通过该指针访问参数。一旦执行完成，它就返回结果值。

> As always, a new object type carries some accoutrements with it. To create an ObjNative, we declare a constructor-like function.

一如既往，一个新的对象类型会带有一些附属品。为了创建ObjNative，我们声明一个类似构造器的函数。

object.h，在newFunction()方法后添加代码：

```
ObjFunction* newFunction();
// 新增部分开始
ObjNative* newNative(NativeFn function);
// 新增部分结束
ObjString* takeString(char* chars, int length);
```

> We implement that like so:

我们这样实现它：

object.c，在newFunction()方法后添加代码：

```
ObjNative* newNative(NativeFn function) {
  ObjNative* native = ALLOCATE_OBJ(ObjNative, OBJ_NATIVE);
  native->function = function;
  return native;
}
```

> The constructor takes a C function pointer to wrap in an ObjNative. It sets up the object header and stores the function. For the header, we need a new object type.

该构造函数接受一个C函数指针，并将其包装在ObjNative中。它会设置对象头并保存传入的函数。至于对象头，我们需要一个新的对象类型。

object.h，在枚举ObjType中添加代码：

```
typedef enum {
  OBJ_FUNCTION,
  // 新增部分结束
  OBJ_NATIVE,
  // 新增部分开始
  OBJ_STRING,
} ObjType;
```

> The VM also needs to know how to deallocate a native function object.

虚拟机也需要知道如何释放本地函数对象。

memory.c，在freeObject()方法中添加代码：

```
    }
    // 新增部分开始
    case OBJ_NATIVE:
      FREE(ObjNative, object);
      break;
    // 新增部分结束
    case OBJ_STRING: {
```

> There isn't much here since ObjNative doesn't own any extra memory. The other capability all Lox objects support is being printed.

因为ObjNative并没有占用任何额外的内存，所以这里没有太多要做的。所有Lox对象需要支持的另一个功能是能够被打印。

object.c，在printObject()方法中添加代码：

```
      break;
    // 新增部分开始
    case OBJ_NATIVE:
      printf("<native fn>");
      break;
    // 新增部分结束
    case OBJ_STRING:
```

> In order to support dynamic typing, we have a macro to see if a value is a native function.

为了支持动态类型，我们用一个宏来检查某个值是否本地函数。

object.h，添加代码：

```
#define IS_FUNCTION(value)     isObjType(value, OBJ_FUNCTION)
// 新增部分开始
#define IS_NATIVE(value)       isObjType(value, OBJ_NATIVE)
// 新增部分结束
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
```

> Assuming that returns true, this macro extracts the C function pointer from a Value representing a native function:

如果返回值为真，下面这个宏可以从一个代表本地函数的Value中提取C函数指针：

object.h，添加代码：

```
#define AS_FUNCTION(value)     ((ObjFunction*)AS_OBJ(value))
// 新增部分开始
#define AS_NATIVE(value) \
    (((ObjNative*)AS_OBJ(value))->function)
// 新增部分结束
#define AS_STRING(value)       ((ObjString*)AS_OBJ(value))
```

> All of this baggage lets the VM treat native functions like any other object. You can store them in variables, pass them around, throw them birthday parties, etc. Of course, the operation we actually care about is *calling* them—using one as the left-hand operand in a call expression.

所有这些使得虚拟机可以像对待其它对象一样对待本地函数。你可以将它们存储在变量中，传递它们，给它们举办生日派对，等等。当然，我们真正关心的是*调用*它们——将一个本地函数作为调用表达式的左操作数。

> Over in `callValue()` we add another type case.

在 `callValue()`中，我们添加另一个类型的case分支。

vm.c，在callValue()方法中添加代码：

```
      case OBJ_FUNCTION:
        return call(AS_FUNCTION(callee), argCount);
      // 新增部分开始
      case OBJ_NATIVE: {
        NativeFn native = AS_NATIVE(callee);
        Value result = native(argCount, vm.stackTop - argCount);
        vm.stackTop -= argCount + 1;
        push(result);
        return true;
      }
      // 新增部分结束
      default:
```

> If the object being called is a native function, we invoke the C function right then and there. There's no need to muck with CallFrames or anything. We just hand off to C, get the result, and stuff it back in the stack. This makes native functions as fast as we can get.

如果被调用的对象是一个本地函数，我们就会立即调用C函数。没有必要使用CallFrames或其它任何东西。我们只需要交给C语言，得到结果，然后把结果塞回栈中。这使得本地函数的运行速度能够尽可能快。

> With this, users should be able to call native functions, but there aren't any to call. Without something like a foreign function interface, users can't define their own native functions. That's our job as VM implementers. We'll start with a helper to define a new native function exposed to Lox programs.

有了这个，用户应该能够调用本地函数了，但是还没有任何函数可供调用。如果没有外部函数接口之类的东西，用户就不能定义自己的本地函数。这就是我们作为虚拟机实现者的工作。我们将从一个辅助函数开始，定义一个新的本地函数暴露给Lox程序。

vm.c，在runtimeError()方法后添加代码：

```
static void defineNative(const char* name, NativeFn function) {
  push(OBJ_VAL(copyString(name, (int)strlen(name))));
  push(OBJ_VAL(newNative(function)));
  tableSet(&vm.globals, AS_STRING(vm.stack[0]), vm.stack[1]);
  pop();
  pop();
}
```

> It takes a pointer to a C function and the name it will be known as in Lox. We wrap the function in an ObjNative and then store that in a global variable with the given name.

它接受一个指向C函数的指针及其在Lox中的名称。我们将函数包装在ObjNative中，然后将其存储在一个带有指定名称的全局变量中。

> You're probably wondering why we push and pop the name and function on the stack. That looks weird, right? This is the kind of stuff you have to worry about when garbage collection gets involved. Both `copyString()` and `newNative()` dynamically allocate memory. That means once we have a GC, they can potentially trigger a collection. If that happens, we need to ensure the collector knows we're not done with the name and ObjFunction so that it doesn't free them out from under us. Storing them on the value stack accomplishes that.

你可能像知道为什么我们要在栈中压入和弹出名称与函数。看起来很奇怪，是吧？当涉及到垃圾回收时，你必须考虑这类问题。`copyString()`和`newNative()`都是动态分配内存的。这意味着一旦我们有了GC，它们就有可能触发一次收集。如果发生这种情况，我们需要确保收集器知道我们还没有用完名称和ObjFunction ，这样垃圾回收就不会将这些数据从我们手下释放出来。将它们存储在值栈中可以做到这一点[^23]。

> It feels silly, but after all of that work, we're going to add only one little native function.

这感觉很傻，但是在完成所有这些工作之后，我们只会添加一个小小的本地函数。

vm.c，在变量vm后添加代码：

```
static Value clockNative(int argCount, Value* args) {
  return NUMBER_VAL((double)clock() / CLOCKS_PER_SEC);
}
```

> This returns the elapsed time since the program started running, in seconds. It's handy for benchmarking Lox programs. In Lox, we'll name it `clock()`.

该函数会返回程序开始运行以来经过的时间，单位是秒。它对Lox程序的基准测试很有帮助。在Lox中，我们将其命名为`clock()`。

vm.c，在initVM()方法中添加代码：

```
  initTable(&vm.strings);
  // 新增部分开始
  defineNative("clock", clockNative);
  // 新增部分结束
}
```

> To get to the C standard library `clock()` function, the "vm" module needs an include.

为了获得C语言标准库中的`clock()`函数，`vm`模块需要引入头文件。

vm.c，添加代码：

```
#include <string.h>
// 新增部分开始
#include <time.h>
// 新增部分结束
#include "common.h"
```

> That was a lot of material to work through, but we did it! Type this in and try it out:

这部分有很多内容要处理，但是我们做到了！输入这段代码试试：

```
fun fib(n) {
  if (n < 2) return n;
  return fib(n - 2) + fib(n - 1);
}

var start = clock();
print fib(35);
print clock() - start;
```

> We can write a really inefficient recursive Fibonacci function. Even better, we can measure just *how* inefficient it is. This is, of course, not the smartest way to calculate a Fibonacci number. But it is a good way to stress test a language implementation's support for function calls. On my machine, running this in clox is about five times faster than in jlox. That's quite an improvement.

我们已经可以编写一个非常低效的递归斐波那契函数。更妙的是，我们可以测量它有多低效。当然，这不是计算斐波那契数的最聪明的方法，但这是一个针对语言实现对函数调用的支持进行压力测试的好方法。在我的机器上，clox中运行这个程序大约比jlox快5倍。这是个相当大的提升^24。

[^20]: 这里的`-1`是因为IP已经指向了下一条待执行的指令上 ，但我们希望堆栈跟踪指向前一条失败的指令。
[^21]: 关于栈帧在跟踪信息中显示的顺序，存在一些不同的意见。大部分把最内部的函数放在第一行，然后向堆栈的底部。Python则以相反的顺序打印出来。因此，从上到下阅读可以告诉你程序是如何达到现在的位置的，而最后一行是错误实际发生的地方。
这种风格有一个逻辑。它可以确保你始终可以看到最里面的函数，即使堆栈跟踪信息太长而无法在一个屏幕上显示。另一方面，新闻业中的"倒金字塔"告诉我们，我们应该把最重要的信息放在一段文字的前面。在堆栈跟踪中，这就是实际发生错误的函数。大多数其它语言的实现都是如此。 [^22]: 允许在顶层返回并不是世界上最糟糕的主意。它可以为你提供一种自然的方式来提前终止脚本。你甚至可以用返回的数字来表示进程的退出码。 [^23]: 如果你没搞懂也不用担心，一旦我们开始实现GC，它就会变得更有意义。 ^24: 它比在Ruby 2.4.3p205中运行的同类Ruby程序稍慢，比在Python 3.7.3中运行的程序快3倍左右。而且我们仍然可以在我们的虚拟机中做很多简单的优化。

习题

1. > Reading and writing the `ip` field is one of the most frequent operations inside the bytecode loop. Right now, we access it through a pointer to the current CallFrame. That requires a pointer indirection which may force the CPU to bypass the cache and hit main memory. That can be a real performance sink.
   >
   > Ideally, we'd keep the `ip` in a native CPU register. C doesn't let us *require* that without dropping into inline assembly, but we can structure the code to encourage the compiler to make that optimization. If we store the `ip` directly in a C local variable and mark it `register`, there's a good chance the C compiler will accede to our polite request.
   >
   > This does mean we need to be careful to load and store the local `ip` back into the correct CallFrame when starting and ending function calls. Implement this optimization. Write a couple of benchmarks and see how it affects the performance. Do you think the extra code complexity is worth it?

读写`ip`字段是字节码循环中最频繁的操作之一。新增，我们通过一个指向当前CallFrame的指针来访问它。这里需要一次指针间接引用，可能会迫使CPU绕过缓存而进入主存。这可能是一个真正的性能损耗。

理想情况下，我们一个将`ip`保存在一个本地CPU寄存器中。在不引入内联汇编的情况下，C语言中不允许我们这样做，但是我们可以通过结构化的代码来鼓励编译器进行优化。如果我们将`ip`直接存储在C局部变量中，并将其标记为`register`，那么C编译器很可能会同意我们的礼貌请求。

这确实意味着在开始和结束函数调用时，我们需要谨慎地从正确的CallFrame中加载和保存局部变量`ip`。请实现这一优化。写几个基准测试，看看它对性能有什么影响。您认为增加的代码复杂性值得吗？

2. > Native function calls are fast in part because we don't validate that the call passes as many arguments as the function expects. We really should, or an incorrect call to a native function without enough arguments could cause the function to read uninitialized memory. Add arity checking.

   本地函数调用之所以快，部分原因是我们没有验证调用时传入的参数是否与期望的一样多。我们确实应该这样做，否则在没有足够参数的情况下错误地调用本地函数，会导致函数读取未初始化的内存空间。请添加参数数量检查。

3. > Right now, there's no way for a native function to signal a runtime error. In a real implementation, this is something we'd need to support because native functions live in the statically typed world of C but are called from dynamically typed Lox land. If a user, say, tries to pass a string to `sqrt()`, that native function needs to report a runtime error.
   >
   > Extend the native function system to support that. How does this capability affect the performance of native calls?

   目前，本机函数还没有办法发出运行时错误的信号。在一个真正的语言实现中，这是我们需要支持的，因为本机函数存在于静态类型的C语言世界中，却被动态类型的Lox调用。假如说，用户试图向`sqrt()`传递一个字符串，则该本地函数需要报告一个运行时错误。

   扩展本地函数系统，以支持该功能。这个功能会如何影响本地调用的性能？

4. > Add some more native functions to do things you find useful. Write some programs using those. What did you add? How do they affect the feel of the language and how practical it is?

   添加一些本地函数来做你认为有用的事情。用它们写一些程序。你添加了什么？它们是如何影响语言的感觉和实用性的？

25.闭包 Closures

> As the man said, for every complex problem there's a simple solution, and it's wrong.
>
> ——Umberto Eco, *Foucault's Pendulum*

正如那人所说，每一个复杂的问题都有一个简单的解决方案，而且是错误的。（翁贝托·艾柯，《傅科摆》）

> Thanks to our diligent labor in the last chapter, we have a virtual machine with working functions. What it lacks is closures. Aside from global variables, which are their own breed of animal, a function has no way to reference a variable declared outside of its own body.

感谢我们在上一章的辛勤劳动，我们得到了一个拥有函数的虚拟机。现在虚拟机缺失的是闭包。除了全局变量（也就是函数的同类）之外，函数没有办法引用其函数体之外声明的变量。

```
var x = "global";
fun outer() {
  var x = "outer";
  fun inner() {
    print x;
  }
  inner();
}
outer();
```

> Run this example now and it prints "global". It's supposed to print "outer". To fix this, we need to include the entire lexical scope of all surrounding functions when resolving a variable.

现在运行这个示例，它打印的是"global"。但它应该打印"outer"。为了解决这个问题，我们需要在解析变量时涵盖所有外围函数的整个词法作用域。

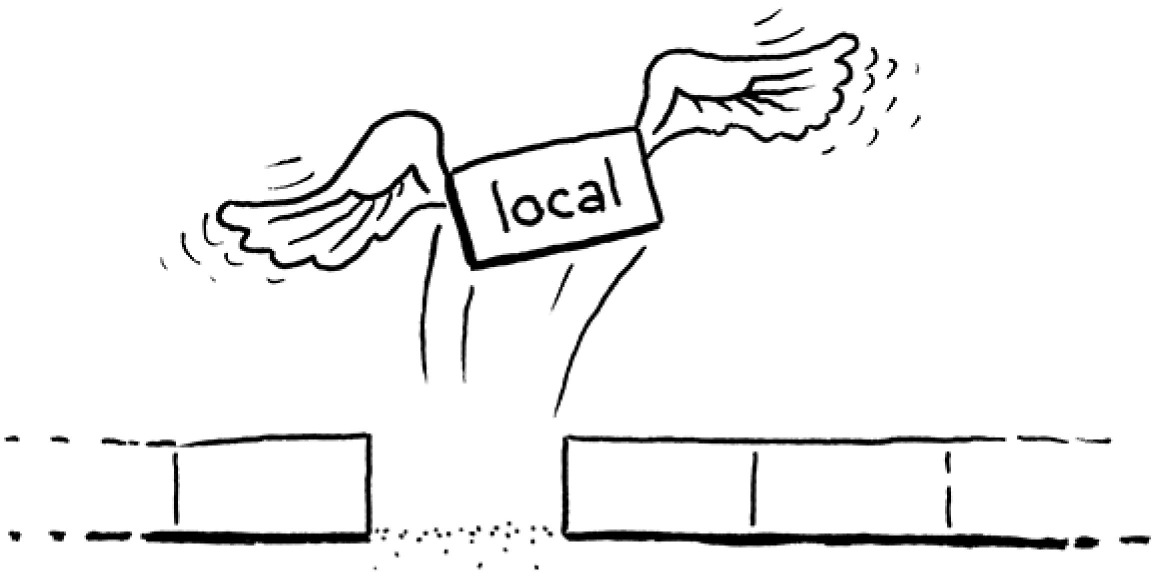
> This problem is harder in clox than it was in jlox because our bytecode VM stores locals on a stack. We used a stack because I claimed locals have stack semantics—variables are discarded in the reverse order that they are created. But with closures, that's only *mostly* true.

这个问题在clox中比在jlox中更难解决，因为我们的字节码虚拟机将局部变量存储在栈中。我们使用堆栈是因为，我声称局部变量具有栈语义——变量被丢弃的顺序与创建的顺序正好相反。但对于闭包来说，这只在大部分情况下是正确的。

```
fun makeClosure() {
  var local = "local";
  fun closure() {
    print local;
  }
  return closure;
}

var closure = makeClosure();
closure();
```

> The outer function `makeClosure()` declares a variable, `local`. It also creates an inner function, `closure()` that captures that variable. Then `makeClosure()` returns a reference to that function. Since the closure escapes while holding on to the local variable, `local` must outlive the function call where it was created.

外层函数`makeClosure()`声明了一个变量`local`。它还创建了一个内层函数`closure()`，用于捕获该变量。然后`makeClosure()`返回对该内层函数的引用。因为闭包要在保留局部变量的同时进行退出，所以`local`必须比创建它的函数调用存活更长的时间。



> We could solve this problem by dynamically allocating memory for all local variables. That's what jlox does by putting everything in those Environment objects that float around in Java's heap. But we don't want to. Using a stack is *really* fast. Most local variables are *not* captured by closures and do have stack semantics. It would suck to make all of those slower for the benefit of the rare local that is captured.

我们可以通过为所有局部变量动态地分配内存来解决这个问题。这就是jlox所做的，它将所有对象都放在Java堆中漂浮的Environment对象中。但我们并不想这样做。使用堆栈*非常*快。大多数局部变量都不会被闭包捕获，并且具有栈语义。如果为了极少数被捕获的局部变量而使所有变量的速度变慢，那就糟糕了^1。

> This means a more complex approach than we used in our Java interpreter. Because some locals have very different lifetimes, we will have two implementation strategies. For locals that aren't used in closures, we'll keep them just as they are on the stack. When a local is captured by a closure, we'll adopt another solution that lifts them onto the heap where they can live as long as needed.

这意味着一种比我们在Java解释器中所用的更复杂的方法。因为有些局部变量具有非常不同的生命周期，我们将有两种实现策略。对于那些不在闭包中使用的局部变量，我们将保持它们在栈中的原样。当某个局部变量被闭包捕获时，我们将采用另一种解决方案，将它们提升到堆中，在那里它们存活多久都可以。

> Closures have been around since the early Lisp days when bytes of memory and CPU cycles were more precious than emeralds. Over the intervening decades, hackers devised all manner of ways to compile closures to optimized runtime representations. Some are more efficient but require a more complex compilation process than we could easily retrofit into clox.

闭包早在Lisp时代就已经存在了，当时内存字节和CPU周期比祖母绿还要珍贵。在过去的几十年里，黑客们设计了各种各样的方式来编译闭包，以优化运行时表示^2。有些方法更有效，但也需要更复杂的编译过程，我们无法轻易地在clox中加以改造。

> The technique I explain here comes from the design of the Lua VM. It is fast, parsimonious with memory, and implemented with relatively little code. Even more impressive, it fits naturally into the single-pass compilers clox and Lua both use. It is somewhat intricate, though. It might take a while before all the pieces click together in your mind. We'll build them one step at a time, and I'll try to introduce the concepts in stages.

我在这里解释的技术来自于Lua虚拟机的设计。它速度快，内存占用少，并且只用相对较少的代码就实现了。更令人印象深刻的是，它很自然地适用于clox和Lua都在使用的单遍编译器。不过，它有些复杂，可能需要一段时

间才能把所有的碎片在你的脑海中拼凑起来。我们将一步一步地构建它们，我将尝试分阶段介绍这些概念。

25.1 Closure Objects

25.1 闭包对象

> Our VM represents functions at runtime using ObjFunction. These objects are created by the front end during compilation. At runtime, all the VM does is load the function object from a constant table and bind it to a name. There is no operation to "create" a function at runtime. Much like string and number literals, they are constants instantiated purely at compile time.

我们的虚拟机在运行时使用ObjFunction表示函数。这些对象是由前端在编译时创建的。在运行时，虚拟机所做的就是从一个常量表中加载函数对象，并将其与一个名称绑定。在运行时，没有"创建"函数的操作。与字符串和数字字面量一样，它们是纯粹在编译时实例化的常量^3。

> That made sense because all of the data that composes a function is known at compile time: the chunk of bytecode compiled from the function's body, and the constants used in the body. Once we introduce closures, though, that representation is no longer sufficient. Take a gander at:

这是有道理的，因为组成函数的所有数据在编译时都是已知的：根据函数主体编译的字节码块，以及函数主体中使用的常量。一旦我们引入闭包，这种表示形式就不够了。请看一下：

```
fun makeClosure(value) {
  fun closure() {
    print value;
  }
  return closure;
}

var doughnut = makeClosure("doughnut");
var bagel = makeClosure("bagel");
doughnut();
bagel();
```

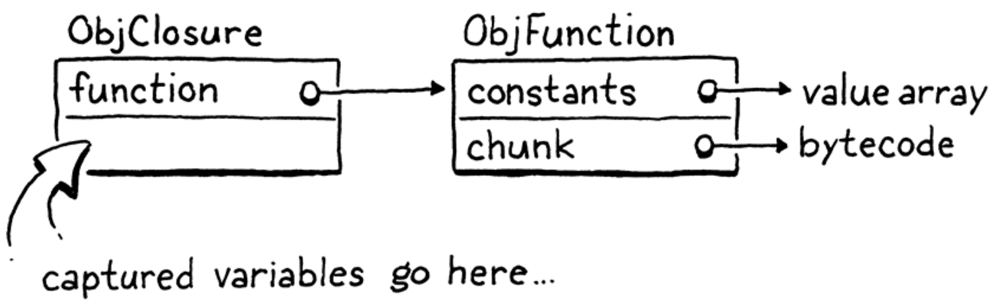
> The `makeClosure()` function defines and returns a function. We call it twice and get two closures back. They are created by the same nested function declaration, `closure`, but close over different values. When we call the two closures, each prints a different string. That implies we need some runtime representation for a closure that captures the local variables surrounding the function as they exist when the function declaration is *executed*, not just when it is compiled.

`makeClosure()`函数会定义并返回一个函数。我们调用它两次，得到两个闭包。它们都是由相同的嵌套函数声明`closure`创建的，但关闭在不同的值上。当我们调用这两个闭包时，每个闭包都打印出不同的字符串。这意味着我们需要一些闭包运行时表示，以捕获函数外围的局部变量，因为这些变量要在函数声明被*执行*时存在，而不仅仅是在编译时存在。

> We'll work our way up to capturing variables, but a good first step is defining that object representation. Our existing ObjFunction type represents the "raw" compile-time state of a function declaration, since all closures created from a single declaration share the same code and constants. At runtime, when we execute a function declaration, we wrap the ObjFunction in a new ObjClosure

> structure. The latter has a reference to the underlying bare function along with runtime state for the variables the function closes over.

我们会逐步来捕获变量，但良好的第一步是定义对象表示形式。我们现有的ObjFunction类型表示了函数声明的“原始”编译时状态，因为从同一个声明中创建的所有闭包都共享相同的代码和常量。在运行时，当我们执行函数声明时，我们将ObjFunction包装进一个新的ObjClosure结构体中。后者有一个对底层裸函数的引用，以及该函数关闭的变量的运行时状态^4。



> We'll wrap every function in an ObjClosure, even if the function doesn't actually close over and capture any surrounding local variables. This is a little wasteful, but it simplifies the VM because we can always assume that the function we're calling is an ObjClosure. That new struct starts out like this:

我们将用ObjClosure包装每个函数，即使该函数实际上并没有关闭或捕获任何外围局部变量。这有点浪费，但它简化了虚拟机，因为我们总是可以认为我们正在调用的函数是一个ObjClosure。这个新结构体是这样开始的：

object.h，在结构体ObjString后添加代码：

```
typedef struct {
  Obj obj;
  ObjFunction* function;
} ObjClosure;
```

> Right now, it simply points to an ObjFunction and adds the necessary object header stuff. Grinding through the usual ceremony for adding a new object type to clox, we declare a C function to create a new closure.

现在，它只是简单地指向一个ObjFunction，并添加了必要的对象头内容。遵循向clox中添加新对象类型的常规步骤，我们声明一个C函数来创建新闭包。

object.h，在结构体ObjClosure后添加代码：

```
ObjFunction
// 新增部分开始
ObjClosure* newClosure(ObjFunction* function);
// 新增部分结束
ObjFunction* newFunction();
```

> Then we implement it here:

然后我们在这里实现它：

object.c，在allocateObject()方法后添加代码：

```
ObjClosure* newClosure(ObjFunction* function) {
  ObjClosure* closure = ALLOCATE_OBJ(ObjClosure, OBJ_CLOSURE);
  closure->function = function;
  return closure;
}
```

> It takes a pointer to the ObjFunction it wraps. It also initializes the type field to a new type.

它接受一个指向待包装ObjFunction的指针。它还将类型字段初始为一个新类型。

object.h，在枚举ObjType中添加代码：

```
typedef enum {
  // 新增部分开始
  OBJ_CLOSURE,
  // 新增部分结束
  OBJ_FUNCTION,
```

> And when we're done with a closure, we release its memory.

以及，当我们用完闭包后，要释放其内存。

memory.c，在freeObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_CLOSURE: {
      FREE(ObjClosure, object);
      break;
    }
    // 新增部分结束
    case OBJ_FUNCTION: {
```

> We free only the ObjClosure itself, not the ObjFunction. That's because the closure doesn't *own* the function. There may be multiple closures that all reference the same function, and none of them claims any special privilege over it. We can't free the ObjFunction until *all* objects referencing it are gone—including even the surrounding function whose constant table contains it. Tracking that sounds tricky, and it is! That's why we'll write a garbage collector soon to manage it for us.

我们只释放ObjClosure本身，而不释放ObjFunction。这是因为闭包*不拥有*函数。可能会有多个闭包都引用了同一个函数，但没有一个闭包声称对该函数有任何特殊的权限。我们不能释放某个ObjFunction，直到引用它的*所有*对象全部消失——甚至包括那些常量表中包含该函数的外围函数。要跟踪这个信息听起来很棘手，事实也的确如此！这就是我们很快就会写一个垃圾收集器来管理它们的原因。

> We also have the usual macros for checking a value's type.

我们还有用于检查值类型的常用宏^5。

object.h，添加代码：

```
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分开始
#define IS_CLOSURE(value)      isObjType(value, OBJ_CLOSURE)
// 新增部分结束
#define IS_FUNCTION(value)     isObjType(value, OBJ_FUNCTION)
```

> And to cast a value:

还有值转换：

object.h，添加代码：

```
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分开始
#define AS_CLOSURE(value)      ((ObjClosure*)AS_OBJ(value))
// 新增部分结束
#define AS_FUNCTION(value)     ((ObjFunction*)AS_OBJ(value))
```

> Closures are first-class objects, so you can print them.

闭包是第一类对象，因此你可以打印它们。

object.c，在printObject()方法中添加代码：

```
  switch (OBJ_TYPE(value)) {
    // 新增部分开始
    case OBJ_CLOSURE:
      printFunction(AS_CLOSURE(value)->function);
      break;
    // 新增部分结束
    case OBJ_FUNCTION:
```

> They display exactly as ObjFunction does. From the user's perspective, the difference between ObjFunction and ObjClosure is purely a hidden implementation detail. With that out of the way, we have a working but empty representation for closures.

它们的显示和ObjFunction一样。从用户的角度来看，ObjFunction和ObjClosure之间的区别纯粹是一个隐藏的实现细节。有了这些，我们就有了一个可用但空白的闭包表示形式。

> 25.1.1 Compiling to closure objects

25.1.1 编译为闭包对象

> We have closure objects, but our VM never creates them. The next step is getting the compiler to emit instructions to tell the runtime when to create a new ObjClosure to wrap a given ObjFunction. This happens right at the end of a function declaration.

我们有了闭包对象，但是我们的VM还从未创建它们。下一步就是让编译器发出指令，告诉运行时何时创建一个新的ObjClosure来包装指定的ObjFunction。这就发生在函数声明的末尾。

compiler.c，在function()方法中替换1行：

```
  ObjFunction* function = endCompiler();
  // 替换部分开始
  emitBytes(OP_CLOSURE, makeConstant(OBJ_VAL(function)));
  // 替换部分结束
}
```

> Before, the final bytecode for a function declaration was a single `OP_CONSTANT` instruction to load the compiled function from the surrounding function's constant table and push it onto the stack. Now we have a new instruction.

之前，函数声明的最后一个字节码是一条`OP_CONSTANT`指令，用于从外围函数的常量表中加载已编译的函数，并将其压入堆栈。现在我们有了一个新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CALL,
  // 新增部分开始
  OP_CLOSURE,
  // 新增部分结束
  OP_RETURN,
```

> Like `OP_CONSTANT`, it takes a single operand that represents a constant table index for the function. But when we get over to the runtime implementation, we do something more interesting.

和`OP_CONSTANT`一样，它接受一个操作数，表示函数在常量表中的索引。但是等到进入运行时实现时，我们会做一些更有趣的事情。

> First, let's be diligent VM hackers and slot in disassembler support for the instruction.

首先，让我们做一个勤奋的虚拟机黑客，为该指令添加反汇编器支持。

debug.c，在disassembleInstruction()方法中添加代码：

```
    case OP_CALL:
      return byteInstruction("OP_CALL", chunk, offset);
    // 新增部分开始
    case OP_CLOSURE: {
      offset++;
      uint8_t constant = chunk->code[offset++];
      printf("%-16s %4d ", "OP_CLOSURE", constant);
      printValue(chunk->constants.values[constant]);
      printf("\n");
      return offset;
    }
    // 新增部分结束
    case OP_RETURN:
```

> There's more going on here than we usually have in the disassembler. By the end of the chapter, you'll discover that OP_CLOSURE is quite an unusual instruction. It's straightforward right now—just a single byte operand—but we'll be adding to it. This code here anticipates that future.

这里做的事情比我们通常在反汇编程序中看到的要多。在本章结束时，你会发现OP_CLOSURE是一个相当不寻常的指令。它现在很简单——只有一个单字节的操作数——但我们会增加它的内容。这里的代码预示了未来。

> 25.1.2 Interpreting function declarations

25.1.2 解释函数声明

> Most of the work we need to do is in the runtime. We have to handle the new instruction, naturally. But we also need to touch every piece of code in the VM that works with ObjFunction and change it to use ObjClosure instead—function calls, call frames, etc. We'll start with the instruction, though.

我们需要做的大部分工作是在运行时。我们必须处理新的指令，这是自然的。但是我们也需要触及虚拟机中每一段使用ObjFunction的代码，并将其改为使用ObjClosure——函数调用、调用帧，等等。不过，我们会从指令开始。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_CLOSURE: {
        ObjFunction* function = AS_FUNCTION(READ_CONSTANT());
        ObjClosure* closure = newClosure(function);
        push(OBJ_VAL(closure));
        break;
      }
      // 新增部分结束
      case OP_RETURN: {
```

> Like the OP_CONSTANT instruction we used before, first we load the compiled function from the constant table. The difference now is that we wrap that function in a new ObjClosure and push the result onto the stack.

与我们前面使用的OP_CONSTANT类似，首先从常量表中加载已编译的函数。现在的不同之处在于，我们将该函数包装在一个新的ObjClosure中，并将结果压入堆栈。

> Once you have a closure, you'll eventually want to call it.

一旦你有了一个闭包，你最终就会想要调用它。

vm.c，在callValue()方法中替换2行：

```
    switch (OBJ_TYPE(callee)) {
      // 替换部分开始
      case OBJ_CLOSURE:
        return call(AS_CLOSURE(callee), argCount);
      // 替换部分结束
      case OBJ_NATIVE: {
```

> We remove the code for calling objects whose type is `OBJ_FUNCTION`. Since we wrap all functions in ObjClosures, the runtime will never try to invoke a bare ObjFunction anymore. Those objects live only in constant tables and get immediately wrapped in closures before anything else sees them.

我们删除了调用`OBJ_FUNCTION`类型对象的代码。因为我们用ObjClosures包装了所有的函数，运行时永远不会再尝试调用原生的ObjFunction。这些原生函数对象只存在于常量表中，并在其它部分看到它们之前立即被封装在闭包中。

> We replace the old code with very similar code for calling a closure instead. The only difference is the type of object we pass to `call()`. The real changes are over in that function. First, we update its signature.

我们用非常相似的调用闭包的代码来代替旧代码。唯一的区别是传递给`call()`的类型。真正的变化在这个函数中。首先，我们更新它的签名。

vm.c，在函数call()中，替换1行：

```
// 替换部分开始
static bool call(ObjClosure* closure, int argCount) {
// 替换部分结束
  if (argCount != function->arity) {
```

> Then, in the body, we need to fix everything that referenced the function to handle the fact that we've introduced a layer of indirection. We start with the arity checking:

然后，在主体中，我们需要修正所有引用该函数的内容，以便处理我们引入中间层的问题。首先从元数检查开始：

vm.c，在call()方法中，替换3行：

```
static bool call(ObjClosure* closure, int argCount) {
  // 替换部分开始
  if (argCount != closure->function->arity) {
    runtimeError("Expected %d arguments but got %d.",
        closure->function->arity, argCount);
    // 替换部分结束
    return false;
```

> The only change is that we unwrap the closure to get to the underlying function. The next thing `call()` does is create a new CallFrame. We change that code to store the closure in the CallFrame and get the bytecode pointer from the closure's function.

唯一的变化是，我们解开闭包获得底层函数。`call()`做的下一件事是创建一个新的CallFrame。我们修改这段代码，将闭包存储在CallFrame中，并从闭包内的函数中获取字节码指针。

vm.c，在call()方法中，替换2行：

```
  CallFrame* frame = &vm.frames[vm.frameCount++];
  // 替换部分开始
  frame->closure = closure;
  frame->ip = closure->function->chunk.code;
  // 替换部分结束
  frame->slots = vm.stackTop - argCount - 1;
```

> This necessitates changing the declaration of CallFrame too.

这就需要修改CallFrame的声明。

vm.h，在结构体CallFrame中，替换1行：

```
typedef struct {
  // 替换部分开始
  ObjClosure* closure;
  // 替换部分结束
  uint8_t* ip;
```

> That change triggers a few other cascading changes. Every place in the VM that accessed CallFrame's function needs to use a closure instead. First, the macro for reading a constant from the current function's constant table:

这一更改触发了其它一些级联更改。VM中所有访问CallFrame中函数的地方都需要使用闭包来代替。首先，是从当前函数常量表中读取常量的宏：

vm.c，在run()方法中，替换2行：

```
    (uint16_t)((frame->ip[-2] << 8) | frame->ip[-1]))
// 替换部分开始
#define READ_CONSTANT() \
    (frame->closure->function->chunk.constants.values[READ_BYTE()])
// 替换部分结束
#define READ_STRING() AS_STRING(READ_CONSTANT())
```

> When `DEBUG_TRACE_EXECUTION` is enabled, it needs to get to the chunk from the closure.

当DEBUG_TRACE_EXECUTION被启用时，它需要从闭包中获取字节码块。

vm.c，在run()方法中，替换2行：

```
    printf("\n");
    // 替换部分开始
    disassembleInstruction(&frame->closure->function->chunk,
        (int)(frame->ip - frame->closure->function->chunk.code));
    // 替换部分结束
#endif
```

> Likewise when reporting a runtime error:

同样地，在报告运行时错误时也是如此：

vm.c，在runtimeError()方法中，替换1行：

```
    CallFrame* frame = &vm.frames[i];
    // 替换部分开始
    ObjFunction* function = frame->closure->function;
    // 替换部分结束
    size_t instruction = frame->ip - function->chunk.code - 1;
```

> Almost there. The last piece is the blob of code that sets up the very first CallFrame to begin executing the top-level code for a Lox script.

差不多完成了。最后一部分是用来设置第一个CallFrame以开始执行Lox脚本顶层程序的代码块。

vm.c，在interpret()方法中，替换1行^6：

```
  push(OBJ_VAL(function));
  // 替换部分开始
  ObjClosure* closure = newClosure(function);
  pop();
  push(OBJ_VAL(closure));
  call(closure, 0);
  // 替换部分结束
  return run();
```

> The compiler still returns a raw ObjFunction when compiling a script. That's fine, but it means we need to wrap it in an ObjClosure here, before the VM can execute it.

编译脚本时，编译器仍然返回一个原始的ObjFunction。这是可以的，但这意味着我们现在（也就是在VM能够执行它之前），需要将其包装在一个ObjClosure中。

> We are back to a working interpreter. The *user* can't tell any difference, but the compiler now generates code telling the VM to create a closure for each function declaration. Every time the VM executes a

> function declaration, it wraps the ObjFunction in a new ObjClosure. The rest of the VM now handles those ObjClosures floating around. That's the boring stuff out of the way. Now we're ready to make these closures actually *do* something.

我们又得到了一个可以工作的解释器。*用户*看不出有什么不同，但是编译器现在生成的代码会告诉虚拟机，为每一个函数声明创建一个闭包。每当VM执行一个函数声明时，它都会将ObjFunction包装在一个新的ObjClosure中。VM的其余部分会处理那些四处漂浮的ObjClosures。无聊的事情就到此为止吧。现在，我们准备让这些闭包实际*做*一些事情。

25.2 Upvalues

25.2 上值

> Our existing instructions for reading and writing local variables are limited to a single function's stack window. Locals from a surrounding function are outside of the inner function's window. We're going to need some new instructions.

我们现有的读写局部变量的指令只限于单个函数的栈窗口。来自外围函数的局部变量是在内部函数的窗口之外。我们需要一些新的指令。

> The easiest approach might be an instruction that takes a relative stack slot offset that can reach *before* the current function's window. That would work if closed-over variables were always on the stack. But as we saw earlier, these variables sometimes outlive the function where they are declared. That means they won't always be on the stack.

最简单的方法可能是一条指令，接受一个栈槽相对偏移量，可以访问当前函数窗口*之前*的位置。如果闭包变量始终在栈上，这是有效的。但正如我们前面看到的，这些变量的生存时间有时会比声明它们的函数更长。这意味着它们不会一直在栈中。

> The next easiest approach, then, would be to take any local variable that gets closed over and have it always live on the heap. When the local variable declaration in the surrounding function is executed, the VM would allocate memory for it dynamically. That way it could live as long as needed.

然后，次简单的方法是获取闭包使用的任意局部变量，并让它始终存活在堆中。当执行外围函数中的局部变量声明时，虚拟机会为其动态分配内存。这样一来，它就可以根据需要长期存活。

> This would be a fine approach if clox didn't have a single-pass compiler. But that restriction we chose in our implementation makes things harder. Take a look at this example:

如果clox不是单遍编译器，这会是一种很好的方法。但是我们在实现中所选择的这种限制使事情变得更加困难。看看这个例子：

```
fun outer() {
  var x = 1;    // (1)
  x = 2;        // (2)
  fun inner() { // (3)
    print x;
  }
  inner();
}
```

> Here, the compiler compiles the declaration of `x` at `(1)` and emits code for the assignment at `(2)`. It does that before reaching the declaration of `inner()` at `(3)` and discovering that `x` is in fact closed over. We don't have an easy way to go back and fix that already-emitted code to treat `x` specially. Instead, we want a solution that allows a closed-over variable to live on the stack exactly like a normal local variable *until the point that it is closed over*.

在这里，编译器在`(1)`处编译了`x`的声明，并在`(2)`处生成了赋值代码。这些发生在编译器到达在`(3)`处的`inner()`声明并发现`x`实际上被闭包引用之前。我们没有一种简单的方法来回溯并修复已生成的代码，以特殊处理`x`。相反，我们想要的解决方案是，*在变量被关闭之前*，允许它像常规的局部变量一样存在于栈中。

> Fortunately, thanks to the Lua dev team, we have a solution. We use a level of indirection that they call an **upvalue**. An upvalue refers to a local variable in an enclosing function. Every closure maintains an array of upvalues, one for each surrounding local variable that the closure uses.

幸运的是，感谢Lua开发团队，我们有了一个解决方案。我们使用一种他们称之为**上值**的中间层。上值指的是一个闭包函数中的局部变量。每个闭包都维护一个上值数组，每个上值对应闭包使用的外围局部变量。

> The upvalue points back into the stack to where the variable it captured lives. When the closure needs to access a closed-over variable, it goes through the corresponding upvalue to reach it. When a function declaration is first executed and we create a closure for it, the VM creates the array of upvalues and wires them up to “capture” the surrounding local variables that the closure needs.

上值指向栈中它所捕获的变量所在的位置。当闭包需要访问一个封闭的变量时，它会通过相应的上值(`upvalues`)得到该变量。当某个函数声明第一次被执行，而且我们为其创建闭包时，虚拟机会创建一个上值数组，并将其与闭包连接起来，以“捕获”闭包需要的外围局部变量。

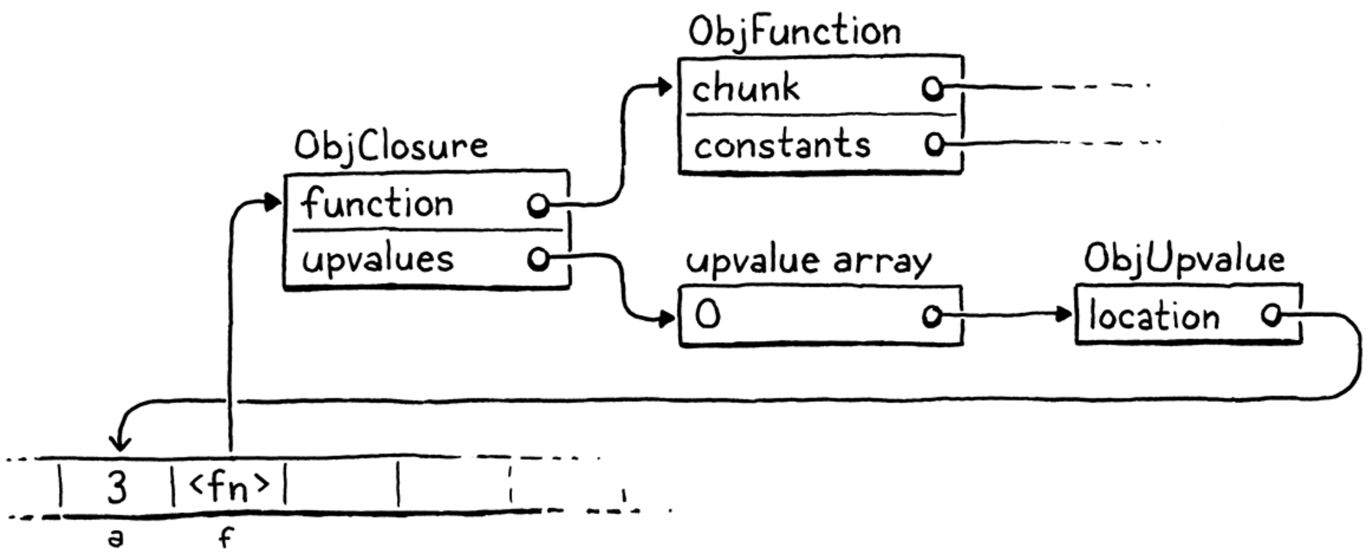
> For example, if we throw this program at clox,

举个例子，如果我们把这个程序扔给clox

```
{
  var a = 3;
  fun f() {
    print a;
  }
}
```

> the compiler and runtime will conspire together to build up a set of objects in memory like this:

编译器和运行时会合力在内存中构建一组这样的对象：



> That might look overwhelming, but fear not. We'll work our way through it. The important part is that upvalues serve as the layer of indirection needed to continue to find a captured local variable even after it moves off the stack. But before we get to all that, let's focus on compiling captured variables.

这可能看起来让人不知所措，但不要害怕。我们会用自己的方式来完成的。重要的部分是，上值充当了中间层，以便在被捕获的局部变量离开堆栈后能继续找到它。但在此之前，让我们先关注一下编译捕获的变量。

> ### 25.2.1 Compiling upvalues

25.2.1 编译上值

> As usual, we want to do as much work as possible during compilation to keep execution simple and fast. Since local variables are lexically scoped in Lox, we have enough knowledge at compile time to resolve which surrounding local variables a function accesses and where those locals are declared. That, in turn, means we know *how many* upvalues a closure needs, *which* variables they capture, and *which stack slots* contain those variables in the declaring function's stack window.

像往常一样，我们希望在编译期间做尽可能多的工作，从而保持执行的简单快速。由于局部变量在Lox是具有词法作用域的，我们在编译时有足够的信息来确定某个函数访问了哪些外围的局部变量，以及这些局部变量是在哪里声明的。反过来，这意味着我们知道闭包需要*多少个*上值，它们捕获了*哪个*变量，以及在声明函数的栈窗口中的*哪个栈槽*中包含这些变量。

> Currently, when the compiler resolves an identifier, it walks the block scopes for the current function from innermost to outermost. If we don't find the variable in that function, we assume the variable must be a global. We don't consider the local scopes of enclosing functions—they get skipped right over. The first change, then, is inserting a resolution step for those outer local scopes.

目前，当编译器解析一个标识符时，它会从最内层到最外层遍历当前函数的块作用域。如果我们没有在函数中找到该变量，我们就假定该变量一定是一个全局变量。我们不考虑封闭函数的局部作用域——它们会被直接跳过。那么，第一个变化就是为这些外围局部作用域插入一个解析步骤。

compiler.c，在namedVariable()方法中添加代码：

```
  if (arg != -1) {
    getOp = OP_GET_LOCAL;
    setOp = OP_SET_LOCAL;
  // 新增部分开始
  } else if ((arg = resolveUpvalue(current, &name)) != -1) {
    getOp = OP_GET_UPVALUE;
    setOp = OP_SET_UPVALUE;
  // 新增部分结束
  } else {
```

> This new `resolveUpvalue()` function looks for a local variable declared in any of the surrounding functions. If it finds one, it returns an “upvalue index” for that variable. (We’ll get into what that means later.) Otherwise, it returns -1 to indicate the variable wasn’t found. If it was found, we use these two new instructions for reading or writing to the variable through its upvalue:

这个新的`resolveUpvalue()`函数会查找在任何外围函数中声明的局部变量。如果找到了，就会返回该变量的“上值索引”。（我们稍后会解释这是什么意思）否则，它会返回`-1`，表示没有找到该变量。如果找到变量，我们就使用这两条新指令，通过其上值对变量进行读写：

chunk.h，在枚举OpCode中添加代码：

```
  OP_SET_GLOBAL,
  // 新增部分开始
  OP_GET_UPVALUE,
  OP_SET_UPVALUE,
  // 新增部分结束
  OP_EQUAL,
```

> We’re implementing this sort of top-down, so I’ll show you how these work at runtime soon. The part to focus on now is how the compiler actually resolves the identifier.

我们是自上而下实现的，所以我们很快会向你展示这些在运行时是如何工作的。现在要关注的部分是编译器实际上是如何解析标识符的。

compiler.c，在resolveLocal()方法后添加代码：

```
static int resolveUpvalue(Compiler* compiler, Token* name) {
  if (compiler->enclosing == NULL) return -1;

  int local = resolveLocal(compiler->enclosing, name);
  if (local != -1) {
    return addUpvalue(compiler, (uint8_t)local, true);
  }

  return -1;
}
```

> We call this after failing to resolve a local variable in the current function’s scope, so we know the variable isn’t in the current compiler. Recall that Compiler stores a pointer to the Compiler for the

> enclosing function, and these pointers form a linked chain that goes all the way to the root Compiler for the top-level code. Thus, if the enclosing Compiler is `NULL`, we know we've reached the outermost function without finding a local variable. The variable must be global, so we return -1.

在当前函数作用域中解析局部变量失败后，我们才会调用这个方法，因此我们知道该变量不在当前编译器中。回顾一下，Compiler中存储了一个指向外层函数Compiler的指针，这些指针形成了一个链，一直到顶层代码的根Compiler。因此，如果外围的Compiler是`NULL`，我们就知道已经到达最外层的函数，而且没有找到局部变量。那么该变量一定是全局的^7，所以我们返回`-1`。

> Otherwise, we try to resolve the identifier as a *local* variable in the *enclosing* compiler. In other words, we look for it right outside the current function. For example:

否则，我们尝试将标识符解析为一个在*外围*编译器中的*局部*变量。换句话说，我们在当前函数外面寻找它。举例来说：

```
fun outer() {
  var x = 1;
  fun inner() {
    print x; // (1)
  }
  inner();
}
```

> When compiling the identifier expression at `(1)`, `resolveUpvalue()` looks for a local variable `x` declared in `outer()`. If found—like it is in this example—then we've successfully resolved the variable. We create an upvalue so that the inner function can access the variable through that. The upvalue is created here:

当在`(1)`处编译标识符表达式时，`resolveUpvalue()`会查找在`outer()`中定义的局部变量`x`。如果找到了（就像本例中这样），那我们就成功解析了该变量。我们创建一个上值，以便内部函数可以通过它访问变量。上值是在这里创建的：

compiler.c，在resolveLocal()方法后添加代码：

```
static int addUpvalue(Compiler* compiler, uint8_t index,
                      bool isLocal) {
  int upvalueCount = compiler->function->upvalueCount;
  compiler->upvalues[upvalueCount].isLocal = isLocal;
  compiler->upvalues[upvalueCount].index = index;
  return compiler->function->upvalueCount++;
}
```

> The compiler keeps an array of upvalue structures to track the closed-over identifiers that it has resolved in the body of each function. Remember how the compiler's Local array mirrors the stack slot indexes where locals live at runtime? This new upvalue array works the same way. The indexes in the compiler's array match the indexes where upvalues will live in the ObjClosure at runtime.

编译器保留了一个上值结构的数组，用以跟踪每个函数主体中已解析的封闭标识符。还记得编译器的Local数组是如何反映局部变量在运行时所在的栈槽索引的吗？这个新的上值数组也使用相同的方式。编译器数组中的索引，与运行时ObjClosure中上值所在的索引相匹配。

> This function adds a new upvalue to that array. It also keeps track of the number of upvalues the function uses. It stores that count directly in the ObjFunction itself because we'll also need that number for use at runtime.

这个函数向数组中添加了一个新的上值。它还记录了该函数所使用的上值的数量。它直接在ObjFunction中存储了这个计数值，因为我们在运行时也需要使用这个数字^8。

> The `index` field tracks the closed-over local variable's slot index. That way the compiler knows *which* variable in the enclosing function needs to be captured. We'll circle back to what that `isLocal` field is for before too long. Finally, `addUpvalue()` returns the index of the created upvalue in the function's upvalue list. That index becomes the operand to the `OP_GET_UPVALUE` and `OP_SET_UPVALUE` instructions.

`index`字段记录了封闭局部变量的栈槽索引。这样，编译器就知道需要捕获外部函数中的*哪个*变量。用不了多久，我们会回过头来讨论`isLocal`字段的用途。最后，`addUpvalue()`返回已创建的上值在函数的上值列表中的索引。这个索引会成为`OP_GET_UPVALUE`和`OP_SET_UPVALUE`指令的操作数。

> That's the basic idea for resolving upvalues, but the function isn't fully baked. A closure may reference the same variable in a surrounding function multiple times. In that case, we don't want to waste time and memory creating a separate upvalue for each identifier expression. To fix that, before we add a new upvalue, we first check to see if the function already has an upvalue that closes over that variable.

这就是解析上值的基本思路，但是这个函数还没有完全成熟。一个闭包可能会多次引用外围函数中的同一个变量。在这种情况下，我们不想浪费时间和内存来为每个标识符表达式创建一个单独的上值。为了解决这个问题，在我们添加新的上值之前，我们首先要检查该函数是否已经有封闭该变量的上值。

compiler.c，在addUpvalue()方法中添加代码：

```
  int upvalueCount = compiler->function->upvalueCount;
  // 新增部分开始
  for (int i = 0; i < upvalueCount; i++) {
    Upvalue* upvalue = &compiler->upvalues[i];
    if (upvalue->index == index && upvalue->isLocal == isLocal) {
      return i;
    }
  }
  // 新增部分结束
  compiler->upvalues[upvalueCount].isLocal = isLocal;
```

> If we find an upvalue in the array whose slot index matches the one we're adding, we just return that *upvalue* index and reuse it. Otherwise, we fall through and add the new upvalue.

如果我们在数组中找到与待添加的上值索引相匹配的上值，我们就返回该*上值*的索引并复用它。否则，我们就放弃，并添加新的上值。

> These two functions access and modify a bunch of new state, so let's define that. First, we add the upvalue count to ObjFunction.

这两个函数访问并修改了一些新的状态，所以我们来定义一下。首先，我们将上值计数添加到ObjFunction中。

object.h，在结构体ObjFunction中添加代码：

```
  int arity;
  // 新增部分开始
  int upvalueCount;
  // 新增部分结束
  Chunk chunk;
```

> We're conscientious C programmers, so we zero-initialize that when an ObjFunction is first allocated.

我们是负责的C程序员，所以当ObjFunction第一次被分配时，我们将其初始化为0。

object.c，在newFunction()方法中添加代码：

```
  function->arity = 0;
  // 新增部分开始
  function->upvalueCount = 0;
  // 新增部分结束
  function->name = NULL;
```

> In the compiler, we add a field for the upvalue array.

在编译器中，我们添加一个字段来存储上值数组。

compiler.c，在结构体Compiler中添加代码：

```
  int localCount;
  // 新增部分开始
  Upvalue upvalues[UINT8_COUNT];
  // 新增部分结束
  int scopeDepth;
```

> For simplicity, I gave it a fixed size. The OP_GET_UPVALUE and OP_SET_UPVALUE instructions encode an upvalue index using a single byte operand, so there's a restriction on how many upvalues a function can have—how many unique variables it can close over. Given that, we can afford a static array that large. We also need to make sure the compiler doesn't overflow that limit.

为了简单起见，我给了它一个固定的大小。OP_GET_UPVALUE和OP_SET_UPVALUE指令使用一个单字节操作数来编码上值索引，所以一个函数可以有多少个上值（可以封闭多少个不同的变量）是有限制的。鉴于此，我们可以负担得起这么大的静态数组。我们还需要确保编译器不会超出这个限制。

compiler.c，在addUpvalue()方法中添加代码：

```
      if (upvalue->index == index && upvalue->isLocal == isLocal) {
        return i;
      }
    }
    // 新增部分开始
    if (upvalueCount == UINT8_COUNT) {
      error("Too many closure variables in function.");
      return 0;
    }
    // 新增部分结束
    compiler->upvalues[upvalueCount].isLocal = isLocal;
```

> Finally, the Upvalue struct type itself.

最后，是Upvalue结构体本身。

compiler.c，在结构体Local后添加代码：

```
typedef struct {
  uint8_t index;
  bool isLocal;
} Upvalue;
```

> The `index` field stores which local slot the upvalue is capturing. The `isLocal` field deserves its own section, which we'll get to next.

`index`字段存储了上值捕获的是哪个局部变量槽。`isLocal`字段值得有自己的章节，我们接下来会讲到。

> 25.2.2 Flattening upvalues

25.2.2 扁平化上值

> In the example I showed before, the closure is accessing a variable declared in the immediately enclosing function. Lox also supports accessing local variables declared in *any* enclosing scope, as in:

在我之前展示的例子中，闭包访问的是在紧邻的外围函数中声明的变量。Lox还支持访问在*任何*外围作用域中声明的局部变量，如：

```
fun outer() {
  var x = 1;
  fun middle() {
    fun inner() {
      print x;
    }
  }
}
```

> Here, we're accessing `x` in `inner()`. That variable is defined not in `middle()`, but all the way out in `outer()`. We need to handle cases like this too. You *might* think that this isn't much harder since the variable will simply be somewhere farther down on the stack. But consider this devious example:

这里，我们在`inner()`中访问`x`。这个变量不是在`middle()`中定义的，而是要一直追溯到`outer()`中。我们也需要处理这样的情况。你*可能*认为这并不难，因为变量只是位于栈中更下面的某个位置。但是考虑一下这个复杂的例子：

> If you work on programming languages long enough, you will develop a finely honed skill at creating bizarre programs like this that are technically valid but likely to trip up an implementation written by someone with a less perverse imagination than you.

如果你在编程语言方面工作的时间足够长，你就会开发出一种精细的技能，能够创造出像这样的怪异程序，这些程序在技术上是有效的，但很可能会在一个由想象力没你那么变态的人编写的实现中出错。

```
fun outer() {
  var x = "value";
  fun middle() {
    fun inner() {
      print x;
    }

    print "create inner closure";
    return inner;
  }

  print "return from outer";
  return middle;
}

var mid = outer();
var in = mid();
in();
```

> When you run this, it should print:

当你运行这段代码时，应该打印出来：

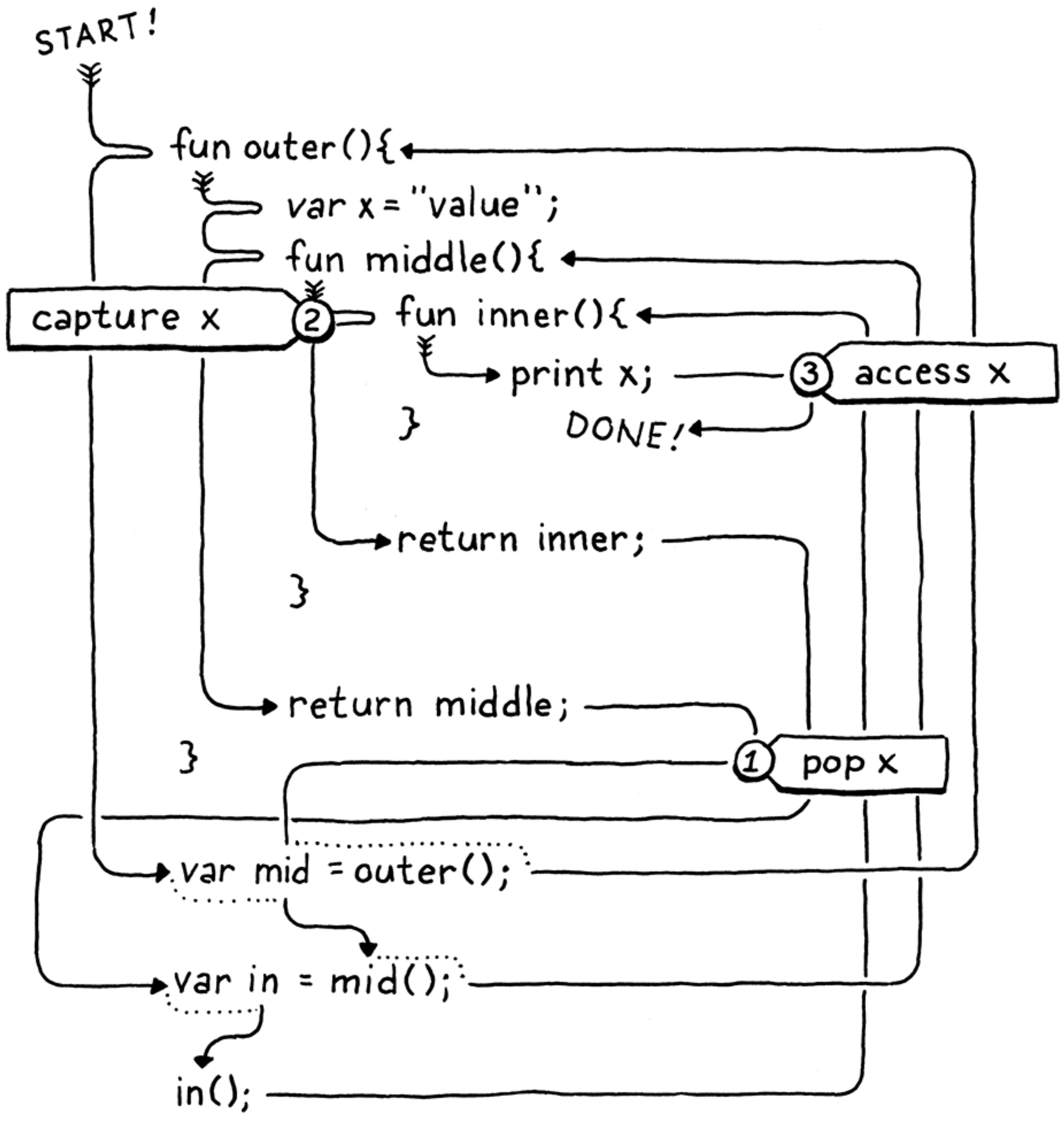
```
return from outer
create inner closure
value
```

> I know, it's convoluted. The important part is that `outer()`—where `x` is declared—returns and pops all of its variables off the stack before the *declaration* of `inner()` executes. So, at the point in time that we create the closure for `inner()`, `x` is already off the stack.

我知道，这很复杂。重要的是，在`inner()`的声明执行之前，`outer()`（`x`被声明的地方）已经返回并弹出其所有变量。因此，在我们为`inner()`创建闭包时，`x`已经离开了堆栈。

> Here, I traced out the execution flow for you:

下面，我为你绘制了执行流程：



> See how x is popped before it is captured and then later accessed ? We really have two problems:

看到了吗，x在被捕获②之前，先被弹出 ①，随后又被访问③？我们确实有两个问题：

> 1. We need to resolve local variables that are declared in surrounding functions beyond the immediately enclosing one.
> 2. We need to be able to capture variables that have already left the stack.

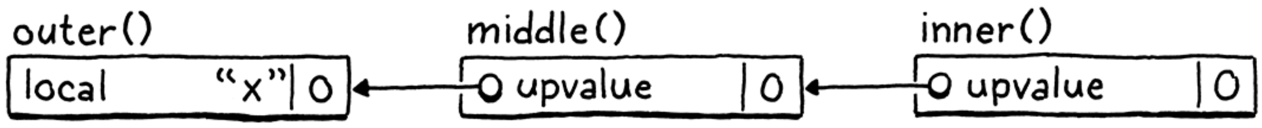
1. 我们需要解析在紧邻的函数之外的外围函数中声明的局部变量。
2. 我们需要能够捕获已经离开堆栈的变量。

> Fortunately, we're in the middle of adding upvalues to the VM, and upvalues are explicitly designed for tracking variables that have escaped the stack. So, in a clever bit of self-reference, we can use upvalues to allow upvalues to capture variables declared outside of the immediately surrounding function.

幸运的是，我们正在向虚拟机中添加上值，而上值是明确为跟踪已退出栈的变量而设计的。因此，通过一个巧妙的自我引用，我们可以使用上值来允许上值捕获紧邻函数之外声明的变量。

> The solution is to allow a closure to capture either a local variable or *an existing upvalue* in the immediately enclosing function. If a deeply nested function references a local variable declared several hops away, we'll thread it through all of the intermediate functions by having each function capture an upvalue for the next function to grab.

解决方案是允许闭包捕获局部变量或紧邻函数中*已有的上值*。如果一个深度嵌套的函数引用了几跳之外声明的局部变量，我们让每个函数捕获一个上值，供下一个函数抓取，从而穿透所有的中间函数。



> In the above example, `middle()` captures the local variable `x` in the immediately enclosing function `outer()` and stores it in its own upvalue. It does this even though `middle()` itself doesn't reference `x`. Then, when the declaration of `inner()` executes, its closure grabs the *upvalue* from the ObjClosure for `middle()` that captured `x`. A function captures—either a local or upvalue—*only* from the immediately surrounding function, which is guaranteed to still be around at the point that the inner function declaration executes.

在上面的例子中，`middle()`捕获了紧邻的外层函数`outer()`中的局部变量`x`，并将其存储在自己的上值中。即使`middle()`本身不引用`x`，它也会这样做。然后，当`inner()`的声明执行时，它的闭包会从已捕获`x`的`middle()`对应的ObjClosure中抓取*上值*。函数只会从紧邻的外层函数中捕获局部变量或上值，因为这些值在内部函数声明执行时仍然能够确保存在。

> In order to implement this, `resolveUpvalue()` becomes recursive.

为了实现这一点，`resolveUpvalue()`变成递归的。

compiler.c，在resolveUpvalue()方法中添加代码：

```
  if (local != -1) {
    return addUpvalue(compiler, (uint8_t)local, true);
  }
  // 新增部分开始
  int upvalue = resolveUpvalue(compiler->enclosing, name);
  if (upvalue != -1) {
    return addUpvalue(compiler, (uint8_t)upvalue, false);
  }
  // 新增部分结束
  return -1;
```

> It's only another three lines of code, but I found this function really challenging to get right the first time. This in spite of the fact that I wasn't inventing anything new, just porting the concept over from Lua. Most recursive functions either do all their work before the recursive call (a **pre-order traversal**, or "on the way down"), or they do all the work after the recursive call (a **post-order traversal**, or "on the way back up"). This function does both. The recursive call is right in the middle.

这只是另外加了三行代码，但我发现这个函数真的很难一次就正确完成。尽管我并没有发明什么新东西，只是从Lua中移植了这个概念。大多数递归函数要么在递归调用之前完成所有工作（**先序遍历**，或"下行"），要么在

递归调用之后完成所有工作（**后续遍历**，或“回退”）。这个函数两者都是，递归调用就在中间。

> We'll walk through it slowly. First, we look for a matching local variable in the enclosing function. If we find one, we capture that local and return. That's the base case.

我们来慢慢看一下。首先，我们在外部函数中查找匹配的局部变量。如果我们找到了，就捕获该局部变量并返回。这就是基本情况^9。

> Otherwise, we look for a local variable beyond the immediately enclosing function. We do that by recursively calling `resolveUpvalue()` on the *enclosing* compiler, not the current one. This series of `resolveUpvalue()` calls works its way along the chain of nested compilers until it hits one of the base cases—either it finds an actual local variable to capture or it runs out of compilers.

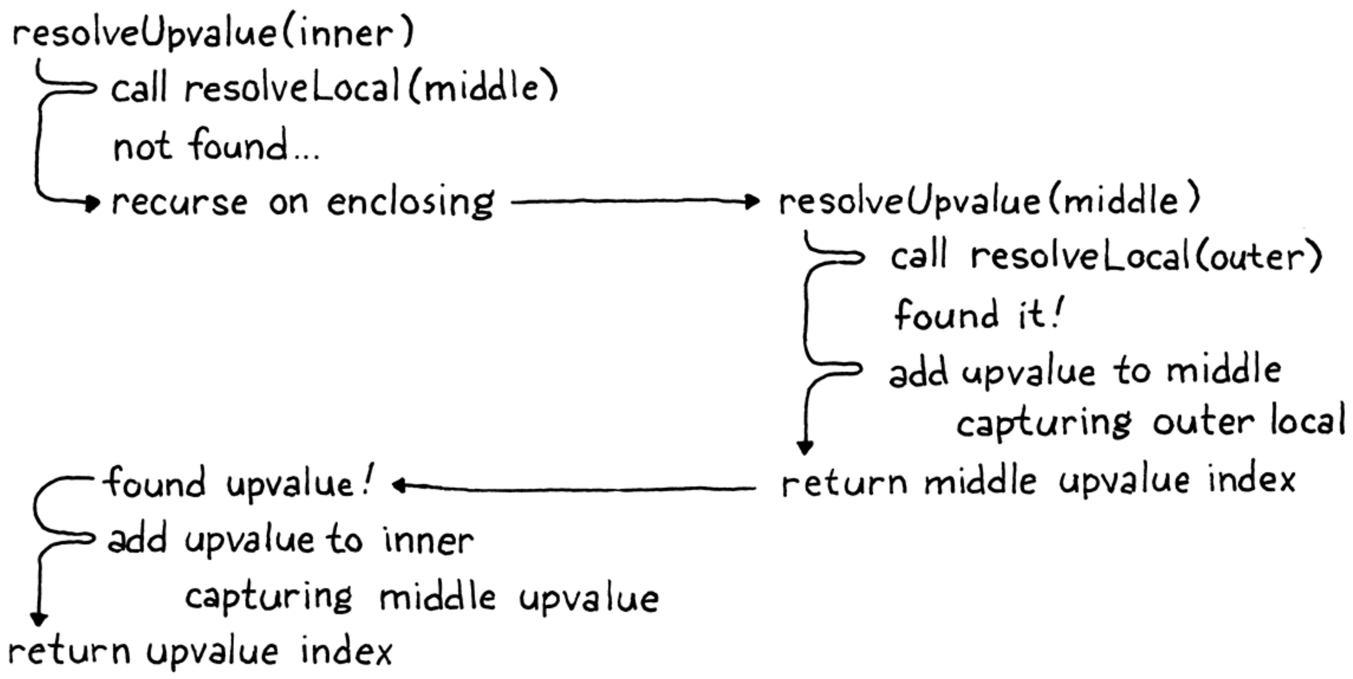
否则，我们会在紧邻的函数之外寻找局部变量。我们通过递归地对外层编译器（而不是当前编译器）调用`resolveUpvalue()`来实现这一点。这一系列的`resolveUpvalue()`调用沿着嵌套的编译器链运行，直到遇见基本情况——要么找到一个事件的局部变量来捕获，要么是遍历完了所有编译器。

> When a local variable is found, the most deeply nested call to `resolveUpvalue()` captures it and returns the upvalue index. That returns to the next call for the inner function declaration. That call captures the *upvalue* from the surrounding function, and so on. As each nested call to `resolveUpvalue()` returns, we drill back down into the innermost function declaration where the identifier we are resolving appears. At each step along the way, we add an upvalue to the intervening function and pass the resulting upvalue index down to the next call.

当找到局部变量时，嵌套最深的`resolveUpvalue()`调用会捕获它并返回上值的索引。这就会返回到内层函数声明对应的下一级调用。该调用会捕获外层函数中的*上值*，以此类推。随着对`resolveUpvalue()`的每个嵌套调用的返回，我们会往下钻到最内层函数声明，即我们正在解析的标识符出现的地方。在这一过程中的每一步，我们都向中间函数添加一个上值，并将得到的上值索引向下传递给下一个调用^10。

> It might help to walk through the original example when resolving `x`:

在解析`x`的时候，走一遍原始的例子可能会有帮助：



> Note that the new call to `addUpvalue()` passes `false` for the `isLocal` parameter. Now you see that that flag controls whether the closure captures a local variable or an upvalue from the surrounding function.

请注意，对`addUpvalue()`的新调用为`isLocal`参数传递了`false`。现在你可以看到，该标志控制着闭包捕获的是局部变量还是来自外围函数的上值。

> By the time the compiler reaches the end of a function declaration, every variable reference has been resolved as either a local, an upvalue, or a global. Each upvalue may in turn capture a local variable from the surrounding function, or an upvalue in the case of transitive closures. We finally have enough data to emit bytecode which creates a closure at runtime that captures all of the correct variables.

当编译器到达函数声明的结尾时，每个变量的引用都已经被解析为局部变量、上值或全局变量。每个上值可以依次从外围函数中捕获一个局部变量，或者在传递闭包的情况下捕获一个上值。我们终于有了足够的数据来生成字节码，该字节码在运行时创建一个捕获所有正确变量的闭包。

compiler.c，在function()方法中添加代码：

```
  emitBytes(OP_CLOSURE, makeConstant(OBJ_VAL(function)));
  // 新增部分开始
  for (int i = 0; i < function->upvalueCount; i++) {
    emitByte(compiler.upvalues[i].isLocal ? 1 : 0);
    emitByte(compiler.upvalues[i].index);
  }
  // 新增部分结束
}
```

> The `OP_CLOSURE` instruction is unique in that it has a variably sized encoding. For each upvalue the closure captures, there are two single-byte operands. Each pair of operands specifies what that upvalue captures. If the first byte is one, it captures a local variable in the enclosing function. If zero, it captures one of the function's upvalues. The next byte is the local slot or upvalue index to capture.

`OP_CLOSURE`指令的独特之处在于，它是不定长编码的。对于闭包捕获的每个上值，都有两个单字节的操作数。每一对操作数都指定了上值捕获的内容。如果第一个字节是1，它捕获的就是外层函数中的一个局部变量。如果是0，它捕获的是函数的一个上值。下一个字节是要捕获局部变量插槽或上值索引。

> This odd encoding means we need some bespoke support in the disassembly code for `OP_CLOSURE`.

这种奇怪的编码意味着我们需要在反汇编程序中对`OP_CLOSURE`提供一些定制化的支持。

debug.c，在disassembleInstruction()方法中添加代码：

```
      printf("\n");
      // 新增部分开始
      ObjFunction* function = AS_FUNCTION(
          chunk->constants.values[constant]);
      for (int j = 0; j < function->upvalueCount; j++) {
        int isLocal = chunk->code[offset++];
        int index = chunk->code[offset++];
        printf("%04d      |                     %s %d\n",
               offset - 2, isLocal ? "local" : "upvalue", index);
      }
      // 新增部分结束
      return offset;
```

> For example, take this script:

举例来说，请看这个脚本：

```
fun outer() {
  var a = 1;
  var b = 2;
  fun middle() {
    var c = 3;
    var d = 4;
    fun inner() {
      print a + c + b + d;
    }
  }
}
```

> If we disassemble the instruction that creates the closure for `inner()`, it prints this:

如果我们反汇编为`inner()`创建闭包的指令，它会打印如下内容：

```
0004    9 OP_CLOSURE          2 <fn inner>
0006      |                     upvalue 0
0008      |                     local 1
0010      |                     upvalue 1
0012      |                     local 2
```

> We have two other, simpler instructions to add disassembler support for.

我们还有两条更简单的指令需要添加反汇编支持。

debug.c，在disassembleInstruction()方法中添加代码：

```
    case OP_SET_GLOBAL:
      return constantInstruction("OP_SET_GLOBAL", chunk, offset);
    // 新增部分开始
    case OP_GET_UPVALUE:
      return byteInstruction("OP_GET_UPVALUE", chunk, offset);
    case OP_SET_UPVALUE:
      return byteInstruction("OP_SET_UPVALUE", chunk, offset);
    // 新增部分结束
    case OP_EQUAL:
```

> These both have a single-byte operand, so there's nothing exciting going on. We do need to add an include so the debug module can get to `AS_FUNCTION()`.

这两条指令都是单字节操作数，所有没有什么有趣的内容。我们确实需要添加一个头文件引入，以便调试模块能够访问`AS_FUNCTION()`。

debug.c，添加代码：

```
#include "debug.h"
// 新增部分开始
#include "object.h"
// 新增部分结束
#include "value.h"
```

> With that, our compiler is where we want it. For each function declaration, it outputs an `OP_CLOSURE` instruction followed by a series of operand byte pairs for each upvalue it needs to capture at runtime. It's time to hop over to that side of the VM and get things running.

有了这些，我们的编译器就达到了我们想要的效果。对于每个函数声明，它都会输出一条`OP_CLOSURE`指令，后跟一系列操作数字节对，对应需要在运行时捕获的每个上值。现在是时候跳到虚拟机那边，让整个程序运转起来。

25.3 Upvalue Objects

25.3 Upvalue对象

> Each `OP_CLOSURE` instruction is now followed by the series of bytes that specify the upvalues the ObjClosure should own. Before we process those operands, we need a runtime representation for upvalues.

现在每条`OP_CLOSURE`指令后面都跟着一系列字节，这些字节指定了ObjClosure应该拥有的上值。在处理这些操作数之前，我们需要一个上值的运行时表示。

object.h，在结构体ObjString后添加代码：

```
typedef struct ObjUpvalue {
  Obj obj;
  Value* location;
} ObjUpvalue;
```

> We know upvalues must manage closed-over variables that no longer live on the stack, which implies some amount of dynamic allocation. The easiest way to do that in our VM is by building on the object system we already have. That way, when we implement a garbage collector in the next chapter, the GC can manage memory for upvalues too.

我们知道上值必须管理已关闭的变量，这些变量不再存活于栈上，这意味着需要一些动态分配。在我们的虚拟机中，最简单的方法就是在已有的对象系统上进行构建。这样，当我们在下一章中实现垃圾收集器时，GC也可

以管理上值的内存。

> Thus, our runtime upvalue structure is an ObjUpvalue with the typical Obj header field. Following that is a `location` field that points to the closed-over variable. Note that this is a *pointer* to a Value, not a Value itself. It's a reference to a *variable*, not a *value*. This is important because it means that when we assign to the variable the upvalue captures, we're assigning to the actual variable, not a copy. For example:

因此，我们的运行时上值结构是一个具有典型Obj头字段的ObjUpvalue。之后是一个指向关闭变量的`location`字段。注意，这是一个指向Value的指针，而不是Value本身。它是一个*变量*的引用，而不是一个*值*。这一点很重要，因为它意味着当我们向上值捕获的变量赋值时，我们是在给实际的变量赋值，而不是对一个副本赋值。举例来说：

```
fun outer() {
  var x = "before";
  fun inner() {
    x = "assigned";
  }
  inner();
  print x;
}
outer();
```

> This program should print "assigned" even though the closure assigns to `x` and the surrounding function accesses it.

这个程序应该打印"assigned"，尽管是在闭包中对`x`赋值，而在外围函数中访问它。

> Because upvalues are objects, we've got all the usual object machinery, starting with a constructor-like function:

因为上值是对象，我们已经有了所有常见的对象机制，首先是类似构造器的函数：

object.h，在copyString()方法后添加代码：

```
ObjString* copyString(const char* chars, int length);
// 新增部分开始
ObjUpvalue* newUpvalue(Value* slot);
// 新增部分结束
void printObject(Value value);
```

> It takes the address of the slot where the closed-over variable lives. Here is the implementation:

它接受的是封闭变量所在的槽的地址。下面是其实现：

object.c，在copyString()方法后添加代码：

```
ObjUpvalue* newUpvalue(Value* slot) {
  ObjUpvalue* upvalue = ALLOCATE_OBJ(ObjUpvalue, OBJ_UPVALUE);
  upvalue->location = slot;
  return upvalue;
}
```

> We simply initialize the object and store the pointer. That requires a new object type.

我们简单地初始化对象并存储指针。这需要一个新的对象类型。

object.h，在枚举ObjType中添加代码：

```
  OBJ_STRING,
  // 新增部分开始
  OBJ_UPVALUE
  // 新增部分结束
} ObjType;
```

> And on the back side, a destructor-like function:

在后面，还有一个类似析构函数的方法：

memory.c，在freeObject()方法中添加代码：

```
      FREE(ObjString, object);
      break;
    }
    // 新增部分开始
    case OBJ_UPVALUE:
      FREE(ObjUpvalue, object);
      break;
    // 新增部分结束
  }
```

> Multiple closures can close over the same variable, so ObjUpvalue does not own the variable it references. Thus, the only thing to free is the ObjUpvalue itself.

多个闭包可以关闭同一个变量，所以ObjUpvalue并不拥有它引用的变量。因此，唯一需要释放的就是ObjUpvalue本身。

> And, finally, to print:

最后，是打印：

object.c，在printObject()方法中添加代码：

```
    case OBJ_STRING:
      printf("%s", AS_CSTRING(value));
      break;
    // 新增部分开始
    case OBJ_UPVALUE:
      printf("upvalue");
      break;
    // 新增部分结束
  }
```

> Printing isn't useful to end users. Upvalues are objects only so that we can take advantage of the VM's memory management. They aren't first-class values that a Lox user can directly access in a program. So this code will never actually execute . . . but it keeps the compiler from yelling at us about an unhandled switch case, so here we are.

打印对终端用户没有用。上值是对象，只是为了让我们能够利用虚拟机的内存管理。它们并不是Lox用户可以在程序中直接访问的一等公民。因此，这段代码实际上永远不会执行……但它使得编译器不会因为未处理的case分支而对我们大喊大叫，所以我们这样做了。

> ## 25.3.1 Upvalues in closures

25.3.1 闭包中的上值

> When I first introduced upvalues, I said each closure has an array of them. We've finally worked our way back to implementing that.

我在第一次介绍上值时，说过每个闭包中都有一个上值数组。我们终于回到了实现它的道路上。

object.h，在结构体ObjClosure中添加代码：

```
  ObjFunction* function;
  // 新增部分开始
  ObjUpvalue** upvalues;
  int upvalueCount;
  // 新增部分结束
} ObjClosure;
```

> Different closures may have different numbers of upvalues, so we need a dynamic array. The upvalues themselves are dynamically allocated too, so we end up with a double pointer—a pointer to a dynamically allocated array of pointers to upvalues. We also store the number of elements in the array.

不同的闭包可能会有不同数量的上值，所以我们需要一个动态数组。上值本身也是动态分配的，因此我们最终需要一个二级指针——一个指向动态分配的上值指针数组的指针。我们还会存储数组中的元素数量^11。

> When we create an ObjClosure, we allocate an upvalue array of the proper size, which we determined at compile time and stored in the ObjFunction.

当我们创建ObjClosure时，会分配一个适当大小的上值数组，这个大小在编译时就已经确定并存储在ObjFunction中。

object.c，在newClosure()方法中添加代码：

```
ObjClosure* newClosure(ObjFunction* function) {
  // 新增部分开始
  ObjUpvalue** upvalues = ALLOCATE(ObjUpvalue*,
                                   function->upvalueCount);
  for (int i = 0; i < function->upvalueCount; i++) {
    upvalues[i] = NULL;
  }
  // 新增部分结束
  ObjClosure* closure = ALLOCATE_OBJ(ObjClosure, OBJ_CLOSURE);
```

> Before creating the closure object itself, we allocate the array of upvalues and initialize them all to `NULL`. This weird ceremony around memory is a careful dance to please the (forthcoming) garbage collection deities. It ensures the memory manager never sees uninitialized memory.

在创建闭包对象本身之前，我们分配了上值数组，并将其初始化为`NULL`。这种围绕内存的奇怪仪式是一场精心的舞蹈，为了取悦（即将到来的）垃圾收集器神灵。它可以确保内存管理器永远不会看到未初始化的内存。

> Then we store the array in the new closure, as well as copy the count over from the ObjFunction.

然后，我们将数组存储在新的闭包中，并将计数值从ObjFunction中复制过来。

object.c，在newClosure()方法中添加代码：

```
  closure->function = function;
  // 新增部分开始
  closure->upvalues = upvalues;
  closure->upvalueCount = function->upvalueCount;
  // 新增部分结束
  return closure;
```

> When we free an ObjClosure, we also free the upvalue array.

当我们释放ObjClosure时，也需要释放上值数组。

memory.c，在freeObject()方法中添加代码：

```
    case OBJ_CLOSURE: {
      // 新增部分开始
      ObjClosure* closure = (ObjClosure*)object;
      FREE_ARRAY(ObjUpvalue*, closure->upvalues,
                 closure->upvalueCount);
      // 新增部分结束
      FREE(ObjClosure, object);
```

> ObjClosure does not own the ObjUpvalue objects themselves, but it does own *the array* containing pointers to those upvalues.

ObjClosure并不拥有ObjUpvalue本身，但它确实拥有包含指向这些上值的指针的数组。

> We fill the upvalue array over in the interpreter when it creates a closure. This is where we walk through all of the operands after `OP_CLOSURE` to see what kind of upvalue each slot captures.

当解释器创建闭包时，我们会填充上值数组。在这里，我们会遍历`OP_CLOSURE`之后的所有操作数，以查看每个槽捕获了什么样的上值。

vm.c，在run()方法中添加代码：

```
        push(OBJ_VAL(closure));
        // 新增部分开始
        for (int i = 0; i < closure->upvalueCount; i++) {
          uint8_t isLocal = READ_BYTE();
          uint8_t index = READ_BYTE();
          if (isLocal) {
            closure->upvalues[i] =
                captureUpvalue(frame->slots + index);
          } else {
            closure->upvalues[i] = frame->closure->upvalues[index];
          }
        }
        // 新增部分结束
        break;
```

> This code is the magic moment when a closure comes to life. We iterate over each upvalue the closure expects. For each one, we read a pair of operand bytes. If the upvalue closes over a local variable in the enclosing function, we let `captureUpvalue()` do the work.

这段代码是闭包诞生的神奇时刻。我们遍历了闭包所期望的每个上值。对于每个上值，我们读取一对操作数字节。如果上值在外层函数的一个局部变量上关闭，我们就让`captureUpvalue()`完成这项工作。

> Otherwise, we capture an upvalue from the surrounding function. An `OP_CLOSURE` instruction is emitted at the end of a function declaration. At the moment that we are executing that declaration, the *current* function is the surrounding one. That means the current function's closure is stored in the CallFrame at the top of the callstack. So, to grab an upvalue from the enclosing function, we can read it right from the `frame` local variable, which caches a reference to that CallFrame.

否则，我们从外围函数中捕获一个上值。`OP_CLOSURE`指令是在函数声明的末尾生成。在我们执行该声明时，*当前*函数就是外围的函数。这意味着当前函数的闭包存储在调用栈顶部的CallFrame中。因此，要从外层函数中抓取上值，我们可以直接从局部变量`frame`中读取，该变量缓存了一个对CallFrame的引用。

> Closing over a local variable is more interesting. Most of the work happens in a separate function, but first we calculate the argument to pass to it. We need to grab a pointer to the captured local's slot in the surrounding function's stack window. That window begins at `frame->slots`, which points to slot zero. Adding `index` offsets that to the local slot we want to capture. We pass that pointer here:

关闭局部变量更有趣。大部分工作发生在一个单独的函数中，但首先我们要计算传递给它的参数。我们需要在外围函数的栈窗口中抓取一个指向捕获的局部变量槽的指针。该窗口起点在`frame->slots`，指向槽0。在其上添加`index`偏移量，以指向我们想要捕获的局部变量槽。我们将该指针传入这里：

vm.c，在callValue()方法后添加代码：

```
static ObjUpvalue* captureUpvalue(Value* local) {
  ObjUpvalue* createdUpvalue = newUpvalue(local);
  return createdUpvalue;
}
```

> This seems a little silly. All it does is create a new ObjUpvalue that captures the given stack slot and returns it. Did we need a separate function for this? Well, no, not *yet*. But you know we are going to end up sticking more code in here.

这看起来有点傻。它所做的就是创建一个新的捕获给定栈槽的ObjUpvalue，并将其返回。我们需要为此建一个单独的函数吗？嗯，不，*现在还*不用。但你懂的，我们最终会在这里插入更多代码。

> First, let's wrap up what we're working on. Back in the interpreter code for handling `OP_CLOSURE`, we eventually finish iterating through the upvalue array and initialize each one. When that completes, we have a new closure with an array full of upvalues pointing to variables.

首先，来总结一下我们的工作。回到处理`OP_CLOSURE`的解释器代码中，我们最终完成了对上值数组的迭代，并初始化了每个值。完成后，我们就有了一个新的闭包，它的数组中充满了指向变量的上值。

> With that in hand, we can implement the instructions that work with those upvalues.

有了这个，我们就可以实现与这些上值相关的指令。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_GET_UPVALUE: {
        uint8_t slot = READ_BYTE();
        push(*frame->closure->upvalues[slot]->location);
        break;
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> The operand is the index into the current function's upvalue array. So we simply look up the corresponding upvalue and dereference its location pointer to read the value in that slot. Setting a variable is similar.

操作数是当前函数的上值数组的索引。因此，我们只需查找相应的上值，并对其位置指针解引用，以读取该槽中的值。设置变量也是如此。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_SET_UPVALUE: {
        uint8_t slot = READ_BYTE();
        *frame->closure->upvalues[slot]->location = peek(0);
        break;
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> We take the value on top of the stack and store it into the slot pointed to by the chosen upvalue. Just as with the instructions for local variables, it's important that these instructions are fast. User programs are constantly reading and writing variables, so if that's slow, everything is slow. And, as usual, the way we make them fast is by keeping them simple. These two new instructions are pretty good: no control flow, no complex arithmetic, just a couple of pointer indirections and a `push()`.

我们取栈顶的值，并将其存储的选中的上值所指向的槽中。就像局部变量的指令一样，这些指令的速度很重要。用户程序在不断的读写变量，因此如果这个操作很慢，一切都会很慢。而且，像往常一样，我们让它变快的方法就是保持简单。这两条新指令非常好：没有控制流，没有复杂的算术，只有几个指针间接引用和一个`push()`^12。

> This is a milestone. As long as all of the variables remain on the stack, we have working closures. Try this:

这是一个里程碑。只要所有的变量都留存在栈上，闭包就可以工作。试试这个：

```
fun outer() {
  var x = "outside";
  fun inner() {
    print x;
  }
  inner();
}
outer();
```

> Run this, and it correctly prints "outside".

运行这个，它就会正确地打印"outside"。

> ## 25.4 Closed Upvalues

25.4 关闭的上值

> Of course, a key feature of closures is that they hold on to the variable as long as needed, even after the function that declares the variable has returned. Here's another example that *should* work:

当然，闭包的一个关键特性是，只要有需要，它们就会一直保留这个变量，即使声明变量的函数已经返回。下面是另一个应该*有效*的例子：

```
fun outer() {
  var x = "outside";
  fun inner() {
    print x;
  }

  return inner;
}

var closure = outer();
closure();
```

> But if you run it right now . . . who knows what it does? At runtime, it will end up reading from a stack slot that no longer contains the closed-over variable. Like I've mentioned a few times, the crux of the issue is that variables in closures don't have stack semantics. That means we've got to hoist them off the stack when the function where they were declared returns. This final section of the chapter does that.

但是如果你现在运行它......天知道它会做什么？在运行时，他会从不包含关闭变量的栈槽中读取数据。正如我多次提到的，问题的关键在于闭包中的变量不具有栈语义。这意味着当声明它们的函数返回时，我们必须将它们从栈中取出。本章的最后一节就是实现这一点的。

> ## 25.4.1 Values and variables

25.4.1 值与变量

> Before we get to writing code, I want to dig into an important semantic point. Does a closure close over a *value* or a *variable?* This isn't purely an academic question. I'm not just splitting hairs. Consider:

在我们开始编写代码之前，我想深入探讨一个重要的语义问题。闭包关闭的是一个*值*还是一个*变量*？这并不是一个纯粹的学术问题^13。我并不是在胡搅蛮缠。考虑一下：

```
var globalSet;
var globalGet;

fun main() {
  var a = "initial";

  fun set() { a = "updated"; }
  fun get() { print a; }

  globalSet = set;
  globalGet = get;
}

main();
```

```
  globalSet();
  globalGet();
```

> The outer `main()` function creates two closures and stores them in global variables so that they outlive the execution of `main()` itself. Both of those closures capture the same variable. The first closure assigns a new value to it and the second closure reads the variable.

外层的`main()`方法创建了两个闭包，并将它们存储在全局变量中，这样它们的存活时间就比`main()`本身的执行时间更长。这两个闭包都捕获了相同的变量。第一个闭包为其赋值，第二个闭包则读取该变量的值^14。

> What does the call to `globalGet()` print? If closures capture *values* then each closure gets its own copy of `a` with the value that `a` had at the point in time that the closure's function declaration executed. The call to `globalSet()` will modify `set()`'s copy of `a`, but `get()`'s copy will be unaffected. Thus, the call to `globalGet()` will print "initial".

调用`globalGet()`会打印什么？如果闭包捕获的是*值*，那么每个闭包都会获得自己的`a`副本，该副本的值为`a`在执行闭包函数声明的时间点上的值。对`globalSet()`的调用会修改`set()`中的`a`副本，但是`get()`中的副本不受影响。因此，对`globalGet()`的调用会打印"initial"。

> If closures close over variables, then `get()` and `set()` will both capture—reference—the *same mutable variable*. When `set()` changes `a`, it changes the same `a` that `get()` reads from. There is only one `a`. That, in turn, implies the call to `globalGet()` will print "updated".

如果闭包关闭的是变量，那么`get()`和`set()`都会捕获（引用）*同一个可变变量*。当`set()`修改`a`时，它改变的是`get()`所读取的那个`a`。这里只有一个`a`。这意味着对`globalGet()`的调用会打印"updated"。

> Which is it? The answer for Lox and most other languages I know with closures is the latter. Closures capture variables. You can think of them as capturing *the place the value lives*. This is important to keep in mind as we deal with closed-over variables that are no longer on the stack. When a variable moves to the heap, we need to ensure that all closures capturing that variable retain a reference to its *one* new location. That way, when the variable is mutated, all closures see the change.

到底是哪一个呢？对于Lox和我所知的其它大多数带闭包的语言来说，答案是后者。闭包捕获的是变量。你可以把它们看作是对*值所在位置*的捕获。当我们处理不再留存于栈上的闭包变量时，这一点很重要，要牢牢记住。当一个变量移动到堆中时，我们需要确保所有捕获该变量的闭包都保留对其新位置的引用。这样一来，当变量发生变化时，所有闭包都能看到这个变化。

> ## 25.4.2 Closing upvalues

25.4.2 关闭上值

> We know that local variables always start out on the stack. This is faster, and lets our single-pass compiler emit code before it discovers the variable has been captured. We also know that closed-over variables need to move to the heap if the closure outlives the function where the captured variable is declared.

我们知道，局部变量总是从堆栈开始。这样做更快，并且可以让我们的单遍编译器在发现变量被捕获之前先生成字节码。我们还知道，如果闭包的存活时间超过声明被捕获变量的函数，那么封闭的变量就需要移动到堆中。

> Following Lua, we'll use **open upvalue** to refer to an upvalue that points to a local variable still on the stack. When a variable moves to the heap, we are *closing* the upvalue and the result is, naturally, a **closed upvalue**. The two questions we need to answer are:

跟随Lua，我们会使用**开放上值**来表示一个指向仍在栈中的局部变量的上值。当变量移动到堆中时，我们就*关闭*上值，而结果自然就是一个**关闭的上值**。我们需要回答两个问题：

> 1. Where on the heap does the closed-over variable go?
> 2. When do we close the upvalue?

1. 被关闭的变量放在堆中的什么位置？
2. 我们什么时候关闭上值？

> The answer to the first question is easy. We already have a convenient object on the heap that represents a reference to a variable—ObjUpvalue itself. The closed-over variable will move into a new field right inside the ObjUpvalue struct. That way we don't need to do any additional heap allocation to close an upvalue.

第一个问题的答案很简单。我们在堆上已经有了一个便利的对象，它代表了对某个变量（ObjUpvalue本身）的引用。被关闭的变量将移动到ObjUpvalue结构体中的一个新字段中。这样一来，我们不需要做任何额外的堆分配来关闭上值。

> The second question is straightforward too. As long as the variable is on the stack, there may be code that refers to it there, and that code must work correctly. So the logical time to hoist the variable to the heap is as late as possible. If we move the local variable right when it goes out of scope, we are certain that no code after that point will try to access it from the stack. After the variable is out of scope, the compiler will have reported an error if any code tried to use it.

第二个问题也很直截了当。只要变量在栈中，就可能存在引用它的代码，而且这些代码必须能够正确工作。因此，将变量提取到堆上的逻辑时间越晚越好。如果我们在局部变量超出作用域时将其移出，我们可以肯定，在那之后没有任何代码会试图从栈中访问它。在变量超出作用域之后^15，如果有任何代码试图访问它，编译器就会报告一个错误。

> The compiler already emits an `OP_POP` instruction when a local variable goes out of scope. If a variable is captured by a closure, we will instead emit a different instruction to hoist that variable out of the stack and into its corresponding upvalue. To do that, the compiler needs to know which locals are closed over.

当局部变量超出作用域时，编译器已经生成了`OP_POP`指令^16。如果变量被某个闭包捕获，我们会发出一条不同的指令，将该变量从栈中提取到其对应的上值。为此，编译器需要知道哪些局部变量被关闭了。

> The compiler already maintains an array of Upvalue structs for each local variable in the function to track exactly that state. That array is good for answering "Which variables does this closure use?" But it's poorly suited for answering, "Does *any* function capture this local variable?" In particular, once the Compiler for some closure has finished, the Compiler for the enclosing function whose variable has been captured no longer has access to any of the upvalue state.

编译器已经为函数中的每个局部变量维护了一个Upvalue结构体的数组，以便准确地跟踪该状态。这个数组很好地回答了"这个闭包使用了哪个变量"，但他不适合回答"是否有*任何*函数捕获了这个局部变量？"特别是，一旦某个闭包的Compiler 执行完成，变量被捕获的外层函数的Compiler就不能再访问任何上值状态了。

> In other words, the compiler maintains pointers from upvalues to the locals they capture, but not in the other direction. So we first need to add some extra tracking inside the existing Local struct so that we can tell if a given local is captured by a closure.

换句话说，编译器保持着从上值指向它们捕获的局部变量的指针，而没有相反方向的指针。所以，我们首先需要在现有的Local结构体中添加额外的跟踪信息，这样我们就能够判断某个给定的局部变量是否被某个闭包捕获。

compiler.c，在Local结构体中添加代码：

```
  int depth;
  // 新增部分开始
  bool isCaptured;
  // 新增部分结束
} Local;
```

> This field is `true` if the local is captured by any later nested function declaration. Initially, all locals are not captured.

如果局部变量被后面嵌套的任何函数声明捕获，字段则为`true`。最初，所有的局部数据都没有被捕获。

compiler.c，在addLocal()方法中添加代码：

```
  local->depth = -1;
  // 新增部分开始
  local->isCaptured = false;
  // 新增部分结束
}
```

> Likewise, the special “slot zero local” that the compiler implicitly declares is not captured.

同样地，编译器隐式声明的特殊的“槽0中的局部变量”不会被捕获^17。

compiler.c，在initCompiler()方法中添加代码：

```
  local->depth = 0;
  // 新增部分开始
  local->isCaptured = false;
  // 新增部分结束
  local->name.start = "";
```

> When resolving an identifier, if we end up creating an upvalue for a local variable, we mark it as captured.

在解析标识符时，如果我们最终为某个局部变量创建了一个上值，我们将其标记为已捕获。

compiler.c，在resolveUpvalue()方法中添加代码：

```
  if (local != -1) {
    // 新增部分开始
    compiler->enclosing->locals[local].isCaptured = true;
    // 新增部分结束
    return addUpvalue(compiler, (uint8_t)local, true);
```

> Now, at the end of a block scope when the compiler emits code to free the stack slots for the locals, we can tell which ones need to get hoisted onto the heap. We'll use a new instruction for that.

现在，在块作用域的末尾，当编译器生成字节码来释放局部变量的栈槽时，我们可以判断哪些数据需要被提取到堆中。我们将使用一个新指令来实现这一点。

compiler.c，在endScope()方法中，替换1行：

```
  while (current->localCount > 0 &&
         current->locals[current->localCount - 1].depth >
            current->scopeDepth) {
    // 新增部分开始
    if (current->locals[current->localCount - 1].isCaptured) {
      emitByte(OP_CLOSE_UPVALUE);
    } else {
      emitByte(OP_POP);
    }
    // 新增部分结束
    current->localCount--;
  }
```

> The instruction requires no operand. We know that the variable will always be right on top of the stack at the point that this instruction executes. We declare the instruction.

这个指令不需要操作数。我们知道，在该指令执行时，变量一定在栈顶。我们来声明这条指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CLOSURE,
  // 新增部分开始
  OP_CLOSE_UPVALUE,
  // 新增部分结束
  OP_RETURN,
```

> And add trivial disassembler support for it:

并为它添加简单的反汇编支持：

debug.c，在disassembleInstruction()方法中添加代码：

```
    }
    // 新增部分开始
    case OP_CLOSE_UPVALUE:
      return simpleInstruction("OP_CLOSE_UPVALUE", offset);
    // 新增部分结束
    case OP_RETURN:
```

> Excellent. Now the generated bytecode tells the runtime exactly when each captured local variable must move to the heap. Better, it does so only for the locals that *are* used by a closure and need this special treatment. This aligns with our general performance goal that we want users to pay only for functionality that they use. Variables that aren't used by closures live and die entirely on the stack just as they did before.

太好了。现在，生成的字节码准确地告诉运行时，每个被捕获的局部变量必须移动到堆中的确切时间。更好的是，它只对被闭包使用并需要这种特殊处理的局部变量才会这样做。这与我们的总体性能目标是一致的，即我们希望用户只为他们使用的功能付费。那些不被闭包使用的变量只会出现于栈中，就像以前一样。

> 25.4.3 Tracking open upvalues

25.4.3 跟踪开放的上值

> Let's move over to the runtime side. Before we can interpret `OP_CLOSE_UPVALUE` instructions, we have an issue to resolve. Earlier, when I talked about whether closures capture variables or values, I said it was important that if multiple closures access the same variable that they end up with a reference to the exact same storage location in memory. That way if one closure writes to the variable, the other closure sees the change.

让我们转到运行时方面。在解释`OP_CLOSE_UPVALUE`指令之前，我们还有一个问题需要解决。之前，在谈到闭包捕获的是变量还是值时，我说过，如果多个闭包访问同一个变量，它们最终将引用内存中完全相同的存储位置，这一点很重要。这样一来，如果某个闭包对变量进行写入，另一个闭包就会看到这一变化。

> Right now, if two closures capture the same local variable, the VM creates a separate Upvalue for each one. The necessary sharing is missing. When we move the variable off the stack, if we move it into only one of the upvalues, the other upvalue will have an orphaned value.

现在，如果两个闭包捕获同一个局部变量，虚拟机就会为每个闭包创建一个单独的Upvalue。必要的共享是缺失的^18。当我们把变量移出堆栈时，如果我们只是将它移入其中一个上值中，其它上值就会有一个孤儿值。

> To fix that, whenever the VM needs an upvalue that captures a particular local variable slot, we will first search for an existing upvalue pointing to that slot. If found, we reuse that. The challenge is that all of the previously created upvalues are squirreled away inside the upvalue arrays of the various closures. Those closures could be anywhere in the VM's memory.

为了解决这个问题，每当虚拟机需要一个捕获特定局部变量槽的上值时，我们会首先搜索指向该槽的现有上值。如果找到了，我们就重用它。难点在于，之前创建的所有上值都存储在各个闭包的上值数组中。这些闭包可能位于虚拟机内存中的任何位置。

> The first step is to give the VM its own list of all open upvalues that point to variables still on the stack. Searching a list each time the VM needs an upvalue sounds like it might be slow, but in practice, it's

> not bad. The number of variables on the stack that actually get closed over tends to be small. And function declarations that create closures are rarely on performance critical execution paths in the user's program.

第一步是给虚拟机提供它自己的所有开放上值的列表，这些上值指向仍在栈中的变量。每次虚拟机需要一个上值时，都要搜索列表，这听起来似乎很慢，但是实际上，这并没有那么坏。栈中真正被关闭的变量的数量往往很少。而且创建闭包的函数声明很少出现在用户程序中的性能关键执行路径上^19。

> Even better, we can order the list of open upvalues by the stack slot index they point to. The common case is that a slot has *not* already been captured—sharing variables between closures is uncommon—and closures tend to capture locals near the top of the stack. If we store the open upvalue array in stack slot order, as soon as we step past the slot where the local we're capturing lives, we know it won't be found. When that local is near the top of the stack, we can exit the loop pretty early.

更妙的是，我们可以根据开放上值所指向的栈槽索引对列表进行排序。常见的情况是，某个栈槽还*没有*被捕获（在闭包之间共享变量是不常见的），而闭包倾向于捕获靠近栈顶的局部变量。如果我们按照栈槽的顺序存储开放上值数组，一旦我们越过正在捕获的局部变量所在的槽，我们就知道它不会被找到。当这个局部变量在栈顶时，我们可以很早就退出循环。

> Maintaining a sorted list requires inserting elements in the middle efficiently. That suggests using a linked list instead of a dynamic array. Since we defined the ObjUpvalue struct ourselves, the easiest implementation is an intrusive list that puts the next pointer right inside the ObjUpvalue struct itself.

维护有序列表需要能高效地在中间插入元素。这一点建议我们使用链表而不是动态数组。因为我们自己定义了ObjUpvalue结构体，最简单的实现是一个插入式列表，将指向下一元素的指针放在ObjUpvalue结构体本身中。

object.h，在结构体ObjUpvalue中添加代码：

```
  Value* location;
  // 新增部分开始
  struct ObjUpvalue* next;
  // 新增部分结束
} ObjUpvalue;
```

> When we allocate an upvalue, it is not attached to any list yet so the link is `NULL`.

当我们分配一个上值时，它还没有附加到任何列表，因此链接是`NULL`。

object.c，在newUpvalue()方法中添加代码：

```
  upvalue->location = slot;
  // 新增部分开始
  upvalue->next = NULL;
  // 新增部分结束
  return upvalue;
```

> The VM owns the list, so the head pointer goes right inside the main VM struct.

VM拥有该列表，因此头指针放在VM主结构体中。

vm.h，在结构体VM中添加代码：

```
  Table strings;
  // 新增部分开始
  ObjUpvalue* openUpvalues;
  // 新增部分结束
  Obj* objects;
```

> The list starts out empty.

列表在开始时为空。

vm.c，在resetStack()方法中添加代码：

```
  vm.frameCount = 0;
  // 新增部分开始
  vm.openUpvalues = NULL;
  // 新增部分结束
}
```

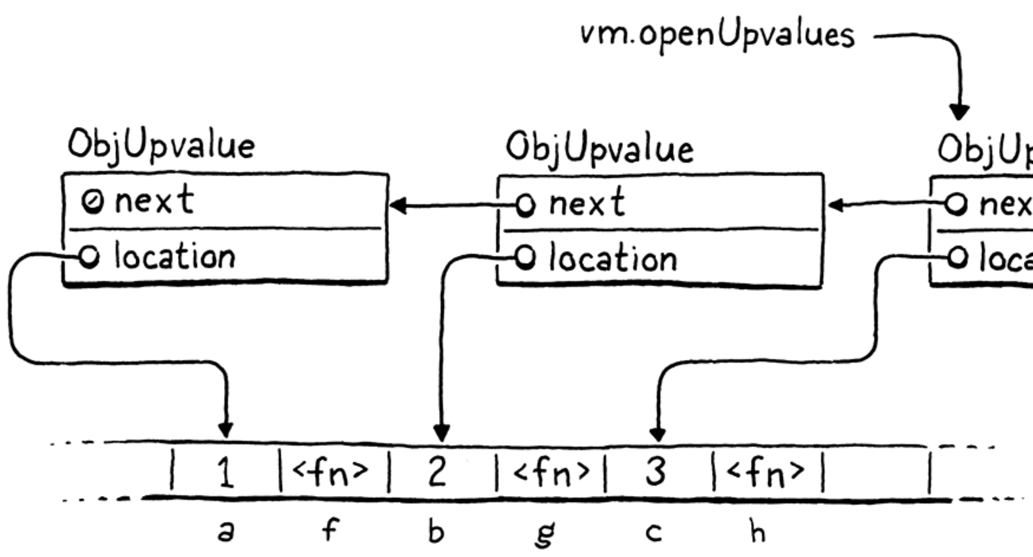
> Starting with the first upvalue pointed to by the VM, each open upvalue points to the next open upvalue that references a local variable farther down the stack. This script, for example,

从VM指向的第一个上值开始，每个开放上值都指向下一个引用了栈中靠下位置的局部变量的开放上值。以这个脚本为例

```
{
  var a = 1;
  fun f() {
    print a;
  }
  var b = 2;
  fun g() {
    print b;
  }
  var c = 3;
  fun h() {
    print c;
  }
}
```

> should produce a series of linked upvalues like so:

它应该产生如下所示的一系列链接的上值：



> Whenever we close over a local variable, before creating a new upvalue, we look for an existing one in the list.

每当关闭一个局部变量时，在创建新的上值之前，先在该列表中查找现有的上值。

vm.c，在captureUpvalue()方法中添加代码：

```
static ObjUpvalue* captureUpvalue(Value* local) {
  // 新增部分开始
  ObjUpvalue* prevUpvalue = NULL;
  ObjUpvalue* upvalue = vm.openUpvalues;
  while (upvalue != NULL && upvalue->location > local) {
    prevUpvalue = upvalue;
    upvalue = upvalue->next;
  }

  if (upvalue != NULL && upvalue->location == local) {
    return upvalue;
  }
  // 新增部分结束
  ObjUpvalue* createdUpvalue = newUpvalue(local);
```

> We start at the head of the list, which is the upvalue closest to the top of the stack. We walk through the list, using a little pointer comparison to iterate past every upvalue pointing to slots above the one we're looking for. While we do that, we keep track of the preceding upvalue on the list. We'll need to update that node's `next` pointer if we end up inserting a node after it.

我们从列表的头部开始，它是最接近栈顶的上值。我们遍历列表，使用一个小小的指针比较，对每一个指向的槽位高于当前查找的位置的上值进行迭代[^20]。当我们这样做时，我们要跟踪列表中前面的上值。如果我们在某个节点后面插入了一个节点，就需要更新该节点的`next`指针。

> There are three reasons we can exit the loop:

我们有三个原因可以退出循环：

1. > **The local slot we stopped at *is* the slot we're looking for.** We found an existing upvalue capturing the variable, so we reuse that upvalue.

   我们停止时的局部变量槽是我们要找的槽。我在找到了一个现有的上值捕获了这个变量，因此我们重用这个上值。

2. > **We ran out of upvalues to search.** When `upvalue` is `NULL`, it means every open upvalue in the list points to locals above the slot we're looking for, or (more likely) the upvalue list is empty. Either way, we didn't find an upvalue for our slot.

   我们找不到需要搜索的上值了。当`upvalue`为`NULL`时，这意味着列表中每个开放上值都指向位于我们要找的槽之上的局部变量，或者（更可能是）上值列表是空的。无论怎样，我们都没有找到对应该槽的上值。

3. > **We found an upvalue whose local slot is *below* the one we're looking for.** Since the list is sorted, that means we've gone past the slot we are closing over, and thus there must not be an existing upvalue for it.

   我们找到了一个上值，其局部变量槽低于我们正查找的槽位。因为列表是有序的，这意味着我们已经超过了正在关闭的槽，因此肯定没有对应该槽的已有上值。

> In the first case, we're done and we've returned. Otherwise, we create a new upvalue for our local slot and insert it into the list at the right location.

在第一种情况下，我们已经完成并且返回了。其它情况下，我们为局部变量槽创建一个新的上值，并将其插入到列表中的正确位置。

vm.c，在captureUpvalue()方法中添加代码：

```
  ObjUpvalue* createdUpvalue = newUpvalue(local);
  // 新增部分开始
  createdUpvalue->next = upvalue;

  if (prevUpvalue == NULL) {
    vm.openUpvalues = createdUpvalue;
  } else {
    prevUpvalue->next = createdUpvalue;
  }
  // 新增部分结束
  return createdUpvalue;
```

> The current incarnation of this function already creates the upvalue, so we only need to add code to insert the upvalue into the list. We exited the list traversal by either going past the end of the list, or by stopping on the first upvalue whose stack slot is below the one we're looking for. In either case, that means we need to insert the new upvalue *before* the object pointed at by `upvalue` (which may be `NULL` if we hit the end of the list).

这个函数的当前版本已经创建了上值，我们只需要添加代码将上值插入到列表中。我们退出列表遍历的原因，要么是到达了列表末尾，要么是停在了第一个栈槽低于待查找槽位的上值。无论哪种情况，这都意味着我们需要在`upvalue`指向的对象（如果到达列表的末尾，则该对象可能是`NULL`）之前插入新的上值。

> As you may have learned in Data Structures 101, to insert a node into a linked list, you set the `next` pointer of the previous node to point to your new one. We have been conveniently keeping track of that preceding node as we walked the list. We also need to handle the special case where we are inserting a new upvalue at the head of the list, in which case the “next” pointer is the VM’s head pointer.

正如你在《数据结构101》中所学到的，要将一个节点插入到链表中，你需要将前一个节点的`next`指针指向新的节点。当我们遍历列表时，我们一直很方便地跟踪着前面的节点。我们还需要处理一种特殊情况，即我们在列表头部插入一个新的上值，在这种情况下，“next”指针是VM的头指针[^21]。

> With this updated function, the VM now ensures that there is only ever a single ObjUpvalue for any given local slot. If two closures capture the same variable, they will get the same upvalue. We’re ready to move those upvalues off the stack now.

有了这个升级版函数，VM现在可以确保每个指定的局部变量槽都只有一个ObjUpvalue。如果两个闭包捕获了相同的变量，它们会得到相同的上值。现在，我们准备将这些上值从栈中移出。

> ### 25.4.4 Closing upvalues at runtime

25.4.4 在运行时关闭上值

> The compiler helpfully emits an `OP_CLOSE_UPVALUE` instruction to tell the VM exactly when a local variable should be hoisted onto the heap. Executing that instruction is the interpreter’s responsibility.

编译器会生成一个有用的`OP_CLOSE_UPVALUE`指令，以准确地告知VM何时将局部变量提取到堆中。执行该指令是解释器的责任。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_CLOSE_UPVALUE:
        closeUpvalues(vm.stackTop - 1);
        pop();
        break;
      // 新增部分结束
      case OP_RETURN: {
```

> When we reach the instruction, the variable we are hoisting is right on top of the stack. We call a helper function, passing the address of that stack slot. That function is responsible for closing the upvalue and moving the local from the stack to the heap. After that, the VM is free to discard the stack slot, which it does by calling `pop()`.

当我们到达该指令时，我们要提取的变量就在栈顶。我们调用一个辅助函数，传入栈槽的地址。该函数负责关闭上值，并将局部变量从栈中移动到堆上。之后，VM就可以自由地丢弃栈槽，这是通过调用`pop()`实现的。

> The fun stuff happens here:

有趣的事情发生在这里：

vm.c，在captureUpvalue()方法后添加代码：

```
static void closeUpvalues(Value* last) {
  while (vm.openUpvalues != NULL &&
         vm.openUpvalues->location >= last) {
    ObjUpvalue* upvalue = vm.openUpvalues;
    upvalue->closed = *upvalue->location;
    upvalue->location = &upvalue->closed;
    vm.openUpvalues = upvalue->next;
  }
}
```

> This function takes a pointer to a stack slot. It closes every open upvalue it can find that points to that slot or any slot above it on the stack. Right now, we pass a pointer only to the top slot on the stack, so the “or above it” part doesn’t come into play, but it will soon.

这个函数接受一个指向栈槽的指针。它会关闭它能找到的指向该槽或栈上任何位于该槽上方的所有开放上值。现在，我们只传递了一个指向栈顶的指针，所以“或其上方”的部分没有发挥作用，但它很快就会起作用了。

> To do this, we walk the VM’s list of open upvalues, again from top to bottom. If an upvalue’s location points into the range of slots we’re closing, we close the upvalue. Otherwise, once we reach an upvalue outside of the range, we know the rest will be too, so we stop iterating.

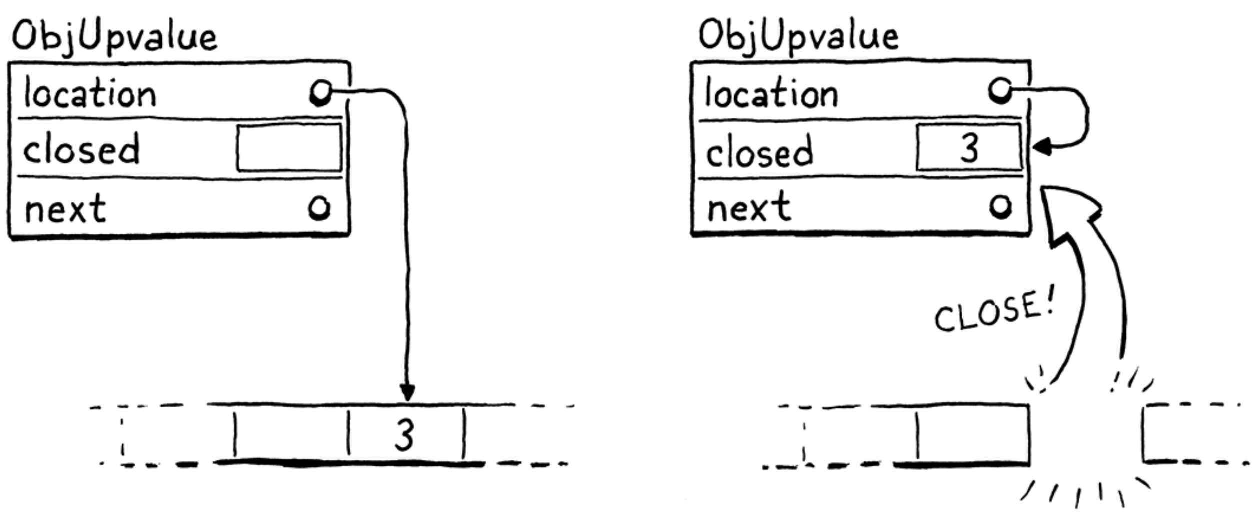
为此，我们再次从上到下遍历VM的开放上值列表。如果某个上值的位置指向我们要关闭的槽位范围，则关闭该上值。否则，一旦我们遇到范围之外的上值，我们知道其它上值也在范围之外，所以我们停止迭代。

> The way an upvalue gets closed is pretty cool. First, we copy the variable’s value into the `closed` field in the ObjUpvalue. That’s where closed-over variables live on the heap. The `OP_GET_UPVALUE` and `OP_SET_UPVALUE` instructions need to look for the variable there after it’s been moved. We could add some conditional logic in the interpreter code for those instructions to check some flag for whether the upvalue is open or closed.

关闭上值的方式非常酷[^22]。首先，我们将变量的值复制到ObjUpvalue的`closed`字段。这就是被关闭的变量在堆中的位置。在变量被移动之后，`OP_GET_UPVALUE`和`OP_SET_UPVALUE`指令需要在那里查找它。我们可以在解释器代码中为这些指令添加一些条件逻辑，检查一些标志，以确定上值是开放的还是关闭的。

> But there is already a level of indirection in play—those instructions dereference the `location` pointer to get to the variable’s value. When the variable moves from the stack to the `closed` field, we simply update that `location` to the address of the ObjUpvalue’s *own* `closed` field.

但是已经有一个中间层在起作用了——这些指令对`location`指针解引用以获取变量的值。当变量从栈移动到`closed`字段时，我们只需将`location`更新为ObjUpvalue*自己的*`closed`字段。



> We don't need to change how OP_GET_UPVALUE and OP_SET_UPVALUE are interpreted at all. That keeps them simple, which in turn keeps them fast. We do need to add the new field to ObjUpvalue, though.

我们根本不需要改变OP_GET_UPVALUE和OP_SET_UPVALUE的解释方式。这使得它们保持简单，反过来又使它们保持快速。不过，我们确实需要向ObjUpvalue添加新的字段。

object.h，在结构体ObjUpvalue中添加代码：

```
Value* location;
// 新增部分开始
Value closed;
// 新增部分结束
struct ObjUpvalue* next;
```

> And we should zero it out when we create an ObjUpvalue so there's no uninitialized memory floating around.

当我们创建一个ObjUpvalue时，应该将其置为0，这样就不会有未初始化的内存了。

object.c，在newUpvalue()方法中添加代码：

```
ObjUpvalue* upvalue = ALLOCATE_OBJ(ObjUpvalue, OBJ_UPVALUE);
// 新增部分开始
upvalue->closed = NIL_VAL;
// 新增部分结束
upvalue->location = slot;
```

> Whenever the compiler reaches the end of a block, it discards all local variables in that block and emits an OP_CLOSE_UPVALUE for each local variable that was closed over. The compiler does *not* emit any instructions at the end of the outermost block scope that defines a function body. That scope contains the function's parameters and any locals declared immediately inside the function. Those need to get closed too.

每当编译器到达一个块的末尾时，它就会丢弃该代码块中的所有局部变量，并为每个关闭的局部变量生成一个`OP_CLOSE_UPVALUE`指令。编译器*不会*在定义某个函数主体的最外层块作用域的末尾生成任何指令[^23]。这个作用域包含函数的形参和函数内部声明的任何局部变量。这些也需要被关闭。

> This is the reason `closeUpvalues()` accepts a pointer to a stack slot. When a function returns, we call that same helper and pass in the first stack slot owned by the function.

这就是`closeUpvalues()`接受一个指向栈槽的指针的原因。当函数返回时，我们调用相同的辅助函数，并传入函数拥有的第一个栈槽。

vm.c，在run()方法中添加代码：

```
        Value result = pop();
        // 新增部分开始
        closeUpvalues(frame->slots);
        // 新增部分结束
        vm.frameCount--;
```

> By passing the first slot in the function's stack window, we close every remaining open upvalue owned by the returning function. And with that, we now have a fully functioning closure implementation. Closed-over variables live as long as they are needed by the functions that capture them.

通过传递函数栈窗口中的第一个槽，我们关闭了正在返回的函数所拥有的所有剩余的开放上值。有了这些，我们现在就有了一个功能齐全的闭包实现。只要捕获变量的函数需要，被关闭的变量就一直存在。

> This was a lot of work! In jlox, closures fell out naturally from our environment representation. In clox, we had to add a lot of code—new bytecode instructions, more data structures in the compiler, and new runtime objects. The VM very much treats variables in closures as different from other variables.

这是一项艰巨的工作！在jlox中，闭包很自然地从我们的环境表示形式中分离出来。在clox中，我们必须添加大量的代码——新的字节码指令、编译器中的更多数据结构和新的运行时对象。VM在很大程度上将闭包中的变量与其它变量进行区别对待。

> There is a rationale for that. In terms of implementation complexity, jlox gave us closures "for free". But in terms of *performance*, jlox's closures are anything but. By allocating *all* environments on the heap, jlox pays a significant performance price for *all* local variables, even the majority which are never captured by closures.

这是有道理的。就实现复杂性而言，jlox"免费"为我们提供了闭包。但是就*性能*而言，jlox的闭包完全不是这样。由于在堆上分配*所有*环境，jlox为*所有*局部变量付出了显著的性能代价，甚至是未被闭包捕获的大部分变量。

> With clox, we have a more complex system, but that allows us to tailor the implementation to fit the two use patterns we observe for local variables. For most variables which do have stack semantics, we allocate them entirely on the stack which is simple and fast. Then, for the few local variables where that doesn't work, we have a second slower path we can opt in to as needed.

在clox中，我们有一个更复杂的系统，但这允许我们对实现进行调整以适应我们观察到的局部变量的两种使用模式。对于大多数具有堆栈语义的变量，我们完全可用在栈中分配，这既简单又快速。然后，对于少数不适用

的局部变量，我们可以根据需要选择第二条较慢的路径。

> Fortunately, users don't perceive the complexity. From their perspective, local variables in Lox are simple and uniform. The *language itself* is as simple as jlox's implementation. But under the hood, clox is watching what the user does and optimizing for their specific uses. As your language implementations grow in sophistication, you'll find yourself doing this more. A large fraction of "optimization" is about adding special case code that detects certain uses and provides a custom-built, faster path for code that fits that pattern.

幸运的是，用户并不会察觉到这种复杂性。在他们看来，Lox中的局部变量简单而统一。语言本身就像jlox一样简单。但在内部，clox会观察用户的行为，并针对他们的具体用途进行优化。随着你的语言实现越来越复杂，你会发现自己要做的事情越来越多。"优化"的很大一部分是关于添加特殊情况的代码，以检测特定的使用，并为符合该模式的代码提供定制化的、更快速的路径。

> We have lexical scoping fully working in clox now, which is a major milestone. And, now that we have functions and variables with complex lifetimes, we also have a *lot* of objects floating around in clox's heap, with a web of pointers stringing them together. The next step is figuring out how to manage that memory so that we can free some of those objects when they're no longer needed.

我们现在已经在clox中完全实现了词法作用域，这是一个重要的里程碑。而且，现在我们有了具有复杂生命周期的函数和变量，我们也要了很多漂浮在clox堆中的对象，并有一个指针网络将它们串联起来。下一步是弄清楚如何管理这些内存，以便我们可以在不再需要这些对象的时候释放它们。

^2: 搜索"闭包转换 closure conversion"和"Lambda提升 lambda lifting"就可以开始探索了。 ^3: 换句话说，Lox中的函数声明是一种字面量——定义某个内置类型的常量值的一段语法。 ^4: Lua实现中将包含字节码的原始函数对象称为"原型"，这个一个很好的形容词，只不过这个词也被重载以指代原型继承。^5: 或许我应该定义一个宏，以便更容易地生成这些宏。也许这有点太玄了。 ^6: 这段代码看起来有点傻，因为我们仍然把原始的ObjFunction压入栈中，然后在创建完闭包之后弹出它，然后再将闭包压入栈。为什么要把ObjFunction放在这里呢？像往常一样，当你看到奇怪的堆栈操作发生时，它是为了让即将到来的垃圾回收器知道一些堆分配的对象。 ^7: 它最终可能会是一个完全未定义的变量，甚至不是全局变量。但是在Lox中，我们直到运行时才能检测到这个错误，所以从编译器的角度看，它是"期望是全局的"。^8: 就像常量和函数元数一样，上值计数也是连接编译器与运行时的一些小数据。 ^9: 当然，另一种基本情况是，没有外层函数。在这种情况下，该变量不能在词法上解析，并被当作全局变量处理。 ^10: 每次递归调用`resolveUpvalue()`都会*走出*一层函数嵌套。因此，内部的*递归调用*指向的是*外部*的嵌套声明。查找局部变量的最内层的`resolveUpvalue()`递归调用对应的将是*最外层*的函数，就是实际声明该变量的外层函数的内部。 ^11: 在闭包中存储上值数量是多余的，因为ObjClosure引用的ObjFunction也保存了这个数量。通常，这类奇怪的代码是为了适应GC。在闭包对应的ObjFunction已经被释放后，收集器可能也需要知道ObjClosure对应上值数组的大小。 ^12: 设置指令不会从栈中*弹出*值，因为，请记住，赋值在Lox中是一个表达式。所以赋值的结果（所赋的值）需要保留在栈中，供外围的表达式使用。 ^13: 如果Lox不允许赋值，这就是一个学术问题。 ^14: 我使用了多个全局变量的事实并不重要。我需要某种方式从一个函数中返回两个值。而在Lox中没有任何形式的聚合类型，我的选择很有限。 ^15: 这里 的"之后"，指的是词法或文本意义上的——在包含关闭变量的声明语句的代码块的`}`之后的代码。 ^16: 编译器不会弹出参数和在函数体中声明的局部变量。这些我们也会在运行时处理。 ^17: 在本书的后面部分，用户将有可能捕获这个变量。这里只是建立一些预期。 ^18: 如果某个闭包从外围函数中捕获了一个*上值*，那么虚拟机确实会共享上值。嵌套的情况下，工作正常。但是如果两个同级闭包捕获了同一个局部变量，它们会各自创建一个单独的ObjUpvalue。^19: 闭包经常在热循环中被 *调用*。想想传递给集合的典型高阶函数，如`map()`和`filter()`。这应该是很快的。但是创建闭包的函数声明只发生一次，而且通常是在循环之外。 [^20]: 这是个单链表。除了从头指针开始遍历，我们没有其它选择。 [^21]: 还有一种更简短的实现，通过使用一个指向指针的指针，来统一处理更新头部指针或前一个上值的`next`指针两种情况，但这种代码几乎会让所有未达到指针专业

水平的人感到困惑。我选择了基本的`if`语句的方法。 [^22]: 我并不是在自夸。这都是Lua开发团队的创新。 [^23]: 没有什么*阻止*我们在编译器中关闭最外层的函数作用域，并生成`OP_POP`和`OP_CLOSE_UPVALUE`指令。这样做只是没有必要，因为运行时在弹出调用帧时，隐式地丢弃了函数使用的所有栈槽。

习题

1. > Wrapping every ObjFunction in an ObjClosure introduces a level of indirection that has a performance cost. That cost isn't necessary for functions that do not close over any variables, but it does let the runtime treat all calls uniformly.
   >
   > Change clox to only wrap functions in ObjClosures that need upvalues. How does the code complexity and performance compare to always wrapping functions? Take care to benchmark programs that do and do not use closures. How should you weight the importance of each benchmark? If one gets slower and one faster, how do you decide what trade-off to make to choose an implementation strategy?

   将每个ObjFunction 包装在ObjClosure中，会引入一个有性能代价的中间层。这个代价对于那些没有关闭任何变量的函数来说是不必要的，但它确实让运行时能够统一处理所有的调用。

   将clox改为只用ObjClosure包装需要上值的函数。与包装所有函数相比，代码的复杂性与性能如何？请注意对使用闭包和不使用闭包的程序进行基准测试。你应该如何衡量每个基准的重要性？如果一个变慢了，另一个变快了，你决定通过什么权衡来选择实现策略？

2. > Read the design note below. I'll wait. Now, how do you think Lox *should* behave? Change the implementation to create a new variable for each loop iteration.

   请阅读下面的设计笔记。我在这里等着。现在，你觉得Lox应该怎么做？改变实现方式，为每个循环迭代创建一个新的变量。

3. > A famous koan teaches us that "objects are a poor man's closure" (and vice versa). Our VM doesn't support objects yet, but now that we have closures we can approximate them. Using closures, write a Lox program that models two-dimensional vector "objects". It should:
   >
   > - Define a "constructor" function to create a new vector with the given *x* and *y* coordinates.
   > - Provide "methods" to access the *x* and *y* coordinates of values returned from that constructor.
   > - Define an addition "method" that adds two vectors and produces a third.

   一个著名的公案告诉我们："对象是简化版的闭包"（反之亦然）。我们的虚拟机还不支持对象，但现在我们有了闭包，我们可以近似地使用它们。使用闭包，编写一个Lox程序，建模一个二维矢量"对象"。它应该：

   - 定义一个"构造器"函数，创建一个具有给定x和y坐标的新矢量。
   - 提供"方法"来访问构造函数返回值的x和y坐标。
   - 定义一个相加"方法"，将两个向量相加并产生第三个向量。

设计笔记：关闭循环变量

> Closures capture variables. When two closures capture the same variable, they share a reference to the same underlying storage location. This fact is visible when new values are assigned to the variable. Obviously, if two closures capture *different* variables, there is no sharing.

闭包捕获变量。当两个闭包捕获相同的变量时，它们共享对相同的底层存储位置的引用。当将新值赋给该变量时，这一事实是可见的。显然，如果两个闭包捕获*不同*的变量，就不存在共享。

```
var globalOne;
var globalTwo;

fun main() {
  {
    var a = "one";
    fun one() {
      print a;
    }
    globalOne = one;
  }

  {
    var a = "two";
    fun two() {
      print a;
    }
    globalTwo = two;
  }
}

main();
globalOne();
globalTwo();
```

> This prints “one” then “two”. In this example, it’s pretty clear that the two `a` variables are different. But it’s not always so obvious. Consider:

这里会打印“one”然后是“two”。在这个例子中，很明显两个`a`变量是不同的。但一点这并不总是那么明显。考虑一下：

```
var globalOne;
var globalTwo;

fun main() {
  for (var a = 1; a <= 2; a = a + 1) {
    fun closure() {
      print a;
    }
    if (globalOne == nil) {
      globalOne = closure;
    } else {
      globalTwo = closure;
    }
  }
}

main();
globalOne();
globalTwo();
```

> The code is convoluted because Lox has no collection types. The important part is that the `main()` function does two iterations of a `for` loop. Each time through the loop, it creates a closure that captures the loop variable. It stores the first closure in `globalOne` and the second in `globalTwo`.

这段代码很复杂，因为Lox没有集合类型。重要的部分是，`main()`函数进行了`for`循环的两次迭代。每次循环执行时，它都会创建一个捕获循环变量的闭包。它将第一个闭包存储在`globalOne`中，并将第二个闭包存储在`globalTwo`中。

> There are definitely two different closures. Do they close over two different variables? Is there only one `a` for the entire duration of the loop, or does each iteration get its own distinct `a` variable?

这无疑是两个不同的闭包。它们是在两个不同的变量上闭合的吗？在整个循环过程中只有一个`a`，还是每个迭代都有自己单独的`a`变量？

> The script here is strange and contrived, but this does show up in real code in languages that aren't as minimal as clox. Here's a JavaScript example:

这里的脚本很奇怪，而且是人为设计的，但它确实出现在实际的代码中，而且这些代码使用的语言并不是像clox这样的小语言。下面是一个JavaScript的示例：

```
var closures = [];
for (var i = 1; i <= 2; i++) {
  closures.push(function () { console.log(i); });
}

closures[0]();
closures[1]();
```

> Does this print "1" then "2", or does it print "3" twice? You may be surprised to hear that it prints "3" twice. In this JavaScript program, there is only a single `i` variable whose lifetime includes all iterations of the loop, including the final exit.

这里会打印"1"再打印"2"，还是打印两次"3"？你可能会惊讶地发现，它打印了两次"3"^24。在这个JavaScript程序中，只有一个`i`变量，它的生命周期包括循环的所有迭代，包括最后的退出。

> If you're familiar with JavaScript, you probably know that variables declared using `var` are implicitly *hoisted* to the surrounding function or top-level scope. It's as if you really wrote this:

如果你熟悉JavaScript，你可能知道，使用`var`声明的变量会隐式地被提取到外围函数或顶层作用域中。这就好像你是这样写的：

```
var closures = [];
var i;
for (i = 1; i <= 2; i++) {
  closures.push(function () { console.log(i); });
}

closures[0]();
closures[1]();
```

> At that point, it’s clearer that there is only a single `i`. Now consider if you change the program to use the newer `let` keyword:

此时，很明显只有一个`i`。现在考虑一下，如果你将程序改为使用更新的`let`关键字：

```
var closures = [];
for (let i = 1; i <= 2; i++) {
  closures.push(function () { console.log(i); });
}

closures[0]();
closures[1]();
```

> Does this new program behave the same? Nope. In this case, it prints “1” then “2”. Each closure gets its own `i`. That’s sort of strange when you think about it. The increment clause is `i++`. That looks very much like it is assigning to and mutating an existing variable, not creating a new one.

这个新程序的行为是一样的吗？不是。在本例中，它会打印“1”然后打印“2”。每个闭包都有自己的`i`。仔细想想会觉得有点奇怪，增量子句是`i++`，这看起来很像是对现有变量进行赋值和修改，而不是创建一个新变量。

> Let’s try some other languages. Here’s Python:

让我们试试其它语言。下面是Python：

```
closures = []
for i in range(1, 3):
  closures.append(lambda: print(i))

closures[0]()
closures[1]()
```

> Python doesn’t really have block scope. Variables are implicitly declared and are automatically scoped to the surrounding function. Kind of like hoisting in JS, now that I think about it. So both closures capture the same variable. Unlike C, though, we don’t exit the loop by incrementing `i` *past* the last value, so this prints “2” twice.

Python并没有真正的块作用域。变量是隐式声明的，并自动限定在外围函数的作用域中。现在我想起来，这有点像JS中的“悬挂”。所以两个闭包都捕获了同一个变量。但与C不同的是，我们不会通过增加`i`超过最后一个值来

退出循环，所以这里会打印两次“2”。

> What about Ruby? Ruby has two typical ways to iterate numerically. Here’s the classic imperative style:

那Ruby呢？Ruby有两种典型的数值迭代方式。下面是典型的命令式风格：

```
closures = []
for i in 1..2 do
  closures << lambda { puts i }
end

closures[0].call
closures[1].call
```

> This, like Python, prints “2” twice. But the more idiomatic Ruby style is using a higher-order `each()` method on range objects:

这有点像是Python，会打印两次“2”。但是更惯用的Ruby风格是在范围对象上使用高阶的`each()`方法：

```
closures = []
(1..2).each do |i|
  closures << lambda { puts i }
end

closures[0].call
closures[1].call
```

> If you’re not familiar with Ruby, the `do |i| ... end` part is basically a closure that gets created and passed to the `each()` method. The `|i|` is the parameter signature for the closure. The `each()` method invokes that closure twice, passing in 1 for `i` the first time and 2 the second time.

如果你不熟悉Ruby，`do |i| ... end`部分基本上就是一个闭包，它被创建并传递给`each()`方法。`|i|`是闭包的参数签名。`each()`方法两次调用该闭包，第一次传入1，第二次传入2。

> In this case, the “loop variable” is really a function parameter. And, since each iteration of the loop is a separate invocation of the function, those are definitely separate variables for each call. So this prints “1” then “2”.

在这种情况下，“循环变量”实际上是一个函数参数。而且，由于循环的每次迭代都是对函数的单独调用，所以每次调用都是单独的变量。因此，这里先打印“1”然后打印“2”。

> If a language has a higher-level iterator-based looping structure like `foreach` in C#, Java’s “enhanced for”, `for-of` in JavaScript, `for-in` in Dart, etc., then I think it’s natural to the reader to have each iteration create a new variable. The code *looks* like a new variable because the loop header looks like a variable declaration. And there’s no increment expression that looks like it’s mutating that variable to advance to the next step.

如果一门语言具有基于迭代器的高级循环结果，比如C#中的`foreach`，Java中的“增强型for循环”，JavaScript中的`for-of`，Dart中的`for-in`等等，那我认为读者很自然地会让每次迭代都创建一个新变量。代码*看起来*像一个新变量，是因为循环头看起来像是一个变量声明。看起来没有任何增量表达式通过改变变量以推进到下一步。

> If you dig around StackOverflow and other places, you find evidence that this is what users expect, because they are very surprised when they *don't* get it. In particular, C# originally did *not* create a new loop variable for each iteration of a `foreach` loop. This was such a frequent source of user confusion that they took the very rare step of shipping a breaking change to the language. In C# 5, each iteration creates a fresh variable.

如果你在StackOverflow和其它地方挖掘一下，你会发现这正是用户所期望的，因为当他们*没有*看到这个结果时，他们会非常惊讶。特别是，C#最初并没有为`foreach`循环的每次迭代创建一个新的循环变量。这一点经常引起用户的困惑，所以他们采用了非常罕见的措施，对语言进行了突破性的修改。在C# 5中，每个迭代都会创建一个新的变量。

> Old C-style `for` loops are harder. The increment clause really does look like mutation. That implies there is a single variable that's getting updated each step. But it's almost never *useful* for each iteration to share a loop variable. The only time you can even detect this is when closures capture it. And it's rarely helpful to have a closure that references a variable whose value is whatever value caused you to exit the loop.

旧的C风格的`for`循环更难了。增量子句看起来像是修改。这意味着每一步更新的是同一个变量。但是每个迭代共享一个循环变量几乎是*没有用*的。只有在闭包捕获它时，你才能检测到这一现象。而且，如果闭包引用的变量的值是导致循环退出的值，那么它也几乎没有帮助。

> The pragmatically useful answer is probably to do what JavaScript does with `let` in `for` loops. Make it look like mutation but actually create a new variable each time, because that's what users want. It is kind of weird when you think about it, though.

实用的答案可能是像JavaScript在`for`循环中的`let`那样。让它看起来像修改，但实际上每次都创建一个新变量，因为这是用户想要的。不过，仔细想想，还是有点奇怪的。

26.垃圾回收 Garbage Collection

> I wanna, I wanna, I wanna, I wanna, I wanna be trash.
>
> —— The Whip, “Trash”

【译者注：The Whip乐队的歌曲，歌词也没必要翻译了】

> We say Lox is a “high-level” language because it frees programmers from worrying about details irrelevant to the problem they're solving. The user becomes an executive, giving the machine abstract goals and letting the lowly computer figure out how to get there.

我们说Lox是一种“高级”语言，因为它使得程序员不必担心那些与他们要解决的问题无关的细节。用户变成了执行者，给机器设定抽象的目标，让底层的计算机自己想办法实现目标。

> Dynamic memory allocation is a perfect candidate for automation. It's necessary for a working program, tedious to do by hand, and yet still error-prone. The inevitable mistakes can be catastrophic, leading to crashes, memory corruption, or security violations. It's the kind of risky-yet-boring work that machines excel at over humans.

动态内存分配是自动化的最佳选择。这是一个基本可用的程序所必需的，手动操作很繁琐，而且很容易出错。不可避免的错误可能是灾难性的，会导致崩溃、内存损坏或安全漏洞。机器比人类更擅长这种既有风险又无聊的工作。

> This is why Lox is a **managed language**, which means that the language implementation manages memory allocation and freeing on the user's behalf. When a user performs an operation that requires some dynamic memory, the VM automatically allocates it. The programmer never worries about deallocating anything. The machine ensures any memory the program is using sticks around as long as needed.

这就是为什么Lox是一种**托管语言**，这意味着语言实现会代表用户管理内存的分配与释放。当用户执行某个需要动态内存的操作时，虚拟机会自动分配。程序员不用担心任何释放内存的事情。机器会确保程序使用的任意内存会在需要的时候存在。

> Lox provides the illusion that the computer has an infinite amount of memory. Users can allocate and allocate and allocate and never once think about where all these bytes are coming from. Of course, computers do not yet *have* infinite memory. So the way managed languages maintain this illusion is by going behind the programmer's back and reclaiming memory that the program no longer needs. The component that does this is called a **garbage collector**.

Lox提供了一种计算机拥有无限内存的错觉。用户可以不停地分配、分配、再分配，而不用考虑这些内存是从哪里来的。当然，计算机还没有无限的内存。因此，托管语言维持这种错觉的方式是背着程序员，回收程序不再需要的内存。实现这一点的组件被称为**垃圾回收器**。

26.1 Reachability

26.1 可达性

> This raises a surprisingly difficult question: how does a VM tell what memory is *not* needed? Memory is only needed if it is read in the future, but short of having a time machine, how can an implementation tell what code the program *will* execute and which data it *will* use? Spoiler alert: VMs cannot travel into the future. Instead, the language makes a conservative approximation: it considers a piece of memory to still be in use if it *could possibly* be read in the future.

这就引出了一个非常困难的问题：虚拟机如何分辨哪些内存是不需要的？内存只有在未来被读取时才需要，但是如果没有时间机器，语言如何知道程序将执行哪些代码，使用哪些数据？剧透警告：虚拟机不能穿越到未来。相反，语言做了一个保守的估计^1：如果一块内存在未来*有可能*被读取，就认为它仍然在使用。

> That sounds *too* conservative. Couldn't *any* bit of memory potentially be read? Actually, no, at least not in a memory-safe language like Lox. Here's an example:

这听起来*太*保守了。难道不是内存中的*任何*比特都可能被读取吗？事实上，不是，至少在Lox这样内存安全的语言中不是。下面是一个例子：

```
var a = "first value";
a = "updated";
// GC here.
print a;
```

> Say we run the GC after the assignment has completed on the second line. The string “first value” is still sitting in memory, but there is no way for the user’s program to ever get to it. Once `a` got reassigned, the program lost any reference to that string. We can safely free it. A value is **reachable** if there is some way for a user program to reference it. Otherwise, like the string “first value” here, it is **unreachable**.

假设我们在完成第二行的赋值之后运行GC。字符串“first value”仍然在内存中，但是用户的程序没有办法访访问它。一旦`a`被重新赋值，程序就失去了对该字符串的任何引用，我们可以安全地释放它。如果用户程序可以通过某种方式引用一个值，这个值就是可达的。否则，就像这里的字符串“first value”一样，它是不可达的。

> Many values can be directly accessed by the VM. Take a look at:

许多值可以被虚拟机直接访问。请看：

```
var global = "string";
{
  var local = "another";
  print global + local;
}
```

> Pause the program right after the two strings have been concatenated but before the `print` statement has executed. The VM can reach `"string"` by looking through the global variable table and finding the entry for `global`. It can find `"another"` by walking the value stack and hitting the slot for the local variable `local`. It can even find the concatenated string `"stringanother"` since that temporary value is also sitting on the VM’s stack at the point when we paused our program.

在两个字符串连接之后但是`print`语句执行之前暂停程序。虚拟机可以通过查看全局变量表，并查找`global`条目到达`"string"`。它可以通过遍历值栈，并找到局部变量`local`的栈槽来找到`"another"`。它甚至也可以找到连接后的字符串`"stringanother"`，因为在我们暂停程序的时候，这个临时值也在虚拟机的栈中。

> All of these values are called **roots**. A root is any object that the VM can reach directly without going through a reference in some other object. Most roots are global variables or on the stack, but as we’ll see, there are a couple of other places the VM stores references to objects that it can find.

所有这些值都被称为**根**。根是虚拟机可以无需通过其它对象的引用而直接到达的任何对象。大多数根是全局变量或在栈上，但正如我们将看到的，还有其它一些地方，虚拟机会在其中存储它可以找到的对象的引用。

> Other values can be found by going through a reference inside another value. Fields on instances of classes are the most obvious case, but we don’t have those yet. Even without those, our VM still has indirect references. Consider:

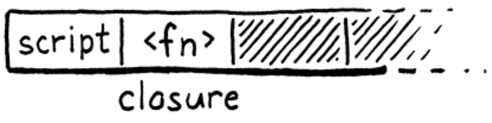
其它值可以通过另一个值中的引用来找到。类实例中的字段是最明显的情况，但我们目前还没有类。即使没有这些，我们的虚拟机中仍然存在间接引用。考虑一下：

```
fun makeClosure() {
  var a = "data";

  fun f() { print a; }
  return f;
}
```

```
{
  var closure = makeClosure();
  // GC here.
  closure();
}
```

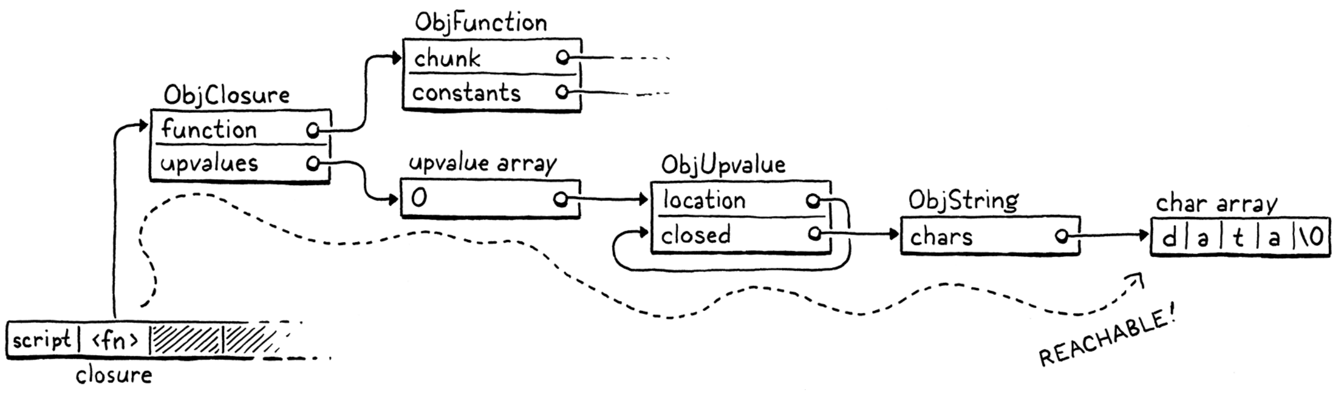
> Say we pause the program on the marked line and run the garbage collector. When the collector is done and the program resumes, it will call the closure, which will in turn print `"data"`. So the collector needs to *not* free that string. But here's what the stack looks like when we pause the program:

假设我们在标记的行上暂停并运行垃圾回收器。当回收器完成、程序恢复时，它将会调用闭包，然后输出`"data"`。所以回收器需要*不释放*那个字符串。但当我们暂停程序时，栈是这样的：



> The `"data"` string is nowhere on it. It has already been hoisted off the stack and moved into the closed upvalue that the closure uses. The closure itself is on the stack. But to get to the string, we need to trace through the closure and its upvalue array. Since it *is* possible for the user's program to do that, all of these indirectly accessible objects are also considered reachable.

`"data"`字符串并不在上面。它已经被从栈中提取出来，并移动到闭包所使用的关闭上值中。闭包本身在栈上。但是要得到字符串，我们需要跟踪闭包及其上值数组。因为用户的程序可能会这样做，所有这些可以间接访问的对象也被认为是可达的。



> This gives us an inductive definition of reachability:

这给了我们一个关于可达性的归纳定义：

> - All roots are reachable.
> - Any object referred to from a reachable object is itself reachable.

- 所有根都是可达的。
- 任何被某个可达对象引用的对象本身是可达的。

> These are the values that are still "live" and need to stay in memory. Any value that *doesn't* meet this definition is fair game for the collector to reap. That recursive pair of rules hints at a recursive algorithm we can use to free up unneeded memory:

这些是仍然“存活”、需要留在内存中的值。任何*不符合*这个定义的值，对于回收器来说都是可收割的猎物。这一对递归规则暗示了我们可以用一个递归算法来释放不需要的内存：

> 1. Starting with the roots, traverse through object references to find the full set of reachable objects.
> 2. Free all objects *not* in that set.

1. 从根开始，遍历对象引用，找到可达对象的完整集合。
2. 释放不在集合中的所有对象。

> Many different garbage collection algorithms are in use today, but they all roughly follow that same structure. Some may interleave the steps or mix them, but the two fundamental operations are there. They mostly differ in *how* they perform each step.

如今有很多不同的垃圾回收算法，但是它们都大致遵循相同的结构。有些算法可能会将这些步骤进行交叉或混合，但这两个基本操作是存在的。不同算法的区别在于*如何*执行每个步骤^2。

> ## 26.2 Mark-Sweep Garbage Collection

26.2 标记-清除垃圾回收

> The first managed language was Lisp, the second “high-level” language to be invented, right after Fortran. John McCarthy considered using manual memory management or reference counting, but eventually settled on (and coined) garbage collection—once the program was out of memory, it would go back and find unused storage it could reclaim.

第一门托管语言是Lisp，它是继Fortran之后发明的第二种“高级”语言。John McCarthy曾考虑使用手动内存管理或引用计数，但最终还是选择（并创造了）垃圾回收——一旦程序的内存用完了，它就会回去寻找可以回收的未使用的存储空间^3。

> He designed the very first, simplest garbage collection algorithm, called **mark-and-sweep** or just **mark-sweep**. Its description fits in three short paragraphs in the initial paper on Lisp. Despite its age and simplicity, the same fundamental algorithm underlies many modern memory managers. Some corners of CS seem to be timeless.

他设计了最早的、最简单的垃圾回收算法，被称为**标记并清除（mark-and-sweep）**，或者就叫**标记清除（mark-sweep）**。在最初的Lisp论文中，关于它的描述只有短短的三段。尽管它年代久远且简单，但许多现代内存管理器都使用了相同的基本算法。CS中的一些角落似乎是永恒的。

> As the name implies, mark-sweep works in two phases:

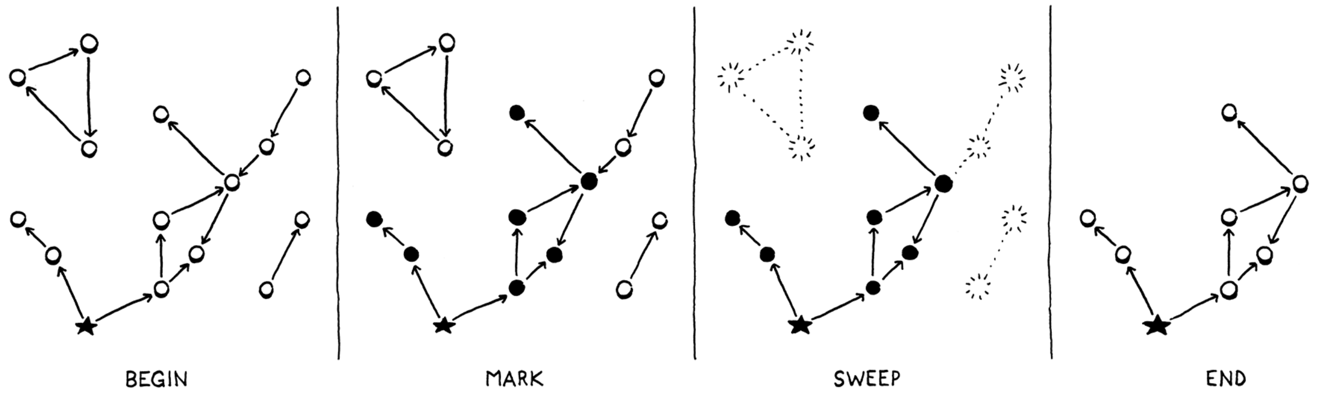
顾名思义，标记-清除分两个阶段工作：

> - **Marking:** We start with the roots and traverse or *trace* through all of the objects those roots refer to. This is a classic graph traversal of all of the reachable objects. Each time we visit an object, we *mark* it in some way. (Implementations differ in how they record the mark.)
> - **Sweeping:** Once the mark phase completes, every reachable object in the heap has been marked. That means any unmarked object is unreachable and ripe for reclamation. We go through all the unmarked objects and free each one.

- **标记**：我们从根开始，遍历或跟踪这些根所引用的所有对象。这是对所有可达对象的经典图式遍历。每次我们访问一个对象时，我们都用某种方式来标记它。（不同的实现方式，记录标记的方法也不同）
- **清除**：一旦标记阶段完成，堆中的每个可达对象都被标记了。这意味着任何未被标记的对象都是不可达的，可以被回收。我们遍历所有未被标记的对象，并逐个释放。

> It looks something like this:

它看起来像是这样的^4：



> That's what we're gonna implement. Whenever we decide it's time to reclaim some bytes, we'll trace everything and mark all the reachable objects, free what didn't get marked, and then resume the user's program.

这就是我们要实现的。每当我们决定是时候回收一些字节的时候，我们就会跟踪一切，并标记所有可达的对象，释放没有被标记的对象，然后恢复用户的程序。

> 26.2.1 Collecting garbage

26.2.1 回收垃圾

> This entire chapter is about implementing this one function:

整个章节都是关于实现这一个函数的^5：

memory.h，在reallocate()方法后添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize);
// 新增部分开始
void collectGarbage();
// 新增部分结束
void freeObjects();
```

> We'll work our way up to a full implementation starting with this empty shell:

我们会从这个空壳开始逐步完整实现：

memory.c，在freeObject()方法后添加代码：

```
void collectGarbage() {
}
```

> The first question you might ask is, When does this function get called? It turns out that's a subtle question that we'll spend some time on later in the chapter. For now we'll sidestep the issue and build ourselves a handy diagnostic tool in the process.

你可能会问的第一个问题是，这个函数在什么时候被调用？事实证明，这是一个微妙的问题，我们会在后面的章节中花些时间讨论。现在，我们先回避这个问题，并在这个过程中为自己构建一个方便的诊断工具。

common.h，添加代码：

```
#define DEBUG_TRACE_EXECUTION
// 新增部分开始
#define DEBUG_STRESS_GC
// 新增部分结束
#define UINT8_COUNT (UINT8_MAX + 1)
```

> We'll add an optional "stress test" mode for the garbage collector. When this flag is defined, the GC runs as often as it possibly can. This is, obviously, horrendous for performance. But it's great for flushing out memory management bugs that occur only when a GC is triggered at just the right moment. If *every* moment triggers a GC, you're likely to find those bugs.

我们将为垃圾回收器添加一个可选的"压力测试"模式。当定义这个标志后，GC就会尽可能频繁地运行。显然，这对性能来说是很糟糕的。但它对于清除内存管理bug很有帮助，这些bug只有在适当的时候触发GC时才会出现。如果每时每刻都触发GC，那你很可能会找到这些bug。

memory.c，在reallocate()方法中添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
  // 新增部分开始
  if (newSize > oldSize) {
#ifdef DEBUG_STRESS_GC
    collectGarbage();
#endif
  }
  // 新增部分结束
  if (newSize == 0) {
```

> Whenever we call `reallocate()` to acquire more memory, we force a collection to run. The if check is because `reallocate()` is also called to free or shrink an allocation. We don't want to trigger a GC for that—in particular because the GC itself will call `reallocate()` to free memory.

每当我们调用`reallocate()`来获取更多内存时，都会强制运行一次回收。这个`if`检查是因为，在释放或收缩分配的内存时也会调用`reallocate()`。我们不希望在这种时候触发GC——特别是因为GC本身也会调用`reallocate()`来释放内存。

> Collecting right before allocation is the classic way to wire a GC into a VM. You're already calling into the memory manager, so it's an easy place to hook in the code. Also, allocation is the only time when you really *need* some freed up memory so that you can reuse it. If you *don't* use allocation to trigger a GC, you have to make sure every possible place in code where you can loop and allocate memory also has a way to trigger the collector. Otherwise, the VM can get into a starved state where it needs more memory but never collects any.

在分配之前进行回收是将GC引入虚拟机的经典方式^6。你已经在调用内存管理器了，所以这是个很容易挂接代码的地方。另外，分配是唯一你真的*需要*一些释放出来的内存的时候，这样你就可以重新使用它。如果*不*使用分配来触发GC，则必须确保代码中每个可以循环和分配内存的地方都有触发回收器的方法。否则，虚拟机会进入饥饿状态，它需要更多的内存，但却没有回收到任何内存。

> ## 26.2.2 Debug logging

26.2.2 调试日志

> While we're on the subject of diagnostics, let's put some more in. A real challenge I've found with garbage collectors is that they are opaque. We've been running lots of Lox programs just fine without any GC *at all* so far. Once we add one, how do we tell if it's doing anything useful? Can we tell only if we write programs that plow through acres of memory? How do we debug that?

既然我们在讨论诊断的问题，那我们再加入一些内容。我发现垃圾回收器的一个真正的挑战在于它们是不透明的。到目前为止，我们已经在没有任何GC的情况下运行了很多Lox程序。一旦我们添加了GC，我们如何知道它是否在做有用的事情？只有当我们编写的程序耗费了大量的内存时，我们才能知道吗？我们该如何调试呢？

> An easy way to shine a light into the GC's inner workings is with some logging.

了解GC内部工作的一种简单方式是进行一些日志记录。

common.h，添加代码：

```
#define DEBUG_STRESS_GC
// 新增部分开始
#define DEBUG_LOG_GC
// 新增部分结束
#define UINT8_COUNT (UINT8_MAX + 1)
```

> When this is enabled, clox prints information to the console when it does something with dynamic memory.

启用这个功能后，当clox使用动态内存执行某些操作时，会将信息打印到控制台。

> We need a couple of includes.

我们需要一些引入。

memory.c，添加代码：

```
```

```
#include "vm.h"
// 新增部分开始
#ifdef DEBUG_LOG_GC
#include <stdio.h>
#include "debug.h"
#endif
// 新增部分结束
void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
```

> We don't have a collector yet, but we can start putting in some of the logging now. We'll want to know when a collection run starts.

我们还没有回收器，但我们现在可以开始添加一些日志记录。我们想要知道回收是在何时开始的。

memory.c，在collectGarbage()方法中添加代码：

```
void collectGarbage() {
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("-- gc begin\n");
#endif
// 新增部分结束
}
```

> Eventually we will log some other operations during the collection, so we'll also want to know when the show's over.

最终，我们会在回收过程中记录一些其它操作，因此我们也想知道回收什么时候结束。

memory.c，在collectGarbage()方法中添加代码：

```
  printf("-- gc begin\n");
#endif
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("-- gc end\n");
#endif
// 新增部分结束
}
```

> We don't have any code for the collector yet, but we do have functions for allocating and freeing, so we can instrument those now.

我们还没有关于回收器的任何代码，但是我们有分配和释放的函数，所以我们现在可以对这些函数进行检测。

object.c，在allocateObject()方法中添加代码：

```
  vm.objects = object;
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("%p allocate %zu for %d\n", (void*)object, size, type);
#endif
// 新增部分结束
  return object;
```

> And at the end of an object's lifespan:

在对象的生命周期结束时：

memory.c，在freeObject()方法中添加代码：

```
static void freeObject(Obj* object) {
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("%p free type %d\n", (void*)object, object->type);
#endif
// 新增部分结束
  switch (object->type) {
```

> With these two flags, we should be able to see that we're making progress as we work through the rest of the chapter.

有了这两个标志，我们应该能够看到我们在本章其余部分的学习中取得了进展。

26.3 Marking the Roots

26.3 标记根

> Objects are scattered across the heap like stars in the inky night sky. A reference from one object to another forms a connection, and these constellations are the graph that the mark phase traverses. Marking begins at the roots.

对象就像漆黑夜空中的星星一样散落在堆中。从一个对象到另一个对象的引用形成了一种连接，而这些星座就是标记阶段需要遍历的图。标记是从根开始的。

memory.c，在collectGarbage()方法中添加代码：

```
#ifdef DEBUG_LOG_GC
  printf("-- gc begin\n");
#endif
  // 新增部分开始
  markRoots();
  // 新增部分结束
#ifdef DEBUG_LOG_GC
```

> Most roots are local variables or temporaries sitting right in the VM’s stack, so we start by walking that.

大多数根是虚拟机栈中的局部变量或临时变量，因此我们从遍历栈开始：

memory.c，在freeObject()方法后添加代码：

```
static void markRoots() {
  for (Value* slot = vm.stack; slot < vm.stackTop; slot++) {
    markValue(*slot);
  }
}
```

> To mark a Lox value, we use this new function:

为了标记Lox值，我们使用这个新函数：

memory.h，在reallocate()方法后添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize);
// 新增部分开始
void markValue(Value value);
// 新增部分结束
void collectGarbage();
```

> Its implementation is here:

它的实现在这里：

memory.c，在reallocate()方法后添加代码：

```
void markValue(Value value) {
  if (IS_OBJ(value)) markObject(AS_OBJ(value));
}
```

> Some Lox values—numbers, Booleans, and `nil`—are stored directly inline in Value and require no heap allocation. The garbage collector doesn’t need to worry about them at all, so the first thing we do is ensure that the value is an actual heap object. If so, the real work happens in this function:

一些Lox值（数字、布尔值和`nil`）直接内联存储在Value中，不需要堆分配。垃圾回收器根本不需要担心这些，因此我们要做的第一件事是确保值是一个真正的堆对象。如果是这样，真正的工作就发生在这个函数中：

memory.h，在reallocate()方法后添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize);
// 新增部分开始
void markObject(Obj* object);
```

```
// 新增部分结束
void markValue(Value value);
```

> Which is defined here:

下面是定义：

memory.c，在reallocate()方法后添加代码：

```
void markObject(Obj* object) {
  if (object == NULL) return;
  object->isMarked = true;
}
```

> The `NULL` check is unnecessary when called from `markValue()`. A Lox Value that is some kind of Obj type will always have a valid pointer. But later we will call this function directly from other code, and in some of those places, the object being pointed to is optional.

从`markValue()`中调用时，`NULL`检查是不必要的。某种Obj类型的Lox Value一定会有一个有效的指针。但稍后我们将从其它代码中直接调用这个函数，在其中一些地方，被指向的对象是可选的。

Assuming we do have a valid object, we mark it by setting a flag. That new field lives in the Obj header struct all objects share.

假设我们确实有一个有效的对象，我们通过设置一个标志来标记它。这个新字段存在于所有对象共享的Obj头中。

object.h，在结构体Obj中添加代码：

```
  ObjType type;
  // 新增部分开始
  bool isMarked;
  // 新增部分结束
  struct Obj* next;
```

> Every new object begins life unmarked because we haven't yet determined if it is reachable or not.

每个新对象在开始时都没有标记，因为我们还不确定它是否可达。

object.c，在allocateObject()方法中添加代码：

```
  object->type = type;
  // 新增部分开始
  object->isMarked = false;
  // 新增部分结束
  object->next = vm.objects;
```

> Before we go any farther, let’s add some logging to `markObject()`.

在进一步讨论之前，我们先在`markObject()`中添加一些日志。

memory.c，在markObject()方法中添加代码：

```
void markObject(Obj* object) {
  if (object == NULL) return;
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("%p mark ", (void*)object);
  printValue(OBJ_VAL(object));
  printf("\n");
#endif
// 新增部分结束
  object->isMarked = true;
```

> This way we can see what the mark phase is doing. Marking the stack takes care of local variables and temporaries. The other main source of roots are the global variables.

这样我们就可以看到标记阶段在做什么。对栈进行标记可以处理局部变量和临时变量。另一个根的主要来源就是全局变量。

memory.c，在markRoots()方法中添加代码：

```
    markValue(*slot);
  }
  // 新增部分开始
  markTable(&vm.globals);
  // 新增部分结束
}
```

> Those live in a hash table owned by the VM, so we’ll declare another helper function for marking all of the objects in a table.

它们位于VM拥有的一个哈希表中，因此我们会声明另一个辅助函数来标记表中的所有对象。

table.h，在tableFindString()方法后添加代码：

```
ObjString* tableFindString(Table* table, const char* chars,
                           int length, uint32_t hash);
// 新增部分开始
void markTable(Table* table);
// 新增部分结束
#endif
```

> We implement that in the “table” module here:

我们在“table”模块中实现它：

table.c，在tableFindString()方法后添加代码：

```
void markTable(Table* table) {
  for (int i = 0; i < table->capacity; i++) {
    Entry* entry = &table->entries[i];
    markObject((Obj*)entry->key);
    markValue(entry->value);
  }
}
```

> Pretty straightforward. We walk the entry array. For each one, we mark its value. We also mark the key strings for each entry since the GC manages those strings too.

非常简单。我们遍历条目数组。对于每个条目，我们标记其值。我们还会标记每个条目的键字符串，因为GC也要管理这些字符串。

> ### 26.3.1 Less obvious roots

26.3.1 不明显的根

> Those cover the roots that we typically think of—the values that are obviously reachable because they’re stored in variables the user’s program can see. But the VM has a few of its own hidey-holes where it squirrels away references to values that it directly accesses.

这些覆盖了我们通常认为的根——那些明显可达的值，因为它们存储在用户程序可以看到的变量中。但是虚拟机也有一些自己的藏身之所，可以隐藏对直接访问的值的引用。

> Most function call state lives in the value stack, but the VM maintains a separate stack of CallFrames. Each CallFrame contains a pointer to the closure being called. The VM uses those pointers to access constants and upvalues, so those closures need to be kept around too.

大多数函数调用状态都存在于值栈中，但是虚拟机维护了一个单独的CallFrame栈。每个CallFrame都包含一个指向被调用闭包的指针。VM使用这些指针来访问常量和上值，所以这些闭包也需要被保留下来。

memory.c，在markRoots()方法中添加代码：

```
  }
  // 新增部分开始
  for (int i = 0; i < vm.frameCount; i++) {
    markObject((Obj*)vm.frames[i].closure);
  }
  // 新增部分结束
  markTable(&vm.globals);
```

> Speaking of upvalues, the open upvalue list is another set of values that the VM can directly reach.

说到上值，开放上值列表是VM可以直接访问的另一组值。

memory.c，在markRoots()方法中添加代码：

```
  for (int i = 0; i < vm.frameCount; i++) {
    markObject((Obj*)vm.frames[i].closure);
  }
  // 新增部分开始
  for (ObjUpvalue* upvalue = vm.openUpvalues;
       upvalue != NULL;
       upvalue = upvalue->next) {
    markObject((Obj*)upvalue);
  }
  // 新增部分结束
  markTable(&vm.globals);
```

> Remember also that a collection can begin during *any* allocation. Those allocations don't just happen while the user's program is running. The compiler itself periodically grabs memory from the heap for literals and the constant table. If the GC runs while we're in the middle of compiling, then any values the compiler directly accesses need to be treated as roots too.

还要记住，回收可能会在*任何*分配期间开始。这些分配并不仅仅是在用户程序运行的时候发生。编译器本身会定期从堆中获取内存，用于存储字面量和常量表。如果GC在编译期间运行，那么编译器直接访问的任何值也需要被当作根来处理。

> To keep the compiler module cleanly separated from the rest of the VM, we'll do that in a separate function.

为了保持编译器模块与虚拟机的其它部分完全分离，我们在一个单独的函数中完成这一工作。

memory.c，在markRoots()方法中添加代码：

```
  markTable(&vm.globals);
  // 新增部分开始
  markCompilerRoots();
  // 新增部分结束
}
```

> It's declared here:

它是在这里声明的：

compiler.h，在compile()方法后添加代码：

```
ObjFunction* compile(const char* source);
// 新增部分开始
void markCompilerRoots();
```

```
// 新增部分结束
#endif
```

> Which means the “memory” module needs an include.

这意味着“memory”模块需要引入头文件。

memory.c，添加代码：

```
#include <stdlib.h>
// 新增部分开始
#include "compiler.h"
// 新增部分结束
#include "memory.h"
```

> And the definition is over in the “compiler” module.

定义在“compiler”模块中。

compiler.c，在compile()方法后添加代码：

```
void markCompilerRoots() {
  Compiler* compiler = current;
  while (compiler != NULL) {
    markObject((Obj*)compiler->function);
    compiler = compiler->enclosing;
  }
}
```

> Fortunately, the compiler doesn’t have too many values that it hangs on to. The only object it uses is the ObjFunction it is compiling into. Since function declarations can nest, the compiler has a linked list of those and we walk the whole list.

幸运的是，编译器并没有太多挂载的值。它唯一使用的对象是它正在编译的ObjFunction。由于函数声明可以嵌套，编译器有一个函数声明的链表，我们遍历整个列表。

> Since the “compiler” module is calling `markObject()`, it also needs an include.

因为“compiler”模块会调用`markObject()`，也需要引入。

compiler.c，添加代码：

```
#include "compiler.h"
// 新增部分开始
#include "memory.h"
// 新增部分结束
#include "scanner.h"
```

> Those are all the roots. After running this, every object that the VM—runtime and compiler—can get to *without* going through some other object has its mark bit set.

这些就是所有的根。运行这段程序后，虚拟机（运行时和编译器）无需通过其它对象就可以达到的每个对象，其标记位都被设置了。

26.4 Tracing Object References

26.4 跟踪对象引用

> The next step in the marking process is tracing through the graph of references between objects to find the indirectly reachable values. We don't have instances with fields yet, so there aren't many objects that contain references, but we do have some. In particular, ObjClosure has the list of ObjUpvalues it closes over as well as a reference to the raw ObjFunction that it wraps. ObjFunction, in turn, has a constant table containing references to all of the literals created in the function's body. This is enough to build a fairly complex web of objects for our collector to crawl through.

标记过程的下一步是跟踪对象之间的引用图，找到间接可达的对象。我们现在还没有带有字段的实例，因此包含引用的对象不多，但确实有一些。特别的，ObjClosure拥有它所关闭的ObjUpvalue列表，以及它所包装的指向原始ObjFunction的引用^7。反过来，ObjFunction有一个常量表，包含函数体中创建的所有字面量的引用。这足以构建一个相当复杂的对象网络，供回收器爬取。

> Now it's time to implement that traversal. We can go breadth-first, depth-first, or in some other order. Since we just need to find the *set* of all reachable objects, the order we visit them mostly doesn't matter.

现在是时候实现遍历了。我们可以按照广度优先、深度优先或其它顺序进行遍历。因为我们只需要找到所有可达对象的集合，所以访问它们的顺序几乎没有影响^8。

26.4.1 The tricolor abstraction

26.4.1 三色抽象

> As the collector wanders through the graph of objects, we need to make sure it doesn't lose track of where it is or get stuck going in circles. This is particularly a concern for advanced implementations like incremental GCs that interleave marking with running pieces of the user's program. The collector needs to be able to pause and then pick up where it left off later.

当回收器在对象图中漫游时，我们需要确保它不会失去对其位置的跟踪或者陷入循环。这对于像增量GC这样的高级实现来说尤其值得关注，因为增量GC将标记与用户程序的的运行部分交织在一起。回收器需要能够暂停，稍后在停止的地方重新开始。

> To help us soft-brained humans reason about this complex process, VM hackers came up with a metaphor called the **tricolor abstraction**. Each object has a conceptual "color" that tracks what state the object is in, and what work is left to do.

为了帮助我们这些愚蠢的人类理解这个复杂的过程，虚拟机专家们想出了一个称为**三色抽象**的比喻。每个对象都有一个概念上的"颜色"，用于追踪对象处于什么状态，以及还需要做什么工作^9。

> - ○ **White:** At the beginning of a garbage collection, every object is white. This color means we have not reached or processed the object at all.
> - ◍ **Gray:** During marking, when we first reach an object, we darken it gray. This color means we know the object itself is reachable and should not be collected. But we have not yet traced *through* it to see what *other* objects it references. In graph algorithm terms, this is the *worklist*—the set of objects we know about but haven't processed yet.
> - ● **Black:** When we take a gray object and mark all of the objects it references, we then turn the gray object black. This color means the mark phase is done processing that object.

- ○ **白色:** 在垃圾回收的开始阶段，每个对象都是白色的。这种颜色意味着我们根本没有达到或处理该对象。
- ◍ **灰色:** 在标记过程中，当我们第一次达到某个对象时，我们将其染为灰色。这种颜色意味着我们知道这个对象本身是可达的，不应该被收集。但我们还没有*通过*它来跟踪它引用的*其它*对象。用图算法的术语来说，这就是*工作列表（worklist）*——我们知道但还没有被处理的对象集合。
- ● **黑色:** 当我们接受一个灰色对象，并将其引用的所有对象全部标记后，我们就把这个灰色对象变为黑色。这种颜色意味着标记阶段已经完成了对该对象的处理。

> In terms of that abstraction, the marking process now looks like this:

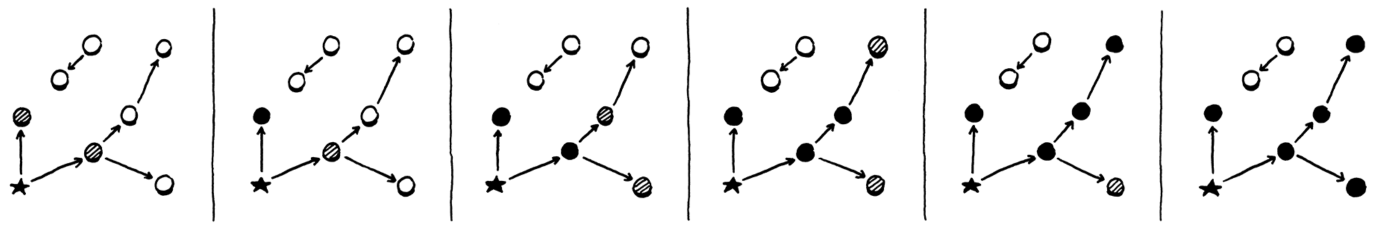
从这个抽象的角度看，标记过程新增看起来是这样的：

> 1. Start off with all objects white.
> 2. Find all the roots and mark them gray.
> 3. Repeat as long as there are still gray objects:
>   1. Pick a gray object. Turn any white objects that the object mentions to gray.
>   2. Mark the original gray object black.

1. 开始时，所有对象都是白色的。
2. 找到所有的根，将它们标记为灰色。
3. 只要还存在灰色对象，就重复此过程：
   1. 选择一个灰色对象。将该对象引用的所有白色对象标记为灰色。
   2. 将原来的灰色对象标记为黑色。

> I find it helps to visualize this. You have a web of objects with references between them. Initially, they are all little white dots. Off to the side are some incoming edges from the VM that point to the roots. Those roots turn gray. Then each gray object's siblings turn gray while the object itself turns black. The full effect is a gray wavefront that passes through the graph, leaving a field of reachable black objects behind it. Unreachable objects are not touched by the wavefront and stay white.

我发现把它可视化很有帮助。你有一个对象网络，对象之间有引用。最初，它们都是小白点。旁边是一些虚拟机的传入边，这些边指向根。这些根变成了灰色。然后每个灰色对象的兄弟节点变成灰色，而该对象本身变成黑色。完整的效果是一个灰色波前穿过图，在它后面留下一个可达的黑色对象区域。不可达对象不会被波前触及，并保持白色。



> At the end, you're left with a sea of reached, black objects sprinkled with islands of white objects that can be swept up and freed. Once the unreachable objects are freed, the remaining objects—all black—are reset to white for the next garbage collection cycle.

最后，你会看到一片可达的、黑色对象组成的海洋，其中点缀着可以清除和释放的白色对象组成的岛屿。一旦不可达的对象被释放，剩下的对象（全部为黑色）会被重置为白色，以便在下一个垃圾收集周期使用^10。

> ## 26.4.2 A worklist for gray objects

26.4.2 灰色对象的工作列表

> In our implementation we have already marked the roots. They're all gray. The next step is to start picking them and traversing their references. But we don't have any easy way to find them. We set a field on the object, but that's it. We don't want to have to traverse the entire object list looking for objects with that field set.

在我们的实现中，我们已经对根进行了标记。它们都是灰色的。下一步是开始挑选灰色对象并遍历其引用。但是我们没有任何简单的方法来查找灰色对象。我们在对象上设置了一个字段，但也仅此而已。我们不希望遍历整个对象列表来查找设置了该字段的对象。

> Instead, we'll create a separate worklist to keep track of all of the gray objects. When an object turns gray, in addition to setting the mark field we'll also add it to the worklist.

相反，我们创建一个单独的工作列表来跟踪所有的灰色对象。当某个对象变成灰色时，除了设置标记字段外，我们还会将它添加到工作列表中。

memory.c，在markObject()方法中添加代码：

```
  object->isMarked = true;
  // 新增部分开始
  if (vm.grayCapacity < vm.grayCount + 1) {
    vm.grayCapacity = GROW_CAPACITY(vm.grayCapacity);
    vm.grayStack = (Obj**)realloc(vm.grayStack,
                                  sizeof(Obj*) * vm.grayCapacity);
  }

  vm.grayStack[vm.grayCount++] = object;
  // 新增部分结束
}
```

> We could use any kind of data structure that lets us put items in and take them out easily. I picked a stack because that's the simplest to implement with a dynamic array in C. It works mostly like other dynamic arrays we've built in Lox, *except*, note that it calls the *system* `realloc()` function and not our

> own `reallocate()` wrapper. The memory for the gray stack itself is *not* managed by the garbage collector. We don't want growing the gray stack during a GC to cause the GC to recursively start a new GC. That could tear a hole in the space-time continuum.

我们可以使用任何类型的数据结构，让我们可以轻松地放入和取出项目。我选择了栈，因为这是用C语言实现动态数组最简单的方法。它的工作原理与我们在Lox中构建的其它动态数组基本相同，*除了一点*，要注意它调用了*系统*的`realloc()`函数，而不是我们自己包装的`reallocate()`。灰色对象栈本身的内存是*不被*垃圾回收器管理的。我们不希望因为GC过程中增加灰色对象栈，导致GC递归地发起一个新的GC。这可能会在时空连续体上撕开一个洞。

> We'll manage its memory ourselves, explicitly. The VM owns the gray stack.

我们会自己显式地管理它的内存。VM拥有这个灰色栈。

vm.h，在结构体VM中添加代码：

```
  Obj* objects;
  // 新增部分开始
  int grayCount;
  int grayCapacity;
  Obj** grayStack;
  // 新增部分结束
} VM;
```

> It starts out empty.

开始时是空的。

vm.c，在initVM()方法中添加代码：

```
  vm.objects = NULL;
  // 新增部分开始
  vm.grayCount = 0;
  vm.grayCapacity = 0;
  vm.grayStack = NULL;
  // 新增部分结束
  initTable(&vm.globals);
```

> And we need to free it when the VM shuts down.

当VM关闭时，我们需要释放它。

memory.c，在freeObjects()方法中添加代码：

```
    object = next;
  }
  // 新增部分开始
  free(vm.grayStack);
```

```
    // 新增部分结束
  }
```

> We take full responsibility for this array. That includes allocation failure. If we can't create or grow the gray stack, then we can't finish the garbage collection. This is bad news for the VM, but fortunately rare since the gray stack tends to be pretty small. It would be nice to do something more graceful, but to keep the code in this book simple, we just abort.

我们对这个数组负担全部责任，其中包括分配失败。如果我们不能创建或扩张灰色栈，那我们就无法完成垃圾回收。这对VM来说是个坏消息，但幸运的是这很少发生，因为灰色栈往往是非常小的。如果能做得更优雅一些就好了，但是为了保持本书中的代码简单，我们就停在这里吧^11。

memory.c，在markObject()方法中添加代码：

```
    vm.grayStack = (Obj**)realloc(vm.grayStack,
                                  sizeof(Obj*) * vm.grayCapacity);
    // 新增部分开始
    if (vm.grayStack == NULL) exit(1);
    // 新增部分结束
  }
```

> 26.4.3 Processing gray objects

26.4.3 处理灰色对象

> OK, now when we're done marking the roots, we have both set a bunch of fields and filled our work list with objects to chew through. It's time for the next phase.

好了，现在我们在完成对根的标记后，既设置了一堆字段，也用待处理的对象填满了我们的工作列表。是时候进入下一阶段了。

memory.c，在collectGarbage()方法中添加代码：

```
  markRoots();
  // 新增部分开始
  traceReferences();
  // 新增部分结束
#ifdef DEBUG_LOG_GC
```

> Here's the implementation:

下面是其实现：

memory.c，在markRoots()方法后添加代码：

```c
static void traceReferences() {
  while (vm.grayCount > 0) {
    Obj* object = vm.grayStack[--vm.grayCount];
    blackenObject(object);
  }
}
```

> It's as close to that textual algorithm as you can get. Until the stack empties, we keep pulling out gray objects, traversing their references, and then marking them black. Traversing an object's references may turn up new white objects that get marked gray and added to the stack. So this function swings back and forth between turning white objects gray and gray objects black, gradually advancing the entire wavefront forward.

这与文本描述的算法已经尽可能接近了。在栈清空之前，我们会不断取出灰色对象，遍历它们的引用，然后将它们标记为黑色。遍历某个对象的引用可能会出现新的白色对象，这些对象被标记为灰色并添加到栈中。所以这个函数在把白色对象变成灰色和把灰色对象变成黑色之间来回摆动，逐渐把整个波前向前推进。

> Here's where we traverse a single object's references:

下面是我们遍历某个对象的引用的地方：

memory.c，在markValue()方法后添加代码：

```c
static void blackenObject(Obj* object) {
  switch (object->type) {
    case OBJ_NATIVE:
    case OBJ_STRING:
      break;
  }
}
```

> Each object kind has different fields that might reference other objects, so we need a specific blob of code for each type. We start with the easy ones—strings and native function objects contain no outgoing references so there is nothing to traverse.

每种对象类型都有不同的可能引用其它对象的字段，因此我们需要为每种类型编写一块特定的代码。我们从简单的开始——字符串和本地函数对象不包含向外的引用，因此没有任何东西需要遍历^12。

> Note that we don't set any state in the traversed object itself. There is no direct encoding of "black" in the object's state. A black object is any object whose `isMarked` field is set and that is no longer in the gray stack.

注意，我们没有在已被遍历的对象本身中设置任何状态。在对象的状态中，没有对"black"的直接编码。黑色对象是`isMarked`字段被设置且不再位于灰色栈中的任何对象^13。

> Now let's start adding in the other object types. The simplest is upvalues.

现在让我们开始添加其它的对象类型。最简单的是上值。

memory.c，在blackenObject()方法中添加代码：

```
static void blackenObject(Obj* object) {
  switch (object->type) {
    // 新增部分开始
    case OBJ_UPVALUE:
      markValue(((ObjUpvalue*)object)->closed);
      break;
    // 新增部分结束
    case OBJ_NATIVE:
```

> When an upvalue is closed, it contains a reference to the closed-over value. Since the value is no longer on the stack, we need to make sure we trace the reference to it from the upvalue.

当某个上值被关闭后，它包含一个指向关闭值的引用。由于该值不在栈上，我们需要确保从上值中跟踪对它的引用。

> Next are functions.

接下来是函数。

memory.c，在blackenObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_FUNCTION: {
      ObjFunction* function = (ObjFunction*)object;
      markObject((Obj*)function->name);
      markArray(&function->chunk.constants);
      break;
    }
    // 新增部分结束
    case OBJ_UPVALUE:
```

> Each function has a reference to an ObjString containing the function's name. More importantly, the function has a constant table packed full of references to other objects. We trace all of those using this helper:

每个函数都有一个对包含函数名称的ObjString 的引用。更重要的是，函数有一个常量表，其中充满了对其它对象的引用。我们使用这个辅助函数来跟踪它们：

memory.c，在markValue()方法后添加代码：

```
static void markArray(ValueArray* array) {
  for (int i = 0; i < array->count; i++) {
    markValue(array->values[i]);
  }
}
```

> The last object type we have now—we'll add more in later chapters—is closures.

我们现在拥有的最后一种对象类型（我们会在后面的章节中添加更多）是闭包。

memory.c，在blackenObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_CLOSURE: {
      ObjClosure* closure = (ObjClosure*)object;
      markObject((Obj*)closure->function);
      for (int i = 0; i < closure->upvalueCount; i++) {
        markObject((Obj*)closure->upvalues[i]);
      }
      break;
    }
    // 新增部分结束
    case OBJ_FUNCTION: {
```

> Each closure has a reference to the bare function it wraps, as well as an array of pointers to the upvalues it captures. We trace all of those.

每个闭包都有一个指向其包装的裸函数的引用，以及一个指向它所捕获的上值的指针数组。我们要跟踪所有这些。

> That's the basic mechanism for processing a gray object, but there are two loose ends to tie up. First, some logging.

这就是处理灰色对象的基本机制，但还有两个未解决的问题。首先，是一些日志记录。

memory.c，在blackenObject()中添加代码：

```
static void blackenObject(Obj* object) {
// 新增部分开始
#ifdef DEBUG_LOG_GC
  printf("%p blacken ", (void*)object);
  printValue(OBJ_VAL(object));
  printf("\n");
#endif
// 新增部分结束
  switch (object->type) {
```

> This way, we can watch the tracing percolate through the object graph. Speaking of which, note that I said *graph*. References between objects are directed, but that doesn't mean they're *acyclic!* It's entirely possible to have cycles of objects. When that happens, we need to ensure our collector doesn't get stuck in an infinite loop as it continually re-adds the same series of objects to the gray stack.

这样一来，我们就可以观察到跟踪操作在对象图中的渗入情况。说到这里，请注意，我说的是图。对象之间的引用是有方向的，但这并不意味着它们是无循环的！完全有可能出现对象的循环。当这种情况发生时，我们需要确保，我们的回收器不会因为持续将同一批对象添加到灰色堆栈而陷入无限循环。

> The fix is easy.

解决方法很简单。

memory.c，在markObject()方法中添加代码：

```
  if (object == NULL) return;
  // 新增部分开始
  if (object->isMarked) return;
  // 新增部分结束
#ifdef DEBUG_LOG_GC
```

> If the object is already marked, we don’t mark it again and thus don’t add it to the gray stack. This ensures that an already-gray object is not redundantly added and that a black object is not inadvertently turned back to gray. In other words, it keeps the wavefront moving forward through only the white objects.

如果对象已经被标记，我们就不会再标记它，因此也不会把它添加到灰色栈中。这就保证了已经是灰色的对象不会被重复添加，而且黑色对象不会无意中变回灰色。换句话说，它使得波前只通过白色对象向前移动。

26.5 Sweeping Unused Objects

26.5 清除未使用的对象

> When the loop in `traceReferences()` exits, we have processed all the objects we could get our hands on. The gray stack is empty, and every object in the heap is either black or white. The black objects are reachable, and we want to hang on to them. Anything still white never got touched by the trace and is thus garbage. All that’s left is to reclaim them.

当`traceReferences()`中的循环退出时，我们已经处理了所有能接触到的对象。灰色栈是空的，堆中的每个对象不是黑色就是白色。黑色对象是可达的，我们想要抓住它们。任何仍然是白色的对象都没有被追踪器接触过，因此是垃圾。剩下的就是回收它们了。

memory.c，在collectGarbage()方法中添加代码：

```
  traceReferences();
  // 新增部分开始
  sweep();
  // 新增部分结束
#ifdef DEBUG_LOG_GC
```

> All of the logic lives in one function.

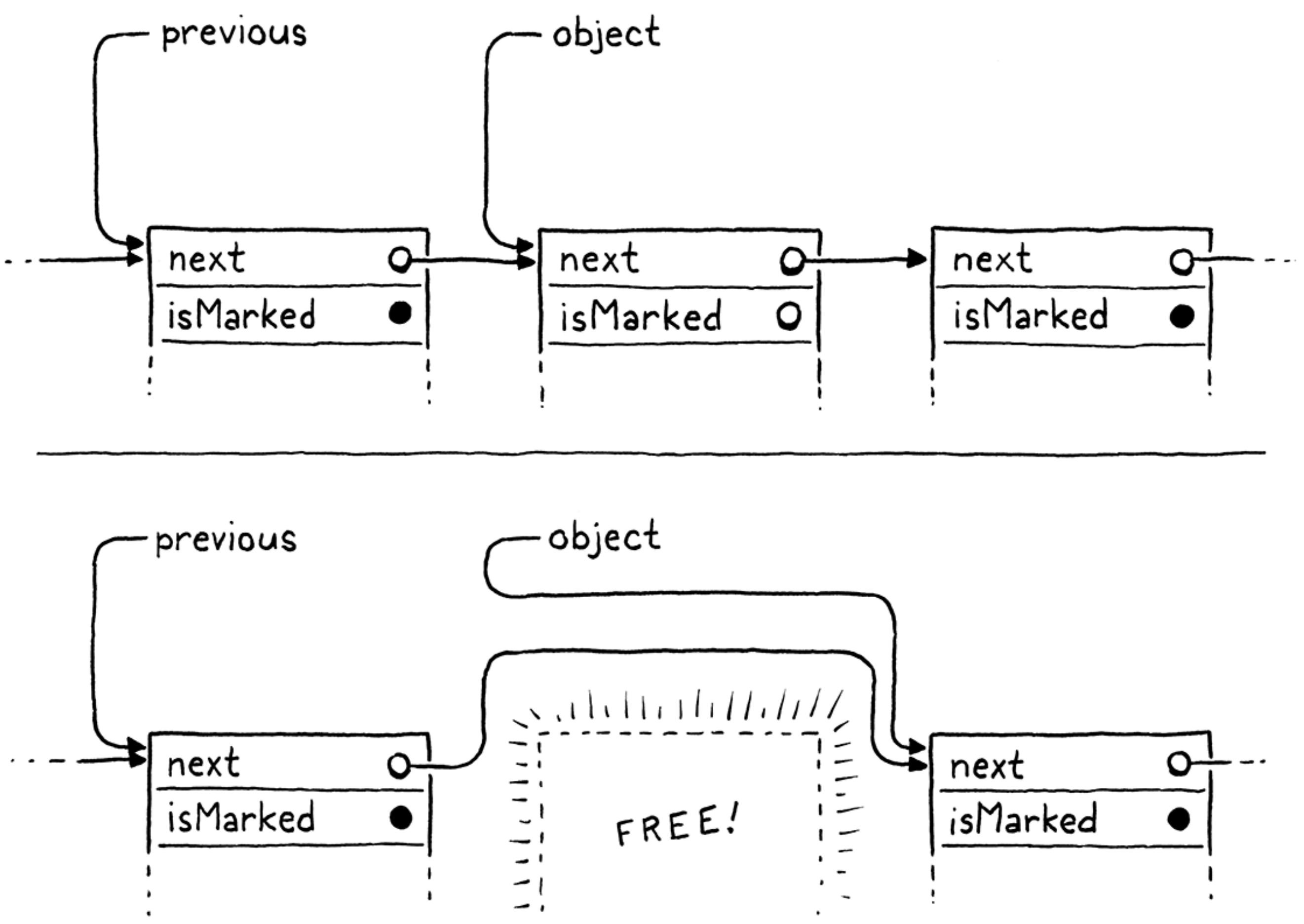
所有的逻辑都在一个函数中。

memory.c，在traceReferences()方法后添加代码：

```
static void sweep() {
  Obj* previous = NULL;
  Obj* object = vm.objects;
  while (object != NULL) {
    if (object->isMarked) {
      previous = object;
      object = object->next;
    } else {
      Obj* unreached = object;
      object = object->next;
      if (previous != NULL) {
        previous->next = object;
      } else {
        vm.objects = object;
      }

      freeObject(unreached);
    }
  }
}
```

> I know that's kind of a lot of code and pointer shenanigans, but there isn't much to it once you work through it. The outer `while` loop walks the linked list of every object in the heap, checking their mark bits. If an object is marked (black), we leave it alone and continue past it. If it is unmarked (white), we unlink it from the list and free it using the `freeObject()` function we already wrote.

我知道这有点像是一堆代码和指针的诡计，不过一旦你完成了，就没什么好说的。外层的`while`循环会遍历堆中每个对象组成的链表，检查它们的标记位。如果某个对象被标记（黑色），我们就不管它，继续进行。如果它没有被标记（白色），我们将它从链表中断开，并使用我们已经写好的`freeObject()`函数释放它。



> Most of the other code in here deals with the fact that removing a node from a singly linked list is cumbersome. We have to continuously remember the previous node so we can unlink its next pointer, and we have to handle the edge case where we are freeing the first node. But, otherwise, it's pretty simple—delete every node in a linked list that doesn't have a bit set in it.

这里大多数其它代码都在处理这样一个事实：从单链表中删除节点非常麻烦。我们必须不断地记住前一个节点，这样我们才能断开它的next指针，而且我们还必须处理释放第一个节点这种边界情况。但是，除此之外，它非常简单——删除链表中没有设置标记位的每个节点。

> There's one little addition:

还有一点需要补充：

memory.c，在sweep()方法中添加代码：

```
    if (object->isMarked) {
      // 新增部分开始
      object->isMarked = false;
      // 新增部分结束
      previous = object;
```

> After `sweep()` completes, the only remaining objects are the live black ones with their mark bits set. That's correct, but when the *next* collection cycle starts, we need every object to be white. So whenever we reach a black object, we go ahead and clear the bit now in anticipation of the next run.

在`sweep()`完成后，仅剩下的对象是带有标记位的活跃黑色对象。这是正确的，但在 下一个回收周期开始时，我们需要每个对象都是白色的。因此，每当我们碰到黑色对象时，我们就继续并清除标记位，为下一轮作业做好准备。

> ## 26.5.1 Weak references and the string pool

26.5.1 弱引用与字符串池

> We are almost done collecting. There is one remaining corner of the VM that has some unusual requirements around memory. Recall that when we added strings to clox we made the VM intern them all. That means the VM has a hash table containing a pointer to every single string in the heap. The VM uses this to de-duplicate strings.

我们差不多已经回收完毕了。虚拟机中还有一个剩余的角落，它对内存有着一些不寻常的要求。回想一下，我们在clox中添加字符串的时，我们让虚拟机对所有字符串进行驻留。这意味着VM拥有一个哈希表，其中包含指向堆中每个字符串的指针。虚拟机使用它来对字符串去重。

> During the mark phase, we deliberately did *not* treat the VM's string table as a source of roots. If we had, no string would *ever* be collected. The string table would grow and grow and never yield a single byte of memory back to the operating system. That would be bad.

在标记阶段，我们故意*不将*虚拟机的字符串表作为根的来源。如果我们这样做，就不会有字符串被回收。字符串表会不断增长，并且永远不会向操作系统让出一比特的内存。那就糟糕了^14。

> At the same time, if we *do* let the GC free strings, then the VM's string table will be left with dangling pointers to freed memory. That would be even worse.

同时，如果我们真的让GC释放字符串，那么VM的字符串表就会留下指向已释放内存的悬空指针。那就更糟糕了。

> The string table is special and we need special support for it. In particular, it needs a special kind of reference. The table should be able to refer to a string, but that link should not be considered a root when determining reachability. That implies that the referenced object can be freed. When that happens, the dangling reference must be fixed too, sort of like a magic, self-clearing pointer. This particular set of semantics comes up frequently enough that it has a name: a **weak reference**.

字符串表是很特殊的，我们需要对它进行特殊的支持。特别是，它需要一种特殊的引用。这个表应该能够引用字符串，但在确定可达性时，不应该将该链接视为根。这意味着被引用的对象也可以被释放。当这种情况发生时，悬空的引用也必须被修正，有点像一个神奇的、自我清除的指针。这组特定的语义出现得非常频繁，所以它有一个名字：弱引用。

> We have already implicitly implemented half of the string table's unique behavior by virtue of the fact that we *don't* traverse it during marking. That means it doesn't force strings to be reachable. The remaining piece is clearing out any dangling pointers for strings that are freed.

我们已经隐式地实现了一半的字符串表的独特行为，因为我们在标记阶段没有遍历它。这意味着它不强制要求字符串可达。剩下的部分就是清除任何指向被释放字符串的悬空指针。

> To remove references to unreachable strings, we need to know which strings *are* unreachable. We don't know that until after the mark phase has completed. But we can't wait until after the sweep phase is

> done because by then the objects—and their mark bits—are no longer around to check. So the right time is exactly between the marking and sweeping phases.

为了删除对不可达字符串的引用，我们需要知道哪些字符串不可达。在标记阶段完成之后，我们才能知道这一点。但是我们不能等到清除阶段完成之后，因为到那时对象（以及它们的标记位）已经无法再检查了。因此，正确的时机正好是在标记和清除阶段之间。

memory.c，在collectGarbage()方法中添加代码：

```
  traceReferences();
  // 新增部分开始
  tableRemoveWhite(&vm.strings);
  // 新增部分结束
  sweep();
```

> The logic for removing the about-to-be-deleted strings exists in a new function in the “table” module.

清除即将被删除的字符串的逻辑存在于“table”模块中的一个新函数中。

table.h，在tableFindString()方法后添加代码：

```
ObjString* tableFindString(Table* table, const char* chars,
                           int length, uint32_t hash);
// 新增部分开始
void tableRemoveWhite(Table* table);
// 新增部分结束
void markTable(Table* table);
```

> The implementation is here:

实现在这里：

table.c，在tableFindString()方法后添加代码：

```
void tableRemoveWhite(Table* table) {
  for (int i = 0; i < table->capacity; i++) {
    Entry* entry = &table->entries[i];
    if (entry->key != NULL && !entry->key->obj.isMarked) {
      tableDelete(table, entry->key);
    }
  }
}
```

> We walk every entry in the table. The string intern table uses only the key of each entry—it’s basically a hash *set* not a hash *map*. If the key string object’s mark bit is not set, then it is a white object that is moments from being swept away. We delete it from the hash table first and thus ensure we won’t see any dangling pointers.

我们遍历表中的每一项。字符串驻留表只使用了每一项的键——它基本上是一个HashSet而不是HashMap。如果键字符串对象的标记位没有被设置，那么它就是一个白色对象，很快就会被清除。我们首先从哈希表中删除它，从而确保不会看到任何悬空指针。

26.6 When to Collect

26.6 何时回收

> We have a fully functioning mark-sweep garbage collector now. When the stress testing flag is enabled, it gets called all the time, and with the logging enabled too, we can watch it do its thing and see that it is indeed reclaiming memory. But, when the stress testing flag is off, it never runs at all. It's time to decide when the collector should be invoked during normal program execution.

我们现在有了一个功能完备的标记-清除垃圾回收器。当压力测试标志启用时，它会一直被调用，而且在日志功能也被启用的情况下，我们可以观察到它正在工作，并看到它确实在回收内存。但是，当压力测试标志关闭时，它根本就不会运行。现在是时候决定，在正常的程序执行过程中，何时应该调用回收器。

> As far as I can tell, this question is poorly answered by the literature. When garbage collectors were first invented, computers had a tiny, fixed amount of memory. Many of the early GC papers assumed that you set aside a few thousand words of memory—in other words, most of it—and invoked the collector whenever you ran out. Simple.

据我所知，这个问题在文献中没有得到很好的回答。在垃圾回收器刚被发明出来的时候，计算机只有一个很小的、固定大小的内存。许多早期的GC论文假定你预留了几千个字的内存（换句话说，其中大部分是这样），并在内存用完时调用回收器。这很简单。

> Modern machines have gigs of physical RAM, hidden behind the operating system's even larger virtual memory abstraction, which is shared among a slew of other programs all fighting for their chunk of memory. The operating system will let your program request as much as it wants and then page in and out from the disc when physical memory gets full. You never really "run out" of memory, you just get slower and slower.

现代计算机拥有数以G计的物理内存，而操作系统在其基础上提供了更多的虚拟内存抽象，这些物理内存是由一系列其它程序共享的，所有程序都在争夺自己的那块内存。操作系统会允许你的程序尽可能多地申请内存，然后当物理内存满时会利用磁盘进行页面换入换出。你永远不会真的"耗尽"内存，只是变得越来越慢。

26.6.1 Latency and throughput

26.6.1 延迟和吞吐量

> It no longer makes sense to wait until you "have to", to run the GC, so we need a more subtle timing strategy. To reason about this more precisely, it's time to introduce two fundamental numbers used when measuring a memory manager's performance: *throughput* and *latency*.

等到"不得不做"的时候再去运行GC，就没有意义了，因此我们需要一种更巧妙的选时策略。为了更精确地解释这个问题，现在应该引入在度量内存管理器性能时使用的两个基本数值：*吞吐量*和*延迟*。

> Every managed language pays a performance price compared to explicit, user-authored deallocation. The time spent actually freeing memory is the same, but the GC spends cycles figuring out *which* memory to free. That is time *not* spent running the user's code and doing useful work. In our

> implementation, that's the entirety of the mark phase. The goal of a sophisticated garbage collector is to minimize that overhead.

与显式的、用户自发的释放内存相比，每一种托管语言都要付出性能代价。实际释放内存所花费的时间是相同的，但是GC花费了一些周期来计算要释放*哪些*内存。这些时间没有花在运行用户的代码和做有用的工作。在我们的实现中，这就是整个标记阶段。复杂的垃圾回收器的模板就是使这种开销最小化。

> There are two key metrics we can use to understand that cost better:

我们可以使用这两个关键指标来更好地理解成本：

- > **Throughput** is the total fraction of time spent running user code versus doing garbage collection work. Say you run a clox program for ten seconds and it spends a second of that inside `collectGarbage()`. That means the throughput is 90%—it spent 90% of the time running the program and 10% on GC overhead.
  >
  > Throughput is the most fundamental measure because it tracks the total cost of collection overhead. All else being equal, you want to maximize throughput. Up until this chapter, clox had no GC at all and thus 100% throughput. That's pretty hard to beat. Of course, it came at the slight expense of potentially running out of memory and crashing if the user's program ran long enough. You can look at the goal of a GC as fixing that "glitch" while sacrificing as little throughput as possible.

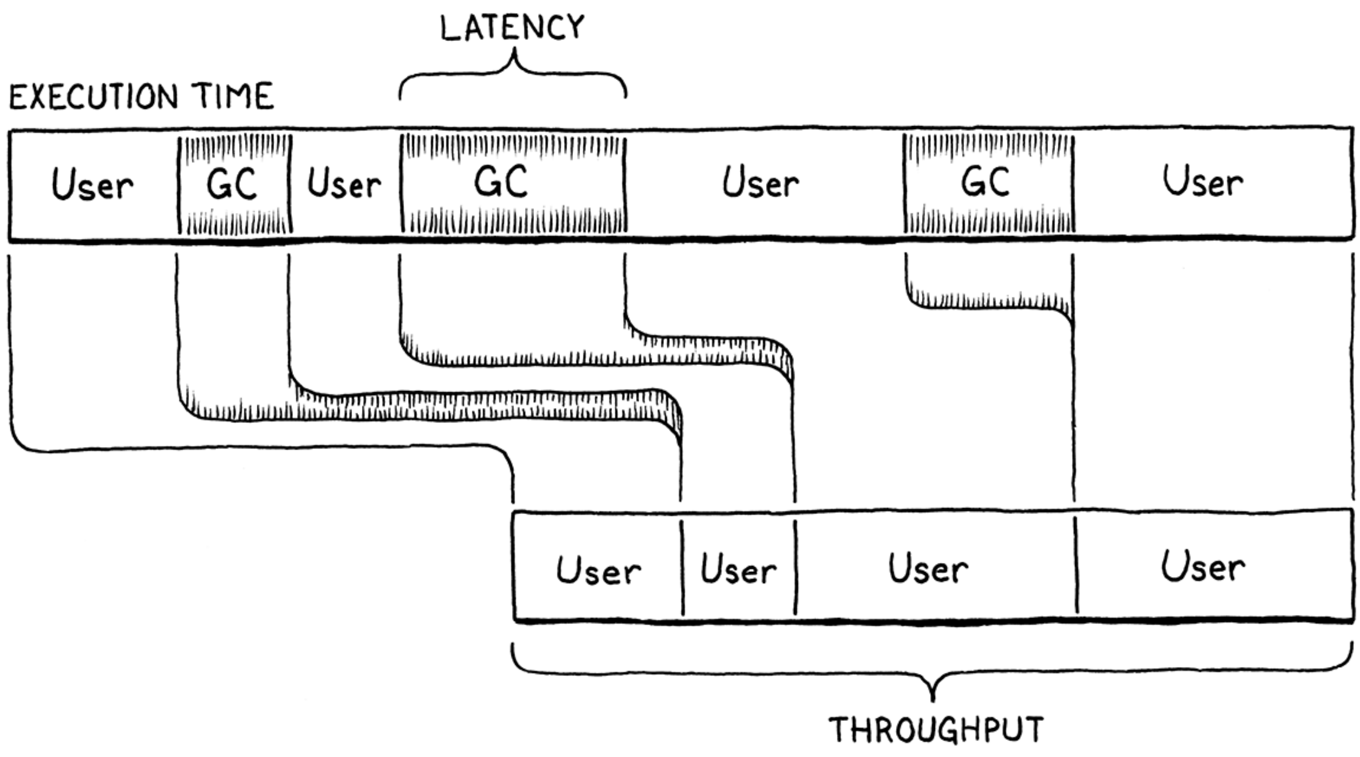
  吞吐量是指运行用户代码的时间与执行垃圾回收工作所花费的时间的总比例。假设你运行一个clox程序10秒钟，其中有1秒花在`collectGarbage()`中。这意味是吞吐量是90%——它花费了90%的时间运行程序，10%的时间用于GC开销。

  吞吐量是最基本的度量值，因为它跟踪的是回收开销的总成本。在其它条件相同的情况下，你会希望最大化吞吐量。在本章之前，clox完全没有GC，因此吞吐量为100%^15。这是很难做到的。当然，它的代价是，如果用户的程序运行时间足够长的话，可能会导致内存耗尽和程序崩溃。你可以把GC的目标看作是修复这个"小故障"，同时以牺牲尽可能少的吞吐量为代价。

- > **Latency** is the longest *continuous* chunk of time where the user's program is completely paused while garbage collection happens. It's a measure of how "chunky" the collector is. Latency is an entirely different metric than throughput.
  >
  > Consider two runs of a clox program that both take ten seconds. In the first run, the GC kicks in once and spends a solid second in `collectGarbage()` in one massive collection. In the second run, the GC gets invoked five times, each for a fifth of a second. The *total* amount of time spent collecting is still a second, so the throughput is 90% in both cases. But in the second run, the latency is only 1/5th of a second, five times less than in the first.

  延迟是指当垃圾回收发生时，用户的程序完全暂停的最长连续时间块。这是衡量回收器"笨重"程度的指标。延迟是一个与吞吐量完全不同的指标。

  考虑一下，一个程序的两次运行都花费了10秒。第一次运行时，GC启动了一次，并在`collectGarbage()`中花费了整整1秒钟进行了一次大规模的回收。在第二次运行中，GC被调用了五次，每次调用1/5秒。回收所花费的总时间仍然是1秒，所以这两种情况下的吞吐量都是90%。但是在第二次运行中，延迟只有1/5秒，比第一次少了5倍^16。



> If you like analogies, imagine your program is a bakery selling fresh-baked bread to customers. Throughput is the total number of warm, crusty baguettes you can serve to customers in a single day. Latency is how long the unluckiest customer has to wait in line before they get served.

如果你喜欢打比方，可以将你的程序想象成一家面包店，向顾客出售新鲜出炉的面包。吞吐量是指你在一天内可以为顾客提供的温暖结皮的法棍的总数。延迟是指最不走运的顾客在得到服务之前需要排队等候多长时间。

> Running the garbage collector is like shutting down the bakery temporarily to go through all of the dishes, sort out the dirty from the clean, and then wash the used ones. In our analogy, we don't have dedicated dishwashers, so while this is going on, no baking is happening. The baker is washing up.

运行垃圾回收器就像暂时关闭面包店，去检查所有的盘子，把脏的和干净的分开，然后把用过的洗掉。在我们的比喻中，我们没有专门的洗碗机，所以在这个过程中，没有烘焙发生。面包师正在清洗^17。

> Selling fewer loaves of bread a day is bad, and making any particular customer sit and wait while you clean all the dishes is too. The goal is to maximize throughput and minimize latency, but there is no free lunch, even inside a bakery. Garbage collectors make different trade-offs between how much throughput they sacrifice and latency they tolerate.

每天卖出更少的面包是糟糕的，让任何一个顾客坐着等你洗完所有的盘子也是如此。我们的目标是最大化吞吐量和最小化延迟，但是没有免费的午餐，即使是在面包店里。不同垃圾回收器在牺牲多少吞吐量和容忍多大延迟之间做出了不同的权衡。

> Being able to make these trade-offs is useful because different user programs have different needs. An overnight batch job that is generating a report from a terabyte of data just needs to get as much work done as fast as possible. Throughput is queen. Meanwhile, an app running on a user's smartphone needs to always respond immediately to user input so that dragging on the screen feels buttery smooth. The app can't freeze for a few seconds while the GC mucks around in the heap.

能够进行这些权衡是很有用的，因为不同的用户程序有不同的需求。一个从TB级数据中生成报告的夜间批处理作业，只需要尽可能快地完成尽可能多的工作。吞吐量为王。与此同时，在用户智能手机上运行的应用程序需

要总是对用户输入立即做出响应，这样才能让用户在屏幕上拖拽时感觉非常流畅。应用程序不能因为GC在堆中乱翻而冻结几秒钟。

> As a garbage collector author, you control some of the trade-off between throughput and latency by your choice of collection algorithm. But even within a single algorithm, we have a lot of control over *how frequently* the collector runs.

作为一个垃圾回收器作者，你可以通过选择收集算法来控制吞吐量和延迟之间的一些权衡。但即使在单一的算法中，我们也可以对回收器的运行频率有很大的控制。

> Our collector is a **stop-the-world GC** which means the user's program is paused until the entire garbage collection process has completed. If we wait a long time before we run the collector, then a large number of dead objects will accumulate. That leads to a very long pause while the collector runs, and thus high latency. So, clearly, we want to run the collector really frequently.

我们的回收器是一个**stop-the-world GC**，这意味着会暂停用户的程序，直到垃圾回收过程完成^18。如果我们在运行回收器之前等待很长时间，那么将会积累大量的死亡对象。这会导致回收器在运行时会出现很长时间的停顿，从而导致高延迟。所以，很明显，我们希望频繁地运行回收器。

> But every time the collector runs, it spends some time visiting live objects. That doesn't really *do* anything useful (aside from ensuring that they don't incorrectly get deleted). Time visiting live objects is time not freeing memory and also time not running user code. If you run the GC *really* frequently, then the user's program doesn't have enough time to even generate new garbage for the VM to collect. The VM will spend all of its time obsessively revisiting the same set of live objects over and over, and throughput will suffer. So, clearly, we want to run the collector really *in*frequently.

但是每次回收器运行时，它都要花一些时间来访问活动对象。这其实并没有什么用处（除了确保它们不会被错误地删除之外）。访问活动对象的时间是没有释放内存的时间，也是没有运行用户代码的时间。如果你*真的*非常频繁地运行GC，那么用户的程序甚至没有足够的时间生成新的垃圾供VM回收。VM会花费所有的时间反复访问相同的活动对象，吞吐量将会受到影响。所以，很明显，我们也不希望频繁地运行回收器。

> In fact, we want something in the middle, and the frequency of when the collector runs is one of our main knobs for tuning the trade-off between latency and throughput.

事实上，我们想要的是介于两者之间的东西，而回收器的运行频率是我们调整延迟和吞吐量之间权衡的主要因素之一。

> ### 26.6.2 Self-adjusting heap

26.6.2 自适应堆

> We want our GC to run frequently enough to minimize latency but infrequently enough to maintain decent throughput. But how do we find the balance between these when we have no idea how much memory the user's program needs and how often it allocates? We could pawn the problem onto the user and force them to pick by exposing GC tuning parameters. Many VMs do this. But if we, the GC authors, don't know how to tune it well, odds are good most users won't either. They deserve a reasonable default behavior.

我们希望GC运行得足够频繁，以最小化延迟，但又不能太频繁，以维持良好的吞吐量。但是，当我们不知道用户程序需要多少内存以及内存分配的频率时，我们如何在两者之间找到平衡呢？我们可以把问题推给用户，并

通过暴露GC调整参数来迫使他们进行选择。许多虚拟机都是这样做的。但是，如果我们这些GC的作者都不知道如何很好地调优回收器，那么大多数用户可能也不知道。他们理应得到一个合理的默认行为。

> I'll be honest with you, this is not my area of expertise. I've talked to a number of professional GC hackers—this is something you can build an entire career on—and read a lot of the literature, and all of the answers I got were . . . vague. The strategy I ended up picking is common, pretty simple, and (I hope!) good enough for most uses.

说实话，这不是我的专业领域。我曾经和一些专业的GC专家交谈过（GC是一项可以投入整个职业生涯的东西），并且阅读了大量的文献，我得到的所有答案都是......模糊的。我最终选择的策略很常见，也很简单，而且（我希望！）对大多数用途来说足够好。

> The idea is that the collector frequency automatically adjusts based on the live size of the heap. We track the total number of bytes of managed memory that the VM has allocated. When it goes above some threshold, we trigger a GC. After that, we note how many bytes of memory remain—how many were *not* freed. Then we adjust the threshold to some value larger than that.

其思想是，回收器的频率根据堆的大小自动调整。我们根据虚拟机已分配的托管内存的总字节数。当它超过某个阈值时，我们就触发一次GC。在那之后，我们关注一下有多少字节保留下来——多少没有被释放。然后我们将阈值调整为比它更大的某个值。

> The result is that as the amount of live memory increases, we collect less frequently in order to avoid sacrificing throughput by re-traversing the growing pile of live objects. As the amount of live memory goes down, we collect more frequently so that we don't lose too much latency by waiting too long.

其结果是，随着活动内存数量的增加，我们回收的频率会降低，以避免因为重新遍历不断增长的活动对象而牺牲吞吐量。随着活动内存数量的减少，我们会更频繁地收集，这样我们就不会因为等待时间过长而造成太多的延迟。

> The implementation requires two new bookkeeping fields in the VM.

这个实现需要在虚拟机中设置两个新的簿记字段。

vm.h，在结构体VM中添加代码：

```
  ObjUpvalue* openUpvalues;
  // 新增部分开始
  size_t bytesAllocated;
  size_t nextGC;
  // 新增部分结束
  Obj* objects;
```

> The first is a running total of the number of bytes of managed memory the VM has allocated. The second is the threshold that triggers the next collection. We initialize them when the VM starts up.

第一个是虚拟机已分配的托管内存实时字节总数。第二个是触发下一次回收的阈值。我们在虚拟机启动时初始化它们。

vm.c，在initVM()方法中添加代码：

```
  vm.objects = NULL;
  // 新增部分开始
  vm.bytesAllocated = 0;
  vm.nextGC = 1024 * 1024;
  // 新增部分结束
  vm.grayCount = 0;
```

> The starting threshold here is arbitrary. It’s similar to the initial capacity we picked for our various dynamic arrays. The goal is to not trigger the first few GCs *too* quickly but also to not wait too long. If we had some real-world Lox programs, we could profile those to tune this. But since all we have are toy programs, I just picked a number.

这里的起始阈值是任意的。它类似于我们为各种动态数据选择的初始容量。我们的目标是不要太快触发最初的几次GC，但是也不要等得太久。如果我们有一些真实的Lox程序，我们可以对程序进行剖析来调整这个参数。但是因为我们写的都是一些玩具程序，我只是随意选了一个数字^19。

> Every time we allocate or free some memory, we adjust the counter by that delta.

每当我们分配或释放一些内存时，我们就根据差值来调整计数器。

memory.c，在reallocate()方法中添加代码：

```
void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
  // 新增部分开始
  vm.bytesAllocated += newSize - oldSize;
  // 新增部分结束
  if (newSize > oldSize) {
```

> When the total crosses the limit, we run the collector.

当总数超过限制时，我们运行回收器。

memory.c，在reallocate()方法中添加代码：

```
    collectGarbage();
#endif
    // 新增部分开始
    if (vm.bytesAllocated > vm.nextGC) {
      collectGarbage();
    }
    // 新增部分结束
  }
```

> Now, finally, our garbage collector actually does something when the user runs a program without our hidden diagnostic flag enabled. The sweep phase frees objects by calling `reallocate()`, which lowers the value of `bytesAllocated`, so after the collection completes, we know how many live bytes remain. We adjust the threshold of the next GC based on that.

现在，终于，即便用户运行一个没有启用隐藏诊断标志的程序时，我们的垃圾回收器实际上也做了一些事情。扫描阶段通过调用`reallocate()`释放对象，这会降低`bytesAllocated`的值，所以在收集完成后，我们知道还有多少活动字节。我们在此基础上调整下一次GC的阈值。

memory.c，在collectGarbage()方法中添加代码：

```
  sweep();
  // 新增部分开始
  vm.nextGC = vm.bytesAllocated * GC_HEAP_GROW_FACTOR;
  // 新增部分结束
#ifdef DEBUG_LOG_GC
```

> The threshold is a multiple of the heap size. This way, as the amount of memory the program uses grows, the threshold moves farther out to limit the total time spent re-traversing the larger live set. Like other numbers in this chapter, the scaling factor is basically arbitrary.

该阈值是堆大小的倍数。这样一来，随着程序使用的内存量的增加长，阈值会向上移动，以限制重新遍历更大的活动集合所花费的总时间。和本章中的其它数字一样，比例因子基本上是任意的。

memory.c，添加代码：

```
#endif
// 新增部分开始
#define GC_HEAP_GROW_FACTOR 2
// 新增部分结束
void* reallocate(void* pointer, size_t oldSize, size_t newSize) {
```

> You'd want to tune this in your implementation once you had some real programs to benchmark it on. Right now, we can at least log some of the statistics that we have. We capture the heap size before the collection.

一旦你有了一些真正的程序来对其进行基准测试，你就需要在实现中对该参数进行调优。现在，我们至少可以记录一些统计数据。我们在回收之前捕获堆的大小。

memory.c，在collectGarbage()方法中添加代码：

```
  printf("-- gc begin\n");
  // 新增部分开始
  size_t before = vm.bytesAllocated;
  // 新增部分结束
#endif
```

> And then print the results at the end.

最后把结果打印出来。

memory.c，在collectGarbage()方法中添加代码：

```
  printf("-- gc end\n");
  // 新增部分开始
  printf("   collected %zu bytes (from %zu to %zu) next at %zu\n",
         before - vm.bytesAllocated, before, vm.bytesAllocated,
         vm.nextGC);
  // 新增部分结束
#endif
```

> This way we can see how much the garbage collector accomplished while it ran.

这样，我们就可以看到垃圾回收器在运行时完成了多少任务。

26.7 Garbage Collection Bugs

26.7 垃圾回收Bug

> In theory, we are all done now. We have a GC. It kicks in periodically, collects what it can, and leaves the rest. If this were a typical textbook, we would wipe the dust from our hands and bask in the soft glow of the flawless marble edifice we have created.

理论上讲，我们现在已经完成了。我们有了一个GC，它周期性启动，回收可以回收的东西，并留下其余的东西。如果这是一本典型的教科书，我们会擦掉手上的灰尘，沉浸在我们所创造的完美无瑕的大理石建筑的柔和光芒中。

> But I aim to teach you not just the theory of programming languages but the sometimes painful reality. I am going to roll over a rotten log and show you the nasty bugs that live under it, and garbage collector bugs really are some of the grossest invertebrates out there.

但是，我的目的不仅仅是教授编程语言的理论，还要教你有时令人痛苦的现实。我要掀开一根烂木头，向你展示生活在下面的讨厌的虫子，垃圾回收器虫真的是世界上最恶心的无脊椎动物之一【译者注：bug双关】。

> The collector's job is to free dead objects and preserve live ones. Mistakes are easy to make in both directions. If the VM fails to free objects that aren't needed, it slowly leaks memory. If it frees an object that is in use, the user's program can access invalid memory. These failures often don't immediately cause a crash, which makes it hard for us to trace backward in time to find the bug.

回收器的工作是释放已死对象并保留活动对象。在这两个方面都很容易出现错误。如果虚拟机不能释放不需要的对象，就会慢慢地泄露内存。如果它释放了一个正在使用的对象，用户的程序就会访问无效的内存。这些故障通常不会立即导致崩溃，这使得我们很难即时追溯以找到错误。

> This is made harder by the fact that we don't know when the collector will run. Any call that eventually allocates some memory is a place in the VM where a collection could happen. It's like musical chairs. At any point, the GC might stop the music. Every single heap-allocated object that we want to keep needs to find a chair quickly—get marked as a root or stored as a reference in some other object—before the sweep phase comes to kick it out of the game.

由于我们不知道回收器何时会运行，这就更加困难了。任何发生内存分配的地方恰好可能是发生回收的地方。这就像抢椅子游戏。在任何时候，GC都可能停止音乐。我们想保留的每一个堆分配对象都需要快速找到一个椅子（被标记为根或作为引用保存在其它对象中），在清除阶段将其踢出游戏之前。

> How is it possible for the VM to use an object later—one that the GC itself doesn't see? How can the VM find it? The most common answer is through a pointer stored in some local variable on the C stack. The GC walks the *VM's* value and CallFrame stacks, but the C stack is hidden to it.

VM怎么可能会在稍后使用一个GC自己都看不到的对象呢？VM如何找到它？最常见的答案是通过存储在C栈中的一些局部变量。GC会遍历VM的值和CallFrame栈，但C的栈对它来说是隐藏的[^20]。

> In previous chapters, we wrote seemingly pointless code that pushed an object onto the VM's value stack, did a little work, and then popped it right back off. Most times, I said this was for the GC's benefit. Now you see why. The code between pushing and popping potentially allocates memory and thus can trigger a GC. We had to make sure the object was on the value stack so that the collector's mark phase would find it and keep it alive.

在前面的章节中，我们编写了一些看似无意义的代码，将一个对象推到VM的值栈上，执行一些操作，然后又把它弹了出来。大多数时候，我说这是为了便于GC。现在你知道为什么了。压入和弹出之间的代码可能会分配内存，因此可能会触发GC。我们必须确保对象在值栈上，这样回收器的标记阶段才能找到它并保持它存活。

> I wrote the entire clox implementation before splitting it into chapters and writing the prose, so I had plenty of time to find all of these corners and flush out most of these bugs. The stress testing code we put in at the beginning of this chapter and a pretty good test suite were very helpful.

在把整个clox拆分为不同章节并编写文章之前，我已经写完了整个clox实现，因此我有足够的时间来找到这些角落，并清除大部分的bug。我们在本章开始时放入的压力测试代码和一个相当好的测试套件都非常有帮助。

> But I fixed only *most* of them. I left a couple in because I want to give you a hint of what it's like to encounter these bugs in the wild. If you enable the stress test flag and run some toy Lox programs, you can probably stumble onto a few. Give it a try and *see if you can fix any yourself*.

但我只修复了其中的大部分。我留下了几个，因为我想给你一些提示，告诉你在野外遇到这些虫子是什么感觉。如果你启用压力测试标志并运行一些玩具Lox程序，你可能会偶然发现一些。试一试，看看你是否能自己解决问题。

> 26.7.1 Adding to the constant table

26.7.1 添加到常量表中

> You are very likely to hit the first bug. The constant table each chunk owns is a dynamic array. When the compiler adds a new constant to the current function's table, that array may need to grow. The constant itself may also be some heap-allocated object like a string or a nested function.

你很有可能会碰到第一个bug。每个块拥有的常量表是一个动态数组。当编译器向当前函数的表中添加一个新常量时，这个数组可能需要增长。常量本身也可以是一些堆分配的对象，如字符串或嵌套函数。

> The new object being added to the constant table is passed to `addConstant()`. At that moment, the object can be found only in the parameter to that function on the C stack. That function appends the object to the constant table. If the table doesn't have enough capacity and needs to grow, it calls `reallocate()`. That in turn triggers a GC, which fails to mark the new constant object and thus sweeps it right before we have a chance to add it to the table. Crash.

待添加到常量表的新对象会被传递给`addConstant()`。此时，该对象只能在C栈上该函数的形参中找到。该函数将对象追加到常量表中。如果表中没有足够的容量并且需要增长，它会调用`reallocate()`。这反过来又触发了一次GC，它无法标记新的常量对象，因此在我们有机会将该对象添加到常量表之前便将其清除了。崩溃。

> The fix, as you've seen in other places, is to push the constant onto the stack temporarily.

正如你在其它地方所看到的，解决方法是将常量临时推入栈中。

chunk.c，在addConstant()方法中添加代码：

```
int addConstant(Chunk* chunk, Value value) {
  // 新增部分开始
  push(value);
  // 新增部分结束
  writeValueArray(&chunk->constants, value);
```

> Once the constant table contains the object, we pop it off the stack.

一旦常量表中有了该对象，我们就将其从栈中弹出。

chunk.c，在addConstant()方法中添加代码：

```
  writeValueArray(&chunk->constants, value);
  // 新增部分开始
  pop();
  // 新增部分结束
  return chunk->constants.count - 1;
```

> When the GC is marking roots, it walks the chain of compilers and marks each of their functions, so the new constant is reachable now. We do need an include to call into the VM from the “chunk” module.

当GC标记根时，它会遍历编译器链并标记它们的每个函数，因此现在新的常量是可达的。我们确实需要引入头文件开从“chunk”模块调用到VM中。

chunk.c，添加代码：

```
#include "memory.h"
// 新增部分开始
#include "vm.h"
// 新增部分结束
void initChunk(Chunk* chunk) {
```

> 26.7.2 Interning strings

26.7.2 驻留字符串

> Here's another similar one. All strings are interned in clox, so whenever we create a new string, we also add it to the intern table. You can see where this is going. Since the string is brand new, it isn't reachable anywhere. And resizing the string pool can trigger a collection. Again, we go ahead and stash the string on the stack first.

下面是另一个类似的例子。所有字符串在clox都是驻留的，因此每当创建一个新的字符串时，我们也会将其添加到驻留表中。你知道将会发生什么。因为字符串是全新的，所以它在任何地方都是不可达的。调整字符串池的大小会触发一次回收。同样，我们先去把字符串藏在栈上。

object.c，在allocateString()方法中添加代码：

```
  string->chars = chars;
  string->hash = hash;
  // 新增部分开始
  push(OBJ_VAL(string));
  // 新增部分结束
  tableSet(&vm.strings, string, NIL_VAL);
```

> And then pop it back off once it's safely nestled in the table.

等它稳稳地进入表中，再把它弹出来。

object.c，在allocateString()方法中添加代码：

```
  tableSet(&vm.strings, string, NIL_VAL);
  // 新增部分开始
  pop();
  // 新增部分结束
  return string;
}
```

> This ensures the string is safe while the table is being resized. Once it survives that, `allocateString()` will return it to some caller which can then take responsibility for ensuring the string is still reachable before the next heap allocation occurs.

这确保了在调整表大小时字符串是安全的。一旦它存活下来，`allocateString()`会把它返回给某个调用者，随后调用者负责确保，在下一次堆分配之前字符串仍然是可达的。

> 26.7.3 Concatenating strings

26.7.3 连接字符串

> One last example: Over in the interpreter, the `OP_ADD` instruction can be used to concatenate two strings. As it does with numbers, it pops the two operands from the stack, computes the result, and pushes that new value back onto the stack. For numbers that's perfectly safe.

最后一个例子：在解释器中，`OP_ADD`指令可以用来连接两个字符串。就像处理数字一样，它会从栈中取出两个操作数，计算结果，并将新值压入栈中。对于数字来说，这是绝对安全的。

> But concatenating two strings requires allocating a new character array on the heap, which can in turn trigger a GC. Since we've already popped the operand strings by that point, they can potentially be missed by the mark phase and get swept away. Instead of popping them off the stack eagerly, we peek them.

但是连接两个字符串需要在堆中分配一个新的字符数组，这又会触发一次GC。因为此时我们已经弹出了操作数字符串，它们可能被标记阶段遗漏并被清除。我们不急于从栈中弹出这些字符串，而只是查看一下它们。

vm.c，在concatenate()方法中，替换2行：

```
static void concatenate() {
  // 新增部分开始
  ObjString* b = AS_STRING(peek(0));
  ObjString* a = AS_STRING(peek(1));
  // 新增部分结束
  int length = a->length + b->length;
```

> That way, they are still hanging out on the stack when we create the result string. Once that's done, we can safely pop them off and replace them with the result.

这样，当我们创建结果字符串时，它们仍然挂在栈上。一旦完成操作，我们就可以放心的将它们弹出，并用结果字符串替换它们。

vm.c，在concatenate()方法中添加代码：

```
  ObjString* result = takeString(chars, length);
  // 新增部分开始
  pop();
  pop();
  // 新增部分结束
  push(OBJ_VAL(result));
```

> Those were all pretty easy, especially because I *showed* you where the fix was. In practice, *finding* them is the hard part. All you see is an object that *should* be there but isn't. It's not like other bugs where you're looking for the code that *causes* some problem. You're looking for the *absence* of code which fails to *prevent* a problem, and that's a much harder search.

这些都很简单，特别是因为我告诉了你解决方法在哪里。实际上，*找到*它们才是困难的部分。你所看到的只是一个*本该*存在但却不存在的对象。它不像其它错误那样，你需要找的是*导致*某些问题的代码。这里你要找的是那些无法防止问题发生的代码*缺失*，而这是一个更困难的搜索。

> But, for now at least, you can rest easy. As far as I know, we've found all of the collection bugs in clox, and now we have a working, robust, self-tuning, mark-sweep garbage collector.

但是，至少现在，你可以放心了。据我所知，我们已经找到了clox中的所有回收错误，现在我们有了一个有效的、强大的、自我调整的标记-清除垃圾回收器。

^3: 在John McCarthy的《Lisp的历史》中，他指出：“一旦我们决定进行垃圾回收，它的实际实现就会被推迟，因为当时只做了玩具性的例子。”我们选择推迟在clox中加入GC，是追随了巨人的脚步。 ^4: 跟踪式垃圾回收器是指任何通过对象引用图来追踪的算法。这与引用计数相反，后者使用不同的策略来追踪可达对象。 ^5: 当然，我们最终也会添加一些辅助函数。 ^6: 更复杂的回收器可能运行在单独的线程上，或者在程序执行过程中定期交错运行——通常是在函数调用边界处或发生后向跳转时。 ^7: 我把这一章安排在这里，特别是因为我们现在有了闭包，它给我们提供了有趣的对象，让垃圾回收器来处理。 ^8: 我说“几乎”是因为有些垃圾回收器是按照对象被访问的顺序来移动对象的，所以遍历顺序决定了哪些对象最终会在内存中相邻。这会影响性能，因为CPU使用位置来决定哪些内存要预加载到缓存中。
即便在遍历顺序很重要的时候，也不清楚哪种顺序是最好的。很难确定对象在未来会以何种顺序被使用，因此GC很难知道哪种顺序有助于提高性能。 ^9: 高级的垃圾回收算法经常为这个抽象概念加入其它颜色。我见过多种深浅不一的灰色，甚至在一些设计中见过紫色。可惜的是，我的黄绿色-紫红色-孔雀石回收器论文没有被接受发表。 ^10: 请注意，在此过程的每一步，都没有黑色节点指向白色节点。这个属性被称为**三色不变性**。变量过程中保持这一不变性，以确保没有任何可达对象被回收。 ^11: 为了更加健壮，我们可以在启动虚拟机时分配一个“雨天基金”内存块。如果灰色栈分配失败，我们就释放这个块并重新尝试。这可能会为我们在堆上提供足够的空间来创建灰色栈，完成GC并释放更多内存。 ^12: 我们可以在`markObject()`中做一个简单的优化，就是不要向灰色栈中添加字符串和本地函数，因为我们知道它们不需要处理。相对地，它们可以从白色直接变黑。
^13: 你可能正在好奇为什么要有`isMarked`字段。别急，朋友。 ^14: 这可能是一个真正的问题。Java并没有驻留*所有*字符串，但它确实驻留了字符串*字面量*。它还提供了向字符串表添加字符串的API。多年以来，该表的容量是固定的，添加到其中的字符串永远无法被删除。如果用户不谨慎使用`String.intern()`，他们可能会耗尽内存导致崩溃。
Ruby多年以来也存在类似的问题，符号（驻留的类似字符串的值）不会被垃圾回收。两者最终都启用了GC来回收这些字符串。 ^15: 嗯，不完全是100%。它仍然将分配的对象放入了一个链表中，所以在设置这些指针时有一些微小的开销。 ^16: 每个条带表示程序的执行，分为运行用户代码的时间和在GC中花费的时间。运行GC的最大单个时间片的大小就是延迟。所有用户代码片的大小相加就是吞吐量。 ^17: 如果每个人代表一个线程，那么一个明显的优化就是让单独的线程进行垃圾回收，提供一个**并发垃圾回收器**。换句话说，在其他人烘焙的时候，雇佣一些洗碗工来清洗。这就是非常复杂的GC工作方式，因为它确实允许烘焙师（工作线程）在几乎没有中断的情况下持续运行用户代码。
但是，协调是必须的。你不会想让洗碗工从面包师手中抢走碗吧！这种协调增加了开销和大量的复杂性。并发回收器速度很快，但要正确实现却很有挑战性。 ^18: 相比之下，**增量式垃圾回收器**可以做一点回收工作，然后运行一些用户代码，然后再做一点回收工作，以此类推。 ^19: 学习垃圾回收器的一个挑战是，在孤立的实验室环境中很难发现最佳实践。除非你在大型的、混乱的真实世界的程序上运行回收器，否则你无法看到它的实际表现。这就像调校一辆拉力赛车——你需要把它带到赛道上。 [^20]: 我们的GC无法在C栈中查找地址，但很多GC可以。保守的垃圾回收器会查看所有内存，包括本机堆栈。这类垃圾回收器中最著名的是**Boehm–Demers–Weiser垃圾回收器**，通常就叫作“Boehm回收器”。（在CS中，成名的捷径是姓氏在字母顺序上靠前，这样就能在排序的名字列表中出现在第一位）
许多精确GC也在C栈中遍历。即便是这些GC，也必须对指向仅存于CPU寄存器中的活动对象的指针加以注意。

习题

1. > The Obj header struct at the top of each object now has three fields: `type`, `isMarked`, and `next`. How much memory do those take up (on your machine)? Can you come up with something more compact? Is there a runtime cost to doing so?

   每个对象顶部的Obj头结构体现在有三个字段：`type`, `isMarked`和`next`。它们（在你的机器上）占用了多少内存？你能想出更紧凑的办法吗？这样做是否有运行时成本？

2. > When the sweep phase traverses a live object, it clears the `isMarked` field to prepare it for the next collection cycle. Can you come up with a more efficient approach?

   当清除阶段遍历某个活动对象时，它会清除`isMarked`字段，以便为下一个回收周期做好准备。你能想出一个更有效的方法吗？

3. > Mark-sweep is only one of a variety of garbage collection algorithms out there. Explore those by replacing or augmenting the current collector with another one. Good candidates to consider are reference counting, Cheney's algorithm, or the Lisp 2 mark-compact algorithm.

   标记-清除只是众多垃圾回收算法中的一种。通过用另一种回收器来替换或增强当前的回收器来探索这些算法。可以考虑引用计数、Cheney算法或Lisp 2标记-压缩算法。

设计笔记：分代回收器

A collector loses throughput if it spends a long time re-visiting objects that are still alive. But it can increase latency if it avoids collecting and accumulates a large pile of garbage to wade through. If only there were some way to tell which objects were likely to be long-lived and which weren't. Then the GC could avoid revisiting the long-lived ones as often and clean up the ephemeral ones more frequently.

It turns out there kind of is. Many years ago, GC researchers gathered metrics on the lifetime of objects in real-world running programs. They tracked every object when it was allocated, and eventually when it was no longer needed, and then graphed out how long objects tended to live.

They discovered something they called the **generational hypothesis**, or the much less tactful term **infant mortality**. Their observation was that most objects are very short-lived but once they survive beyond a certain age, they tend to stick around quite a long time. The longer an object *has* lived, the longer it likely will *continue* to live. This observation is powerful because it gave them a handle on how to partition objects into groups that benefit from frequent collections and those that don't.

They designed a technique called **generational garbage collection**. It works like this: Every time a new object is allocated, it goes into a special, relatively small region of the heap called the "nursery". Since objects tend to die young, the garbage collector is invoked frequently over the objects just in this region.

Nurseries are also usually managed using a copying collector which is faster at allocating and freeing objects than a mark-sweep collector.

Each time the GC runs over the nursery is called a "generation". Any objects that are no longer needed get freed. Those that survive are now considered one generation older, and the GC tracks this for each object. If an object survives a certain number of generations—often just a single collection—it gets *tenured*. At this point, it is copied out of the nursery into a much larger heap region for long-lived objects. The garbage collector runs over that region too, but much less frequently since odds are good that most of those objects will still be alive.

Generational collectors are a beautiful marriage of empirical data—the observation that object lifetimes are *not* evenly distributed—and clever algorithm design that takes advantage of that fact. They're also conceptually quite simple. You can think of one as just two separately tuned GCs and a pretty simple policy for moving objects from one to the other.

如果回收器花费很长时间重新访问仍然活动的对象，则会损失吞吐量。但是，如果它避免了回收并积累了一大堆需要处理的垃圾，就会增加延迟。要是能有某种办法可以告诉我们哪些对象可能是长寿的以及哪些对象不是就好了。这样GC就可以避免频繁地重新访问寿命较长的数据，而更频繁地清理那些短暂寿命短暂的对象。

事实证明，确实如此。许多年前，GC研究人员收集了关于真实运行程序中对象生命周期的指标。他们跟踪了每个对象被分配时，以及它最终不再需要时的情况，然后用图表显示出对象的寿命。

他们发现了一种被称为"**代际假说**"的东西，或者是一个不太委婉的术语"**早夭**"。他们的观察结果是，大多数对象的寿命都很短，但是一旦它们存活超过了一定的年龄，它们往往会存活相当长的时间。一个对象*已经*存活的时间越长，它将*继续*存活的时间就越长。这一观察结果非常有说服力，因为这为他们提供了将对象划分为频繁回收的群体和不频繁回收群体的方法。

他们设计了一种叫作**分代垃圾回收**的技术。它的工作原理是这样的：每次分配一个新对象时，它会进入堆中一个特殊的、相对较小的区域，称为"nursery"（意为托儿所）。由于对象倾向于早夭，所以垃圾回收器会在这个区域中的对象上被频繁调用。

【nursery通常也是要复制回收器进行管理，它在分配和释放对象方面比标记-清除回收器更快。】

GC在nursery的每次运行都被称为"一代"。任何不再需要的对象都会被释放。那些存活下来对象现在被认为老了一代，GC会为每个对象记录这一属性。如果一个对象存活了一定数量的代（通常只是一次回收），它就会被永久保留。此时，将它从nursery中复制处理，放入一个更大的、用于存放 长寿命对象的堆区域。垃圾回收器也会在这个区域内运行，但频率要低得多，因为这些对象中的大部分都很有可能还活着。

分代回收器是经验数据（观察到对象生命周期不是均匀分布的）以及利用这一事实的聪明算法设计的完美结合。它们在概念上也很简单。你可以把它看作是两个单独调优的GC和把对象从一个区域移到另一个区域的一个非常简单的策略。

27.类与实例 Classes and Instances

> Caring too much for objects can destroy you. Only—if you care for a thing enough, it takes on a life of its own, doesn't it? And isn't the whole point of things—beautiful things—that they connect you to some larger beauty?
>
> ——Donna Tartt, *The Goldfinch*

对物品过于关心会毁了你。只是，如果你对一件事物足够关心，它就有了自己的生命，不是吗？而事物——美丽的事物——的全部意义不就是把你和一些更大的美联系起来吗？（唐娜 塔特，《金翅雀》）

> The last area left to implement in clox is object-oriented programming. OOP is a bundle of intertwined features: classes, instances, fields, methods, initializers, and inheritance. Using relatively high-level Java, we packed all that into two chapters. Now that we're coding in C, which feels like building a model of the Eiffel tower out of toothpicks, we'll devote three chapters to covering the same territory. This makes for a leisurely stroll through the implementation. After strenuous chapters like closures and the garbage collector, you have earned a rest. In fact, the book should be easy from here on out.

clox中需要实现的最后一个领域是面向对象编程。OOP是一堆交织在一起的特性：类、实例、字段、方法、初始化式和继承^1。使用相对高级的Java，我们可以把这些内容都装进两章中。现在我们用C语言编写代码，感觉就像用牙签搭建埃菲尔铁塔的模型，我们将用三章的篇幅来涵盖这些内容。这使得我们可以悠闲地漫步在实现中。在经历了闭包和垃圾回收器这样艰苦的章节之后，你赢得了休息的机会。事实上，从这里开始，这本书都是很容易的。

> In this chapter, we cover the first three features: classes, instances, and fields. This is the stateful side of object orientation. Then in the next two chapters, we will hang behavior and code reuse off of those objects.

在本章中，我们会介绍前三个特性：类、实例和字段。这就是面向对象中表现出状态的一面。然后在接下来的两章中，我们会对这些对象挂上行为和代码重用能力。

27.1 Class Objects

27.1 Class对象

> In a class-based object-oriented language, everything begins with classes. They define what sorts of objects exist in the program and are the factories used to produce new instances. Going bottom-up, we'll start with their runtime representation and then hook that into the language.

在一门基于类的面向对象的语言中，一切都从类开始。它们定义了程序中存在什么类型的对象，并且它们也是用来生产新实例的工厂。自下向上，我们将从它们的运行时表示形式开始，然后将其挂接到语言中。

> By this point, we're well-acquainted with the process of adding a new object type to the VM. We start with a struct.

至此，我们已经非常熟悉向VM添加新对象类型的过程了。我们从一个结构体开始。

object.h，在结构体ObjClosure后添加代码：

```
} ObjClosure;
// 新增部分开始
typedef struct {
  Obj obj;
  ObjString* name;
} ObjClass;
// 新增部分结束
ObjClosure* newClosure(ObjFunction* function);
```

> After the Obj header, we store the class's name. This isn't strictly needed for the user's program, but it lets us show the name at runtime for things like stack traces.

在Obj头文件之后，我们存储了类的名称。对于用户的程序来说，这一信息并不是严格需要的，但是它让我们可以在运行时显示名称，例如堆栈跟踪。

> The new type needs a corresponding case in the ObjType enum.

新类型需要在ObjType枚举中有一个对应的项。

object.h，在枚举ObjType中添加代码：

```
typedef enum {
  // 新增部分开始
  OBJ_CLASS,
  // 新增部分结束
  OBJ_CLOSURE,
```

> And that type gets a corresponding pair of macros. First, for testing an object's type:

而该类型会有一组对应的宏。首先，用于测试对象的类型：

object.h，添加代码：

```
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分开始
#define IS_CLASS(value)        isObjType(value, OBJ_CLASS)
// 新增部分结束
#define IS_CLOSURE(value)      isObjType(value, OBJ_CLOSURE)
```

> And then for casting a Value to an ObjClass pointer:

然后是用于将一个Value转换为一个ObjClass指针：

object.h，添加代码：

```
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分开始
#define AS_CLASS(value)        ((ObjClass*)AS_OBJ(value))
// 新增部分结束
#define AS_CLOSURE(value)      ((ObjClosure*)AS_OBJ(value))
```

> The VM creates new class objects using this function:

VM使用这个函数创建新的类对象：

object.h，在结构体ObjClass后添加代码：

```
} ObjClass;
// 新增部分开始
ObjClass* newClass(ObjString* name);
// 新增部分结束
ObjClosure* newClosure(ObjFunction* function);
```

> The implementation lives over here:

实现在这里：

object.c，在allocateObject()方法后添加代码：

```
ObjClass* newClass(ObjString* name) {
  ObjClass* klass = ALLOCATE_OBJ(ObjClass, OBJ_CLASS);
  klass->name = name;
  return klass;
}
```

> Pretty much all boilerplate. It takes in the class's name as a string and stores it. Every time the user declares a new class, the VM will create a new one of these ObjClass structs to represent it.

几乎都是模板代码。它接受并保存字符串形式的类名。每当用户声明一个新类时，VM会创建一个新的ObjClass结构体来表示它^2。

> When the VM no longer needs a class, it frees it like so:

当VM不再需要某个类时，这样释放它：

memory.c，在freeObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_CLASS: {
      FREE(ObjClass, object);
      break;
    }
    // 新增部分结束
    case OBJ_CLOSURE: {
```

> We have a memory manager now, so we also need to support tracing through class objects.

我们现在有一个内存管理器，所以我们也需要支持通过类对象进行跟踪。

memory.c，在blackenObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_CLASS: {
      ObjClass* klass = (ObjClass*)object;
      markObject((Obj*)klass->name);
      break;
    }
    // 新增部分结束
    case OBJ_CLOSURE: {
```

> When the GC reaches a class object, it marks the class's name to keep that string alive too.

当GC到达一个类对象时，它会标记该类的名称，以保持该字符串也能存活。

> The last operation the VM can perform on a class is printing it.

VM可以对类执行的最后一个操作是打印它。

object.c，在printObject()方法中添加代码：

```
  switch (OBJ_TYPE(value)) {
    // 新增部分开始
    case OBJ_CLASS:
      printf("%s", AS_CLASS(value)->name->chars);
      break;
    // 新增部分结束
    case OBJ_CLOSURE:
```

> A class simply says its own name.

类只是简单地说出它的名称。

27.2 Class Declarations

27.2 类声明

> Runtime representation in hand, we are ready to add support for classes to the language. Next, we move into the parser.

有了运行时表示形式，我们就可以向语言中添加对类的支持了。接下来，我们进入语法分析部分。

compiler.c，在declaration()方法中替换1行：

```
static void declaration() {
  // 替换部分开始
  if (match(TOKEN_CLASS)) {
    classDeclaration();
  } else if (match(TOKEN_FUN)) {
  // 替换部分结束
    funDeclaration();
```

> Class declarations are statements, and the parser recognizes one by the leading `class` keyword. The rest of the compilation happens over here:

类声明是语句，解释器通过前面的`class`关键字识别声明语句。剩下部分的编译工作在这里进行：

compiler.c，在function()方法后添加代码：

```
static void classDeclaration() {
  consume(TOKEN_IDENTIFIER, "Expect class name.");
  uint8_t nameConstant = identifierConstant(&parser.previous);
  declareVariable();

  emitBytes(OP_CLASS, nameConstant);
```

```
    defineVariable(nameConstant);

    consume(TOKEN_LEFT_BRACE, "Expect '{' before class body.");
    consume(TOKEN_RIGHT_BRACE, "Expect '}' after class body.");
  }
```

> Immediately after the `class` keyword is the class's name. We take that identifier and add it to the surrounding function's constant table as a string. As you just saw, printing a class shows its name, so the compiler needs to stuff the name string somewhere that the runtime can find. The constant table is the way to do that.

紧跟在`class`关键字之后的是类名。我们将这个标识符作为字符串添加到外围函数的常量表中。正如你刚才看到的，打印一个类会显示它的名称，所以编译器需要把这个名称字符串放在运行时可以找到的地方。常量表就是实现这一目的的方法。

> The class's name is also used to bind the class object to a variable of the same name. So we declare a variable with that identifier right after consuming its token.

类名也被用来将类对象与一个同名变量绑定。因此，我们在使用完它的词法标识后，马上用这个标识符声明一个变量^3。

> Next, we emit a new instruction to actually create the class object at runtime. That instruction takes the constant table index of the class's name as an operand.

接下来我们发出一条新指令，在运行时实际创建类对象。该指令以类名的常量表索引作为操作数。

> After that, but before compiling the body of the class, we define the variable for the class's name. *Declaring* the variable adds it to the scope, but recall from a previous chapter that we can't *use* the variable until it's *defined*. For classes, we define the variable before the body. That way, users can refer to the containing class inside the bodies of its own methods. That's useful for things like factory methods that produce new instances of the class.

在此之后，但是在编译类主体之前，我们使用类名定义变量。*声明*变量会将其添加到作用域中，但请回想一下前一章的内容，在定义变量之前我们不能使用它。对于类，我们在解析主体之前定义变量。这样，用户就可以在类自己的方法主体中引用类本身。这对于产生类的新实例的工厂方法等场景来说是很有用的。

> Finally, we compile the body. We don't have methods yet, so right now it's simply an empty pair of braces. Lox doesn't require fields to be declared in the class, so we're done with the body—and the parser—for now.

最后，我们编译主体。我们现在还没有方法，所以现在它只是一对空的大括号。Lox不要求在类中声明字段，因此我们目前已经完成了主体（和解析器）的工作。

> The compiler is emitting a new instruction, so let's define that.

编译器会发出一条新指令，所以我们来定义它。

chunk.h，在枚举OpCode中添加代码：

```
  OP_RETURN,
  // 新增部分开始
  OP_CLASS,
  // 新增部分结束
} OpCode;
```

> And add it to the disassembler:

然后将其添加到反汇编程序中：

debug.c，在disassembleInstruction()方法中添加代码：

```
    case OP_RETURN:
      return simpleInstruction("OP_RETURN", offset);
    // 新增部分开始
    case OP_CLASS:
      return constantInstruction("OP_CLASS", chunk, offset);
    // 新增部分结束
    default:
```

> For such a large-seeming feature, the interpreter support is minimal.

对于这样一个看起来很大的特性，解释器支持是最小的。

vm.c，在run()方法中添加代码：

```
        break;
      }
      // 新增部分开始
      case OP_CLASS:
        push(OBJ_VAL(newClass(READ_STRING())));
        break;
      // 新增部分结束
    }
```

> We load the string for the class's name from the constant table and pass that to `newClass()`. That creates a new class object with the given name. We push that onto the stack and we're good. If the class is bound to a global variable, then the compiler's call to `defineVariable()` will emit code to store that object from the stack into the global variable table. Otherwise, it's right where it needs to be on the stack for a new local variable.

我们从常量表中加载类名的字符串，并将其传递给`newClass()`。这将创建一个具有给定名称的新类对象。我们把它推入栈中就可以了。如果该类被绑定到一个全局变量上，那么编译器对`defineVariable()`的调用就会生成字节码，将该对象从栈中存储到全局变量表。否则，它就正好位于栈中新的局部变量所在的位置^4。

> There you have it, our VM supports classes now. You can run this:

好了，我们的虚拟机现在支持类了。你可以运行这段代码：

```
class Brioche {}
print Brioche;
```

> Unfortunately, printing is about *all* you can do with classes, so next is making them more useful.

不幸的是，打印是你对类所能做的全部事情，所以接下来是让它们更有用。

27.3 Instances of Classes

27.3 类的实例

> Classes serve two main purposes in a language:

类在一门语言中主要有两个作用：

> - **They are how you create new instances.** Sometimes this involves a `new` keyword, other times it's a method call on the class object, but you usually mention the class by name *somehow* to get a new instance.
> - **They contain methods.** These define how all instances of the class behave.

- **它们是你创建新实例的方式**。有时这会涉及到`new`关键字，有时则是对类对象的方法调用，但是你通常会以某种方式通过类的名称来获得一个新的实例。
- **它们包含方法**。这些方法定义了类的所有实例的行为方式。

> We won't get to methods until the next chapter, so for now we will only worry about the first part. Before classes can create instances, we need a representation for them.

我们要到下一章才会讲到方法，所以我们现在只关心第一部分。在类能够创建实例之前，我们需要为它们提供一个表示形式。

object.h，在结构体ObjClass后添加代码：

```
} ObjClass;
// 新增部分开始
typedef struct {
  Obj obj;
  ObjClass* klass;
  Table fields;
} ObjInstance;
// 新增部分结束
ObjClass* newClass(ObjString* name);
```

> Instances know their class—each instance has a pointer to the class that it is an instance of. We won't use this much in this chapter, but it will become critical when we add methods.

实例知道它们的类——每个实例都有一个指向它所属类的指针。在本章中我们不会过多地使用它，但是等我们添加方法时，它将会变得非常重要。

> More important to this chapter is how instances store their state. Lox lets users freely add fields to an instance at runtime. This means we need a storage mechanism that can grow. We could use a dynamic array, but we also want to look up fields by name as quickly as possible. There's a data structure that's just perfect for quickly accessing a set of values by name and—even more conveniently—we've already implemented it. Each instance stores its fields using a hash table.

对本章来说，更重要的是实例如何存储它们的状态。Lox允许用户在运行时自由地向实例中添加字段。这意味着我们需要一种可以增长的存储机制。我们可以使用动态数组，但我们也希望尽可能快地按名称查找字段。有一种数据结构非常适合于按名称快速访问一组值——甚至更方便的是——我们已经实现了它。每个实例都使用哈希表来存储其字段^5。

> We only need to add an include, and we've got it.

我们只需要添加一个头文件引入，就可以了。

object.h，添加代码：

```
#include "chunk.h"
// 新增部分开始
#include "table.h"
// 新增部分结束
#include "value.h"
```

> This new struct gets a new object type.

新结构体有新的对象类型。

object.h，在枚举ObjType中添加代码：

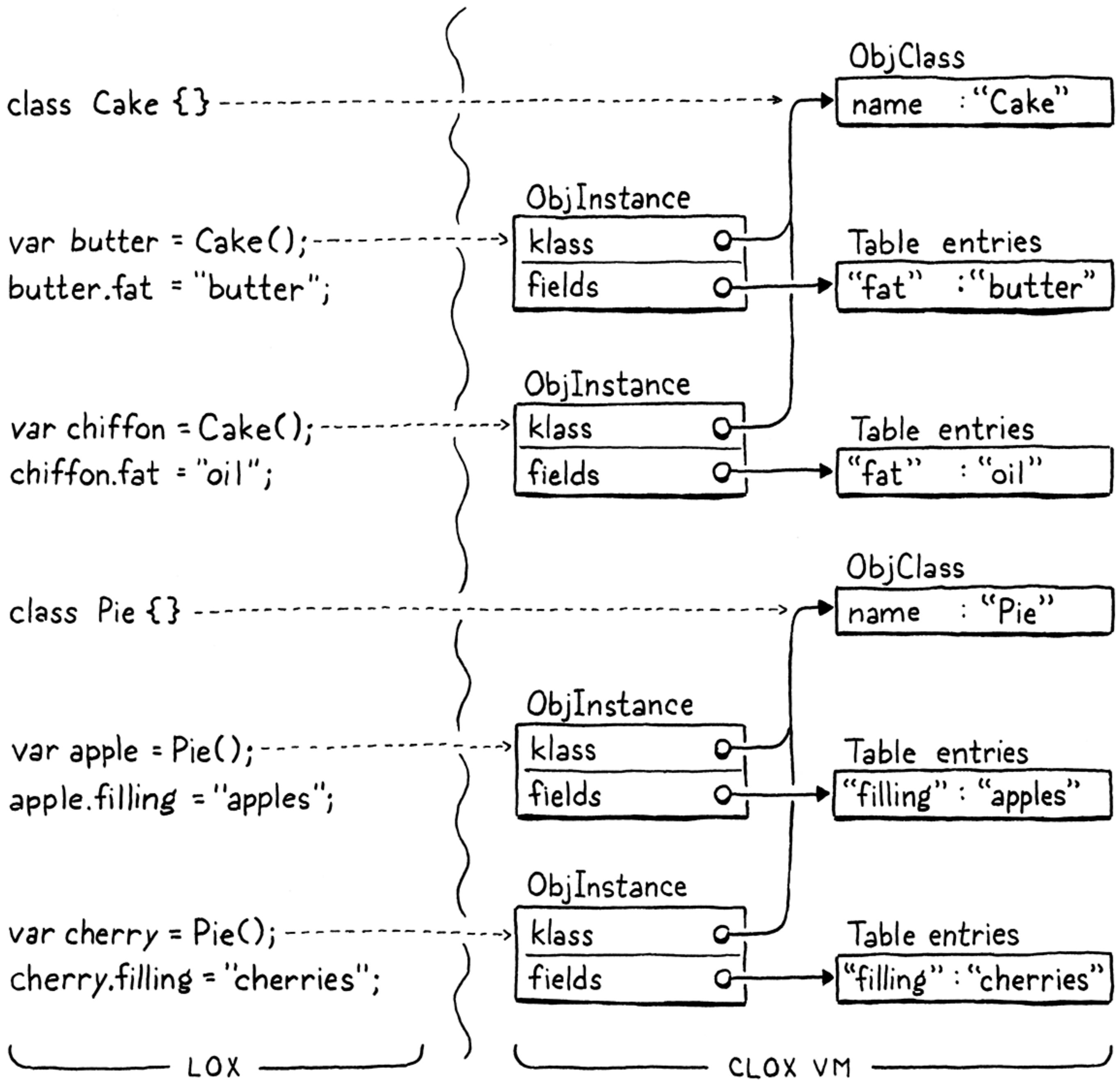
```
  OBJ_FUNCTION,
  // 新增部分开始
  OBJ_INSTANCE,
  // 新增部分结束
  OBJ_NATIVE,
```

> I want to slow down a bit here because the Lox *language's* notion of "type" and the VM *implementation's* notion of "type" brush against each other in ways that can be confusing. Inside the C code that makes clox, there are a number of different types of Obj—ObjString, ObjClosure, etc. Each has its own internal representation and semantics.

这里我想放慢一点速度，因为Lox*语言*中的"type"概念和*虚拟机实现*中的"type"概念是相互抵触的，可能会造成混淆。在生成clox 的C语言代码中，有许多不同类型的Obj——ObjString、ObjClosure等等。每个都有自己的内部表示和语义。

> In the Lox *language*, users can define their own classes—say Cake and Pie—and then create instances of those classes. From the user's perspective, an instance of Cake is a different type of object than an instance of Pie. But, from the VM's perspective, every class the user defines is simply another value of type ObjClass. Likewise, each instance in the user's program, no matter what class it is an instance of, is an ObjInstance. That one VM object type covers instances of all classes. The two worlds map to each other something like this:

在Lox*语言*中，用户可以定义自己的类——比如Cake和Pie——然后创建这些类的实例。从用户的角度来看，Cake实例与Pie实例是不同类型的对象。但是，从虚拟机的角度来看，用户定义的每个类都只是另一个ObjClass类型的值。同样，用户程序中的每个实例，无论它是什么类的实例，都是一个ObjInstance。这一虚拟机对象类型涵盖了所有类的实例。这两个世界之间的映射是这样的：



> Got it? OK, back to the implementation. We also get our usual macros.

明白了吗？好了，回到实现中。我们新增了一些熟悉的宏。

object.h，添加代码：

```
#define IS_FUNCTION(value)     isObjType(value, OBJ_FUNCTION)
// 新增部分开始
#define IS_INSTANCE(value)     isObjType(value, OBJ_INSTANCE)
// 新增部分结束
#define IS_NATIVE(value)       isObjType(value, OBJ_NATIVE)
```

> And:

以及：

object.h，添加代码：

```
#define AS_FUNCTION(value)     ((ObjFunction*)AS_OBJ(value))
// 新增部分开始
#define AS_INSTANCE(value)     ((ObjInstance*)AS_OBJ(value))
// 新增部分结束
#define AS_NATIVE(value) \
```

> Since fields are added after the instance is created, the “constructor” function only needs to know the class.

因为字段是在实例创建之后添加的，所以“构造器”函数只需要知道类。

object.h，在newFunction()方法后添加代码：

```
ObjFunction* newFunction();
// 新增部分开始
ObjInstance* newInstance(ObjClass* klass);
// 新增部分结束
ObjNative* newNative(NativeFn function);
```

> We implement that function here:

我们在这里实现该函数：

object.c，在newFunction()方法后添加代码：

```
ObjInstance* newInstance(ObjClass* klass) {
  ObjInstance* instance = ALLOCATE_OBJ(ObjInstance, OBJ_INSTANCE);
  instance->klass = klass;
  initTable(&instance->fields);
  return instance;
}
```

> We store a reference to the instance's class. Then we initialize the field table to an empty hash table. A new baby object is born!

我们存储了对实例的类的引用。然后我们将字段表初始化为一个空的哈希表。一个全新的对象诞生了！

> At the sadder end of the instance's lifespan, it gets freed.

在实例生命周期的最后阶段，它被释放了。

memory.c，在freeObject()方法中添加代码：

```
      FREE(ObjFunction, object);
      break;
    }
    // 新增部分开始
    case OBJ_INSTANCE: {
      ObjInstance* instance = (ObjInstance*)object;
      freeTable(&instance->fields);
      FREE(ObjInstance, object);
      break;
    }
    // 新增部分结束
    case OBJ_NATIVE:
```

> The instance owns its field table so when freeing the instance, we also free the table. We don't explicitly free the entries *in* the table, because there may be other references to those objects. The garbage collector will take care of those for us. Here we free only the entry array of the table itself.

实例拥有自己的字段表，所以当释放实例时，我们也会释放该表。我们没有显式地释放表中的条目，因为可能存在对这些对象的其它引用。垃圾回收器会帮我们处理这些问题。这里我们只释放表本身的条目数组。

> Speaking of the garbage collector, it needs support for tracing through instances.

说到垃圾回收，它需要支持通过实例进行跟踪。

memory.c，在blackenObject()方法中添加代码：

```
      markArray(&function->chunk.constants);
      break;
    }
    // 新增部分开始
    case OBJ_INSTANCE: {
      ObjInstance* instance = (ObjInstance*)object;
      markObject((Obj*)instance->klass);
      markTable(&instance->fields);
      break;
    }
    // 新增部分结束
    case OBJ_UPVALUE:
```

> If the instance is alive, we need to keep its class around. Also, we need to keep every object referenced by the instance's fields. Most live objects that are not roots are reachable because some instance refers to the object in a field. Fortunately, we already have a nice `markTable()` function to make tracing them easy.

如果这个实例是活动的，我们需要保留它的类。此外，我们还需要保留每个被实例字段引用的对象。大多数不是根的活动对象都是可达的，因为某些实例会在某个字段中引用该对象。幸运的是，我们已经有了一个很好的`markTable()`函数，可以轻松地跟踪它们。

> Less critical but still important is printing.

不太关键但仍然重要的是打印。

object.c，在printObject()方法中添加代码：

```
      break;
    // 新增部分开始
    case OBJ_INSTANCE:
      printf("%s instance",
             AS_INSTANCE(value)->klass->name->chars);
      break;
    // 新增部分结束
    case OBJ_NATIVE:
```

> An instance prints its name followed by "instance". (The "instance" part is mainly so that classes and instances don't print the same.)

实例会打印它的名称，并在后面加上"instance"^6。（"instance"部分主要是为了使类和实例不会打印出相同的内容）

> The real fun happens over in the interpreter. Lox has no special `new` keyword. The way to create an instance of a class is to invoke the class itself as if it were a function. The runtime already supports function calls, and it checks the type of object being called to make sure the user doesn't try to invoke a number or other invalid type.

真正有趣的部分在解释器中，Lox没有特殊的`new`关键字。创建类实例的方法是调用类本身，就像调用函数一样。运行时已经支持函数调用，它会检查被调用对象的类型，以确保用户不会试图调用数字或其它无效类型。

> We extend that runtime checking with a new case.

我们用一个新的case分支来扩展运行时的检查。

vm.c，在callValue()方法中添加代码：

```
    switch (OBJ_TYPE(callee)) {
      // 新增部分开始
      case OBJ_CLASS: {
        ObjClass* klass = AS_CLASS(callee);
        vm.stackTop[-argCount - 1] = OBJ_VAL(newInstance(klass));
        return true;
      }
      // 新增部分结束
      case OBJ_CLOSURE:
```

> If the value being called—the object that results when evaluating the expression to the left of the opening parenthesis—is a class, then we treat it as a constructor call. We create a new instance of the called class and store the result on the stack.

如果被调用的值（在左括号左边的表达式求值得到的对象）是一个类，则将其视为一个构造函数调用。我们创建一个被调用类的新实例，并将结果存储在栈中^7。

> We're one step farther. Now we can define classes and create instances of them.

我们又前进了一步。现在我们可以定义类并创建它们的实例了。

```
class Brioche {}
print Brioche();
```

> Note the parentheses after `Brioche` on the second line now. This prints "Brioche instance".

注意第二行`Brioche`后面的括号。这里会打印"Brioche instance"。

> ## 27.4 Get and Set Expressions

27.4 Get和SET表达式

> Our object representation for instances can already store state, so all that remains is exposing that functionality to the user. Fields are accessed and modified using get and set expressions. Not one to break with tradition, Lox uses the classic "dot" syntax:

实例的对象表示形式已经可以存储状态了，所以剩下的就是把这个功能暴露给用户。字段是使用get和set表达式进行访问和修改的。Lox并不喜欢打破传统，这里也沿用了经典的"点"语法：

```
eclair.filling = "pastry creme";
print eclair.filling;
```

> The period—full stop for my English friends—works sort of like an infix operator. There is an expression to the left that is evaluated first and produces an instance. After that is the `.` followed by a field name. Since there is a preceding operand, we hook this into the parse table as an infix expression.

句号——对英国朋友来说是句号——其作用有点像一个中缀运算符^8。左边有一个表达式，首先被求值并产生一个实例。之后是`.`后跟一个字段名称。由于前面有一个操作数，我们将其作为中缀表达式放到解析表中。

compiler.c，替换1行：

```
  [TOKEN_COMMA]           = {NULL,     NULL,    PREC_NONE},
  // 替换部分开始
  [TOKEN_DOT]             = {NULL,     dot,     PREC_CALL},
  // 替换部分结束
  [TOKEN_MINUS]           = {unary,    binary, PREC_TERM},
```

> As in other languages, the `.` operator binds tightly, with precedence as high as the parentheses in a function call. After the parser consumes the dot token, it dispatches to a new parse function.

和其它语言一样，`.`操作符绑定紧密，其优先级和函数调用中的括号一样高。解析器消费了点标识之后，会分发给一个新的解析函数。

compiler.c，在call()方法后添加代码：

```
static void dot(bool canAssign) {
  consume(TOKEN_IDENTIFIER, "Expect property name after '.'.");
  uint8_t name = identifierConstant(&parser.previous);

  if (canAssign && match(TOKEN_EQUAL)) {
    expression();
    emitBytes(OP_SET_PROPERTY, name);
  } else {
    emitBytes(OP_GET_PROPERTY, name);
  }
}
```

> The parser expects to find a property name immediately after the dot. We load that token's lexeme into the constant table as a string so that the name is available at runtime.

解析器希望在点运算符后面立即找到一个属性名称^9。我们将该词法标识的词素作为字符串加载到常量表中，这样该名称在运行时就是可用的。

> We have two new expression forms—getters and setters—that this one function handles. If we see an equals sign after the field name, it must be a set expression that is assigning to a field. But we don't *always* allow an equals sign after the field to be compiled. Consider:

我们将两种新的表达式形式——getter和setter——都交由这一个函数处理。如果我们看到字段名称后有一个等号，那么它一定是一个赋值给字段的set表达式。但我们并不总是允许编译字段后面的等号。考虑一下：

```
a + b.c = 3
```

> This is syntactically invalid according to Lox's grammar, which means our Lox implementation is obligated to detect and report the error. If `dot()` silently parsed the `= 3` part, we would incorrectly interpret the code as if the user had written:

根据Lox的文法，这在语法上是无效的，这意味着我们的Lox实现有义务检测和报告这个错误。如果`dot()`默默地解析`=3`的部分，我们就会错误地解释代码，就像用户写的是：

```
a + (b.c = 3)
```

> The problem is that the `=` side of a set expression has much lower precedence than the `.` part. The parser may call `dot()` in a context that is too high precedence to permit a setter to appear. To avoid incorrectly allowing that, we parse and compile the equals part only when `canAssign` is true. If an equals token appears when `canAssign` is false, `dot()` leaves it alone and returns. In that case, the compiler will eventually unwind up to `parsePrecedence()`, which stops at the unexpected `=` still sitting as the next token and reports an error.

问题是，set表达式中的`=`侧优先级远低于`.`部分。解析器有可能会在一个优先级高到不允许出现setter的上下文中调用`dot()`。为了避免错误地允许这种情况，我们只有在`canAssign`为true时才去解析和编译等号部分。如果在`canAssign`为false时出现等号标识，`dot()`会保留它并返回。在这种情况下，编译器最终会进入`parsePrecedence()`，而该方法会在非预期的`=`（仍然作为下一个标识）处停止，并报告一个错误。

> If we find an `=` in a context where it *is* allowed, then we compile the expression that follows. After that, we emit a new `OP_SET_PROPERTY` instruction. That takes a single operand for the index of the property name in the constant table. If we didn't compile a set expression, we assume it's a getter and emit an `OP_GET_PROPERTY` instruction, which also takes an operand for the property name.

如果我们在允许使用等号的上下文中找到`=`，则编译后面的表达式。之后，我们发出一条新的`OP_SET_PROPERTY`指令^10。这条指令接受一个操作数，作为属性名称在常量表中的索引。如果我们没有编译set表达式，就假定它是getter，并发出一条`OP_GET_PROPERTY`指令，它也接受一个操作数作为属性名。

> Now is a good time to define these two new instructions.

现在是定义这两条新指令的好时机。

chunk.h，在枚举OpCode中添加代码：

```
  OP_SET_UPVALUE,
  // 新增部分开始
  OP_GET_PROPERTY,
  OP_SET_PROPERTY,
  // 新增部分结束
  OP_EQUAL,
```

> And add support for disassembling them:

并在反汇编程序中为它们添加支持：

debug.c，在disassembleInstruction()方法中添加代码：

```
      return byteInstruction("OP_SET_UPVALUE", chunk, offset);
    // 新增部分开始
    case OP_GET_PROPERTY:
      return constantInstruction("OP_GET_PROPERTY", chunk, offset);
    case OP_SET_PROPERTY:
      return constantInstruction("OP_SET_PROPERTY", chunk, offset);
    // 新增部分结束
    case OP_EQUAL:
```

> ### 27.4.1 Interpreting getter and setter expressions

27.4.1 解释getter和setter表达式

> Sliding over to the runtime, we'll start with get expressions since those are a little simpler.

进入运行时，我们从获取表达式开始，因为它们更简单一些。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_GET_PROPERTY: {
        ObjInstance* instance = AS_INSTANCE(peek(0));
        ObjString* name = READ_STRING();

        Value value;
        if (tableGet(&instance->fields, name, &value)) {
          pop(); // Instance.
          push(value);
          break;
        }
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> When the interpreter reaches this instruction, the expression to the left of the dot has already been executed and the resulting instance is on top of the stack. We read the field name from the constant pool and look it up in the instance's field table. If the hash table contains an entry with that name, we pop the instance and push the entry's value as the result.

当解释器到达这条指令时，点左边的表达式已经被执行，得到的实例就在栈顶。我们从常量池中读取字段名，并在实例的字段表中查找该名称。如果哈希表中包含具有该名称的条目，我们就弹出实例，并将该条目的值作为结果压入栈。

> Of course, the field might not exist. In Lox, we've defined that to be a runtime error. So we add a check for that and abort if it happens.

当然，这个字段可能不存在。在Lox中，我们将其定义为运行时错误。所以我们添加了一个检查，如果发生这种情况就中止。

vm.c，在run()方法中添加代码：

```
          push(value);
          break;
        }
        // 新增部分开始
        runtimeError("Undefined property '%s'.", name->chars);
        return INTERPRET_RUNTIME_ERROR;
        // 新增部分结束
      }
      case OP_EQUAL: {
```

> There is another failure mode to handle which you've probably noticed. The above code assumes the expression to the left of the dot did evaluate to an ObjInstance. But there's nothing preventing a user from writing this:

你可能已经注意到了，还有另一种需要处理的失败模式。上面的代码中假定了点左边的表达式计算结果确实是一个ObjInstance。但是没有什么可以阻止用户这样写：

```
var obj = "not an instance";
print obj.field;
```

> The user's program is wrong, but the VM still has to handle it with some grace. Right now, it will misinterpret the bits of the ObjString as an ObjInstance and, I don't know, catch on fire or something definitely not graceful.

用户的程序是错误的，但是虚拟机仍然需要以某种优雅的方式来处理它。现在，它会把ObjString 数据误认为是一个ObjInstance ，并且，我不确定，代码起火或发生其它事情绝对是不优雅的。

> In Lox, only instances are allowed to have fields. You can't stuff a field onto a string or number. So we need to check that the value is an instance before accessing any fields on it.

在Lox中，只有实例才允许有字段。你不能把字段塞到字符串或数字中。因此，在访问某个值上的任何字段之前，检查该值是否是一个实例^11。

vm.c，在run()方法中添加代码：

```
      case OP_GET_PROPERTY: {
        // 新增部分开始
        if (!IS_INSTANCE(peek(0))) {
          runtimeError("Only instances have properties.");
          return INTERPRET_RUNTIME_ERROR;
        }
        // 新增部分结束
        ObjInstance* instance = AS_INSTANCE(peek(0));
```

> If the value on the stack isn't an instance, we report a runtime error and safely exit.

如果栈中的值不是实例，则报告一个运行时错误并安全退出。

> Of course, get expressions are not very useful when no instances have any fields. For that we need setters.

当然，如果实例没有任何字段，get表达式就不太有用了。因此，我们需要setter。

vm.c，在run()方法中添加代码：

```
      return INTERPRET_RUNTIME_ERROR;
    }
    // 新增部分开始
    case OP_SET_PROPERTY: {
      ObjInstance* instance = AS_INSTANCE(peek(1));
      tableSet(&instance->fields, READ_STRING(), peek(0));
      Value value = pop();
      pop();
      push(value);
      break;
    }
    // 新增部分结束
    case OP_EQUAL: {
```

> This is a little more complex than `OP_GET_PROPERTY`. When this executes, the top of the stack has the instance whose field is being set and above that, the value to be stored. Like before, we read the instruction's operand and find the field name string. Using that, we store the value on top of the stack into the instance's field table.

这比`OP_GET_PROPERTY`要复杂一些。当执行此指令时，栈顶有待设置字段的实例，在该实例之上有要存储的值。与前面一样，我们读取指令的操作数，并查找字段名称字符串。使用该方法，我们将栈顶的值存储到实例的字段表中。

> After that is a little stack juggling. We pop the stored value off, then pop the instance, and finally push the value back on. In other words, we remove the *second* element from the stack while leaving the top alone. A setter is itself an expression whose result is the assigned value, so we need to leave that value on the stack. Here's what I mean:

在那之后是一些栈技巧。我们将存储的值弹出，然后弹出实例，最后再把值压回栈中。换句话说，我们从栈中删除第二个元素，而保留最上面的元素。setter本身是一个表达式，其结果就是所赋的值，所以我们需要将值保留在栈上。我的意思是^12：

```
class Toast {}
var toast = Toast();
print toast.jam = "grape"; // Prints "grape".
```

> Unlike when reading a field, we don't need to worry about the hash table not containing the field. A setter implicitly creates the field if needed. We do need to handle the user incorrectly trying to store a field on a value that isn't an instance.

与读取字段不同，我们不需要担心哈希表中不包含该字段。如果需要的话，setter会隐式地创建这个字段。我们确实需要处理用户不正确地试图在非实例的值上存储字段的情况。

vm.c，在run()方法中添加代码：

```
      case OP_SET_PROPERTY: {
        // 新增部分开始
        if (!IS_INSTANCE(peek(1))) {
          runtimeError("Only instances have fields.");
          return INTERPRET_RUNTIME_ERROR;
        }
        // 新增部分结束
        ObjInstance* instance = AS_INSTANCE(peek(1));
```

> Exactly like with get expressions, we check the value's type and report a runtime error if it's invalid. And, with that, the stateful side of Lox's support for object-oriented programming is in place. Give it a try:

就像get表达式一样，我们检查值的类型，如果无效就报告一个运行时错误。这样一来，Lox对面向对象编程中有状态部分的支持就到位了。试一试：

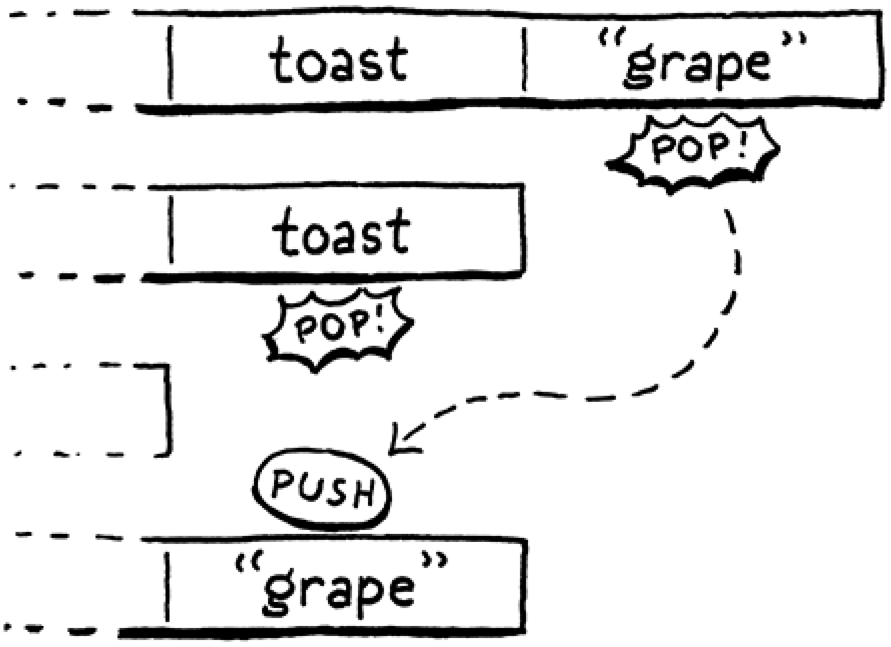
```
class Pair {}

var pair = Pair();
pair.first = 1;
pair.second = 2;
print pair.first + pair.second; // 3.
```

> This doesn't really feel very *object*-oriented. It's more like a strange, dynamically typed variant of C where objects are loose struct-like bags of data. Sort of a dynamic procedural language. But this is a big step in expressiveness. Our Lox implementation now lets users freely aggregate data into bigger units. In the next chapter, we will breathe life into those inert blobs.

这感觉不太面向对象。它更像是一种奇怪的、动态类型的C语言变体，其中的对象是松散的类似结构体的数据包。有点像动态过程化语言。但这是表达能力的一大进步。我们的Lox实现现在允许用户自由地将数据聚合成更大的单元。在下一章中，我们将为这些迟缓的数据注入活力。

^2: 我将变量命名为“klass”，不仅仅是为了给虚拟机一种古怪的幼儿园的"Kidz Korner "感觉。它使得clox更容易被编译为C++，而C++中“class”是一个保留字。 ^3: 我们可以让类声明成为表达式而不是语句——比较它们本质上是一个产生值的字面量。然后用户必须自己显式地将类绑定到一个变量，比如：`var Pie = class {}`。这有点像lambda函数，但只是针对类的。但由于我们通常希望类被命名，所以将其视为声明是有意义的。 ^4: “局部（Local）”类——在函数或块主体中声明的类，是一个不寻常的概念。许多语言根本不允许这一特性。但由于Lox是一种动态类型脚本语言，它会对程序的顶层代码和函数以及块的主体进行统一处理。类只是另一种声明，既然你可以在块中声明变量和函数，那你也可以在块中声明类。 ^5: 能够在运行时自由地向对象添加字段，是大多数动态语言和静态语言之间的一个很大的实际区别。静态类型语言通常要求显式声明字段。这样，编译器就确切知道每个实例有哪些字段。它可以利用这一点来确定每个实例所需的精确内存量，以及每个字段在内存中的偏移量。
在Lox和其它动态语言中，访问字段通常是一次哈希表查询。常量时间复杂度，但仍然是相当重的。在C++这样的语言中，访问一个字段就像对指针偏移一个整数常量一样快。 ^6: 大多数面向对象的语言允许类定义某种形式的`toString()`方法，让该类指定如何将其实例转换为字符串并打印出来。如果Lox不是一门玩具语言，我也想要支持它。 ^7: 我们暂时忽略传递给调用的所有参数。在下一章添加对初始化器的支持时，我们会重新审视这一段代码。 ^8: 我说“有点”是因为`.`右边的不是表达式，而是一个标识符，其语义由get或set表达式本身来处理。它实际上更接近于一个后缀表达式。 ^9: 编译器在这里使用“属性（property）”而不是“字段（field）”，因为，请记住，Lox还允许你使用点语法来访问一个方法而不调用它。“属性”是一个通用术语，我们用来指代可以在实例上访问的任何命名实体。字段是基于实例状态的属性子集。 ^10: 你不能设置非字段属性，所以我认为这个指令本该是`OP_SET_FIELD`，但是我认为它与get指令一致看起来更漂亮。 ^11: Lox *可以*支持向其它类型的值中添加字段。这是我们的语言，我们可以做我们想做的。但这可能是个坏主意。它大大增加了实现的复杂性，从而损害了性能——例如，字符串驻留变得更加困难。
此外，它还引起了关于数值的相等和同一性的复杂语义问题。如果我给数字`3`附加一个字段，那么`1+2`的结果也有这个字段吗？如果是的话，实现上如何跟踪它？如果不是，这两个结果中的“3”仍然被认为是相等的吗？ ^12:

栈的操作是这样的：



习题

1. > Trying to access a non-existent field on an object immediately aborts the entire VM. The user has no way to recover from this runtime error, nor is there any way to see if a field exists *before* trying to access it. It's up to the user to ensure on their own that only valid fields are read.
   >
   > How do other dynamically typed languages handle missing fields? What do you think Lox should do? Implement your solution.

   试图访问一个对象上不存在的字段会立即中止整个虚拟机。用户没有办法从这个运行时错误中恢复过来，也没有办法在试图访问一个字段之前看它是否存在。需要由用户自己来确保只读取有效字段。

   其它动态类型语言是如何处理缺少字段的？你认为Lox应该怎么做？实现你的解决方案。

2. > Fields are accessed at runtime by their *string* name. But that name must always appear directly in the source code as an *identifier token*. A user program cannot imperatively build a string value and then use that as the name of a field. Do you think they should be able to? Devise a language feature that enables that and implement it.

   字段在运行时是通过它们的*字符串*名称来访问的。但是该名称必须总是作为标识符直接出现在源代码中。用户程序不能命令式地构建字符串值，然后将其用作字段名。你认为应该这样做吗？那就设计一种语言特性来实现它。

3. > Conversely, Lox offers no way to *remove* a field from an instance. You can set a field's value to `nil`, but the entry in the hash table is still there. How do other languages handle this? Choose and implement a strategy for Lox.

   反过来说，Lox没有提供从实例中*删除*字段的方法。你可以将一个字段的值设置为`nil`，但哈希表中的条目仍然存在。其它语言如何处理这个问题？为Lox选择一个策略并实现。

4. > Because fields are accessed by name at runtime, working with instance state is slow. It's technically a constant-time operation—thanks, hash tables—but the constant factors are

> relatively large. This is a major component of why dynamic languages are slower than statically typed ones.
>
> How do sophisticated implementations of dynamically typed languages cope with and optimize this?

因为字段在运行时是按照名称访问的，所以对实例状态的操作是很慢的。从技术上讲，这是一个常量时间的操作（感谢哈希表），但是常量因子比较大。这就是动态语言比静态语言慢的一个主要原因。

动态类型语言的复杂实现是如何应对和优化这一问题的？

28.方法和初始化器 Methods and Initializers

> When you are on the dancefloor, there is nothing to do but dance.
>
> —— Umberto Eco, *The Mysterious Flame of Queen Loana*

当你在舞池里时，除了跳舞，别无选择。（翁贝托·艾柯，《洛安娜女王的神秘火焰》）

> It is time for our virtual machine to bring its nascent objects to life with behavior. That means methods and method calls. And, since they are a special kind of method, initializers too.

对于我们的虚拟机来说，现在是时候通过赋予行为的方式为新生对象赋予生命了。也就是方法和方法调用。而且由于初始化器同样也属于这种特殊的方法，所以也要予以考虑。

> All of this is familiar territory from our previous jlox interpreter. What's new in this second trip is an important optimization we'll implement to make method calls over seven times faster than our baseline performance. But before we get to that fun, we gotta get the basic stuff working.

所有这些都是我们以前的jlox解释器中所熟悉的领域。第二次旅行中的新内容是我们将实现一个重要的优化，使方法调用的速度比基线性能快7倍以上。但在此之前，我们得先把基本的东西弄好。

> ## 28.1 Method Declarations

28.1 方法声明

> We can't optimize method calls before we have method calls, and we can't call methods without having methods to call, so we'll start with declarations.

没有方法调用，我们就无法优化方法调用，而没有可供调用的方法，我们就无法调用方法，因此我们从声明开始。

> ### 28.1.1 Representing methods

28.1.1 表示方法

> We usually start in the compiler, but let's knock the object model out first this time. The runtime representation for methods in clox is similar to that of jlox. Each class stores a hash table of methods. Keys are method names, and each value is an ObjClosure for the body of the method.

我们通常从编译器开始，但这次让我们先搞定对象模型。clox中方法的运行时表示形式与jlox相似。每个类都存储了一个方法的哈希表。键是方法名，每个值都是方法主体对应的ObjClosure。

object.h，在结构体ObjClass中添加代码：

```
typedef struct {
  Obj obj;
  ObjString* name;
  // 新增部分开始
  Table methods;
  // 新增部分结束
} ObjClass;
```

> A brand new class begins with an empty method table.

一个全新的类初始时得到的是空方法表。

object.c，在newClass()方法中添加代码：

```
  klass->name = name;
  // 新增部分开始
  initTable(&klass->methods);
  // 新增部分结束
  return klass;
```

> The ObjClass struct owns the memory for this table, so when the memory manager deallocates a class, the table should be freed too.

ObjClass 结构体拥有该表的内存，因此当内存管理器释放某个类时，该表也应该被释放。

memory.c，在freeObject()方法中添加代码：

```
    case OBJ_CLASS: {
      // 新增部分开始
      ObjClass* klass = (ObjClass*)object;
      freeTable(&klass->methods);
      // 新增部分结束
      FREE(ObjClass, object);
```

> Speaking of memory managers, the GC needs to trace through classes into the method table. If a class is still reachable (likely through some instance), then all of its methods certainly need to stick around too.

说到内存管理器，GC需要通过类追踪到方法表。如果某个类仍然是可达的（可能是通过某个实例），那么它的所有方法当然也需要保留。

memory.c，在blackenObject()方法中添加代码：

```
      markObject((Obj*)klass->name);
      // 新增部分开始
      markTable(&klass->methods);
      // 新增部分结束
      break;
```

> We use the existing `markTable()` function, which traces through the key string and value in each table entry.

我们使用现有的`markTable()`函数，该函数可以追踪每个表项中的键字符串和值。

> Storing a class's methods is pretty familiar coming from jlox. The different part is how that table gets populated. Our previous interpreter had access to the entire AST node for the class declaration and all of the methods it contained. At runtime, the interpreter simply walked that list of declarations.

存储类方法的方式与jlox是非常类似的。不同之处在于如何填充该表。我们以前的解释器可以访问整个类声明及其包含的所有方法对应的AST节点。在运行时，解释器只是简单地遍历声明列表。

> Now every piece of information the compiler wants to shunt over to the runtime has to squeeze through the interface of a flat series of bytecode instructions. How do we take a class declaration, which can contain an arbitrarily large set of methods, and represent it as bytecode? Let's hop over to the compiler and find out.

现在，编译器想要分发到运行时的每一条信息都必须通过一个扁平的字节码指令序列形式。我们如何接受一个可以包含任意大的方法集的类声明，并以字节码的形式将其表现出来？让我们跳到编译器上看看。

> ## 28.1.2 Compiling method declarations

28.1.2 编译方法声明

> The last chapter left us with a compiler that parses classes but allows only an empty body. Now we insert a little code to compile a series of method declarations between the braces.

上一章留给我们一个能解析类但只允许空主体的编译器。现在我们添加一些代码来解析大括号之间的一系列方法声明。

compiler.c，在classDeclaration()方法中添加代码：

```
  consume(TOKEN_LEFT_BRACE, "Expect '{' before class body.");
  // 新增部分开始
  while (!check(TOKEN_RIGHT_BRACE) && !check(TOKEN_EOF)) {
    method();
  }
  // 新增部分结束
  consume(TOKEN_RIGHT_BRACE, "Expect '}' after class body.");
```

> Lox doesn't have field declarations, so anything before the closing brace at the end of the class body must be a method. We stop compiling methods when we hit that final curly or if we reach the end of

> the file. The latter check ensures our compiler doesn't get stuck in an infinite loop if the user accidentally forgets the closing brace.

Lox没有字段声明，因此，在主体块末尾的右括号之前的任何内容都必须是方法。当我们碰到最后的大括号或到达文件结尾时，就会停止编译方法。后一项检查可以确保我们的编译器不会在用户不小心忘记关闭大括号时陷入无限循环。

> The tricky part with compiling a class declaration is that a class may declare any number of methods. Somehow the runtime needs to look up and bind all of them. That would be a lot to pack into a single `OP_CLASS` instruction. Instead, the bytecode we generate for a class declaration will split the process into a *series* of instructions. The compiler already emits an `OP_CLASS` instruction that creates a new empty ObjClass object. Then it emits instructions to store the class in a variable with its name.

编译类声明的棘手之处在于，一个类可以声明任意数量的方法。运行时需要以某种方式查找并绑定所有这些方法。这会导致一个`OP_CLASS`指令中纳入了太多内容。相反，我们为类声明生成的字节码将这个过程分为一系列的指令。编译器已经发出了一条`OP_CLASS`指令，用来创建一个新的空ObjClass对象。然后它发出指令，将类存储在一个具有其名称的变量中^1。

> Now, for each method declaration, we emit a new `OP_METHOD` instruction that adds a single method to that class. When all of the `OP_METHOD` instructions have executed, we're left with a fully formed class. While the user sees a class declaration as a single atomic operation, the VM implements it as a series of mutations.

现在，对于每个方法声明，我们发出一条新的`OP_METHOD`指令，将一个方法添加到该类中。当所有的`OP_METHOD`指令都执行完毕后，我们就得到了一个完整的类。尽管用户将类声明看作是单个原子操作，但虚拟机却将其实现为一系列的变化。

> To define a new method, the VM needs three things:

要定义一个新方法，VM需要三样东西：

> 1. The name of the method.
> 2. The closure for the method body.
> 3. The class to bind the method to.

1. 方法名称。
2. 方法主体的闭包。
3. 绑定该方法的类。

> We'll incrementally write the compiler code to see how those all get through to the runtime, starting here:

我们会逐步编写编译器代码，看看它们是如何进入运行时的，从这里开始：

compiler.c，在function()方法后添加代码：

```
static void method() {
  consume(TOKEN_IDENTIFIER, "Expect method name.");
  uint8_t constant = identifierConstant(&parser.previous);
  emitBytes(OP_METHOD, constant);
}
```

> Like `OP_GET_PROPERTY` and other instructions that need names at runtime, the compiler adds the method name token's lexeme to the constant table, getting back a table index. Then we emit an `OP_METHOD` instruction with that index as the operand. That's the name. Next is the method body:

像`OP_GET_PROPERTY`和其它在运行时需要名称的指令一样，编译器将方法名称标识的词素添加到常量表中，获得表索引。然后发出一个`OP_METHOD`指令，以该索引作为操作数。这就是名称。接下来是方法主体：

compiler.c，在method()方法中添加代码：

```
  uint8_t constant = identifierConstant(&parser.previous);
  // 新增部分开始
  FunctionType type = TYPE_FUNCTION;
  function(type);
  // 新增部分结束
  emitBytes(OP_METHOD, constant);
```

> We use the same `function()` helper that we wrote for compiling function declarations. That utility function compiles the subsequent parameter list and function body. Then it emits the code to create an ObjClosure and leave it on top of the stack. At runtime, the VM will find the closure there.

我们使用为编译函数声明而编写的`function()`辅助函数。该工具函数会编译后续的参数列表和函数主体。然后它发出创建ObjClosure的代码，并将其留在栈顶。在运行时，VM会在那里找到这个闭包。

> Last is the class to bind the method to. Where can the VM find that? Unfortunately, by the time we reach the `OP_METHOD` instruction, we don't know where it is. It could be on the stack, if the user declared the class in a local scope. But a top-level class declaration ends up with the ObjClass in the global variable table.

最后是要绑定方法的类。VM在哪里可以找到它呢？不幸的是，当我们到达`OP_METHOD`指令时，我们还不知道它在哪里。如果用户在局部作用域中声明该类，那它可能在栈上。但是顶层的类声明最终会成为全局变量表中的ObjClass^2。

> Fear not. The compiler does know the *name* of the class. We can capture it right after we consume its token.

不要担心。编译器确实知道类的*名称*。我们可以在消费完名称标识后捕获这个值。

compiler.c，在classDeclaration()方法中添加代码：

```
  consume(TOKEN_IDENTIFIER, "Expect class name.");
  // 新增部分开始
  Token className = parser.previous;
  // 新增部分结束
  uint8_t nameConstant = identifierConstant(&parser.previous);
```

> And we know that no other declaration with that name could possibly shadow the class. So we do the easy fix. Before we start binding methods, we emit whatever code is necessary to load the class back on top of the stack.

我们知道，其它具有该名称的声明不可能会遮蔽这个类。所以我们选择了简单的处理方式。在我们开始绑定方法之前，通过一些必要的代码，将类加载回栈顶。

compiler.c，在classDeclaration()方法中添加代码：

```
  defineVariable(nameConstant);
  // 新增部分开始
  namedVariable(className, false);
  // 新增部分结束
  consume(TOKEN_LEFT_BRACE, "Expect '{' before class body.");
```

> Right before compiling the class body, we call `namedVariable()`. That helper function generates code to load a variable with the given name onto the stack. Then we compile the methods.

在编译类主体之前，我们调用`namedVariable()`。这个辅助函数会生成代码，将一个具有给定名称的变量加载到栈中^3。然后，我们编译方法。

> This means that when we execute each `OP_METHOD` instruction, the stack has the method's closure on top with the class right under it. Once we've reached the end of the methods, we no longer need the class and tell the VM to pop it off the stack.

这意味着，当我们执行每一条`OP_METHOD`指令时，栈顶是方法的闭包，它下面就是类。一旦我们到达了方法的末尾，我们就不再需要这个类，并告诉虚拟机将该它从栈中弹出。

compiler.c，在classDeclaration()方法中添加代码：

```
  consume(TOKEN_RIGHT_BRACE, "Expect '}' after class body.");
  // 新增部分开始
  emitByte(OP_POP);
  // 新增部分结束
}
```

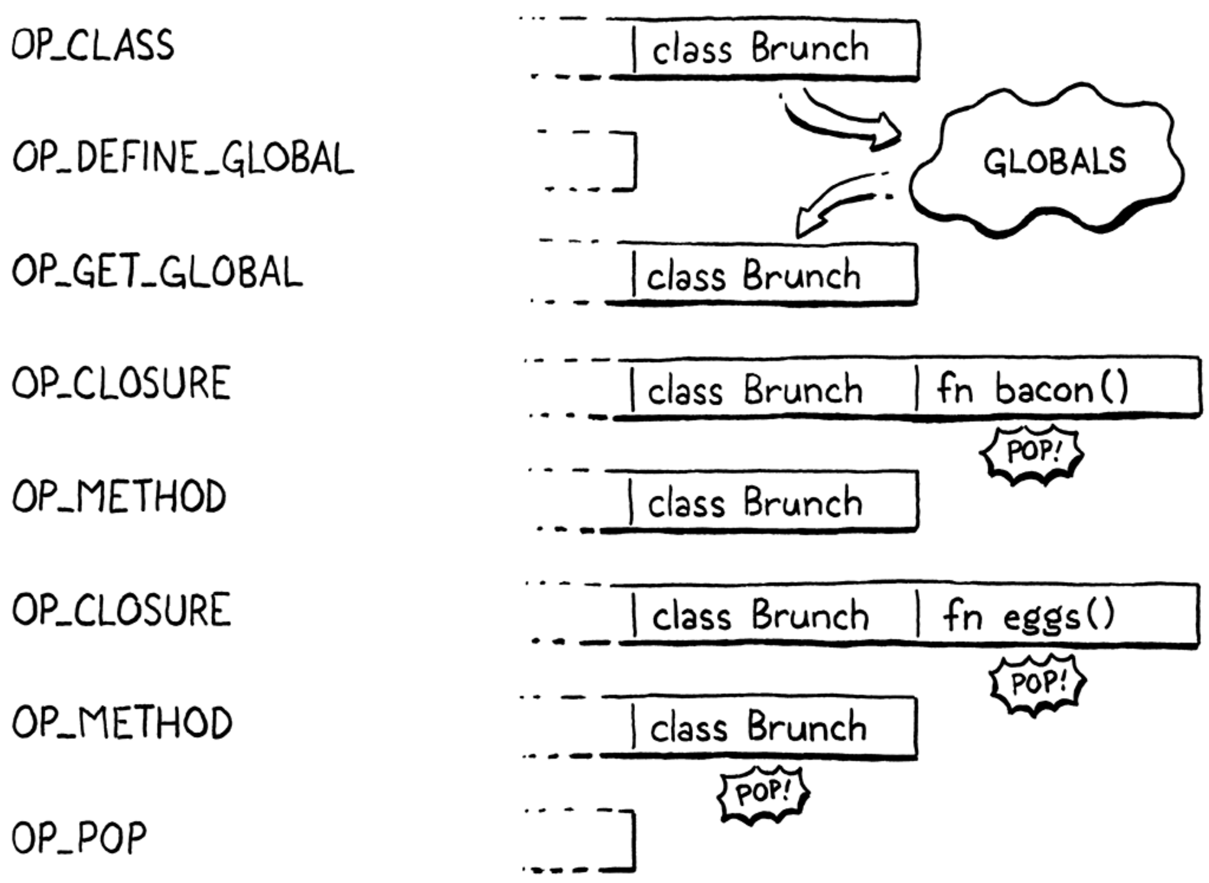
> Putting all of that together, here is an example class declaration to throw at the compiler:

把所有这些放在一起，下面是一个可以扔给编译器的类声明示例：

```
class Brunch {
  bacon() {}
  eggs() {}
}
```

> Given that, here is what the compiler generates and how those instructions affect the stack at runtime:

鉴于此，下面是编译器生成的内容以及这些指令在运行时如何影响堆栈：



> All that remains for us is to implement the runtime for that new `OP_METHOD` instruction.

我们剩下要做的就是为这个新的`OP_METHOD`指令实现运行时。

> 28.1.3 Executing method declarations

28.1.3 执行方法声明

> First we define the opcode.

首先我们定义操作码。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CLASS,
  // 新增部分开始
  OP_METHOD
  // 新增部分结束
} OpCode;
```

> We disassemble it like other instructions that have string constant operands.

我们像其它具有字符串常量操作数的指令一样对它进行反汇编。

debug.c，在disassembleInstruction()方法中添加代码：

```
    case OP_CLASS:
      return constantInstruction("OP_CLASS", chunk, offset);
    // 新增部分开始
    case OP_METHOD:
      return constantInstruction("OP_METHOD", chunk, offset);
    // 新增部分结束
    default:
```

> And over in the interpreter, we add a new case too.

在解释器中，我们也添加一个新的case分支。

vm.c，在run()方法中添加代码：

```
        break;
      // 新增部分开始
      case OP_METHOD:
        defineMethod(READ_STRING());
        break;
      // 新增部分结束
    }
```

> There, we read the method name from the constant table and pass it here:

其中，我们从常量表中读取方法名称，并将其传递到这里：

vm.c，在closeUpvalues()方法后添加代码：

```
static void defineMethod(ObjString* name) {
  Value method = peek(0);
  ObjClass* klass = AS_CLASS(peek(1));
  tableSet(&klass->methods, name, method);
  pop();
}
```

> The method closure is on top of the stack, above the class it will be bound to. We read those two stack slots and store the closure in the class's method table. Then we pop the closure since we're done with it.

方法闭包位于栈顶，在它将绑定的类的上方。我们读取这两个栈槽并将闭包存储到类的方法表中。然后弹出闭包，因为我们已经用完了。

> Note that we don't do any runtime type checking on the closure or class object. That `AS_CLASS()` call is safe because the compiler itself generated the code that causes the class to be in that stack slot. The VM trusts its own compiler.

注意，我们没有对闭包或类对象做任何的运行时类型检查。`AS_CLASS()`调用是安全的，因为编译器本身会生成使类位于栈槽的代码。虚拟机信任自己的编译器^4。

> After the series of `OP_METHOD` instructions is done and the `OP_POP` has popped the class, we will have a class with a nicely populated method table, ready to start doing things. The next step is pulling those methods back out and using them.

在完成一系列的`OP_METHOD`指令并且`OP_POP`弹出类后，我们将得到一个已填充好方法表的类，可以开始做事情了。下一步是将这些方法拉出来并使用它们。

28.2 Method References

28.2 方法引用

> Most of the time, methods are accessed and immediately called, leading to this familiar syntax:

大多数情况下，方法被访问并立即被调用，导致了这种熟悉的语法：

```
instance.method(argument);
```

> But remember, in Lox and some other languages, those two steps are distinct and can be separated.

但是请记住，在Lox和其它一些语言中，这两个步骤是不同的，可以分开。

```
var closure = instance.method;
closure(argument);
```

> Since users *can* separate the operations, we have to implement them separately. The first step is using our existing dotted property syntax to access a method defined on the instance's class. That should return some kind of object that the user can then call like a function.

由于用户可以将这些操作分开，所以我们必须分别实现它们。第一步是使用现有的点属性语法来访问实例的类中定义的方法。这应该返回某种类型的对象，然后用户可以向函数一样调用它。

> The obvious approach is to look up the method in the class's method table and return the ObjClosure associated with that name. But we also need to remember that when you access a method, `this` gets bound to the instance the method was accessed from. Here's the example from when we added methods to jlox:

明显的方式是，在类的方法表中查找该方法，并返回与该名称关联的ObjClosure。但是我们也需要记住，当你访问一个方法时，`this`绑定到访问该方法的实例上。下面是我们在向jlox添加方法时的例子：

```
class Person {
  sayName() {
    print this.name;
  }
}
```

```
var jane = Person();
jane.name = "Jane";

var method = jane.sayName;
method(); // ?
```

> This should print “Jane”, so the object returned by `.sayName` somehow needs to remember the instance it was accessed from when it later gets called. In jlox, we implemented that “memory” using the interpreter’s existing heap-allocated Environment class, which handled all variable storage.

这里应该打印“Jane”，因此`.sayName`返回的对象在以后被调用时需要记住访问它的实例。在jlox中，我们通过解释器已有的堆分配的Environment类来实现这个“记忆”，该Environment类会处理所有的变量存储。。

> Our bytecode VM has a more complex architecture for storing state. Local variables and temporaries are on the stack, globals are in a hash table, and variables in closures use upvalues. That necessitates a somewhat more complex solution for tracking a method’s receiver in clox, and a new runtime type.

我们的字节码虚拟机用一个更复杂的结构来存储状态。局部变量和临时变量在栈中，全局变量在哈希表中，而闭包中的变量使用上值。这就需要一个更复杂的跟踪clox中方法接收者的解决方案，以及一个新的运行时类型。

> ### 28.2.1 Bound methods

28.2.1 已绑定方法

> When the user executes a method access, we’ll find the closure for that method and wrap it in a new “bound method” object that tracks the instance that the method was accessed from. This bound object can be called later like a function. When invoked, the VM will do some shenanigans to wire up `this` to point to the receiver inside the method’s body.

当用户执行一个方法访问时，我们会找到该方法的闭包，并将其包装在一个新的“已绑定方法（bound method）”对象中^5，该对象会跟踪访问该方法的实例。这个已绑定对象可以像一个函数一样在稍后被调用。当被调用时，虚拟机会做一些小动作，将`this`连接到方法主体中的接收器。

> Here’s the new object type:

下面是新的对象类型：

object.h，在结构体ObjInstance后添加代码：

```
} ObjInstance;
// 新增部分开始
typedef struct {
  Obj obj;
  Value receiver;
  ObjClosure* method;
} ObjBoundMethod;
// 新增部分结束
ObjClass* newClass(ObjString* name);
```

> It wraps the receiver and the method closure together. The receiver’s type is Value even though methods can be called only on ObjInstances. Since the VM doesn’t care what kind of receiver it has anyway, using Value means we don’t have to keep converting the pointer back to a Value when it gets passed to more general functions.

它将接收器和方法闭包包装在一起。尽管方法只能在ObjInstances上调用，但接收器类型是Value。因为虚拟机并不关心它拥有什么样的接收器，使用Value意味着当它需要传递给更多通用函数时，我们不必将指针转换回Value。

> The new struct implies the usual boilerplate you’re used to by now. A new case in the object type enum:

新的结构体暗含了你现在已经熟悉的常规模板。对象类型枚举中的新值：

object.h，在枚举ObjType中添加代码：

```
typedef enum {
  // 新增部分开始
  OBJ_BOUND_METHOD,
  // 新增部分结束
  OBJ_CLASS,
```

> A macro to check a value’s type:

一个检查值类型的宏：

object.h，添加代码：

```
#define OBJ_TYPE(value)        (AS_OBJ(value)->type)
// 新增部分开始
#define IS_BOUND_METHOD(value) isObjType(value, OBJ_BOUND_METHOD)
// 新增部分结束
#define IS_CLASS(value)        isObjType(value, OBJ_CLASS)
```

> Another macro to cast the value to an ObjBoundMethod pointer:

另一个将值转换为ObjBoundMethod 指针的宏：

object.h，添加代码：

```
#define IS_STRING(value)       isObjType(value, OBJ_STRING)
// 新增部分开始
#define AS_BOUND_METHOD(value) ((ObjBoundMethod*)AS_OBJ(value))
// 新增部分结束
#define AS_CLASS(value)        ((ObjClass*)AS_OBJ(value))
```

> A function to create a new ObjBoundMethod:

一个创建新ObjBoundMethod的函数：

object.h，在结构体ObjBoundMethod后添加代码：

```
} ObjBoundMethod;
// 新增部分开始
ObjBoundMethod* newBoundMethod(Value receiver,
                               ObjClosure* method);
// 新增部分结束
ObjClass* newClass(ObjString* name);
```

> And an implementation of that function here:

以及该函数的实现：

object.c，在allocateObject()方法后添加代码：

```
ObjBoundMethod* newBoundMethod(Value receiver,
                               ObjClosure* method) {
  ObjBoundMethod* bound = ALLOCATE_OBJ(ObjBoundMethod,
                                       OBJ_BOUND_METHOD);
  bound->receiver = receiver;
  bound->method = method;
  return bound;
}
```

> The constructor-like function simply stores the given closure and receiver. When the bound method is no longer needed, we free it.

这个类似构造器的函数简单地存储了给定的闭包和接收器。当不再需要某个已绑定方法时，我们将其释放。

memory.c，在freeObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_BOUND_METHOD:
      FREE(ObjBoundMethod, object);
      break;
    // 新增部分结束
    case OBJ_CLASS: {
```

> The bound method has a couple of references, but it doesn't *own* them, so it frees nothing but itself. However, those references do get traced by the garbage collector.

已绑定方法有几个引用，但并不*拥有*它们，所以它只释放自己。但是，这些引用确实要被垃圾回收器跟踪到。

memory.c，在blackenObject()方法中添加代码：

```
  switch (object->type) {
    // 新增部分开始
    case OBJ_BOUND_METHOD: {
      ObjBoundMethod* bound = (ObjBoundMethod*)object;
      markValue(bound->receiver);
      markObject((Obj*)bound->method);
      break;
    }
    // 新增部分结束
    case OBJ_CLASS: {
```

> This ensures that a handle to a method keeps the receiver around in memory so that `this` can still find the object when you invoke the handle later. We also trace the method closure.

这可以确保方法的句柄会将接收器保持在内存中，以便后续当你调用这个句柄时，`this`仍然可以找到这个对象。我们也会跟踪方法闭包^6。

> The last operation all objects support is printing.

所有对象要支持的最后一个操作是打印。

object.c，在printObject()方法中添加代码：

```
  switch (OBJ_TYPE(value)) {
    // 新增部分开始
    case OBJ_BOUND_METHOD:
      printFunction(AS_BOUND_METHOD(value)->method->function);
      break;
    // 新增部分结束
    case OBJ_CLASS:
```

> A bound method prints exactly the same way as a function. From the user's perspective, a bound method *is* a function. It's an object they can call. We don't expose that the VM implements bound methods using a different object type.

已绑定方法的打印方式与函数完全相同。从用户的角度来看，已绑定方法*就是*一个函数，是一个可以被他们调用的对象。我们不会暴露虚拟机中使用不同的对象类型来实现已绑定方法的事实。

> Put on your party hat because we just reached a little milestone. ObjBoundMethod is the very last runtime type to add to clox. You've written your last `IS_` and `AS_` macros. We're only a few chapters from the end of the book, and we're getting close to a complete VM.

庆祝一下，因为我们刚刚到达了一个小小的里程碑。ObjBoundMethod是要添加到clox中的最后一个运行时类型。你已经写完了最后的`IS_`和`AS_`宏。我们离本书的结尾只有几章了，而且我们已经接近一个完整的虚拟机了。

> ## 28.2.2 Accessing methods

28.2.2 访问方法

> Let's get our new object type doing something. Methods are accessed using the same "dot" property syntax we implemented in the last chapter. The compiler already parses the right expressions and emits `OP_GET_PROPERTY` instructions for them. The only changes we need to make are in the runtime.

我们来让新对象类型做点什么。方法是通过我们在上一章中实现的"点"属性语法进行访问的。编译器已经能够解析正确的表达式，并为它们发出`OP_GET_PROPERTY`指令。我们接下来只需要在运行时做适当改动。

> When a property access instruction executes, the instance is on top of the stack. The instruction's job is to find a field or method with the given name and replace the top of the stack with the accessed property.

当执行某个属性访问指令时，实例在栈顶。该指令的任务是找到一个具有给定名称的字段或方法，并将栈顶替换为所访问的属性。

> The interpreter already handles fields, so we simply extend the `OP_GET_PROPERTY` case with another section.

解释器已经处理了字段，所以我们只需要在`OP_GET_PROPERTY`分支中扩展另一部分。

vm.c，在run()方法中替换2行：

```
          pop(); // Instance.
          push(value);
          break;
        }
        // 替换部分开始
        if (!bindMethod(instance->klass, name)) {
          return INTERPRET_RUNTIME_ERROR;
        }
        break;
        // 替换部分结束
      }
```

> We insert this after the code to look up a field on the receiver instance. Fields take priority over and shadow methods, so we look for a field first. If the instance does not have a field with the given property name, then the name may refer to a method.

我们在查找接收器实例上字段的代码后面插入这部分逻辑。字段优先于方法，因此我们首先查找字段。如果实例确实不包含具有给定属性名称的字段，那么这个名称可能指向的是一个方法。

> We take the instance's class and pass it to a new `bindMethod()` helper. If that function finds a method, it places the method on the stack and returns `true`. Otherwise it returns `false` to indicate a method with that name couldn't be found. Since the name also wasn't a field, that means we have a runtime error, which aborts the interpreter.

我们获取实例的类，并将其传递给新的`bindMethod()`辅助函数。如果该函数找到了方法，它会将该方法放在栈中并返回`true`。否则返回`false`，表示找不到具有该名称的方法。因为这个名称也不是字段，这意味着我们遇到了一个运行时错误，从而中止了解释器。

> Here is the good stuff:

下面是这段精彩的逻辑：

vm.c，在callValue()方法后添加代码：

```
static bool bindMethod(ObjClass* klass, ObjString* name) {
  Value method;
  if (!tableGet(&klass->methods, name, &method)) {
    runtimeError("Undefined property '%s'.", name->chars);
    return false;
  }

  ObjBoundMethod* bound = newBoundMethod(peek(0),
                                         AS_CLOSURE(method));
  pop();
  push(OBJ_VAL(bound));
  return true;
}
```

> First we look for a method with the given name in the class's method table. If we don't find one, we report a runtime error and bail out. Otherwise, we take the method and wrap it in a new ObjBoundMethod. We grab the receiver from its home on top of the stack. Finally, we pop the instance and replace the top of the stack with the bound method.

首先，我们在类的方法表中查找具有指定名称的方法。如果我们没有找到，我们就报告一个运行时错误并退出。否则，我们获取该方法，并将其包装为一个新的ObjBoundMethod。我们从栈顶获得接收器。最后，我们弹出实例，并将这个已绑定方法替换到栈顶。

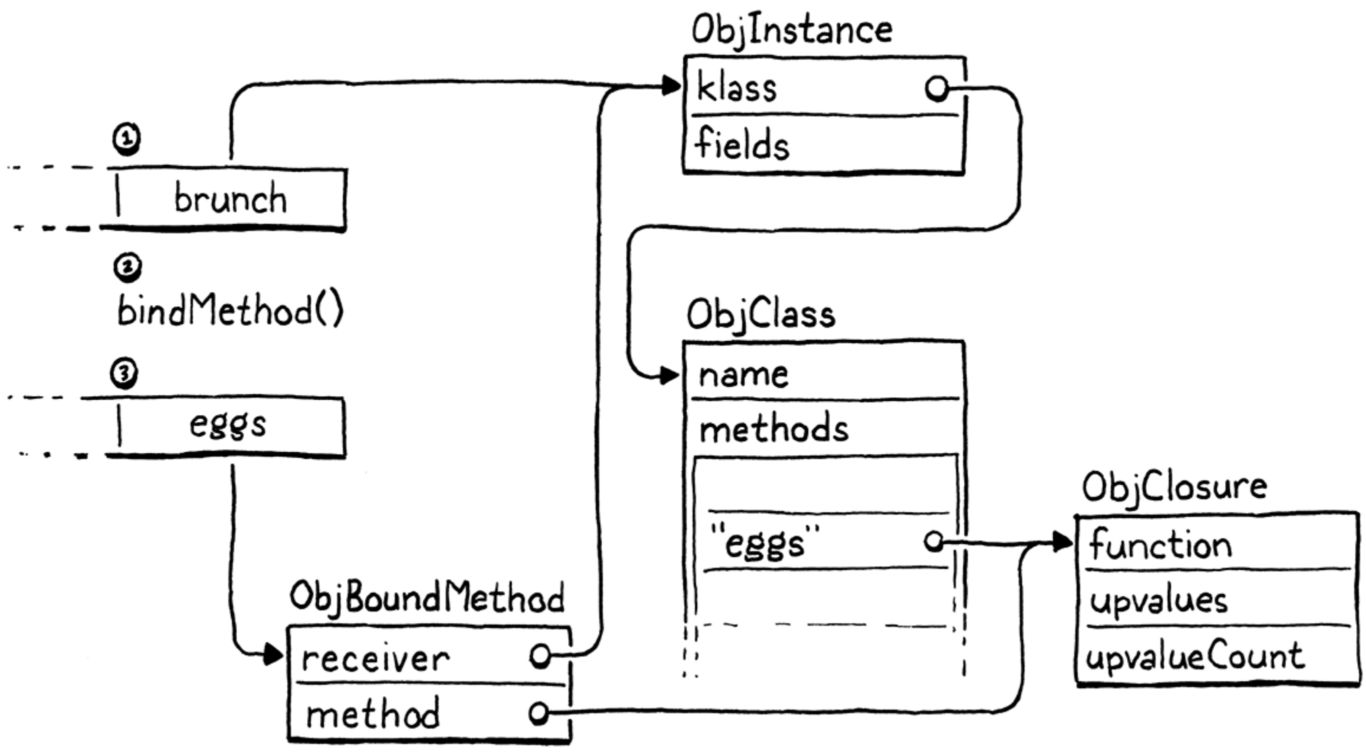
> For example:

举例来说：

```
class Brunch {
  eggs() {}
}

var brunch = Brunch();
var eggs = brunch.eggs;
```

> Here is what happens when the VM executes the `bindMethod()` call for the `brunch.eggs` expression:

下面是虚拟机执行`brunch.eggs`表达式的`bindMethod()`调用时发生的情况：



> That's a lot of machinery under the hood, but from the user's perspective, they simply get a function that they can call.

在底层有很多机制，但从用户的角度来看，他们只是得到了一个可以调用的函数。

> ## 28.2.3 Calling methods

28.2.3 调用方法

> Users can declare methods on classes, access them on instances, and get bound methods onto the stack. They just can't *do* anything useful with those bound method objects. The operation we're missing is calling them. Calls are implemented in `callValue()`, so we add a case there for the new object type.

用户可以在类上声明方法，在实例上访问这些方法，并将已绑定的方法放到栈上^7。他们目前还不能使用这些已绑定方法做任何有意义的事。我们所缺少的操作就是调用他们。调用在`callValue()`中实现，所以我们在其中为新的对象类型添加一个case分支。

vm.c，在callValue()方法中添加代码：

```
    switch (OBJ_TYPE(callee)) {
      // 新增部分开始
      case OBJ_BOUND_METHOD: {
        ObjBoundMethod* bound = AS_BOUND_METHOD(callee);
        return call(bound->method, argCount);
      }
      // 新增部分结束
      case OBJ_CLASS: {
```

> We pull the raw closure back out of the ObjBoundMethod and use the existing `call()` helper to begin an invocation of that closure by pushing a CallFrame for it onto the call stack. That's all it takes to be able to run this Lox program:

我们从ObjBoundMethod中抽取原始闭包，并使用现有的`call()`辅助函数，通过将对应CallFrame压入调用栈，来开始对该闭包的调用。有了这些，就能够运行下面这个Lox程序：

```
class Scone {
  topping(first, second) {
    print "scone with " + first + " and " + second;
  }
}

var scone = Scone();
scone.topping("berries", "cream");
```

> That's three big steps. We can declare, access, and invoke methods. But something is missing. We went to all that trouble to wrap the method closure in an object that binds the receiver, but when we invoke the method, we don't use that receiver at all.

这是三大步。我们可以声明、访问和调用方法。但我们缺失了一些东西。我们费尽心思将方法闭包包装在一个绑定了接收器的对象中，但当我们调用方法时，根本没有使用那个接收器。

28.3 This

> The reason bound methods need to keep hold of the receiver is so that it can be accessed inside the body of the method. Lox exposes a method's receiver through `this` expressions. It's time for some new syntax. The lexer already treats `this` as a special token type, so the first step is wiring that token up in the parse table.

已绑定方法中需要保留接收器的原因在于，这样就可以在方法体内部访问接收器实例。Lox通过`this`表达式暴露方法的接收器。现在是时候用一些新语法了。词法解析器已经将`this`当作一个特殊的标识类型，因此第一步是将该标识链接到解析表中。

compiler.c，替换1行：

```
  [TOKEN_SUPER]         = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_THIS]          = {this_,    NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_TRUE]          = {literal,  NULL,   PREC_NONE},
```

> When the parser encounters a `this` in prefix position, it dispatches to a new parser function.

当解析器在前缀位置遇到一个`this`时，会派发给新的解析器函数^8。

compiler.c，在variable()方法后添加：

```
static void this_(bool canAssign) {
  variable(false);
}
```

> We'll apply the same implementation technique for this in clox that we used in jlox. We treat this as a lexically scoped local variable whose value gets magically initialized. Compiling it like a local variable means we get a lot of behavior for free. In particular, closures inside a method that reference this will do the right thing and capture the receiver in an upvalue.

对于clox中的`this`，我们将使用与jlox相同的技术。我们将`this`看作是一个具有词法作用域的局部变量，它的值被神奇地初始化了。像局部变量一样编译它意味着我们可以免费获得很多行为。特别是，引用`this`的方法对应的闭包会做正确的事情，并在上值中捕获接收器。

> When the parser function is called, the this token has just been consumed and is stored as the previous token. We call our existing variable() function which compiles identifier expressions as variable accesses. It takes a single Boolean parameter for whether the compiler should look for a following = operator and parse a setter. You can't assign to this, so we pass false to disallow that.

当解析器函数被调用时，`this`标识刚刚被使用，并且存储在上一个标识中。我们调用已有的`variable()`函数，它将标识符表达式编译为变量访问。它需要一个Boolean参数，用于判断编译器是否应该查找后续的`=`运算符并解析setter。你不能给`this`赋值，所以我们传入`false`来禁止它。

> The variable() function doesn't care that this has its own token type and isn't an identifier. It is happy to treat the lexeme "this" as if it were a variable name and then look it up using the existing scope resolution machinery. Right now, that lookup will fail because we never declared a variable whose name is "this". It's time to think about where the receiver should live in memory.

`variable()`函数并不关心`this`是否有自己的标识类型，也不关心它是否是一个标识符。它很乐意将词素`this`当作一个变量名，然后用现有的作用域解析机制来查找它。现在，这种查找会失败，因为我们从未声明过名称为`this`的变量。现在是时候考虑一下接收器在内存中的位置了。

> At least until they get captured by closures, clox stores every local variable on the VM's stack. The compiler keeps track of which slots in the function's stack window are owned by which local variables. If you recall, the compiler sets aside stack slot zero by declaring a local variable whose name is an empty string.

至少在每个局部变量被闭包捕获之前，clox会将其存储在VM的栈中。编译器持续跟踪函数栈窗口中的哪个槽由哪些局部变量所拥有。如果你还记得，编译器通过声明一个名称为空字符串的局部变量来预留出栈槽0。

> For function calls, that slot ends up holding the function being called. Since the slot has no name, the function body never accesses it. You can guess where this is going. For *method* calls, we can repurpose that slot to store the receiver. Slot zero will store the instance that this is bound to. In order to compile this expressions, the compiler simply needs to give the correct name to that local variable.

对于函数调用来说，这个槽会存储被调用的函数。因为这个槽没有名字，所以函数主体永远不会访问它。你可以猜到接下来会发生什么。对于方法调用，我们可以重新利用这个槽来存储接收器。槽0会存储`this`绑定的实例。为了编译`this`表达式，编译器只需要给这个局部变量一个正确的名称。

compiler.c，在initCompiler()方法中替换2行：

```
    local->isCaptured = false;
    // 替换部分开始
    if (type != TYPE_FUNCTION) {
      local->name.start = "this";
      local->name.length = 4;
    } else {
      local->name.start = "";
      local->name.length = 0;
    }
    // 替换部分结束
  }
```

> We want to do this only for methods. Function declarations don't have a `this`. And, in fact, they *must not* declare a variable named "this", so that if you write a `this` expression inside a function declaration which is itself inside a method, the `this` correctly resolves to the outer method's receiver.

我们只想对方法这样做。函数声明中没有`this`。事实上，它们不能声明一个名为`this`的变量，因此，如果你在函数声明中写了一个`this`表达式，而该函数本身又在某个方法中，这个`this`会被正确地解析为外部方法的接收器。

```
class Nested {
  method() {
    fun function() {
      print this;
    }

    function();
  }
}

Nested().method();
```

> This program should print "Nested instance". To decide what name to give to local slot zero, the compiler needs to know whether it's compiling a function or method declaration, so we add a new case to our FunctionType enum to distinguish methods.

这个程序应该打印"Nested instance"。为了决定给局部槽0取什么名字，编译器需要知道它正在编译一个函数还是方法声明，所以我们向FunctionType枚举中增加一个新的类型来区分方法。

compiler.c，在枚举FunctionType中添加代码：

```
    TYPE_FUNCTION,
    // 新增部分开始
    TYPE_METHOD,
    // 新增部分结束
    TYPE_SCRIPT
```

> When we compile a method, we use that type.

当我们编译方法时，就使用这个类型。

compiler.c，在method()方法中替换1行：

```
  uint8_t constant = identifierConstant(&parser.previous);
  // 替换部分开始
  FunctionType type = TYPE_METHOD;
  // 替换部分结束
  function(type);
```

> Now we can correctly compile references to the special “this” variable, and the compiler will emit the right `OP_GET_LOCAL` instructions to access it. Closures can even capture `this` and store the receiver in upvalues. Pretty cool.

现在我们可以正确地编译对特殊的`this`变量的引用，编译器会发出正确的`OP_GET_LOCAL`来访问它。闭包甚至可以捕获`this`，并将接收器存储在上值中。非常酷。

> Except that at runtime, the receiver isn’t actually *in* slot zero. The interpreter isn’t holding up its end of the bargain yet. Here is the fix:

除了在运行时，接收器实际上并不在槽0*中*。解释器还没有履行它的承诺。下面是修复方法：

vm.c，在callValue()方法中添加代码：

```
      case OBJ_BOUND_METHOD: {
        ObjBoundMethod* bound = AS_BOUND_METHOD(callee);
        // 新增部分开始
        vm.stackTop[-argCount - 1] = bound->receiver;
        // 新增部分结束
        return call(bound->method, argCount);
      }
```

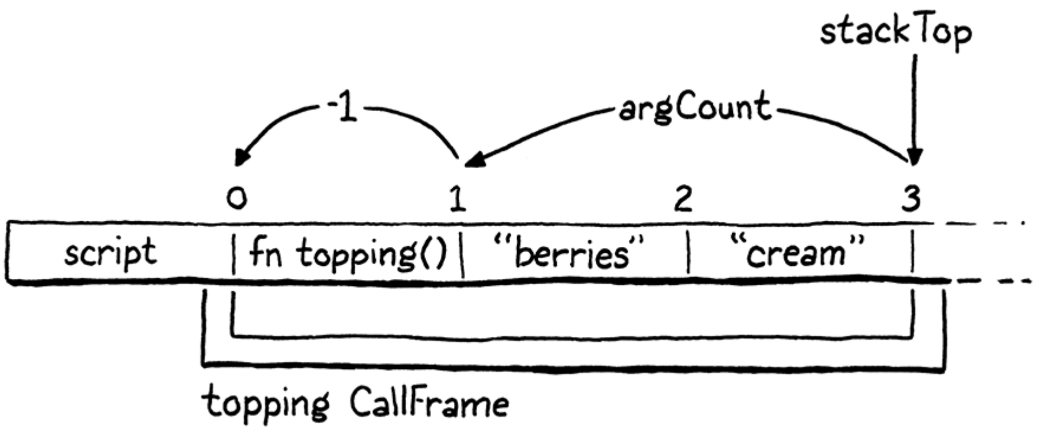
> When a method is called, the top of the stack contains all of the arguments, and then just under those is the closure of the called method. That’s where slot zero in the new CallFrame will be. This line of code inserts the receiver into that slot. For example, given a method call like this:

当某个方法被调用时，栈顶包含所有的参数，然后在这些参数下面是被调用方法的闭包。这就是新的CallFrame中槽0所在的位置。这一行代码会向该槽中插入接收器。例如，给出一个这样的方法调用：

```
scone.topping("berries", "cream");
```

> We calculate the slot to store the receiver like so:

我们像这样计算存储接收器的槽：



> The `-argCount` skips past the arguments and the `- 1` adjusts for the fact that `stackTop` points just *past* the last used stack slot.

`-argCount`跳过传递的参数，而`-1`则是因为`stackTop`指向刚刚最后一个实用的栈槽而做的调整。

> ### 28.3.1 Misusing this

28.3.1 误用this

> Our VM now supports users *correctly* using `this`, but we also need to make sure it properly handles users *mis*using `this`. Lox says it is a compile error for a `this` expression to appear outside of the body of a method. These two wrong uses should be caught by the compiler:

我们的虚拟机现在支持用户正确地使用`this`，但我们还需要确保它能正确地处理用户误用`this`的情况。Lox表示，如果`this`表达式出现在方法主体之外，就是一个编译错误。这两个错误的用法是编译器应该捕获的：

```
print this; // At top level.

fun notMethod() {
  print this; // In a function.
}
```

> So how does the compiler know if it's inside a method? The obvious answer is to look at the FunctionType of the current Compiler. We did just add an enum case there to treat methods specially. However, that wouldn't correctly handle code like the earlier example where you are inside a function which is, itself, nested inside a method.

那么编译器如何知道它是否在一个方法中呢？显而易见的答案是，查看当前Compiler的FunctionType。我们在其中添加了一个新的枚举值来特殊对待方法。但是，这并不能正确地处理前面那个示例中的代码，即你在一个函数里面，而这个函数本身又嵌套在一个方法中。

> We could try to resolve "this" and then report an error if it wasn't found in any of the surrounding lexical scopes. That would work, but would require us to shuffle around a bunch of code, since right now the code for resolving a variable implicitly considers it a global access if no declaration is found.

我们可以尝试解析`this`，如果在外围的词法作用域中没有找到它，就报告一个错误。这样做是可行的，但需要我们修改一堆代码，因为如果没有找到声明，解析变量的代码现在会隐式地将其视为全局变量访问。

> In the next chapter, we will need information about the nearest enclosing class. If we had that, we could use it here to determine if we are inside a method. So we may as well make our future selves' lives a little easier and put that machinery in place now.

在下一章中，我们将需要关于最近邻外层类的信息。如果我们有这些信息，就可以在这里使用它来确定我们是否在某个方法中。因此，我们不妨让未来的自己生活得轻松一些，现在就把这种机制搞定。

compiler.c，在变量current后添加代码：

```
Compiler* current = NULL;
// 新增部分开始
ClassCompiler* currentClass = NULL;
// 新增部分结束
static Chunk* currentChunk() {
```

> This module variable points to a struct representing the current, innermost class being compiled. The new type looks like this:

这个模块变量指向一个表示当前正在编译的最内部类的结构体，新的类型看起来像这样：

compiler.c，在结构体Compiler后添加代码：

```
} Compiler;
// 新增部分开始
typedef struct ClassCompiler {
  struct ClassCompiler* enclosing;
} ClassCompiler;
// 新增部分结束
Parser parser;
```

> Right now we store only a pointer to the ClassCompiler for the enclosing class, if any. Nesting a class declaration inside a method in some other class is an uncommon thing to do, but Lox supports it. Just like the Compiler struct, this means ClassCompiler forms a linked list from the current innermost class being compiled out through all of the enclosing classes.

现在，我们只存储一个指向外层类（如果存在的话）的ClassCompiler的指针。将类声明嵌套在其它类的某个方法中并不常见，但Lox支持这种做法。就像Compiler结构体一样，这意味着ClassCompiler形成了一个链表，从当前正在被编译的最内层类一直到所有的外层类。

> If we aren't inside any class declaration at all, the module variable `currentClass` is `NULL`. When the compiler begins compiling a class, it pushes a new ClassCompiler onto that implicit linked stack.

如果我们根本不在任何类的声明中，则模块变量`currentClass`是`NULL`。当编译器开始编译某个类时，它会将一个新的ClassCompiler推入这个隐式的链栈中。

compiler.c，在classDeclaration()方法中添加代码：

```
  defineVariable(nameConstant);
  // 新增部分开始
  ClassCompiler classCompiler;
  classCompiler.enclosing = currentClass;
  currentClass = &classCompiler;
  // 新增部分结束
  namedVariable(className, false);
```

> The memory for the ClassCompiler struct lives right on the C stack, a handy capability we get by writing our compiler using recursive descent. At the end of the class body, we pop that compiler off the stack and restore the enclosing one.

ClassCompiler结构体的内存正好位于C栈中，这是通过使用递归下降来编写编译器而获得的便利。在类主体的最后，我们将该编译器从栈中弹出，并恢复外层的编译器。

compiler.c，在classDeclaration()方法中添加代码：

```
  emitByte(OP_POP);
  // 新增部分开始
  currentClass = currentClass->enclosing;
  // 新增部分结束
}
```

> When an outermost class body ends, `enclosing` will be `NULL`, so this resets `currentClass` to `NULL`. Thus, to see if we are inside a class—and therefore inside a method—we simply check that module variable.

当最外层的类主体结束时，`enclosing`将是`NULL`，因此这里会将`currentClass`重置为`NULL`。因此，要想知道我们是否在一个类内部——也就是是否在一个方法中——我们只需要检查模块变量。

compiler.c，在this_()方法中添加代码：

```
static void this_(bool canAssign) {
  // 新增部分开始
  if (currentClass == NULL) {
    error("Can't use 'this' outside of a class.");
    return;
  }
  // 新增部分结束
  variable(false);
```

> With that, `this` outside of a class is correctly forbidden. Now our methods really feel like *methods* in the object-oriented sense. Accessing the receiver lets them affect the instance you called the method on. We're getting there!

有个这个，类之外的`this`就被正确地禁止了。现在我们的方法就像是面向对象意义上的*方法*。对接收器的访问使得它们可以影响调用方法的实例。我们正在走向成功！

> ## 28.4 Instance Initializers

28.4 实例初始化器

> The reason object-oriented languages tie state and behavior together—one of the core tenets of the paradigm—is to ensure that objects are always in a valid, meaningful state. When the only way to touch an object's state is through its methods, the methods can make sure nothing goes awry. But that presumes the object is *already* in a proper state. What about when it's first created?

面向对象语言之所以将状态和行为结合在一起（范式的核心原则之一），是为了确保对象总是处于有效的、有意义的状态。当接触对象状态的唯一形式是通过它的方法时，这些方法可以确保不会出错^9。但前提是对象*已经*处于正常状态。那么，当对象第一次被创建时呢？

> Object-oriented languages ensure that brand new objects are properly set up through constructors, which both produce a new instance and initialize its state. In Lox, the runtime allocates new raw instances, and a class may declare an initializer to set up any fields. Initializers work mostly like normal methods, with a few tweaks:

面向对象的语言通过构造函数确保新对象是被正确设置的，构造函数会生成一个新实例并初始化其状态。在Lox中，运行时会分配新的原始实例，而类可以声明一个初始化器来设置任何字段。初始化器的工作原理和普通方法差不多，只是做了一些调整：

> 1. The runtime automatically invokes the initializer method whenever an instance of a class is created.
> 2. The caller that constructs an instance always gets the instance back after the initializer finishes, regardless of what the initializer function itself returns. The initializer method doesn't need to explicitly return `this`.
> 3. In fact, an initializer is *prohibited* from returning any value at all since the value would never be seen anyway.

1. 每当一个类的实例被创建时，运行时会自动调用初始化器方法。
2. 构建实例的调用方总是在初始化器完成后得到实例，而不管初始化器本身返回什么。初始化器方法不需要显式地返回`this`^10。
3. 事实上，初始化器根本不允许返回任何值，因为这些值无论如何都不会被看到。

> Now that we support methods, to add initializers, we merely need to implement those three special rules. We'll go in order.

既然我们支持方法，为了添加初始化式，我们只需要实现这三条特殊规则。我们按顺序进行。

> ### 28.4.1 Invoking initializers

28.4.1 调用初始化器

> First, automatically calling `init()` on new instances:

首先，在新实例上自动调用`init()`：

vm.c，在callValue()方法中添加代码：

```
        vm.stackTop[-argCount - 1] = OBJ_VAL(newInstance(klass));
        // 新增部分开始
        Value initializer;
        if (tableGet(&klass->methods, vm.initString,
                     &initializer)) {
          return call(AS_CLOSURE(initializer), argCount);
        }
        // 新增部分结束
        return true;
```

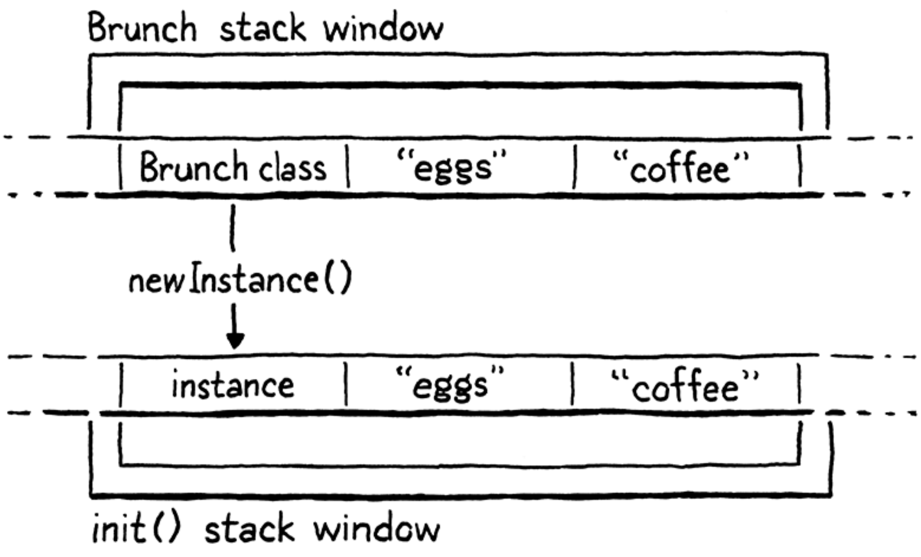
> After the runtime allocates the new instance, we look for an `init()` method on the class. If we find one, we initiate a call to it. This pushes a new CallFrame for the initializer's closure. Say we run this program:

在运行时分配了新实例后，我们在类中寻找`init()`方法。如果找到了，就对其发起调用。这就为初始化器的闭包压入了一个新的CallFrame。假设我们运行这个程序：

```
class Brunch {
  init(food, drink) {}
}

Brunch("eggs", "coffee");
```

> When the VM executes the call to `Brunch()`, it goes like this:

当VM执行对`Brunch()`的调用时，情况是这样的：



> Any arguments passed to the class when we called it are still sitting on the stack above the instance. The new CallFrame for the `init()` method shares that stack window, so those arguments implicitly get forwarded to the initializer.

我们在调用该类时传入的所有参数都仍然在实例上方的栈中。`init()`方法的新CallFrame共享了这个栈窗口，因此这些参数会隐式地转发给初始化器。

> Lox doesn't require a class to define an initializer. If omitted, the runtime simply returns the new uninitialized instance. However, if there is no `init()` method, then it doesn't make any sense to pass arguments to the class when creating the instance. We make that an error.

Lox并不要求类定义初始化器。如果省略，运行时只是简单地返回新的未初始化的实例。然而，如果没有`init()`方法，那么在创建实例时向类传递参数就没有意义了。我们将其当作一个错误。

vm.c，在callValue()方法中添加代码：

```
          return call(AS_CLOSURE(initializer), argCount);
        // 新增部分开始
        } else if (argCount != 0) {
          runtimeError("Expected 0 arguments but got %d.",
                       argCount);
          return false;
        // 新增部分结束
        }
```

> When the class *does* provide an initializer, we also need to ensure that the number of arguments passed matches the initializer's arity. Fortunately, the `call()` helper does that for us already.

当类*确实*提供了初始化器时，我们还需要确保传入参数的数量与初始化器的元数匹配。幸运的是，`call()`辅助函数已经为我们做到了这一点。

> To call the initializer, the runtime looks up the `init()` method by name. We want that to be fast since it happens every time an instance is constructed. That means it would be good to take advantage of the string interning we've already implemented. To do that, the VM creates an ObjString for "init" and reuses it. The string lives right in the VM struct.

为了调用初始化器，运行时会按名称查找`init()`方法。我们希望这个过程是快速的，因为这在每次构造实例时都会发生。这意味着我们可以很好地利用已经实现的字符串驻留。为此，VM为"init"创建了一个ObjString并重用它。这个字符串就位于VM结构体中。

vm.h，在结构体VM中添加代码：

```
  Table strings;
  // 新增部分开始
  ObjString* initString;
  // 新增部分结束
  ObjUpvalue* openUpvalues;
```

> We create and intern the string when the VM boots up.

当虚拟机启动时，我们创建并驻留该字符串。

vm.c，在initVM()方法中添加代码：

```
  initTable(&vm.strings);
  // 新增部分开始
  vm.initString = copyString("init", 4);
  // 新增部分结束
  defineNative("clock", clockNative);
```

> We want it to stick around, so the GC considers it a root.

我们希望它一直存在，因此GC将其视为根。

memory.c，在markRoots()方法中添加代码：

```
  markCompilerRoots();
  // 新增部分开始
  markObject((Obj*)vm.initString);
  // 新增部分结束
}
```

> Look carefully. See any bug waiting to happen? No? It's a subtle one. The garbage collector now reads `vm.initString`. That field is initialized from the result of calling `copyString()`. But copying a string allocates memory, which can trigger a GC. If the collector ran at just the wrong time, it would read `vm.initString` before it had been initialized. So, first we zero the field out.

仔细观察。看到什么潜藏的bug了吗？没有吗？这是一个微妙的问题。垃圾回收器现在读取`vm.initString`。这个字段是由调用`copyString()`的结果来初始化的。但复制字符串会分配内存，这可能会触发GC。如果回收器在错误的时间运行时，它就会在`vm.initString`初始化之前读取它。所以，我们首先将这个字段清零。

vm.c，在initVM()方法中添加代码：

```
  initTable(&vm.strings);
  // 新增部分开始
  vm.initString = NULL;
  // 新增部分结束
  vm.initString = copyString("init", 4);
```

> We clear the pointer when the VM shuts down since the next line will free it.

我们在VM关闭时清除指针，因为下一行会释放它。

vm.c，在freeVM()方法中添加代码：

```
  freeTable(&vm.strings);
  // 新增部分开始
  vm.initString = NULL;
  // 新增部分结束
  freeObjects();
```

> OK, that lets us call initializers.

好，这样我们就可以调用初始化器了。

> ### 28.4.2 Initializer return values

28.4.2 返回值的初始化器

> The next step is ensuring that constructing an instance of a class with an initializer always returns the new instance, and not `nil` or whatever the body of the initializer returns. Right now, if a class defines an initializer, then when an instance is constructed, the VM pushes a call to that initializer onto the CallFrame stack. Then it just keeps on trucking.

下一步是确保用初始化器构造类实例时，总是返回新的实例，而不是`nil`或初始化式返回的任何内容。现在，如果某个类定义了一个初始化器，那么当构建一个实例时，虚拟机会把对该初始化器的调用压入CallFrame栈。然后，它就可以自动被执行了。

> The user's invocation on the class to create the instance will complete whenever that initializer method returns, and will leave on the stack whatever value the initializer puts there. That means that unless the user takes care to put `return this;` at the end of the initializer, no instance will come out. Not very helpful.

只要初始化器方法返回，用户对类的创建实例的调用就会结束，并把初始化器方法放入栈中的值遗留在那里。这意味着，除非用户特意在初始化器的末尾写上`return this;`，否则不会出现任何实例。不太有用。

> To fix this, whenever the front end compiles an initializer method, it will emit different bytecode at the end of the body to return `this` from the method instead of the usual implicit `nil` most functions return. In order to do *that*, the compiler needs to actually know when it is compiling an initializer. We detect that by checking to see if the name of the method we're compiling is "init".

为了解决这个问题，每当前端编译初始化器方法时，都会在主体末尾生成一个特殊的字节码，以便从方法中返回`this`，而不是大多数函数通常会隐式返回的`nil`。为了做到这一点，编译器需要真正知道它在何时编译一个初始化器。我们通过检查正在编译的方法名称是否为"init"进行确认。

compiler.c，在method()方法中添加代码：

```
  FunctionType type = TYPE_METHOD;
  // 新增部分开始
  if (parser.previous.length == 4 &&
      memcmp(parser.previous.start, "init", 4) == 0) {
    type = TYPE_INITIALIZER;
  }
  // 新增部分结束
  function(type);
```

> We define a new function type to distinguish initializers from other methods.

我们定义一个新的函数类型来区分初始化器和其它方法。

compiler.c，在枚举FunctionType中添加代码：

```
  TYPE_FUNCTION,
  // 新增部分开始
  TYPE_INITIALIZER,
  // 新增部分结束
  TYPE_METHOD,
```

> Whenever the compiler emits the implicit return at the end of a body, we check the type to decide whether to insert the initializer-specific behavior.

每当编译器准备在主体末尾发出隐式返回指令时，我们会检查其类型以决定是否插入初始化器的特定行为。

compiler.c，在emitReturn()方法中替换1行：

```
static void emitReturn() {
  // 新增部分开始
  if (current->type == TYPE_INITIALIZER) {
    emitBytes(OP_GET_LOCAL, 0);
  } else {
    emitByte(OP_NIL);
  }
  // 新增部分结束
  emitByte(OP_RETURN);
```

> In an initializer, instead of pushing `nil` onto the stack before returning, we load slot zero, which contains the instance. This `emitReturn()` function is also called when compiling a `return` statement without a value, so this also correctly handles cases where the user does an early return inside the initializer.

在初始化器中，我们不再在返回前将`nil`压入栈中，而是加载包含实例的槽0。在编译不带值的`return`语句时，这个`emitReturn()`函数也会被调用，因此它也能正确处理用户在初始化器中提前返回的情况。

> ### 28.4.3 Incorrect returns in initializers

28.4.3 初始化器中的错误返回

> The last step, the last item in our list of special features of initializers, is making it an error to try to return anything *else* from an initializer. Now that the compiler tracks the method type, this is straightforward.

最后一步，也就是我们的初始化器特性列表中的最后一条，是让试图从初始化器中返回任何*其它*值的行为成为错误。既然编译器跟踪了方法类型，这就很简单了。

compiler.c，在returnStatement()方法中添加代码：

```
  if (match(TOKEN_SEMICOLON)) {
    emitReturn();
  } else {
    // 新增部分开始
    if (current->type == TYPE_INITIALIZER) {
      error("Can't return a value from an initializer.");
    }
    // 新增部分结束
    expression();
```

> We report an error if a `return` statement in an initializer has a value. We still go ahead and compile the value afterwards so that the compiler doesn't get confused by the trailing expression and report a bunch of cascaded errors.

如果初始化式中的`return`语句中有值，则报告一个错误。我们仍然会在后面编译这个值，这样编译器就不会因为被后面的表达式迷惑而报告一堆级联错误。

> Aside from inheritance, which we'll get to soon, we now have a fairly full-featured class system working in clox.

除了继承（我们很快会讲到），我们在clox中有了一个功能相当齐全的类系统。

```
class CoffeeMaker {
  init(coffee) {
    this.coffee = coffee;
  }

  brew() {
    print "Enjoy your cup of " + this.coffee;

    // No reusing the grounds!
    this.coffee = nil;
  }
}

var maker = CoffeeMaker("coffee and chicory");
maker.brew();
```

> Pretty fancy for a C program that would fit on an old floppy disk.

对于一个可以放在旧软盘上的C程序来说，这真是太神奇了^11。

> ## 28.5 Optimized Invocations

28.5 优化调用

> Our VM correctly implements the language's semantics for method calls and initializers. We could stop here. But the main reason we are building an entire second implementation of Lox from scratch is to execute faster than our old Java interpreter. Right now, method calls even in clox are slow.

我们的虚拟机正确地实现了语言中方法调用和初始化器的语义。我们可以到此为止。但是，我们从头开始构建Lox的第二个完整实现的主要原因是，它的执行速度比我们的旧Java解释器要更快。现在，即使在clox中，方法调用也很慢。

> Lox’s semantics define a method invocation as two operations—accessing the method and then calling the result. Our VM must support those as separate operations because the user *can* separate them. You can access a method without calling it and then invoke the bound method later. Nothing we’ve implemented so far is unnecessary.

Lox的语义将方法调用定义为两个操作——访问方法，然后调用结果。我们的虚拟机必须支持这些单独的操作，因为用户 *可以*将它们区分对待。你可以在不调用方法的情况下访问它，接着稍后再调用已绑定的方法。我们目前还未实现的一切内容，都是不必要的。

> But *always* executing those as separate operations has a significant cost. Every single time a Lox program accesses and invokes a method, the runtime heap allocates a new ObjBoundMethod, initializes its fields, then pulls them right back out. Later, the GC has to spend time freeing all of those ephemeral bound methods.

但是，*总是*将它们作为两个单独的操作来执行会产生很大的成本。每次Lox程序访问并调用一个方法时，运行时堆都会分配一个新的ObjBoundMethod，初始化其字段，然后再把这些字段拉出来。之后，GC必须花时间释放所有这些临时绑定的方法。

> Most of the time, a Lox program accesses a method and then immediately calls it. The bound method is created by one bytecode instruction and then consumed by the very next one. In fact, it’s so immediate that the compiler can even textually *see* that it’s happening—a dotted property access followed by an opening parenthesis is most likely a method call.

大多数情况下，Lox程序会访问一个方法并立即调用它。已绑定方法是由一条字节码指令创建的，然后由下一条指令使用。事实上，它是如此直接，以至于编译器甚至可以从文本上看到它的发生——一个带点的属性访问后面跟着一个左括号，很可能是一个方法调用。

> Since we can recognize this pair of operations at compile time, we have the opportunity to emit a new, special instruction that performs an optimized method call.

因为我们可以在编译时识别这对操作，所以我们有机会发出一条新的特殊指令，执行优化过的方法调用^12。

> We start in the function that compiles dotted property expressions.

我们从编译点属性表达式的函数中开始。

compiler.c，在dot()方法中添加代码：

```
  if (canAssign && match(TOKEN_EQUAL)) {
    expression();
    emitBytes(OP_SET_PROPERTY, name);
  // 新增部分开始
  } else if (match(TOKEN_LEFT_PAREN)) {
    uint8_t argCount = argumentList();
    emitBytes(OP_INVOKE, name);
    emitByte(argCount);
    // 新增部分结束
    } else {
```

> After the compiler has parsed the property name, we look for a left parenthesis. If we match one, we switch to a new code path. There, we compile the argument list exactly like we do when compiling a call expression. Then we emit a single new `OP_INVOKE` instruction. It takes two operands:

在编译器解析属性名称之后，我们寻找一个左括号。如果匹配到了，则切换到一个新的代码路径。在那里，我们会像编译调用表达式一样来编译参数列表。然后我们发出一条新的`OP_INVOKE`指令。它需要两个操作数：

> 1. The index of the property name in the constant table.
> 2. The number of arguments passed to the method.

1. 属性名称在常量表中的索引。
2. 传递给方法的参数数量。

> In other words, this single instruction combines the operands of the `OP_GET_PROPERTY` and `OP_CALL` instructions it replaces, in that order. It really is a fusion of those two instructions. Let's define it.

换句话说，这条指令结合了它所替换的`OP_GET_PROPERTY` 和 `OP_CALL`指令的操作数，按顺序排列。它实际上是这两条指令的融合。让我们来定义它。

chunk.h，在枚举OpCode中添加代码：

```
    OP_CALL,
    // 新增部分开始
    OP_INVOKE,
    // 新增部分结束
    OP_CLOSURE,
```

> And add it to the disassembler:

将其添加到反汇编程序：

debug.c，在disassembleInstruction()方法中添加代码：

```
      case OP_CALL:
        return byteInstruction("OP_CALL", chunk, offset);
      // 新增部分开始
      case OP_INVOKE:
        return invokeInstruction("OP_INVOKE", chunk, offset);
      // 新增部分结束
      case OP_CLOSURE: {
```

> This is a new, special instruction format, so it needs a little custom disassembly logic.

这是一种新的、特殊的指令格式，所以需要一些自定义的反汇编逻辑。

debug.c，在constantInstruction()方法后添加代码：

```
static int invokeInstruction(const char* name, Chunk* chunk,
                                int offset) {
  uint8_t constant = chunk->code[offset + 1];
  uint8_t argCount = chunk->code[offset + 2];
  printf("%-16s (%d args) %4d '", name, argCount, constant);
  printValue(chunk->constants.values[constant]);
  printf("'\n");
  return offset + 3;
}
```

> We read the two operands and then print out both the method name and the argument count. Over in the interpreter's bytecode dispatch loop is where the real action begins.

我们读取两个操作数，然后打印出方法名和参数数量。解释器的字节码调度循环中才是真正的行动开始的地方。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_INVOKE: {
        ObjString* method = READ_STRING();
        int argCount = READ_BYTE();
        if (!invoke(method, argCount)) {
          return INTERPRET_RUNTIME_ERROR;
        }
        frame = &vm.frames[vm.frameCount - 1];
        break;
      }
      // 新增部分结束
      case OP_CLOSURE: {
```

> Most of the work happens in `invoke()`, which we'll get to. Here, we look up the method name from the first operand and then read the argument count operand. Then we hand off to `invoke()` to do the heavy lifting. That function returns `true` if the invocation succeeds. As usual, a `false` return means a runtime error occurred. We check for that here and abort the interpreter if disaster has struck.

大部分工作都发生在`invoke()`中，我们会讲到这一点。在这里，我们从第一个操作数中查找方法名称，接着读取参数数量操作数。然后我们交给`invoke()`来完成繁重的工作。如果调用成功，该函数会返回`true`。像往常一样，返回`false`意味着发生了运行时错误。我们在这里进行检查，如果灾难发生就中止解释器的运行。

> Finally, assuming the invocation succeeded, then there is a new CallFrame on the stack, so we refresh our cached copy of the current frame in `frame`.

最后，假如调用成功，那么栈中会有一个新的CallFrame，所以我们刷新`frame`中缓存的当前帧副本。

> The interesting work happens here:

有趣的部分在这里：

vm.c，在callValue()方法后添加代码：

```
static bool invoke(ObjString* name, int argCount) {
  Value receiver = peek(argCount);
  ObjInstance* instance = AS_INSTANCE(receiver);
  return invokeFromClass(instance->klass, name, argCount);
}
```

> First we grab the receiver off the stack. The arguments passed to the method are above it on the stack, so we peek that many slots down. Then it's a simple matter to cast the object to an instance and invoke the method on it.

首先我们从栈中抓取接收器。传递给方法的参数在栈中位于接收器上方，因此我们要查看从上往下跳过多个位置的栈槽。然后，将对象转换成实例并对其调用方法就很简单了。

> That does assume the object *is* an instance. As with `OP_GET_PROPERTY` instructions, we also need to handle the case where a user incorrectly tries to call a method on a value of the wrong type.

它确实假定了对象*是*一个实例。与`OP_GET_PROPERTY`指令一样，我们也需要处理这种情况：用户错误地试图在一个错误类型的值上调用一个方法。

vm.c，在invoke()方法中添加代码：

```
  Value receiver = peek(argCount);
  // 新增部分开始
  if (!IS_INSTANCE(receiver)) {
    runtimeError("Only instances have methods.");
    return false;
  }
  // 新增部分结束
  ObjInstance* instance = AS_INSTANCE(receiver);
```

> That's a runtime error, so we report that and bail out. Otherwise, we get the instance's class and jump over to this other new utility function:

这是一个运行时错误，所以我们报告错误并退出。否则，我们获取实例的类并跳转到另一个新的工具函数^13：

vm.c，在callValue()方法后添加代码：

```
static bool invokeFromClass(ObjClass* klass, ObjString* name,
                            int argCount) {
  Value method;
  if (!tableGet(&klass->methods, name, &method)) {
    runtimeError("Undefined property '%s'.", name->chars);
    return false;
  }
  return call(AS_CLOSURE(method), argCount);
}
```

> This function combines the logic of how the VM implements `OP_GET_PROPERTY` and `OP_CALL` instructions, in that order. First we look up the method by name in the class's method table. If we don't find one, we report that runtime error and exit.

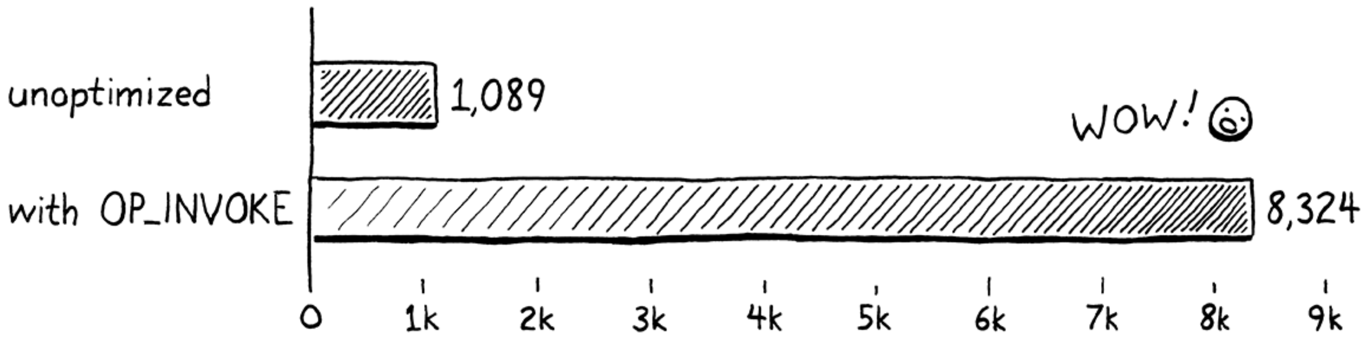
这个函数按顺序结合了VM中实现`OP_GET_PROPERTY` 和`OP_CALL`指令的逻辑。首先，我们在类的方法表中按名称查找方法。如果没有找到，则报告错误并退出。

> Otherwise, we take the method's closure and push a call to it onto the CallFrame stack. We don't need to heap allocate and initialize an ObjBoundMethod. In fact, we don't even need to juggle anything on the stack. The receiver and method arguments are already right where they need to be.

否则，我们获取方法闭包并将对它的调用压入CallFrame栈。我们不需要在堆上分配并初始化ObjBoundMethod。实际上，我们甚至不需要在栈上做什么操作。接收器和方法参数已经位于它们应在的位置了^14。

> If you fire up the VM and run a little program that calls methods now, you should see the exact same behavior as before. But, if we did our job right, the *performance* should be much improved. I wrote a little microbenchmark that does a batch of 10,000 method calls. Then it tests how many of these batches it can execute in 10 seconds. On my computer, without the new `OP_INVOKE` instruction, it got through 1,089 batches. With this new optimization, it finished 8,324 batches in the same time. That's *7.6 times faster*, which is a huge improvement when it comes to programming language optimization.

如果你现在启动虚拟机并运行一个调用方法的小程序，你应该会看到和以前完全相同的行为。但是，如果我们的工作做得好，*性能*应该会大大提高。我写了一个小小的微基准测试，执行每批10000次方法调用。然后测试在10秒钟内可以执行多少个批次。在我的电脑上，如果没有新的`OP_INVOKE`指令，它完成了1089个批次。通过新的优化，它在相同的时间中完成了8324个批次。速度提升了7.6倍，对于编程语言优化来说，这是一个巨大的改进^15。



> ### 28.5.1 Invoking fields

28.5.1 调用字段

> The fundamental creed of optimization is: "Thou shalt not break correctness." Users like it when a language implementation gives them an answer faster, but only if it's the *right* answer. Alas, our implementation of faster method invocations fails to uphold that principle:

优化的基本信条是："你不应该破坏正确性"。用户喜欢语言实现能更快地给出答案，但前提是这个答案是正确的^16。唉，我们这个快速的方法调用实现并没有坚持这一原则：

```
class Oops {
  init() {
    fun f() {
      print "not a method";
    }

    this.field = f;
  }
}

var oops = Oops();
oops.field();
```

> The last line looks like a method call. The compiler thinks that it is and dutifully emits an `OP_INVOKE` instruction for it. However, it's not. What is actually happening is a *field* access that returns a function which then gets called. Right now, instead of executing that correctly, our VM reports a runtime error when it can't find a method named "field".

最后一行看起来像是一个方法调用。编译器认为它是，并尽职尽责地为它发出一条`OP_INVOKE`指令。然而，事实并非如此。实际发生的是一个*字段*访问，它会返回一个函数，然后该函数被调用。现在，我们的虚拟机没有正确地执行它，而在找不到名为"field"的方法时报告一个运行时错误。

> Earlier, when we implemented `OP_GET_PROPERTY`, we handled both field and method accesses. To squash this new bug, we need to do the same thing for `OP_INVOKE`.

之前，当我实现`OP_GET_PROPERTY`时，我们同时处理了字段和方法的访问。为了消除这个新bug，我们需要对`OP_INVOKE`做同样的事情。

vm.c，在invoke()方法中添加代码：

```
  ObjInstance* instance = AS_INSTANCE(receiver);
  // 新增部分开始
  Value value;
  if (tableGet(&instance->fields, name, &value)) {
    vm.stackTop[-argCount - 1] = value;
    return callValue(value, argCount);
  }
  // 新增部分结束
  return invokeFromClass(instance->klass, name, argCount);
```

> Pretty simple fix. Before looking up a method on the instance's class, we look for a field with the same name. If we find a field, then we store it on the stack in place of the receiver, *under* the argument list. This is how `OP_GET_PROPERTY` behaves since the latter instruction executes before a subsequent parenthesized list of arguments has been evaluated.

非常简单的解决方法。在查找实例类上的方法之前，我们先查找具有相同名称的字段。如果我们找到一个字段，那我们就将其存储在栈中代替接收器，放在参数列表 *下面*。这就是`OP_GET_PROPERTY`的行为方式，因为后者的指令执行时机是在随后括号内的参数列表被求值之前。

> Then we try to call that field's value like the callable that it hopefully is. The `callValue()` helper will check the value's type and call it as appropriate or report a runtime error if the field's value isn't a callable type like a closure.

然后，我们尝试调用该字段的值（就像它如期望的那样是可调用的）。`callValue()`辅助函数会检查值的类型并适当地调用它，如果该字段的值不是像闭包这样的可调用类型，则报告运行时错误。

> That's all it takes to make our optimization fully safe. We do sacrifice a little performance, unfortunately. But that's the price you have to pay sometimes. You occasionally get frustrated by optimizations you *could* do if only the language wouldn't allow some annoying corner case. But, as language implementers, we have to play the game we're given.

这就是使我们的优化完全安全的全部工作。不幸的是，我们确实牺牲了一点性能。但有时候这是你必须要付出的代价。如果语言不允许出现一些令人讨厌的极端情况，你可能会对某些 *可做*的优化感到沮丧。但是，作为语言实现者，我们必须玩我们被赋予的游戏^17。

> The code we wrote here follows a typical pattern in optimization:

我们在这里编写的代码遵循一个优化中的典型模式：

> 1. Recognize a common operation or sequence of operations that is performance critical. In this case, it is a method access followed by a call.
> 2. Add an optimized implementation of that pattern. That's our `OP_INVOKE` instruction.
> 3. Guard the optimized code with some conditional logic that validates that the pattern actually applies. If it does, stay on the fast path. Otherwise, fall back to a slower but more robust unoptimized behavior. Here, that means checking that we are actually calling a method and not accessing a field.

1. 识别出对性能至关重要的常见操作或操作序列。在本例中，它是一个方法访问后跟一个调用。
2. 添加该模式的优化实现。也就是我们的`OP_INVOKE`指令。
3. 用一些条件逻辑来验收是否适用该模式，从而保护优化后的代码。如果适用，就走捷径。否则，就退回到较慢但更稳健的非优化行为。在这里，意味着要检查我们是否真的在调用一个方法而不是访问一个字段。

> As your language work moves from getting the implementation working *at all* to getting it to work *faster*, you will find yourself spending more and more time looking for patterns like this and adding guarded optimizations for them. Full-time VM engineers spend much of their careers in this loop.

随着你的语言工作从让语言实现完全工作到让它更快工作，你会发现自己花费了越来越多的时间来寻找这样的模式，并为它们添加保护性优化。全职虚拟机工程师的大部分职业生涯都是在这个循环中度过的。

> But we can stop here for now. With this, clox now supports most of the features of an object-oriented programming language, and with respectable performance.

但是我们可以到此为止了。有了这些，clox现在支持面向对象编程语言的大部分特性，而且具有不错的性能。

^5: 我从CPython中借鉴了“bound method”这个名字。Python跟Lox这里的行为很类似，我通过它的实现获得灵感。 ^6: 跟踪方法的闭包实际上是没有必要的。接收器是一个ObjInstance，它有一个指向其ObjClass的指针，而ObjClass有一个存储所有方法的表。但让ObjBoundMethod依赖于它，我觉得在某种程度上是值得怀疑的。 ^7: 已绑定方法是第一类值，所以他们可以把它存储在变量中，传递给函数，以及用它做“值”可做的事情。 ^8: 解析器函数名称后面的下划线是因为`this`是C++中的一个保留字，我们支持将clox编译为C++。^9: 当然，Lox确实允许外部代码之间访问和修改一个实例的字段，而不需要通过实例的方法。这与Ruby和Smalltalk不同，后者将状态完全封装在对象中。我们的玩具式脚本语言，唉，不那么有原则。 ^10: 就好像初始化器被隐式地包装在这样的一段代码中：

```
fun create(klass) {
  var obj = newInstance(klass);
  obj.init();
  return obj;
}
```

注意`init()`返回的值是如何被丢弃的。 ^11: 我承认，“软盘”对于当前一代程序员来说，可能不再是一个有用的大小参考。也许我应该说“几条推特”之类的。 ^12: 如果你花足够的时间观察字节码虚拟机的运行，你会发现它经常一次次地执行同一系列的字节码指令。一个经典的优化技术是定义新的单条指令，称为**超级指令**，它将这些指令融合到具有与整个序列相同行为的单一指令。
在字节码解释器中，最大的性能消耗之一是每个指令的解码和调度的开销。将几个指令融合在一起可以消除其中的一些问题。
难点在于确定*哪些*指令序列足够常见，并可以从这种优化中受益。每条新的超级指令都要求有一个操作码供自己使用，而这些操作码的数量是有限的。如果添加太多，你就需要对操作码进行更长的编码，这就增加了代码的大小，使得所有指令的解码速度变慢。 ^13: 你应该可以猜到，我们将这段代码拆分成一个单独的函数，是因为我们稍后会复用它——`super`调用中。 ^14: 这就是我们使用栈槽0来存储接收器的一个主要原因——调用方就是这样组织方法调用栈的。高效的调用约定是字节码虚拟机性能故事的重要组成部分。 ^15: 我们不应该过于自信。这种性能优化是相对于我们自己未优化的方法调用实现而言的，而那种方法调用实现相当缓慢。为每个方法调用都进行堆分配不会赢得任何比赛。 ^16: 在有些情况下，当程序偶尔返回错误的答案，以换取显著加快的运行速度或更好的性能边界，用户可能也是满意的。这些就是**蒙特卡洛算法**的领域。对于某些用例来说，这是一个很好的权衡。
不过，重要的是，由用户选择使用这些算法中的某一种。我们这些语言实现者不能单方面地决定牺牲程序的正确性。 ^17: 作为语言*设计者*，我们的角色非常不同。如果我们确实控制了语言本身，我们有时可能会选择限制或改变语言的方式来实现优化。用户想要有表达力的语言，但他们也想要快速实现。有时，如果牺牲一点功能来获得完美回报是很好的语言设计。

习题

1. > The hash table lookup to find a class's `init()` method is constant time, but still fairly slow. Implement something faster. Write a benchmark and measure the performance difference.

   哈希表中查找类的`init()`方法是常量时间复杂度，但仍然相当慢。实现一些更快的方法。写一个基准测试并度量性能差异。

2. > In a dynamically typed language like Lox, a single callsite may invoke a variety of methods on a number of classes throughout a program's execution. Even so, in practice, most of the time a callsite ends up calling the exact same method on the exact same class for the duration of the run. Most calls are actually not polymorphic even if the language says they can be.
   >
   > How do advanced language implementations optimize based on that observation?

在像Lox这样的动态类型语言中，程序执行过程中的同样的一个调用可能会调用多个类上的多个方法。即便如此，在实践中，大多数情况下某个调用在运行期间会执行同一个类上的同一个方法。大多数调用实际上不是多态的，即使语言说它们是多态的。

高级的语言实现是如何基于这一观察进行优化的？

3. > When interpreting an `OP_INVOKE` instruction, the VM has to do two hash table lookups. First, it looks for a field that could shadow a method, and only if that fails does it look for a method. The former check is rarely useful—most fields do not contain functions. But it is *necessary* because the language says fields and methods are accessed using the same syntax, and fields shadow methods.
   >
   > That is a language *choice* that affects the performance of our implementation. Was it the right choice? If Lox were your language, what would you do?

   在解释`OP_INVOKE`指令时，虚拟机必须执行两次哈希表查询。首先，它要查找可能会遮蔽方法的字段，只有这一步失败时才会查找方法。前一个检查很少有用——大多数字段都不包含函数。但它是*必要*的，因为语言要求字段和方法通过同样的语法来访问，并且字段会遮蔽方法。

   这是一种影响实现性能的语言选择。这是个正确的选择吗？如果Lox是你的语言，你会怎么做？

设计笔记：新奇性预算

> I still remember the first time I wrote a tiny BASIC program on a TRS-80 and made a computer do something it hadn't done before. It felt like a superpower. The first time I cobbled together just enough of a parser and interpreter to let me write a tiny program in *my own language* that made a computer do a thing was like some sort of higher-order meta-superpower. It was and remains a wonderful feeling.

我还记得我第一次在TRS-8上写了一个小小的BASIC程序，让电脑做了一些它以前没有做过的事。这感觉就像是一种超能力。我第一次组装出一个解析器和解释器，能够让我使用*自己的语言*写一个小程序，让计算机做了一件事，就像某种更高阶的超能力。这种感觉过去是美妙的，现在仍然是。

> I realized I could design a language that looked and behaved however I chose. It was like I'd been going to a private school that required uniforms my whole life and then one day transferred to a public school where I could wear whatever I wanted. I don't need to use curly braces for blocks? I can use something other than an equals sign for assignment? I can do objects without classes? Multiple inheritance *and* multimethods? A dynamic language that overloads statically, by arity?

我意识到，我可以设计一种外观和行为都由我选择的语言。就好像我一直在一所要求穿制服的私立学校上学，然后有一天转到了一所公立学校，在那里我想穿什么就穿什么。我不要使用大括号来表示代码块？我可以用等号以外的符号进行赋值？我可以实现没有类的对象？多重继承和多分派？根据元数进行静态重载的动态语言？

> Naturally, I took that freedom and ran with it. I made the weirdest, most arbitrary language design decisions. Apostrophes for generics. No commas between arguments. Overload resolution that can fail at runtime. I did things differently just for difference's sake.

很自然地，我接受了这种自由。我做了最古怪、最随意的语言设计决策。撇号表示泛型，参数之间不使用逗号，在运行时可能会失败的重载解析。我做了一些不同的事情，只是为了与众不同。

> This is a very fun experience that I highly recommend. We need more weird, avant-garde programming languages. I want to see more art languages. I still make oddball toy languages for fun sometimes.

这是一个非常有趣的体验，我强烈推荐。我们需要更多奇怪、前卫的编程语言。我希望看到更多的艺术语言。有时候我还会做一些奇怪的玩具语言来玩。

> *However*, if your goal is success where "success" is defined as a large number of users, then your priorities must be different. In that case, your primary goal is to have your language loaded into the brains of as many people as possible. That's *really hard*. It takes a lot of human effort to move a language's syntax and semantics from a computer into trillions of neurons.

然而，如果你的目标是成功，而"成功"被定义为大量的用户，那么你的优先事项必然是不同的。在这种情况下，你的首要目标是让尽可能多的人记住你的语言。这*真的*很难。要将一种语言的语法和语义从计算机转移到数万亿的神经元中，需要付出大量的努力。

> Programmers are naturally conservative with their time and cautious about what languages are worth uploading into their wetware. They don't want to waste their time on a language that ends up not being useful to them. As a language designer, your goal is thus to give them as much language power as you can with as little required learning as possible.

程序员对他们的时间自然是保守的，对于哪些语言值得上传到他们的湿件（即大脑）中。他们不想把时间浪费在一门最终对他们没有用处的语言上。因此，作为语言设计者，你的目标是为他们提供尽可能多的语言能力，并尽可能地减少所需的学习。

> One natural approach is *simplicity*. The fewer concepts and features your language has, the less total volume of stuff there is to learn. This is one of the reasons minimal scripting languages often find success even though they aren't as powerful as the big industrial languages—they are easier to get started with, and once they are in someone's brain, the user wants to keep using them.

一个自然的方法是*简单化*。你的语言拥有的概念和功能越少，你需要学习的东西就越少。这就是小型脚本语言虽然不像大型工业语言那样强大却经常获得成功的原因之一——它们更容易上手，而且它们一旦进入了人们的大脑，用户就想继续使用它们^18。

> The problem with simplicity is that simply cutting features often sacrifices power and expressiveness. There is an art to finding features that punch above their weight, but often minimal languages simply do less.

简单化的问题在于，简单地删减功能通常会牺牲功能和表现力。找到超越其重量的功能是一门艺术，但通常小型语言做得更少，

> There is another path that avoids much of that problem. The trick is to realize that a user doesn't have to load your entire language into their head, *just the part they don't already have in there*. As I mentioned in an earlier design note, learning is about transferring the *delta* between what they already know and what they need to know.

还有另一种方法可以避免这个问题。诀窍是要意识到，用户不必将你的整个语言都装进他们的脑子里，只需要把他们*还没有的部分*装进去就行了。正如我在之前的设计笔记中提到的，学习是转移他们已知内容与需要知道的内容之间的差量。

> Many potential users of your language already know some other programming language. Any features your language shares with that language are essentially "free" when it comes to learning. It's already in

> their head, they just have to recognize that your language does the same thing.

你的语言的许多潜在用户已经了解了一些其它的编程语言。当涉及到学习时，你的语言与这些语言共享的任何功能基本上都是“免费”的。它已经在用户的头脑中了，他们只需要认识到你的语言也做了同样的事情。

> In other words, *familiarity* is another key tool to lower the adoption cost of your language. Of course, if you fully maximize that attribute, the end result is a language that is completely identical to some existing one. That’s not a recipe for success, because at that point there’s no incentive for users to switch to your language at all.

换句话说，*熟悉度*是降低语言采用成本的另一个关键工具。当然，如果你将这一属性完全最大化，最终的结果就是一门与某些现有语言完全相同的语言。这不是成功的秘诀，因为在这一点上，用户根本没有切换到你的语言的动力。

> So you do need to provide some compelling differences. Some things your language can do that other languages can’t, or at least can’t do as well. I believe this is one of the fundamental balancing acts of language design: similarity to other languages lowers learning cost, while divergence raises the compelling advantages.

所以你确实需要提供一些令人信服的差异。某些事情你的语言可以做到，而其它语言做不到，或者至少做得不如你的好。我相信这是语言设计的基本平衡行为之一：与其它语言的相似性降低了学习成本，而差异性提高了令人信服的优势。

> I think of this balancing act in terms of a **novelty budget**, or as Steve Klabnik calls it, a “strangeness budget”. Users have a low threshold for the total amount of new stuff they are willing to accept to learn a new language. Exceed that, and they won’t show up.

我认为这种平衡就像是**新奇性预算**，或者像Steve Klabnik所说，是一种“陌生感预算”^19。用户对于学习新语言时愿意接受的新知识的总量有一个较低的阈值。如果超过这个值，他们就不会来学习了。

> Anytime you add something new to your language that other languages don’t have, or anytime your language does something other languages do in a different way, you spend some of that budget. That’s OK—you *need* to spend it to make your language compelling. But your goal is to spend it *wisely*. For each feature or difference, ask yourself how much compelling power it adds to your language and then evaluate critically whether it pays its way. Is the change so valuable that it is worth blowing some of your novelty budget?

任何时候，你为你的语言添加了其它语言没有的新东西，或者你的语言以不同的方式做了其它语言做的事情，你都会花费一下预算。这没关系——你*需要*花费预算来使你的语言更具有吸引力。但你的目标是明智地使用这些预算。对于每一种特性或差异，问问你自己它为你的语言增加了多少引人注目的能力，然后严格评估它是否值得。这种改变是否有价值，而且值得你花费一些新奇性预算？

> In practice, I find this means that you end up being pretty conservative with syntax and more adventurous with semantics. As fun as it is to put on a new change of clothes, swapping out curly braces with some other block delimiter is very unlikely to add much real power to the language, but it does spend some novelty. It’s hard for syntax differences to carry their weight.

在实践中，我发现这意味着你最终会在语法上相当保守，而在语义上更加大胆。虽然换一套新衣服很有趣，但把花括号换成其它代码块分隔符并不可能给语言增加多少真正的能力，但它确实会花费一些新奇性。语法上的差异很难承载它们的重量。

> On the other hand, new semantics can significantly increase the power of the language. Multimethods, mixins, traits, reflection, dependent types, runtime metaprogramming, etc. can radically level up what a user can do with the language.

另一方面，新的语义可以显著增加语言的能力。多分派、mixins、traits、反射、依赖类型、运行时元编程等可以从根本上提升用户使用语言的能力。

> Alas, being conservative like this is not as fun as just changing everything. But it's up to you to decide whether you want to chase mainstream success or not in the first place. We don't all need to be radio-friendly pop bands. If you want your language to be like free jazz or drone metal and are happy with the proportionally smaller (but likely more devoted) audience size, go for it.

唉，这样的保守并不像直接改变一切那样有趣。但是否追求主流的成功，首先取决于你。我们不需要都成为电台欢迎的流行乐队。如果你想让你的语言像自由爵士或嗡鸣金属那样，并且对比较较小（但可能更忠实）的观众数量感到满意，那就去做吧。

29.超类 Superclasses

> You can choose your friends but you sho' can't choose your family, an' they're still kin to you no matter whether you acknowledge 'em or not, and it makes you look right silly when you don't.
>
> —— Harper Lee, *To Kill a Mockingbird*

你可以选择你的朋友，但无法选择你的家庭，所以不管你承认与否，他们都是你的亲属，而且不承认会让你显得很蠢。（哈珀·李，《杀死一只知更鸟》）

> This is the very last chapter where we add new functionality to our VM. We've packed almost the entire Lox language in there already. All that remains is inheriting methods and calling superclass methods. We have another chapter after this one, but it introduces no new behavior. It only makes existing stuff faster. Make it to the end of this one, and you'll have a complete Lox implementation.

这是我们向虚拟机添加新功能的最后一章。我们已经把几乎所有的Lox语言都装进虚拟机中了。剩下的就是继承方法和调用超类方法。在本章之后还有一章，但是没有引入新的行为。它只是让现有的东西更快^1。坚持到本章结束，你将拥有一个完整的Lox实现。

> Some of the material in this chapter will remind you of jlox. The way we resolve super calls is pretty much the same, though viewed through clox's more complex mechanism for storing state on the stack. But we have an entirely different, much faster, way of handling inherited method calls this time around.

本章中的一些内容会让你想起jlox。我们解决超类调用的方式几乎是一样的，即便是从clox这种在栈中存储状态的更复杂的机制来看。但这次我们会用一种完全不同的、更快的方式来处理继承方法的调用。

> ## 29.1 Inheriting Methods

29.1 继承方法

> We'll kick things off with method inheritance since it's the simpler piece. To refresh your memory, Lox inheritance syntax looks like this:

我们会从方法继承开始，因为它是比较简单的部分。为了恢复你的记忆，Lox的继承语法如下所示：

```
class Doughnut {
  cook() {
    print "Dunk in the fryer.";
  }
}

class Cruller < Doughnut {
  finish() {
    print "Glaze with icing.";
  }
}
```

> Here, the Cruller class inherits from Doughnut and thus, instances of Cruller inherit the `cook()` method. I don't know why I'm belaboring this. You know how inheritance works. Let's start compiling the new syntax.

这里，Culler类继承自Doughnut，因此，Cruller的实例继承了`cook()`方法。我不明白我为什么要反复强调这个，你知道继承是怎么回事。让我们开始编译新语法。

compiler.c，在classDeclaration()方法中添加代码：

```
  currentClass = &classCompiler;
  // 新增部分开始
  if (match(TOKEN_LESS)) {
    consume(TOKEN_IDENTIFIER, "Expect superclass name.");
    variable(false);
    namedVariable(className, false);
    emitByte(OP_INHERIT);
  }
  // 新增部分结束
  namedVariable(className, false);
```

> After we compile the class name, if the next token is a `<`, then we found a superclass clause. We consume the superclass's identifier token, then call `variable()`. That function takes the previously consumed token, treats it as a variable reference, and emits code to load the variable's value. In other words, it looks up the superclass by name and pushes it onto the stack.

在编译类名之后，如果下一个标识是`<`，那我们就找到了一个超类子句。我们消耗超类的标识符，然后调用`variable()`。该函数接受前面消耗的标识，将其视为变量引用，并发出代码来加载变量的值。换句话说，它通过名称查找超类并将其压入栈中。

> After that, we call `namedVariable()` to load the subclass doing the inheriting onto the stack, followed by an `OP_INHERIT` instruction. That instruction wires up the superclass to the new subclass. In the last chapter, we defined an `OP_METHOD` instruction to mutate an existing class object by adding a method to its method table. This is similar—the `OP_INHERIT` instruction takes an existing class and applies the effect of inheritance to it.

之后，我们调用`namedVariable()`将进行继承的子类加载到栈中，接着是`OP_INHERIT`指令。该指令将超类与新的子类连接起来。在上一章中，我们定义了一条`OP_METHOD`指令，通过向已有类对象的方法表中添加方法来改变它。这里是类似的——`OP_INHERIT`指令接受一个现有的类，并对其应用继承的效果。

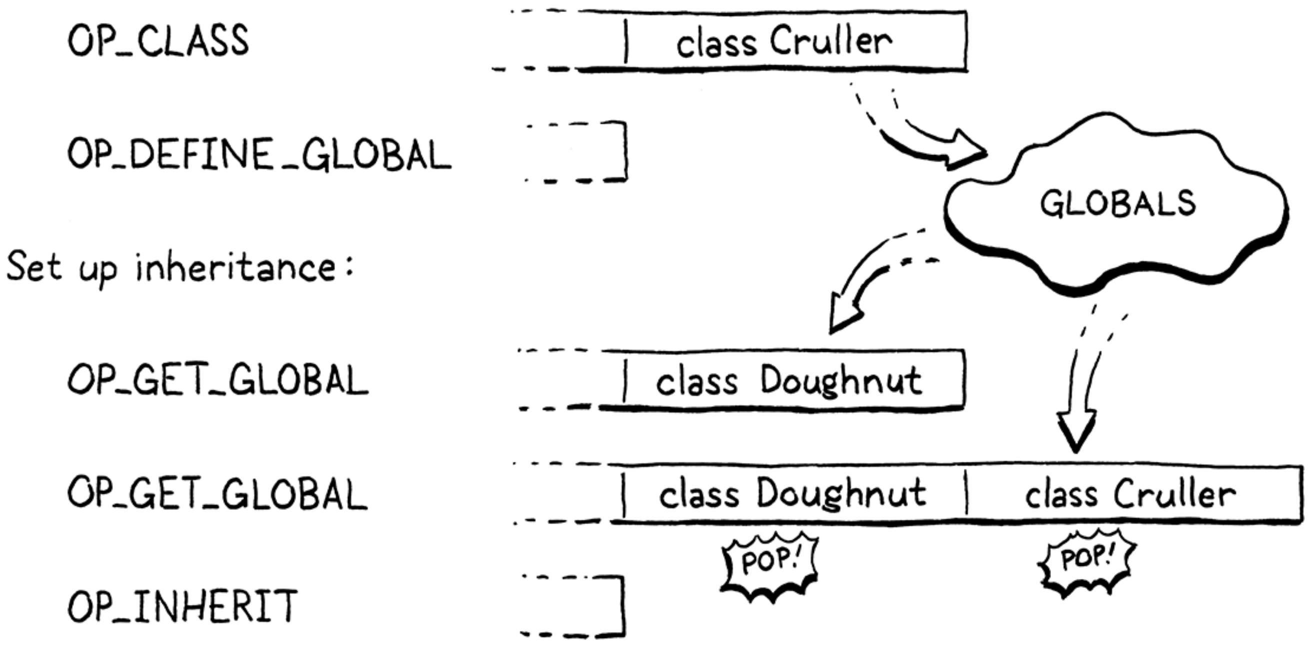
> In the previous example, when the compiler works through this bit of syntax:

在前面的例子中，当编译器处理这些语法时：

```
class Cruller < Doughnut {
```

> The result is this bytecode:

结果就是这个字节码：



> Before we implement the new `OP_INHERIT` instruction, we have an edge case to detect.

在我们实现新的`OP_INHERIT`指令之前，还需要检测一个边界情况。

compiler.c，在classDeclaration()方法中添加代码：

```
    variable(false);
    // 新增部分开始
    if (identifiersEqual(&className, &parser.previous)) {
      error("A class can't inherit from itself.");
    }
    // 新增部分结束
    namedVariable(className, false);
```

> A class cannot be its own superclass. Unless you have access to a deranged nuclear physicist and a very heavily modified DeLorean, you cannot inherit from yourself.

一个类不能成为它自己的超类^2。除非你能接触到一个核物理学家和一辆改装过的DeLorean汽车【译者注：电影《回到未来》的梗】，否则你无法继承自己。

> ### 29.1.1 Executing inheritance

29.1.1 执行继承

> Now onto the new instruction.

现在来看新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_CLASS,
  // 新增部分开始
  OP_INHERIT,
  // 新增部分结束
  OP_METHOD
```

> There are no operands to worry about. The two values we need—superclass and subclass—are both found on the stack. That means disassembling is easy.

不需要担心任何操作数。我们需要的两个值——超类和子类——都可以在栈中找到。这意味着反汇编很容易。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return constantInstruction("OP_CLASS", chunk, offset);
    // 新增部分开始
    case OP_INHERIT:
      return simpleInstruction("OP_INHERIT", offset);
    // 新增部分结束
    case OP_METHOD:
```

> The interpreter is where the action happens.

解释器是行为发生的地方。

vm.c，在run()方法中添加代码：

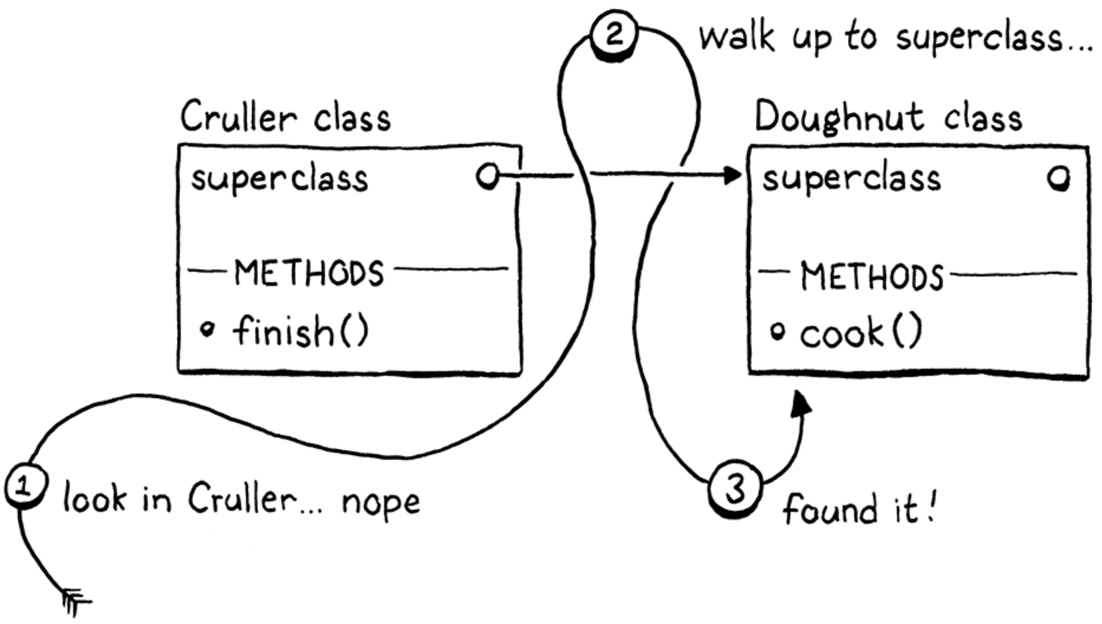
```
        break;
      // 新增部分开始
      case OP_INHERIT: {
        Value superclass = peek(1);
        ObjClass* subclass = AS_CLASS(peek(0));
        tableAddAll(&AS_CLASS(superclass)->methods,
                    &subclass->methods);
        pop(); // Subclass.
        break;
      }
      // 新增部分结束
      case OP_METHOD:
```

> From the top of the stack down, we have the subclass then the superclass. We grab both of those and then do the inherit-y bit. This is where clox takes a different path than jlox. In our first interpreter, each subclass stored a reference to its superclass. On method access, if we didn't find the method in the subclass's method table, we recursed through the inheritance chain looking at each ancestor's method table until we found it.

从栈顶往下，我们依次有子类，然后是超类。我们获取这两个类，然后进行继承。这就是clox与jlox不同的地方。在我们的第一个解释器中，每个子类都存储了一个对其超类的引用。在访问方法时，如果我们没有在子类方法表中找到它，就通过继承链递归遍历每个祖先的方法表，直到找到该方法。

> For example, calling `cook()` on an instance of Cruller sends jlox on this journey:

例如，在Cruller的实例上调用`cook()`方法，jlox会这样做：

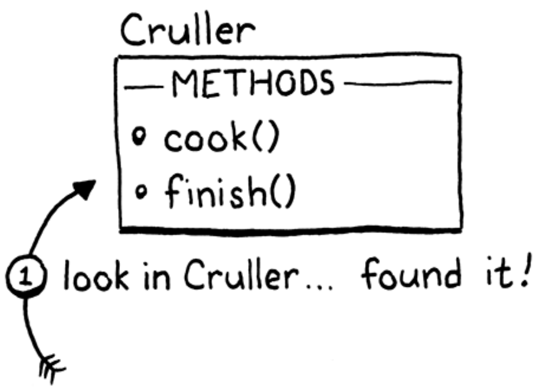


> That's a lot of work to perform during method *invocation* time. It's slow, and worse, the farther an inherited method is up the ancestor chain, the slower it gets. Not a great performance story.

在方法*调用*期间要做大量的工作。这很慢，而且更糟糕的是，继承的方法在祖先链上越远，它就越慢。这不是一个好的性能故事。

> The new approach is much faster. When the subclass is declared, we copy all of the inherited class's methods down into the subclass's own method table. Later, when *calling* a method, any method inherited from a superclass will be found right in the subclass's own method table. There is no extra runtime work needed for inheritance at all. By the time the class is declared, the work is done. This means inherited method calls are exactly as fast as normal method calls—a single hash table lookup.

新方法则要快得多。当子类被声明时，我们将继承类的所有方法复制到子类自己的方法表中。之后，当我们*调用*某个方法时，从超类继承的任何方法都可以在子类自己的方法表中找到。继承根本不需要做额外的运行时工作。当类被声明时，工作就完成了。这意味着继承的方法和普通方法调用一样快——只需要一次哈希表查询^3。



> I've sometimes heard this technique called "copy-down inheritance". It's simple and fast, but, like most optimizations, you get to use it only under certain constraints. It works in Lox because Lox classes are *closed*. Once a class declaration is finished executing, the set of methods for that class can never change.

我有时听到这种技术被称为"向下复制继承"。它简单而快速，但是，与大多数优化一样，你只能在特定的约束条件下使用它。它适用于Lox，是因为Lox的类是*关闭*的。一旦某个类的声明执行完毕，该类的方法集就永远不能更改。

> In languages like Ruby, Python, and JavaScript, it's possible to crack open an existing class and jam some new methods into it or even remove them. That would break our optimization because if those modifications happened to a superclass *after* the subclass declaration executed, the subclass would not pick up those changes. That breaks a user's expectation that inheritance always reflects the current state of the superclass.

在Ruby、Python和JavaScript等语言中，可以打开一个现有的类，并将一些新方法加入其中，甚至删除方法。这会破坏我们的优化，因为如果这些修改在子类声明执行*之后*发生在超类上，子类就不会获得这些变化。这就打破了用户的期望，即继承总是反映超类的当前状态^4。

> Fortunately for us (but not for users who like the feature, I guess), Lox doesn't let you patch monkeys or punch ducks, so we can safely apply this optimization.

幸运的是（我猜对于喜欢这一特性的用户来说不算幸运），Lox不允许猴子补丁或鸭子打洞，所以我们可以安全的应用这种优化。

> What about method overrides? Won't copying the superclass's methods into the subclass's method table clash with the subclass's own methods? Fortunately, no. We emit the `OP_INHERIT` after the `OP_CLASS` instruction that creates the subclass but before any method declarations and `OP_METHOD` instructions have been compiled. At the point that we copy the superclass's methods down, the subclass's method table is empty. Any methods the subclass overrides will overwrite those inherited entries in the table.

那方法重写呢？将超类的方法复制到子类的方法表中，不会与子类自己的方法发生冲突吗？幸运的是，不会。我们是在创建子类的`OP_CLASS`指令之后、但在任何方法声明和`OP_METHOD`指令被编译之前发出`OP_INHERIT`指令。当我们将超类的方法复制下来时，子类的方法表是空的。子类重写的任何方法都会覆盖表中那些继承的条目。

> ### 29.1.2 Invalid superclasses

29.1.2 无效超类

> Our implementation is simple and fast, which is just the way I like my VM code. But it's not robust. Nothing prevents a user from inheriting from an object that isn't a class at all:

我们的实现简单而快速，这正是我喜欢我的VM代码的原因。但它并不健壮。没有什么能阻止用户继承一个根本不是类的对象：

```
var NotClass = "So not a class";
class OhNo < NotClass {}
```

> Obviously, no self-respecting programmer would write that, but we have to guard against potential Lox users who have no self respect. A simple runtime check fixes that.

显然，任何一个有自尊心的程序员都不会写这种东西，但我们必须堤防那些没有自尊心的潜在Lox用户。一个简单的运行时检查就可以解决这个问题。

vm.c，在run()方法中添加代码：

```
        Value superclass = peek(1);
        // 新增部分开始
        if (!IS_CLASS(superclass)) {
          runtimeError("Superclass must be a class.");
          return INTERPRET_RUNTIME_ERROR;
        }
        // 新增部分结束
        ObjClass* subclass = AS_CLASS(peek(0));
```

> If the value we loaded from the identifier in the superclass clause isn't an ObjClass, we report a runtime error to let the user know what we think of them and their code.

如果我们从超类子句的标识符中加载到的值不是ObjClass，就报告一个运行时错误，让用户知道我们对他们及其代码的看法。

> ## 29.2 Storing Superclasses

29.2 存储超类

> Did you notice that when we added method inheritance, we didn't actually add any reference from a subclass to its superclass? After we copy the inherited methods over, we forget the superclass entirely. We don't need to keep a handle on the superclass, so we don't.

你是否注意到，在我们添加方法继承时，实际上并没有添加任何从子类指向超类的引用？我们把继承的方法复制到子类之后，就完全忘记了超类。我们不需要保存超类的句柄，所以我们没有这样做。

> That won't be sufficient to support super calls. Since a subclass may override the superclass method, we need to be able to get our hands on superclass method tables. Before we get to that mechanism, I want to refresh your memory on how super calls are statically resolved.

这不足以支持超类调用。因为子类可能会覆盖超类方法^5，我们需要能够获得超类方法表。在讨论这个机制之前，我想让你回忆一下如何静态解析超类调用。

> Back in the halcyon days of jlox, I showed you this tricky example to explain the way super calls are dispatched:

回顾jlox的光辉岁月，我给你展示了这个棘手的示例，来解释超类调用的分派方式：

```
class A {
  method() {
    print "A method";
  }
}

class B < A {
  method() {
    print "B method";
  }

  test() {
    super.method();
  }
}

class C < B {}

C().test();
```

> Inside the body of the `test()` method, `this` is an instance of C. If super calls were resolved relative to the superclass of the *receiver*, then we would look in C's superclass, B. But super calls are resolved relative to the superclass of the *surrounding class where the super call occurs*. In this case, we are in B's `test()` method, so the superclass is A, and the program should print "A method".

在`test()`方法的主体中，`this`是C的一个实例。如果超类调用是在*接收器*的超类中来解析的，那我们会在C的超类B中寻找方法。但是超类调用是在*发生超类调用的外围类*的超类中解析的。在本例中，我们在B的`test()`方法中，因此超类是A，程序应该打印“A method”。

> This means that super calls are not resolved dynamically based on the runtime instance. The superclass used to look up the method is a static—practically lexical—property of where the call occurs. When we added inheritance to jlox, we took advantage of that static aspect by storing the superclass in the same Environment structure we used for all lexical scopes. Almost as if the interpreter saw the above program like this:

这意味着超类调用不是根据运行时的实例进行动态解析的。用于查找方法的超类是调用发生位置的一个静态（实际上是词法）属性。当我们在jlox中添加继承时，我们利用了这种静态优势，将超类存储在我们用于所有词法作用域的同一个Environment结构中。就好像解释器看到的程序是这样的：

```
class A {
  method() {
    print "A method";
  }
}

var Bs_super = A;
class B < A {
  method() {
    print "B method";
  }

  test() {
    runtimeSuperCall(Bs_super, "method");
  }
}

var Cs_super = B;
class C < B {}

C().test();
```

> Each subclass has a hidden variable storing a reference to its superclass. Whenever we need to perform a super call, we access the superclass from that variable and tell the runtime to start looking for methods there.

每个子类都有一个隐藏变量，用于存储对其超类的引用。当我们需要执行一个超类调用时，我们就从这个变量访问超类，并告诉运行时从那里开始查找方法。

> We'll take the same path with clox. The difference is that instead of jlox's heap-allocated Environment class, we have the bytecode VM's value stack and upvalue system. The machinery is a little different, but the overall effect is the same.

我们在clox中采用相同的方法。不同之处在于，我们使用的是字节码虚拟机的值栈和上值系统，而不是jlox的堆分配的Environment 类。机制有些不同，但总体效果是一样的。

> ### 29.2.1 A superclass local variable

29.2.1 超类局部变量

> Our compiler already emits code to load the superclass onto the stack. Instead of leaving that slot as a temporary, we create a new scope and make it a local variable.

我们的编译器已经发出了将超类加载到栈中的代码。我们不将这个槽看作是临时的，而是创建一个新的作用域，并将其作为一个局部变量。

compiler.c，在classDeclaration()方法中添加代码：

```
    }
    // 新增部分开始
    beginScope();
    addLocal(syntheticToken("super"));
    defineVariable(0);
    // 新增部分结束
    namedVariable(className, false);
    emitByte(OP_INHERIT);
```

> Creating a new lexical scope ensures that if we declare two classes in the same scope, each has a different local slot to store its superclass. Since we always name this variable "super", if we didn't make a scope for each subclass, the variables would collide.

创建一个新的词法作用域可以确保，如果我们在同一个作用域中声明两个类，每个类都有一个不同的局部槽来存储其超类。由于我们总是将该变量命名为"super"，如果我们不为每个子类创建作用域，那么这些变量就会发生冲突。

> We name the variable "super" for the same reason we use "this" as the name of the hidden local variable that `this` expressions resolve to: "super" is a reserved word, which guarantees the compiler's hidden variable won't collide with a user-defined one.

我们将该变量命名为"super"，与我们使用"this"作为`this`表达式解析得到的隐藏局部变量名称的原因相同："super"是一个保留字，它可以保证编译器的隐藏变量不会与用户定义的变量发生冲突。

> The difference is that when compiling `this` expressions, we conveniently have a token sitting around whose lexeme is "this". We aren't so lucky here. Instead, we add a little helper function to create a synthetic token for the given constant string.

不同之处在于，在编译`this`表达式时，我们可以很方便地使用一个标识，词素是`this`。在这里我们就没那么幸运了。相对地，我们添加一个小的辅助函数，来为给定的常量字符串创建一个合成标识^6。

compiler.c，在variable()方法后添加代码：

```
static Token syntheticToken(const char* text) {
  Token token;
  token.start = text;
  token.length = (int)strlen(text);
  return token;
}
```

> Since we opened a local scope for the superclass variable, we need to close it.

因为我们为超类变量打开了一个局部作用域，我们还需要关闭它。

compiler.c，在classDeclaration()方法中添加代码：

```
  emitByte(OP_POP);
  // 新增部分开始
  if (classCompiler.hasSuperclass) {
    endScope();
  }
  // 新增部分结束
  currentClass = currentClass->enclosing;
```

> We pop the scope and discard the “super” variable after compiling the class body and its methods. That way, the variable is accessible in all of the methods of the subclass. It’s a somewhat pointless optimization, but we create the scope only if there *is* a superclass clause. Thus we need to close the scope only if there is one.

在编译完类的主体及其方法后，我们会弹出作用域并丢弃“super”变量。这样，该变量在子类的所有方法中被都可以访问。这是一个有点无意义的优化，但我们只在有超类子句的情况下创建作用域。因此，只有在有超类的情况下，我们才需要关闭这个作用域。

> To track that, we could declare a little local variable in `classDeclaration()`. But soon, other functions in the compiler will need to know whether the surrounding class is a subclass or not. So we may as well give our future selves a hand and store this fact as a field in the ClassCompiler now.

为了记录是否有超类，我们可以在`classDeclaration()`中声明一个局部变量。但是很快，编译器中的其它函数需要知道外层的类是否是子类。所以我们不妨帮帮未来的自己，现在就把它作为一个字段存储在ClassCompiler中。

compiler.c，在结构体ClassCompiler中添加代码：

```
typedef struct ClassCompiler {
  struct ClassCompiler* enclosing;
  // 新增部分开始
  bool hasSuperclass;
  // 新增部分结束
} ClassCompiler;
```

> When we first initialize a ClassCompiler, we assume it is not a subclass.

当我们第一次初始化某个ClassCompiler时，我们假定它不是子类。

compiler.c，在classDeclaration()方法中添加代码：

```
  ClassCompiler classCompiler;
  // 新增部分开始
  classCompiler.hasSuperclass = false;
  // 新增部分结束
  classCompiler.enclosing = currentClass;
```

> Then, if we see a superclass clause, we know we are compiling a subclass.

然后，如果看到超类子句，我们就知道正在编译一个子类。

compiler.c，在classDeclaration()方法中添加代码：

```
    emitByte(OP_INHERIT);
    // 新增部分开始
    classCompiler.hasSuperclass = true;
    // 新增部分结束
  }
```

> This machinery gives us a mechanism at runtime to access the superclass object of the surrounding subclass from within any of the subclass's methods—simply emit code to load the variable named "super". That variable is a local outside of the method body, but our existing upvalue support enables the VM to capture that local inside the body of the method or even in functions nested inside that method.

这种机制在运行时为我们提供了一种方法，可以从子类的任何方法中访问外层子类的超类对象——只需发出代码来加载名为"super"的变量。这个变量是方法主体之外的一个局部变量，但是我们现有的上值支持VM在方法主体内、甚至是嵌套方法内的函数中捕获该局部变量。

29.3 Super Calls

29.3 超类调用

> With that runtime support in place, we are ready to implement super calls. As usual, we go front to back, starting with the new syntax. A super call begins, naturally enough, with the `super` keyword.

有了这个运行时支持，我们就可以实现超类调用了。跟之前一样，我们从前端到后端，先从新语法开始。超类调用，自然是以`super`关键字开始^7。

compiler.c，替换1行：

```
  [TOKEN_RETURN]        = {NULL,     NULL,   PREC_NONE},
  // 替换部分开始
  [TOKEN_SUPER]         = {super_,   NULL,   PREC_NONE},
  // 替换部分结束
  [TOKEN_THIS]          = {this_,    NULL,   PREC_NONE},
```

> When the expression parser lands on a `super` token, control jumps to a new parsing function which starts off like so:

当表达式解析器落在一个`super`标识时，控制流会跳转到一个新的解析函数，该函数的开头是这样的：

compiler.c，在syntheticToken()方法后添加代码：

```
static void super_(bool canAssign) {
  consume(TOKEN_DOT, "Expect '.' after 'super'.");
  consume(TOKEN_IDENTIFIER, "Expect superclass method name.");
  uint8_t name = identifierConstant(&parser.previous);
}
```

> This is pretty different from how we compiled `this` expressions. Unlike `this`, a `super` token is not a standalone expression. Instead, the dot and method name following it are inseparable parts of the

> syntax. However, the parenthesized argument list is separate. As with normal method access, Lox supports getting a reference to a superclass method as a closure without invoking it:

这与我们编译this表达式的方式很不一样。与this不同，super标识不是一个独立的表达式^8。相反，它后面的点和方法名称是语法中不可分割的部分。但是，括号内的参数列表是独立的。和普通的方法访问一样，Lox支持以闭包的方式获得对超类方法的引用，而不必调用它：

```
class A {
  method() {
    print "A";
  }
}

class B < A {
  method() {
    var closure = super.method;
    closure(); // Prints "A".
  }
}
```

> In other words, Lox doesn't really have super *call* expressions, it has super *access* expressions, which you can choose to immediately invoke if you want. So when the compiler hits a super token, we consume the subsequent . token and then look for a method name. Methods are looked up dynamically, so we use identifierConstant() to take the lexeme of the method name token and store it in the constant table just like we do for property access expressions.

换句话说，Lox并没有真正的超类*调用（call）*表达式，它有的是超类*访问（access）*表达式，如果你愿意，可以选择立即调用。因此，当编译器碰到一个super标识时，我们会消费后续的.标识，然后寻找一个方法名称。方法是动态查找的，所以我们使用identifierConstant()来获取方法名标识的词素，并将其存储在常量表中，就像我们对属性访问表达式所做的那样。

> Here is what the compiler does after consuming those tokens:

下面是编译器在消费这些标识之后做的事情：

compiler.c，在super_()方法中添加代码：

```
  uint8_t name = identifierConstant(&parser.previous);
  // 新增部分开始
  namedVariable(syntheticToken("this"), false);
  namedVariable(syntheticToken("super"), false);
  emitBytes(OP_GET_SUPER, name);
  // 新增部分结束
}
```

> In order to access a *superclass method* on *the current instance*, the runtime needs both the receiver *and* the superclass of the surrounding method's class. The first namedVariable() call generates code to look up the current receiver stored in the hidden variable "this" and push it onto the stack. The second

> namedVariable() call emits code to look up the superclass from its “super” variable and push that on top.

为了在*当前实例*上访问一个*超类方法*，运行时需要接收器*和*外围方法所在类的超类。第一个namedVariable()调用产生代码来查找存储在隐藏变量“this”中的当前接收器，并将其压入栈中。第二个namedVariable()调用产生代码，从它的“super”变量中查找超类，并将其推入栈顶。

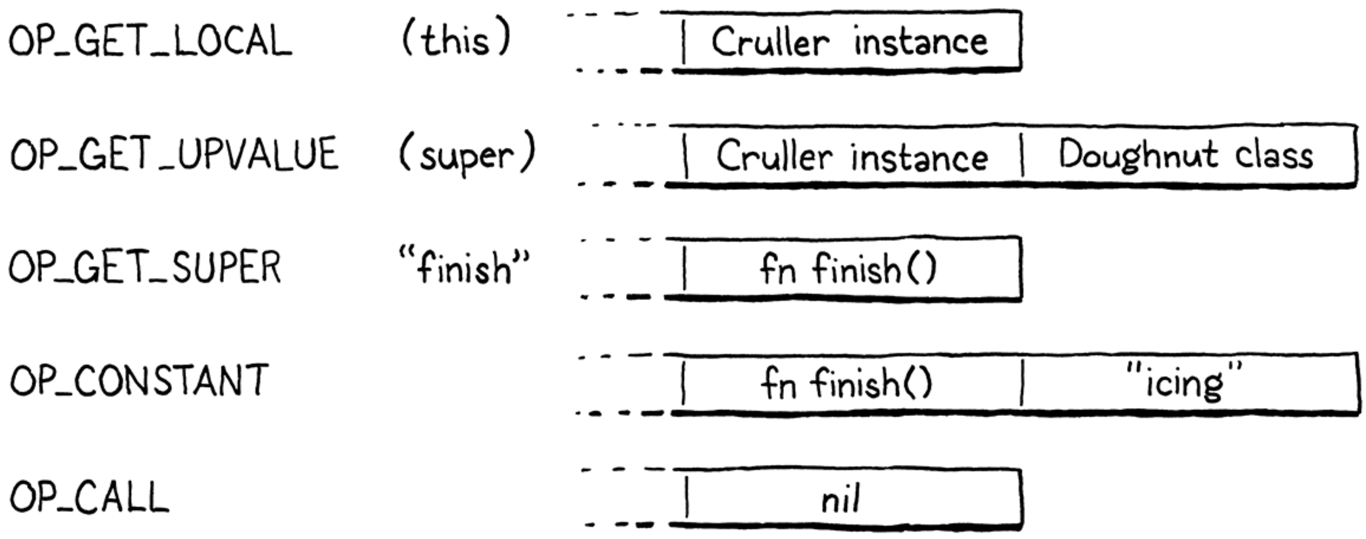
> Finally, we emit a new OP_GET_SUPER instruction with an operand for the constant table index of the method name. That’s a lot to hold in your head. To make it tangible, consider this example program:

最后，我们发出一条新的OP_GET_SUPER指令，其操作数为方法名称的常量表索引。你脑子里装的东西太多了。为了使它具体化，请看下面的示例程序：

```
class Doughnut {
  cook() {
    print "Dunk in the fryer.";
    this.finish("sprinkles");
  }

  finish(ingredient) {
    print "Finish with " + ingredient;
  }
}

class Cruller < Doughnut {
  finish(ingredient) {
    // No sprinkles, always icing.
    super.finish("icing");
  }
}
```

> The bytecode emitted for the super.finish("icing") expression looks and works like this:

super.finish("icing")发出的字节码看起来像是这样的：



> The first three instructions give the runtime access to the three pieces of information it needs to perform the super access:

前三条指令让运行时获得了执行超类访问时需要的三条信息：

> 1. The first instruction loads **the instance** onto the stack.
> 2. The second instruction loads **the superclass where the method is resolved**.
> 3. Then the new `OP_GET_SUPER` instuction encodes **the name of the method to access** as an operand.

1. 第一条指令将**实例**加载到栈中。
2. 第二条指令加载了**将用于解析方法的超类。**
3. 然后，新的`OP_GET_SUPER`指令将**要访问的方法名称**编码为操作数。

> The remaining instructions are the normal bytecode for evaluating an argument list and calling a function.

剩下的指令是用于计算参数列表和调用函数的常规字节码。

> We’re almost ready to implement the new `OP_GET_SUPER` instruction in the interpreter. But before we do, the compiler has some errors it is responsible for reporting.

我们几乎已经准备好在解释器中实现新的`OP_GET_SUPER`指令了。但在此之前，编译器需要负责报告一些错误。

compiler.c，在super_()方法中添加代码：

```
static void super_(bool canAssign) {
  // 新增部分开始
  if (currentClass == NULL) {
    error("Can't use 'super' outside of a class.");
  } else if (!currentClass->hasSuperclass) {
    error("Can't use 'super' in a class with no superclass.");
  }
  // 新增部分结束
  consume(TOKEN_DOT, "Expect '.' after 'super'.");
```

> A super call is meaningful only inside the body of a method (or in a function nested inside a method), and only inside the method of a class that has a superclass. We detect both of these cases using the value of `currentClass`. If that’s `NULL` or points to a class with no superclass, we report those errors.

超类调用只有在方法主体（或方法中嵌套的函数）中才有意义，而且只在具有超类的某个类的方法中才有意义。我们使用`currentClass`的值来检测这两种情况。如果它是`NULL`或者指向一个没有超类的类，我们就报告这些错误。

> ### 29.3.1 Executing super accesses

29.3.1 执行超类访问

> Assuming the user didn't put a `super` expression where it's not allowed, their code passes from the compiler over to the runtime. We've got ourselves a new instruction.

假设用户没有在不允许的地方使用`super`表达式，他们的代码将从编译器传递到运行时。我们已经有了一个新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_SET_PROPERTY,
  // 新增部分开始
  OP_GET_SUPER,
  // 新增部分结束
  OP_EQUAL,
```

> We disassemble it like other opcodes that take a constant table index operand.

我们像对其它需要常量表索引操作数的操作码一样对它进行反汇编。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return constantInstruction("OP_SET_PROPERTY", chunk, offset);
    // 新增部分开始
    case OP_GET_SUPER:
      return constantInstruction("OP_GET_SUPER", chunk, offset);
    // 新增部分结束
    case OP_EQUAL:
```

> You might anticipate something harder, but interpreting the new instruction is similar to executing a normal property access.

你可能预想这是一件比较困难的事，但解释新指令与执行正常的属性访问类似。

vm.c，在run()方法中添加代码：

```
      }
      // 新增部分开始
      case OP_GET_SUPER: {
        ObjString* name = READ_STRING();
        ObjClass* superclass = AS_CLASS(pop());

        if (!bindMethod(superclass, name)) {
          return INTERPRET_RUNTIME_ERROR;
        }
        break;
      }
      // 新增部分结束
      case OP_EQUAL: {
```

> As with properties, we read the method name from the constant table. Then we pass that to `bindMethod()` which looks up the method in the given class's method table and creates an ObjBoundMethod to bundle the resulting closure to the current instance.

和属性一样，我们从常量表中读取方法名。然后我们将其传递给`bindMethod()`，该方法会在给定类的方法表中查找方法，并创建一个ObjBoundMethod将结果闭包与当前实例相绑定。

> The key difference is *which* class we pass to `bindMethod()`. With a normal property access, we use the ObjInstances's own class, which gives us the dynamic dispatch we want. For a super call, we don't use the instance's class. Instead, we use the statically resolved superclass of the containing class, which the compiler has conveniently ensured is sitting on top of the stack waiting for us.

关键的区别在于将*哪个*类传递给`bindMethod()`。对于普通的属性访问，我们使用ObjInstances自己的类，这为我们提供了我们想要的动态分派。对于超类调用，我们不使用实例的类。相反，我们使用静态分析得到的外层类的超类，编译器已经确保它在栈顶等着我们^9。

> We pop that superclass and pass it to `bindMethod()`, which correctly skips over any overriding methods in any of the subclasses between that superclass and the instance's own class. It also correctly includes any methods inherited by the superclass from any of *its* superclasses.

我们弹出该超类并将其传递给`bindMethod()`，该方法会正确地跳过该超类与实例本身的类之间的任何子类覆写的方法。它还正确地包含了超类从其任何超类中继承的方法。

> The rest of the behavior is the same. Popping the superclass leaves the instance at the top of the stack. When `bindMethod()` succeeds, it pops the instance and pushes the new bound method. Otherwise, it reports a runtime error and returns `false`. In that case, we abort the interpreter.

其余的行为都是一样的。超类弹出栈使得实例位于栈顶。当`bindMethod()`成功时，它会弹出实例并压入新的已绑定方法。否则，它会报告一个运行时错误并返回`false`。在这种情况下，我们中止解释器。

> ### 29.3.2 Faster super calls

29.3.2 更快的超类调用

> We have superclass method accesses working now. And since the returned object is an ObjBoundMethod that you can then invoke, we've got super *calls* working too. Just like last chapter, we've reached a point where our VM has the complete, correct semantics.

我们现在有了对超类方法的访问。由于返回的对象是一个你可以稍后调用的ObjBoundMethod，我们也就有了可用的超类*调用*。就像上一章一样，我们的虚拟机现在已经有了完整、正确的语义。

> But, also like last chapter, it's pretty slow. Again, we're heap allocating an ObjBoundMethod for each super call even though most of the time the very next instruction is an `OP_CALL` that immediately unpacks that bound method, invokes it, and then discards it. In fact, this is even more likely to be true for super calls than for regular method calls. At least with method calls there is a chance that the user is actually invoking a function stored in a field. With super calls, you're *always* looking up a method. The only question is whether you invoke it immediately or not.

但是，也和上一章一样，它很慢。同样，我们为每个超类调用在堆中分配了一个ObjBoundMethod，尽管大多数时候下一个指令就是`OP_CALL`，它会立即解包该已绑定方法，调用它，然后丢弃它。事实上，超类调用比普

通方法调用更有可能出现这种情况。至少在方法调用中，用户有可能实际上在调用存储在字段中的函数。在超类调用中，你肯定是在查找一个方法。唯一的问题在于你是否立即调用它。

> The compiler can certainly answer that question for itself if it sees a left parenthesis after the superclass method name, so we'll go ahead and perform the same optimization we did for method calls. Take out the two lines of code that load the superclass and emit `OP_GET_SUPER`, and replace them with this:

如果编译器看到超类方法名称后面有一个左括号，它肯定能自己回答这个问题，所以我们会继续执行与方法调用相同的优化。去掉加载超类并发出`OP_GET_SUPER`的两行代码，替换为这个：

compiler.c，在super_()方法中替换2行：

```
  namedVariable(syntheticToken("this"), false);
  // 替换部分开始
  if (match(TOKEN_LEFT_PAREN)) {
    uint8_t argCount = argumentList();
    namedVariable(syntheticToken("super"), false);
    emitBytes(OP_SUPER_INVOKE, name);
    emitByte(argCount);
  } else {
    namedVariable(syntheticToken("super"), false);
    emitBytes(OP_GET_SUPER, name);
  }
  // 替换部分结束
}
```

> Now before we emit anything, we look for a parenthesized argument list. If we find one, we compile that. Then we load the superclass. After that, we emit a new `OP_SUPER_INVOKE` instruction. This superinstruction combines the behavior of `OP_GET_SUPER` and `OP_CALL`, so it takes two operands: the constant table index of the method name to look up and the number of arguments to pass to it.

现在，在我们发出任何代码之前，我们要寻找一个带括号的参数列表。如果找到了，我们就编译它，任何加载超类，之后，我们发出一条新的`OP_SUPER_INVOKE`指令。这个超级指令结合了`OP_GET_SUPER`和`OP_CALL`的行为，所以它需要两个操作数：待查找的方法名称和要传递给它的参数数量。

> Otherwise, if we don't find a `(`, we continue to compile the expression as a super access like we did before and emit an `OP_GET_SUPER`.

否则，如果没有找到`(`，则继续像前面那样将表达式编译为一个超类访问，并发出一条`OP_GET_SUPER`指令。

> Drifting down the compilation pipeline, our first stop is a new instruction.

沿着编译流水线向下，我们的第一站是一条新指令。

chunk.h，在枚举OpCode中添加代码：

```
  OP_INVOKE,
  // 新增部分开始
  OP_SUPER_INVOKE,
  // 新增部分结束
  OP_CLOSURE,
```

> And just past that, its disassembler support.

在那之后，是它的反汇编器支持。

debug.c，在disassembleInstruction()方法中添加代码：

```
      return invokeInstruction("OP_INVOKE", chunk, offset);
    // 新增部分开始
    case OP_SUPER_INVOKE:
      return invokeInstruction("OP_SUPER_INVOKE", chunk, offset);
    // 新增部分结束
    case OP_CLOSURE: {
```

> A super invocation instruction has the same set of operands as OP_INVOKE, so we reuse the same helper to disassemble it. Finally, the pipeline dumps us into the interpreter.

超类调用指令具有与OP_INVOKE相同的操作数集，因此我们复用同一个辅助函数对其反汇编。最后，流水线将我们带到解释器中。

vm.c，在run()方法中添加代码：

```
        break;
      }
      // 新增部分开始
      case OP_SUPER_INVOKE: {
        ObjString* method = READ_STRING();
        int argCount = READ_BYTE();
        ObjClass* superclass = AS_CLASS(pop());
        if (!invokeFromClass(superclass, method, argCount)) {
          return INTERPRET_RUNTIME_ERROR;
        }
        frame = &vm.frames[vm.frameCount - 1];
        break;
      }
      // 新增部分结束
      case OP_CLOSURE: {
```

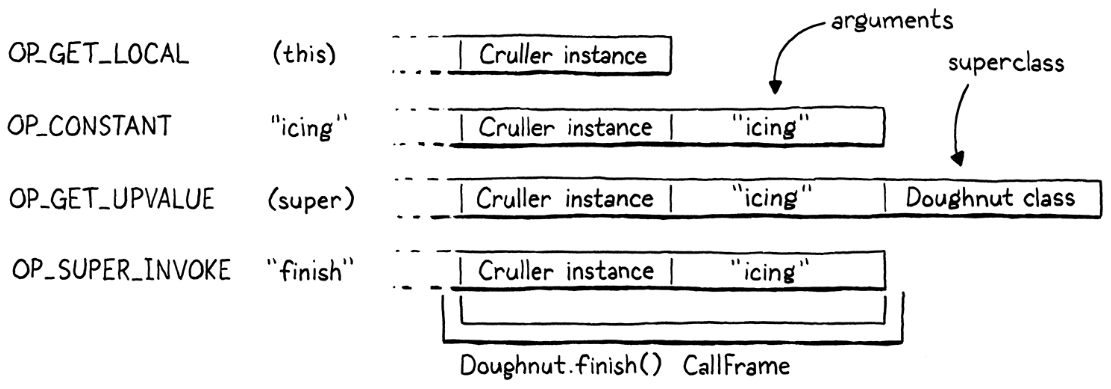
> This handful of code is basically our implementation of OP_INVOKE mixed together with a dash of OP_GET_SUPER. There are some differences in how the stack is organized, though. With an unoptimized super call, the superclass is popped and replaced by the ObjBoundMethod for the resolved function *before* the arguments to the call are executed. This ensures that by the time the OP_CALL is executed, the bound method is *under* the argument list, where the runtime expects it to be for a closure call.

这一小段代码基本上是OP_INVOKE的实现，其中混杂了一点OP_GET_SUPER。不过，在堆栈的组织方式上有些不同。在未优化的超类调用中，超类会被弹出，并在调用的*参数*被执行之前替换为被解析函数的

ObjBoundMethod。这确保了在`OP_CALL`执行时，已绑定方法在参数列表之下，也就是运行时期望闭包调用所在的位置。

> With our optimized instructions, things are shuffled a bit:

在我们优化的指令中，事情有点被打乱：



> Now resolving the superclass method is part of the *invocation*, so the arguments need to already be on the stack at the point that we look up the method. This means the superclass object is on top of the arguments.

现在，解析超类方法是执行的一部分，因此当我们查找方法时，参数需要已经在栈上。这意味着超类对象位于参数之上。

> Aside from that, the behavior is roughly the same as an `OP_GET_SUPER` followed by an `OP_CALL`. First, we pull out the method name and argument count operands. Then we pop the superclass off the top of the stack so that we can look up the method in its method table. This conveniently leaves the stack set up just right for a method call.

除此之外，其行为与`OP_GET_SUPER`后跟`OP_CALL`大致相同。首先，我们取出方法名和参数数量两个操作数。然后我们从栈顶弹出超类，这样我们就可以在它的方法表中查找方法。这方便地将堆栈设置为适合方法调用的状态。

> We pass the superclass, method name, and argument count to our existing `invokeFromClass()` function. That function looks up the given method on the given class and attempts to create a call to it with the given arity. If a method could not be found, it returns `false`, and we bail out of the interpreter. Otherwise, `invokeFromClass()` pushes a new CallFrame onto the call stack for the method's closure. That invalidates the interpreter's cached CallFrame pointer, so we refresh `frame`.

我们将超类、方法名和参数数量传递给现有的`invokeFromClass()`函数。该函数在给定的类上查找给定的方法，并尝试用给定的元数创建一个对它的调用。如果找不到某个方法，它就返回false，并退出解释器。否则，`invokeFromClass()`将一个新的CallFrame压入方法闭包的调用栈上。这会使解释器缓存的CallFrame指针失效，所以我们也要刷新`frame`。

> ## 29.4 A Complete Virtual Machine

29.4 一个完整的虚拟机

> Take a look back at what we've created. By my count, we wrote around 2,500 lines of fairly clean, straightforward C. That little program contains a complete implementation of the—quite high-level!—

> Lox language, with a whole precedence table full of expression types and a suite of control flow statements. We implemented variables, functions, closures, classes, fields, methods, and inheritance.

回顾一下我们创造了什么。根据我的计算，我们编写了大约2500行相当干净、简洁的C语言代码。这个小程序中包含了对Lox语言（相当高级）的完整实现，它有一个满是表达式类型的优先级表和一套控制流语句。我们实现了变量、函数、闭包、类、字段、方法和继承。

> Even more impressive, our implementation is portable to any platform with a C compiler, and is fast enough for real-world production use. We have a single-pass bytecode compiler, a tight virtual machine interpreter for our internal instruction set, compact object representations, a stack for storing variables without heap allocation, and a precise garbage collector.

更令人印象深刻的是，我们的实现可以移植到任何带有C编译器的平台上，而且速度快到足以在实际生产中使用。我们有一个单遍字节码编译器，一个用于内部指令集的严格虚拟机解释器，紧凑的对象表示，一个用于存储变量而不需要堆分配的栈，以及一个精确的垃圾回收器。

> If you go out and start poking around in the implementations of Lua, Python, or Ruby, you will be surprised by how much of it now looks familiar to you. You have seriously leveled up your knowledge of how programming languages work, which in turn gives you a deeper understanding of programming itself. It's like you used to be a race car driver, and now you can pop the hood and repair the engine too.

如果你开始研究Lua、Python或Ruby的实现，你会惊讶于它们现在看起来有多熟悉。你已经真正提高了关于编程语言工作方式的知识水平，这反过来又使你对编程本身有了更深的理解。这就像你以前是个赛车手，现在你可以打开引擎盖，修改发动机了。

> You can stop here if you like. The two implementations of Lox you have are complete and full featured. You built the car and can drive it wherever you want now. But if you are looking to have more fun tuning and tweaking for even greater performance out on the track, there is one more chapter. We don't add any new capabilities, but we roll in a couple of classic optimizations to squeeze even more perf out. If that sounds fun, keep reading . . .

如果你愿意，可以在这里停下来。你拥有的两个Lox实现是完整的、功能齐全的。你造了这俩车，现在可以把它开到你想去的地方。但是，如果你想获得更多改装与调整的乐趣，以期在赛道上获得更佳的性能，还有一个章节。我们没有增加任何新的功能，但我们推出了几个经典的优化，以挤压出更多的性能。如果这听起来很有趣，请继续读下去……

习题

1. > A tenet of object-oriented programming is that a class should ensure new objects are in a valid state. In Lox, that means defining an initializer that populates the instance's fields. Inheritance complicates invariants because the instance must be in a valid state according to all of the classes in the object's inheritance chain.
   >
   > The easy part is remembering to call `super.init()` in each subclass's `init()` method. The harder part is fields. There is nothing preventing two classes in the inheritance chain from accidentally claiming the same field name. When this happens, they will step on each other's fields and possibly leave you with an instance in a broken state.

> If Lox was your language, how would you address this, if at all? If you would change the language, implement your change.

面向对象编程的一个原则是，类应该确保新对象处于有效状态。在Lox中，这意味着要定义一个填充实例字段的初始化器。继承使不变性复杂化，因为对于对象继承链中的所有类，实例必须处于有效状态。

简单的部分是记住在每个子类的`init()`方法中调用`super.init()`。比较难的部分是字段。没有什么方法可以防止继承链中的两个类意外地声明相同的字段名。当这种情况发生时，它们会互相干扰彼此的字段，并可能让你的实例处于崩溃状态。

如果Lox是你的语言，你会如何解决这个问题？如果你想改变语言，请实现你的更改。

2. > Our copy-down inheritance optimization is valid only because Lox does not permit you to modify a class's methods after its declaration. This means we don't have to worry about the copied methods in the subclass getting out of sync with later changes to the superclass.
   >
   > Other languages, like Ruby, *do* allow classes to be modified after the fact. How do implementations of languages like that support class modification while keeping method resolution efficient?

   我们的向下复制继承优化之所以有效，仅仅是因为Lox不允许在类声明之后修改它的方法。这意味着我们不必担心子类中复制的方法与后面对超类的修改不同步。

   其它语言，如Ruby，确实允许在事后修改类。像这样的语言实现如何支持类的修改，同时保持方法解析的效率呢？

3. > In the jlox chapter on inheritance, we had a challenge to implement the BETA language's approach to method overriding. Solve the challenge again, but this time in clox. Here's the description of the previous challenge:

   在jlox关于继承的章节中，我们有一个习题，是实现BETA语言的方法重写。再次解决这个习题，但这次是在clox中。下面是对之前习题的描述：

   > In Lox, as in most other object-oriented languages, when looking up a method, we start at the bottom of the class hierarchy and work our way up—a subclass's method is preferred over a superclass's. In order to get to the superclass method from within an overriding method, you use `super`.

   在Lox中，和其它大多数面向对象的语言一样，当查找一个方法时，我们从类层次结构的底部开始，然后向上查找——子类的方法优于超类的方法。要想在子类方法中访问超类方法，可以使用`super`。

   > The language BETA takes the opposite approach. When you call a method, it starts at the *top* of the class hierarchy and works *down*. A superclass method wins over a subclass method. In order to get to the subclass method, the superclass method can call `inner`, which is sort of like the inverse of `super`. It chains to the next method down the hierarchy.

   BETA语言则采取了相反的方法。当你调用某个方法时，它从类层次结构的顶部开始向下运行。超类方法优于子类方法。要想访问子类方法，超类方法中可以调用`inner()`，这有点像是`super`的反义词。它会链接到层次结构中的下一个方法。

> The superclass method controls when and where the subclass is allowed to refine its behavior. If the superclass method doesn't call `inner` at all, then the subclass has no way of overriding or modifying the superclass's behavior.

超类方法控制着子类何时何地被允许完善其行为。如果超类方法根本不调用`inner`，那么子类就没有办法覆写或修改超类的行为。

> Take out Lox's current overriding and `super` behavior, and replace it with BETA's semantics. In short:

去掉Lox中当前的覆写和`super`行为，替换为BETA的语义。简而言之：

> - When calling a method on a class, the method *highest* on the class's inheritance chain takes precedence.
> - Inside the body of a method, a call to `inner` looks for a method with the same name in the nearest subclass along the inheritance chain between the class containing the `inner` and the class of `this`. If there is no matching method, the `inner` call does nothing.

- 当调用某个类中的方法时，该类继承链上最高的方法优先。
- 在方法体内部，对`inner`的调用，会沿着包含`inner`的类和`this`的类之间的继承链，在最近的子类中查找同名的方法。如果没有匹配的方法，`inner`调用就什么也不做。

> For example:

举例来说：

```
class Doughnut {
  cook() {
    print "Fry until golden brown.";
    inner();
    print "Place in a nice box.";
  }
}

class BostonCream < Doughnut {
  cook() {
    print "Pipe full of custard and coat with chocolate.";
  }
}

BostonCream().cook();
```

> This should print:

这里应该打印：

```
Fry until golden brown.
Pipe full of custard and coat with chocolate.
Place in a nice box.
```

> Since clox is about not just implementing Lox, but doing so with good performance, this time around try to solve the challenge with an eye towards efficiency.

因为clox不仅仅是实现Lox，而是要以良好的性能来实现，所以这次要尝试以效率为导向来解决这个问题。

30.优化 Optimization

> The evening's the best part of the day. You've done your day's work. Now you can put your feet up and enjoy it.
>
> ——Kazuo Ishiguro, *The Remains of the Day*

夜晚是一天中最美好的时光。你已经完成了一天的工作，现在你可以双腿搁平，享受一下。（石黑一雄，《长日将尽》）

> If I still lived in New Orleans, I'd call this chapter a *lagniappe*, a little something extra given for free to a customer. You've got a whole book and a complete virtual machine already, but I want you to have some more fun hacking on clox. This time, we're going for pure performance. We'll apply two very different optimizations to our virtual machine. In the process, you'll get a feel for measuring and improving the performance of a language implementation—or any program, really.

如果我还住在新奥尔良，我会把这一章称为*lagniappe*（小赠品），即免费送给顾客的一点额外的东西。你已经有了一整本书和一个完整的虚拟机，但我希望你能在clox上获得更多的乐趣。这一次，我们要追求的是纯粹的性能。我们将对虚拟机应用两种截然不同的优化。在这个过程中，你将了解如何测量和提高语言实现的性能——或者说任何程序的性能，真的。

> ## 30.1 Measuring Performance

30.1 测量性能

> **Optimization** means taking a working application and improving its performance. An optimized program does the same thing, it just takes less resources to do so. The resource we usually think of when optimizing is runtime speed, but it can also be important to reduce memory usage, startup time, persistent storage size, or network bandwidth. All physical resources have some cost—even if the cost is mostly in wasted human time—so optimization work often pays off.

优化是指拿到一个基本可用的应用程序并提高其性能。一个优化后的程序能做到同样的事情，只是需要更少的资源。我们在优化时通常考虑的资源是运行时速度，但减少内存使用、启动时间、持久化存储大小或网络带宽也很重要。所有的物理资源都有一定的成本——即使成本主要是浪费人力时间，所以优化工作通常都能得到回报。

> There was a time in the early days of computing that a skilled programmer could hold the entire hardware architecture and compiler pipeline in their head and understand a program's performance just by thinking real hard. Those days are long gone, separated from the present by microcode, cache lines, branch prediction, deep compiler pipelines, and mammoth instruction sets. We like to pretend C is a "low-level" language, but the stack of technology between
>
> ```
> printf("Hello, world!");
> ```

> and a greeting appearing on screen is now perilously tall.

在计算机早期，曾经有一段时间，一个熟练的程序员可以把整个硬件架构和编译器管道记在脑子里，只需要认真思考就可以了解程序的性能。那些日子早已一去不复返了，现在已经被微码、缓存线、分支预测、深层编译器管道和庞大的指令集所分隔。我们喜欢假装C语言是一种“低级”语言，但在`printf("Hello, world!");`和屏幕上出现的问候语之间的技术栈现在已经很高了。

> Optimization today is an empirical science. Our program is a border collie sprinting through the hardware’s obstacle course. If we want her to reach the end faster, we can’t just sit and ruminate on canine physiology until enlightenment strikes. Instead, we need to *observe* her performance, see where she stumbles, and then find faster paths for her to take.

今天的优化是一门经验科学。我们的程序是一只在硬件障碍赛中冲刺的边牧。如果我们想让她更快地到达终点，我们不能只是坐在那里思考犬类的生理机能，等着灵光乍现。相反，我们需要*观察*她的表现，看看她在那里出错，然后为她找到更快的路径。

> Much like agility training is particular to one dog and one obstacle course, we can’t assume that our virtual machine optimizations will make *all* Lox programs run faster on *all* hardware. Different Lox programs stress different areas of the VM, and different architectures have their own strengths and weaknesses.

就像敏捷训练要真的一只狗和一项障碍赛，我们不能假设我们的虚拟机优化会使所有的Lox程序在所有硬件上运行得更快。不同的Lox程序侧重虚拟机的不同领域，不同的架构也有其自身的优势和劣势。

30.1.1 Benchmarks

> When we add new functionality, we validate correctness by writing tests—Lox programs that use a feature and validate the VM’s behavior. Tests pin down semantics and ensure we don’t break existing features when we add new ones. We have similar needs when it comes to performance:

当我们添加新功能时，我们通过编写测试来验证正确性——使用某个特性并验证虚拟机行为的Lox程序。测试可以约束语义，并确保在添加新功能时，不会破坏现有的特性。在性能方面，我们也有类似的需求：

> 1. How do we validate that an optimization *does* improve performance, and by how much?
> 2. How do we ensure that other unrelated changes don’t *regress* performance?

1. 我们如何验证一项优化确实提高了性能，以及提高了多少？
2. 我们如何确保其它不相关的修改不会使性能退步？

> The Lox programs we write to accomplish those goals are **benchmarks**. These are carefully crafted programs that stress some part of the language implementation. They measure not *what* the program does, but how *long* it takes to do it.

我们为实现这些目标而编写的Lox程序就是**基准**。这些都是精心设计的程序，侧重于语言实现的某些部分。它们测量的不是程序*做了什么*，而是做完这些需要*多久*^1。

> By measuring the performance of a benchmark before and after a change, you can see what your change does. When you land an optimization, all of the tests should behave exactly the same as they did before, but hopefully the benchmarks run faster.

通过测量修改前后的基准性能，你可以看到修改的效果。当你完成优化时，所有测试都应该与之前的行为完全一样，只是希望基准程序运行更快一点。

> Once you have an entire *suite* of benchmarks, you can measure not just *that* an optimization changes performance, but on which *kinds* of code. Often you'll find that some benchmarks get faster while others get slower. Then you have to make hard decisions about what kinds of code your language implementation optimizes for.

一旦你有了一整套的基准测试，你不仅可以衡量某个优化是否改变了性能，而且可以衡量改变了哪类代码的性能。通常，你会发现一些基准测试变得更快，而另一些则变得更慢。然后你必须作出艰难的决策：你的语言实现要对哪种代码进行优化。

> The suite of benchmarks you choose to write is a key part of that decision. In the same way that your tests encode your choices around what correct behavior looks like, your benchmarks are the embodiment of your priorities when it comes to performance. They will guide which optimizations you implement, so choose your benchmarks carefully, and don't forget to periodically reflect on whether they are helping you reach your larger goals.

你选择编写的基准套件是该决策的一部分。就像你的测试代码编码了关于正确代码行为的选择，你的基准测试是你在性能方面侧重点的体现。它们将指导你实现哪些优化，所以要仔细选择你的基准测试，并且不要忘记定期反思它们是否能帮助你实现更大的目标^2。

> Benchmarking is a subtle art. Like tests, you need to balance not overfitting to your implementation while ensuring that the benchmark does actually tickle the code paths that you care about. When you measure performance, you need to compensate for variance caused by CPU throttling, caching, and other weird hardware and operating system quirks. I won't give you a whole sermon here, but treat benchmarking as its own skill that improves with practice.

基准测试是一门微妙的艺术。就像测试一样，你需要在不过度拟合语言实现的同时，确保基准测试确实适合你所关心的代码路径。在测量性能时，你需要补偿由CPU节流、缓存和其它奇怪的硬件和操作系统特性造成的差异。我不会在这里给你一个完整的说教，但请把基准测试当作可以通过实践来提高的一门技能。

> ### 30.1.2 Profiling

30.1.2 剖析

> OK, so you've got a few benchmarks now. You want to make them go faster. Now what? First of all, let's assume you've done all the obvious, easy work. You are using the right algorithms and data structures —or, at least, you aren't using ones that are aggressively wrong. I don't consider using a hash table instead of a linear search through a huge unsorted array "optimization" so much as "good software engineering".

好的，现在你已经有了一些基准测试。你想让它们走得更快，现在怎么办呢？首先，我们假设你已经完成了所有明显的、简单的工作。你使用了正确的算法和数据结构——或者，至少你没有使用那些严重错误的算法和数据结构。我认为使用哈希表代替巨大的无序数组进行线性搜索不是"优化"，而是"良好的软件工程实现"。

> Since the hardware is too complex to reason about our program's performance from first principles, we have to go out into the field. That means *profiling*. A **profiler**, if you've never used one, is a tool that runs your program and tracks hardware resource use as the code executes. Simple ones show you how

> much time was spent in each function in your program. Sophisticated ones log data cache misses, instruction cache misses, branch mispredictions, memory allocations, and all sorts of other metrics.

由于硬件太过复杂，无法从基本原理推断出程序的性能，所以我们必须深入实地。这意味着*剖析*。**剖析器**（如果你从未使用过）是一种工具，可以运行你的程序并在代码执行过程中跟踪硬件资源的使用情况^3。简单的剖析器可以向你展示程序中每个函数花费了多少时间。复杂的剖析器则会记录数据缓存缺失、指令缓存缺失、分支预测错误、内存分配和其它各种指标。

> There are many profilers out there for various operating systems and languages. On whatever platform you program, it's worth getting familiar with a decent profiler. You don't need to be a master. I have learned things within minutes of throwing a program at a profiler that would have taken me *days* to discover on my own through trial and error. Profilers are wonderful, magical tools.

现在有很多针对不同操作系统和语言的剖析器。无论你在什么平台上编程，熟悉一个像样的剖析器都是值得的。你不需要称为大师。我在把程序扔给剖析器的几分钟内就学到了很多东西，而这些东西是我自己通过反复试验*好几天*才发现的。剖析器是一种绝妙、神奇的工具。

30.2 Faster Hash Table Probing

30.2 更快的哈希表探测

> Enough pontificating, let's get some performance charts going up and to the right. The first optimization we'll do, it turns out, is about the *tiniest* possible change we could make to our VM.

废话说得够多了，我们来让性能图表趋向右上方（提升性能）。我们要做的第一个优化，事实证明也是我们可以对虚拟机所做出的最微小的改变。

> When I first got the bytecode virtual machine that clox is descended from working, I did what any self-respecting VM hacker would do. I cobbled together a couple of benchmarks, fired up a profiler, and ran those scripts through my interpreter. In a dynamically typed language like Lox, a large fraction of user code is field accesses and method calls, so one of my benchmarks looked something like this:

当我第一次让clox派生的字节码虚拟机工作时，我做了任何有自尊心的虚拟机黑客都会做的事情。我拼凑了几个基准测试，启动了一个剖析器，并通过我的解释器运行了这些脚本。在Lox这样的动态类型语言中，用户代码的很大一部分是字段访问和方法调用，所以我的其中一个基准测试看起来是这样的^4：

```
class Zoo {
  init() {
    this.aardvark = 1;
    this.baboon   = 1;
    this.cat      = 1;
    this.donkey   = 1;
    this.elephant = 1;
    this.fox      = 1;
  }
  ant()    { return this.aardvark; }
  banana() { return this.baboon; }
  tuna()   { return this.cat; }
  hay()    { return this.donkey; }
  grass()  { return this.elephant; }
  mouse()  { return this.fox; }
}

var zoo = Zoo();
var sum = 0;
var start = clock();
while (sum < 100000000) {
  sum = sum + zoo.ant()
            + zoo.banana()
            + zoo.tuna()
            + zoo.hay()
            + zoo.grass()
            + zoo.mouse();
}

print clock() - start;
print sum;
```

> If you've never seen a benchmark before, this might seem ludicrous. *What* is going on here? The program itself doesn't intend to do anything useful. What it does do is call a bunch of methods and access a bunch of fields since those are the parts of the language we're interested in. Fields and methods live in hash tables, so it takes care to populate at least a *few* interesting keys in those tables. That is all wrapped in a big loop to ensure our profiler has enough execution time to dig in and see where the cycles are going.

如果你以前从未见过基准测试，那这个看起来可能会很好笑。这是怎么回事？这个程序本身并不打算做任何有用的事情。它所做的就是调用一堆方法和访问一堆字段，因为这些是语言中我们感兴趣的部分。字段和方法都在哈希表中，因此需要小心地在这些表中至少填入几个有趣的键^5。这一切都包装在一个大循环中，以确保我们的剖析器有足够的执行时间来挖掘和查看循环的走向。

> Before I tell you what my profiler showed me, spend a minute taking a few guesses. Where in clox's codebase do you think the VM spent most of its time? Is there any code we've written in previous chapters that you suspect is particularly slow?

在我告诉你剖析器显示了什么之前，先花点时间猜一下。你认为在clox的代码库中，虚拟机的大部分时间都花在了哪里？我们在前几章所写的代码中，有没有你怀疑特别慢的？

> Here's what I found: Naturally, the function with the greatest inclusive time is `run()`. (**Inclusive time** means the total time spent in some function and all other functions it calls—the total time between when you enter the function and when it returns.) Since `run()` is the main bytecode execution loop, it drives everything.

下面是我的发现：自然，非独占时间最大的函数是`run()`。（**非独占时间（Inclusive time）**是指在某个函数及其调用的所有其它函数中所花费的总时间——即从你进入函数到函数返回之间的总时间。）因为`run()`是主要的字节码执行循环，它驱动着一切。

> Inside `run()`, there are small chunks of time sprinkled in various cases in the bytecode switch for common instructions like `OP_POP`, `OP_RETURN`, and `OP_ADD`. The big heavy instructions are `OP_GET_GLOBAL` with 17% of the execution time, `OP_GET_PROPERTY` at 12%, and `OP_INVOKE` which takes a whopping 42% of the total running time.

在`run()`内部，有小块的时间视不同情况分散在如`OP_POP`、`OP_RETURN`、`OP_ADD`等常见指令中。较大的重磅指令是占执行时间17%的`OP_GET_GLOBAL`，占12%的`OP_GET_PROPERTY`，还有占总运行时间的42%的`OP_INVOKE`。

> So we've got three hotspots to optimize? Actually, no. Because it turns out those three instructions spend almost all of their time inside calls to the same function: `tableGet()`. That function claims a whole 72% of the execution time (again, inclusive). Now, in a dynamically typed language, we expect to spend a fair bit of time looking stuff up in hash tables—it's sort of the price of dynamism. But, still, *wow.*

所以我们有三个热点需要优化？事实上，并不是。因为事实证明，这三条指令几乎所有的时间都花在了调用同一个函数上：`tableGet()`。这个函数占用了整整72%的执行时间（同样的，非独占时间）。现在，在一个动态类型语言中，我们预想到会花费相当多的时间在哈希表中查找内容——这算是动态的代价，但是，仍旧让人惊叹。

> ### 30.2.1 Slow key wrapping

30.2.1 缓慢的键包装

> If you take a look at `tableGet()`, you'll see it's mostly a wrapper around a call to `findEntry()` where the actual hash table lookup happens. To refresh your memory, here it is in full:

如果查看一下`tableGet()`，你会发现它主要是对`findEntry()`调用的一个包装，而`findEntry()`是真正进行哈希表查找的地方。为了唤起你的记忆，下面是它的全部内容：

```
static Entry* findEntry(Entry* entries, int capacity,
                        ObjString* key) {
  uint32_t index = key->hash % capacity;
  Entry* tombstone = NULL;

  for (;;) {
    Entry* entry = &entries[index];
    if (entry->key == NULL) {
      if (IS_NIL(entry->value)) {
        // Empty entry.
        return tombstone != NULL ? tombstone : entry;
      } else {
        // We found a tombstone.
        if (tombstone == NULL) tombstone = entry;
      }
    } else if (entry->key == key) {
      // We found the key.
      return entry;
    }

    index = (index + 1) % capacity;
  }
}
```

> When running that previous benchmark—on my machine, at least—the VM spends 70% of the total execution time on *one line* in this function. Any guesses as to which one? No? It's this:

在运行之前的基准测试时——至少在我的机器上是这样——虚拟机将总执行时间的70%花费在这个函数的*一行*代码上。能猜到是哪一行吗？猜不到？是这一行：

```
uint32_t index = key->hash % capacity;
```

> That pointer dereference isn't the problem. It's the little `%`. It turns out the modulo operator is *really* slow. Much slower than other arithmetic operators. Can we do something better?

问题不在于指针解引用，而是那个小小的`%`。事实证明，取模操作符*真的*很慢。比其它算术运算符慢得多^6。我们能做得更好吗？

> In the general case, it's really hard to re-implement a fundamental arithmetic operator in user code in a way that's faster than what the CPU itself can do. After all, our C code ultimately compiles down to the CPU's own arithmetic operations. If there were tricks we could use to go faster, the chip would already be using them.

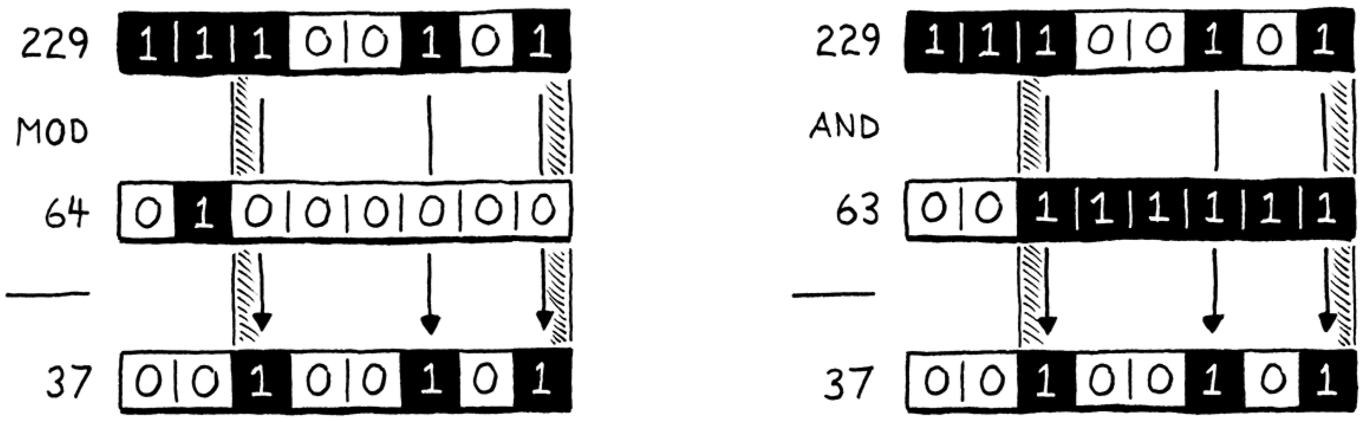
在一般情况下，要想在用户代码中重新实现一个算术运算符，而且要比CPU本身的运算速度更快，这是非常难的。毕竟，我们的C代码最终会被编译为CPU自己的算术运算。如果有什么技巧可以加速运算，芯片早就在使用了。

> However, we can take advantage of the fact that we know more about our problem than the CPU does. We use modulo here to take a key string's hash code and wrap it to fit within the bounds of the table's entry array. That array starts out at eight elements and grows by a factor of two each time. We know—and the CPU and C compiler do not—that our table's size is always a power of two.

然而，我们可以利用这一事实：我们比CPU更了解我们的问题。我们在这里使用取模将键字符串的哈希码包装到表的项数组的大小范围内。该数组开始时有8个元素，每次增加2倍。我们知道（CPU和C编译器都不知道）我们的表大小总是2的幂。

> Because we're clever bit twiddlers, we know a faster way to calculate the remainder of a number modulo a power of two: **bit masking**. Let's say we want to calculate 229 modulo 64. The answer is 37, which is not particularly apparent in decimal, but is clearer when you view those numbers in binary:

因为我们是聪明的位操作者，我们知道一个更快的方法来计算一个数以2的幂为模的余数：**位掩码**。假设我们要计算229对64取模。答案是37，这在十进制中不是特别明显，但当你用二进制查看这些数字时，就会更清楚：



> On the left side of the illustration, notice how the result (37) is simply the dividend (229) with the highest two bits shaved off? Those two highest bits are the bits at or to the left of the divisor’s single 1 bit.

在图的左侧，注意结果（37）是如何简单地将除数（229）的最高两位削除？这两个最高的位是被除数第一个1及其左侧的比特位。

> On the right side, we get the same result by taking 229 and bitwise AND-ing it with 63, which is one less than our original power of two divisor. Subtracting one from a power of two gives you a series of 1 bits. That is exactly the mask we need in order to strip out those two leftmost bits.

在右边，我们将299和63（原来的2的幂除数减一）进行按位与操作，也可以得到同样的结果。2的幂减去1会得到一系列的1比特。这正是我们需要的掩码，以便剥离掉最左侧的两个比特。

> In other words, you can calculate a number modulo any power of two by simply AND-ing it with that power of two minus one. I’m not enough of a mathematician to *prove* to you that this works, but if you think it through, it should make sense. We can replace that slow modulo operator with a very fast decrement and bitwise AND. We simply change the offending line of code to this:

换句话说，要计算某个数与任何2的幂的模数，你可以简单地将该数与2的幂减1进行位相与。我不是一个数学家，无法向你证明这一点，但如果你仔细想想，这应该是有道理的。我们可以用一个非常快的减法和按位与运算来替换那个缓慢的模运算。我们只是简单地将那行代码改为：

table.c，在findEntry()方法中替换1行：

```
static Entry* findEntry(Entry* entries, int capacity,
                        ObjString* key) {
  // 替换部分开始
  uint32_t index = key->hash & (capacity - 1);
  // 替换部分结束
  Entry* tombstone = NULL;
```

> CPUs love bitwise operators, so it’s hard to improve on that.

CPU喜欢位运算，因此很难在此基础上进行改进^7。

> Our linear probing search may need to wrap around the end of the array, so there is another modulo in `findEntry()` to update.

我们的线性探测搜索可能需要在数组的末尾绕回起点，所以在`findEntry()`中还有一个模运算需要更新。

table.c，在findEntry()方法中替换1行：

```
      // We found the key.
        return entry;
      }
      // 替换部分开始
      index = (index + 1) & (capacity - 1);
      // 替换部分结束
    }
```

> This line didn't show up in the profiler since most searches don't wrap.

这一行没有出现在剖析器中，是因为大部分搜索都不需要绕回。

> The `findEntry()` function has a sister function, `tableFindString()` that does a hash table lookup for interning strings. We may as well apply the same optimizations there too. This function is called only when interning strings, which wasn't heavily stressed by our benchmark. But a Lox program that created lots of strings might noticeably benefit from this change.

`findEntry()`函数有一个姊妹函数，`tableFindString()`，它为驻留的字符串做哈希表查询。我们不妨在这里也应用同样的优化。该函数只在对字符串进行驻留时才会被调用，我们的基准测试中没有特别侧重这一点。但是一个创建大量字符串的Lox程序可能会从这个调整中明显受益。

table.c，在tableFindString()方法中替换1行：

```
    if (table->count == 0) return NULL;
    // 替换部分开始
    uint32_t index = hash & (table->capacity - 1);
    // 替换部分结束
    for (;;) {
      Entry* entry = &table->entries[index];
```

> And also when the linear probing wraps around.

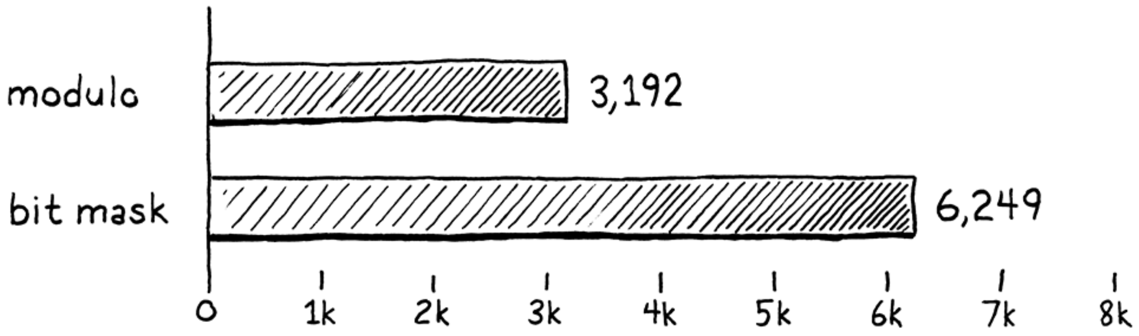
当线性探索绕回起点时也是如此。

table.c，在tableFindString()方法中替换1行：

```
      return entry->key;
      }
      // 替换部分开始
      index = (index + 1) & (table->capacity - 1);
      // 替换部分结束
    }
```

> Let’s see if our fixes were worth it. I tweaked that zoological benchmark to count how many batches of 10,000 calls it can run in ten seconds. More batches equals faster performance. On my machine using the unoptimized code, the benchmark gets through 3,192 batches. After this optimization, that jumps to 6,249.

来看看我们的修补是否值得。我调整了前面的动物学基准测试，计算它能在10秒内运行多少批的10000次调用^8。处理批次越多，性能越快。在我的机器上，使用未优化的代码，基准测试可以执行3192个批次。经过优化之后，这个数字跃升到了6249。



> That’s almost exactly twice as much work in the same amount of time. We made the VM twice as fast (usual caveat: on this benchmark). That is a massive win when it comes to optimization. Usually you feel good if you can claw a few percentage points here or there. Since methods, fields, and global variables are so prevalent in Lox programs, this tiny optimization improves performance across the board. Almost every Lox program benefits.

在同样的时间内，工作量几乎是原来的两倍。我们让虚拟机的速度提高了一倍（警告：在这个基准测试中）。对于优化来说，这是一个巨大的胜利。通常情况下，如果你能在这里或那里提升几个百分点，你都会感觉很好。因为方法、字段和全局变量在Lox程序中非常普遍，因此这个微小的优化可以全面提高性能。几乎每个Lox程序都会受益。

> Now, the point of this section is *not* that the modulo operator is profoundly evil and you should stamp it out of every program you ever write. Nor is it that micro-optimization is a vital engineering skill. It’s rare that a performance problem has such a narrow, effective solution. We got lucky.

现在，本节的重点不是说取模运算符非常邪恶，你要把它从你写的每个程序中剔除。也不是说微优化是一项重要的工程技能。很少有一个性能问题具有如此狭窄、有效的解决方案。我们很幸运。

> The point is that we didn’t *know* that the modulo operator was a performance drain until our profiler told us so. If we had wandered around our VM’s codebase blindly guessing at hotspots, we likely wouldn’t have noticed it. What I want you to take away from this is how important it is to have a profiler in your toolbox.

关键在于，直到剖析器告诉我们，我们才知道取模运算符是一个性能损耗。如果我们在虚拟机的代码库中盲目地猜测热点，我们可能不会注意到它。我想让你从中学到的是，在你的工具箱中拥有一个剖析器是多么重要。

> To reinforce that point, let’s go ahead and run the original benchmark in our now-optimized VM and see what the profiler shows us. On my machine, `tableGet()` is still a fairly large chunk of execution time. That’s to be expected for a dynamically typed language. But it has dropped from 72% of the total execution time down to 35%. That’s much more in line with what we’d like to see and shows that our optimization didn’t just make the program faster, but made it faster *in the way we expected*. Profilers are as useful for verifying solutions as they are for discovering problems.

为了强化这一点，我们继续在现在已优化的虚拟机中运行最初的基准测试，看看剖析器会显示什么。在我的机器上，`tableGet()`仍然占用了相当大的执行时间。对于动态类型的语言来说，这是可以预期的结果。但是它已经从72%下降到了35%。这更符合我们希望看到的情况，表明我们的优化不仅使程序更快，而且是以预期的方式使它更快。剖析器在验证解决方法时和发现问题时一样有用。

30.3 NaN Boxing

30.3 NaN装箱

> This next optimization has a very different feel. Thankfully, despite the odd name, it does not involve punching your grandmother. It's different, but not, like, *that* different. With our previous optimization, the profiler told us where the problem was, and we merely had to use some ingenuity to come up with a solution.

接下来的这个优化有着非常不同的感觉。值得庆幸的是，虽然它的名字很奇怪，但它并不会推翻一切。确实不同，但不会那么不同。在我们之前的优化中，剖析器会告诉我们问题出在哪里，而我们只需要发挥一些聪明才智就可以想出解决方案。

> This optimization is more subtle, and its performance effects more scattered across the virtual machine. The profiler won't help us come up with this. Instead, it was invented by someone thinking deeply about the lowest levels of machine architecture.

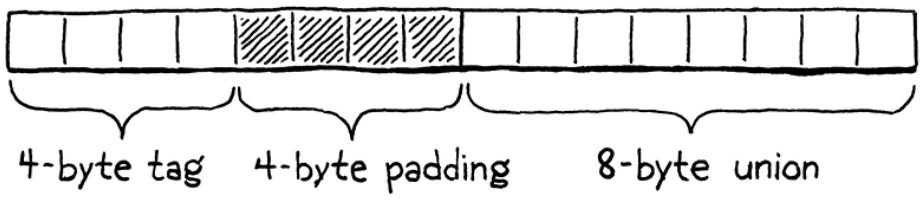
这个优化更加微妙，它对性能的影响在虚拟机在更加分散。剖析器无法帮我们找到它。相反，它是由一个对机器架构底层进行深入思考的人发明的^9。

> Like the heading says, this optimization is called **NaN boxing** or sometimes **NaN tagging**. Personally I like the latter name because "boxing" tends to imply some kind of heap-allocated representation, but the former seems to be the more widely used term. This technique changes how we represent values in the VM.

正如标题所说，这种优化称为**NaN装箱**，有时也被称为**NaN标记**。我个人更喜欢后者，因为"装箱"往往意味着某种堆分配的表示形式，但前者似乎是使用广泛的术语。这种技术改变了我们在虚拟机中表示值的方式。

> On a 64-bit machine, our Value type takes up 16 bytes. The struct has two fields, a type tag and a union for the payload. The largest fields in the union are an Obj pointer and a double, which are both 8 bytes. To keep the union field aligned to an 8-byte boundary, the compiler adds padding after the tag too:

在64位机器上，我们的Value类型占用了16个字节。该结构体中有两个字段，一个类型标签和一个存储有效载荷的联合体。联合体中最大的字段是一个Obj指针和一个double值，都是8字节。为了使联合体字段与8字节边界对齐，编译器也在标签后面添加了填充：



> That's pretty big. If we could cut that down, then the VM could pack more values into the same amount of memory. Most computers have plenty of RAM these days, so the direct memory savings

> aren't a huge deal. But a smaller representation means more Values fit in a cache line. That means fewer cache misses, which affects *speed*.

真可真够大的。如果我们能把它减少，那虚拟机就能在相同的内存中装入更多的值。现在大多数计算机都有足够的RAM，所以节省直接内存不是什么大问题。但是更小的表示方式意味着有更多的值可以放入缓存行中。这意味着更少的缓存失误，从而影响*速度*。

> If Values need to be aligned to their largest payload size, and a Lox number or Obj pointer needs a full 8 bytes, how can we get any smaller? In a dynamically typed language like Lox, each value needs to carry not just its payload, but enough additional information to determine the value's type at runtime. If a Lox number is already using the full 8 bytes, where could we squirrel away a couple of extra bits to tell the runtime "this is a number"?

既然Value需要与最大的有效载荷对齐，而且Lox数值或Obj指针需要完整的8个字节，那我们如何才能变得更小呢？在Lox这样的动态类型语言中，每个值不仅需要携带其有效载荷，还需要携带足够多的附加信息，以便在运行时确定值的类型。如果一个Lox的数字已经用了整整8个字节，那我们可以在哪里偷取两个额外的比特来告诉运行时"这是一个数字"？

> This is one of the perennial problems for dynamic language hackers. It particularly bugs them because statically typed languages don't generally have this problem. The type of each value is known at compile time, so no extra memory is needed at runtime to track it. When your C compiler compiles a 32-bit int, the resulting variable gets *exactly* 32 bits of storage.

这是动态语言黑客长期面临的问题之一。因为静态类型语言通常不存在这个问题，所以这让他们特别困扰。每个值的类型在编译时就已经知道了，所以在运行时不需要额外的内存来记录这些信息。当你的C语言编译器编译一个32位的int时，产生的变量会得到*正好*32位的存储空间。

> Dynamic language folks hate losing ground to the static camp, so they've come up with a number of very clever ways to pack type information and a payload into a small number of bits. NaN boxing is one of those. It's a particularly good fit for languages like JavaScript and Lua, where all numbers are double-precision floating point. Lox is in that same boat.

动态语言的人讨厌输给静态阵营，所以他们想出了许多非常聪明的方法，将类型信息和有效载荷打包到少量的比特中。NaN装箱就是其中之一。它特别适合于像JavaScript和Lua这样的语言，在这些语言中，所有数字都是双精度浮点数。Lox也是如此。

> ### 30.3.1 What is (and is not) a number?

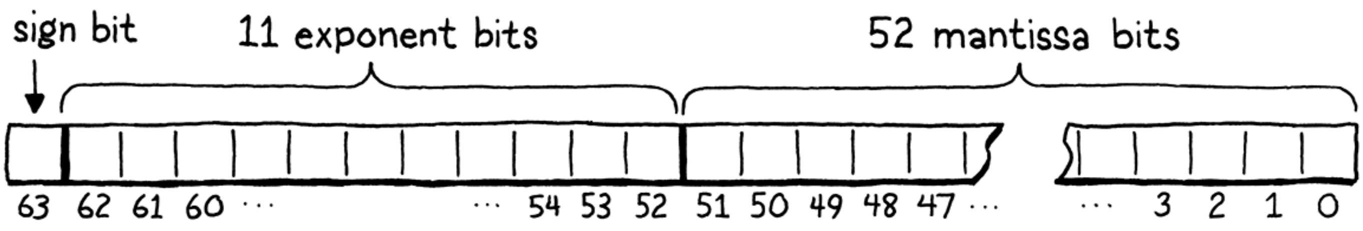
30.3.1 什么是（以及不是）数值？

> Before we start optimizing, we need to really understand how our friend the CPU represents floating-point numbers. Almost all machines today use the same scheme, encoded in the venerable scroll IEEE 754, known to mortals as the "IEEE Standard for Floating-Point Arithmetic".

在开始优化之前，我们需要真正了解我们的朋友CPU是如何表示浮点数的。现在几乎所有的机器都使用相同的方案，编码在古老的卷轴IEEE 754中，凡人们称之为"IEEE浮点运算标准"。

> In the eyes of your computer, a 64-bit, double-precision, IEEE floating-point number looks like this:

在你的计算机看来，一个64位、双精度的IEEE浮点数是这样的：



> - Starting from the right, the first 52 bits are the **fraction**, **mantissa**, or **significand** bits. They represent the significant digits of the number, as a binary integer.
> - Next to that are 11 **exponent** bits. These tell you how far the mantissa is shifted away from the decimal (well, binary) point.
> - The highest bit is the **sign bit**, which indicates whether the number is positive or negative.

- 从右边开始，前52位是**分数**、**尾数**或**有效位**。它们以二进制整数形式表示数值的有效数字。
- 接下来是11个**指数**位。它们会告诉你尾数中的小数点要移动多少位。
- 最高的位是**符号**位，表示这个数值是正数还是负数^10。

> I know that’s a little vague, but this chapter isn’t a deep dive on floating point representation. If you want to know how the exponent and mantissa play together, there are already better explanations out there than I could write.

我知道这有一点模糊，但这一章并不是对浮点数表示法的深入探讨。如果你想知道指数和尾数是如何互相作用的，外面已经有比我写得更好的解释了。

> The important part for our purposes is that the spec carves out a special case exponent. When all of the exponent bits are set, then instead of just representing a really big number, the value has a different meaning. These values are “Not a Number” (hence, **NaN**) values. They represent concepts like infinity or the result of division by zero.

对于我们的目的来说，重要的部分是该规范列出了一个特殊情况下的指数。当指数位全部置为1，这个值就不再表示一个非常大的数字了，而是有着不同的含义。这些值是“非数字”（Not a Number，即**NaN**）值。它们代表了像无穷或除0结果这样的概念。

> *Any* double whose exponent bits are all set is a NaN, regardless of the mantissa bits. That means there’s lots and lots of *different* NaN bit patterns. IEEE 754 divides those into two categories. Values where the highest mantissa bit is 0 are called **signalling NaNs**, and the others are **quiet NaNs**. Signalling NaNs are intended to be the result of erroneous computations, like division by zero. A chip may detect when one of these values is produced and abort a program completely. They may self-destruct if you try to read one.

任何指数位全部被置为1的double数都是NaN，无论尾数位是什么。这意味着有很多*不同*的NaN模式。IEEE 754将其分为两类。最高尾数位为0的值被称为**信号NaN**，其它的是**静默NaN**。信号NaN是错误计算的结果，如除以0。当这些值被生成时，芯片可以检测到并完全中止程序^11。如果你试图读取这些值，它们可能会自毁。

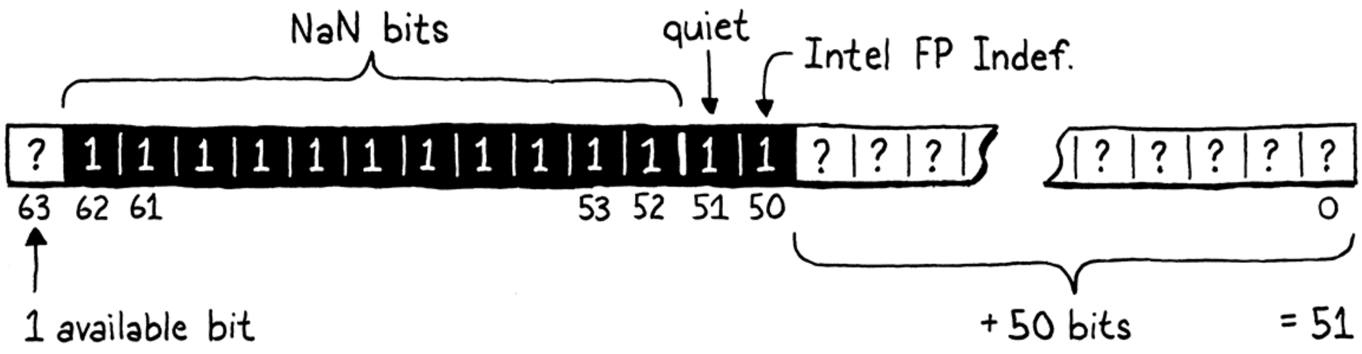
> Quiet NaNs are supposed to be safer to use. They don’t represent useful numeric values, but they should at least not set your hand on fire if you touch them.

静默NaN使用起来更安全。它们不代表有用的数值，但它们至少不会一碰就着。

> Every double with all of its exponent bits set and its highest mantissa bit set is a quiet NaN. That leaves 52 bits unaccounted for. We’ll avoid one of those so that we don’t step on Intel’s “QNaN Floating-Point

> Indefinite" value, leaving us 51 bits. Those remaining bits can be anything. We're talking 2,251,799,813,685,248 unique quiet NaN bit patterns.

每一个所有指数位置1、最高尾数位置1的double都是一个静默NaN。这就留下了52个未解释的位。我们会避开其中一个，这样我们就不会踩到Intel的"QNaN浮点不确定"值，剩下51位。这些剩余的比特可以是任何东西。我们现在说的是2,251,799,813,685,248独一无二的静默NaN位模式。



> This means a 64-bit double has enough room to store all of the various different numeric floating-point values and *also* has room for another 51 bits of data that we can use however we want. That's plenty of room to set aside a couple of bit patterns to represent Lox's `nil`, `true`, and `false` values. But what about Obj pointers? Don't pointers need a full 64 bits too?

这意味着一个64位的double有足够的框架存储所有不同的浮点数值，*还*有52位的数据空间供我们随意使用。这样就有足够的空间来预留几个位来表示Lox的`nil`、`true`和`false`值。但是Obj的指针呢？指针不是也需要64位吗？

> Fortunately, we have another trick up our other sleeve. Yes, technically pointers on a 64-bit architecture are 64 bits. But, no architecture I know of actually uses that entire address space. Instead, most widely used chips today only ever use the low 48 bits. The remaining 16 bits are either unspecified or always zero.

幸运的是，我们还有另一个妙招。是的，从技术上讲，64位架构上的指针是64位的。但是，我所知道的架构中没有一个真正使用了整个地址空间。相反，如今大多数广泛使用的芯片只使用低48位^12。剩余16位要么未指定，要么始终为零。

> If we've got 51 bits, we can stuff a 48-bit pointer in there with three bits to spare. Those three bits are just enough to store tiny type tags to distinguish between `nil`, Booleans, and Obj pointers.

如果我们有51比特位，可以把一个48位的指针塞进去，还有3比特位的空闲。这三个比特位刚好可以用来存储微小的类型标记来区分`nil`、布尔值和Obj指针。

> That's NaN boxing. Within a single 64-bit double, you can store all of the different floating-point numeric values, a pointer, or any of a couple of other special sentinel values. Half the memory usage of our current Value struct, while retaining all of the fidelity.

这就是NaN装箱。在一个64位的double中，你可以存储所有不同的浮点数值、一个指针或其它一些特殊的标示值。这比我们当前Value结构体少了一半的内存占用量，同时保留了所有的精确性。

> What's particularly nice about this representation is that there is no need to *convert* a numeric double value into a "boxed" form. Lox numbers *are* just normal, 64-bit doubles. We still need to *check* their

> type before we use them, since Lox is dynamically typed, but we don't need to do any bit shifting or pointer indirection to go from "value" to "number".

这种表示方法的特别之处在于，不需要将数值类型的double值*转换*为一个"装箱后的"形式。Lox中的数字只是普通的64位double。在使用之前，我们仍然需要*检查*它们的类型，因为Lox是动态类型的，但我们不需要做任何的数位偏移或指针引用来完成从"值"到"数"的转换。

> For the other value types, there is a conversion step, of course. But, fortunately, our VM hides all of the mechanism to go from values to raw types behind a handful of macros. Rewrite those to implement NaN boxing, and the rest of the VM should just work.

对于其它的值类型，当然有一个转换步骤。但幸运的是，我们的虚拟机将从值到原始类型的所有机制都隐藏在少数几个宏后面。重写这些宏来实现NaN装箱，虚拟机的其它部分就可以正常工作了。

> ### 30.3.2 Conditional support

30.3.2 有条件地支持

> I know the details of this new representation aren't clear in your head yet. Don't worry, they will crystallize as we work through the implementation. Before we get to that, we're going to put some compile-time scaffolding in place.

我知道这个新表示形式的细节在你的脑子里还不清晰。不用担心，它们会在我们的实现过程中逐步具现化。在此之前，我们要放置一些编译时的脚手架。

> For our previous optimization, we rewrote the previous slow code and called it done. This one is a little different. NaN boxing relies on some very low-level details of how a chip represents floating-point numbers and pointers. It *probably* works on most CPUs you're likely to encounter, but you can never be totally sure.

对于我们之前的优化，我们重写之前的慢代码就可以宣告完成了。这一次则有点不同。NaN装箱依赖于芯片如何表示浮点数和指针等一些非常底层的细节。它也许适用于你可能遇到的大多数CPU，但你永远无法完全确定。

> It would suck if our VM completely lost support for an architecture just because of its value representation. To avoid that, we'll maintain support for *both* the old tagged union implementation of Value and the new NaN-boxed form. We select which representation we want at compile time using this flag:

如果我们的虚拟机仅仅因为某个架构的值表示形式而完全失去对它的支持，那就太糟糕了。为了避免这种情况，我们会保留对Value的旧的带标记联合体实现方式以及新的NaN装箱形式的支持。我们在编译时使用这个标志来选择我们想要的方法：

common.h，添加代码：

```
#include <stdint.h>
// 新增部分开始
#define NAN_BOXING
// 新增部分结束
#define DEBUG_PRINT_CODE
```

> If that's defined, the VM uses the new form. Otherwise, it reverts to the old style. The few pieces of code that care about the details of the value representation—mainly the handful of macros for wrapping and unwrapping Values—vary based on whether this flag is set. The rest of the VM can continue along its merry way.

如果定义了这个值，虚拟机就会使用新的形式。否则，它就会恢复旧的风格。少数关心值表示形式细节的几段代码——主要是用于包装和解包Value的少数几个宏——会根据这个标志是否被设置而有所不同。虚拟机的其它部分可以继续快乐的旅程。

> Most of the work happens in the "value" module where we add a section for the new type.

大部分工作都发生在"value"模块中，我们在其中为新类型添加一些代码。

value.h，添加代码：

```
typedef struct ObjString ObjString;
// 新增部分开始
#ifdef NAN_BOXING

typedef uint64_t Value;

#else
// 新增部分结束
typedef enum {
```

> When NaN boxing is enabled, the actual type of a Value is a flat, unsigned 64-bit integer. We could use double instead, which would make the macros for dealing with Lox numbers a little simpler. But all of the other macros need to do bitwise operations and uint64_t is a much friendlier type for that. Outside of this module, the rest of the VM doesn't really care one way or the other.

当启用NaN装箱时，Value的实际类型是一个扁平的、无符号的64位整数。我们可以用double代替，这会使处理Lox数字的宏更简单一些。但所有其它宏都需要进行位操作，而uint_64是一个更友好的类型。在这个模块之外，虚拟机的其它部分并不真正关心这一点。

> Before we start re-implementing those macros, we close the `#else` branch of the `#ifdef` at the end of the definitions for the old representation.

在我们开始重新实现这些宏之前，我们先关闭旧表示形式的定义末尾的`#ifdef`的`#else`分支。

value.h，添加代码：

```
#define OBJ_VAL(object)   ((Value){VAL_OBJ, {.obj = (Obj*)object}})
// 新增部分开始
#endif
// 新增部分结束
typedef struct {
```

> Our remaining task is simply to fill in that first #ifdef section with new implementations of all the stuff already in the #else side. We'll work through it one value type at a time, from easiest to hardest.

我们剩下的任务只是在第一个#ifdef部分中填入已经在#else部分存在的所有内容的新实现。我们会从最简单到最难，依次完成每个值类型的工作。

> ### 30.3.3 Numbers

30.3.3 数字

> We'll start with numbers since they have the most direct representation under NaN boxing. To "convert" a C double to a NaN-boxed clox Value, we don't need to touch a single bit—the representation is exactly the same. But we do need to convince our C compiler of that fact, which we made harder by defining Value to be uint64_t.

我们会从数字开始，因为它们在NaN装箱方式中有最直接的表示形式。要将C语言中的double"转换"为一个NaN装箱后的clox Value，我们不需要改动任何一个比特——其表示方式是完全相同的。但我们确实需要说服我们的C编译器相信这一事实，我们将Value定义为uint64_t使之变得更加困难。

> We need to get the compiler to take a set of bits that it thinks are a double and use those same bits as a uint64_t, or vice versa. This is called **type punning**. C and C++ programmers have been doing this since the days of bell bottoms and 8-tracks, but the language specifications have hesitated to say which of the many ways to do this is officially sanctioned.

我们需要让编译器接受一组它认为是double的比特，并作为uint64_t来使用，反之亦然。这就是所谓的**类型双关**。C和C++程序员早在喇叭裤和8音轨的时代就开始这样做了，但语言规范却一直犹豫不决，不知道哪种方法是官方认可的^13。

> I know one way to convert a double to Value and back that I believe is supported by both the C and C++ specs. Unfortunately, it doesn't fit in a single expression, so the conversion macros have to call out to helper functions. Here's the first macro:

我知道一种将double转换为Value并反向转换的方法，我相信C和C++规范都支持该方法。不幸的是，它不适合在一个表达式中使用，因此转换宏必须调用辅助函数。下面是第一个宏：

value.h，添加代码：

```
typedef uint64_t Value;
// 新增部分开始
#define NUMBER_VAL(num) numToValue(num)
// 新增部分结束
#else
```

> That macro passes the double here:

这个宏会将double传递到这里：

value.h，添加代码：

```
#define NUMBER_VAL(num) numToValue(num)
// 新增部分开始
static inline Value numToValue(double num) {
  Value value;
  memcpy(&value, &num, sizeof(double));
  return value;
}
// 新增部分结束
#else
```

> I know, weird, right? The way to treat a series of bytes as having a different type without changing their value at all is `memcpy()`? This looks horrendously slow: Create a local variable. Pass its address to the operating system through a syscall to copy a few bytes. Then return the result, which is the exact same bytes as the input. Thankfully, because this *is* the supported idiom for type punning, most compilers recognize the pattern and optimize away the `memcpy()` entirely.

我知道，很奇怪，对吗？在不改变值的情况下，将一系列字节视为具有不同类型的方式是`memcpy()`？这看起来慢的可怕：创建一个局部变量；通过系统调用 将其地址传递给操作系统，以复制几个字节；然后返回结果，这个结果与输入的字节完全相同。值得庆幸的是，由于这是类型双关的习惯用法，大部分编译器都能识别这种模式，并完全优化掉`memcpy()`。

> “Unwrapping” a Lox number is the mirror image.

“拆包”一个Lox数字就是镜像操作。

value.h，添加代码：

```
typedef uint64_t Value;
// 新增部分开始
#define AS_NUMBER(value)    valueToNum(value)
// 新增部分结束
#define NUMBER_VAL(num) numToValue(num)
```

> That macro calls this function:

这个宏会调用下面的函数：

value.h，添加代码：

```
#define NUMBER_VAL(num) numToValue(num)
// 新增部分开始
static inline double valueToNum(Value value) {
  double num;
  memcpy(&num, &value, sizeof(Value));
  return num;
}
// 新增部分结束
static inline Value numToValue(double num) {
```

> It works exactly the same except we swap the types. Again, the compiler will eliminate all of it. Even though those calls to `memcpy()` will disappear, we still need to show the compiler *which* `memcpy()` we're calling so we also need an include.

它的工作原理完全一样，只是交换了类型。同样，编译器会消除所有这些。尽管对`memcpy()`的那些调用会消失，我们仍然需要向编译器显示我们正在调用*哪个*`memcpy()`，因此我们也需要引入一下^14。

value.h，添加代码：

```
#define clox_value_h
// 新增部分开始
#include <string.h>
// 新增部分结束
#include "common.h"
```

> That was a lot of code to ultimately do nothing but silence the C type checker. Doing a runtime type *test* on a Lox number is a little more interesting. If all we have are exactly the bits for a double, how do we tell that it *is* a double? It's time to get bit twiddling.

其中是大量的代码，最终除了让C语言类型检查器保持沉默之外，什么也没做。对一个Lox数字进行运行时类型*测试*就比较有趣了。如果我们拿到的所有比特位正好是一个double，如何判断它是一个double呢？是时候玩一些位操作技巧了。

value.h，添加代码：

```
typedef uint64_t Value;
// 新增部分开始
#define IS_NUMBER(value)    (((value) & QNAN) != QNAN)
// 新增部分结束
#define AS_NUMBER(value)    valueToNum(value)
```

> We know that every Value that is *not* a number will use a special quiet NaN representation. And we presume we have correctly avoided any of the meaningful NaN representations that may actually be produced by doing arithmetic on numbers.

我们知道，每个*不是*数字的Value都会使用一个特殊的静默NaN表示形式。而且假定我们已经正确地避免了任何有意义的NaN表示形式（这些实际上可能是通过对数字进行算术运算产生的）。

> If the double has all of its NaN bits set, and the quiet NaN bit set, and one more for good measure, we can be pretty certain it is one of the bit patterns we ourselves have set aside for other types. To check that, we mask out all of the bits except for our set of quiet NaN bits. If *all* of those bits are set, it must be a NaN-boxed value of some other Lox type. Otherwise, it is actually a number.

如果某个double值的NaN比特位置为1，而且静默NaN比特位也置为1，还有一个比特位也被置为1，那我们就可以非常肯定它是我们为其它类型预留的比特模式之一^15。为了验证这一点，我们屏蔽掉除静默NaN置为1的

比特之外的所有其它比特位，如果这些位都被置为1了，那它一定是某个其它Lox类型的已NaN装箱的值。否则，它就是一个数字。

> The set of quiet NaN bits are declared like this:

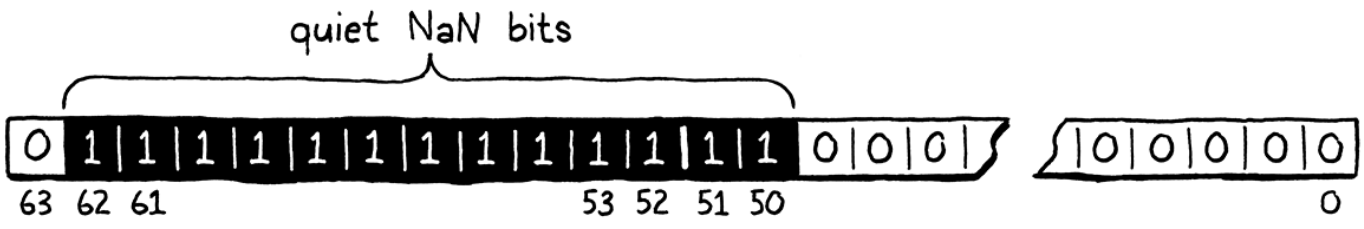
静默NaN的比特集合是这样声明的：

value.h，添加代码：

```
#ifdef NAN_BOXING
// 新增部分开始
#define QNAN     ((uint64_t)0x7ffc000000000000)
// 新增部分结束
typedef uint64_t Value;
```

> It would be nice if C supported binary literals. But if you do the conversion, you'll see that value is the same as this:

如果C支持二进制字面量就好了。但如果你做了转换，你会看到那个值是这样的：



> This is exactly all of the exponent bits, plus the quiet NaN bit, plus one extra to dodge that Intel value.

这正是所有的指数位，加上静默NaN比特位，再加上一个额外的用来规避英特尔值的比特位。

> 30.3.4 Nil, true, and false

30.3.4 Nil、true和false

> The next type to handle is `nil`. That's pretty simple since there's only one `nil` value and thus we need only a single bit pattern to represent it. There are two other singleton values, the two Booleans, `true` and `false`. This calls for three total unique bit patterns.

下一个要处理的类型是`nil`。这非常简单，因为只有一个`nil`值，因此我们只需要1 个比特位模式来表示它。还有另外两个单例值，即两个布尔值，`true`和`false`。这总共需要三种唯一的比特位模式。

> Two bits give us four different combinations, which is plenty. We claim the two lowest bits of our unused mantissa space as a "type tag" to determine which of these three singleton values we're looking at. The three type tags are defined like so:

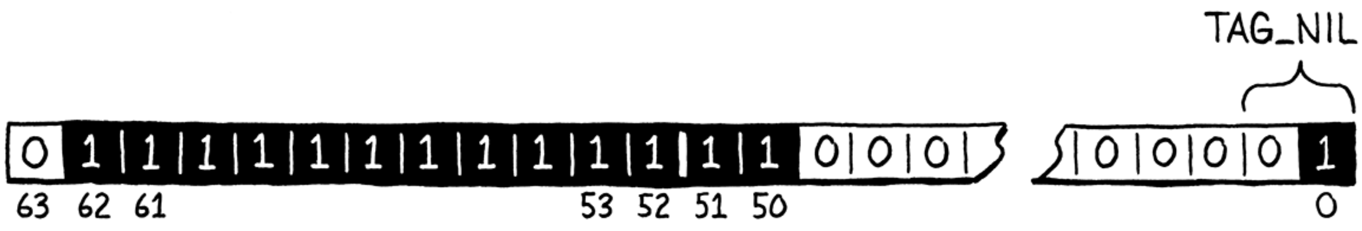
两个比特可以得到四种不同的组合，这已经足够了。我们要求将未使用的尾数中的两个最低位作为"类型标签"，以确定我们正面对的是这三个单例值中的哪一个。这三个类型标签定义如下：

value.h，添加代码：

```
#define QNAN     ((uint64_t)0x7ffc000000000000)
// 新增部分开始
#define TAG_NIL   1 // 01.
#define TAG_FALSE 2 // 10.
#define TAG_TRUE  3 // 11.
// 新增部分结束
typedef uint64_t Value;
```

> Our representation of `nil` is thus all of the bits required to define our quiet NaN representation along with the `nil` type tag bits:

因此，我们的`nil`表示形式的所有比特位就是定义静默NaN表示形式所需的所有比特位，以及`nil`类型的标记位：



> In code, we check the bits like so:

在代码中，我们这样来检查：

value.h，添加代码：

```
#define AS_NUMBER(value)    valueToNum(value)
// 新增部分开始
#define NIL_VAL         ((Value)(uint64_t)(QNAN | TAG_NIL))
// 新增部分结束
#define NUMBER_VAL(num) numToValue(num)
```

> We simply bitwise OR the quiet NaN bits and the type tag, and then do a little cast dance to teach the C compiler what we want those bits to mean.

我们只是将静默NaN比特位与类型标签进行按位或运算，然后做一点强制转换来告诉C编译器我们希望这些位表示什么意思。

> Since `nil` has only a single bit representation, we can use equality on uint64_t to see if a Value is `nil`.

由于`nil`只有一个比特表示形式，我们可以对uint64_t使用等号来判断某个Value是否是`nil`。

value.h，添加代码：

```
typedef uint64_t Value;
// 新增部分开始
#define IS_NIL(value)       ((value) == NIL_VAL)
```

```
// 新增部分结束
#define IS_NUMBER(value)    (((value) & QNAN) != QNAN)
```

> You can guess how we define the `true` and `false` values.

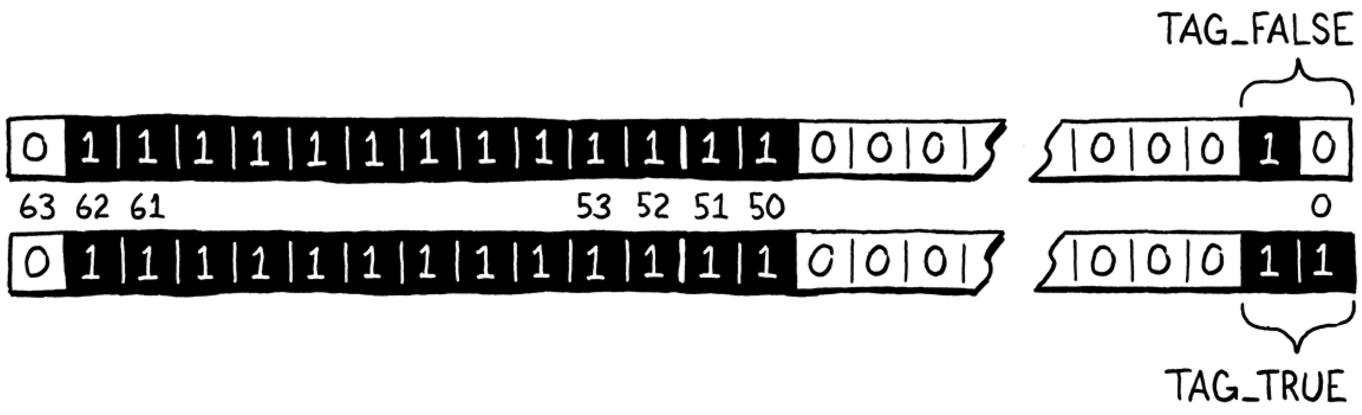
你可以猜到我们如何定义`true`和`false`值。

value.h，添加代码：

```
#define AS_NUMBER(value)    valueToNum(value)
// 新增部分开始
#define FALSE_VAL       ((Value)(uint64_t)(QNAN | TAG_FALSE))
#define TRUE_VAL        ((Value)(uint64_t)(QNAN | TAG_TRUE))
// 新增部分结束
#define NIL_VAL         ((Value)(uint64_t)(QNAN | TAG_NIL))
```

> The bits look like this:

比特位看起来是这样的：



> To convert a C bool into a Lox Boolean, we rely on these two singleton values and the good old conditional operator.

为了将C语言bool转换为Lox的Boolean，我们依靠这两个单例值和古老的条件运算符。

value.h，添加代码：

```
#define AS_NUMBER(value)    valueToNum(value)
// 新增部分开始
#define BOOL_VAL(b)     ((b) ? TRUE_VAL : FALSE_VAL)
// 新增部分结束
#define FALSE_VAL       ((Value)(uint64_t)(QNAN | TAG_FALSE))
```

> There's probably a cleverer bitwise way to do this, but my hunch is that the compiler can figure one out faster than I can. Going the other direction is simpler.

可能有更聪明的位运算方式来实现这一点，但我的直觉是，编译器可以比我更快地找到一个方法。反过来就简单多了。

value.h，添加代码：

```
#define IS_NUMBER(value)    (((value) & QNAN) != QNAN)
// 新增部分开始
#define AS_BOOL(value)      ((value) == TRUE_VAL)
// 新增部分结束
#define AS_NUMBER(value)    valueToNum(value)
```

> Since we know there are exactly two Boolean bit representations in Lox—unlike in C where any non-zero value can be considered "true"—if it ain't `true`, it must be `false`. This macro does assume you call it only on a Value that you know *is* a Lox Boolean. To check that, there's one more macro.

因为我们知道在Lox中正好有两个Boolean的位表示形式——不像C语言中，任何非零值都可以被认为是"true"——如果它不是`true`，就一定是`false`。这个宏假设你只会在明知是Lox布尔值类型的Value上调用该方法。为了检查这一点，还有一个宏。

value.h，添加代码：

```
typedef uint64_t Value;
// 新增部分开始
#define IS_BOOL(value)      (((value) | 1) == TRUE_VAL)
// 新增部分结束
#define IS_NIL(value)       ((value) == NIL_VAL)
```

> That looks a little strange. A more obvious macro would look like this:

这里看起来有点奇怪。一个更直观的宏看起来应该是这样的：

```
#define IS_BOOL(v) ((v) == TRUE_VAL || (v) == FALSE_VAL)
```

> Unfortunately, that's not safe. The expansion mentions `v` twice, which means if that expression has any side effects, they will be executed twice. We could have the macro call out to a separate function, but, ugh, what a chore.

不幸的是，这并不安全。展开式中两次使用了`v`，这意味着如果表达式有任何副作用，它们将被执行两次。我们可以让宏调用到一个单独的函数，但是，唉，真麻烦。

> Instead, we bitwise OR a 1 onto the value to merge the only two valid Boolean bit patterns. That leaves three potential states the value can be in:

相反，我们在值上按位或1，来合并仅有的两个有效的Boolean比特位模式。这样，值就剩下了三种可能的状态：

> 1. It was FALSE_VAL and has now been converted to TRUE_VAL.
> 2. It was TRUE_VAL and the | 1 did nothing and it's still TRUE_VAL.
> 3. It's some other, non-Boolean value.

1. 之前是FALSE_VAL，现在转换为TRUE_VAL。
2. 之前是TRUE_VAL， | 1没有起任何作用，结果仍然是TRUE_VAL。
3. 它是其它的非布尔值。

> At that point, we can simply compare the result to TRUE_VAL to see if we're in the first two states or the third.

在此基础上，我们可以简单地将结果与TRUE_VAL进行比较，看看我们是处于前两个状态还是第三个状态。

> 30.3.5 Objects

30.3.5 对象

> The last value type is the hardest. Unlike the singleton values, there are billions of different pointer values we need to box inside a NaN. This means we need both some kind of tag to indicate that these particular NaNs *are* Obj pointers, and room for the addresses themselves.

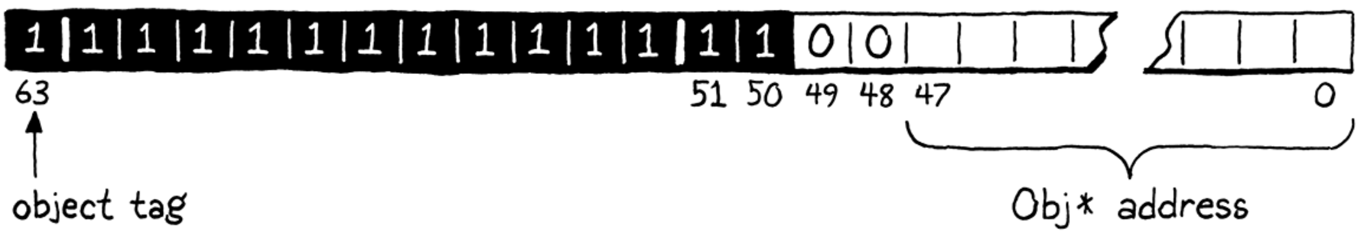
最后一种值类型是最难的。与单例值不同，我们需要在NaN中包含数十亿个不同的指针值。这意味着我们既需要某种标签来表明这些特定的NaN*是*Obj指针，也需要为这些地址本身留出空间。

> The tag bits we used for the singleton values are in the region where I decided to store the pointer itself, so we can't easily use a different bit there to indicate that the value is an object reference. However, there is another bit we aren't using. Since all our NaN values are not numbers—it's right there in the name—the sign bit isn't used for anything. We'll go ahead and use that as the type tag for objects. If one of our quiet NaNs has its sign bit set, then it's an Obj pointer. Otherwise, it must be one of the previous singleton values.

我们用于单例值的标签比特位处于我决定存储指针本身的区域，所以我们不能轻易地在那里使用不同的位来表明该值是一个对象引用^16。不过，还有一个位我们没有用到。因为所有的NaN值都不是数字——正如其名——符号位没有任何用途。我们会继续使用它来作为对象的类型标签。如果某个静默NaN的符号位被置为1，那么它就是一个Obj指针。否则，它一定是前面的单例值之一。

> If the sign bit is set, then the remaining low bits store the pointer to the Obj:

如果符号位被置1，那么剩余的低比特位会存储Obj指针：



> To convert a raw Obj pointer to a Value, we take the pointer and set all of the quiet NaN bits and the sign bit.

为了将一个原生Obj指针转换为Value，我们会接受指针并将所有的静默NaN比特位和符号位置1。

value.h，添加代码：

```
#define NUMBER_VAL(num) numToValue(num)
// 新增部分开始
#define OBJ_VAL(obj) \
    (Value)(SIGN_BIT | QNAN | (uint64_t)(uintptr_t)(obj))
// 新增部分结束
static inline double valueToNum(Value value) {
```

> The pointer itself is a full 64 bits, and in principle, it could thus overlap with some of those quiet NaN and sign bits. But in practice, at least on the architectures I've tested, everything above the 48th bit in a pointer is always zero. There's a lot of casting going on here, which I've found is necessary to satisfy some of the pickiest C compilers, but the end result is just jamming some bits together.

指针本身是一个完整的64位，原则上，它可能因此与某些静默NaN和符号位冲突。但实际上，至少在我测试过的架构中，指针中48位以上的所有内容都是零。这里进行了大量的类型转换。我们发现这对于满足一些最挑剔的C语言编译器来说是必要的，但最终的结果只是将这些比特位塞在一起^17。

> We define the sign bit like so:

我们这样定义符号位：

value.h，添加代码：

```
#ifdef NAN_BOXING
// 新增部分开始
#define SIGN_BIT ((uint64_t)0x8000000000000000)
// 新增部分结束
#define QNAN     ((uint64_t)0x7ffc000000000000)
```

> To get the Obj pointer back out, we simply mask off all of those extra bits.

为了取出Obj指针，我们只需把所有这些额外的比特位屏蔽掉。

value.h，添加代码：

```
#define AS_NUMBER(value)    valueToNum(value)
// 新增部分开始
#define AS_OBJ(value) \
    ((Obj*)(uintptr_t)((value) & ~(SIGN_BIT | QNAN)))
// 新增部分结束
#define BOOL_VAL(b)      ((b) ? TRUE_VAL : FALSE_VAL)
```

> The tilde (~), if you haven't done enough bit manipulation to encounter it before, is bitwise NOT. It toggles all ones and zeroes in its operand. By masking the value with the bitwise negation of the quiet NaN and sign bits, we *clear* those bits and let the pointer bits remain.

如果你没有做过足够多的位运算就可能没有遇到过，波浪号（`~`）是位运算中的NOT（按位取非）。它会切换操作数中所有的1和0。使用静默NaN和符号位按位取非的值作为掩码，对值进行屏蔽，我们可以*清除*这些比特位，并将指针比特保留下来。

> One last macro:

最后一个宏：

value.h，添加代码：

```
#define IS_NUMBER(value)    (((value) & QNAN) != QNAN)
// 新增部分开始
#define IS_OBJ(value) \
    (((value) & (QNAN | SIGN_BIT)) == (QNAN | SIGN_BIT))
// 新增部分结束
#define AS_BOOL(value)      ((value) == TRUE_VAL)
```

> A Value storing an Obj pointer has its sign bit set, but so does any negative number. To tell if a Value is an Obj pointer, we need to check that both the sign bit and all of the quiet NaN bits are set. This is similar to how we detect the type of the singleton values, except this time we use the sign bit as the tag.

存储Obj指针的Value的符号位被置1，但任意负数也是如此。为了判断Value是否为Obj指针，我们需要同时检查符号位和所有的静默NaN比特位。这与我们检测单例值类型的方法类似，这不过这次我们使用符号位作为标签。

> 30.3.6 Value functions

30.3.6 Value函数

> The rest of the VM usually goes through the macros when working with Values, so we are almost done. However, there are a couple of functions in the "value" module that peek inside the otherwise black box of Value and work with its encoding directly. We need to fix those too.

VM的其余部分在处理Value时通常都是通过宏，所以我们基本上已经完成了。但是，在"value"模块中，有几个函数会窥探Value黑匣子内部，并直接处理器编码。我们也需要修复这些问题。

> The first is `printValue()`. It has separate code for each value type. We no longer have an explicit type enum we can switch on, so instead we use a series of type tests to handle each kind of value.

第一个是`printValue()`。它针对每个值类型都有单独的代码。我们不再有一个明确的类型枚举进行switch，因此我们使用一系列的类型检查来处理每一种值。

value.c，在printValue()方法中添加代码：

```
void printValue(Value value) {
// 新增部分开始
#ifdef NAN_BOXING
  if (IS_BOOL(value)) {
      printf(AS_BOOL(value) ? "true" : "false");
    } else if (IS_NIL(value)) {
      printf("nil");
    } else if (IS_NUMBER(value)) {
      printf("%g", AS_NUMBER(value));
    } else if (IS_OBJ(value)) {
      printObject(value);
    }
#else
// 新增部分结束
  switch (value.type) {
```

> This is technically a tiny bit slower than a switch, but compared to the overhead of actually writing to a stream, it’s negligible.

从技术上讲，这比switch语句稍微慢一点点，但是与实际写入流的开销相比，它可以忽略不计。

> We still support the original tagged union representation, so we keep the old code and enclose it in the `#else` conditional section.

我们仍然支持原先的带标签联合体表示形式，因此我们保留旧代码，并将其包含在`#else`条件部分。

value.c，在printValue()方法中添加代码：

```
  }
// 新增部分开始
#endif
// 新增部分结束
}
```

> The other operation is testing two values for equality.

另一个操作是测试两个值是否相等。

value.c，在valuesEqual()方法中添加代码：

```
bool valuesEqual(Value a, Value b) {
// 新增部分开始
#ifdef NAN_BOXING
  return a == b;
#else
// 新增部分结束
  if (a.type != b.type) return false;
```

> It doesn’t get much simpler than that! If the two bit representations are identical, the values are equal. That does the right thing for the singleton values since each has a unique bit representation and they are only equal to themselves. It also does the right thing for Obj pointers, since objects use identity for equality—two Obj references are equal only if they point to the exact same object.

没有比这更简单的了！如果两个比特表示形式是相同的，则值就是相等的。这对于单例值来说是正确的，因为每个单例值都有唯一的位表示形式，而且它们只等于自己。对于Obj指针，它也做了正确的事情，因为对象使用本体来判断相等——只有当两个Obj指向完全相同的对象时，它们才相等。

> It's *mostly* correct for numbers too. Most floating-point numbers with different bit representations are distinct numeric values. Alas, IEEE 754 contains a pothole to trip us up. For reasons that aren't entirely clear to me, the spec mandates that NaN values are *not* equal to *themselves*. This isn't a problem for the special quiet NaNs that we are using for our own purposes. But it's possible to produce a "real" arithmetic NaN in Lox, and if we want to correctly implement IEEE 754 numbers, then the resulting value is not supposed to be equal to itself. More concretely:

对于数字来说，也*基本*是正确的。大多数具有不同位表示形式的浮点数是不同的数值。然而，IEEE 754中有一个坑，会让我们陷入困境。由于我不太清楚的原因，该规范规定NaN值*不*等于*自身*。对于我们自己使用的特殊的静默NaN来说，这不是问题。但是在Lox中产生一个"真正的"算术型NaN是有可能的，如果我们想正确地实现IEEE 754数字，那么产生的结果值就不等于它自身。更具体地说：

```
var nan = 0/0;
print nan == nan;
```

> IEEE 754 says this program is supposed to print "false". It does the right thing with our old tagged union representation because the `VAL_NUMBER` case applies `==` to two values that the C compiler knows are doubles. Thus the compiler generates the right CPU instruction to perform an IEEE floating-point equality.

IEEE 754表明，这个程序应该打印"false"。对于我们原先的带标签联合体表示形式来说，它是正确的，因为`VAL_NUMBER`将`==`应用于两个C编译器知道是double的值。因此，编译器会生成正确的CPU指令来执行IEEE浮点运算。

> Our new representation breaks that by defining Value to be a uint64_t. If we want to be *fully* compliant with IEEE 754, we need to handle this case.

我们的新表示形式由于将Value定义为uint64_t而打破了这一点。如果我们想完全符合IEEE 754的要求，就需要处理这种情况。

value.c，在valuesEqual()方法中添加代码：

```
#ifdef NAN_BOXING
  // 新增部分开始
  if (IS_NUMBER(a) && IS_NUMBER(b)) {
    return AS_NUMBER(a) == AS_NUMBER(b);
  }
  // 新增部分结束
  return a == b;
```

> I know, it's weird. And there is a performance cost to doing this type test every time we check two Lox values for equality. If we are willing to sacrifice a little compatibility—who *really* cares if NaN is not equal to itself?—we could leave this off. I'll leave it up to you to decide how pedantic you want to be.

我知道，这很奇怪。而且每次我们检查两个Lox值是否相等时，都要进行这种类型测试，这是有性能代价的。如果我们愿意牺牲一点兼容性——谁会*真正*关心NaN是否等于其本身呢？——我们可以忽略它。我把这个问题留给你，看看你想要有多“迂腐”^18。

> Finally, we close the conditional compilation section around the old implementation.

最后，我们关闭旧实现中的条件编译部分。

value.c，在valuesEqual()方法中添加代码：

```
  }
// 新增部分开始
#endif
// 新增部分结束
}
```

> And that’s it. This optimization is complete, as is our clox virtual machine. That was the last line of new code in the book.

就是这样。这个优化完成了，我们的clox虚拟机也完成了。这是本书中最后一行新代码。

> ## 30.3.7 Evaluating performance

30.3.7 评估性能

> The code is done, but we still need to figure out if we actually made anything better with these changes. Evaluating an optimization like this is very different from the previous one. There, we had a clear hotspot visible in the profiler. We fixed that part of the code and could instantly see the hotspot get faster.

代码完成了，但我们仍然需要弄清楚，我们是否真的通过这些修改获得了一些改进。评估这样的优化与之前的优化有很大不同。之前，我们可以在剖析器中看到一个明显的热点。我们修复了这部分代码，并立即看到热点部分变快了。

> The effects of changing the value representation are more diffused. The macros are expanded in place wherever they are used, so the performance changes are spread across the codebase in a way that’s hard for many profilers to track well, especially in an optimized build.

改变值表示形式的影响更加分散。在宏的任何地方都会进行对应的扩展，所以性能的变化会分散到整个代码库中，这对很多剖析器来说是很难跟踪的，尤其是在优化的构建中^19。

> We also can’t easily *reason* about the effects of our change. We’ve made values smaller, which reduces cache misses all across the VM. But the actual real-world performance effect of that change is highly dependent on the memory use of the Lox program being run. A tiny Lox microbenchmark may not have enough values scattered around in memory for the effect to be noticeable, and even things like the addresses handed out to us by the C memory allocator can impact the results.

我们也无法轻易推断出我们的改变所带来的影响。我们让Value变得更小，这就减少了虚拟机中的缓存丢失。但是，这一改变在真实世界中的实际性能影响在很大程度上取决于正在运行的Lox程序的内存使用情况。一个很小

的Lox微基准测试可能没有足够的值分散在内存中，因此效果也许不明显，甚至类似C语言地址分配器为我们提供的地址这样的东西也会影响结果。

> If we did our job right, basically everything gets a little faster, especially on larger, more complex Lox programs. But it is possible that the extra bitwise operations we do when NaN-boxing values nullify the gains from the better memory use. Doing performance work like this is unnerving because you can't easily *prove* that you've made the VM better. You can't point to a single surgically targeted microbenchmark and say, "There, see?"

如果我们的工作做对了，基本上所有东西都会变快一点，尤其是在更大、更复杂的Lox程序上。但是，我们对NaN装箱值执行的位操作可能会抵消更高效的内存使用所带来的收益。做这样的性能工作是令人不安的，因为你无法轻易地证明你已经使虚拟机变得更好了。你不能指着一个特定的微基准测试说："看到了吗？"

> Instead, what we really need is a *suite* of larger benchmarks. Ideally, they would be distilled from real-world applications—not that such a thing exists for a toy language like Lox. Then we can measure the aggregate performance changes across all of those. I did my best to cobble together a handful of larger Lox programs. On my machine, the new value representation seems to make everything roughly 10% faster across the board.

相反，我们真正需要的是一套更大的基准测试。理想情况下，这些基准测试应该是从真实世界的应用程序中提炼出来的——对于Lox这样的玩具语言来说，不存在这样的东西。然后我们可以测量所有这些测试的总体性能变化。我尽力拼凑了几个较大的Lox程序。在我的机器是，新的值表示形式似乎使所有的代码都全面提高了大约10%。

> That's not a huge improvement, especially compared to the profound effect of making hash table lookups faster. I added this optimization in large part because it's a good example of a certain *kind* of performance work you may experience, and honestly, because I think it's technically really cool. It might not be the first thing I would reach for if I were seriously trying to make clox faster. There is probably other, lower-hanging fruit.

这并不是一个巨大的改进，尤其是与哈希查找加速的深远影响相比。我添加这个优化，很大程度上是因为它是关于你可能遇到的*某种*性能工作的一个很好的例子，而且说实话，我认为它在技术上真的很酷。如果我真的想让clox变得更快的话，这应该不是我首先要做的事情。可能还有其它更容易实现的目标。

> But, if you find yourself working on a program where all of the easy wins have been taken, then at some point you may want to think about tuning your value representation. I hope this chapter has shined a light on some of the options you have in that area.

但是，如果你发现自己正在处理的程序中，所有容易赢得的东西都已经被拿走了，那么在某些时候，你可能要考虑调整一下值表示形式。我希望这一章能对你在这方面的一些选择有所启发。

> ## 30.4 Where to Next

30.4 前路何方

> We'll stop here with the Lox language and our two interpreters. We could tinker on it forever, adding new language features and clever speed improvements. But, for this book, I think we've reached a natural place to call our work complete. I won't rehash everything we've learned in the past many pages. You were there with me and you remember. Instead, I'd like to take a minute to talk about where you might go from here. What is the next step in your programming language journey?

关于Lox语言和我们的两个解释器，就到此为止了。我们可以一直对它进行修补，添加新的语言功能和巧妙的速度改进。但是，对于本书来说，我认为我们已经达到了一个可以宣告工作完成的状态。我不会重述我们在过去的许多章节中所学到的一切。你和我一起从那里过来，你都记得。相反，我想花点时间谈谈你今后的发展方向。你的编程语言之旅的下一步是什么？

> Most of you probably won't spend a significant part of your career working in compilers or interpreters. It's a pretty small slice of the computer science academia pie, and an even smaller segment of software engineering in industry. That's OK. Even if you never work on a compiler again in your life, you will certainly *use* one, and I hope this book has equipped you with a better understanding of how the programming languages you use are designed and implemented.

你们中的大多数人可能不会把职业生涯的大部分时间花在编译器或解释器上。这在计算机科学学术界中的一个相当小的部分，在工业软件工程中则是一个更小的部分。这也没关系。即使你一生中不再从事编译器工作，你也一定会使用它，而我希望这本书能让你更好地理解你所使用的编程语言是如何设计与实现的。

> You have also learned a handful of important, fundamental data structures and gotten some practice doing low-level profiling and optimization work. That kind of expertise is helpful no matter what domain you program in.

你还学习了一些重要的、基本的数据结构，并进行了一些底层剖析和优化工作的实践。无论你在哪个领域编程，这种专业知识都是有帮助的。

> I also hope I gave you a new way of looking at and solving problems. Even if you never work on a language again, you may be surprised to discover how many programming problems can be seen as language-*like*. Maybe that report generator you need to write can be modeled as a series of stack-based "instructions" that the generator "executes". That user interface you need to render looks an awful lot like traversing an AST.

我也希望我为你们提供了一种看待问题和解决问题的新方法。即使你不再从事语言工作，你也可能会惊讶地发现，有多少编程问题可以被视为*类似于*语言的问题[^20]。也许你需要编写的报告生成器可以被建模为一系列由生成器"执行"的、基于堆栈的"指令"。你需要渲染的用户界面看起来非常像遍历AST。

> If you do want to go further down the programming language rabbit hole, here are some suggestions for which branches in the tunnel to explore:

如果你确实想在编程语言领域中走得更远，这里有一些关于哪些方面可以探索的建议：

- > Our simple, single-pass bytecode compiler pushed us towards mostly runtime optimization. In a mature language implementation, compile-time optimization is generally more important, and the field of compiler optimizations is incredibly rich. Grab a classic compilers book, and rebuild the front end of clox or jlox to be a sophisticated compilation pipeline with some interesting intermediate representations and optimization passes.
  >
  > Dynamic typing will place some restrictions on how far you can go, but there is still a lot you can do. Or maybe you want to take a big leap and add static types and a type checker to Lox. That will certainly give your front end a lot more to chew on.

  我们这个简单的、单遍字节码编译器将我们推向了运行时优化。在一个成熟的语言实现中，编译时优化通常更重要，而且编译器优化的领域也非常丰富。找一本经典的编译器书籍[^21]，将clox或jlox的前端重构为一个复杂的编译管道，其中要包含一些有趣的中间表示形式和优化遍历。

动态类型会对你能走多远加以限制，但你仍然可以做很多事情。或者你想要来个大跃进，给Lox添加静态类型和类型检查器。这肯定会让你的前端有更多的东西可以细细咀嚼。

- > In this book, I aim to be correct, but not particularly rigorous. My goal is mostly to give you an *intuition* and a feel for doing language work. If you like more precision, then the whole world of programming language academia is waiting for you. Languages and compilers have been studied formally since before we even had computers, so there is no shortage of books and papers on parser theory, type systems, semantics, and formal logic. Going down this path will also teach you how to read CS papers, which is a valuable skill in its own right.

  在本书中，我的目标是正确，但不是特别严谨。我的目标主要是给你一个*直观感受*和做语言工作的感觉。如果你想要更精确的感觉，那么整个编程语言学术界都在等着你。在我们拥有计算机之前，语言和编译器就已经被正式研究过了，因此在解析器理论、类型系统、语义学和形式逻辑方面并不缺乏书籍和论文。沿着这条路走下去也会教你如何阅读CS论文，这本身就是一项有价值的技能。

- > Or, if you just really enjoy hacking on and making languages, you can take Lox and turn it into your own plaything. Change the syntax to something that delights your eye. Add missing features or remove ones you don't like. Jam new optimizations in there.
  >
  > Eventually you may get to a point where you have something you think others could use as well. That gets you into the very distinct world of programming language *popularity*. Expect to spend a ton of time writing documentation, example programs, tools, and useful libraries. The field is crowded with languages vying for users. To thrive in that space you'll have to put on your marketing hat and *sell*. Not everyone enjoys that kind of public-facing work, but if you do, it can be incredibly gratifying to see people use your language to express themselves.

  或者，如果你真的喜欢钻研和制造语言，你可以把Lox变成你自己的玩物。把语法改成能让你满意的东西。增加缺失的功能或删除你不喜欢的功能。在其中添加新的优化[^22]。

  最终，你会达到某个境地，有了一些你认为其他人也可以使用的东西。这会带你进入非常独特的编程语言*流行度*的世界。预计你将花费大量的时间来编写文档、示例程序、工具和有用的库。这个领域充斥着很多争夺用户的语言。要想在这个领域取得成功，你将必须带上营销的帽子，进行销售。不是每个人都喜欢这种面对公众的工作，但如果你喜欢，能够看到人们使用你的语言来表达自己，你会感到无比欣慰。

> Or maybe this book has satisfied your craving and you'll stop here. Whichever way you go, or don't go, there is one lesson I hope to lodge in your heart. Like I was, you may have initially been intimidated by programming languages. But in these chapters, you've seen that even really challenging material can be tackled by us mortals if we get our hands dirty and take it a step at a time. If you can handle compilers and interpreters, you can do anything you put your mind to.

或者，也许这本书已经满足了你的需求，你会在这里停下来。无论你走哪条路，或者不走哪条路，我都希望能把这个教训留在你心里。像我一样，你可能一开始就被编程语言吓到了。但在这些章节中，你已经看到，即使是真正具有挑战性的事情，只要亲自动手，一步一步来，我们这些凡人也可以解决。如果你能处理好编译器和解释器，你就可以做到任何你想做的事情。

^9: 我不确定是谁首先提出了这个技巧。我能找到的最早的资料是David Gudeman在1993年发表的论文《在动态类型语言中表示类型信息（Representing Type Information in Dynamically Typed Languages）》。其他人都在引用这篇文章。但是Gudeman自己说这篇论文并不是什么新颖的工作，而是 "收集了大量的民间传说"。
也许发明者已经消失在时间的迷雾中，也许它已经被重新发明了很多次。任何人对IEEE 754进行了足够长时间

的思考，都可能会开始考虑在那些未使用的NaN中加入一些有用的信息。 ^10: 因为符号位一直存在，即使数字是零，这意味着"正零"和"负零"有不同的位表示形式，事实上，IEEE 754确实区分了它们。 ^11: 我不知道是否有CPU真正做到了捕获信号NaN并中止，规范中只是说它们*可以*。^12: 48比特位足以对262,114GB的内存进行寻找。现代操作系统也为每个进程提供了自己的地址空间，所以这应该足够了。 ^13: 规范的作者不喜欢类型双关，因为它使得优化变得更加困难。一个关键的优化技术是对指令进行重新排序，以填充CPU的执行管道。显然，编译器只有在重排序不会产生用户可见的影响时才可以这样做。
指针使得这一点更加困难。如果两个指针指向同一个值，那么通过一个指针进行的写操作和通过另一个指针进行的读操作就不能被重新排序。但是，如果是两个*不同*类型的指针呢？如果这些指针可以指向同一个对象，那么基本上*任意*两个指针都可以成为同一个值的别名。这极大地限制了编译器可以自由地重新排列的代码量。
为了避免这种情况，编译器希望采用**严格别名**——不兼容类型的指针不能指向相同的值。类型双关，从本质上来说，打破了这种假设。 ^14: 如果你发现自己的编译器没有对`memcpy()`进行优化，可以试试这个：

```
double valueToNum(Value value) {
  union {
    uint64_t bits;
    double num;
  } data;
  data.bits = value;
  return data.num;
}
```

^15: 非常肯定，但不是严格保证。据我所知，没有什么可以阻止CPU产生一个NaN值，作为某些操作的结果，而且这些操作的位表示形式会与我们声明的位表示形式相冲突。但在我跨多个架构的测试中，还没有看到这种情况发生。 ^16: 实际上，即使该值是一个Obj指针，我们也*可以*使用最低位来存储类型标签。这是因为Obj指针总是被对齐到8字节边界，因为Obj包含一个64位的字段。这反过来意味着Obj指针的最低三位始终是0。我们可以在其中存储任何我们想要的东西，只是在解引用指针之前要将这些屏蔽掉。
这是另一种被称为**指针标记**的值表示形式优化方案。 ^17: 在涉及到本书中的代码时，我都试图遵循法律条文，所以这一段是值得怀疑的。在优化的时候，你会遇到一个问题，那就是你不仅要突破*规范所规定*的边界，还有突破真正的编译器和芯片所允许的边界。
超出规范之外是有风险的，但在这个无法无天的领域也会有回报。这样做是否值得，取决于你自己。 ^18: 事实上，jlox把NaN相等性搞错了。当你使用`==`来比较基本类型double时，Java做的是正确的，但如果你把这些值包装在Double或Object中，并使用`equals()`来比较它们时，就是错的，而这正是jlox中使用相等性的方式。
^19: 在做剖析工作时，你基本总是想剖析程序的优化后的"发布"构建版本，因为这反映了最终用户体验的性能情况。编译器的优化（如内联）会极大地影响代码中哪些部分是性能热点。手工优化一个调试构建版本，可能会让你去"修复"那些优化编译器本来就会为你解决的问题。
请确保你不会意外地对调试构建版本进行基准测试和优化。我似乎每年都至少要犯一次这样的错误。 [^20]: 这也适用于其它领域。我认为我在编程中所学到的任何一个主题——甚至在编程之外——最终都发现在其它领域中是有用的。我最喜欢软件工程的一个方面正是它对那些兴趣广泛的人的助益。 [^21]: 在这方面，我喜欢Cooper和Torczon的《*编译器工程，Engineering a Compiler*》。Appel的《*现代编译器实现，Modern Compiler Implementation*》一书也广受好评。 [^22]: 本书的文本版权归我所有，但jlox和clox的代码和实现采用了非常宽松的MIT许可。我非常欢迎你使用这些解释器中的任何一个，对它们做任何你想做的事。去吧。
如果你对语言做了重大改动，最好也能改一下名字，主要是为了避免人们对"Lox"这个名字的含义感到困惑。

习题

> Assigning homework on the last day of school seems cruel but if you really want something to do during your summer vacation:

在学校的最后一天布置家庭作业似乎很残酷，但如果你真的想在暑假做点什么的话：

1. > Fire up your profiler, run a couple of benchmarks, and look for other hotspots in the VM. Do you see anything in the runtime that you can improve?

   启动你的剖析器，运行几个基准测试，并查找虚拟机中的其它热点。你在运行时中看到什么可以改进的地方吗？

2. > Many strings in real-world user programs are small, often only a character or two. This is less of a concern in clox because we intern strings, but most VMs don’t. For those that don’t, heap allocating a tiny character array for each of those little strings and then representing the value as a pointer to that array is wasteful. Often, the pointer is larger than the string’s characters. A classic trick is to have a separate value representation for small strings that stores the characters inline in the value.
   >
   > Starting from clox’s original tagged union representation, implement that optimization. Write a couple of relevant benchmarks and see if it helps.

   在现实世界的用户程序中，许多字符串都很小，通常只有一两个字符。这种clox中不太需要考虑，因为我们会驻留字符串，但大树下虚拟机不会这样做。对于那些不这样做的虚拟机来说，为每个小字符串在堆上分配一个很小的字符数组，然后用一个指向该数组的指针来表示该值是很浪费的。通常情况下，这个指针要比字符串的字符大。一个经典的技巧是为小字符串设置一个单独的值表示形式，该形式会将字符内联存储在值中。

   从clox最初的带标签联合体表示形式开始，实现这一优化。写几个相关的基准测试，看看是否有帮助。

3. > Reflect back on your experience with this book. What parts of it worked well for you? What didn’t? Was it easier for you to learn bottom-up or top-down? Did the illustrations help or distract? Did the analogies clarify or confuse?
   >
   > The more you understand your personal learning style, the more effectively you can upload knowledge into your head. You can specifically target material that teaches you the way you learn best.

   回顾一下你在这本书中的经历。哪些部分对你来说很有用？哪些没有？对你来说，自下而上的学习更容易，还是自上而下的学习更简单？插图有帮助还是分散了注意力？类比是澄清了还是混淆了？

   你越了解你的个人学习风格，你就能越有效地将知识输入你的大脑中。你可以有针对性地选择用你最擅长的方式进行教学的材料。

I. WELCOME

> This may be the beginning of a grand adventure. Programming languages encompass a huge space to explore and play in. Plenty of room for your own creations to share with others or just enjoy yourself. Brilliant computer scientists and software engineers have spent entire careers traversing this land without ever reaching the end. If this book is your first entry into the country, welcome.

这也许是一场大冒险的开始。编程语言包含了一个巨大的探索和游戏空间。在其中，你有足够的空间与他人分享自己的创作，或者只是自娱自乐。杰出的计算机科学家和软件工程师穷尽整个职业生涯都在穿越这片土地，却从未到达终点。如果这是你第一次进入这个国度，欢迎你。

> The pages of this book give you a guided tour through some of the world of languages. But before we strap on our hiking boots and venture out, we should familiarize ourselves with the territory. The chapters in this part introduce you to the basic concepts used by programming languages and how they are organized.

本书的内容为你提供了一些语言世界的导览。但是，在我们穿上登山靴开始冒险之前，我们应该先熟悉一下这片土地。本部分的章节将向你介绍编程语言所使用的基本概念以及它们的组织方式。

> We will also get acquainted with Lox, the language we'll spend the rest of the book implementing (twice). Let's go!

我们以后还会熟悉Lox，这门语言我们将用本书的其余部分来实现（两次）。让我们开始吧!

II. A TREE-WALK INTERPRETER

> With this part, we begin jlox, the first of our two interpreters. Programming languages are a huge topic with piles of concepts and terminology to cram into your brain all at once. Programming language theory requires a level of mental rigor that you probably haven't had to summon since your last calculus final. (Fortunately there isn't too much theory in this book.)

在这部分中，我们开始学习jlox，这是我们两个解释器中的第一个。编程语言是一个巨大的话题，其中有大量的概念和术语需要一下子塞进你的大脑。编程语言理论需要一定程度的脑力投入，你可能自上次微积分期末考试后就没这么投入过了。(幸运的是，这本书没有太多的理论。)

> Implementing an interpreter uses a few architectural tricks and design patterns uncommon in other kinds of applications, so we'll be getting used to the engineering side of things too. Given all of that, we'll keep the code we have to write as simple and plain as possible.

实现一个解释器需要一些架构技巧和设计模式，这在其他类型的应用程序中是不常见的，所以我们也要习惯于工程方面的东西。考虑到这些，我们会尽可能地让代码简单明了。

> In less than two thousand lines of clean Java code, we'll build a complete interpreter for Lox that implements every single feature of the language, exactly as we've specified. The first few chapters work front-to-back through the phases of the interpreter—scanning, parsing, and evaluating code. After that, we add language features one at a time, growing a simple calculator into a full-fledged scripting language.

在不到2000行简洁的Java代码中，我们将为Lox构建一个完整的解释器，完全按照我们指定的方式实现该语言的每一个功能。前几章从头到尾介绍解释器的各个阶段——扫描、解析和计算代码。之后，我们逐步添加语言特性，将一个简单的计算器发展成一种成熟的脚本语言。

III.A BYTECODE VIRTUAL MACHINE

> Our Java interpreter, jlox, taught us many of the fundamentals of programming languages, but we still have much to learn. First, if you run any interesting Lox programs in jlox, you'll discover it's achingly slow. The style of interpretation it uses—walking the AST directly—is good enough for some real-world uses, but leaves a lot to be desired for a general-purpose scripting language.

我们的Java解释器jlox教会了我们许多编程语言的基础知识，但我们仍然有许多东西需要学习。首先，如果你在jlox中运行任何Lox程序，你会发现它非常慢。它所使用的解释方式——直接遍历AST，对于某些实际应用来说

已经足够了，但是对于通用脚本语言来说还有很多不足之处。

> Also, we implicitly rely on runtime features of the JVM itself. We take for granted that things like instanceof in Java work somehow. And we never for a second worry about memory management because the JVM's garbage collector takes care of it for us.

另外，我们隐式地依赖于JVM本身的运行时特性。我们想当然地认为像`instanceof`这样的语句在Java中是可以工作的。而且我们从未担心过内存管理，因为JVM的垃圾收集器为我们解决了这个问题。

> When we were focused on high-level concepts, it was fine to gloss over those. But now that we know our way around an interpreter, it's time to dig down to those lower layers and build our own virtual machine from scratch using nothing more than the C standard library . . .

当我们专注于高层次概念时，我们可以忽略这些。但现在我们已经对解释器了如指掌，是时候深入到这些底层，从头开始构建我们自己的虚拟机，只用C语言标准库就可以了......

后记 BACKMATTER

> You've reached the end of the book! There are two pieces of supplementary material you may find helpful:

你已经看完了这本书！有两份补充材料可能对你有所帮助：

> - **Appendix I** contains a complete grammar for Lox, all in one place.
> - **Appendix II** shows the Java classes produced by the AST generator we use for jlox.

附录I Appendix I

> Here is a complete grammar for Lox. The chapters that introduce each part of the language include the grammar rules there, but this collects them all into one place.

这里有一份Lox的完整语法。介绍语言每个部分的章节中都包含对应的语法规则，但这里将它们全部收录在一起了。

> ## A1 . 1 Syntax Grammar

A1.1 语法

> The syntactic grammar is used to parse the linear sequence of tokens into the nested syntax tree structure. It starts with the first rule that matches an entire Lox program (or a single REPL entry).

语法用于将词法标识（token）的线性序列解析为嵌套的语法树结构。它从匹配整个Lox程序（或单条REPL输入）的第一个规则开始。

```
program        → declaration* EOF ;
```

> ### A1 . 1 . 1 Declarations

A1.1.1 声明

> A program is a series of declarations, which are the statements that bind new identifiers or any of the other statement types.

一个程序就是一系列的声明，也就是绑定新标识符或其它statement类型的语句。

```
declaration    → classDecl
               | funDecl
               | varDecl
               | statement ;

classDecl      → "class" IDENTIFIER ( "<" IDENTIFIER )?
                 "{" function* "}" ;
funDecl        → "fun" function ;
varDecl        → "var" IDENTIFIER ( "=" expression )? ";" ;
```

> A1.1.2 Statements

A1.1.2 语句

> The remaining statement rules produce side effects, but do not introduce bindings.

其余的语句规则会产生副作用，但不会引入绑定。

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | returnStmt
               | whileStmt
               | block ;

exprStmt       → expression ";" ;
forStmt        → "for" "(" ( varDecl | exprStmt | ";" )
                           expression? ";"
                           expression? ")" statement ;
ifStmt         → "if" "(" expression ")" statement
                 ( "else" statement )? ;
printStmt      → "print" expression ";" ;
returnStmt     → "return" expression? ";" ;
whileStmt      → "while" "(" expression ")" statement ;
block          → "{" declaration* "}" ;
```

> Note that `block` is a statement rule, but is also used as a nonterminal in a couple of other rules for things like function bodies.

请注意，`block`是一个语句规则，但在其它规则中也作为非终止符使用，用于表示函数体等内容。

> A1.1.3 Expressions

A1.1.3 表达式

> Expressions produce values. Lox has a number of unary and binary operators with different levels of precedence. Some grammars for languages do not directly encode the precedence relationships and specify that elsewhere. Here, we use a separate rule for each precedence level to make it explicit.

表达式会产生值。Lox有许多具有不同优先级的一元或二元运算符。一些语言的语法中没有直接编码优先级关系，而是在其它地方指定。在这里，我们为每个优先级使用单独的规则，使其明确。

```
expression     → assignment ;

assignment     → ( call "." )? IDENTIFIER "=" assignment
               | logic_or ;

logic_or       → logic_and ( "or" logic_and )* ;
logic_and      → equality ( "and" equality )* ;
equality       → comparison ( ( "!=" | "==" ) comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;

unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")" | "." IDENTIFIER )* ;
primary        → "true" | "false" | "nil" | "this"
               | NUMBER | STRING | IDENTIFIER | "(" expression ")"
               | "super" "." IDENTIFIER ;
```

> ### A1.1.4 Utility rules

A1.1.4 实用规则

> In order to keep the above rules a little cleaner, some of the grammar is split out into a few reused helper rules.

为了使上面的规则更简洁一点，一些语法被拆分为几个重复使用的辅助规则。

```
function       → IDENTIFIER "(" parameters? ")" block ;
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;
arguments      → expression ( "," expression )* ;
```

> ## A1.2 Lexical Grammar

A1.2 词法

> The lexical grammar is used by the scanner to group characters into tokens. Where the syntax is context free, the lexical grammar is regular—note that there are no recursive rules.

词法被扫描器用来将字符分组为词法标识（token）。语法是上下文无关的，词法是正则的——注意这里没有递归规则。

```
NUMBER         → DIGIT+ ( "." DIGIT+ )? ;
STRING         → "\"" <any char except "\"">* "\"" ;
IDENTIFIER     → ALPHA ( ALPHA | DIGIT )* ;
ALPHA          → "a" ... "z" | "A" ... "Z" | "_" ;
DIGIT          → "0" ... "9" ;
```

附录II Appendix II

> For your edification, here is the code produced by the little script we built to automate generating the syntax tree classes for jlox.

为了方便你们学习，下面是我们为自动生成jlox语法树类而构建的小脚本所产生的代码。

A2.1 Expressions

A2.1 表达式

> Expressions are the first syntax tree nodes we see, introduced in "Representing Code". The main Expr class defines the visitor interface used to dispatch against the specific expression types, and contains the other expression subclasses as nested classes.

表达式是我们看到的第一个语法树节点，在"表示代码"中介绍过。主要的Expr类定义了用于针对特定表达式类型进行调度的访问者接口，并将其它表达式子类作为嵌套类包含其中。

lox/Expr.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

abstract class Expr {
  interface Visitor<R> {
    R visitAssignExpr(Assign expr);
    R visitBinaryExpr(Binary expr);
    R visitCallExpr(Call expr);
    R visitGetExpr(Get expr);
    R visitGroupingExpr(Grouping expr);
    R visitLiteralExpr(Literal expr);
    R visitLogicalExpr(Logical expr);
    R visitSetExpr(Set expr);
    R visitSuperExpr(Super expr);
    R visitThisExpr(This expr);
    R visitUnaryExpr(Unary expr);
    R visitVariableExpr(Variable expr);
  }
  // Nested Expr classes here...

  abstract <R> R accept(Visitor<R> visitor);
}
```

A2.1.1 Assign expression

> Variable assignment is introduced in “Statements and State”.

变量赋值在“表达式与状态”中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Assign extends Expr {
    Assign(Token name, Expr value) {
      this.name = name;
      this.value = value;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitAssignExpr(this);
    }

    final Token name;
    final Expr value;
  }
```

> A2.1.2 Binary expression

A2.1.2 Binary表达式

> Binary operators are introduced in “Representing Code”.

二元运算符在“表示代码”中介绍过。

lox/Expr.java，嵌套在类Expr中：

```
  static class Binary extends Expr {
    Binary(Expr left, Token operator, Expr right) {
      this.left = left;
      this.operator = operator;
      this.right = right;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitBinaryExpr(this);
    }
    final Expr left;
    final Token operator;
    final Expr right;
  }
```

A2.1.3 Call expression

> Function call expressions are introduced in “Functions”.

函数调用语句在“函数”中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Call extends Expr {
    Call(Expr callee, Token paren, List<Expr> arguments) {
      this.callee = callee;
      this.paren = paren;
      this.arguments = arguments;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitCallExpr(this);
    }

    final Expr callee;
    final Token paren;
    final List<Expr> arguments;
  }
```

A2.1.4 Get expression

> Property access, or “get” expressions are introduced in “Classes”.

属性访问，或者说“get”表达式，在“类”中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Get extends Expr {
    Get(Expr object, Token name) {
      this.object = object;
      this.name = name;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitGetExpr(this);
    }
    final Expr object;
    final Token name;
  }
```

A2.1.5 Grouping expression

> Using parentheses to group expressions is introduced in "Representing Code".

使用括号进行分组的表达式在"表示代码"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Grouping extends Expr {
    Grouping(Expr expression) {
      this.expression = expression;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitGroupingExpr(this);
    }

    final Expr expression;
  }
```

A2.1.6 Literal expression

> Literal value expressions are introduced in "Representing Code".

字面量值表达式在"表示代码"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Literal extends Expr {
    Literal(Object value) {
      this.value = value;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitLiteralExpr(this);
    }

    final Object value;
  }
```

A2.1.7 Logical expression

> The logical and and or operators are introduced in "Control Flow".

逻辑运算符and和or在"控制流"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Logical extends Expr {
    Logical(Expr left, Token operator, Expr right) {
      this.left = left;
      this.operator = operator;
      this.right = right;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitLogicalExpr(this);
    }

    final Expr left;
    final Token operator;
    final Expr right;
  }
```

A2.1.8 Set expression

> Property assignment, or "set" expressions are introduced in "Classes".

属性赋值，或者叫"set"表达式，在"类"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Set extends Expr {
    Set(Expr object, Token name, Expr value) {
      this.object = object;
      this.name = name;
      this.value = value;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitSetExpr(this);
    }

    final Expr object;
    final Token name;
    final Expr value;
  }
```

A2.1.9 Super expression

> The `super` expression is introduced in "Inheritance".

`super`表达式在"继承"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Super extends Expr {
    Super(Token keyword, Token method) {
      this.keyword = keyword;
      this.method = method;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitSuperExpr(this);
    }

    final Token keyword;
    final Token method;
  }
```

A2.1.10 This expression

> The `this` expression is introduced in "Classes".

`this`表达式在"类"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class This extends Expr {
    This(Token keyword) {
      this.keyword = keyword;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitThisExpr(this);
    }

    final Token keyword;
  }
```

A2.1.11 Unary expression

> Unary operators are introduced in "Representing Code".

一元运算符在"表示代码"中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Unary extends Expr {
    Unary(Token operator, Expr right) {
      this.operator = operator;
      this.right = right;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitUnaryExpr(this);
    }

    final Token operator;
    final Expr right;
  }
```

A2.1.12 Variable expression

> Variable access expressions are introduced in “Statements and State”.

变量访问表达式在“语句和状态”中介绍过。

lox/Expr.java，嵌套在Expr类中：

```
  static class Variable extends Expr {
    Variable(Token name) {
      this.name = name;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitVariableExpr(this);
    }

    final Token name;
  }
```

A2.2 Statements

A2.2 语句

> Statements form a second hierarchy of syntax tree nodes independent of expressions. We add the first couple of them in “Statements and State”.

语句形成了独立于表达式的第二个语法树节点层次。我们在“声明和状态”中添加了前几个。

lox/Stmt.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

abstract class Stmt {
  interface Visitor<R> {
    R visitBlockStmt(Block stmt);
    R visitClassStmt(Class stmt);
    R visitExpressionStmt(Expression stmt);
    R visitFunctionStmt(Function stmt);
    R visitIfStmt(If stmt);
    R visitPrintStmt(Print stmt);
    R visitReturnStmt(Return stmt);
    R visitVarStmt(Var stmt);
    R visitWhileStmt(While stmt);
  }

  // Nested Stmt classes here...

  abstract <R> R accept(Visitor<R> visitor);
}
```

A2.2.1 Block statement

> The curly-braced block statement that defines a local scope is introduced in “Statements and State”.

在“语句和状态”中介绍过的花括号块语句，可以定义一个局部作用域。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Block extends Stmt {
    Block(List<Stmt> statements) {
      this.statements = statements;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitBlockStmt(this);
    }

    final List<Stmt> statements;
  }
```

A2.2.2 Class statement

> Class declarations are introduced in, unsurprisingly, “Classes”.

类声明是在“类”中介绍的，毫不意外。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Class extends Stmt {
    Class(Token name,
          Expr.Variable superclass,
          List<Stmt.Function> methods) {
      this.name = name;
      this.superclass = superclass;
      this.methods = methods;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitClassStmt(this);
    }

    final Token name;
    final Expr.Variable superclass;
    final List<Stmt.Function> methods;
  }
```

A2.2.3 Expression statement

> The expression statement is introduced in "Statements and State".

表达式语句在"语句和状态"中介绍过。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Expression extends Stmt {
    Expression(Expr expression) {
      this.expression = expression;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitExpressionStmt(this);
    }

    final Expr expression;
  }
```

A2.2.4 Function statement

> Function declarations are introduced in, you guessed it, "Functions".

函数声明是在"函数"中介绍的。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Function extends Stmt {
    Function(Token name, List<Token> params, List<Stmt> body) {
      this.name = name;
      this.params = params;
      this.body = body;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitFunctionStmt(this);
    }

    final Token name;
    final List<Token> params;
    final List<Stmt> body;
  }
```

A2.2.5 If statement

> The `if` statement is introduced in "Control Flow".

`if`语句在"控制流"中介绍过。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class If extends Stmt {
    If(Expr condition, Stmt thenBranch, Stmt elseBranch) {
      this.condition = condition;
      this.thenBranch = thenBranch;
      this.elseBranch = elseBranch;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitIfStmt(this);
    }

    final Expr condition;
    final Stmt thenBranch;
    final Stmt elseBranch;
  }
```

A2.2.6 Print statement

> The `print` statement is introduced in "Statements and State".

`print`语句在"语句和状态"中介绍过。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Print extends Stmt {
    Print(Expr expression) {
      this.expression = expression;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitPrintStmt(this);
    }

    final Expr expression;
  }
```

A2.2.7 Return statement

> You need a function to return from, so `return` statements are introduced in "Functions".

你需要一个函数才能返回，所以`return`语句是在"函数"中介绍的。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Return extends Stmt {
    Return(Token keyword, Expr value) {
      this.keyword = keyword;
      this.value = value;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitReturnStmt(this);
    }

    final Token keyword;
    final Expr value;
  }
```

A2.2.8 Variable statement

> Variable declarations are introduced in "Statements and State".

变量声明在"语句和状态"中介绍过。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class Var extends Stmt {
    Var(Token name, Expr initializer) {
      this.name = name;
      this.initializer = initializer;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitVarStmt(this);
    }

    final Token name;
    final Expr initializer;
  }
```

A2.2.9 While statement

> The `while` statement is introduced in “Control Flow”.

`while`语句在“控制流”中介绍过。

lox/Stmt.java，嵌套在Stmt类中：

```
  static class While extends Stmt {
    While(Expr condition, Stmt body) {
      this.condition = condition;
      this.body = body;
    }

    @Override
    <R> R accept(Visitor<R> visitor) {
      return visitor.visitWhileStmt(this);
    }

    final Expr condition;
    final Stmt body;
  }
```