NPL 技术体系

文件编号：	版本：V1.0	日期：
编制：陈章统 李文龙 万昌龙 何睿智 彭照志 张一	审核：陈细杰	批准：

DreamTech

修订履历

修改时间	修改概要	修改后版本	修改人	批准人
2017-05-27	初稿	V1.0		

Idream Tech

目录

IdreamTech

IdreamTech

dreamTech

IdreamTech

dreamTech

IdreamTech

1 概述

1.1 NPL 简介

NPL 是 Neural Parallel Language 的缩写，即 NPL 是神经元并行开发语言。NPL 有以下几个特点：

1. NPL 是一种通用开源的语言，其语法 100%与 Lua 兼容；
2. NPL 提供基本的编写 3D/2D/Web 应用程序的功能；
3. NPL 提供丰富的 C/C++ API 并且极多的库被写入 NPL 中；
4. NPL 是一个解决先进的交互式 GUI、复杂的 opengl/DirectX 3D 图像、可扩展的网站服务器及分布式软件框架的单独的语言。它是跨平台的、高性能的、可扩展的且可调试的；
5. NPL 是一个最初被设计像大脑一样运行的语言系统。节点和连接无处不在，线程和网络逻辑被开发者隐藏起来；
6. NPL 可以将抢占式用户模式和非抢占式用户模式混合在一起。这在同一个动态的、弱类型的语言中提供了像 Erlang 一样的并行模式以及像 Java/C++一样的速度。

1.2 Hello World

首先需要搭建开发的环境。详情参见：https://github.com/LiXizhi/NPLRuntime/wiki/InstallGuide

Hello World 程序：

在当前目录地址中编写一个 helloworld.lua 文件，内容为：

```
print(“hello world”)
```

然后在终端命令行中运行 npl helloworld.lua。你将会在当前目录地址中看到一个文件：./log.txt，该文件包含”hello world”文本。

1. 在 Linux 命令行中：

在 Linux 中，如果直接下载源代码的话，NPL 被下载在/usr/local/bin/npl 中。因

dreamTech

此，你可以用以下方式调用独立的 NPL 应用：

```
#!/usr/local/bin/npl
print(“hello world”)
ParaGlobal.Exit(0)
```

然后以下命令可以运行程序：

```
./helloworld.npl
```

注意：只有*.npl 和*.lua 文件的扩展可以使用。

2. 在 Window 命令行中：

在 Windows 中，npl 执行文件需要添加到搜索路径。如果 npl 命令找不到，你就需要将./NPLRuntime/win/bin 添加到搜索路径%path%中，或者你可以直接使用全部的路径去运行你的脚本，例如：

```
__YourSDKpath__/NPLRuntime/win/bin/npl helloworld.lua
```

场景背后：

ParacraftSDK 的初始文件夹包含 Paracraft 的安装程序。在初始文件夹中运行 Paracraft.exe 将会随着 NPLRuntime 一起下载安装最新的 Parascraft 的版本。./NPLRuntime/install.bat 仅仅从初始文件夹复制了 NPL runtime 的相关文件到./NPLRuntime/win/bin 和./NPLRuntime/win/packages

./NPLRuntime/win/packages 包含预编译的 NPL 库的源代码。

在配置你的应用时，你需要同时使用 bin 和 packages 文件夹去开发你自己的脚本及源文件。例如：

```
./bin/          :npl runtime 32bits
./bin64/        :npl runtime 64bits
./packages/     :npl 预编译源文件
./script/       :你自己的脚本
./              :你的应用的（根）目录地址
```

dreamTech

1.3 基础概念

1.3.1 bootstrapper

Bootstrapping:

Bootstrapping 是指定加载第一个脚本当开启 NPL Runtime 时，它也会每 0.5 秒启动一次。通过命令行有很多种方法做到 bootstrapping。

推荐的做法是明确的指定参数 bootstrapper=filename。例如：

npl bootstrapper=”script/gameinterface.lua”

如果没有指定文件名，script/gameinterface.lua 将被使用。更简单方式是用第一个参数来指定文件名，例如：

npl helloworld.lua

内部的启动命令如下：

1. 当 NPL Runtime 启动时，它首先读取所有命令行参数如：名称，数值，然后将它们存入内部的数据结构方便之后使用。
2. NPL 识别多个预编译的命令名称，其中就有 bootstrapper。它具体指定了主循环文件。
3. NPL 自身会初始化，这个过程话费几秒钟的时间，包括读取配置、初始化图像，绑定脚本 API 等。
4. 当所有的初始化完成后，多数的 NPL 的核心 API 对于脚本接口是可以被获取的并且 NPL 加载主循环脚本是被 bootstraper 具体指定的而且每 0.5 秒调用一次它的活跃的函数。
5. 之后发生的任何操作都取决于写主循环脚本的程序。

另外，对于脚本文件来说，XML 文件也可以用来坐 bootstrapper。XML 文件内容样式如下：

```
<?xml version=”1.0”?>
```

dreamTech

<MainGameloop>(gl)script/apps/Taurus/main_loop.lua</MainGameLoop>

我们可以用如下命令运行：

npl script/apps/Taurus/bootstrapper.xml

理解文件路径：

当用 NPL 编程时，代码会根据文件名参考其他脚本或者有用的文件。对一个文件的搜索顺序如下：

1. 检查文件是否存在当前开发的目录地址中（在释放时间，这个和工作目录一致）。
2. 检查文件是否存在于搜索路径（默认当前的工作目录）。
3. 检查文件在加载队列中是否存在于任何已加载的档案文件（主要为 XXX.PKG 或世界/插件的 ZIP 文件）
4. 如果文件为 NPL 脚本文件，在之前的地方或者队列中的./bin/*filename*.o 中寻找预编译的二进制文件。
5. 检查硬盘文件是否存在于所有 npl_package 的搜索路径中。
6. 如果以上所有情况都没有找到，我们将报告文件没有找到。注意，一些 API 需要你使用特定的搜索命令，在一些地方暂时不可用，或者包含确切的目录像或者例如当前可写的目录。

1.3.2 NPL 命令

内置 NPL 命令行

本地参数：

以下命令行是内置且可执行于 NPL 的：

- bootstrapper="myapp/main.lua": set bootstrapper file
- [-d|D] [filename]: -d to run in background (daemon mode), -D to force run in fore ground.

dreamTech

- dev="C:/MyDev/": set dev directory, if empty or "/", it means the current working directory.
- servermode="true": disable 3D rendering and force running in server mode.
- logfile="log2016.5.20.txt": change default log.txt location
- single="true": force only one instance of the executable can be running in the OS.
- loadpackage="folder1,folder1,...": commar separated list of packages to load before bootstrapping. Such as loadpackage="npl_packages/paracraft/" npl_packages/main is always loaded by default.

NPL 系统模型参数：

以下的参数是被 NPL 系统模型处理的：

- debug="main" : Enable IPC debugging to the main NPL thread. No performance penalties.
- resolution="1020 680": force window size in case of client application

Paracraft 模型参数：

以下参数是被 Paracraft Package 处理的：

- world="worlds/DesignHouse/myworld" : Load a given world at start up.
- mc="true": force paracraft (in case of running from MagicHaqi directory)
- httpdebug="true": enable npl http debugger console on

  http://127.0.0.1:8099

从 NPL 脚本中获取命令行参数：

用以下方式可以获取已给的命令行参数：

//举个例子，如果你通过’ npl param1=”abc”‘ 开启 npl 进程，param1 将被赋值为”abc”.

```
local param1 = ParaEngine.GetAppCommandLineByParam("param1",
"default_value");
echo(ParaEngine.GetAppCommandLineByParam("bootstrapper", ""))
```

用以下命令可以获取全部命令行字符串：

```
local cmdline = ParaEngine.GetAppCommandLine();
echo(cmdline)
```

工作目录：

工作目录是读/写文件默认的根目录。默认情况下，NPLRuntime 不更改可执行文件的工作目录，但是有一种情况除外。

如果可执行文件的文件夹或其付上层文件夹包含 ParaEngine.sig 文件（文件内容不重要），那么 NPLRuntime 将会重新设置工作目录到包含 ParaEngine.sig 文件的文件夹中。

一些 NPL 应用，例如 Paracraft 的 redist 文件包含 ParaEngine.sig 文件，这意味着你可以从任何文件夹开启可执行的 paracraft，但是工作目录经常重新设置到 redist 文件夹。这种行为影响到最先的 IO 运行，包括默认的日志位置。

1.3.3 NPLload

NPL 加载文件：

在 NPL 代码中，你可以使用例如 NPL.load（“script/ide/commonlib.lua”)的命令去加载一个文件。为了可以运行，所有的 NPL 代码需要被编译。NPL.load 会自动帮你做如下的操作：

- 在 NPL zip 压缩文件夹或根据定义在 npl_packages 中的 NPL 搜索路径寻找正确的文件并且如果存在的话，自动使用预编译的二进制版本（*.o）
    - 支持相关路径，但是只推荐在私有文件上使用，如：NPL.load（“./A.lua”)或 NPL.load（“../../B.lua”）
    - 当使用相关路径时，文件的扩展将被忽略而*.npl 和*.lua 被搜索，例如：NPL.load（“./A”)或 NPL.load（“../../B”）
- 自动解决递归问题（例如文件 A 加载文件 B，同时文件 B 加载文件 A）
- 编译脚本代码如果还没有编译的话
    - 如果文件扩展是*.npl，NPL meta 编译器将会被激活。
- 代码第一次被编译时，代码块也在保护模式下利用 pcall 被执行。在多数情况下，这意味着添加新代码到全局或输出表中，因此在之后可以使用它。
- 它也可以被用来加载 C++/Mono C#/NPL 文件夹。
- 如果存在的话，返回输出文件模型或 nil（空）。如果模型找不到则返回 false。

代码注入模型：

因为在 NPL/Lua 中，表和函数是第一个类对象，我们在应用的生命周期中使用一个灵活的代码注入模型去管理所有动态的加载代码。

为了注入新的代码，我们使用像 commonlib.gettable,commonlib.inherit 之类的方法，详细请看面向对象的介绍。开发者需要确保使用注入的代码有特别的名字（例如 CompanyName.AppName.ModuleName）。换句话说就是不要污染了全局的表。

要注入新代码，用户可以调用 NPL.load 和 commonlib.gettable 在文件范围内去创建

dreamTech

一个局部存根变量。注意，由于是 NPL.load 命令，存根可能在确切代码注入前创建。

例如，像下面一样，你在 script/MyApp/Myclass.lua 中有一个类文件：

```
local MyClass = commonlib.inherit(nil,
commonlib.gettable("MyApp.MyClass"));

MyClass.default_param = 1;

-- this is the constructor function.
function MyClass:ctor()
   self.map = {};
end

function MyClass:init(param1)
   self.map.param1 = param1;
   return self;
end

function MyClass:Clone()
   return MyClass:new():init(self:GetParam());
end

function MyClass:GetParam()
   return self.map.param1;
end
```

为了使用以上的类，我们可以使用：

```
NPL.load("(gl)script/MyApp/MyClass.lua")
local MyClass = commonlib.gettable("MyApp.MyClass");

local UserClass = commonlib.inherit(nil,
commonlib.gettable("MyApp.UserClass"));

function UserClass:ctor()
   self.map = MyClass:new();  -- use another class
end
```

基于文件的模型：

定义一个基于文件的模型有三种方法。假设你有一个叫做 A.npl 的文件，你想从中输出一些对象。

- 一种方法是再文件加载时调用 NPL.export。

```
-- this is A.npl file
local A = {};
NPL.export(A);

function A.print(s)
  echo(s);
end
```

以下是一种更好的方法，使用了周期性的依赖关系。推荐使用下面这种方法。

```
 -- this is A.npl file
 local A = NPL.export();
 function A.print(s)
     echo(s);
 end
```

- 另一种方法是简单的返回和最近文件的在最后一行代码有关的对象。这也是一种兼容 lua 的方式。当有周期性依赖关系时，这种方法不适用。

```
-- this is A.npl file
local A = {};
return A;
```

- 最后，有一种先进的手动的方法来简单的添加到_file_mod_表中。

```
-- this is A.npl file
local A = {};
_file_mod_[NPL.filename():gsub("([^/]*$)", "").."A.npl"] = A;
```

正如你看到的，基于文件的模型简单的自动的储存从模型全部文件名到隐藏在全局表中的输出对象的映射。如果文件名或文件地址改变，映射键值也改变。这就是为什么你可以加载一个模型的多个版本并且让用户在已给的源文件中选择使用哪一个。

周期性模型参考：

当你写依赖于每个文件的文件模型时，存在周期性依赖。一个应对方法是更改默认输出对象，它往往是一个空表，而不是创建一个本地表。例如，我们有两个文件 A 和 B 相互

引用：

```
-- A.lua
local B = NPL.load("./B.lua")
local A = NPL.export();
function A.print(s)
  B.print(s)
end
```

```
-- B.lua
local A = NPL.load("./A.lua")
local B = NPL.export();
function B.print(s)
  echo(s..type(A));
end
```

（NPL.export()是 NPL 中文件模型系统的核心，它会返回默认的空表来表示当前文件。这和 commonlib.gettable()解决周期性依赖使用的是一样的方法。）

面向对象类的继承模型：

下面的例子介绍了两个表继承及周期性依赖的例子：

```
-- base_class.lua
local derived_class = NPL.load("./derived_class.lua")
local base_class = commonlib.inherit(nil, NPL.export());
function base_class:ctor()
   derived_class:cyclic_dependency_test();
end
```

```
-- derived_class.lua
local base_class = NPL.load("./base_class.lua")
local B = commonlib.inherit(base_class, NPL.export());
function B:ctor()
end
function B:cyclic_dependency_test()
end
```

使用模型名称加载基于文件的模型：

通过调用 NPL,load(modname)可以导入一个基于文件的模型。NPL.load 是一个通用函数，它不仅可以加载标准源文件还可以加载基于文件的模型并且返回输出的模型对象。

（一个模型名（modname）与文件名是不一样的，一个字符串可以当作一个模型名当且仅当它不包含文件扩展或以“./”及”../”开头，如 NPL.load(“sample_mod”)) NPL.load(modename)将会自动寻找模型的源文件通过按顺序尝试接下来的地址直到找到一个。在 NPL.load 使用模型名称比使用清楚的文件名效率低，因为它在每次调用时都需要搜索，所以请只在文件加载时使用它，例如在文件的开头。

- 如果代码从 *npl_packages/[parentModDir]/*调用 NPL.load
    - npl_packages/[parentModDir]/npl_mod/sample_mod/sample_mod.npl
    - npl_packages/[parentModDir]/npl_packages/sample_mod/npl_mod/sample_mod/sample_mod.npl
- npl_mod/sample_mod/sample_mod.npl
- npl_packages/sample_mod/npl_mod/sample_mod/sample_mod.npl

第二个例子：NPL,load(“sample_mod.xxx.yyy”)对于*.npl 和*.lua 文件将会测试下面的文件地址：

- 如果调用的代码是来自 npl_packages/[parentModDir]/
    - npl_packages/[parentModDir]/npl_mod/sample_mod/xxx/yyy.npl
    - npl_package/[parentModDir]/npl_packages/sample_mod/npl_mod/xxx/yyy.npl
- npl_mod/sample_mod/xxx/yyy.npl
- npl_packages/sample_mod/npl_mod/sample_mod/xxx/yyy.npl

正如你所见的，npl_packages/和 npl_mod 是两个特殊的文件夹，它们是用来搜索什么时间英国加载基础模型的。在搜索全局的模型前，在 npl_packages/xxx/folder 中的代码经常使用本地 npl_mod 和本地 npl_packages，这允许同一个 npl_mod 的多个版本在不同的 npl_package 中同时存在。这取决于开发者如何决定去封装和发布他们的应用或模型。

dreamTech

加载文件夹：

使用以/.结尾的文件名调用 NPL.load 也是可以的。默认加载文件夹只是在很多地址中寻找到那个文件夹并将它加入全局搜索路径。然而，当文件夹包含一个叫做 package.npl 的文件时有例外。

例如，假设 npl_mod/sample_mod/package.npl 是像下面代码一样：

```
-- example of NPL.load folder with package folder configuration file
{
        -- do not add search path when loading the containing folder via
NPL.load. default to true.
        searchpath = false,
        -- bootstrapper = "",
        -- main script
        main = "fileA.lua",
}
```

同时 npl_mod/sample_mod/fileA.lua 是这样的代码：

```
local fileA = NPL.export();
function fileA:print()
   echo("fileA")
end
```

那么我们可以用以下方式加载文件夹：

```
local fileA = NPL.load("npl_mod/sample_mod/");
echo(fileA:print())
```

文件夹不会被添加到搜索路径因为 searchpath=false、并且从 fileA 中会返回输出对象。

加载文件夹是一个 NPL package 的功能。

什么时候使用 require:

require 是加载基于文件的模型的 Lua 方法。NPL 与这种函数挂钩，经常在下达命令前使用相同的输入调用 NPL.load。更特殊的是，NPL 往 lua 的 package.loader 中注入一个自定义加载器。所以调用 require 是几乎与调用 NPL.load 是相同的。到目前为止，多种代码版本无法同时存在。

不要使用 require:

因为 NPL.load 现在与原始的 lua 的需求函数挂钩，实际中这一部分是不确定的，但是我们依然不推荐在你自己的代码中使用 require，除非你在使用第三方 lua 库。理由很简单，原始的 require 在 NPL.load 中是反向兼容的，但是原始的 require 不支持 NPL.load 提供的特点。在你自己的代码中使用 require 会使其他 lua 开发者迷惑。因此 NPL 开发者写的代码通常会使用 NPL.load.

只有在加载 lua 中 C 语言插件时才会使用 require，不要将它是用在你自己的 NPL 脚本中。require 在团体中速率低。它原始的实现时相同的，但是在平面方式使用了相似的全局隐藏表，这是个你从不知道的逻辑。而且它至今都不支持周期性依赖。

此外，一些应用需要创建沙盒环境来隔离代码的执行（代码注入到特定的全局表中），使用 require 的话无法控制代码注入。

不同的 NPL 应用有自己的沙盒环境，其中有它们自己专用的全局表，例如在 NPL 网络服务器 APP 的所有*.page 文件中每一个 URL 请求使用一个分离的全局表。

最后，相比于 NPL，require 运行慢且使用不同的搜索路径。同时 NPL.load 快且在 zip 文件和在 npl_packages 定义的搜索路径中都接受预编译，并且在所有平台可以保持一致。

1.3.4 Activation File

NPL 在使用 NPL.activate 函数时可以和 NPL 脚本远程对话。这是一个强大的函数且在 C++和单插件中都可以使用。

基础语法：

像在神经网络中一样，所有交流都是异步且单向的，没有回调。尽管函数返回一个整数值。

```
NPL.activate(url, {any_pure_data_table_here, })
```

- @param url: 一种 NPL 文件名实例的一个全局的独一无二的名字。一个 NPL 文件名的字符串形式如下：[(sRuntimeStateName|gl)][sNID:]sRelativePath[]下面的列表是所有的有效的文件名的结合：

  - user001@paraengine.com:script/hello.lua --这是一个在默认游戏线程 user001 中的文件
  - (world1)server001@paraengine.com:script/hello.lua --这是 server001 中在线程 world1 中的一个文件
  - (worker1)script/hello.lua --这是在 worker1 线程中的一个本地文件
  - (gl)script/hello.lua --这是在当前 runtime state 的线程中的一个本地文件
  - script/hello.lua --这是当前线程中的一个文件。对于一个单线程应用来说这往往够用了。
  - (worker1)NPLRouter.dll --激活一个 C++文件。注意，在 Windows 中，这是寻找 NPLRonter.dll.在 Linux 中这是寻找./libNPLRouter.so
  - (worker1)DBServer.dll/DBServer.DBServer.cs --对于 C#文件，类必须在 CS 文件中定义有一个静态的激活函数
- @param msg: 这是一个纯数据表块，将被传递至目标文件。

1.3.5 Neuron File

只有与激活函数关联的文件可以被激活。但在 NPL/C++/C#插件中是不同的：

- 在 NPL 中，内置的 NPL.this 函数可以被更灵活的使用.

```
local function activate()
   -- input is passed in a global "msg" variable
   echo(msg);
end
NPL.this(activate);
```

- msg 被传入所有文件都可见的全局的 msg 变量，并且 msg 变量将会持续存在知道线程接收到下一个激活消息
- 在 NPL 的 C++插件中，你需要定义一个 C 函数。请在 ParacraftSDK 的样例文件中查看详细
- 在 NPL 的单一 C#插件中，用静态的 activate 函数简单地定义一个类。

dreamTech

输入消息，msg.nid 和 msg.tid

在上述的代码中，msg 包含从发送方接受的消息加上发送者的 source id。对于未经身份验证的发送方，source id 会存在 msg.tid 中，msg.tid 是一个自动生成的数字字符串如“~1”。接收方可以一直使用这个暂时的 id：msg.tid 返回消息，例如：

```
local function activate()
   -- input is passed in a global "msg" variable
   NPL.activate(format("%s:some_reply_file.lua", msg.tid or msg.nid),
{"replied"});
end
NPL.this(activate);
```

接受方也可以通过调用 NPL.accept(msg.tid, nid_name)来重命名这个暂时的 msg.id，所以如果下次接收方从同一个发送方获得了一个消息（例如相同的 TCP 连接），msg.nid 会包含最后分配的名称同时 msg.tid 不再存在。我们通常使用 NPL.accept 去区分经过身份验证和未经身份验证的发送方，并且尽早的通过调用 NPL.reject(msg.tid)来拒绝未经身份验证的消息来节省 CPU 的循环。例如：

```
local function activate()
   -- input is passed in a global "msg" variable
   if(msg.tid) then
      -- unauthenticated? reject as early as possible or accept it.
      if(msg.password=="123") then
         NPL.accept(msg.tid, msg.username or "default_user");
      else
         NPL.reject(msg.tid);
      end
   elseif(msg.nid) then
      -- only respond to authenticated messages.
      NPL.activate(format("%s:some_reply_file.lua", msg.nid), {"replied"});
   end
end
NPL.this(activate);
```

注意，msg.tid 或 msg.nid 经常连接到一个单独的低层的 TCP 连接，因此它们的名字被分享到所有的进程中的神经元文件。例如：你在一个神经元文件中接收了，其他所有神经元文件接收到的形式都为 msg.nid

Neuron File 可见性：

因为安全性的原因，所有神经元文件可以在同一进程中被其他文件激活。这包括在同一进程的其他线程中的脚本。

为了将脚本暴露给远程电脑，有两件事需要做：

- 第一个是通过监听一个 IP 地址和端口来开启 NPL 服务端，NPL 对所有的对话都使用 TCP 协议
- 第二个是告诉 NPL runtime 给出的文件是公共神经元文件

例如：

```
NPL.StartNetServer("0.0.0.0", 8080);
NPL.AddPublicFile(filename, id);
```

其中“0.0.0.0”代表所有的 IP 地址，也可以使用“127.0.0.1”，“localhost”或其他任何 IP 地址；“8080”是端口数字。选择你想用的就可以。

NPL.AddPublicFile 第二个参数是整数，它是代表长文件名保存带宽而被传输的。所以如果你添加多个公共文件的话它必须是独一无二的。

注意，文件名必须和工作目录相对应，例如：NPL.AddPublicFile(“script/test/test.lua”, 1)。绝对路径在当时是不支持的。

激活远程文件:

当一个 NPL runtime 服务端暴露一个公共文件，其他客户端的 NPL runtime 可以使用 NPL.activate 函数来激活它。注意，一个 NPL runtime 既可以是服务端也可以是客户端。发起连接的一般称它为客户端。为了激活服务端，单纯的客户端也叫做 NPL.StartNetServer。但是它可以具体指明 port=”0”来表明它不会监听进来的连接。

然而，在客户端，我们需要使用 NPL.AddNPLRuntime 去分配一个本地名称到远程的服务端，所以我们可以在之后所有的 NPL.activate 调用中通过名称查到这个服务端。

```
NPL.AddNPLRuntimeAddress({host = "127.0.0.1", port = "8099", nid =
"server1"})
```

我们通常在初始化的时候进行一次这种操作。在那之后我们可以在远程服务端上激活公共文件，如：

```
NPL.activate("server1:helloworld.lua", {})
```

dreamTech

注意，nid 具体指定的名字是随机的并且仅仅用在客户端电脑去查找另一个电脑。换句话说，不同客户端可以用不同的名字命名同一台远程电脑。

信息传递的保障：

请注意，一个电脑第一次激活远程文件时，一个 TCP 连接会自动建立，但是第一个消息未被发出。这是因为 NPL.activate()时异步的，它必须在建立连接前返回一个值。通常返回 0 当消息通过一个已存在的路径发送，并且要是非 0 的话第一条消息被发送到了远程系统。

如果没有已经建立的路径（例如没有 TCP 连接），NPL 将会立马尝试建立连接。然而，需要注意的是，消息返回非 0 是没有被传递的，尽管 NPL 立刻在之后成功的建立了一个路径到远程系统。因此，重新再激活直到 NPL.activate 返回 0 是程序员的工作.这保证了一个返回值为 0 的消息至少再在 NPL runtime 的角度发送出去了。

同样的机制可以被用来恢复断开的连接。

要写一个可容忍错误的消息传递代码，请考虑以下方法：

- 当一个 NPL runtime 进程开启，使用 NPL.activate ping 远程的进程直到返回 0 来建立 TCP 连接。这保证了跨这两个系统的所有之后的 NPL.activate 可以被投递。
- 发现断开的连接：
    - 方法一：使用一个计时器去 ping 或监听断开连接的系统事件并重新连接以防连接丢失。但是，单独的消息在期间可能会丢失。
    - 方法二：使用一个封装函数去调用 NPL.activate，函数会检查返回值。如果它非 0，无论重新连接超时还是将消息放到一个等待的队列以防连接被立即覆盖和重新发送队列消息。

当 NPL.activate 返回非 0 值时，我们将它交给程序员去解决所有的情况，因为不同的事物逻辑可能会使用不同的方法。

dreamTech

服务端/客户端 app 例子：

要运行示例，调用下列命令：

```
npl "script/test/network/SimpleClientServer.lua" server="true"
npl "script/test/network/SimpleClientServer.lua" client="true"
```

这个示例的源代码同时也在 ParaCraftSDK/examples 文件夹。

文件名：script/test/network/SimpleClientServer.lua

```
--[[
Author: Li,Xizhi
Date: 2009-6-29
Desc: start one server, and at least one client.
-----------------------------------------------
npl "script/test/network/SimpleClientServer.lua" server="true"
npl "script/test/network/SimpleClientServer.lua" client="true"
-----------------------------------------------
]]
NPL.load("(gl)script/ide/commonlib.lua"); -- many sub dependency included

local nServerThreadCount = 2;
local initialized;
local isServerInstance =
ParaEngine.GetAppCommandLineByParam("server","false") == "true";

-- expose these files. client/server usually share the same public files
local function AddPublicFiles()
    NPL.AddPublicFile("script/test/network/SimpleClientServer.lua", 1);
end

-- NPL simple server
local function InitServer()
    AddPublicFiles();

    NPL.StartNetServer("127.0.0.1", "60001");

        for i=1, nServerThreadCount do
                local rts_name = "worker"..i;
                local worker = NPL.CreateRuntimeState(rts_name, 0);
                worker:Start();
        end

    LOG.std(nil, "info", "Server", "server is started with %d threads",
nServerThreadCount);
```

```
end

-- NPL simple client
local function InitClient()
    AddPublicFiles();

    -- since this is a pure client, no need to listen to any port.
         NPL.StartNetServer("0", "0");

         -- add the server address
         NPL.AddNPLRuntimeAddress({host="127.0.0.1", port="60001",
nid="simpleserver"})

    LOG.std(nil, "info", "Client", "started");

         -- activate a remote neuron file on each thread on the server
         for i=1, nServerThreadCount do
                 local rts_name = "worker"..i;

         while( NPL.activate(string.format("(%s)simpleserver:script/test/net
work/SimpleClientServer.lua", rts_name),
            {TestCase = "TP", data="from client"}) ~=0 ) do
            -- if can not send message, try again.
            echo("failed to send message");
            ParaEngine.Sleep(1);
        end
         end
end

local function activate()
    if(not initialized) then
        initialized = true;
        if(isServerInstance) then
            InitServer();
        else
            InitClient();
        end
         elseif(msg and msg.TestCase) then
         LOG.std(nil, "info", "test", "%s got a message", isServerInstance
and "server" or "client");
                 echo(msg);
         end
end
NPL.this(activate);
```

上面的服务端是多线程的。

开始时，NPL.activate 调用一个新的远程服务端（这个服务端还未建立 TCP 连接），消息被丢弃同时返回一个非 0 的值。NPLExtension.lua 包含大量的帮助函数去帮助你发送一个保障消息，例如：NPL.activate_with_timeout。你需要 include commonlib 来使用它。

信任的连接和 NID：

接收者的激活函数可以分配任何名称或 nid 来接入连接的 NPL runtime。

HelloWorld 示例：

现在这里有一个更复杂的 helloworld。它通过把一个激活函数与其关联将一个普通的 helloworld.lua 转变为一个神经元文件。文件接下来就可以被任何 NPL 线程或远程电脑通过它们的 NPL 地址（url）调用了。

```
local function activate()
   if(msg) then
      print(msg.data or "");
   end
   NPL.activate("(gl)helloworld.lua", {data="hello world!"})
end
NPL.this(activate);
```

简单的网络服务端示例：

NPL 使用了一个兼容 HTTP 的协议，所以它可以使用同一个 NPL 服务端去处理标准 HTTP 请求。当 NPL runtime 接收到一个 HTTP 请求消息，它将会通过 id -10 把消息发送到一个公共的可见文件。所以我们可以仅仅使用几行代码来创建一个简单的 HTTP 网站服务端，例如：

文件名：main.lua

```
NPL.load("(gl)script/ide/commonlib.lua");

local function StartWebServer()
```

```
        local host = "127.0.0.1";
        local port = "8099";
        -- tell NPL runtime to route all HTTP message to the public neuron
file `http_server.lua`
        NPL.AddPublicFile("source/SimpleWebServer/http_server.lua", -10);
        NPL.StartNetServer(host, port);
        LOG.std(nil, "system", "WebServer", "NPL Server started on
ip:port %s %s", host, port);
end
StartWebServer();

local function activate()
end
NPL.this(activate);
```

文件名：http_server.lua

```
NPL.load("(gl)script/ide/Json.lua");
NPL.load("(gl)script/ide/LuaXML.lua");

local tostring = tostring;
local type = type;

local npl_http = commonlib.gettable("MyCompany.Samples.npl_http");

-- whether to dump all incoming stream;
npl_http.dump_stream = false;

-- keep statistics
local stats = {
        request_received = 0,
}

local default_msg = "HTTP/1.1 200 OK\r\nContent-Length: 31\r\nContent-Type:
text/html\r\n\r\n<html><body>hello</body></html>";

local status_strings = {
        ok ="HTTP/1.1 200 OK\r\n",
        created ="HTTP/1.1 201 Created\r\n",
        accepted ="HTTP/1.1 202 Accepted\r\n",
        no_content = "HTTP/1.1 204 No Content\r\n",
        multiple_choices = "HTTP/1.1 300 Multiple Choices\r\n",
        moved_permanently = "HTTP/1.1 301 Moved Permanently\r\n",
        moved_temporarily = "HTTP/1.1 302 Moved Temporarily\r\n",
```

```
        not_modified = "HTTP/1.1 304 Not Modified\r\n",
        bad_request = "HTTP/1.1 400 Bad Request\r\n",
        unauthorized = "HTTP/1.1 401 Unauthorized\r\n",
        forbidden = "HTTP/1.1 403 Forbidden\r\n",
        not_found = "HTTP/1.1 404 Not Found\r\n",
        internal_server_error = "HTTP/1.1 500 Internal Server Error\r\n",
        not_implemented = "HTTP/1.1 501 Not Implemented\r\n",
        bad_gateway = "HTTP/1.1 502 Bad Gateway\r\n",
        service_unavailable = "HTTP/1.1 503 Service Unavailable\r\n",
};
npl_http.status_strings = status_strings;

-- make an HTML response
-- @param return_code: nil if default to "ok"(200)
function npl_http.make_html_response(nid, html, return_code, headers)
        if(type(html) == "table") then
                html = commonlib.Lua2XmlString(html);
        end
        npl_http.make_response(nid, html, return_code, headers);
end

-- make a json response
-- @param return_code: nil if default to "ok"(200)
function npl_http.make_json_response(nid, json, return_code, headers)
        if(type(html) == "table") then
                json = commonlib.Json.Encode(json)
        end
        npl_http.make_response(nid, json, return_code, headers);
end

-- make a string response
-- @param return_code: nil if default to "ok"(200)
-- @param body: must be string
-- @return true if send.
function npl_http.make_response(nid, body, return_code, headers)
        if(type(body) == "string" and nid) then
                local out = {};
                out[#out+1] = status_strings[return_code or "ok"] or
return_code["not_found"];
                if(body~="") then
                        out[#out+1] = format("Content-Length: %d\r\n",
#body);
                end
                if(headers) then
```

```
                        local name, value;
                        for name, value in pairs(headers) do
                                if(name ~= "Content-Length") then
                                        out[#out+1] =
format("%s: %s\r\n", name, value);
                                end
                        end
                end
                out[#out+1] = "\r\n";
                out[#out+1] = body;

                -- if file name is "http",  the message body is raw http
stream
                return NPL.activate(format("%s:http", nid),
table.concat(out));
        end
end


local function activate()
        stats.request_received = stats.request_received + 1;
        local msg=msg;
        local nid = msg.tid or msg.nid;
        if(npl_http.dump_stream) then
                log("HTTP:"); echo(msg);
        end
        npl_http.make_response(nid, format("<html><body>hello world.
req: %d. input is %s</body></html>", stats.request_received,
commonlib.serialize_compact(msg)));
end
NPL.this(activate)
```

1.4 架构

本节将介绍 NPL 的一些基本架构，通过各种架构图将更加清晰直观的揭示 NPL 的内部组成。

首先 NPL 整体的架构图如图 1.1 所示：

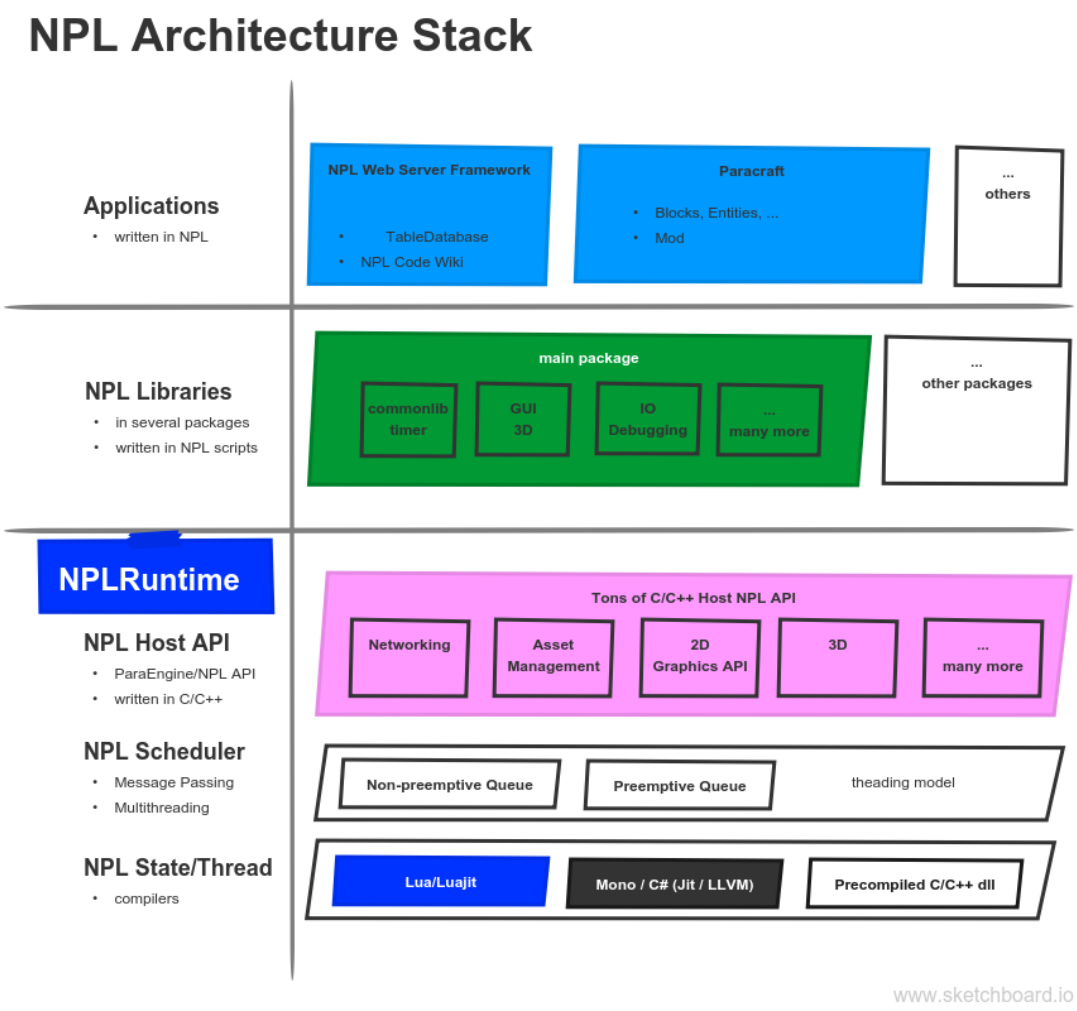

图 1.1 NPL 架构图

NPL 这个体系架构可分为三层：底层是 NPL Runtime 的源代码，在操作系统上用 C++实现的，提供一系列的基础 API；中间层是在基础 API 上编写的一些方便调用的库，例如 main package 和其他库； 顶层便是基于 NPL 的应用软件，例如广受欢迎的我的世界 paracraft。

底层又分为 3 个部分，最下面是一些运行环境，包括 lua， C#，C++的运行环境；然后在运行环境之上便是线程的调度，包含优先级和非优先级的线程调度；再往上就是用 C++写的 Host API。 我们把这些 Host API 进一步展开，便是如图 1.2 所示的结构：

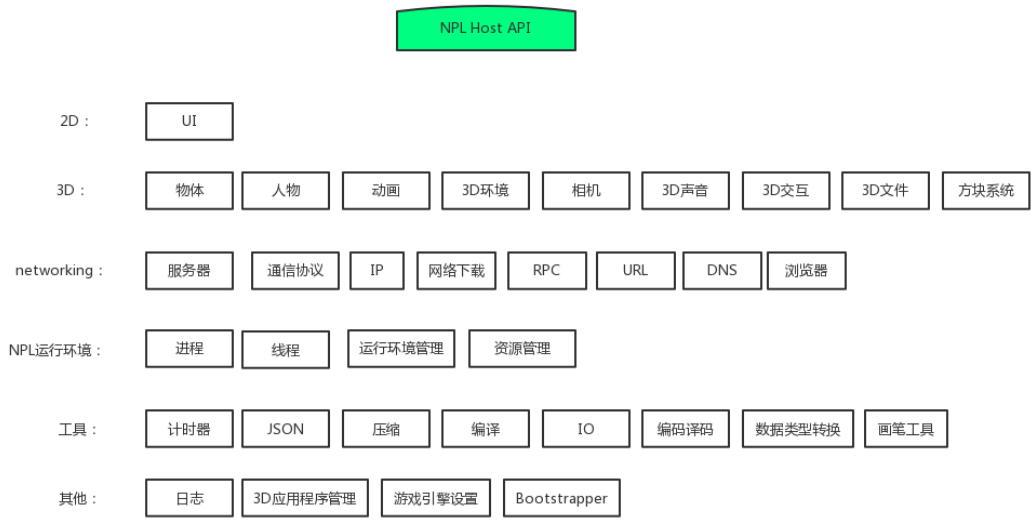

图 1.2 NPL Host API

可以看到 NPL 提供了很多类别的 API 来供用户完成一个应用软件的开发。2D 中包括 UI；3D 中有 3D 世界中的各种元素，相机是用于将 3D 世界转换成 2D 画面的工具，方块系统作为一个 3D 世界的特例，全部由各种不同的方块搭建而成； 工具中的画笔工具可以用编写脚本的方式画出 2D 和 3D 物体。其他中的 Bootstrapper 表示一个应用的主控循环程序。

中间层的 NPL 库主要包含 main package，这部分提供如下图所示的一些功能：

图 1.3 NPL Main package 组成

可以看到 main package 中包含以下部分：通用的函数库，例如数据结构、2D/3D 图形等； 面向对象：提供面向对象编程的一些方法，例如查找一个对象的子类；计时器；

文件加载；序列化；HTTP 客户端；网络 API；文件和 IO；本地化：例如设置字体；LOG；数据库； UI 库；数学库；和其他。

2 代码结构

本章要介绍与 NPL 相关的源代码，NPL 的初学者经常会接触以下三个源代码文件：NPL Runtime, Main package, Paracraft， ParacraftSDK. NPL Runtime 是 NPL 运行环境的实现代码，用 C++语言编写，这个文件可分为三层来理解：底层是 NPL 线程的实现，对 lua, C#, C/C++语言的编译执行；中间层是 NPL 线程，文件的调度；顶层是 NPL 的 API，涵盖网络连接，资源管理，2D 图形，3D 引擎等 。 Main package 是用 NPL API 编写的高级函数库， 这样用户可以更加方便的编写 NPL 程序， 当然用户也可以用 NPL API 开发自己的库。Paracraft 是用 NPL 语言编写的一个 3D 应用，同时用到了 NPL API 和 Main package，可作为 NPL 的开发环境来使用。 ParacraftSDK 是 NPL 软件开发工具包，集成了 NPLRuntime 和 Paracraft . 前三者的关系可以用下面的图来描述：

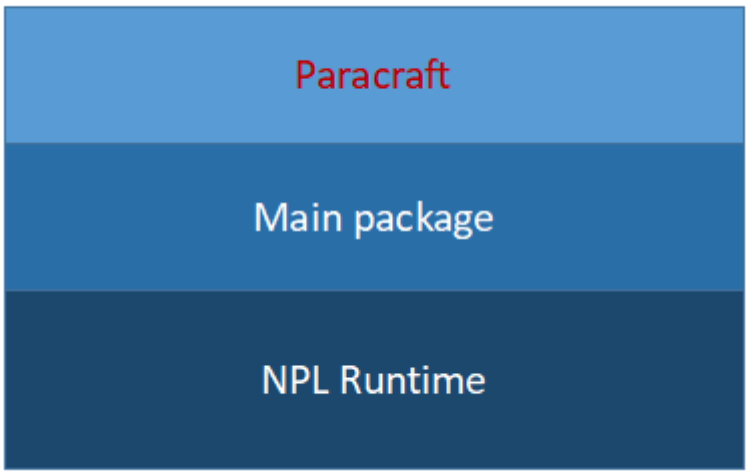

图 2.1 NPL 代码层次关系图

2.1 NPL Runtime

NPL Runtime 源代码主要包含两个部分，一个 NPL（neuron parallel language）部分，该部分的功能是让 NPL 脚本文件能够被解释执行；另一个是 Paraengine 部分，给部分是一个 3D 引擎的实现代码。用 C++编写的源代码中有两个命名空间：NPL 和 Paraengine，分别对应着上述两部分。

NPL Runtime 源文件根目录下包含 5 个文件和一些配置说明文档。五个文件夹有：Client，Server，NPLRuntime，npl_packages，installer。配置说明文档包括版本控制工具 git, 持续集成服务工具 Travis-cli、appveyor 的配置文件； linux, mac 操作系统下的相关配置文件；项目部署文件，许可证，安装 npl 的配置文件以及 readme 文件。各种类型文件的作用参见附录。

Client 文件夹包含了作为 NPL 客户端所必须的文件。Server 文件夹中包含了作为 NPL 服务器的必要文件. 如果一个 NPL 运行环境既要作为客户端又要作为服务器，那么 client 和 server 文件夹都是必须的。NPLRuntime 文件夹中主要包含 pareengine 客户端和服务器端的 CMakeLists 文件，还有一些测试程序。npl_packages 文件夹存放着用 NPL API 编写的库，包括 Main package，用户自己写的库也可以放在这个目录下。可能用户下载的 NPL Runtime 源代码的这个文件夹下并没有包含 Main package，如果需要用到 Main package 可以单独下载并放在该目录下。installer 文件夹中包含与安装，卸载相关的文件。包含环境变量的设置，文件的操作：安装后会生成哪些文件，文件的存放位置，文件的删除，文件的压缩解压。各个子文件夹作用如下表所示：

|——Client

| |——External 包含一些外部的函数库

| | |——assimp-3.1.1 Assimp（Open Asset Import Library）是一个支持读取多种模型资源的开源库。

| | |——bullet-2.75 这是一个物体碰撞检测和刚体的动态库。

| | |——freetype2 字体引擎，它提供统一的接口来访问多种字体格式文件。

| | |——Jpeg 与 jpeg 格式图像的交互库，包含对 jpeg 格式图像的各种操作。

| |——ParaEngineClient NPL 客户端的主体

| | |——2dengine 2D 用户 UI 引擎，如直观的按钮、平面图形等。

| | |——3dengine 主要的 3D 引擎实现，如摄像机、场景、物体、方块系统等。

dreamTech

| | |——BlockEngine 单独实现的针对方块世界运转的引擎。

| | |——BMaxModel 与 Bmax 格式的 3D 模型文件的交互库。

| | |——CadModel 与 cad 文件的交互库。

| | |——common 用于渲染 3D 的窗口的实现。

| | |——Core 程序运转的核心，如事件驱动系统、程序资源管理系统等。

| | |——Engine Win32 主体窗口的实现。

| | |——ic 与数据库交互相关的库。

| | |——IO 文件的输入输出操作库，比如实现文件的读写， 管理所有文件，提供对文件的路径搜索。

| | |——jabber jabber 服务器相关。

| | |——NPL 以实现 NPL 语言运转设计的主要运作流程。

| | |——PaintEngine 像 2D 引擎中的 GDI 引擎，用来画 2D 物体和简单的 3D 物体，比如线条，矩形等。

| | |——ParascriptBindings 将 C++接口暴露给 Lua/NPL 语言调用。

| | |——ParaXModel 与 ParaXModel 文件交互库， 这是一种 2 进制文件格式，只用于表示静态和动态 3D 角色的信息。

| | |——physics 一些生成模型物理面的实现。

| | |——platform 操作平台相关。

| | |——protocol 负责解析应用通信协议。

| | |——renderer 渲染相关。

| | |——shaders 着色相关。

| | |——terrain 3D 地形系统。

IdreamTech

| | |——util 常用工具，如字符串编码转换等。

| | |——VoxelMesh 立体网格。

| | |——WebBrowser 网络浏览器相关。

| | |——WebSocket 网络套接字相关。

| |——ParaEngineClientApp paraengine 应用文件夹。

| |——PhysicsBT 实现了 IParaPhysics.h 接口并且静态链接 bullet 库

|——Server

| |——curl-7.47.1 curl 是利用 URL 语法在命令行方式下工作的开源文件传输工具。

| |——jsoncpp-0.5.0 解析 json 格式数据。

| |——lua-5.1.4 lua 语言的解释器。

| |——luabind-0.9 把 c++编写的函数和类暴露给 lua。

| |——LuaJIT 一个运行时解释的 Lua 代码的解释器， 相对与上述的 lua 更为高效。

| |——luasql 这是一个从 lua 到数据库管理系统的简单接口。

| |——NPLMono 一个插件，可以把 ParaEngine API 暴露给 C# mono 脚本接口。

| |——sqlite-3.6.23.1 一款轻型数据库。

| |——tinyxpath_1_3_1 一种轻型的 xpath 语言解析器，可以用来解析 xml。

| |——zlib-1.2.3 压缩与解压。

|——NPLRuntime

|——npl_packages

|——installer

2.2 Main package

Main package 是在 NPL API 的基础上用 lua 语言编写的库，像 java 中的各种类库，

dreamTech

方便用户使用，使得 NPL 编程更为简单高效。

Main package 源代码目录下包含 3 个文件夹：Documentation，npl_mod，script；和一些相关的配置说明等文件。

Documentation 文件夹中包含一些文档，npl_mod 文件夹中存放着 NPL 的插件，script 文件夹是主要的文件夹。script 文件夹下又分为 4 部分： test, sqlite, ide, apps；下面注意介绍其功能：

Test

测试文件夹，包含测试各种函数的代码。

Sqlite

轻型的 paraengine 本地数据库。

Ide

标准库，包含时钟，动画，音频引擎，编码解码，NPL 环境配置，debug，显示 2D,3D 画面，播放视频，文件交互，GUI 引擎，数学工具，数据库，网络编程等。详细内容用户可自行查看源代码或 NPL 相关网站。

Apps

该文件夹下存放着各种 NPL 的应用程序，目前只有一个 WebServer, 这是一个基于 HTTP 协议的网络服务器，类似于 Apache。有两个实现版本，旧版本调用 httpd.lua ，并且通过 luasocket 建立 tcp 连接和其他操作，使用 lua 的一个服务器函数库 copas 在同一个线程中处理来自客户端的所有请求；新的实现版本用 NPL 自带的异步消息调度系统，这种方法比旧版本更加快速且灵活。该文件夹下两个版本都有。

2.3 Paracraft

Paracraft 是用 NPL 语言编写的一个应用游戏，该游戏是一个世界构建游戏，玩家可以

dreamTech

在一个空白的世界构建成自己想象的样子，玩家也可以把它当成 NPL 的集成开发环境。在这个游戏中，世界的基本组成元素的是一个方块，世界便是由许许多多的不同类型的方块组成的。

在 Paracraft 源代码中，根目录下包含一些配置说明文件和两个文件夹：config, script。config 文件夹包含 Paracraft 的配置文件，比如该应用中有哪些方块的类型。Script 文件夹包含该应用游戏的主要实现程序。

2.4 ParacraftSDK

这个是 NPL 软件开发工具包，集成了 NPLRuntime 和 Paracraft，安装的时候会方便的自动配置环境变量，不用用户手动设置；里面还有一些用 NPL 写的项目例子供用户参考。

IdreamTech

3 线程

NPL 支持多线程技术，用户可以在本地 NPL 运行环境中创建新线程，调度本地或是远程 NPL 运行环境上的线程。

本章首先会介绍一个 NPL 引擎中的线程模型；然后介绍本地的线程调度，再介绍远程线程调度和远程程序调度（RPC），接着介绍贯穿本地和远程线程调度的 activate 函数的实现；最后再讲讲 NPL 引擎启动后存在的一些工作线程。

3.1 线程模型

每个 NPL 引擎启动之后会维护一个线程池，线程池由已经创建好的线程组成。对于线程池的操作包括创建新线程，删除线程，查找线程等。

每个 NPL 线程都有自己的名称，类型，对应 C++中的线程，输入消息队列，一个 C# 运行环境，一个 lua 运行环境。线程的名称起标识线程的作用。线程的类型有 4 种：NPL，NPL_LITE, DLL, NPL_ExternalLuaState。NPL 类型的线程是默认被创建的线程，如果用户没有指定所要创建线程的类型，那么 NPL 类型的线程会被创建，这种线程在创建是会载入所有的 NPL 函数和 ParaEngine 函数，大概需要 1M 的内存。NPL_LITE 类型的线程是 NPL 类型线程的简化版， 只有 NPL 函数和一些有限的函数在创建时被载入。DLL 类型什么都不会载入，这种线程通常和动态链接库插件一起使用。NPL_ExternalLuaState 类型线程载入的函数与 NPL 类型线程相同，不同的是这种类型下没有嵌入 lua 运行环境，需要在外部进行设定。NPL 线程的实际线程实现是依赖于 C++boost 库中的 thread 类，每个 NPL 线程都包含一个 C++ boost 库中的 tread 对象。NPL 线程中的输入消息队列管理着该线程要执行的文件；输入消息队列由一个个的 NPL 消息组成的，NPL 消息中包含要执行的文件。C# 和 lua 运行环境分别用于执行 C#和 lua 代码。线程池模型如图 3.1 所示。

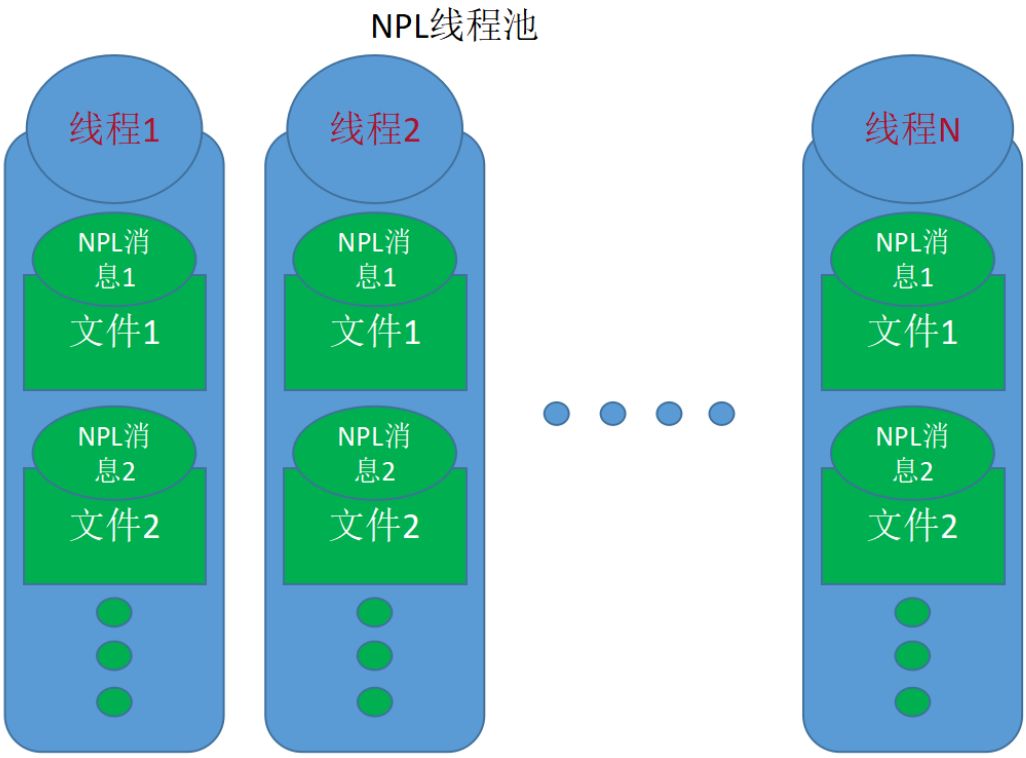

图 3.1 NPL 线程池模型

创建线程可用 NPL.CreateRuntimeState（name, type）函数，创建线程时需指定线程名称 name 和线程类型 type，线程类型即为上面介绍的四种类型之一，分别用 0~3 的数字表示。如

NPL.CreateRuntimeState（" A "，0）表示创建 NPL 类型的线程 A。

NPL.CreateRuntimeState（" B "，2）表示创建 DLL 类型的线程 B。

在 NPL 运行环境中创建线程后，该线程会自动添加到线程池中。调用 NPL.CreateRuntimeState 函数之后，NPL 内部的执行流程图如图 3.2：

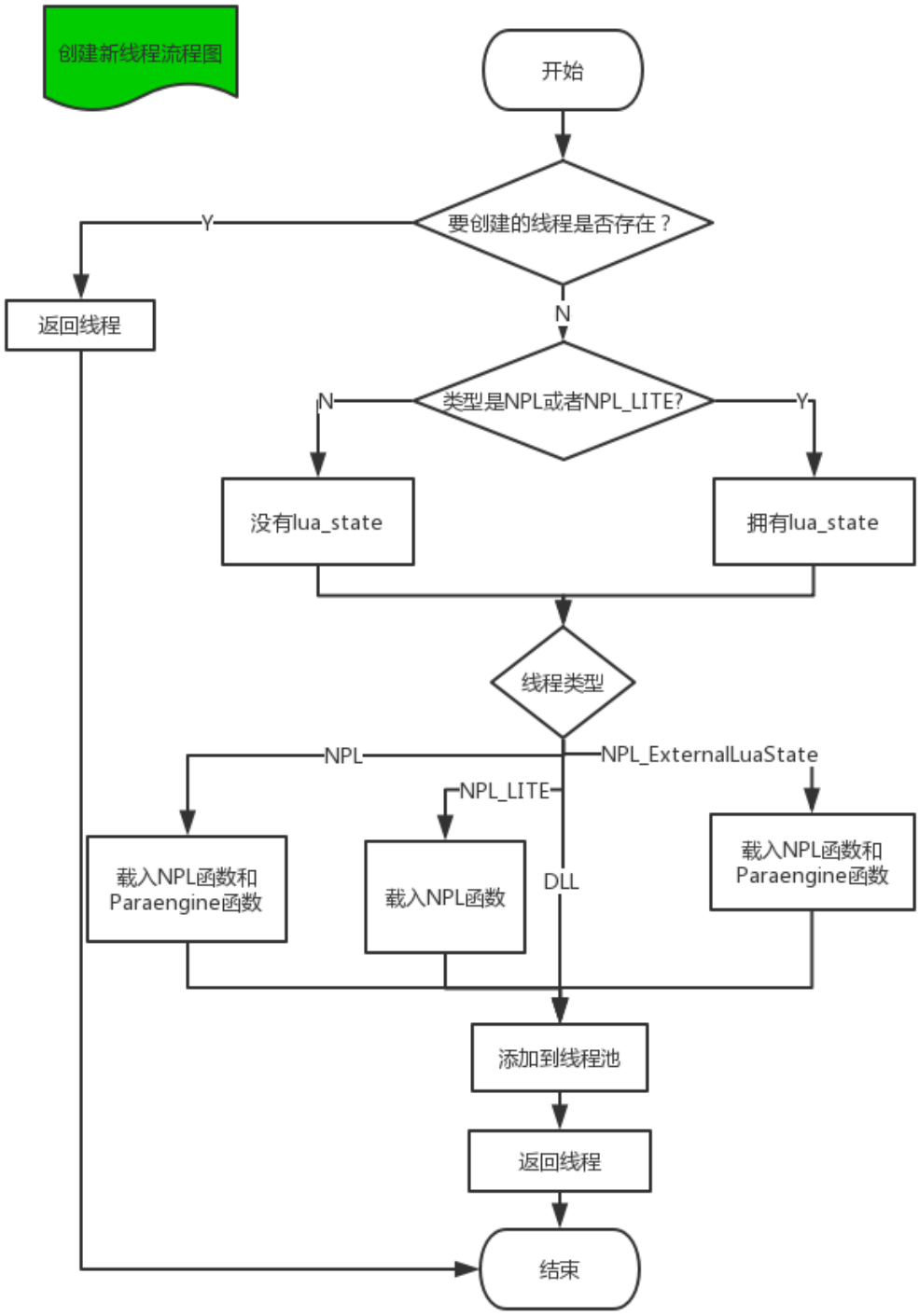

图 3.2 NPL 线程创建流程图

3.2 在线程中运行程序

创建好一个线程之后，接下来就是让一个线程开始工作，具体的来讲，是让一个线程开始运行程序。 我们可以调用 Start 函数来进行这项操作，一般我们创建好一个线程之后会

立即启动，例如

—>NPL.CreateRuntimeState("A", 0):Start()

简单来讲，启动线程的过程就是让从线程维护的输入消息队列中取出消息，然后调用 process 函数处理消息，线程怎么知道输入消息队列里是否有消息需要处理呢？这里有两种方法，一种是利用计时器每隔一段时间就看看消息队列是否有待处理的消息，NPL 的主线程用的便是这种方法；另一种方法是利用一个 boost 库中的条件变量，如果消息队列为空，则该线程处于阻塞状态，如果有消息就在条件变量上调用 notification 函数来唤醒当前线程。线程处理处理消息的流程图如图 4.3 所示。nRes 变量表示消息处理过程是否结束。

IdreamTech

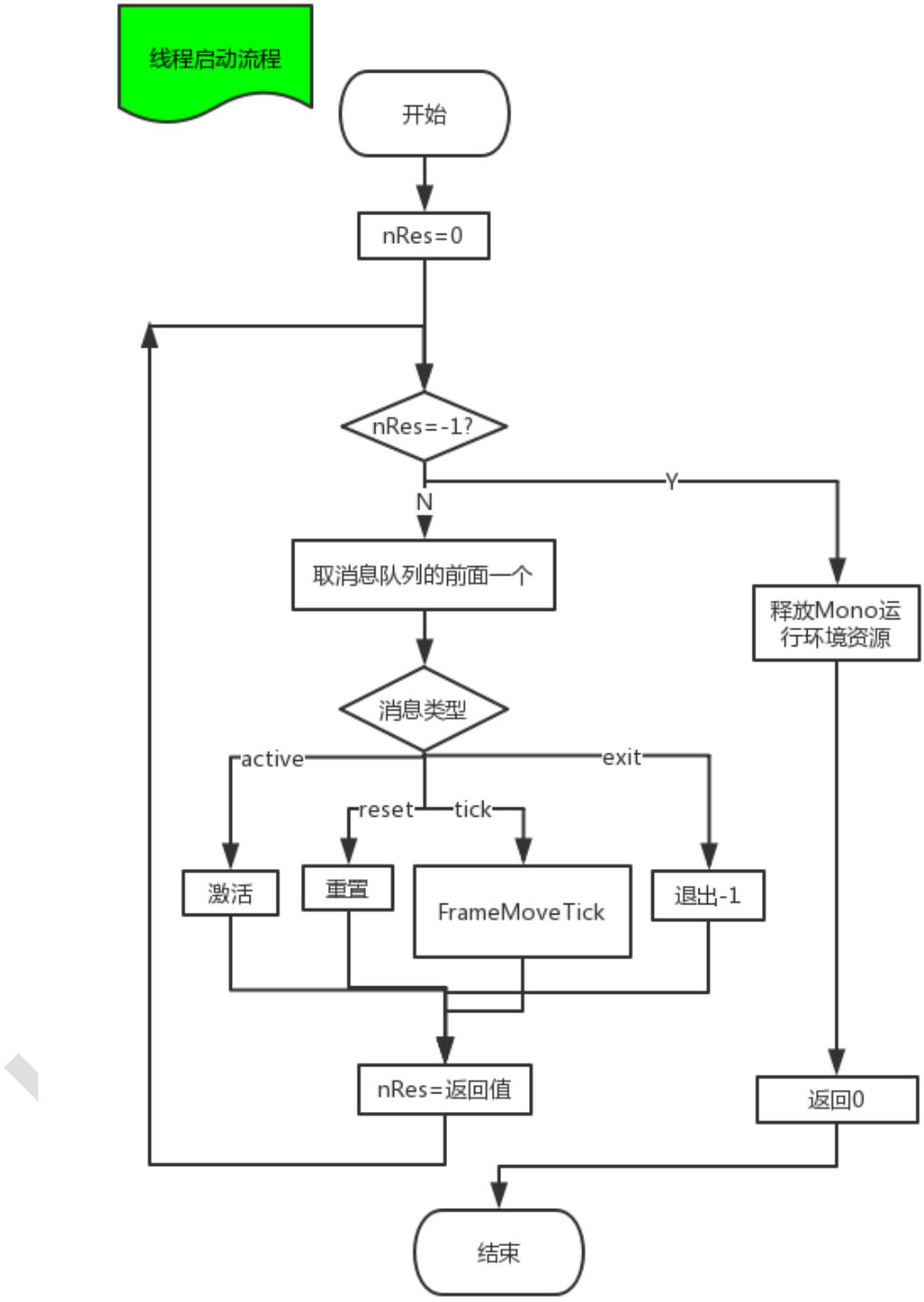

图 3.3 线程启动流程

3.3 本地 NPL 运行环境的多线程调度

用户可以在一个指定的线程中执行指定的文件。这一功能的实现依赖于 NPL.activate 函数。使用该函数的语法如下：

NPL.activate(url, msg)

参数 url：一个 NPL 文件的 URL 地址，完整格式为：

(sRuntimeStateName|gl) sNID: sRelativePath@DNSServerName

(sRuntimeStateName|gl)中的 sRuntimeStateName 是所选择线程的名称， 如果选择不写 sRuntimeStateName 而是只写 gl，那么表示本地当前的线程； sNID 表示 NPL 运行环境的标识符； sRelativePath 为文件的相对路径，一般以 main 文件夹目录作为参考；@sDNSServerName 表示 DNS 服务器的名称。比如以下的 url 都是正确的：

—> (worker1)script/hello.lua 本地 NPL 运行环境上的 worker1 线程上的一个文件

—>（gl）script/hello.lua 本地当前线程上的一个文件。

—>(worker1)NPLRouter.dll worker1 线程上的一个 dll 文件

—>(worker1)DBServer.dll/DBServer.DBServer.cs worker1 线程上的一个 C#文件

参数 msg: 发送给 url 地址上的的数据。

调用该函数就能在指定的线程中执行选择的文件，可能该文件不会立即被执行，而是首先会用该文件名字包装成一个 NPL 信息，然后插入该线程的输入消息队列中，等待被执行。文件被成功激活则返回 0，失败返回非 0 值。

通过 NPL.activate 函数激活的文件被称为神经元文件，神经元文件中必须包含 active 函数，神经元文件被激活后，程序执行的入口便是 active 函数。神经元文件有三种格式：Lua 脚本文件、C#脚本文件、dll 文件。其中 C#文件的激活并不是在 NPL 的线程中，而是通过一个相关插件（NPLMono2_d.dll）在 Mono 运行环境中的线程中激活。 三种格式神经元文件的 active 函数形式不同，下面通过例子一一讲解。

（1） Lua 脚本文件中需包含 active 函数，例如：

Lua 文件名: script/HelloWorld.lua

激活: NPL.activate("script/HelloWorld.lua", {data})

local function activate()

end

NPL.this(activate)

（2） C# 脚本文件中需包含一个与文件同名的类，然后在该类中定义一个 active 函数，传入 NPL.active 函数的文件名也可以包含命名空间，例如：

C#文件名: HelloWorld.cs

激活:

```
NPL.activate("mono/MyMonoLib.dll/HelloWorld.cs", {data})
NPL.activate("mono/MyMonoLib.dll/ParaMono.HelloWorld.cs", {data})
class HelloWorld
{
	public static void activate(ref int type, ref string msg)
	{
	}
}
namespace ParaMono
{
	class HelloWorld
	{
		public static void activate(ref int type, ref string msg)
		{
		}
```

dreamTech

```
        }
}
```

（3） Dll 文件需要暴露一个 active（）函数，例如：

Dll 文件名: HelloWorld.cpp inside C++ MyPlugin.dll

激活: NPL.activate("MyPlugin.dll", {data})

```
CORE_EXPORT_DECL void LibActivate(int nType, void* pVoid)
{
        if(nType == ParaEngine::PluginActType_STATE)
        {
        }
}
```

本地多线程调度模型如图 3.4 所示：一个 NPLruntime 的线程池种有很多线程，不同线程之间的调度通过 activate 函数进行， 而 NPLruntime 种文件列表的文件又是通过 activate 函数和一个线程进行绑定的。这种调度模式就像计算机网络的星型拓扑网络，，每个线程可以通过在中心位置的 activate 函数去调度另一个线程。

dreamTech

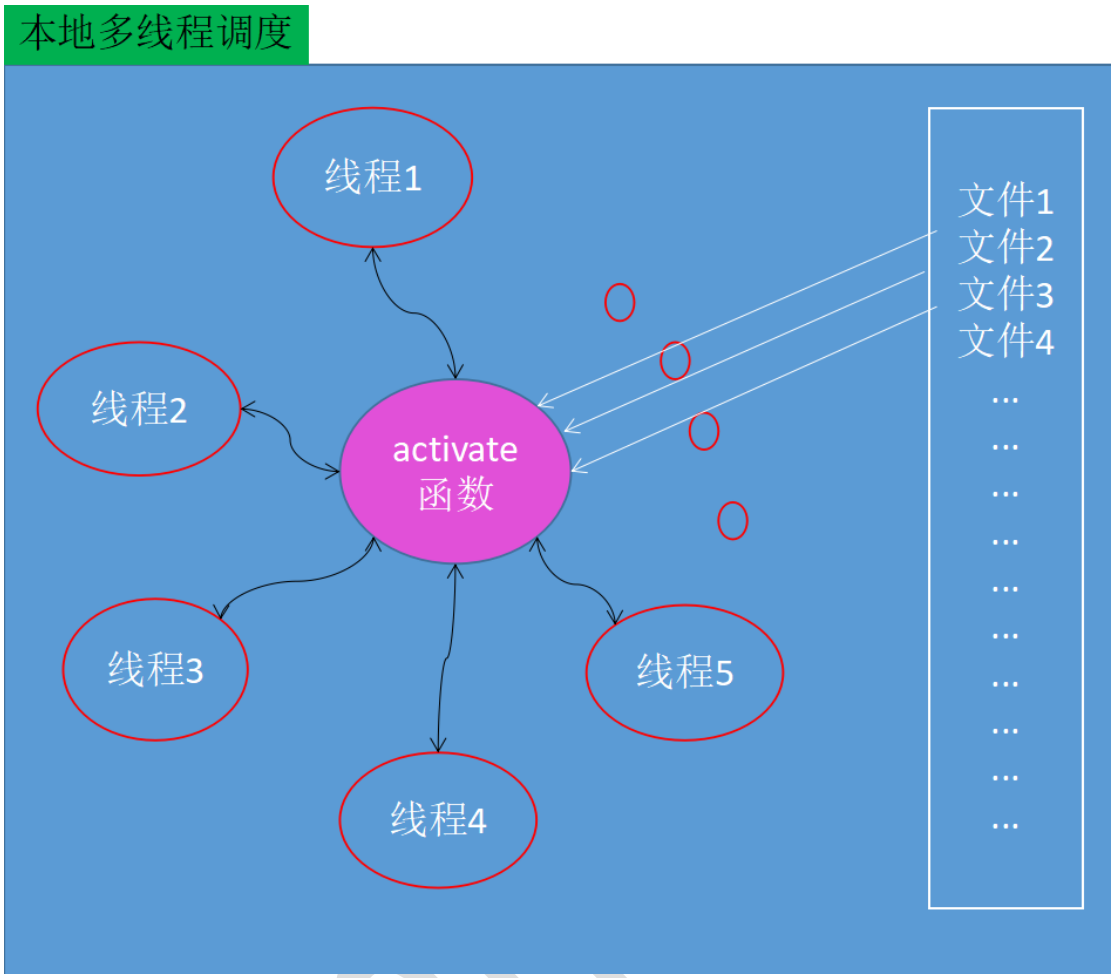

图 3.4 本地多线程调度

3.4 Async RPC

线程调度

上一节考虑的是在一个 NPL 引擎中的多线程的调度，在这一节要介绍的是在一个 NPL 引擎中对于另一个 NPL 引擎中的线程的调用。通过这种调用可以实现在远程 NPL 运行环境上执行指定的文件，这种技术就是 RPC（remote procedure call） 。

同样这种跨 NPL 运行环境的线程调用的实现还是依赖于 NPL.activate 函数，使用起来与本地多线程调度主要有 3 个区别。

第一个区别在于 NPL 运行环境，在一次远程程序调度中涉及到两个 NPL 运行环境。其中调度者是客户端，被调度者是服务器端。所以作为服务器端的 NPL 运行环境需要开启服务器，以便能够被客户端访问到。开启服务可以用 NPL.StartNetServer(ip, port)函数， ip

为 ip 地址，port 为端口号. NPL 引擎在设计的时候都带有一个服务器端，对应与源代码中的 CNPLNetServer 类，需要服务器的功能的时候就开启。

第二个区别是文件的相对路径，因为是远程程序调度，自然而然的，这里说的文件变成了远程 NPL 运行环境上（即服务器）的文件。要想让服务器上的文件能够被其他 NPL 运行环境访问到，必须把该文件添加到当前 NPL 环境的公共文件列表中。这个公共文件列表由服务器的调度器维护，调度器的角色是线程消息队列和底层套接字的接口，决定服务器接受到的信息送往哪个线程处理。将文件添加到文件列表可以借助 NPL.AddPublicFile（filename,id）函数，Filename 为文件名，id 为文件的编号。如：

-> NPL.AddPublicFile(“script/test/test.lua”,1) 将 test.lua 文件加入公共文件列表，并且将编号设为 1.

第三个区别就在于 url 地址中的 sNID 参数的变化，这里的 sNID 变成了远程 NPL 运行环境的标识符。例如以下的 url：

-> user001@paraengine.com:script/hello.lua user001 NPL 运行环境的默认线程上的一个文件。

-> (world1)server001@paraengine.com:script/hello.lua server001 NPL 运行环境上的 world1 线程上的一个文件。

调用 NPL.activate 函数之后，首先客户端会和服务器端建立一条 TCP 连接，然后客户端会用指定的文件名字构造一个 NPL 输出消息，接着将消息通过 TCP 连接发送到服务器端，接着服务器会将收到的消息转换为 NPL 消息再插入到对应线程的输入消息队列中，等待被处理。

从线程的层面上理解，远程程序调度可以由图 3.5 来描述。可以注意到不论在客户端还是服务器端，都有一个专门负责通信的线程： 调度线程；这个线程扮演一个消息中心的作用，所有进出的消息都要经过该线程。

dreamTech

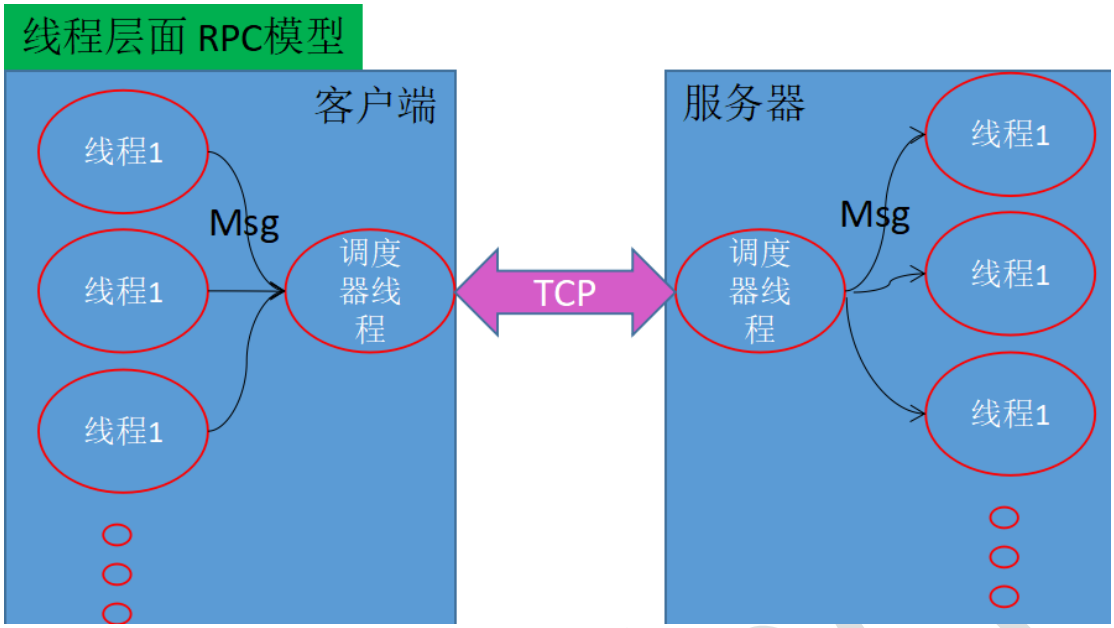

图 3.5 线程层面 RPC 模型

服务器端的连接管理

一个服务器端的 NPL 运行环境可能需要和不止一个的客户端 NPL 运行环境进行交互，这时候就涉及服务器端的 NPL 运行环境对客户端的 NPL 标识符的管理问题。服务器端中维护一个认证连接列表，列表中的每条记录是一个从 TCP 连接到 NID 的键值对。如果服务器认为一条新的连接可靠，就可以给这条连接命名并且添加到认证连接列表中。同样可以从认证连接列表中删除指定的记录。

3.5 Activate 函数原理

不论是本地线程调度还是远程程序调用都用到了 activate 函数，理解该函数是理解 NPL 多线程的关键。这一节将介绍 activate 函数是怎么样在 NPLRuntime 源代码中实现的。先介绍相关的类设计，NPL 线程概念对应着 CNPLRuntimeState 类，创建新线程就是对 CNPLRuntimeState 类的实例化；主要属性在 4.1 节作过介绍。NPL 运行环境对应着 CNPLRuntime 类，CNPLRuntime 类中主要有以下属性：一个线程向量，用以储存当前 NPL 运行环境中的所有线程；一个 CNPLNetServer 类，如果调用该类中的 start()函数，则会启动服务器（即监听一个端口号），那么该 NPL 运行环境也就变成了服务器端，否则就是客户端。Server 对象要负责接受来自客户端的信息并处理，自然需要包含一个调度器类 CNPLDispatcher；调度器有一个 NPLConnection 类用于和其他的 NPL 建立连接。

NPL.activate 函数的原型便是 CNPLRuntime 类中的 Active 函数，现在介绍该函数的执行流程，首先需要理解其中的几个概念。

（1）NPL 文件名是一个结构体类型数据，由四部分组成，也就是上述的文件 URL 地址组成部分。

（2）神经元文件的相对路径是以 main package 目录为参考的路径。

（3）神经元文件的优先级是一个决定 NPL 线程中各个神经元文件的执行顺序的参数，总共有四种优先级：SYSTEM， HIGH，MEDIUM，LOW。这四种优先级分别和数字 0~3 对应，数字越小，优先级越高。在文件激活过程中，SYSTEM 和 HIGH 这两种神经元文件的优先级被设定为一类，即优先级一样； 同样的，MEDIUM，LOW 优先级又是一样的。但是 SYSTEM 和 HIGH 的优先级要比 MEDIUM 和 LOW。可能读者会问，既然神经元文件只有两类优先级，为什么要用一个有四类有限的参数，这是为了方便管理，定义了一个统一的优先级枚举类型数据，有些地方只需要 2 类就够了，可能在其地方需要用上四类优先级的参数。

（4）各个神经元文件不可能孤立的存在着，它们需要相互之间进行交流，像神经元一样交流需要消息载体。因为 NPL 中的神经元文件交流有两类：本地神经元文件之间，两个 NPL 运行环境上的神经元文件之间；所以自然而然两种交流方式需要两种不同的消息载体：NPL 消息（本地）， NPL 输出消息和 NPL 输入消息（远程）。 NPL 消息概念在 4.1 节提过了，这里详细介绍，NPL 消息是神经元文件之间交流的载体。神经元文件之间的交流过程我们可以类比人和人之间的信件交流，一封信要能够准确送到收信人手中需要包含三个要素：收信人地址、信件内容（可以为空）、邮票。收信人地址告诉邮递员往哪送；邮票的种类告诉邮递员怎么送， 如果是加急的就会快一些，如果是普通的就会慢一些；信件内容当然是给收件人传递的信息。同样的，NPL 信息也包含三个要素：NPL 消息类型、文件地址（url）、数据。NPL 消息类型对应着邮票，要执行文件激活操作就将 NPL 消息类型设为 MSG_TYPE_FILE_ACTIVATION（0）；文件地址对应着收件人地址；数据便是信件内容。同样 NPL 输出和输入消息也包含这三个要素，只是表现为更适合在网络上发收的格式，更详细的信息参考网络章节（第五章）。

（5）NPL 线程的输入消息队列管理着该线程要处理的 NPL 消息队列，越靠前的消息越优先被处理。

activate 函数的声明如下：

int Activate(INPLRuntimeState* pRuntimeState, const char * sNeuronFile, const char * code = NULL,int nLength=0, int channel=0, int priority=2, int reliability=3)

该函数的执行流程如图 3.6 所示。决定一个线程调用是本地还是远程的关键一步就是 NPL 文件名的运行环境是否为空，为空则被视为本地调用，不为空则按照 RPC 的流程进行处理。

IdreamTech

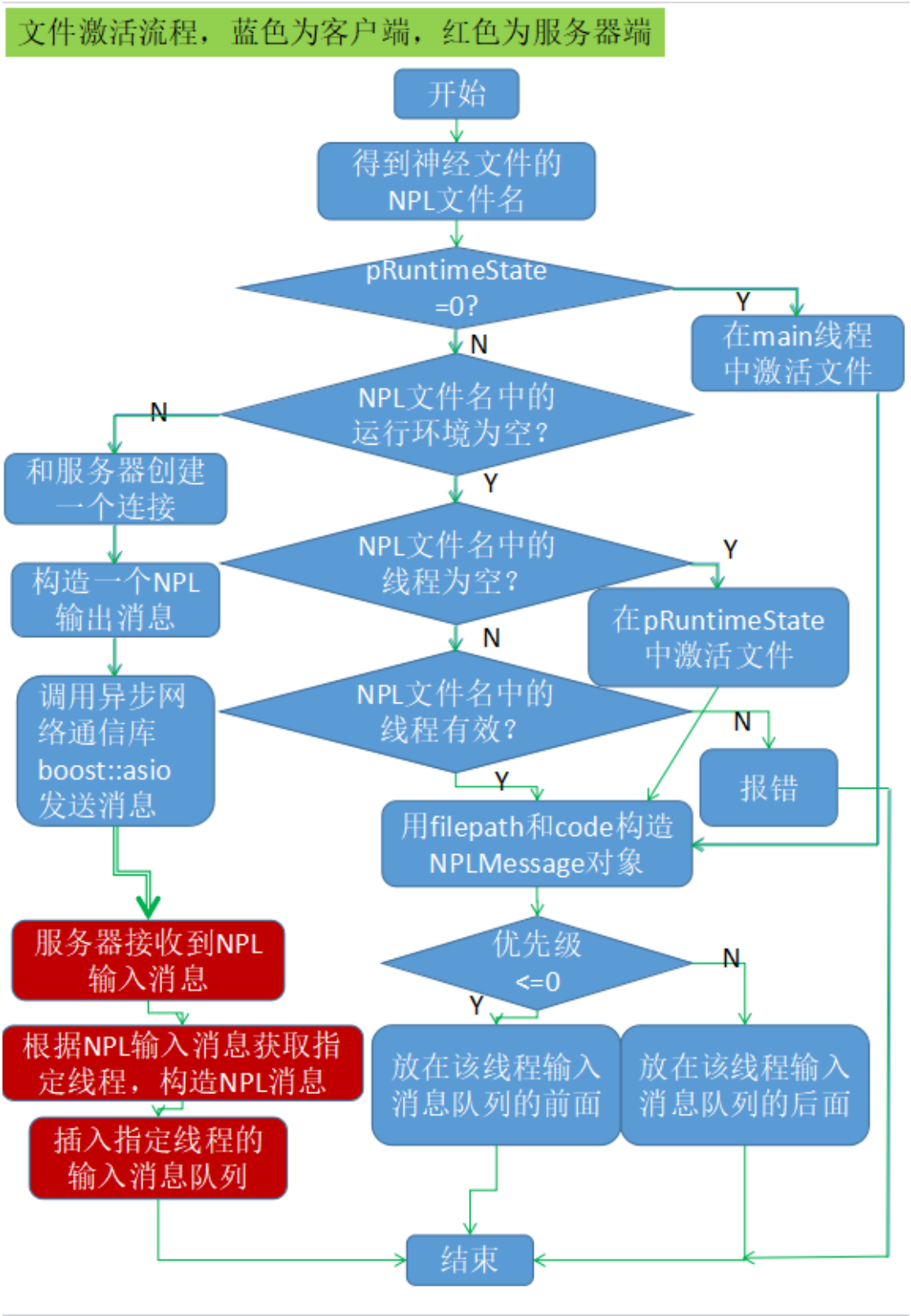
文件激活流程，蓝色为客户端，红色为服务器端
开始
得到神经文件的NPL文件名
pRuntimeState =0?
Y
在main线程中激活文件
N
NPL文件名中的运行环境为空？
N
和服务器创建一个连接
Y
NPL文件名中的线程为空？
Y
在pRuntimeState中激活文件
构造一个NPL输出消息
N
NPL文件名中的线程有效？
N
报错
调用异步网络通信库boost::asio发送消息
Y
用filepath和code构造NPLMessage对象
服务器接收到NPL输入消息
优先级 <=0
N
Y
根据NPL输入消息获取指定线程，构造NPL消息
放在该线程输入消息队列的前面
放在该线程输入消息队列的后面
插入指定线程的输入消息队列
结束

图 3.6 文件激活流程

3.6 工作线程

在 NPL 启动之后会存在五种线程：主线程、窗口线程、网络线程、资源线程和用户自定义线程。主线程也是默认线程、NPL 引擎的入口，用作主控循环，程序初始化、lua/luajit 执行、其他预编译代码的执行、3D 窗口的管理（帧率管理、每个场景物体的状态变化管理、AI 计算、渲染过程控制等），也是创建其他线程的线程。窗口线程中处理 Windows 消息（窗口消息循环），当用户的 Paraengine 应用开始启动之后该线程就会被创建。网络线程处理网络数据的收发，这个线程在启动 NPL 网络服务器后会被创建。资源线程用于处理资源文件的加载，在用户使用载入资源的的时候被创建（IO 库中的异步载入）。用户自定义线程可通过 NPL.createNPLRuntimeState 函数进行创建。工作线程图如图 3.7 所示：

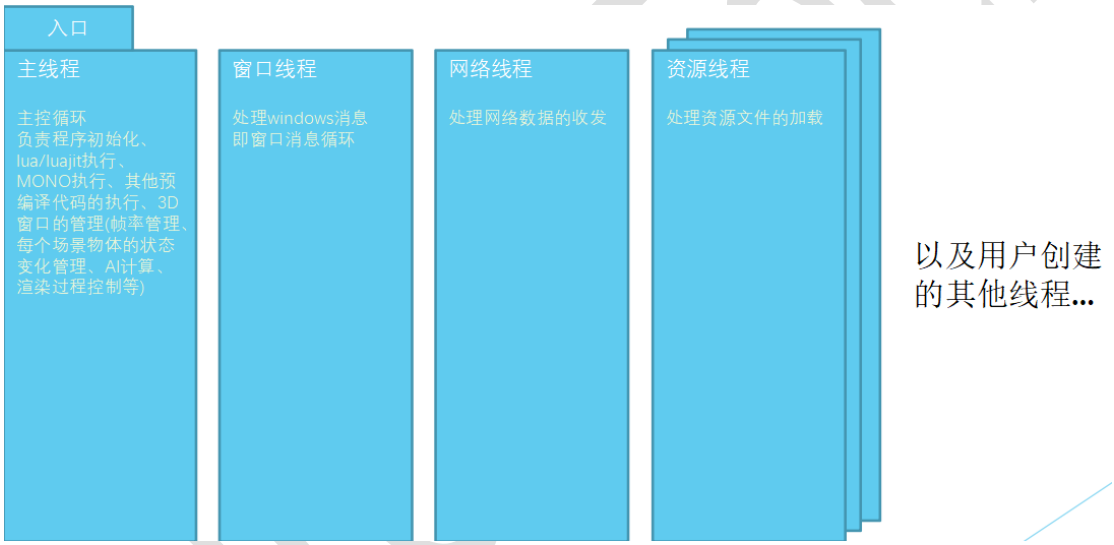

图 3.7 工作线程图

只有主线程支持线程不安全的函数，像一些渲染函数；其他创建的工作线程仅支持线程安全的函数，在其他线程中调用函数的时候需要注意函数是否被标记为线程安全。

4 网络

4.1 网络通信的机制

4.1.1 概述

NPL 网络通信主要组成部分有三个，客户端及服务端，事件调度器，网络连接及 IO 线程。当进行通信时，先建立连接以及 IO 线程，然后通过建好的连接进行数据的发送。在接收到消息后，会将消息读取并把消息传到事件调度器进行异步调度操作，操作完成后返回消息，结束通信。

4.1.2 结构

IdreamTech

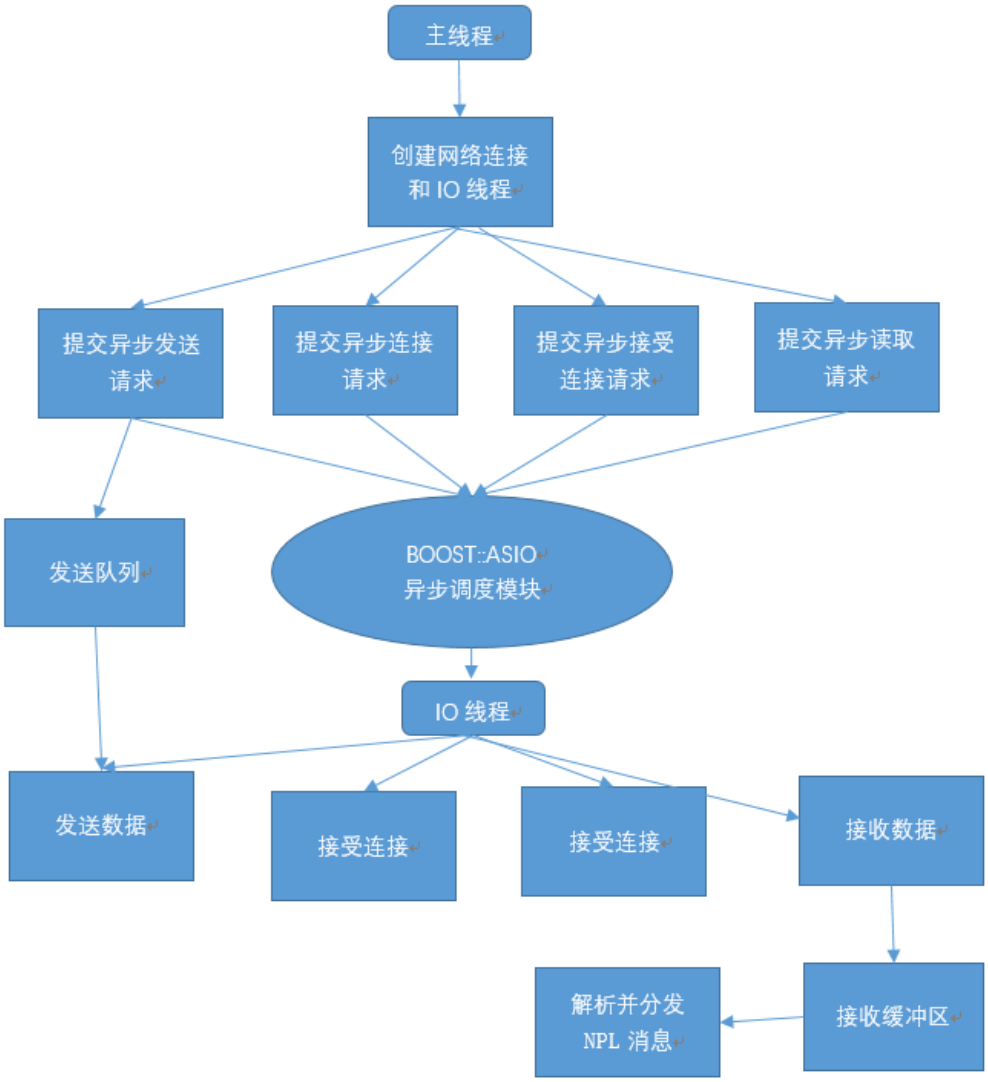

图 4.1　网络结构

4.1.3 重点说明

在网络通信中，最重要的部分是建立连接以及数据发送和读取。在通信中使用了 Boost.Asio 库。

Boost.Asio是一个跨平台的、主要用于网络和其他一些底层输入/输出编程的C++库，在网络通信、COM串行端口和文件上成功地抽象了输入输出的概念。基于这些可以进行同步或者异步的输入输出编程。

在建立连接时，最主要的是使用了 Boost.Asio 库，利用这个库进行处理各种建立连接

时 的操作。

建立连接的流程为：

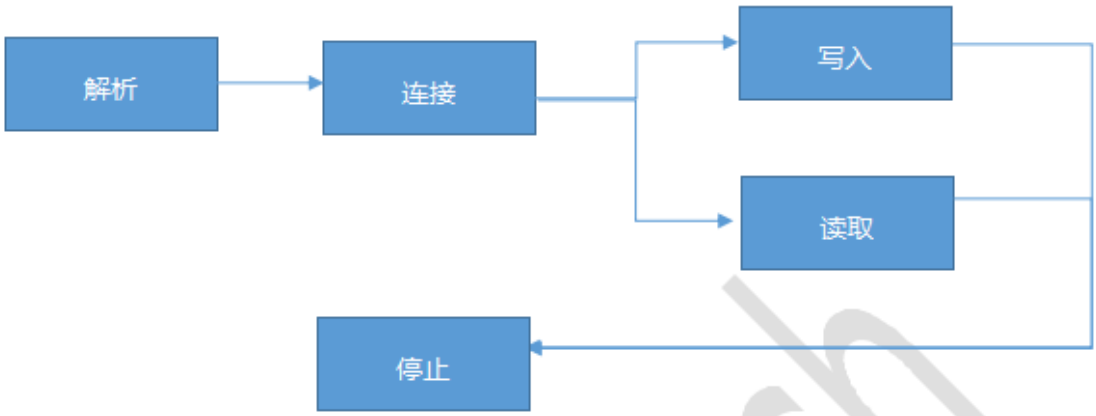

图 4.2 建立连接的流程

其中解析部分是先查询 socket 的信息，它会接收一个错误代码，如果错误代码为 0，迭代器遍历每个端点尝试建立连接。端点是使用某个端口连接到的一个地址，不同了类型的 socket 有自己的 socket 类。

然后连接时也会接收一个错误代码，如果错误代码为 0，则开始连接，否则将会检测其他的端点，如果没有任何端点可连接，则直接提示 warning,然后停止连接。

接下来读取和写入分别是在连接后读取服务端消息及向服务端写入消息的函数，可以并行执行。

最后停止是在连接断开后将 socket 关闭的函数。

当要进行客户端与服务端交互时，在建立好连接后，则开启交互，然后会设置 compression 并将活跃的连接加入到事件调度器中。

Dispatcher 是事件调度器，在事件调度时,经历了创建连接、通过 NID 获取 NPLConnection、添加 NPLConnection、异步调度等操作。其中最重要的是事件调度器对消息的调度。

消息调度部分首先对三种消息类型进行判断，第一种为来自底层的消息，第二种为 NPL 网络消息，其他则为 HTTP 消息。msg 包含从发送方接受的消息加上发送者的 source id。对于未经身份验证的发送方，source id 会存在 msg.tid 中，msg.tid 是一个自动生成的数

字字符串如“~1”。接收方可以一直使用这个暂时的 id：msg.tid 返回消息。如果为第一种消息，事件调度器则会再次判断其访问文件所在的 NPL runtime 端口是否暴露出来，如果暴露出来，则添加 nid 或 tid 使得与线程连接建立；若没有暴露出来则报错。

4.2 消息传递的机制

4.2.1 概述

NPL 的消息传递分为内部传递以及网际传递两部分。在进行网际传递时，是客户端与服务端之间的通信。在进行内部传递时，消息被封装，在建立连接之后消息进行转换，然后异步处理最后若发送消息，则发送至底层。

消息的结构一般被封装为三种形式：接入的消息，发出的消息和 NPL runtime 内部的消息。

4.2.2 结构

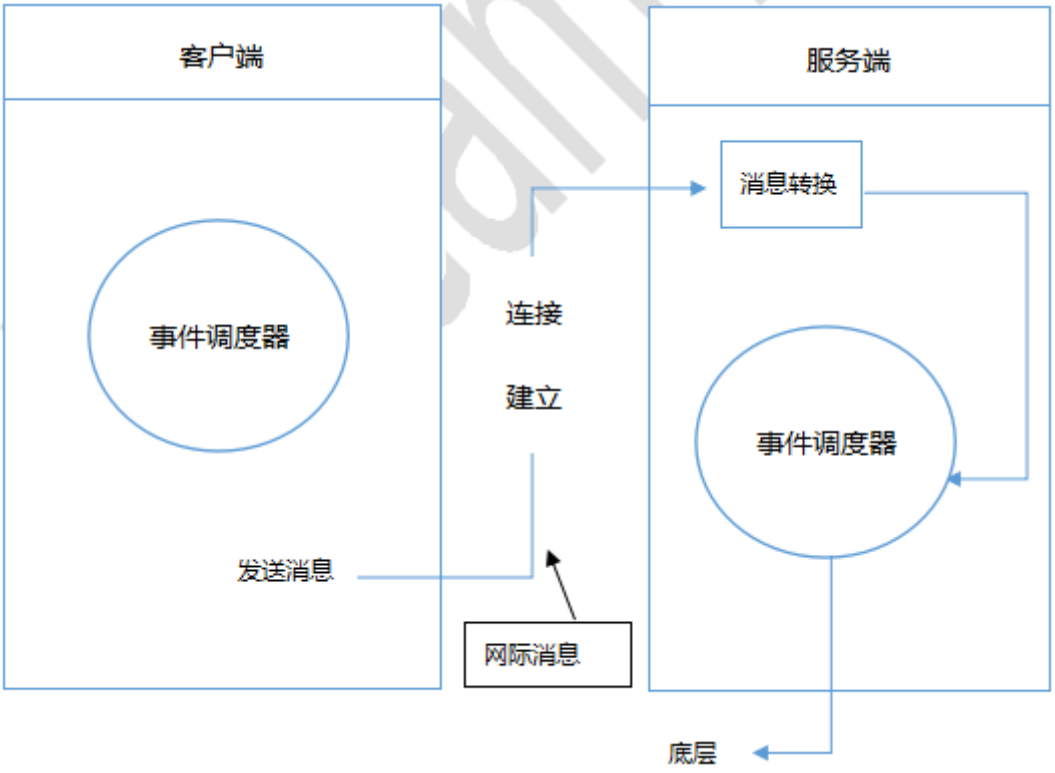

图 4.3 消息传递机制的结构

4.2.3 重点说明

NPL 消息传递部分主要为连接和事件调度两部分并且包含内部的消息传递以及网际消息的传递。

消息的结构一般被封装为三种形式：接入的消息，发出的消息和 NPL runtime 内部的消息。接入的消息适用于从网间接收到的消息格式，在此格式上可以实现 HTTP 协议，发出的消息适用于通过 socket 发送的网间消息格式，NPL runtime 内部消息适用于 NPLRuntime 队列里执行的消息格式 。

当连接已经建立之后，如果缓冲区未满，一个 NPL runtime 则会 SendMessage,否则则会报错。

事件调度器中，负责消息传递的主要是消息调度的部分。在消息调度的部分中，函数会根据获取到的消息将消息转换，然后判断是否将消息传递进行异步处理，若传递，则处理后传至底层。

4.3 激活远程文件

4.3.1 概述

NPL 远程激活文件综合了之前的通信模型和消息传递机制，实现方式为在建立好网络连接及 IO 线程之后，客户端向服务端发送网际的文件消息（这里的客户端和服务端都是一个 NPL runtime，发起连接的时客户端），服务端接受之后读取消息内容并对消息进行初步处理，然后传递消息到事件调度器。在事件调度器中，消息会先进行转换，然后消息传到主控循环中进行异步处理。异步处理时，消息根据其类型被激活。至此，这个客户端发出的消息被激活。

4.3.2 结构

IreamTech

创建好连接及 IO 线程后：

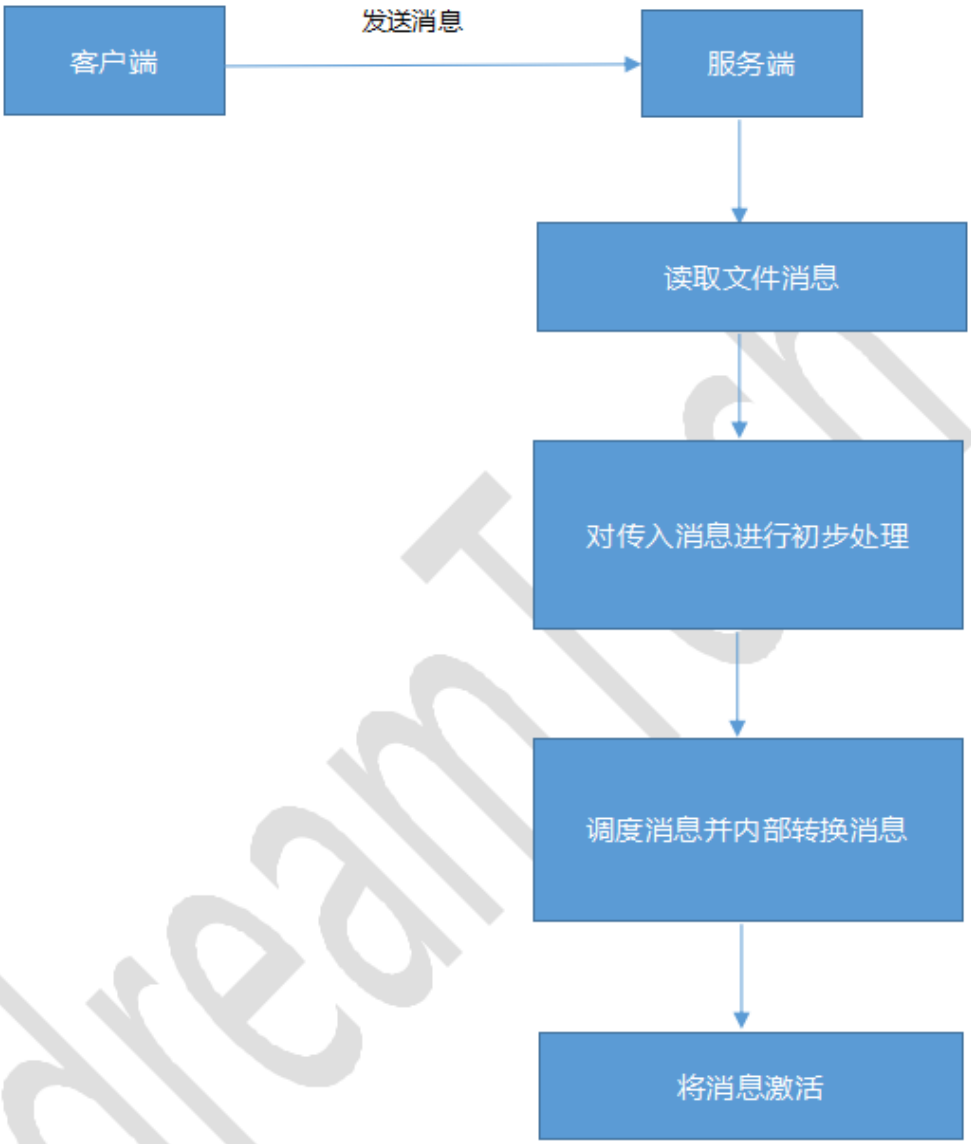

图 4.4 建立连接和线程 IO 后激活文件

4.3.3 重点说明

NPL 远程文件的激活核心部分还是消息传递，在将消息传递后，主控循环内会将其处理激活，完成整个远程激活的操作。

当一个 NPL runtime 服务端暴露一个公共文件，其他客户端的 NPL runtime 可以使用函数来激活它。为了激活服务端，单纯的客户端也叫做初始的服务端。但是它可以具体指明端口为”0”来表明它不会监听进来的连接，成为单纯的客户端。然而，在客户端，我们需要分配一个本地名称到远程的服务端，所以我们可以在之后所有的激活函数的调用中通过

名称查到这个服务端。注意，nid 具体指定的名字是随机的并且仅仅用在客户端电脑去查找另一个电脑。换句话说，不同客户端可以用不同的名字命名同一台远程电脑。

在消息调度过程中，事件调度器将从网际接受的消息转换成 NPL runtime 内部消息格式的 msg，使得其可以在 NPLRuntime 队列中执行。然后对不同类型的 msg 进行处理。如果类型为激活文件，则进行激活操作，文件最终激活。类型为计时器的消息会返回帧移动标记的函数，该函数是随时间频率进行运行的，实际中它是一直处于运行状态的。若为重启类型则重重启，则重新进行激活，若为退出类型则退出激活。

IdreamTech

5 API、库

本章将会介绍 NPLRuntime 的 API 和程序库。API 即应用程序接口，我们说 API 的时候，一般会直接说 API 函数。即 API 的本质是一个个写好的函数或者说方法，操作。这些函数实现了某一个或者某一类功能，当它们被定义好之后，我们需要对系统中的某些物体进行操作，这些 API 函数提供了现成的方法，甚至决定了某些事物能够被操作的类型。Library 又名程序库，它是一类 API，或者有关联的几类 API 的集合，一个 library 通常会针对某一类操作提供大量的 API，例如 Timer，内有大量关于定时器的相关操作 API 函数，通过把相关的 API 函数集合到一个 library 中，能够让我们使用起来非常方便。

在 NPLRuntime 体系中，API 和程序库将定义好的一系列函数集成起来，为运行在 NPLRuntime 中的应用程序提供支持。它大量通过 luabind 绑定给 lua 调用的接口，使我们能够通过直接调用的方式，搭建出我们的应用程序。其中 C++API 往往是底层低级 API，它们的作用是提供原生态的 API，有许多 NPL API 如关于 GUI 方面的 API，是对 C++ API 进行包装，并结合一些自己特有的 API，为 NPLRuntime 提供 API 支持。 它们之间的关系如下图所示：

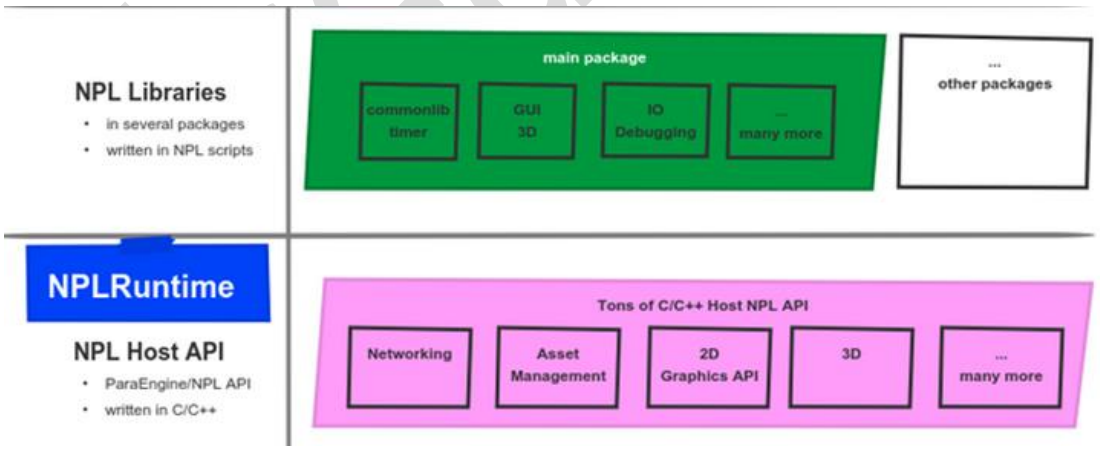

图 5.1 NPL API、库之间的关系

整个 NPLRuntime 的 API 和程序库可分为 NPL Common Libraries、面向对象、NPLload 的文件加载机制、计时器、序列化、HTTP 客户端、网络 API、文件与 IO、本地化、LOG、数据库和 UI 库，同时还借用了 C++中的相关 API 如数学库。如下图所示：

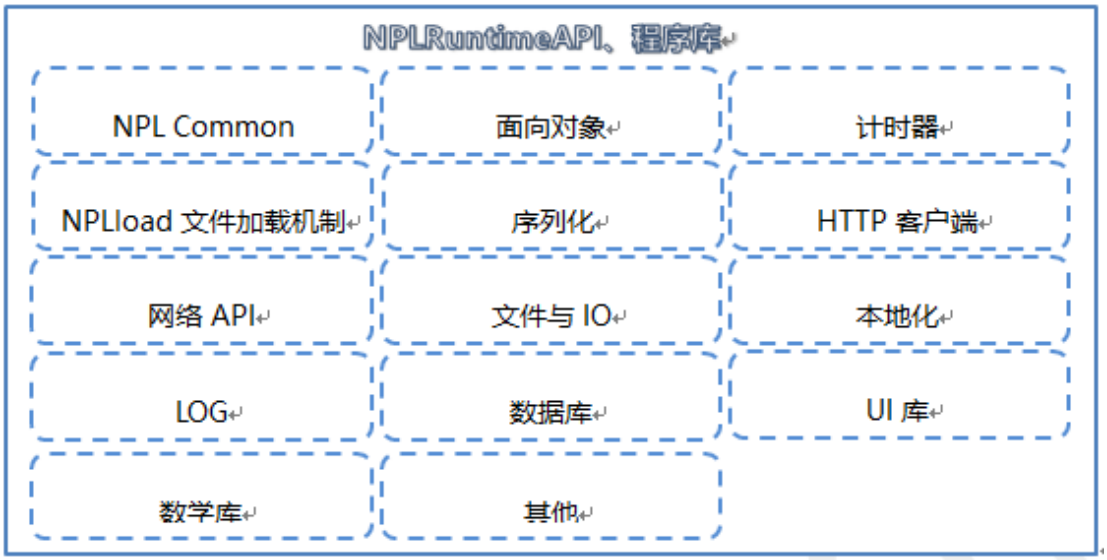

图 5.2 NPLRuntime API、程序库的组成

在 NPLRuntime 体系中，根据语言环境，API 又可以总体分为两类，即 lua 部分和 C++部分。那就让我们一一道来。

5.1 Lua

5.1.1 NPL Common Libraries

NPL Common Libraries 包含了一个丰富且开源的函数库集合，包括：io, networking, 2d/3d graphics, web server, data structure, 和许多其他软件框架。

主要库：

script/ide/commomlib.lua 和 script/ide/System/System.lua：用作最小化服务端应用开发。

script/ide/IDE.lua 和 script/kids/ParaWorldCore.lua：用作全面的重量级 2d/3d 应用开发。

IDE 中所有通用库：common_control.lua、commonlib.lua、Debugger/NPLProfiler.lua、Locale.lua、script/lang/lang.lua、gui_helper.lua、headon_speech.lua、event_mapping.lua、action_table.lua、os.lua、object_editor.lua、sandbox.lua、ParaEngineExtension.lua、AudioEngine/AudioEngine.lua、System/Core/Core.lua、System/Windows/Window.lua

MCML 相关库：script/kids/3DMapSystemApp/mcml/mcml.lua 、script/kids/3DMapSystemApp/mcml/PageCtrl.lua

重要文件说明：

文件	**script/ide/commonlib.lua**
概述	包含基本的调试、序列化等等 API 函数
	函数
function algorithm.sort_by_predicate(input, predicate_func)	通过谓词排序，将输入的列表调整为合适的顺序，对于那些通过和没有通过谓词测试的元素，其本身顺序是不变的。换句话说，只会改变输入的 input 参数表中元素的顺序，而不会改变元素本身内部的顺序
local function quicksort(t, compareFunc, start, endi)	快速排序，虽然是一个局部函数，但是通过 algorithm.quicksort = quicksort 将之加入到了全局表中。之所以写为局部函数是因为快速排序通过函数递归实现
function commonlib.GetUIObject(name, parent)	通过一种指定的用“#”号分隔的名称来获取 UI 对象，例如 childname#childname#childname
function commonlib.getfield(f, rootEnv)	从 rootEnv 中迭代查找所有的 f，f 为一个字符串
function commonlib.setfield(f, v, rootEnv)	将变量 v 设为 f
function commonlib.gettable(f, rootEnv)	获取一个表 f
function commonlib.createtable(f,	获取或者创建一个表 f，并用初始化参数

init_params, rootEnv)	init_params 将它初始化
function commonlib.ResetModelAsset(_obj, assetfilename)	重置模型资源
function commonlib.MetaClone(obj)	使用元表克隆一个对象，它实际上并没有复制参数。
function commonlib.deepcopy(object)	这个函数返回一个指定表的深度赋值对象，该函数也可以将元表复制到新表中。
function commonlib.copy(object)	除了不能复制原表以外和 commonlib.clone 一样
function commonlib.partialcopy(dest, src)	该函数就简单地将 src 赋值给 dest，但是会将 src 中的值复制到 dest。多数情况下用于表复制，只会将 src 中 dest 不存在的元素复制过去
function commonlib.mincopy(dest, src)	该函数就简单地将 src 赋值给 dest，但是会将 src 中的值复制到 dest。同样用于复制表，但是它会根据字段名字来判断元素在 dest 中存不存在
function commonlib.partialcompare(dest, src, tolerance)	将 src 与 dest 中的所有字段一一进行比较，如果它们相等，则返回 true。否则，返回 nil 或 false
function commonlib.compare(dest, src, tolerance)	严格地将所有字段一一进行比较
function commonlib.resize(t, size, v)	将一个表调整为一个新的大小，它确保所有新元素都为 nil
function table.resize(t, size, v)	同上

function commonlib.removeArrayItem(t, nIndex)	移除表中的一个元素
function commonlib.insertArrayItem(t, nIndex, item)	将一个表元素插入表中
function commonlib.removeArrayItems(t, callback_func)	从数组表中删除所有与符合某个标准的所有元素
commonlib.swapArrayItem(t, nIndex1, nIndex2)	交换一个表中的两个元素
function commonlib.moveArrayItem(t, nIndex1, nIndex2)	将表中 nIndex1 处的元素移到 nIndex2 处
function commonlib.cleartable(tab)	将传入的表清空
function utf8.len(unicode_string)	返回 UTF8 编码中字符的数量
function utf8.sub(ustring, nFrom, nTo)	除了 nFrom，nFrom 是指字符，而不是字节外与 string.sub()类似
function commonlib.SearchFiles(...)	搜索文件
function commonlib.Absolute(num)	获得传入的 num 的绝对值
function commonlib.tonumber(s)	将字符串转换为数字
function commonlib.tolower(o)	使所有表中字段(包括子表字段)小写，这对于不区分大小写的表非常有用
function commonlib.GetEmptyList(source_list,goal_num,bSort)	从列表中随机搜索若干项
function	在一个范围内随机选取一定的数目

commonlib.GetRandomList(source_num,goal_num,bSort)	
function commonlib.GetRandomIndex(range,excludeindex)	产生随机数
function commonlib.module(modname, ...)	定义一个模块，用来替换全局环境中的_G
function commonlib.split(str,delimiter)	用 delimiter 来拆分字符串 str
文件	script/ide/System/System.lua

5.1.2 面向对象

所有的 NPL 库的源代码均采用面向对象的方式写成。面向对象 API 函数主要是提供类继承和与 c++类似的类构造函数机制。

重要文件说明：

文件	**script/ide/oo.lua**
概述	它提供了类继承和与 c++类似的类构造函数机制。
	函数
local function search (k, plist)	在表“plist”中查找父类中的第 k 个值，并返回第一个有效的值
function commonlib.multi_inherit(...)	多继承
function commonlib.inherit(baseClass, new_class, ctor)	创建一个继承一个基础类的新类，新类有 new(),_super,isa()函数 @param baseClass:新类所继承的基础类，

	可以为 nil，表明没有基础类 @param new_class:nil 或者一个原始表 @param ctor:nil 或者一个构造函数
function commonlib.add_interface(target_class, interface_class)	这个函数能够被同一个有着不同接口类的目标类多次调用。它从接口类复制所有目标类中不存在的 string_key, value pair 到目标类，使用这个函数比多继承和单继承快，因为在目标类的元表中包含着所有接口函数
function commonlib.enable__gc(target_class, gc_function)	当对象需要被垃圾回收，可以调用这个函数来使能_gc 函数

5.1.3 NPLload 的文件加载机制

在 NPL 代码中，我们能够使用像 NPL.load("script/ide/commonlib.lua")这样的 load 机制去加载一个文件，所有加载的文件均会被编译。NPL.load 为我们自动完成下列几件事情：

- 从 NPL zip 文件包或根据定义在 npl_packages 中的 NPL 搜索路径找到正确的文件，并且在可行的情况下自动使用预编译成二进制版本。
    a) 支持相对路径，但只建议用于私有文件，如：NPL.load("./A.lua") or NPL.load("../../B.lua")
    b) 当使用相对路径时，文件名可省略，*.npl 和*.lua 文件都将被检索，如：NPL.load("./A") or NPL.load("../../B")
- 自动解决递归。（例如在文件 A 中 load 文件 B，同时在文件 B 中 load 文件 A）
- 编译还没有编译完成的脚本文件。
    a) 如果文件扩展名是*.npl，将调用 NPL 元编译器。

- 当代码被编译完成，它将在保护模式下使用 pcall 来执行代码块。在大多数情况下，这意味着有新代码加入全局或者对外输出表，所以我们之后能够使用它们。
- 如果有的话，返回对外输出的文件模块或 nil。如果模块没有找到返回 false。

代码注入模块

在 NPL/lua 中，表和函数均是第一类对象，在一个应用的生命周期中，我们使用一种非常可靠的代码注入模块去管理所有动态加载代码。

为了注入新代码，我们使用诸如 commonlib.gettable，commonlib.inherit 这样的方法。开发者必须确保他们注入的代码有统一的名字（例如 CompanyName.AppName.ModuleName 等等）。换句话说，不要污染全局表。

为了引用这些代码，我们需要像上一节说到的使用 commonlib.lua 一样调用 NPL.load 来在文件作用域中创建一个本地存根变量。请注意，由于 NPL.load 的次序，存根可能在实际代码注入之前创建。

文件基础模块

文件基础模块使用包含源代码的文件名来储存对外输出对象。因此程序员不需要明确地在代码中指定对象命名空间，这样解决了以下几个问题：

- 用这种风格编写的代码独立于它包含的文件位置
- 代码看起来更加独立
- 同一个代码的多版本可以共存

定义文件基础模块

存在三种方法来定义一个文件基础模块，假如你有一个文件叫做 A.npl，你想对外输出一些它的对象：

第一种方法可以在文件被加载的情况下调用 NPL.export。

```
-- this is A.npl file
local A = NPL.export();
function A.print(s)
  echo(s);
end
```

第二种方法是简单地在最后一行代码中返回与当前文件相关的对象，如下所示。这同样是与

dreamTech

lua 兼容的方式，但是在有循环依赖的情况下不适用。

```
-- this is A.npl file
local A = {};
return A;
```

最后一种方法是一种高级的手动方法，简单地添加到_file_mod_表。

```
-- this is A.npl file
local A = {};
_file_mod_[NPL.filename():gsub("([^/]*$)", "").."A.npl"] = A;
```

正如你所看到的,文件基础模块简单的在一个隐藏的全局表中自动存储从模块的完整文件名映射到它的对外对象。如果更改文件名或文件位置,映射核心也在改变。这就是为什么你可以加载相同模块的多个版本,并可以让用户在一个给定的源文件选择使用哪一个。

模块循环引用

当编写依赖于每个文件的文件模块时，会存在循环依赖。一种变通方案是默认对外对象，这个对外对象总是一个空表，而不是一个局部表。

面向对象的类继承模块

如下所示：

```
-- base_class.lua
local derived_class = NPL.load("./derived_class.lua")
local base_class = commonlib.inherit(nil, NPL.export());
function base_class:ctor()
   derived_class:cyclic_dependency_test();
end
-- derived_class.lua
local base_class = NPL.load("./base_class.lua")
local B = commonlib.inherit(base_class, NPL.export());
function B:ctor()
end
function B:cyclic_dependency_test()
end
```

使用文件名加载文件基础模块

导入一个文件基础模块，需要调用 NPL.load(modname)。NPL.load 是一个非常通用的函数，能够用来加载标准源文件或者文件基础模块，并且返回对外模块对象。

NPL.load(modname)将自动在模块按照下列次序查找源文件，直到查找到为止。在 NPL.load 中使用模块名不如使用明确的文件名那么高效，因为每次被调用的时候都需要进

行搜索，所以应该只在文件加载的时候使用它，例如在一个文件的开头。

查找次序：

- 如果在代码中从 npl_packages/[parentModDir]/调用 NPL.load
  a) try:
     npl_packages/[parentModDir]/npl_mod/[modname]/[modname].lua|npl
  b) try:
     npl_packages/[parentModDir]/npl_packages/[modname]/npl_mod/[modname]/[modname].lua|npl
- try: npl_mod/[modname]/[modname].npl|lua
- try: npl_packages/[modname]/npl_mod/[modname]/[modname].npl|lua

加载文件夹

我们还可以使用一个以“/”结尾的文件夹名称调用 NPL.load。在默认情况下，加载文件夹只会找到相关文件夹并且将之加入全局搜索路径。然而，如果文件夹中包含着一个叫做 package.npl 的文件就会有例外。如果 package.npl 中指定 searchpath = false，那么相应文件夹将不会加入搜索路径。

5.1.4 Timer

Timer 属于系统库，其核心文件是 timer.lua。这个文件包含许多计时器函数，提供了在指定间隔执行一个方法的机制。它只是包装了原生的 NPL 计时器函数，我们可以创建任意数量的计时器，只存在一个高精准的真实计时器来区分所有计时器对象实例。在场景重置时虚拟计时器不会被清除。

重要文件说明

文件	script/ide/timer.lua
	属性
id	表示计时器 id，必须独一无二

dreamTech

callbackFunc	回调函数
dueTime	调用回调函数之前要延迟的时间
period	调用回调函数之间的时间间隔，指定为 nil 表明禁用周期信号
enabled	是否启用计时器
lastTick	上一次激活调用 ParaGlobal.timeGetTime()的返回值
	函数
function Timer.GetNextTimerID()	获得下一个计时器 id
function Timer:Change(dueTime,period)	调整计时器
function Timer:Enable()	调用该函数启动计时器
function Timer:Tick(nTickCount)	这个函数被计时器管理器调用来处理时间，设置时针计数，如果计时器被激活，将返回 true。这个函数定期地用新的时针计数调用
function Timer:GetDelta(max_delta)	获取 delta 参数值
function Timer:Activate()	激活回调函数
TimerManager.Start()	为所有的子计时器创建一个全局计时器
function TimerManager.Stop()	终止计时器
function TimerManager.Clear()	清除所有计时器
function TimerManager.AddTimer(timer)	调用这个函数可添加一个计时器到计时器池，或者改变计时器设置，它将自动把 timer.enabled 设置为 true
function TimerManager.RemoveTimer(timer)	通过 id 删除指定的计时器
function TimerManager.DumpTimerCount()	转存计时器信息

function TimerManager.OnTimer()	ParaEngine 全局高精准计时器
function TimerManager.GetCurrentTime()	获取当前时间
function TimerManager.SetInterval(func, milliSecond)	等待指定的时间后，执行指定的函数。它也将每间隔一定的时间间隔继续执该函数
function TimerManager.SetTimeout(func, milliSecond)	创建一个计时器对象，暂停一次之后调用指定函数

5.1.5 序列化

序列化(Serialization)是指将对象的状态信息转换为可以存储或传输的形式的过程。在像 java JVM 等等环境中，序列化是一种必不可少的对象操作，在 NPL Runtime 中，我们同样也有相应的序列化函数支持。其核心文件是 serialization.lua。

重要文件说明：

文件	**script/ide/serialization.lua**
概述	序列化 API 函数
函数	
function commonlib.serializeToFile(file, o)	将一个表序列化到当前文件:函数和用户数据被导出为 nil 值
function commonlib.serialize(o, bBeautify)	序列化成字符串
function commonlib.serialize_in_length(o, nMaxLength)	序列化成字符串，它将在字符串长度大于 nMaxLength 之后停止

function commonlib.serialize_compact(o)	使用本地 API 的最快序列化方法
function commonlib.serialize_compact3(o)	和 commonlib.serialize 一样，但是它删除了所有的/r/n 和注释，因此更紧凑
local function serialize_compact2(o)	和 commonlib.serialize_compact 一样，但是它删除了所有括号字符串，因此更紧凑
function commonlib.LoadTableFromFile(filename)	返回一个由指定文件创建的表
function commonlib.LoadTableFromString(body)	根据指定字符串，调用 NPL.LoadTableFromString(body) 并返回相应表
function commonlib.SaveTableToFile(o, filename, bBeautified)	将指定表中的信息保存到一个文件中
function commonlib.serialize2(o, lvl)	序列化成字符串，此序列化有一个良好的组织和易读的表结构
function commonlib.WriteTextToFile(o, filename)	转存文本和行信息，将自动为我们创建目录

5.1.6 HTTP Client

NPL Runtime 支持以下几种 url 请求：

- 下载文件或发出标准请求
- 使用“-I”选项只获取标题
- 使用 http post 从 KV 对发送
- 使用 http post 发送多重二进制表单

- 发送二进制数据
- 将表单作为 json 字符串发送

此外，我们还可以通过设置不同的参数实现 PUT 请求和 DELETE 请求：

HTTP PUT 请求：

```
System.os.GetUrl({
     method = "PUT",
     url = "http://localhost:8099/ajax/log?action=log",
     form = {filecontent = "binary string here", }
}, function(err, msg, data)          echo(data)  end);
```

HTTP DELETE 请求：

```
System.os.GetUrl({
     method = "DELETE",
     url = "http://localhost:8099/ajax/log?action=log",
     form = {filecontent = "binary string here", }
}, function(err, msg, data)          echo(data)  end);
```

重要文件说明：

文件	**script/ide/System/os/GetUrl.lua**
概述	帮助类获取或发布 url 内容，在此文件中没有提供任何进步函数，但是这些函数在只获取 HTTP 大文件的标题时非常有用
函数	
function Request:init(options, callbackFunc)	初始化请求，包括设置 id，操作类型等等
function Request:SetResponse(msg)	设置响应信息，会对 msg.header 和 msg.data 的类型进行判断，
function Request:InvokeCallback()	调用回调函数
function CallbackURLRequest__(id)	根据 id，用相应的请求设置相应信息和回调函数
local function GetUrlOptions(url, option)	获得 url 相关选项

function os.GetUrl(url, callbackFunc, option)	返回给定 url 的内容
function os.SendEmail(params, callbackFunc)	通过 smtp 协议发送一个电子邮件信息

5.1.7 网络 API

在 NPL 中，我们不需要通过创建任何套接字或者编写一个消息循环的方式来和远程计算机进行通讯。NPL 以一种非常高效的方式管理所有通讯，用户仅仅只需要一个函数就可以建立链接，认证和与远程计算机通讯。

在默认情况下，NPL 在启动时不会监听任何端口，除非我们调用一些库，如 WebServer。在 NPL 中启用网络，我们需要调用：

```
-- a server that listen on 8080 for all IP addresses
NPL.StartNetServer("0.0.0.0", 8080);
```

在客户端同样也需要调用 NPL.StartNetServer。

```
-- a client that does not listen on any port. Just start the network
interface.
NPL.StartNetServer("127.0.0.1", "0");
```

如果你想要即使超过 20 秒没有发送消息仍然维持已经认证好的连接，你可能想生成一些周期性的 ping 消息。因为如果不这样，服务器端和客户端就会认为 TCP 链接已经终止，你将丢失在服务器端的身份认证。尽管 NPL 允许在全局范围上我们用 KeepAlive 属性配置服务器端应用水平上的 ping 消息，但是不建议在服务器端启用它或者将其值设置成一个非常大的值（例如几分钟）。推荐的方法是，客户端应该定期地对所有远程进程用 ping 消息进行初始化。

重要文件说明：

文件	**script/ide/Network/StreamRateController.lua**
概述	流速度控制器可以用来限制流量例如聊天消息，我们刻印通过美妙平均消息来限制，等等。它提供了两个消息队列用作测量。最重要

	的特征是提供了一个叫做 history_length 的参数，允许消息速度达到一个临时的高峰值。
属性	
name	名称字符串，只用在当队列满时写 log
max_messages_in_queue	在历史队列中最大的消息数，如果为 nil，则在构造时通过 math.ceil(history_length*max_msg_rate + 1)计算获得
history_length	在几秒钟内，我们将记录这段长度的所有历史消息，然后通过平均来计算当前的消息速率
max_msg_rate	每秒钟允许的最大消息数量，这个值是根据 history_length 计算的，为 nil 表示没有限制
函数	
function StreamRateController:Reset()	清除所有统计资料，包括 start_time、total_msg_count、total_msg_size、max_messages_in_queue、queue
function StreamRateController:GetLastMessageTime()	获得上一条消息时间
function StreamRateController:AddMessage(nSize, handler_callback)	将一条消息加到数据大小 @param nSize:消息数据大小，默认为 1 @param handler_callback:当消息允许被处理时调用的函数
文件	script/ide/socket/socket.lua
概述	luaSocket 是一个 lua 扩展库，由两部分组合成：为 TCP 和 UDP 传输层提供支持的 C 内核和一套为处理互联网的应用程序所需要的功能提供支持的 lua 模块。此文件是基于 luaSocket 的 tcp/udp

套接字	
函数	
function connect(address, port, laddress, lport)	与传入的地址和端口建立链接
function bind(host, port, backlog)	与指定主机绑定
function choose(table)	
文件	script/ide/socket/url.lua
函数	
function escape(s)	将字符串编码转义成十六进制表示
local function make_set(t)	保护一个路径段，防止它干扰 url 解析
function unescape(s)	将字符串编码转义成十六进制表示
function parse(url, default)	解析一个 url 并根据 RFC 2396 返回一个包含所有部分的表
function build(parsed)	从解析完的组件重新构建一个已经解析的 URL，相当于解析 url 的逆过程
function absolute(base_url, relative_url)	根据 RFC 2396 从一个基本、个相对 URL 构建绝对的 URL
function parse_path(path)	将指定路径分为片段
function build_path(parsed, unsafe)	从片段构建路径

5.1.8 文件与 IO

文件与 IO 提供如查找文件、读取文件等等 API 函数。

重要文件说明：

文件	**script/ide/Files.lua**
函数	

function Files.GetFileExtension(file)	获取文件扩展名，没找到扩展名返回 nil
function Files.GetLuaFileSystem()	获取 lua 文件系统，存在 lfs 直接返回，不存在，也就是第一次使用，会调用 luaopen_lfs() 创建 lfs
function Files.IsAbsolutePath(filename)	判断是否为绝对路径
function Files.GetDevDirectory()	获取 dev 目录
function Files.Find(output, rootfolder,nMaxFileLevels, nMaxFilesNum, filter, zipfile)	返回当前文件夹的子文件夹
function Files.SearchFiles(output, sInitDir, sFilePattern, nMaxFileLevels, nMaxNumFiles, listFile, listDir, zipfile)	在给定路径中搜索文件和目录，并将结果存在一个表中返回
function Files.DeleteFolder(foldername)	删除文件夹
function Files.TouchFolder(foldername)	把文件夹的修改时间更改为当前时间，方式是通过创建一个临时文件然后删除它

5.1.9 本地化

本地化实现了对本地语言系统的匹配功能，使用 gettext 方法，涉及到两个文件，*.po 是翻译源文件，里面储存了项目中所有待翻译的字符串和翻译后的结果；*.mo 文件是 po 文件编译后二进制文件，真正读取翻译信息的时候是从 mo 文件中读取的。当 gettext 获取到本地的语言编码信息，会使用对应的*.mo 文件，如果没有找到*.mo 文件，那么会使用默认编码。

重要文件说明：

文件	script/ide/Locale.lua
函数	
function Locale.EnableLog(bEnable)	是否启用错误日志，默认启用。
function Locale.EnableLocale(bEnable)	使或者禁止局部字符串全局可见。当传入参数为 false，查询字符串将始终返回字符串本身
function Locale.AutoLoadFile(file)	如果文件中的语言环境和当前语言环境匹配时自动加载文件
function Locale:GetByName(name)	通过名字获得给定的译文，如果不存在，将返回 nil
function Locale:new(name)	创建或获得给定的译文
function Locale.prototype:EnableDebugging()	允许调试
function Locale.prototype:RegisterTranslations(locale, func)	注册翻译函数
function Locale.prototype:SetStrictness(strictness)	
function Locale.prototype:GetTranslationTable()	这个函数多被“nocheck”译文使用来注册一个新的译文
function Locale.prototype:Reset()	清除所有译文，这个函数可以在运行环境中语言发生改变后调用
function Locale.prototype:GetTranslationNoCheck(text)	检索文本译文

function Locale.prototype:GetTranslationStrict(text, sublevel)	
function Locale.prototype:GetTranslation(text, sublevel)	获取译文
local function initReverse(self)	初始化反转操作
function Locale.prototype:GetReverseTranslation(text)	获取反转的译文
function Locale.prototype:GetIterator()	获取迭代器
function Locale.prototype:HasTranslation(text, sublevel)	判断相应的文本是否有译文
function Locale.prototype:HasReverseTranslation(text)	判断相应的文本是否有反转译文
function Locale.prototype:GetTableStrict(key, key2)	
function Locale.prototype:GetTable(key, key2)	
function Locale.prototype:Debug()	

5.1.10 Log

NPL Runtime 提供了 4 种先进的 log API 函数：

- commonlib.log for ad-hoc log
- commonlib.applog for date formated application log
- commonlib.servicelog for date formated multi-file log with append
- commonlib.logging for 5 level based logging with custom listener(appender)

一条 log 消息具有与之相关的级别，定义在系统中的级别，按严重程度从高到低的顺序是：FATAL, ERROR, WARN, INFO, DEBUG, TRACE。log 记录器的级别能够设置在配置文件中，是一个非常重要的信息。一条消息只有在级别等于 log 记录器的级别或者比之更严重才能通过 log 记录器。

重要文件说明：

文件	**script/ide/log.lua**
	函数
function log(text)	用高级日志代替本机日志，最常见的 log 函数，在其中会对传入的参数 text 的类型进行判断，如果为 string，则直接写入 log 文件，不为 string，则序列化后写入 log 文件
local function GetLogTimeString()	获得标准时间格式字符串，用函数 ParaGlobal_timeGetTime()获取当前时间，返回 the date_str, time_str, and the tick count
function commonlib.log.log_long(str, maxLength)	它通过将输入分成多个部分来支持超过 1024 字节的 log 字符串
function commonlib.log.GetLogPos()	获取当前 log 文件位置，实现方式是通过返回函数 ParaGlobal.GetLogPos()
function	在两个 log 位置之间获取日志文本

commonlib.log.GetLog(fromPos, nCount)	@param fromPos: bytes 位置， 如果为 nil, 默认从第 0 个 byte 开始 @param nCount: bytes 数， 如果为 nil, 默认到 log 文件的末尾
function commonlib.applog.log_long(str, maxLength, level, use_location)	它通过将输入分成多个部分来支持超过 1024 字节的 log 字符串
local function service_log_long(logger_name, str, maxLength)	它支持长度超过 1024 字节的 log 字符串
function commonlib.servicelog.GetLogger(logger_name)	获得一个给定的 log 记录器
function logging.GetLogger(name)	创建后获取一个带有给定名称的 log 记录器
function LOG.SetLogLevel(level)	设置 log 消息级别

5.1.11 Table Database

Table Database 是在 NPL 中实现的默认并且推荐使用的数据库引擎，它可以在没有任何外部依赖和配置的情况下使用。它是一个无模式、服务器较少的 NoSQL 数据库，能够在多个 NPL 线程中处理大数据，具有极其直观的表式 API、自动索引和极其快速的搜索。此外，它将数据和算法保存在一个系统中，就像人脑的工作方式一样。

Table Database 的优点：

Table Database 可同时实现满足以下当前的 SQL/NoSQL 不能同时满足的需求：

- 保持数据和计算相近，就像我们的大脑。
- 在没有模式的情况下存储任意数据

- 基于查询使用的自动索引
- 提供阻塞和非阻塞 API
- 不使用网络连接来实现本地性能的最大化的多线程架构
- 能够在本地存储数百个 Giga 字节的数据
- 为 NPL table 提供本地文档存储格式
- 就像操作标准的 npl/lua table 一样的超级简单的客户端 API
- 易于设置和部署 NPLRuntime
- 不需要服务器配置，调用客户端 API 将在第一次使用时自动启动服务器

实现细节：

- 每个数据表（数据集合）都存储在一个轻量级数据库文件中
- 每个数据库文件包含一个从对象-id 到对象-表的索引映射
- 每个数据库文件都包含一个模式表，用于说明所使用的其他索引键
- 每个数据库文件都包含一个 key 对应每个自定义索引键的对象 id 表
- 所有的数据库 IO 操作都是在一个专用的 NPL 线程中进行的

5.1.12 UI 库

UI 库处在构建应用程序的底层，作为搭建程序界面的核心要素，UI 库提供了大量创建 UI 界面对象的基本要素，如按钮、容器、线条等等。当创建 2DUI 对象时，可通过创建 ParaUI 对象， 在 NPL 脚本中创建，并使用 C++进行组织的。另一种可提供的方式是绘制自己的 ParaUI 对象，即使用绘图 API 所提供的方法，在 NPL 脚本中执行相应的 API 函数。

另外，如果使用在 NPLRuntime 中的高级 UI 库，可以更加轻松地创建 2DUI 界面。可基于 ParaUI 对象控件实现或者基于 System.Window.UIElement 中绘画 API 实现。

NPLRuntime 中 UI 库

在 NPLRuntime 中定义了许多基本的 UI 元素，其中最基础和最经典的是按钮(button)、容器（container）、系统字体(System Fonts)和 GUI 图片（GUI Images）。

Button

dreamtech

下面函数实现创建一个带有纹理和文本的按钮：

```
NPL.load("(gl)script/ide/System/System.lua");

local clickCount = 1;
local function CreateUI()
   local _this = ParaUI.CreateUIObject("button", "MyBtn", "_lt", 10, 10,
64, 22);
   _this.text= "text";
   _this.background = "Texture/alphadot.png";
   _this:SetScript("onclick", function(obj)
      obj.text = "clicked "..clickCount;
      clickCount = clickCount + 1;
   end)
   _this:AttachToRoot();
end
CreateUI();
```

在这个函数中，定义了_this 变量，通过调用 ParaUI 中函数 CreateUIObject，并传入所创建的名称以及设置了该对象的大小。紧接着通过变量赋值的方式设置了前文所说的文本和纹理，即_this.text= "text"，_this.background = "Texture/alphadot.png"。接下来最重要的部分，设置按钮的点击事件：借助 lua 语言的特性，将函数作为一个变量值，创建对应的点击事件。

Container

Container 类似 java 中的 Panel，作为一个容器，容纳他 UI 元素。

```
    local _parent, _this;
        _this = ParaUI.CreateUIObject("container", "MyContainer", "_lt",
10, 110, 200, 64);
        _this:AttachToRoot();
        _this.background = "Texture/alphadot.png";
        _parent = _this;

        -- create a child
        _this = ParaUI.CreateUIObject("button", "b", "_rt", -10-32, 10, 32,
22);
        _this.text= "X"
        _this.background = "";
        _parent:AddChild(_this);
```

在这段代码中，创建了一个容器，并且在其中添加了一个按钮。

所谓 UI 库，即是定义了大量类似上述函数的脚本文件组合，我们通过事先定义并封装好支持各种创建 UI 元素的方法，如具有不同风格的、触发不同点击事件的按钮等等基础的 UI 组成元素，将使用接口暴露出来，这样应用程序或者用户不需要去关注函数的内部具体实现，只需根据暴露出来的接口要传入相应的参数，即可非常便捷的搭建应用的 UI 系统。如在上面所说的创建 button 函数中，将这个函数封装起来，定义成一个计数按钮，并对外暴露相应的接口，这样就成了一个非常基本的 UI 元素，用户可直接调用这个函数，实现创建一个计数按钮。当很多类似这样的函数组合起来，就能够构成我们的 UI 库。

此外，UI 库中比较重要的组成部分还有对系统字体的支持的 AppendTextValue 函数，该函数应该在任何其他 UI 控件被创建之前调用，对每种大小、粗细、名字不同的字体，将会创建一个不同的内部临时图像文件去渲染。另一个就是对图片的处理，ParaUI 库中提供了图像分块的特征操作。支持选用一张图片中特定的区域，并将之分为 9 块，去除四个角，得到我们所想要的大小的图片。这个 API 函数很好地支持了创建大小和外观丰富的按钮。

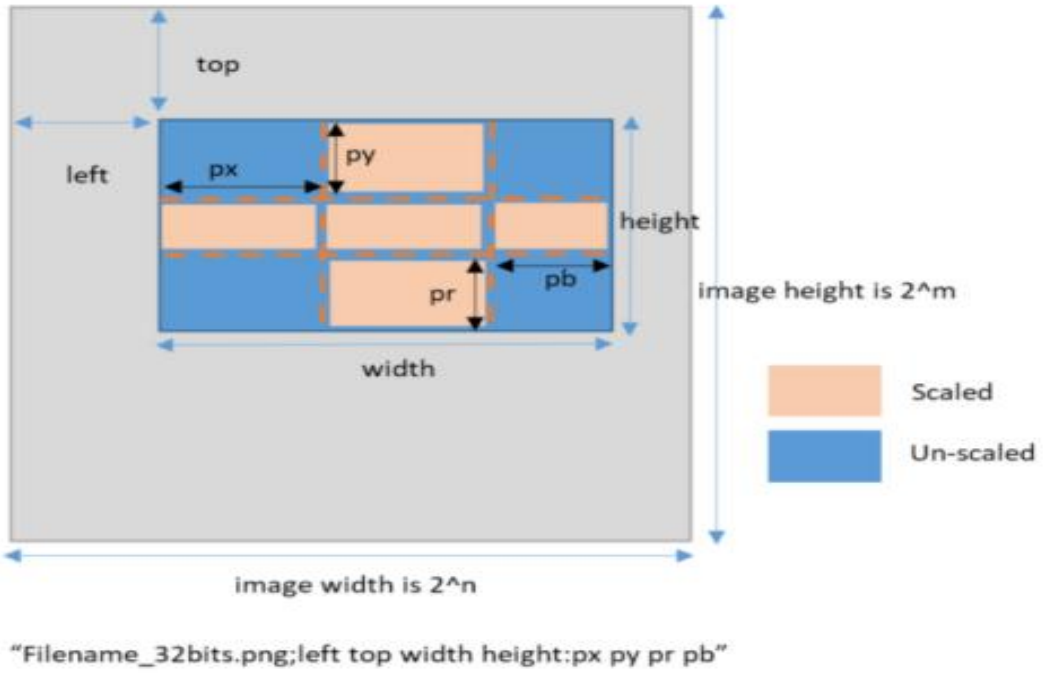

图 5.3 UI 图片处理

UI 系统中重要 API 函数说明：

ParaUIObject ParaUI::CreateUIObject(const char* strType, const char * strObjectName,const char * alignment, int x, int y, int width, int height)

CreateUIObject 函数可以说是整个 UI 系统中最重要的函数，button、editbox、imeeditbox、scrollbar、container、text、listbox、slider 等 UI 元素的创建全都要调用该函数。

参数 strType：新对象的种类，能够是"container", "button", "scrollbar", "editbox", "imeeditbox","slider", "video", "3dcanvas", "listbox", "painter" and "text" 。"container"是唯一可以包含其他对象的控件类型。

参数 strObjectName：新对象的名称

参数 alignment：对象对齐方式，可以是：_lt（对齐到屏幕的左上）、_lb（左下）、_ct（中央）、_ctt（中央顶部）、_ctb（中央底部）、_ctl（左中）、_ctr（右中）、_rt（右上）、_rb（右下）、_mt（中间顶部）、_ml（中间左部）、_mr（中间右部）、_mb（中间底部）、_fi（填充窗口）

参数 x：屏幕坐标 x

参数 y：屏幕坐标 y

参数 width：坐标宽度

参数 height：坐标高度

在该函数中，对传入的参数 strType 进行类型判断，根据不同的对象类别，调用不同的创建对象函数，如 `if(strcmp(strType, "button")==0)`，则 `pNewObj= new CGUIButton();`

5.2 C++ NPL Runtime API

在 C++中的跨平台模块可以分为以下几个部分，这些模块全都通过 NPL scripting API 暴露出来，因此能够通过 NPL 来调用。

NPL 脚本引擎：

- NPL 状态机（NPL state），（或者说 NPL 虚拟代码环境）：一个单 NPL 线程，有自己的内存分配器，并且管理它加载的所有文件。
  - NPL 状态机能够加载 NPL/Lua 脚本或者 C++中 dll 文件。

- Mono 状态机：能够加载 C#中 dll 文件。

- NPL 网络：通过 NPL HTTP/TCP 链接管理所有本地或远程的 NPL 状态机。

图形引擎：

所有 NPL 对象必须在 NPL 主线程中创建，这和渲染线程是一样的。2D/3D 引擎都是面向对象的。所有 2D 对象均用一个父/子树（parent/child tree）组织。3D 对象用四叉树（quad-tree）附加父/子树组织。

视频/音频渲染器：

一个静态/动态缓冲区，纹理，绘制 3d/2d api,字体等等。

- DirectX 完全支持渲染器。
- OpenGL 渲染器：一个跨平台的渲染器

绘画引擎（PaintEngine）：

它就像一个由 2D 引擎使用的 GDI（Graphics Device Interface）引擎，用来绘制 2D 或者简单的 3D 对象，例如线条，矩形等等。

2D 引擎：GUIBase 是所有 2d 对象的基类

- GUIRoot：所有 GUI 对象的根节点
- 此外还有 GUIContainer，GUIButton，GUIText 等等。

3D 引擎：BaseObject 是所有 3D 对象的基类

- ViewportManager:管理 2D 视窗，是我们能够渲染 3d 或 2d 对象
- SceneObject:所有 3D 对象的根节点。它管理所有对象例如增加，删除，查询，渲染，物理等等。
- TerrainTileRoot:它是一个有关所有 3D 对象的四叉树容器，用来根据它们的 3d 位置和当前摄像机的平截头体（current camera frustum）来快速搜索对象。
- CameraObject: 相机平截头体。几个衍生类有 AutoCamera 等等。
- MeshObject:3d 对象中静态三角网格对象的基类。
- MeshPhysicsObject: 它是一个物理的静态网格。
- BipedObject: 它代表一个动画对象。

dreamTech

- MinisceneGraph:它是 SceneObject 的简化版本，通常用来管理少数能够分别被渲染成 2D 纹理的 3d 对象。
- TerrainEngine:拥有一个高度图和多个纹理层的无限大地形。它是用动态水平细节算法渲染。
- BlockEngine:它管理 32000*32000*256 个方块的渲染。
  - BlockRegion:管理 512*512*256 个方块，这些方块都存在单个文件中
  - ChunkColumn:16*16*256
  - Chunk: 16*16*16，一个静态的可渲染对象。

物理引擎：

它为场景中的静态网格物体使用开源的 Bullet 物理引擎。方块物理是由方块引擎单独处理的。

资源管理：

通常每个静态的资源文件都是一个资源实体。AssetEntity 是系统中所有资源类的基类。资源类提供了数据和某些时候被用来渲染任何其他 2d/3d 对象的方法。所有资源在默认情况下均通过他们的 IO 磁盘或者远程网络异步加载。

TextureEntity:2d 纹理

MeshEntity:静态网格文件，如.x

ParaXEntity:动画网格文件，能够从.x，.fbx 中加载，音频，最大模型，数据库，字体等等。

EffectFile:着色文件，能够从 directX .fx 中加载。

IO 和 Util：

FileManager:管理所有文件和搜索路径

CParaFile:管理单个文件的读/写

math:2d/3d 数学库，如向量和矩阵

ParaScripting API：

将以上所有函数或模块暴露到 NPL 脚本环境。

dreamTech

5.2.1 Core C++ API

在 NPL 脚本中，以 Para 开始的 table，例如：ParaIO, ParaUI, ParaScene, ParaEngine 等等一般都是核心 C++ API。此外，NPL 表中的大多数函数也是核心 C++ API，比如 NPL.load, NPL.this 等等。在 C++中暴露了以下所有函数和模块到 NPL 脚本环境中：

- ParaNetwork
- ParaScene
- ParaUI
- ParaGlobal
- ParaAsset
- ParaEngine
- ParaWorld
- ParaTerrain
- ParaCamera
- ParaMovie
- ParaIO
- ParaAudio
- ParaMisc
- ParaBootStrapper
- global functions
- Config
- JabberClient
- Object Attributes
- Shaders

在这里不对这些模块展开说明，详情可见：https://codedocs.xyz/LiXizhi/NPLRuntime/modules.html

dreamTech

5.2.2 数学库

NPL Runtime 中数学库是对 c++中标准数学库进行包装，从而提供对 NPL 中特定操作的数学操作支持。NPL 中数学库主要文件和功能见下表：

数学库文件	概述
AABBPool.lua	aabb 池，在一个框架中创建大量的池对象时非常有用
bit.lua	
complex.lua	提供对操作复数的常见任务
find_bit_limits.lua	
fit.lua	
math3d.lua	一组独立且经常使用的三维数学函数，如矢量旋转等
matrix.lua	矩阵函数
matrix4.lua	该类封装了一个标准的 4x4 正交矩阵
MD5.lua	消息摘要算法
Plane.lua	在三维空间中定义平面
Point.lua	
Quaternion.lua	类封装一个标准的四元组{x，y，z，w}
Rect.lua	矩形
ShapeAABB.lua	3 维空间中的 AABB 模型
ShapeBox.lua	3 维空间中的盒子模型
ShapeRay.lua	
StringUtil.lua	字符串辅助操作函数
TEA.lua	Tiny Encryption 算法函数
vector.lua	2d 和 3d 向量
vector2d.lua	2d 空间中向量
VectorPool.lua	3d 向量池

NPL 的数学库主要是对 C++中标准数学库进行包装，若想详细了解数学库，我们需要去了解 C++中的数学库，并参考 script/ide/math 文件夹下的文件。

IdreamTech

6 UI 系统

6.1 UI

6.1.1 综述

2D GUI 是 ParaEngine 的重要特征，它是玩家和虚拟世界的控制链接。它从接收玩家从鼠标，键盘或者其他输入设备的信号，并激活相应的脚本去处理对应的输入信息，并对虚拟世界做出相应的改变。

在 NPLRuntime 中，提供了许多支持玩家和虚拟世界交互的方式。其中两个关键的特点是低门槛和可靠性。在可靠性方面，有着对虚拟元素进行操作的稳定架构。NPLRuntime 允许我们将不同层级的图片混合起来，这样能够使结果丰富多彩。此外，还提供了丰富的和 UI 引擎交互的 API 函数，使我们能够对 UI 对象的每一个非常微小的部分进行操作。在操作函数方面，有着许多不同种类参数的函数。

对于整个 UI 体系，可以用下图描述：

dreamTech

图 6.1 UI 体系结构

底层是 UI 库，包含按钮，容器等。 有了 UI 库就可以用于构造 UI 模型，NPL 用的是通用的 3 层 UI 模型： 背景-工艺-覆盖。 然后管理 UI 模型的方法是显示和逻辑分离的方法， 用 MCML 语言编写显示部分（也可以用脚本写 UI 显示，但是因为难以管理，所以不推荐）， 用 NPL 脚本控制 UI 逻辑。这种显示，逻辑分离的架构有点像网站开发的 MVC 开发模式。

6.1.2 UI 创建流程

1. 使用 ParaUI.CreateUIObject()创建一个对象。
2. 设置相关属性。
3. 将所创建的对象附加到某一个东西上面。（使用 AttachToRoot()函数将 UI 对象附加到 UI root 上，使用 AddChild()将新 UI 对象附加到旧 UI 对象上，作为其子对象。Root 对象是 UI 树的根，不可见，控制着整个 UI 系统的逻辑）
4. 重复上述过程

7.1.3 UI 对象类别

根控件对象：它是所有 GUI 对象的最上层容器，它收集所有输入设备的状态信息然后发送给它的子对象。同时，它也控制着子对象的渲染过程。

资源对象：它是一种与控制无关的对象，它包含着为渲染一类特殊控件的所有必要的资源。例如，如果我们想要交换按钮 A 和 B，我们不需要交换按钮只需要交换相应的资源对象。

脚本对象：它是一个脚本容器和激活器,通常被附加到一个资源对象上。

7.1.4 GUI 类层次结构

整个 GUI 系统的骨架及之间的关系可以总结为下图：

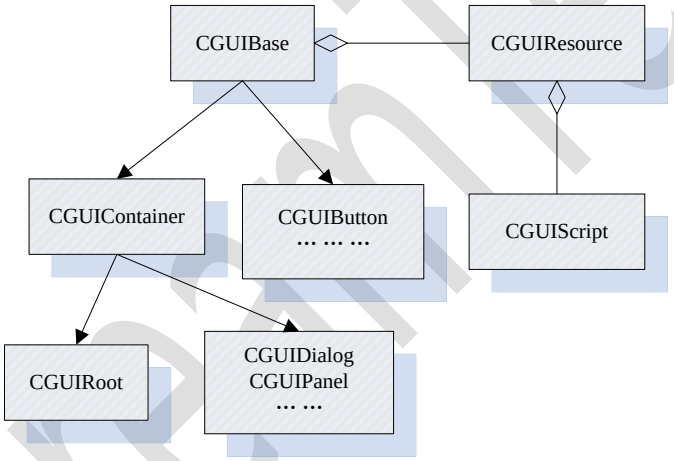

图 6.2 GUI 类层级结构

CGUIbase：所有 GUI 元素的基类，它定义了 GUI 对象的基本行为，如各种对象的 get、set 方法，bool CGUIBase::Equals(const IObject *obj)const 、bool CGUIBase::OnClick(int MouseState, int X, int Y)、bool CGUIBase::OnMouseLeave()等等。

CGUIResource：CGUIResource 类封装了渲染一个类的必要资源，描述一个元素的必要游戏数据，和游戏引擎交互的必要脚本从而达到轻易地交换相同类型的控件。如将一个背包中的元素拖动到活动元素表。这个类中包含了控制控件这种行为的代码。

CGUIScript：提供为 GUI 对象添加控制脚本的接口

CGUIContainer：所有能够拥有子对象的控件都必须继承 CGUIContainer。CGUIContainer 提供了一个容器应有的基本的函数和属性，例如：get/lost focus and mouse in/out。

CGUIRoot：这个类充当 GUI 系统的基本容器和控制器，它负责管理键盘和鼠标，将键盘鼠标还有其他事件调度到合适的位置，帮助管理 GUI 对象。

7.1.5 UI 资源架构

资源对象被设计成独立对象，意思是它们不会和某一对象有关联。要交换两个对象的展示效果非常简单，只需要把两个对象的资源对象交换。下图展示了资源对象的架构。

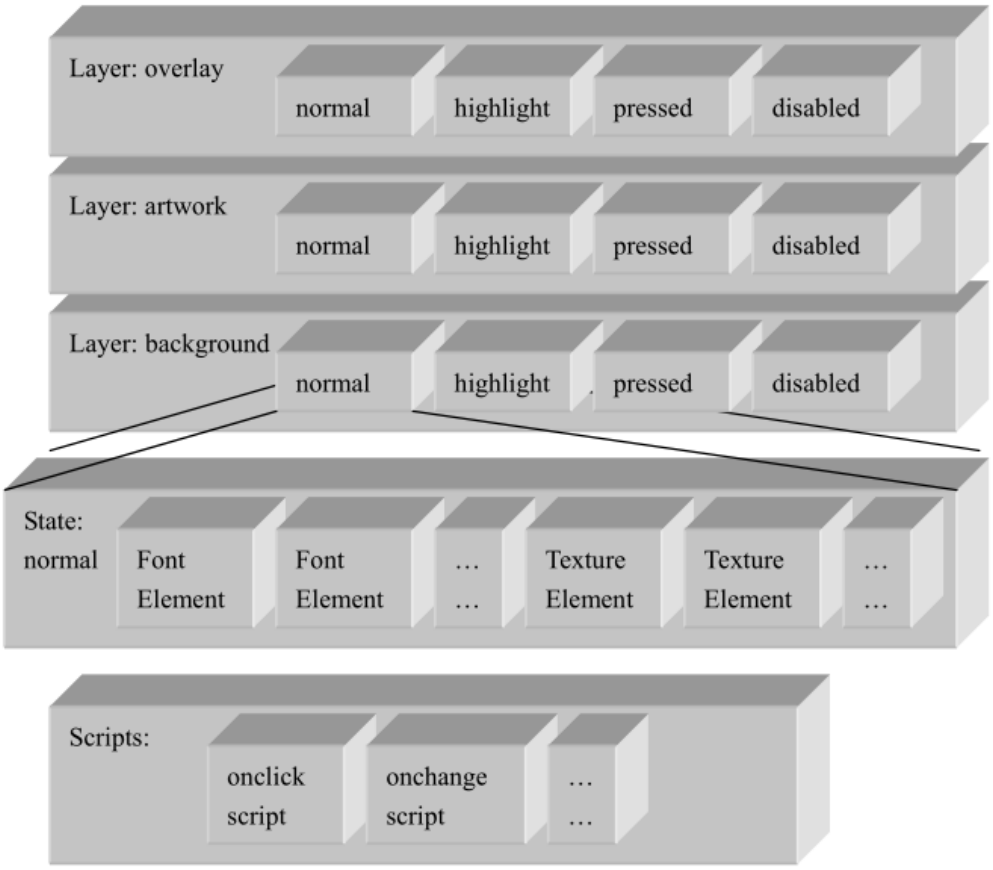

图 6.3 UI 资源架构

从上面的架构图中，我们可以看到 UI 资源架构包含三个层级：overlay，artwork 和

background。这三个层级都是用 *alpha blending* 绘制的。（详见 Direct X SDK 帮助文档）

层（layer）被用来将不同的纹理结合在一起，来达到特殊的效果。例如在下图中所展示的效果至少是由两个层级所结合起来的。支持不同的层级不仅仅使 UI 设计者能够将不同的纹理结合在一起，同时也减少了艺术效果方面的工作量。如果只有一个层，则必须为每个按钮制作这样的倒计时效果。如果存在多层，则只需要为一个按钮做出这种倒计时效果，然后根据对应的按钮事件，对不同的按钮实现这种效果。

图 6.4 按钮倒计时效果

覆盖层（overlay layer）：覆盖层处在工艺层之上，它展示对象的上层工艺。只有在明确设置它的时候才会存在。

工艺层（artwork layer）：工艺层在背景层之上，它是一个默认的层级。它包含着对象的默认工艺。在大多数情况下，使用该层就足够了。（当背景层和覆盖层存在的时候，引擎会自动绘制它们。因此将增加引擎的渲染时间且影响引擎的性能。因此只有在必要的情况下才会使用另外两个层级）

背景层（background layer）：背景层是处在最底端的一个层级。它用来展示背景工艺。只有在明确设置它的时候才会存在。

对于每一个层级，有四种状态：normal, highlight, pressed and disabled。

Normal 状态：一个对象的默认状态。

Highlight 状态：一个对象的 hightlight 状态。例如鼠标停留在一个按钮上的时候。

Pressed 状态：一个对象的 pressed 状态。例如一个按钮被鼠标点击的时候。

Disabled 状态：一个对象的 disabled 状态。一些对象当它们 disabled，它们将有不同的表现。

对于每一种状态，有许多字体元素和纹理元素。每个元素包含着有关渲染的许多信息。（更多信息见 NPL 参考文献）

dreamTech

一个对象的脚本同样也储存在 UI 资源架构体系中。对于 UI 资源架构体系中每一种事件，最多只有一个脚本实例。

7.1.6 常见属性

属性	描述
name	String 类型，对象的名字
x	number 类型，对象左上角的坐标
y	number 类型，对象左下角的坐标
width	number 类型，对象的宽度
height	number 类型，对象的高度
candrag	boolean 类型，对象是否能被拖动，默认为 false
receivedrag	boolean 类型，对象是否能接收一个拖动事件，默认为 false。尽管所有 UI 控件均有该属性，仍不建议在除 container 之外的控件中使用它
type	string 类型，只读，对象的种类。可以是“button”，“text”，“editbox”，“imeeditbox”和“container”
lifetime	number 类型，离对象被自动删除的帧数。如果为正数，lifetime 将在每帧自动减少。默认值为-1，意味着永久有效
parent	ParaUI 对象，gets 或 sets 对象的 parent
tooltip	string 类型，gets 或 sets 当前对象的提示信息。默认为“”。tooltip 是一种在帮助人们理解对象的含义或将人们注意力转移到对象上非常有用的弹出信息对话框
enabled	boolean 类型，enables 者 disables 指定对象。一个 disabled 对象是可见的但是不能接收任何事件，默认为 true
visible	boolean 类型，控制对象是否可见。一个不可见的对象是看不见但是

	任然能够接受事件，默认为 true

7.1.7 常见方法

方法	描述
IsValid	获取对象是否是一个有效 deGUI 对象。如果对象无效，用户将不能够使用这个对象，则结果不可预测。
AttachToRoot	将对象附加到 root 上
AttachTo3D	将对象附加到 3D 对象上
GetFont	根据给定的名字获取对象的字体
GetTexture	根据给定的名字获取对象的纹理
SetCurrentState	设置当前状态
SetActiveLayer	设置活动层

7.1.8 UI 对象实例

button

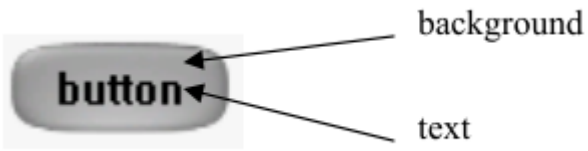

图 6.5 按钮

代表着 GUI 的按钮控件。

一个按钮能够使用鼠标点击。它能够接收以下事件： onclick,ondoubleclick, ondragbegin, ondragend, onmousedown, onmouseup, onmousehover

创建方法:

object= ParaUI.CreateUIObject("button","buttonname","_lt",50,20,600,400)

属性	
text	string 类型，对象的文本信息
background	string 类型，对象的背景

	图片文件路径+[";" +left+" " +top+" " +width+" " +height] "left"，"top "， "width"和"height"一起形成了一个矩形，表明了背景图片展示的区域。"[]"内均是可选的，如果没有指定，则表明整张图片 图片文件路径+[":" +left+" " +top+" " +width+" " +height] "；"表示截取图，" ：" 表示点九图，即保持四个角不变，对选中的区域进行扩展
	纹理对象
background	对象的背景纹理
	字体对象
text	对象的文本信息，用来在按钮上显示文字

container

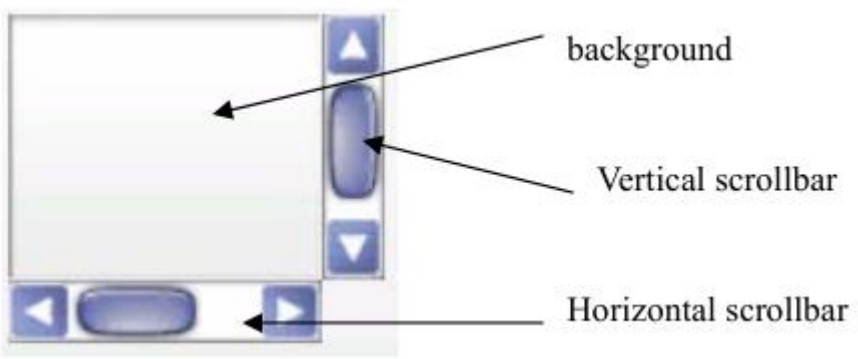

图 6.6 容器

表示一个 GUI 中的容器控件。它能够有子控件，一个容器有两个滚动条——垂直滚动条和水平滚动条。如果属性"scrollable"是 false 或者在容器中的控件完全在容器的客户区，那么滚动条将不会显示出来。

一个容器不接收任何事件。

创建方法：

object= ParaUI.CreateUIObject("container","containername","_lt",50,20,600,400)

属性

scrollable	boolean 类型，控制容器是否可滚动，默认为 false
scrollbarwidth	number 类型，容器的滚动条的宽度，默认为 25
fastrender	boolean 类型，决定是否在容器内使用快速渲染模式，默认为 true
background	string 类型，对象的背景
	方法
GetChild	获取容器的给定名字的第一个子对象（如果存在重复的名字），根据给定的名字返回 ParaUI 对象，如果对象不存在，返回 ParaUI 对象的 IsValid()方法的 false 返回值
GetChildAt	根据给定的索引获得子对象
AddChild	添加一个子对象到容器
	纹理对象
background	对象的背景纹理
	字体对象
在容器中不存在字体对象	

scrollbar

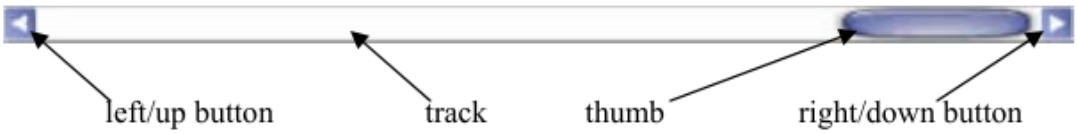

图 6.7 滚动条

表示一个 GUI 中的滚动条控件。滚动条不是独立存在的，它是容器的一部分，所以用户不能创建滚动条。

一个滚动条能够被鼠标点击后拖动。

	方法
SetTrackRange	设置滚动条的范围。默认值为 0 到 1

SetPageSize	设置滚动条的页尺寸，默认值为 1
	纹理对象
track	滚动条轨道的纹理
up_left	滚动条左/上按钮的纹理
down_right	滚动条右/下按钮的纹理
thumb	滚动条滑块的纹理

editbox & imeeditbox

图 6.8 文本编辑框

表示一个 GUI 中的文本框控件。

一个文本框接收键盘的输入，一个 imeeditbox 不仅仅接收键盘的输入，还能接收输入法的输入，例如中文拼音。

一个 editbox 或者 imeeditbox 当被选定就可接收来自键盘的输入，点击控件就可选定。editbox 和 meeditbox 中的内容能够被选中，删除，复制，粘贴和插入。某种意义上说就和 windows 标准输入框一样。

当内容改变时会触发 onchange 事件，当 RETURN 按下会触发 onstring。

创建方法：

object= ParaUI.CreateUIObject("editbox","editboxname","_lt",50,20,600,400)

object= ParaUI.CreateUIObject("imeeditbox","editboxname","_lt",50,20,600,400)

属性	
text	string 类型，对象的文本
background	string 类型，对象的背景
readonly	boolean 类型，控制 editbox 和 meeditbox 是否只读

纹理对象	
background	edibox 的背景纹理
字体纹理对象	
text	editbox 的文本字体
selected_text	editbox 中被选中的文本字体

text

图 6.9 文本

表示一个 GUI 中的文本。

一个文本控件和按钮一样可以接受鼠标点击。

创建方法：

object= ParaUI.CreateUIObject("text","textname","_lt",50,20,600,400)

属性	
text	string 类型，对象的文本
background	string 类型，对象的背景
纹理对象	
background	对象的背景纹理
字体对象	
text	对象的文本字体，用来显示按钮中的文本

listbox

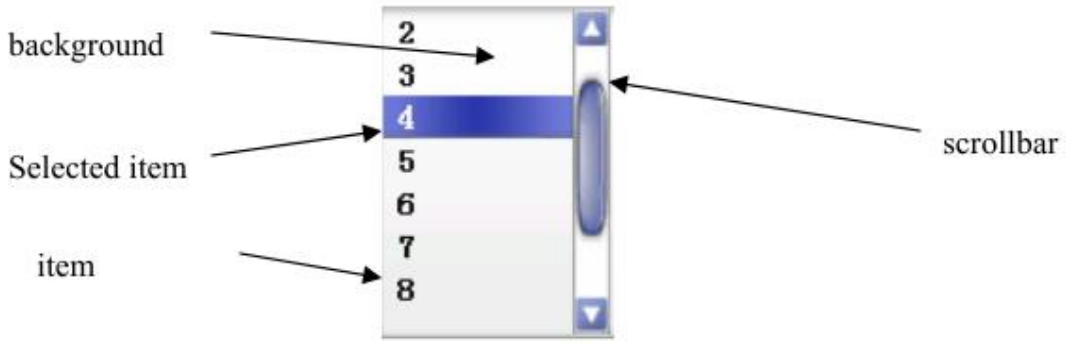

图 6.10 列表框

表示一个 GUI 中的列表框。

一个列表框允许用户添加文本元素并且按照一定顺序展示它们。属性“text“总是返回当前选中的元素。如果没有任何元素被选中，将返回” “。设置属性”text“是不允许的。

一个列表框接收鼠标点击和一些键盘输入。键盘控制行为和 windows 标准列表框控件一样。当遇一个元素被鼠标或键盘选取，会出发 onselect 事件，当一个元素被双击，会触发 ondoubleclick 事件。

创建方法：

object= ParaUI.CreateUIObject("listbox","listboxname","_lt",50,20,600,400)

属性	
itemheight	number 类型，列表元素中的元素高度。
scrollable	boolean 类型，列表框是否能被滚动，默认为 true
scrollbarwidth	number 类型，列表框滚动条的宽度，默认为 25
background	string 类型，对象的背景
wordbreak	boolean 类型，决定每个文本元素在太长的情况下是否可以被分作多行，默认为 false。当在 false 的情况下，超出部分将被截除掉
方法	
AddTextItem	将一个文本元素添加到列表框中
RemoveItem	移除给定索引的元素
RemoveAll	移除列表框中所有的元素

	纹理对象
background	对象的背景纹理
	字体对象
text	对象的文本字体，用来显示按钮中的文本

slider

图 6.11 滑块

表示一个 GUI 的滑块控件。

一个滑块可以接收鼠标和键盘的输入，击背景条或者拖动按钮将改变滑块控件的值。点击左下将值减少一，右上值增加一。使用向上翻页可将值增加范围的 1/10,向下减少 1/10。无论值怎么改变，都会触发 onchange 事件。

创建方法：

属性	
value	number 类型，滑块的当前值，默认为范围的一半
button	string 类型，对象中的按钮
background	string 类型，对象中的背景
	方法
SetTrackRange	设置滑块的范围，默认 0 到 100
	纹理对象
background	对象的背景纹理
button	对象中的按钮的纹理

6.2 MCML

6.2.1 什么是 MCML

MCML——类似 HTML 的标记语言， 用类似 HTML/Java 的模式快速开发游戏 UI。

MicroCosmos Markup Language，是 ParaEngine 定义并开发在 ParaWorld 中使用的标记语言。 MCML 基于 XML 标准，通过嵌套的元素和属性，使用了基于 Tag 的结构，支持大部分 CSS 和 HTML Tag，可以用来描述社区或游戏中的许多 UI，例如几乎所有 windows 常用控件，任务，道具，等等。我们可以将它理解为 Web 中的 HTML，打个比方，可以把它看作是三维社交网络中的 HTML，用来描述二维和三维网络中的可渲染对象。其中以 pe 开头的标记是 ParaEngine 中所特有的 Tag，常常对应一个 NPL 类库中的 UI 控件，它的作用是将数据绑定到 NPL 控件。 MCML 的渲染器是用 NPL 脚本编写的，具有较高的扩展性，它可以让开发者定义自己的 Tag。

6.2.2 MCML Tags

作为和 HTML 类似的标记语言，MCML 面向的是三维社交网络，它支持的标签更加广泛，可以分为 8 类：

MCML 支持的标签	
HTML tags	_text_ h1 h2 h3 h4 li p div hr font span strong a(href) form img(attr: src,height, width, title)等等 支持大量 HTML 标记，更多信息见 http://www.paraengine.com/twiki/bin/view/Main/HTMLTags
Design tags	pe:gridview pe:xmldatasource pe:mqldatasource pe:dialog pe:tabs pe:tab-item pe:treeview pe:treenode pe:image pe:flash pe:container pe:editor pe:editor input(button, listbox, text, radio, checkbox, file, etc) pe:slide (interval=3 order="sequence"|"random"), pe:filebrowser(rootfolder="script" filter="*.lua;*.txt") pe:fileupload pe:progressbar pe:canvas3d pe:numericupdown

	pe:sliderbar pe:colorpicker pe:ribbonbar
Social tags	pe:name pe:profile-photo pe:avatar pe:profile pe:userinfo pe:friends pe:app pe:profile-action pe:profile-box pe:app-home-button
Map tags	pe:land pe:map pe:map-mark pe:map-mark2d pe:map-tile
Control tags	pe:if-is-user pe:if pe:if-not
Component tags	pe:roomhost pe:market pe:comments pe:ribbonbar pe:command pe:asset pe:bag
Worlds tags	pe:world pe:world-ip pe:model
Motion tags	pe:animgroup pe:animlayer pe:animator

6.2.3 后台代码解析

在 MCML 脚本体系中，存在两种脚本，行内脚本和外部脚本。行内脚本是存在于 html 文件中的脚本，行外脚本通常定义在 lua 文件中。

行内脚本

script/kids/3DMapSystemUI/Desktop/LoginPage.html

```
<script type="text/npl">
    <![CDATA[
    function OnInit()
        local self = document:GetPageCtrl();
        self:SetNodeValue("username", Map3DSystem.User.Name);
        self:SetNodeValue("password", Map3DSystem.User.Password);
    end
    OnInit()
    ]]>
</script>
```

dreamtech

行内代码范围(页面)。每次加载或刷新页面时都要编译。所调用的函数均在页面内。

外部脚本

为了将 MCML 与表达逻辑分离开来，可以将逻辑代码放入一个单独的 lua 文件中。这样的 lua 文件称为外部脚本。可使用以下代码在 MCML 文件中设置外部脚本:

<script type="text/npl" src="*********.lua"></script>

上述代码等同于 NPL.load()，外部脚本中的代码仅仅会被编译一次。

PageCtrl.lua 解析

MCML 文件和相应的外部脚本文件定义了一个页面模版。我们可以使用这个相同的模板创建许多实例，只要我们创建的页面源自 PageCtrl，我们就能够使用 PageCtrl:Create(name, _parent, alignment, left, top, width, height)来为我们的界面创建新的实例。

概述	**一个从 MCML 文件初始化的交互式 NPL 控件**
文件位置	script/kids/3DMapSystemApp/mcml/PageCtrl.lua
描述	PageCtrl 通过在 MCML 文件中创建 NPL 控件的方式，自动实现从静态或异步的 MCML 文件中构建交互式控件，这个 MCML 文件包含着所有 UI 元素的布局和默认数据绑定。我们可以在其中实现预定义或 MCML 页面定义事件处理器，同时我们能够轻松地接入在 MCML 中定义好的 UI 和数据绑定控件。我们将之称为 MCML/NPL Code Behind 模式，用作分离 UI 和实现逻辑。

6.2.4 MCML 架构

lreanTech

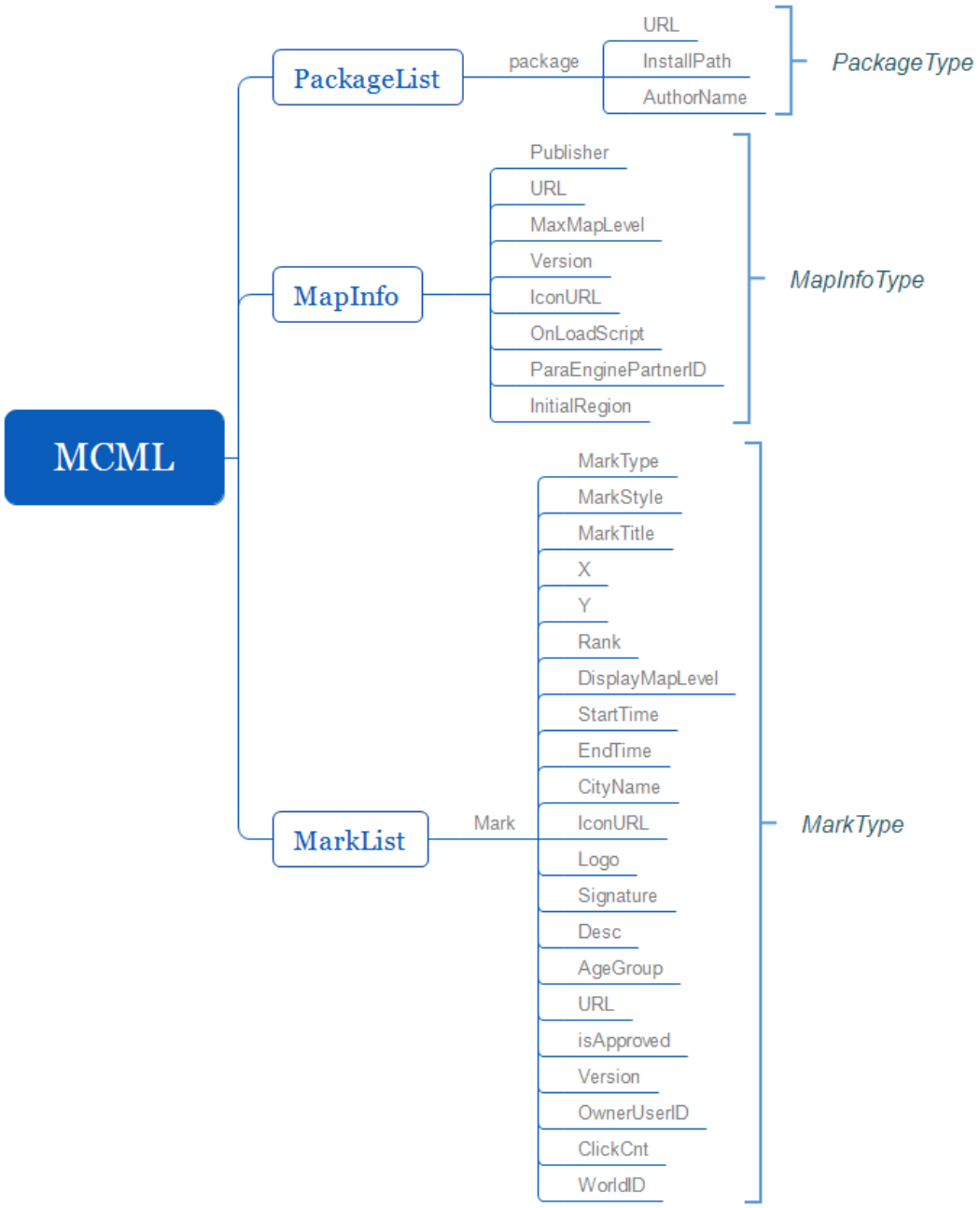

图 6.12　MCML 架构

6.2.5 MCML 作用

MCML 用来在地图浏览器中展示地理数据。例如 ParaEngine 3D 地图系统。MCML 的作用可以总结为以下 6 条：

- 指定地图标记（图标和标签）以便在地图上确定位置
- 创建不同的相机位置为每个特性定义独特的视点

- 在 MCML 中嵌入 NPL 脚本来在地图上创建复杂的 UI
- 从远程或本地网络位置中动态地获取和更新 KML 文件
- 在地图观察器上基于变化获取 KML 数据
- 在地图上显示与 ParaX 兼容的 3D 对象

6.3 系统窗口

系统窗口即 2D GUI 的主容器。最通用的创建系统窗口的方法是使用 mcml 标记语言，编程模式就像编写 HTML/js 网页。

要创建系统窗口，首先要创建一个 html 文件。要使用 html 文件创建并显示窗口，有两种方式，称为 V1 和 V2：

v1：

```
NPL.load("(gl)script/kids/3DMapSystemApp/mcml/PageCtrl.lua");
        local page =
System.mcml.PageCtrl:new({url="source/HelloWorldMCML/mcml_window.html"});
        page:Create("testpage", nil, "_lt", 0, 0, 300, 400);
```

v2：

```
NPL.load("(gl)script/ide/System/Windows/Window.lua");
        local Window = commonlib.gettable("System.Windows.Window")
        local window = Window:new();
        window:Show({
                url="source/HelloWorldMCML/mcml_window.html",
                alignment="_lt", left = 300, top = 100, width = 300,
height = 400,
        });
```

v1 和 v2 的差异：

v1 使用 NPL 内置的用 C++写成的对象。它们的实现隐藏在 NPL 脚本中，用户 IO 通过回调函数暴露给 NPL 脚本。

v2 使用 NPL 脚本通过 3d-triangle drawing API 创建所有 GUI 对象并且在 NPL 脚本中处理用户的输入。因此，可实现用户的完全控制。

7 3D 引擎

7.1 资源管理

7.1.1 介绍

在计算机所描绘的 3D 世界中，所有的物体模型（如树木，人物，山峦）都是通过多边形网格来逼近表示的。网格模型是一种将物体模型的顶点数据、纹理、材质等信息存储在一个外部文件中的 3D 物体模型。对于那些简单的图元描述的图形，比如点，线，三角形等等，我们可以通过写代码指定顶点数据，索引数据，法线向量，纹理和材质等等信息。但对于复杂的 3D 物体的话，采用这种方式显然是不现实的。因此我们需要使用网格模型的技术，从各种特定的文件格式中读取和绘制 3D 图形。使用网格模型最普遍的方式是从外部的 3D 模型文件中加载一个网格。而这些 3D 模型通常都是由 3D 建模软件生成的，比较复杂的网格数据。

7.1.2 网格

7.1.2.1 网格 Mesh 分析

由多边形网格构成的物体储存着不同种类的基本元素：包括顶点，边线，面片和多边形。在许多情况下，只有顶点，边线以及面片会被存储。渲染器有可能只支持三角形，这时候多边形必须得由许多三角形组成。其他情况下，一些渲染器要么可以支持四边形或更多边形，要么能把多边形转化为三角形，这时候就没必要把网格储存为三角形格式。

网格参数介绍：

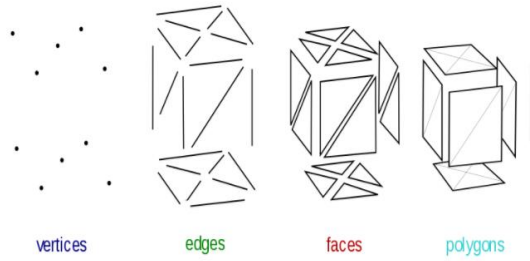

图 7.1 网格结构图

-顶点 Vertex：包含了诸如颜色，法线向量，纹理坐标等信息的位置因素

-边线 Edge: 连接两个顶点形成的个体

-面 Face：一套紧密相邻的边线 Edge 集合。一套共面的面 Face 构成多边形 Polygon。一般情况下面 Face 和多边形 Polygon 是等价的，但是由于渲染硬件一般只支持 3~4 边的面-Face,所以多边形也表示为多重面。由于多边形和面的从属关系，可能会带来几何结构，形状等额外性质。

-UV 坐标：用于进行顶点和像素点的映射，便于贴图。

Mesh 解析流程如图：

dreamTech

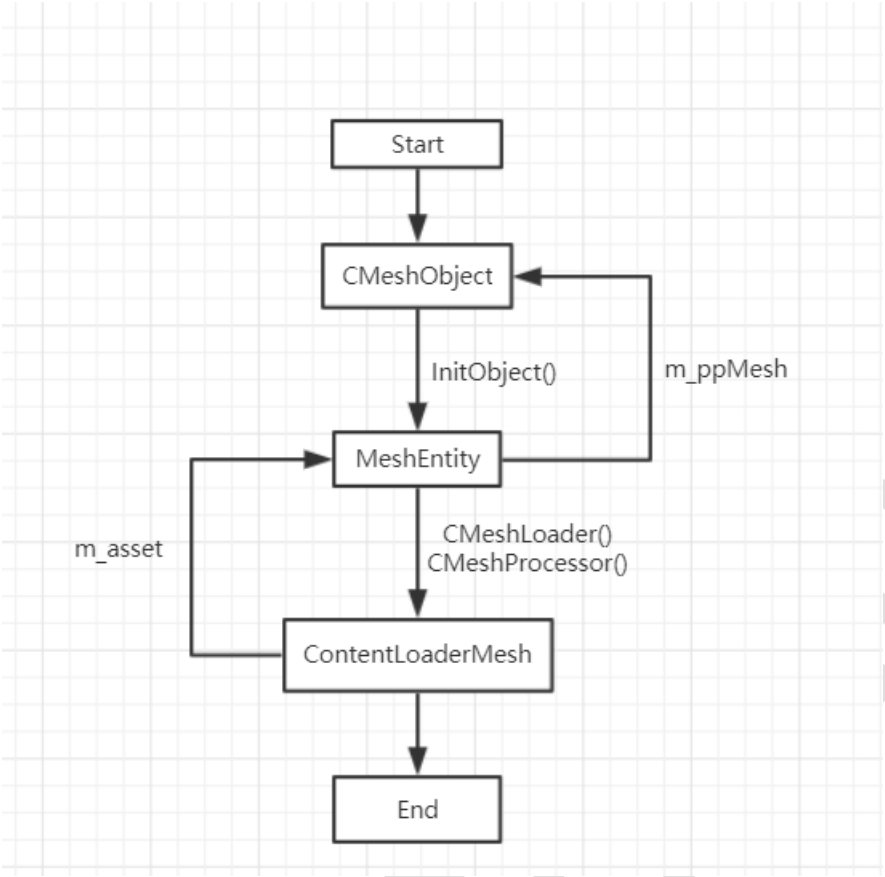

图 7.2 Mesh 解析大致流程图

1. CMeshObject 在生成各种网格对象之前进行初始化，向下层索取如网格，纹理等信息。
2. MeshEntity 负责产生网格信息，它会产生一个调用下层的 CMeshLoader 加载器和 CMeshProcessor 去目录或者网络上搜寻资源文件，解析并提取资源信息，呈递给渲染器，最后将结果返回到上层的 CMeshObject。
3. ContentLoader 在收到上层的调用动作后会检索文件，解析文件分类提取信息更新到 m_asset 呈递到上层，此外还负责关于网格 LOD 的设置。

Note：-m_ppMesh: 涵盖网格的结构和基本信息

-m_asset:网格加载处理器将呈递给上层的资源文件

7.1.3 ParaX

ParaEngine 给它自己的文件定义了一种格式叫 ParaX。它是基于微软.x 文件模板建立的，

也因此它也被设计为可以良好兼容 DirectX V9.0b D3DX API。ParaX 格式定义了骨骼，皮肤网格，骨骼动画，动画序列，动画速度，骨骼 alpha animations 等属性。

ParaX 文件的顶层部分如下图. 在图中一共有 3 块顶层部分被定义。前两块对于 DirectX 保留的模式 API 可以良好兼容。第三块由 ParaEngine 定义。并且由于这些顶层部分并没有任何重叠，ParaX 文件可以被任何支持 DirectX 的商业软件查看或修改

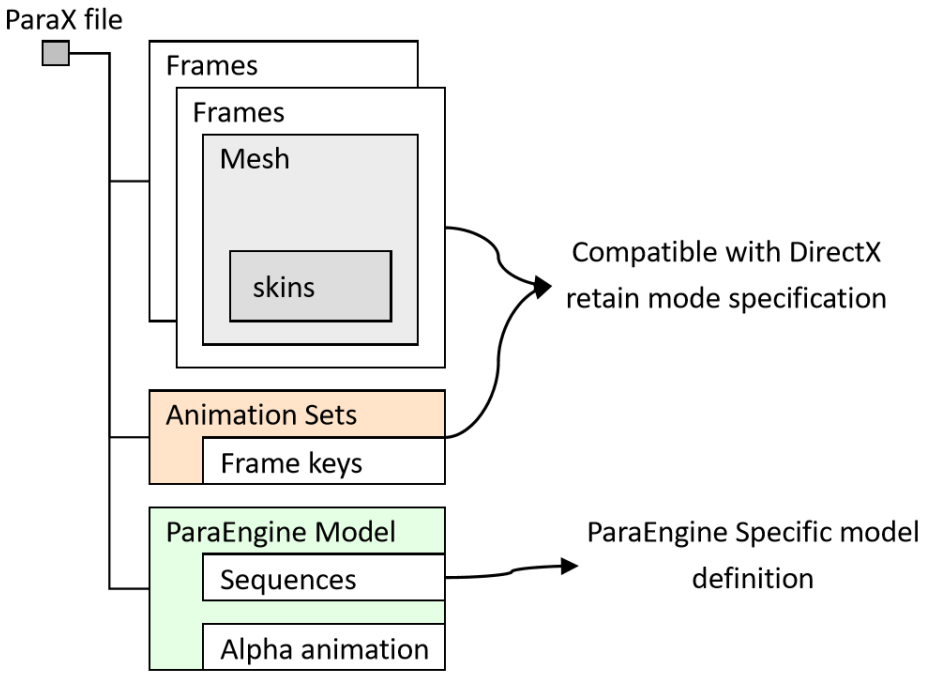

图 7.3 ParaX 文件层级

下表所示是一个典型的 ParaX 模板实例，文件结构对应着图 Figure 1.包含两块骨骼部分：SCENE_ROOT 和 BODY，一个网格对象，一个动画设置，一个动画序列。网格对象包含三个顶点，一个面，一种材质和两块蒙皮。

```
xof 0302txt 0032
Header {
  1;
  0;
  1;
}
Frame SCENE_ROOT {
        FrameTransformMatrix {
```

```
1,0,0,0,
0,1,0,0,
0,0,1,0,
0,0,0,1;;
 }
Frame Body {
    FrameTransformMatrix { // frame's local matrix relative to its parent
1,0,0,0,
0,1,0,0,
0,0,1,0,
2,2,2,1;;
 }
        Mesh pCubeShape2 { // a static mesh object
          3; // number of vertices defined below
        1.5;0;0;,
        0;0;1.5;,
        0;1.5;0;,
          1; // number of faces defined below
        3;2,1,0;, // each face has three indexes of the previously defined vertices
            MeshMaterialList {
            1; // number of materials
            1; // number of faces whose material is defined below
            0;; // material index
                    Material { // the material
                      0.800000;0.800000;0.800000;1.000000;; // diffuse
                    2.000000;
                      0.000000;0.000000;0.000000;;
                      0.000000;0.000000;0.000000;;
                    TextureFilename { // texture
                    "Texture\\dao.bmp";
                    }
            }
            MeshNormals {
             2; // number norm values defined below
            0.565854;0.233529;-0.790743;,
            -0.445748;0.525722;-0.724517;,
              1; // number of faces whose norm is defined.
            3;1,1,0;, // the three norm indexes for the three vertices of the face
            }
            MeshTextureCoords {
              3; // number of vertices whose texture coordinates are defined.
            0.709954;-0.303127;,
            0.651408;-0.155958;,
            0.651391;-0.334976;;
```

dreamTech

```
                }
                XSkinMeshHeader {
                  2; // max number of weights per vertex
                  2; // max number of weights per face
                  2; // number of bones
                }
                SkinWeights {
                    "Body";
                     3; // number of vertices which are bound to this bone
                     0, // index of the vertices in the current mesh
                     1,
                     2;
                     0.55; // weight of the vertex
                     0.78;
                     0.12;
                     1,0,0,0, // offset matrix
                     0,1,0,0,
                     0,0,1,0,
                     2,2,2,1;;
                }
                SkinWeights {
                    " SCENE_ROOT";
                     1; // number of vertices which are bound to this bone
                     2,
                     0.89; // weight of the vertex
                     1,0,0,0, // offset matrix
                     0,1,0,0,
                     0,0,1,0,
                     0,0,0,1;;
                }
            } // end of mesh
        }// end of frame
}// end of frame

AnimationSet {
  Animation {
    {body}
    AnimationKey {
    0; // key type (rotation, translation or scaling)
    2; // number of keys
    0; 4; -0.185982, -0.721465, -0.648574, 0.155728;;, // timed keys
    120; 4; -0.185982, -0.721465, -0.648574, 0.155728;;
  }
  Animation {
```

IdreamTech

```
    {SCENE_ROOT}
    AnimationKey {
    1; // key type (rotation, translation or scaling)
    2;
    0; 3; 1.000000, 1.000000, 1.000000;;,
    120; 3; 1.000000, 1.000000, 1.000000;;
   }
}
ParaEngine female1{
  Sequences{
    Anim{
      "Stand"; // animation name
      0; // from
      120; // to
      -1;      // loop on its self
      0;       // no speed
      0;0;0;;
      0;0;0;;
      100.0;        //bounding sphere radius
    }
  }
}
```

表 7.1 ParaX Simple File

7.1.3.1 ParaX 模型解析

ParaX 解析流程如图：

IdreamTech

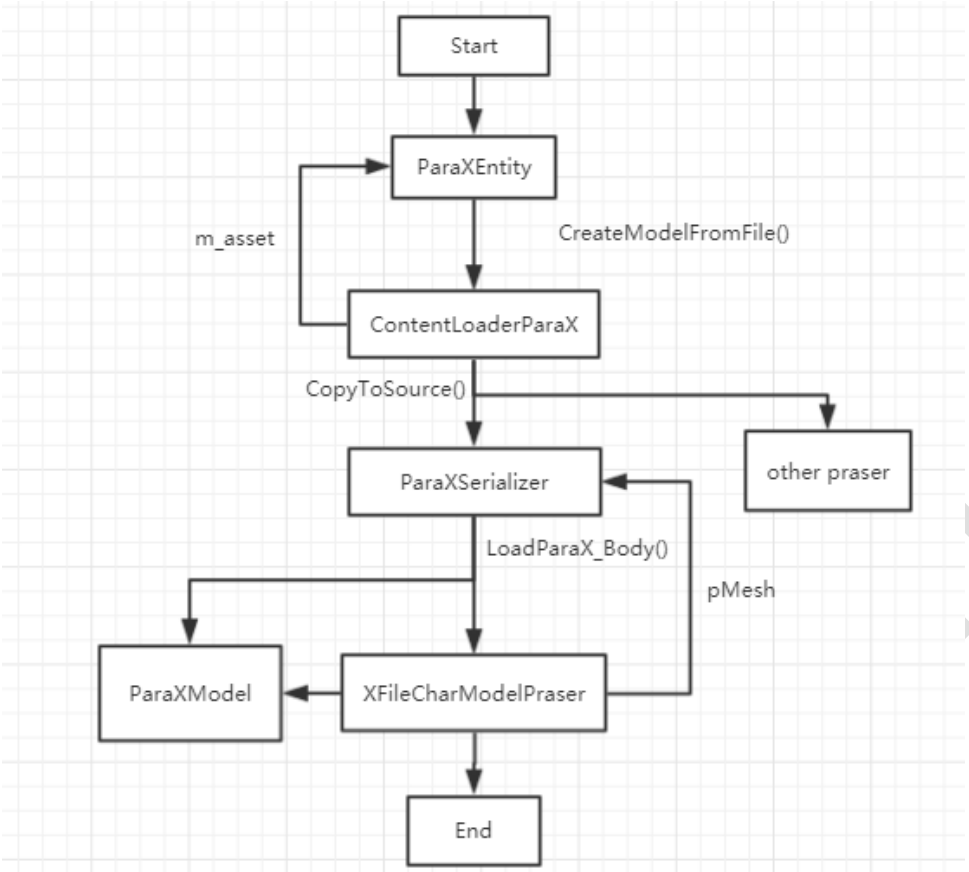

图 7.4 ParaX 大致解析流程

1. ParaXEntity 负责向上层传递 m_asset.它会调动下层的 ContentLoaderParaX 执行资源文件的读取加载工作进行信息收集。
2. ContentLoaderParaX 在加载和处理文件的时候，会通过文件名检索种类来分配对应的解析器。对于标准 ParaX 格式文件，ParaXSerializer 会被调动。
3. ParaXSerializer 类是 ParaX 解析的核心部分，由 ParaXSerializer(文件读写)和 ParaXPraser(文件解析)两部分组成。ParaX 解析结构如下列代码所示

```
void* CParaXSerializer::LoadParaXMesh(CParaFile &f, ParaXParser& p)
{
    void* pMesh=NULL;
    if(LoadParaX_Header(p)){
        pMesh = LoadParaX_Body(p);
        LoadParaX_Finalize(p);
    }
    return pMesh;
```

```
}
```

分为 Header，Body，Finalize 三个部分。

Hearder 部分会判定文件各部分种类并执行对应操作，解析过程中 Header 一定得最先完成，没有完成或者失败都会终止后续操作并报错

```
bool CParaXSerializer::LoadParaX_Header(ParaXParser& Parser)
{       ......
        if(Type == TID_ParaXHeader){...}
        else if(Type == TID_ParaXBody)
        else if(Type == TID_XDWORDArray)
        else if(Type == TID_ParaXRefSection)
        else if(Type == TID_D3DRMMesh)
        else if(Type == TID_D3DRMFrame)
        else{...}
        }
    }
    return true;
}
```

Body 部分会先将模型分为有动画和静态两类，对于有动画的模型会分析文件信息判定种类，分类读取文件资源，再然后进行详细的动画配置。

```
    if(Parser.m_xheader.type == PARAX_MODEL_ANIMATED || Parser.m_xheader.type
== PARAX_MODEL_BMAX)
        {           ......
                if(SUCCEEDED(pSubData->GetName(szName, &nSize))){
                ...
                if(Type == TID_XDWORDArray){...}
                else if(Type == TID_XVertices) {...}
                else if(Type == TID_XTextures) {...}
                else if(Type == TID_XAttachments){...}
                else if(Type == TID_XTransparency){...}
                else if(Type == TID_XViews){...}
                else if(Type == TID_XIndices0){...}
                else if(Type == TID_XGeosets){...}
                else if(Type == TID_XRenderPass){...}
                else if(Type == TID_XBones){...}
                else if(Type == TID_XTexAnims) {...}
```

IdreamTech

```
                else if(Type == TID_XParticleEmitters){...}
                else if(Type == TID_XRibbonEmitters){...}
                else if(Type == TID_XColors){...}
                else if(Type == TID_XCameras){...}
                else if(Type == TID_XLights){...}
                else if(Type == TID_XAnimations){...}
                }
              }
                SAFE_RELEASE(pSubData);
            }

                ...\\很大部分关于动画animation的代码
        else if(Parser.m_xheader.type == PARAX_MODEL_DX_STATIC)
        {
            // TODO: for original dx model file.
        }
```

Finailize 部分负责资源释放，准备下轮解析

```
void ParaXParser::Finalize()
{
    SAFE_RELEASE(m_pParaXBody);
    SAFE_RELEASE(m_pParaXRawData);
    SAFE_RELEASE(m_pDXEnum);
    SAFE_RELEASE(m_pParaXRef);
    SAFE_RELEASE(m_pD3DMesh);
    SAFE_RELEASE(m_pD3DRootFrame);}
```

4. ParaXModel 是被定义为解释人物动画信息等数据的类，在完整的 ParaX 解析过程中它会被许多地方所引用到，对于 ParaXSerializer，它提供了动画的配置方法以及相关模型信息的读写提取。

7.1.3.2 ParaX 静态模型

ParaXStaticModel 即 ParaX 静态模型，Create() / ClonePhysiceMesh() / LoadToSystemBuffer() / Render() 是最主要的功能方法，侧重于整体的纹理铺设，渲染，物理相关加载和数据到内存的加载缓冲。它基本作用按照 ParaEngine 的作者意思应该和 StaticMesh

是差不多，用于辅助生成 LOD，但是实际的解析过程中，ParaXStaticModel 却没有被任何地方引用。

7.1.4 ParaX 扩展

ParaEnigine 有着自己定义的专属格式 ParaX，但市面上制作 3D 模型的软件太多，他们的产品格式也各不相同，为了保证 ParaEngine 的兼容性和适用性，ParaX 添加了对几种主流模型文件的扩展，例如：FBX,BMax(ParaEnigine 自定义格式),CAD,MDX。具体方法就是在解析文件的过程中进行文件种类筛选，分配对应的解析器解析，流程如图：

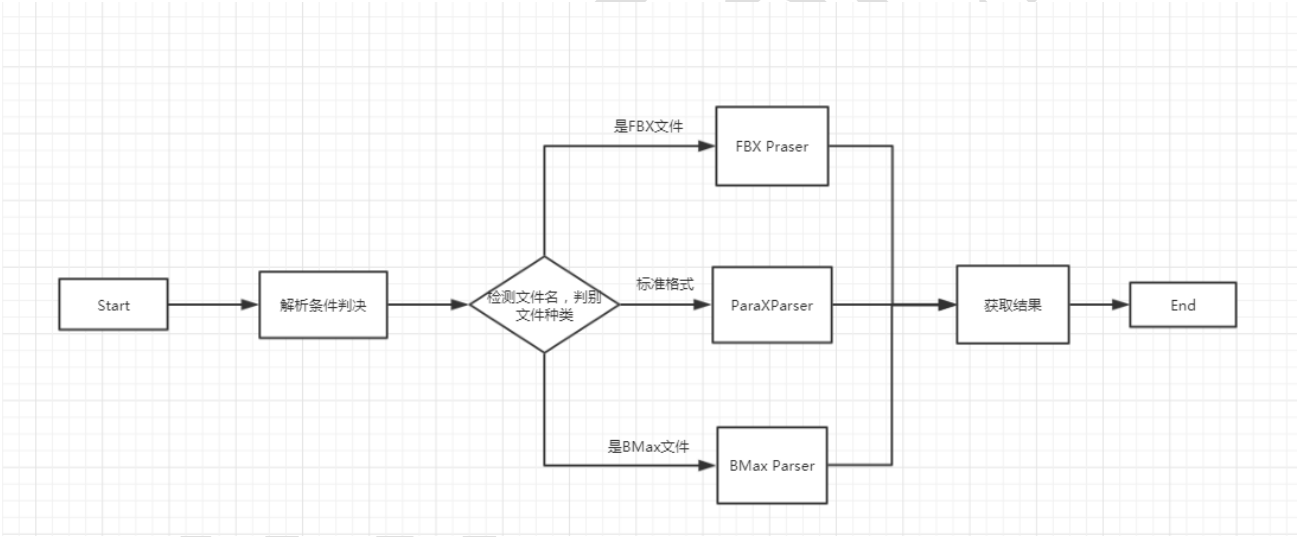

图 7.5 ParaX 扩展解析流程

7.1.4.1 FBX

FBX 格式是一种被 AutoDesk 公司的诸如 MAYA/3dsmax 等软件广泛使用的文件格式，也是工业上被用得最多的文件格式之一。

ParaEnigine 的 FBX 解析器将所有的解析方法，对象全都高度整合到一起，只声明了解析器 FBXPraser 和用来读取动画 xml 文件的 FBXModelInfo 这两个类。

FBXPraser 承担了材质，纹理，色域，动画，骨骼分配与家属关系确立，可见性等多个方面

的功能，但具体可以概括为两类：

```
XFile::Scene* ParaEngine::FBXParser::ParseFBXFile(const char* buffer, int nSize)
{
    Assimp::Importer importer;
    Reset();
    const aiScene* pFbxScene = importer.ReadFileFromMemory(buffer, nSize,
aiProcess_Triangulate | aiProcess_GenSmoothNormals, "fbx");
    if (pFbxScene) {...}
    else {...}
    return m_pScene;
}
```

ParseFBXFile():会读取网格和材料信息并进行相应处理后然后返回结果，用来解析静态模型

```
CParaXModel* FBXParser::ParseParaXModel(const char* buffer, int nSize)
{
    CParaXModel* pMesh = NULL;
    Assimp::Importer importer;
    Reset();
    SetAnimSplitterFilename();
    const aiScene* pFbxScene = importer.ReadFileFromMemory(buffer, nSize,
aiProcess_Triangulate | aiProcess_GenSmoothNormals | aiProcess_FlipUVs, "fbx");
    if (pFbxScene) {...}
    else {...}
    return pMesh;
}
```

PaserParaXModel():会读取材料和动画信息并进行相应处理，之后会建立骨骼间的父类子类关系，以及从 entity 或者库中提取必要数据。用来解析有动画的模型

几个重要方法:

-HasMaterials():读取 mNumMaterials 并执行 ProcessStaticMaterials()

-HasAnimations()：读取 mAnimations 并执行 ProcessFBXAnimation()

-ProcessEBXBoneNodes()：建立骨骼家属关系，提供在必要地方的网格补加方法

-FillParaXModelData():从各个 Entity 或者库里提取模型解析必要的数据

-PostProcessParaXModelData():预计算处理骨骼相关参数值来优化网格

iRealTech

7.1.4.2 BMax

BMax 是内置的 block max 文件格式。BMax 即 Block Max，是一种专门被用在 Paracraft 中来储存和交换 Block 数据的一种文件格式。

BMax 解析器和 FBX 解析器不一样，BMax 解析器不像 FBX 解析器将所有的东西集成到一块，BMax 解析器只是简单的分析文件结构，分类处理部分，再交由下层对应各类处理，更加的灵活。BMax 解析结构如代码所示：

```
void BMaxParser::Load(const char* pBuffer, int32 nSize)
    {
        BMaxXMLDocument doc;
        doc.Parse(pBuffer);
        ParseHead(doc);
        ParseBlocks(doc);
    }
```

其中 ParseHead 只是简单判定文件是否有预设参数,解析步骤置于 ParseBlocks 中,如下：

```
        const char* value = blocks_element->GetText();
                ParseBlocks_Internal(value);
                ParseBlockFrames();
                CalculateBoneWeights();
                ParseVisibleBlocks();
                if (m_bAutoScale)
                    ScaleModels();
```

NOTE：ParseBlocks_Internal(value) / ParseVisibleBlocks()：返回节点索引

ParseBlockFrames()：解析骨骼

CalculateBoneWeights()：判定骨骼家属关系

整体基本结构如图：

IdreamTech

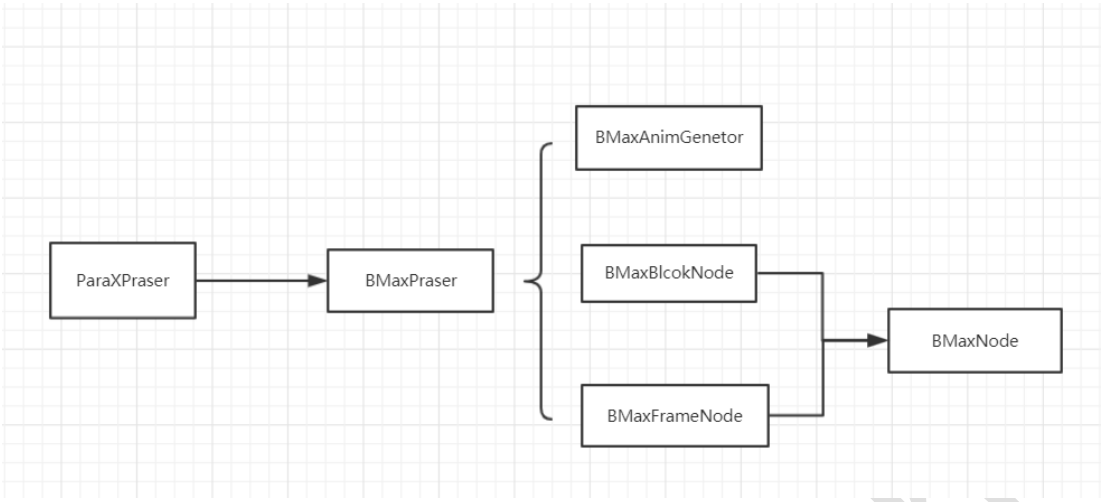

图 7.6 BMax 解析结构类图

7.1.4.3 CAD

ParaEngine 中对 CAD 的解析进行了比较细致的方法说明，但是似乎已经舍弃了对于这种模型的引用，所以这些方法并没有在任何地方被调用过，这里仅对 CAD 解析过程进行简单的介绍。

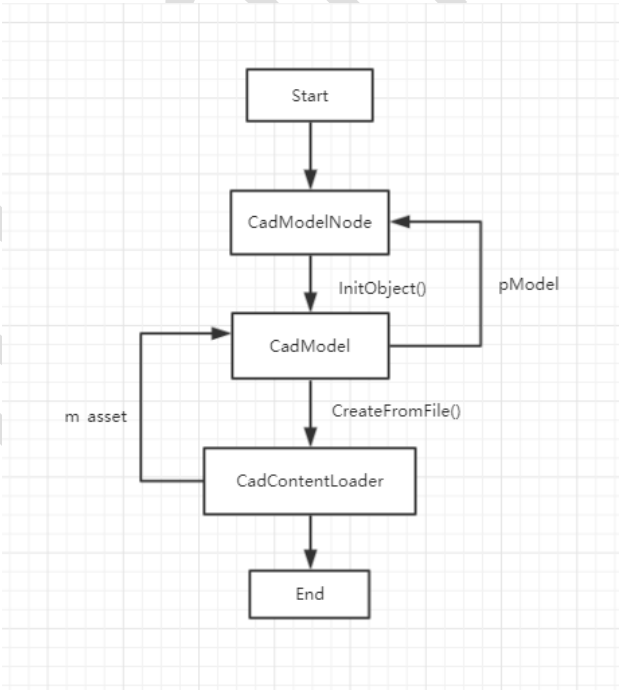

图 7.7 CAD 解析结构类图

CadModelNode 继承自 CTileObject，后者继承于 CBaseObject，负责场景，3D 相关功

能的集成。在处理 CAD 模型时：

- CadModelNode 将被启用负责参数处理和渲染工作，它首先会执行 InitObject()，需要向下层所要 m_pModel.

- CadModel 会调用下层的加载器和处理器来解析文件提取信息，并将获取的文件呈递给渲染器，最后将结果呈递给上层。

- CadContentLoader 负责最底层的文件搜寻解析，信息提取工作。

7.1.4.4 MDX

目前 ParaEngine 并未使用 MDX 相关文件,只是简单了预声明了一个 MDXEntity 类，并没有构写相关功。能方法，没有使用实例

7.2 场景管理

在场景管理中，一般用场景图（scene graph）来管理场景中的物体。场景图是一个有向无循环图（DAG），一个包含能够定义“3d 世界”的所有信息，它包含了底层信息，也包括了高层信息。对场景的管理实际上就是对场景图的管理。而对场景图管理主要需要两个任务，一个是场景的组织，另一个是场景的渲染。

在场景中，物体虽然很多，但是大多数都是不可见的，在摄像机的视锥体范围的物体只是一小部分。为了能够提高运行速度，可以用小的计算代价剔除不可见物体（可见性剔除），减少绘制元素。而为了实现这样的方法，就需要涉及到空间排序（spatial sorting），最基本的算法就是二叉空间分割树（BSP）。简单点说就是用平面对给定的一组对象分组（如果有对象和这个平面相交，则将该物体分成两个对象 作为该节点的两个子节点），然后一直分到一个特定条件（一般是节点上只有一个对象）为止。

当然只有 BSP 还是不能满足游戏的需求，BSP 树的构建非常耗时，因此适合静态场景。因为现在的游戏场景都很大很复杂，又有很多物体，就算剔除了不在视锥体的对象，还是有大量的物体需要送入渲染器。但是在这些物体中有很多物体实际上是看不到的，所以还需要

用到 Protal 技术。首先引入 region 的概念。一个 region 是一个封闭空间，不同 region 之间通过 Protal 连通。

八叉树类似轴对称的 BSP 树，沿着轴对包装盒进行分割，产生 8 个新的包装盒，依次递归直到到达最大层次或者包围盒中包含的图元小于某个阈值。

一般来说，对于室内场景使用 BSP 树，因为 1）室内场景遮挡比较严重，使用 BSP 树在特定的位置分割有助于提升效率；2）室内很可能朝某个方向延伸比较多，比如一条细长的走廊。对于大规模室外场景，使用八叉树比较好，因为场景中的物体比较分散，而且不会出现太多的遮挡，由于不用存储分割平面位置，使用八叉树这种规则的空间结构能够提高效率。当然，如果这个大规模室外场景的物体主要集中在地面，使用四叉树进行空间管理会比较好。

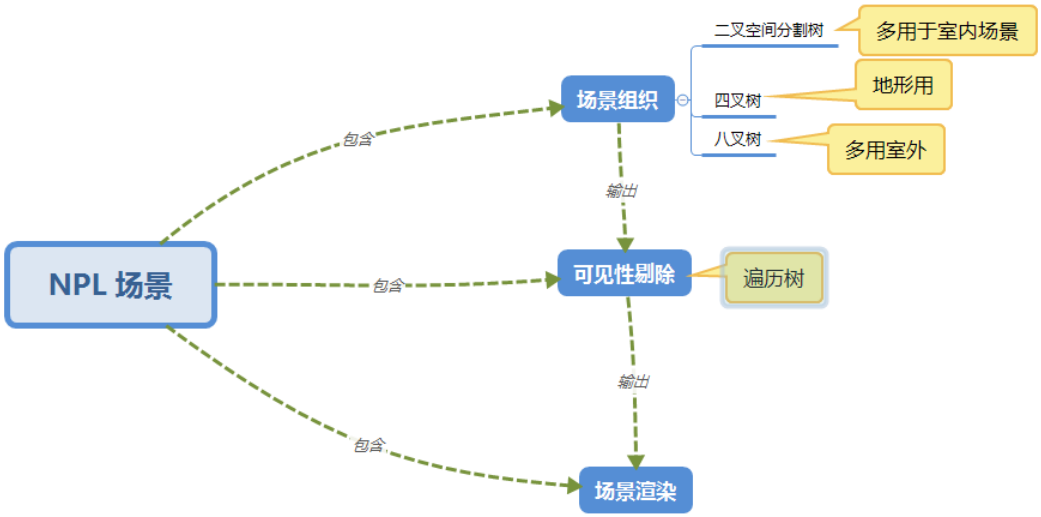

图 7.8　NPL 场景管理概要

在场景渲染部分，根据给定的相机，场景，光源，光照和纹理等信息生成二维图像。实际上在场景管理中，场景的组织，渲染，都是为了能提高图像帧率，场景的组织方式的改变是为了减少渲染时间，或者减少可见性判断所用的时间。归更到底，无论是结构还是算法都是为了提高程序运行效率。

接下来将场景场景管理分为两个类来介绍，他们都需要继承 CBaseObject 类，这个类是所有有关场景管理的基类。第一个是场景管理类，这些类负责对场景中组成对象的管理。

另一个是场景组成类，这些类一般表示场景中的一类物体（可见/不可见的都有），这些类都包含了用于自己的方法和属性。

7.2.1 基础场景对象

CBaseObject 类是基础场景对象的抽象类，它是所有 3d 对象的基础。所有的 3d 对象都需要继承这个类。

它包含了 4 类方法:

1.对象属性查询，像位置，朝向，弧度，高度，容器类型等等。

2.碰撞检测，实现了 IViewClippingObject 接口，提供了 TestCollisionObject, TestCollisionSphere 等操作，用于碰撞检测

3.帧移动，实现了人物和根场景（sceneRoot）的帧移动。其他对象的帧移动在他们自己的类中实现。

4.渲染。差不多所有对象都用 Draw 方法来渲染他们自己（静态对象）。有些对象（动态对象），比如人物需要额外的提前时间 AdvanceTime 来计算动画。不过有些动态对象不需要提前，比如 sprite 图片精灵，因为这些对象只需要很小的时间就可以提前算好帧或者对他们的帧时间上要求不是很精确，因此把这些动态的对象当静态处理。

当然也包含了通用的数据结构和方法：

1.object type and identifier：m_objType，对象的类型，是个枚举类型。

2.tree node management: m_children 是子树的数组，这里存了强引用。

3.event management。

dreamTech

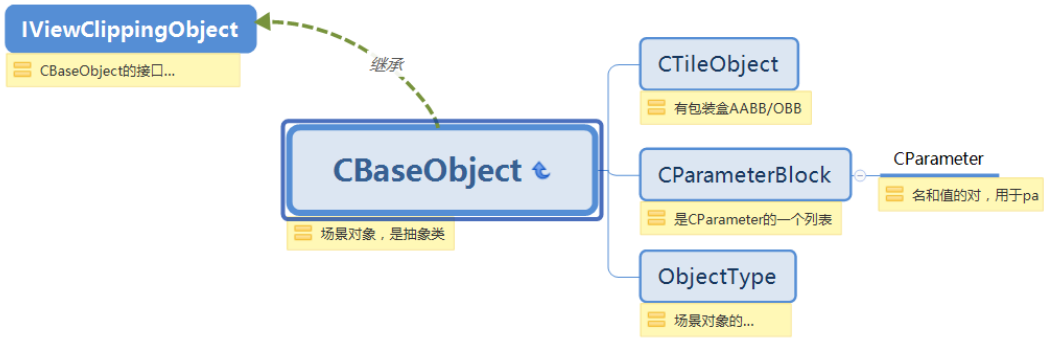

图 7.9 CbaseObject 的类的练习图

在 CBaseObject 中还有对渲染优先级的排序。

```
/** 0 if automatic, larger number renders after smaller numbered object.
[1.0-2.0): solid big objects
[2.0-3.0): solid small objects
[3.0-3.0): sprites
[4.0-4.0): characters
[5.0-5.0): selection
[6.0-7.0): transparent object
[100.0, ...): rendered last and sorted by m_fRenderOrder
*/
float m_fRenderOrder;
```

场景中所有的类，包括管理类和组成类，都是继承自 CBaseObject 类的。

7.2.2 场景管理类

在 NPL 场景管理中，场景管理类提供了许多方法以及相应的数据结构来对场景中的物来管理。

1） CSceneObject，这个是场景中最高等级的管理类，他实现了对整个场景的组织。

2） CMiniSceneGraph，是对场景中动态物体，即时物体的管理类。

3） CZoneNode，区域和门渲染技术中的区域，用于对场景中物体分区域。

CPortalNode，区域和门渲染技术中的门，用于联通两个区域。

4） 视口管理器

5） 电影播放

6） 水面模拟

7） 物理空间

8） 选中对象管理器

9） 场景对象拾取

10） 地形空间

11） 天气效果

7.2.2.1 CSceneObject

CSceneObject 是最高等级的场景管理类，并且向上提供了很多方法来管理场景。它用了设计模式中的单例模式，继承了 CBaseObject，Iscene 类。它通过树形结构管理了所有的游戏对象， 例如全体地形，摄像机，全体二足动物（例如人），物理，资源管理，渲染状态（render state）和天空盒子。

首先是对场景的组织，在 NPL 引擎中，实际是用了 map 这个数据结构来存储 region 和 portal 的信息，而不是通用的用二叉树。并且场景的大部分对象是用四叉树来组织的，并没有用到上面提到的更多的数据机构。

主要属性：

1）m_pTileRoot，是四叉树的根节点，也是场景管理的核心，保存了大多数通过空间分割的静态对象。

2）m_globalTerrain，是动态全球地形对象，是针对地形数据。

3）m_sceneState，包含了仿真和渲染状态，例如那个对象被渲染或者动画。

4）m_pPhysicsWorld 物理场景和 SDK 包装类的对象。

5）m_pBlockWorldClient 方块系统使用（ParaCraft 专用）。

6）m_pSunLight 太阳光的各种参数。

dreamTech

7）m_zones 和 m_portals 这两个参数用作区域划分以及渲染。

主要方法分类：

1）对场景中物体的操作，例如显示/隐藏物体等。

2）对场景中的雾的操作，设置雾的颜色，密度等。

3）对阴影的操作，例如设置是否渲染阴影，设置/获取阴影投射器/阴影接收器的最大数量。

4）设置/获取光晕效果。

5）创建/删除天空盒子，太阳光照等。

6）将场景中的物体附着/解除附着到场景图中，在每一个场景图的节点中对象用 map 映射。

7）设置/创建/删除微场景图。

8）设备输入处理，鼠标拾取，拉拽等。

9）渲染参数设置，渲染角色，阴影，光晕，水面等等。

10）数据库存取操作，存/取角色/NPC。

11）region 和 portal 的设置等。

12）物理遮罩的添加和设置。

IdreamTech

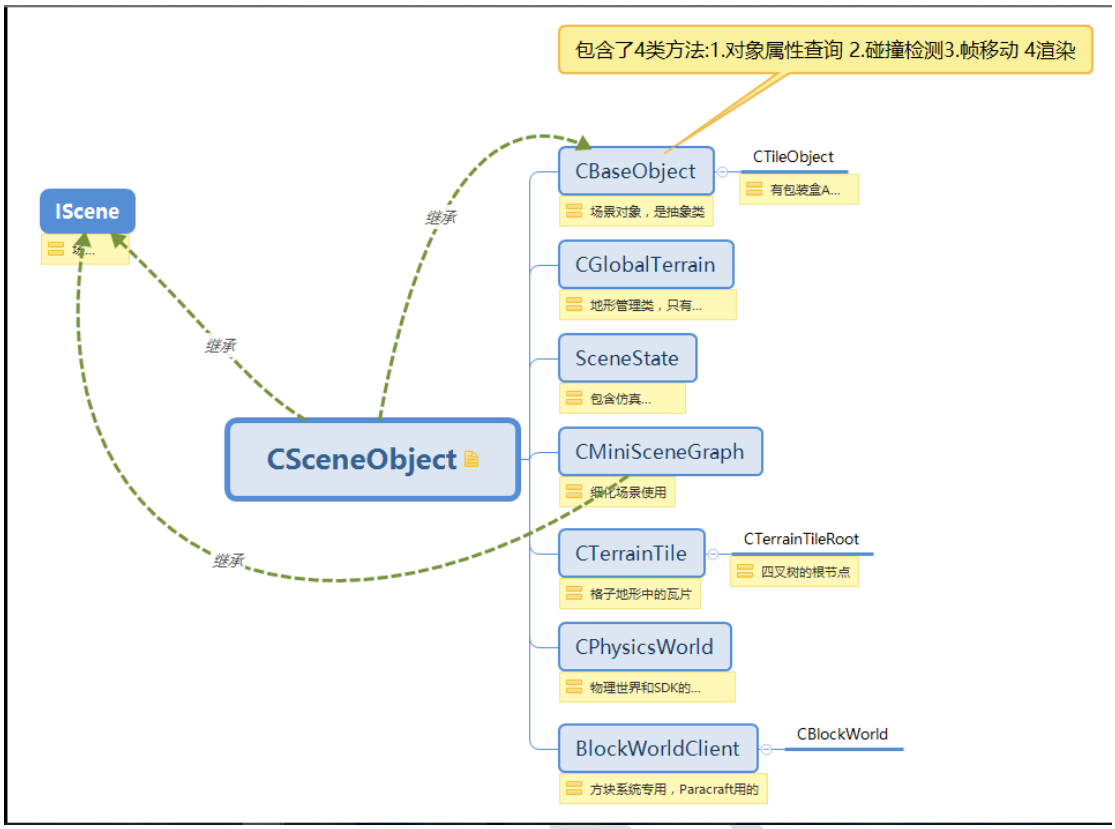

图 7.10　CSceneObject 的类的联系图

7.2.2.2 CMiniSceneGraph

微场景图，是简化版的 CSceneObject，它不是用空间分区（即 CSceneObject 中的对象存储方法）来存储物体对象，它只是实现了一个父子关系，大部分场景的对象可以附着到这里，我们也可以对这些增加，删除，显示或者隐藏这些对象。由于没有复杂的数据结构，因此只用于管理少量的 3d 物体。在微场景图中的物体会在主场景渲染结束后渲染。

这里主要实现对动态物体的组织和渲染，对于动态场景，不适合实时建立包含关系的树形结构，更需要根据运动关系建立父、子节点相关联的场景树。

例如：1 简单的雪和雨，我们可以在摄像机周围创建新的粒子系统.2 现实安装点，或者附着点。3，显示 3D 光标，例如显示鼠标拉拽的效果

主要参数：

1）m_pBatchedElementDraw 绘画特殊物体，例如线，粒子等。

2）m_SunLight，太阳光参数。

3）m_bRenderFog，m_dwFogColor，m_FogColorFactor，m_fFogStart，m_fFogEnd，m_fFogDensity 雾的绘制参数。

4）m_skymeshes，存有 CSkyMesh 的指针，CSkyMesh*数组

5）m_sceneState，场景状态信息，用于记录场景中的物体的渲染以及动画信息。

6）m_pCamera，是画布摄像机，用于 3d 画布上物体的呈现。

7）m_name_map，基础场景对象存储数组，用 map 这个数据结构。

主要方法分类：

1）设置是否渲染天空盒子，创建/删除天空盒子。

2）设置/获取雾的参数。包括雾的颜色因子，开始/结束的地方，密度等。

3）对太阳光的操作。获取太阳光，设置光照是否实现。

4）有关相机的设置：设置玩家角色，用摄像机关注这个角色。获取摄像机。摄像机变焦，移动，旋转等操作。

5）渲染前的预先设置，包括场景和场景中的物体。获取渲染区域，渲染的优先等级等等。

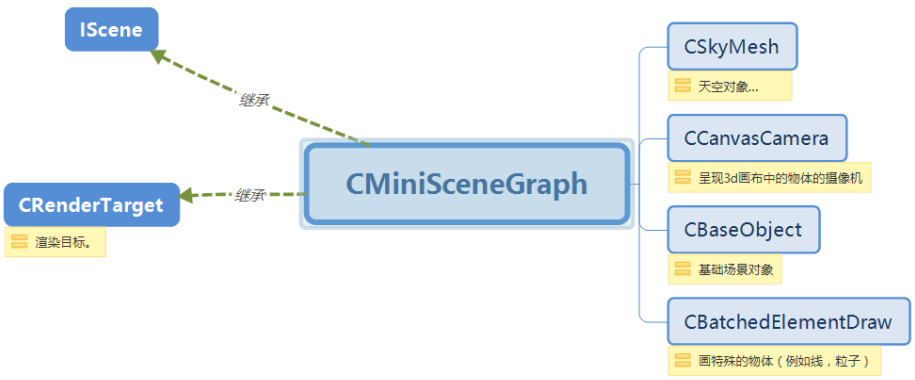

图 7.11　CMiniSceneGraph 类的联系图

7.2.2.3 区域节点

Portal 系统是一种内景渲染技术，让你的场景可以容纳更多的多边形. 它的基本原理是将 3D 空间划分许多通过门(Portal)联通的凸空间(Room). 摄影机所处的空间只能通过门看到其他凸空间中的内容. 这样!ParaEngine 的渲染系统将只需渲染当前房间以及能见的门所联通的房间中的能见的物体，这样大大减少了需要渲染的物品数量。可以实时的渲染宏大的内、外景混合场景. 而 ParaEngine 的动态 Portal 内景技术使得包含 Portal 的场景可以实时的编辑.

CzoneNode，就是在 Portal 内景渲染技术中，表示一个空间的类（room/region），在这里我们假设一个区域不会包含另一个区域。

它包含了：

1）m_planes，表示组成空间网格的平面。

2）m_vPos，当前在世界坐标系中的位置。

3）对空间平面的生成/填充/设置/获取。

4）对空间的设置。在当前节点中添加/移除空间节点。

5）对空间设置/移除门。

6）渲染方法，draw

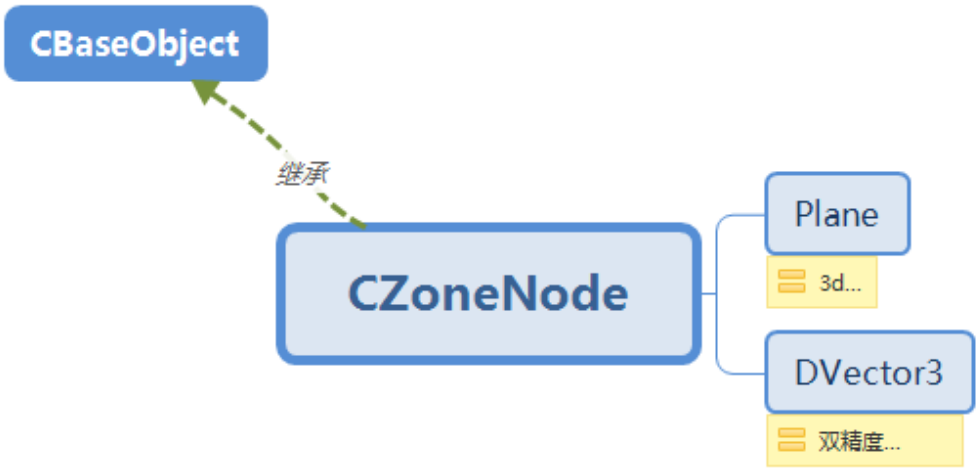

图 7.12　CZoneNode 类的联系图

7.2.2.4 视口管理

Viewport，视口，是基于屏幕的一个概念。它表示了能显示的具体场景。CViewport，就是一个包含了视口渲染的区域信息的类。

CViewportManager，视口管理器，是管理这些视口信息的类。有了这些视口信息，就可以在屏幕上指定位置渲染出场景中的画面。

在视口管理类中有：

1）m_viewportList 和 m_viewportSorted，所有的视口扣对象的集合。

2）m_viewport，是 ParaViewPort，里面有视口的基本属性，例如长宽高等等。

3）更新方法，UpdateViewport，当窗口大小发生变化，根据这个变化改变视口大小。

4）渲染所有视口的方法 Render

5）添加新的视口 CreateGetViewPort。

图 7.13　CZoneNode 类的联系图

7.2.2.5 电影播放

CMovieCtrler，是控制角色根据特定的电影文件做出对应的动作，以求达到电影效果的控制类。简单说这个类就是电影的播放器和记录器。

在 CMovieCtrler 中有位置帧，动作帧，效果帧并且提供方法来记录这些帧

```
/** Position key frames of the movie
* if one want to define a looped movie, just insert the first frame to the end of the movie key frames.
* so that it will loop from the beginning.
*/
MovieKeyFrame<PosKey> m_keyframesPos;
/** all positions in the position key frames are relative to this location.*/
Vector3 m_vPosOffset;
/** action key frames of the movie
*/
MovieKeyFrame<ActionKey> m_keyframesAction;
/** effect key frames for the current biped. */
MovieKeyFrame<EffectKey> m_keyframesEffect;
```

有 MovieMode 来表示当前电影的状态

```
enum MovieMode
{
    MOVIE_PLAYING,
    MOVIE_RECORDING,
    MOVIE_SUSPENDED
};
```

CMoviePlatform，管理和渲染电影（一系列的片段构成电影）。

这里有 m_pMovieCodec，是 IMovieCodec，而 CMoviePlatform 播放的就是 IMovieCodec。

提供了控制记录和播放的方法：

1）设置/获取抓取的 GUI，

2）设置/获取电影名，

3）设置/获取电影在屏幕上的宽度和高度，

4）设置电/获取影在屏幕中的大小，

5）开始和结束，暂停/继续抓取电影

6）设置电影专区 fps。

JoreamTech

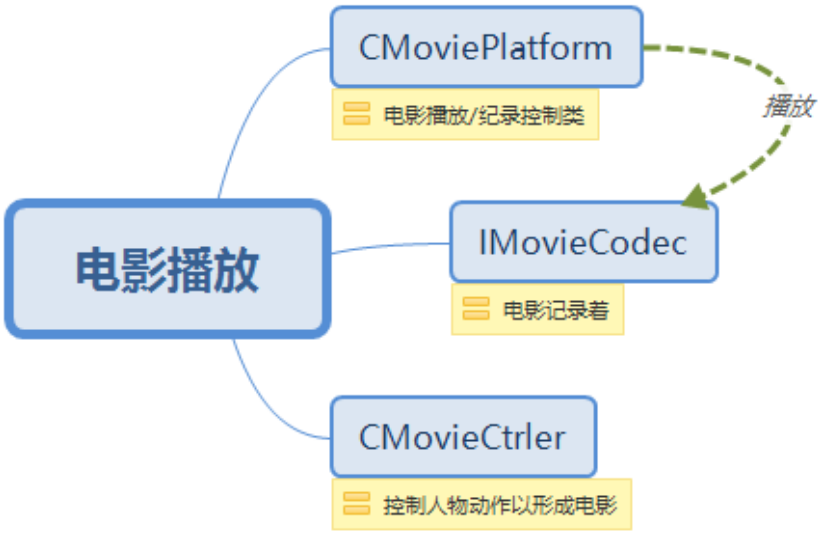

图 7.14 电影播放

7.2.2.6 物理空间

CPhysicsWorld 是物理世界和 SDK 的包装/管理类，它包括了对物理场景的管理方法。环境模拟器从这里可以检索物理场景对象。

主要方法：

1） 初始化物理场景。

2） 退出物理场景，这个方法会先删除所有场景中的对象，然后删除自己。还会停止 SDK。

3） 通过网格实体在物理世界中创建静态角色，释放角色。

4） 获取物理接口。

主要属性：

1） m_pPhysicsWorld，主要物理接口

2） m_listMeshShapes，组成物理实际的形状

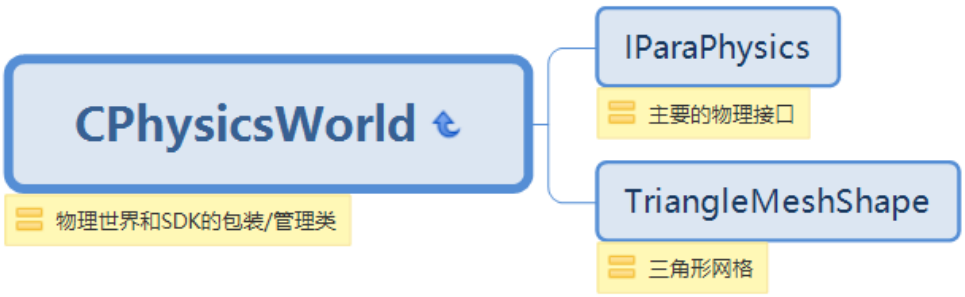

图 7.15 物理空间

7.2.2.7 选中对象管理器

CselectionManager，有 CSelectionGroup 属性。而 CSelectionGroup 是实现了一个由 SelectedItem 的向量表，

```
/** all items in the selection group. */
vector<SelectedItem> m_items;
```

用于存储所有被选中的 SelectedItem，并且提供了一些方法来操作这个表。

SelectedItem 是表示选中对象的结构体。

CselectionManager 用的是单例模式。他提供了添加/删除等方法来管理这些选中对象，

```
/**
* remove a given object in all selections.
* @param pObject pointer to the object to delete
*/
void RemoveObject(CBaseObject* pObject);
void RemoveObject(CGUIBase* pObject);


/**
* Add a new object to a given group. An object may belong to multiple groups.
* @param pObject pointer to the object to add
* @param nGroupID which group the should be added to. be default it is added to group 0.
* group ID must be smaller than 32.
*/
void AddObject(CBaseObject* pObject, int nGroupID = 0);
void AddObject(CGUIBase* pObject, int nGroupID = 0);
```

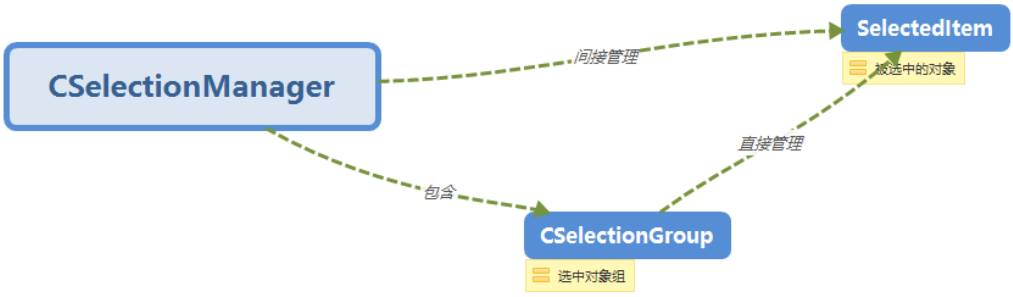

图 7.16　选中对象管理的类的联系图

7.2.2.8 场景对象拾取

场景对象拾取是在场景中通过鼠标拾取选中的物体。基本原理是射线拾取，就是通过鼠标点击屏幕产生的坐标，通过变换到 3D 坐标空间，然后从摄像机原点与该 3D 空间坐标产生一个射线（ray），通过该射线与 3D 世界中的物体做相交检测，即可实现 3D 物体的拾取。

其中涉及到视口坐标转换到世界坐标，射线相交法拾取对象。

```
*/
bool PickObject(const CShapeRay& ray, CBaseObject** pTouchedObject, float fMaxDistance=0, OBJECT_FILTER_CALLBACK pFnctFilter=NULL);
/** @see PickObject() above */
bool PickObject(int nScreenX, int nScreenY, CBaseObject** pTouchedObject, float fMaxDistance=0, OBJECT_FILTER_CALLBACK pFnctFilter=NULL);
```

其中有 PickedObject 结构体来表示被拾取的对象。拾取方法放在 CSceneObject 中。

7.2.2.9 地形空间

CTerrainTile，地形空间是分格地形中的一个块。这个类既是一个管理类也是一个组成类，首先他作为一个节点类，表示四叉树地形中的一个节点

```
/// -- data structuring
#define     MAX_NUM_SUBTILE     4
CTerrainTile*           m_subtiles[MAX_NUM_SUBTILE];
```

其次他作为管理类，提供了许多方法来对这个四叉树操作。

```
/
CBaseObject* SearchObject(const char* pSearchString, int search_mode=0, int reserved = 0);
/** get the object by its name. If there have been several objects with the same name,
* the most recently attached object is returned.
* @param sName: exact name of the object
*/
CBaseObject* GetObject(const string& sName);

/** destroy an object by its name. If there are multiple object with the same name, they will all be deleted.
* @param sName: exact name of the object
*/
void DestroyObjectByName(const char* sName);
```

查找对象，获取对象，销毁对象等等方法

7.2.2.10 天气效果

WeatherParticle 天气效果用于产生对应天气对象的例子效果。在这个类中的命名域中定义了天气类型的枚举类型

```
enum WeatherType {
    WeatherType_Rain = 0,
    WeatherType_Snow,
    WeatherType_RainSplash,
    WeatherType_Last,
};
```

还定义了不同天气的例子效果添加方法，以及对应的渲染方法

```
WeatherParticle* AddParticle(float x, float y, float z, WeatherType weather_type);
```

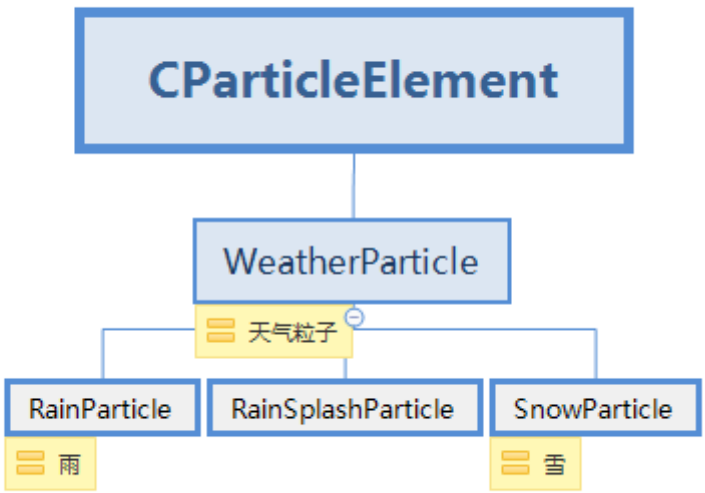

图 7.17 天气效果的类联系图

7.2.3 场景组成类

场景是由许许多多的对象组成的，这些对象都有一些基本属性，他们被定义在 CBaseObject 这个类中。而场景的组成类继承 CBaseObject，是 CTileObject（空间对象）。场景中所有组成场景的对象都需要继承 CTileObject。

他有：

1） 摄像机

2） 人物对象

3） 网格对象

4） 网格物理对象

5） 覆盖层对象

6） 天空网格

等等组成类。

7.2.3.1 空间对象

CTileObject 含有对称轴属性，空间中要被渲染的物体都需要继承此类。它有 AABB 属性，位置和包装盒子。

主要方法：

1） 设置/获取包装盒。

2） 设置 AABB 属性，OBB 属性

3） 获取对象的位置

主要属性：

1） m_vPos，对象的位置，在该对象的包装盒的底部中心。

DreamTech

2） m_aabb，包装盒

3） m_fYaw，朝向信息

4） m_fRadius，包装半径

7.2.3.2 摄像机

CBaseCamera 是摄像机的基类，它记录了鼠标和键盘的。

CFirstPersonCamera 是第一人称摄像机。他有偏航角，俯仰角，但是没有滚动角。

CAutoCamera 是自动跟随摄像机，它可以自动的在第一人称摄像机，第三人称摄像机，旋转摄像机之间转变。还可以把它设置跟随一个角色。

对于一个摄像机应该包含：

1） m_frustum，摄像机的视窗。包括对象视窗，阴影视窗，门视窗。

2） m_fog_plane，雾结束的平面。

3） m_mView，m_mProj 视窗矩阵和映射矩阵。

4） 键盘鼠标的输入信息。

5） 摄像机的位置运动参数，例如：速度，加速度，角度等；

6） m_mCameraWorld，摄像机的世界矩阵

dreamTech

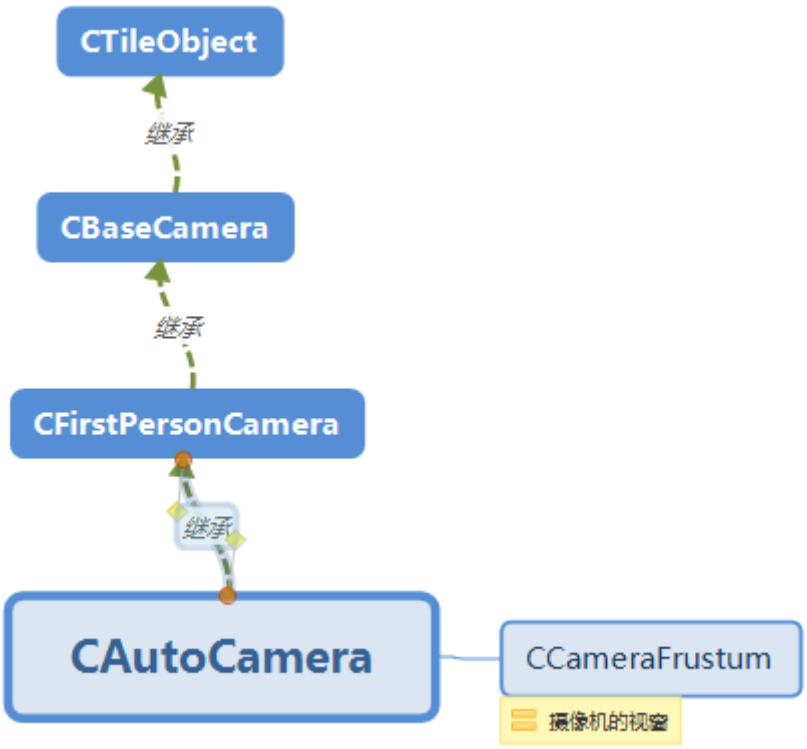

图 7.18　CBaseObject 类的联系图

7.2.3.3 人物对象

CBipedObject，用来表示一个人物对象。这个类和它的子类会让它感应周围环境，并且实现运动和移动。它提供了很多很多虚函数供其他模块调用，因此它的子类实现这些方法会很简单。

人物对象还可以连接到用于渲染 3d 动画的 MultiAnimationEntity 或者 MDXEntity。AI 控制模块也会连接到任务对象上面，用于任务对象的自我控制。

人物对象的行为是像所有基础对象一样，通过事件按控制的。高等级的事件可以分配给人物对象去控制该对象的所有事情。例如播放新的动画，走到新的位置，或者分配智能任务。

主要任务：

1） AI 控制，有 m_pAIModule 属性，是 CAIBase 的实例。提供了替换/获取/使用当前 AI 模块的方法。

2） 脸向控制，有 m_fFacingTarget 属性，有设置朝向的方法。并且能获取朝向信息。

3） 根据朝向，向前/向后移动到指定位置。

4） 设置以及获取人物对象运动的各种信息（速度，朝向， 位置等等）

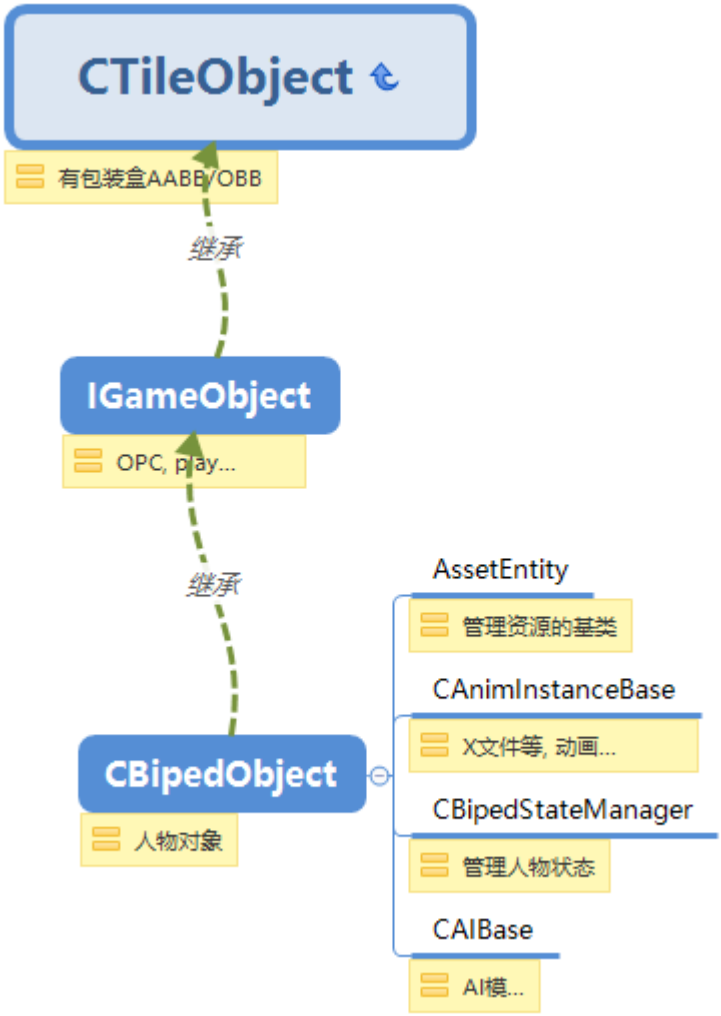

图 7.19 CBipedObject 类的联系图

7.2.3.4 网格对象

CMeshObject 静态的网格对象，只支持 X 文件的资源，并且没有动画。当然，静态网格对象可以被光照和纹理。例子：地形网格，地板，石头，屋子。

这个类实现了：

1）网格对象的初始化和删除。

2）播放这个对象的动画。

3）渲染。

4）m_pViewClippingObject，设置/获取视觉剔除对象。

5）m_ppMesh，m_ReplaceableTextures。对网格和纹理的操作

6）m_XRefObjects，对 x 动画文件的操作

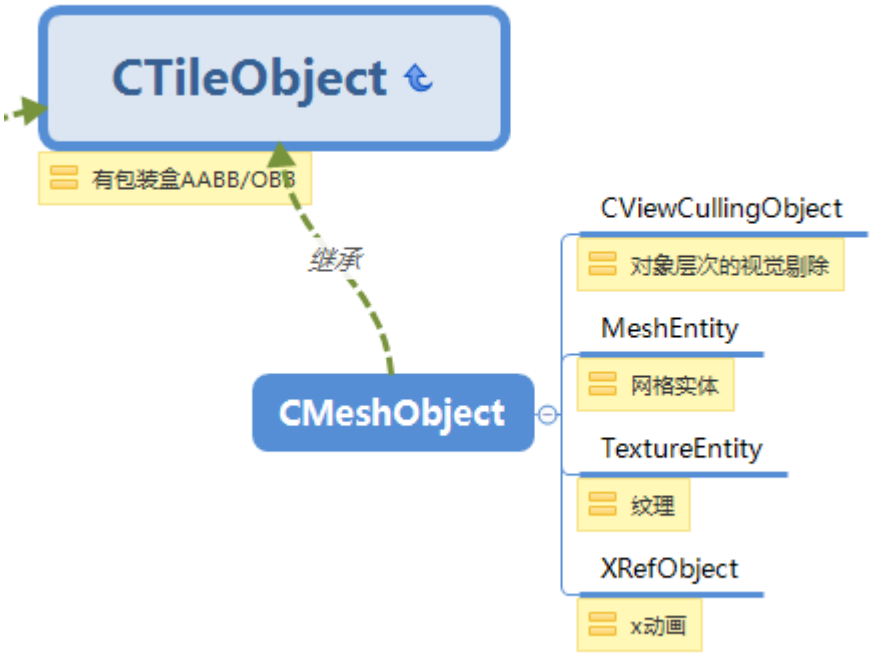

图 7.20　CMeshObject 类的联系图

7.2.3.5 网格物理对象

CMeshPhysicsObject，是基于物理场景对象的静态网格。并且将网格对象作为成员。在这个类中有：

1）m_pMeshObject，是网格对象的指针。

2）m_staticActors，静态物理角色的指针。

3）设置物理组 ID，默认值是 0，最大不超过 31

4）初始化对象。对象的旋转，放缩，重置。

5）加载/释放物理对象。

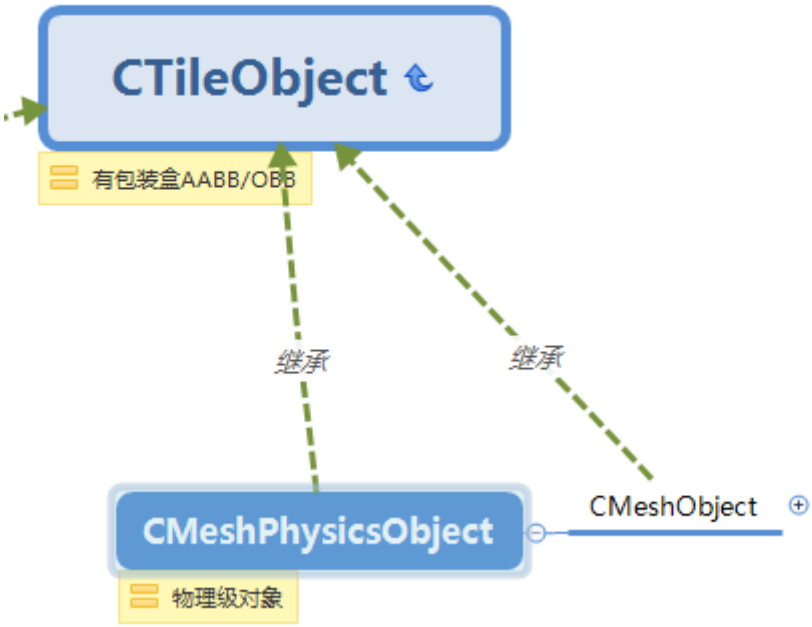

图 7.21 CMeshPhysicsObject 类的联系图

7.2.3.6 覆盖层对象

COverlayObject，覆盖层对象提供了定制好的 3d 绘制策略。它经常用于渲染辅助对象，比如移动/旋转/伸缩变换的操纵器。也用于内部 3d 对象的顶部展示接口和 GUI 的 API。除了绘制线和三角形，这个类可以渲染所有 GUI 对象，甚至提供和 3d 场景的交互。

里面用到的熟悉和方法：

1）m_pGUIObject，是 CGUIButton 的指针。

2）预渲染方法，通过这个方法将对象丢入渲染队列

3）检测/设置是不是空间对象。

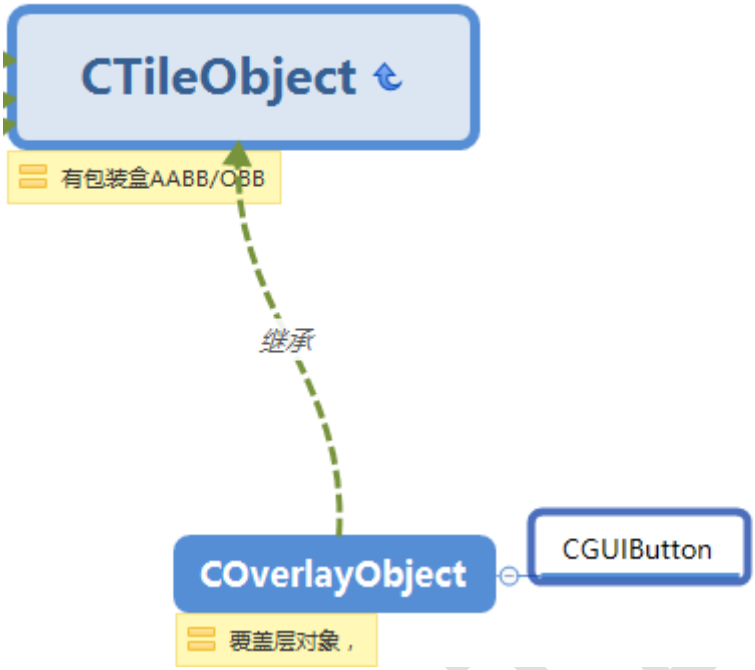

图 7.22 COverlayObject 类的联系图

7.2.3.7 天空网格

CSkyMesh，场景中的天空对象。天空有很多种，天空网格可以自己适应不同的雾参数。在渲染过程中天空网格是第一个被渲染的物体。

在这个类中：

1）首先设置了天空网格文件， 设置纹理，天空颜色因子。

2）提供了天空盒子的渲染方法。

3）天空网格的创建。

4）模拟天空的数据和方法。

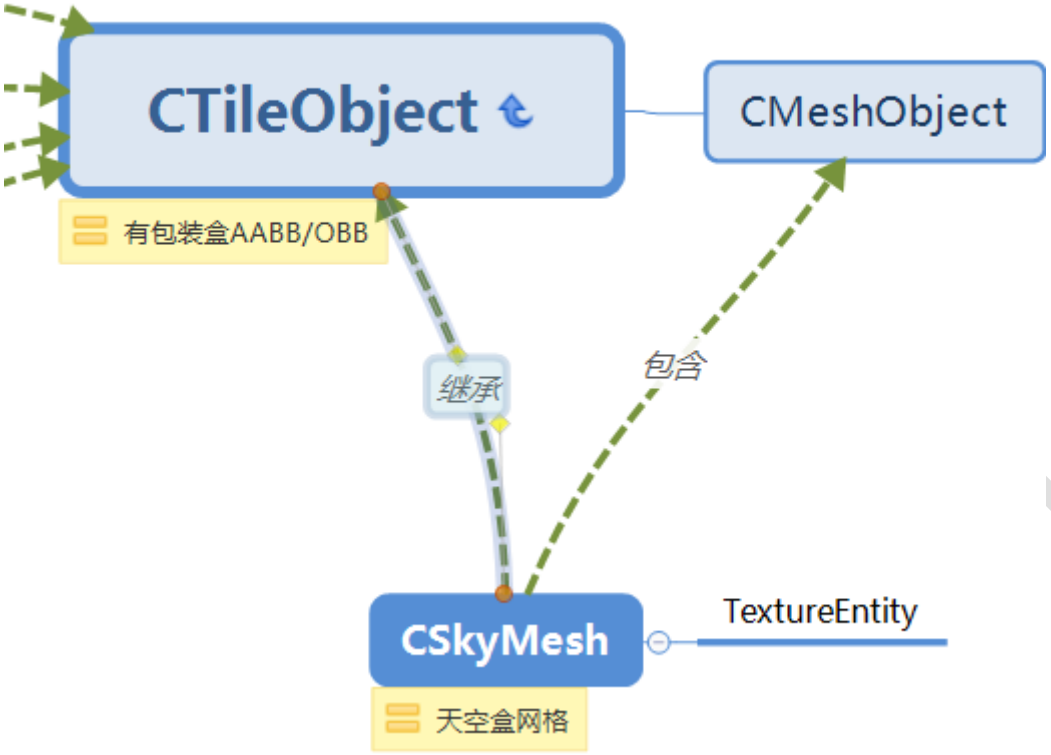

图 7.23　CSkyMesh 类的联系图

7.2.4 地形

3d 引擎中的地形是由一张高度图（heightMap）构成的，高度图通常是一张二值图，大小为（2^n+1）。它的每个像素的灰度值表示地形每一点的高度值。用这些高度值作为顶点信息绘制三角面片。

为了高效，实时绘制地形，Clark 提出了层次细节模型（LOD）。这个模型认为当物体覆盖屏幕较小区域时，可以使用该物体描述较粗的模型。

基于 LOD 的地形网格简化算法分为动态 LOD 和静态 LOD。在 NPL 中使用的 ROAM 算法（实时优化自适应网格），这个算法就是动态的 LOD。而 GeoMipMap 算法是静态 LOD。在 NPL 中提供了了基于四叉树的 ROAM 算法和 geomipmap 算法，可以根据不同需求使用这两种算法。

ROAM 算法的思想是在对地形现实的时候，根据视点和视线的位置和方向等多种因素，对地形表面的三角片面进行三角形的分裂和合并，最终形成与原始表面近似的简化表面。

ROAM 算法的基础是等腰三角形可以从直角顶点到斜边引一条垂线，将这个等腰直角三角形分成大小相等的两个小等腰直角三角形，并且可以无限递归下去。

在 NPL 中用四叉树来组织描述地形，先把可视范围内的地形分割成 4 等份矩形子块，依靠计算判定影子来检测 4 个子块，如果检测到某个子块的网格精度达到绘制要求，就不再往下分割；否则就继续分割为更小的子块，知道所有子块的矩形网格达到渲染精度。

如果在 NPL 中使用 GeoMipMap 算法模拟地形，那么就要地形必须是(2^n+1)*(2^n+1)。GeoMipMap 算法实际是把地形在 xz 平面分块（block），每块 block 都有自己的层次细节等级，边长满足 2^n+1，用不同的分辨率网格模型来描述。

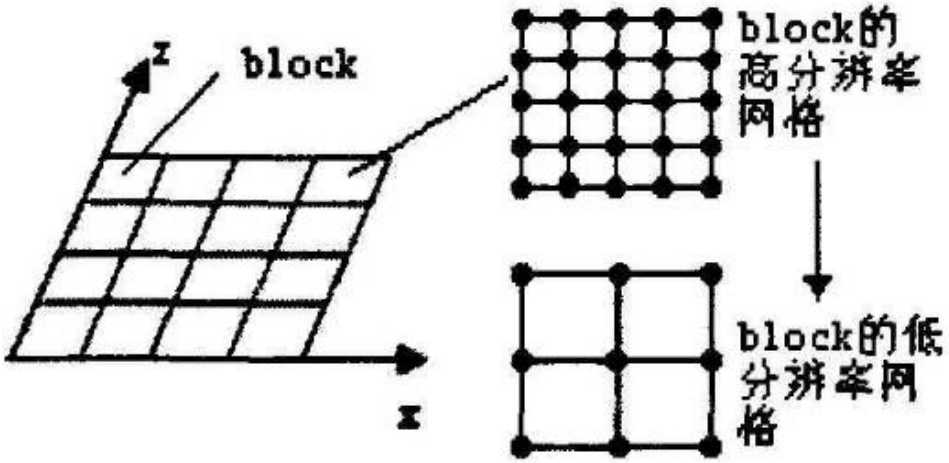

图 7.24　GeoMipMap 算法分割地形示意图

层次细节中基于四叉树的 ROAM 算法比较考验 CPU 的计算能力，而 GeoMipMap 算法则将计算中心放在 GPU。

7.2.4.1 地形的组成

NPL 地形中，无论哪种方法都是将 TerrainBlock 类作为层次细节算法的节点类。在这个类中定义了他的子节点和父节点：

```
//子节点 和父节点
TerrainBlock * m_pParent;
TerrainBlock **m_pChildren;
```

如果使用 GeoMipMap 算法则还有：

```
//geoMipmapCode
IndexInfo* m_indexInfo;
bool m_useGeoMipmap;
int m_chunkCountX;

int m_lodLevel;
GeoMipmapChunkType m_chunkType;
```

等信息。

并且提供了构造不同三角带的方法

```
TriangleStrip* CreateGetTriangleStrip(int nIndex, Terrain * pTerrain);
TriangleFan* CreateGetTriangleFan(int nIndex, Terrain * pTerrain);
```

还有修复裂缝的方法

```
/** repair cracks in this block and child blocks. This is a recursive function.*/
void RepairCracks(Terrain * pTerrain, uint32 *pCountFans);
```

7.2.4.2 地形的管理

地形管理中主要用到了 Terrain 和 CGlobalTerrain 这两个类。在 Terrain 中实现了 ROAM 算法的主体里面有四叉树根节点：

```
/// this is the main structure for ROAM algorithm
TerrainBlock *m_pRootBlock;
```

Terrain 中，实现了对上诉两种方法的算法支持，都提供了许多方法。首先是对基于四叉树的 ROAM 算法，有分割和合并方法，还有裂缝修复等。

```
/**
* Breaks the Terrain down into triangles strips.
* Based on the current viewing parameters, this method breaks the terrain down into a
* visually optimum set of triangles that can be rendered. Normally, an application will never
* call this method directly, but will instead call the method ModelViewMatrixChanged() to allow
* Demeter to take care of this automatically.
*/
int Tessellate();
/**
* repair cracks during tessellation.
* the triangle strips built from Tessellate() will has cracks between neighboring triangles of different size.
* in these cases, the bigger triangle strips are replaced with triangle fans, so that there will be no cracks.
* Here "cracks" means that the surface is not a continuous surface.
* @see Tessellate()
*/
void RepairCracks();
```

在地形管理流程:

Geomipmap 的管理流程（证据不足版本）：

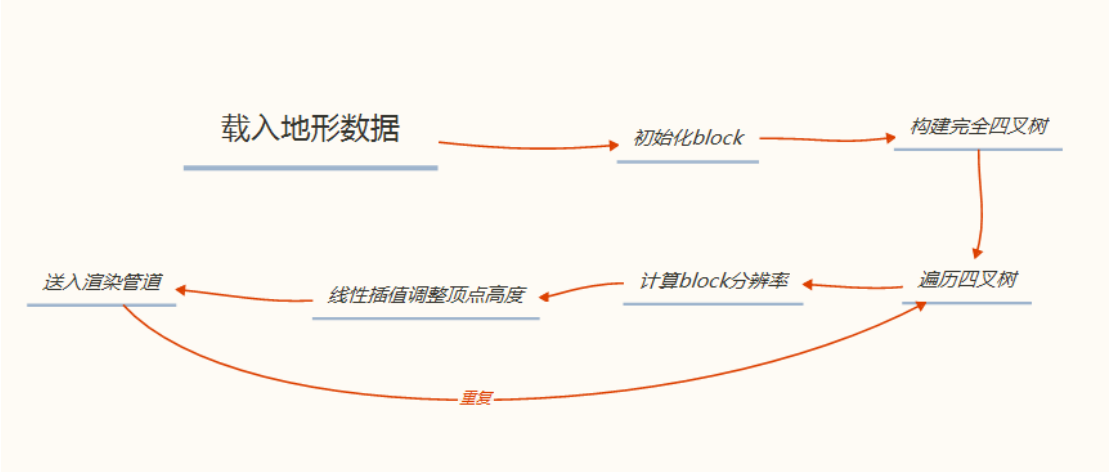

图 7.25　GepMipMap 的管理流程

基于四叉树的管理流程：

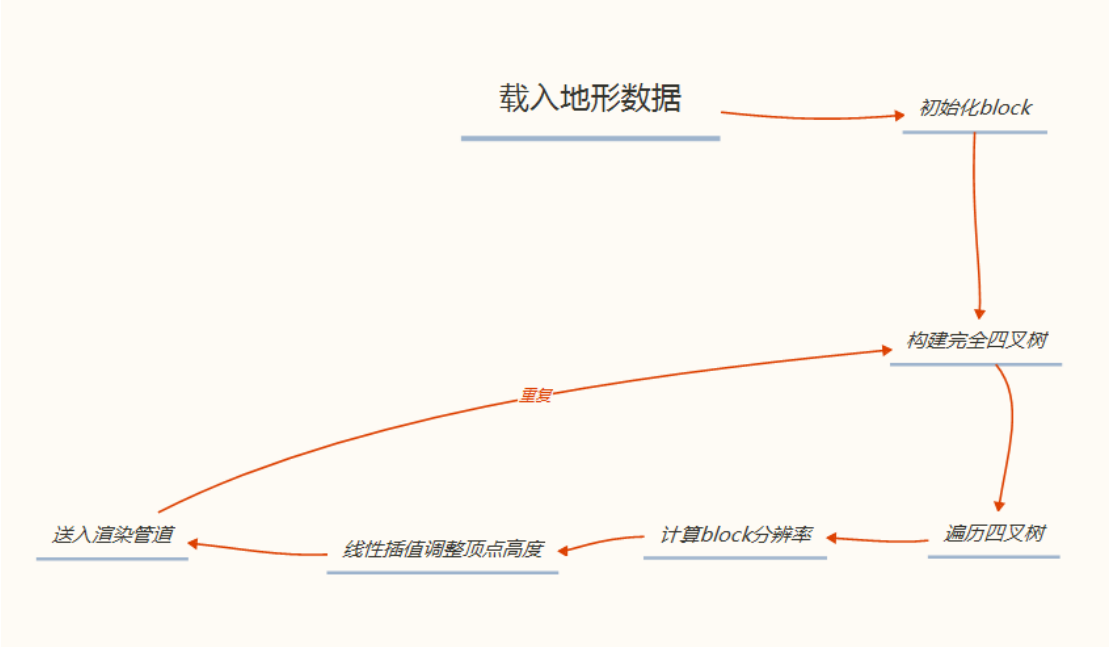

图 7.26　四叉树管理流程

7.3 渲染

7.3.1 渲染过程

渲染过程主要由 CSceneObject 类的 AdvanceScene()方法实现，它的渲染步骤如下图所

示：

IdreamTech

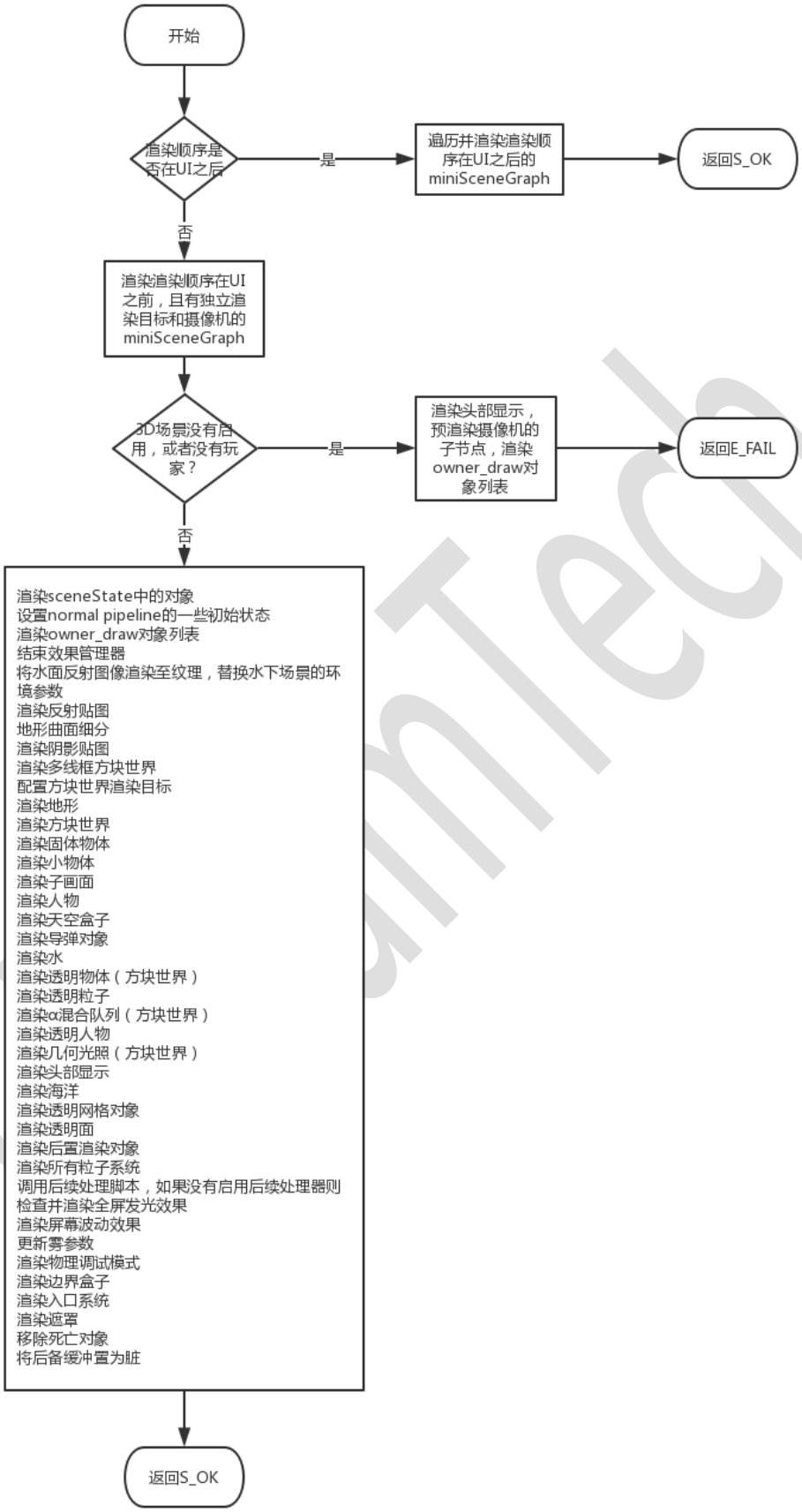
开始
渲染顺序是否在UI之后
是
遍历并渲染渲染顺序在UI之后的miniSceneGraph
返回S_OK
否
渲染渲染顺序在UI之前，且有独立渲染目标和摄像机的miniSceneGraph
3D场景没有启用，或者没有玩家？
是
渲染头部显示，预渲染摄像机的子节点，渲染owner_draw对象列表
返回E_FAIL
否
渲染sceneState中的对象
设置normal pipeline的一些初始状态
渲染owner_draw对象列表
结束效果管理器
将水面反射图像渲染至纹理，替换水下场景的环境参数
渲染反射贴图
地形曲面细分
渲染阴影贴图
渲染多线框方块世界
配置方块世界渲染目标
渲染地形
渲染方块世界
渲染固体物体
渲染小物体
渲染子画面
渲染人物
渲染天空盒子
渲染导弹对象
渲染水
渲染透明物体（方块世界）
渲染透明粒子
渲染α混合队列（方块世界）
渲染透明人物
渲染几何光照（方块世界）
渲染头部显示
渲染海洋
渲染透明网格对象
渲染透明面
渲染后置渲染对象
渲染所有粒子系统
调用后续处理脚本，如果没有启用后续处理器则检查并渲染全屏发光效果
渲染屏幕波动效果
更新雾参数
渲染物理调试模式
渲染边界盒子
渲染入口系统
渲染遮罩
移除死亡对象
将后备缓冲置为脏
返回S_OK

图 7.40　渲染主流程

7.3.2 特殊效果

指除了渲染三维固体物体的渲染管道之上，专门渲染视觉效果的渲染系统，如降雨降雪、水、全屏的雾效果以及粒子效果等。

7.3s.2.1 雾效果

对摄像机来说，雾是两个与视线垂直的平面之间的区域，如下图所示

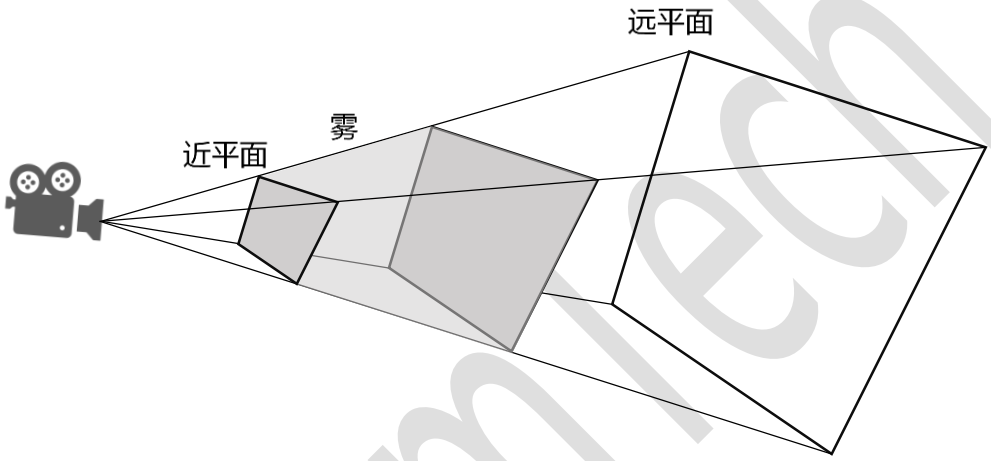

图 7.41 雾

实现雾效果的类：CSceneObject、CMiniSceneGraph

雾的参数：

m_dwFogColor：雾的颜色

m_FogColorFactor：雾的颜色受环境光的影响程度

m_fFogStart：雾的起始距离

m_fFogEnd：雾的终止距离

m_fFogDensity：雾浓度

渲染的时候，如果当前对象的渲染优先级高于 UI，且启用了雾效果，则会清除后备缓冲，并将后备缓冲颜色设置成雾的实际颜色，即雾颜色与雾浓度的积（结果的 alpha 值设为 1）。

Dreamtech

除以上两个类外，还有一些类具有与雾相关的属性和方法：

CBaseCamera	m_fog_plan：最远的雾平面，垂直于视线，到视野近平面的距离为 fogEnd
COceanManager	m_useScreenSpaceFog：默认为 true，摄像机在水下时是否显示全屏的水下效果 m_bDisableFogInReflection：默认 false，渲染反射效果时是否启用雾效果
CSunLight	LinearColor ComputeFogColor()：用环境光计算雾的颜色
Terrain	bool CuboidInFogRadius(const CShapeBox & cuboid) Plane::Side GetCuboidFogSide(const CShapeBox & cuboid) 计算雾与立方体的位置关系

7.4.2.2 水效果

COceanManager，管理了水面效果。在 NPL 中是基于 FFT 的水面模拟。

在水面模拟中，水面是由成千上万个三角片面构成的。模拟水面实际就是将三角片面的高度信息由一系列不同的振幅，频率和传播方向的余弦波的叠加。因此。对于水面某点(x0,y0)的瞬时坐标有由：

$$z = z_0 - \sum_{i=1}^{n} A_i \cos[k_i(x_0 \cos\theta_i + y_0 \sin\theta_i) - \omega_i t + \varphi_i] \qquad (1)$$

计算获得。这个一般适用于平静的水面模拟。

当风作用于水面的时，用 Gerster 模型描述

dreamTech

Gerstner wave函数为：

$$\boldsymbol{P}(x,y,t)=\begin{pmatrix} x+\sum\left(Q_i A_i\times \boldsymbol{D}_i.x\times\cos\left(w_i\boldsymbol{D}_i\cdot(x,y)+\varphi_i t\right)\right), \\ y+\sum\left(Q_i A_i\times \boldsymbol{D}_i.y\times\cos\left(w_i\boldsymbol{D}_i\cdot(x,y)+\varphi_i t\right)\right), \\ \sum\left(A_i\times\sin\left(w_i\boldsymbol{D}_i\cdot(x,y)+\varphi_i t\right)\right) \end{pmatrix}$$

- Q_i为控制波浪陡峭程度的参数；

对于这个算式，用高效的 FFT 算法来计算。

在 COceanManager 中，提供了相关方法来水面模拟：

```
/*****************************************************************************/
/* The FFT computational stages                                              */
/*****************************************************************************/
void animateHeightTable();
void animateNormalTable();
void horizontalFFT(sComplex* pCmpTable );
void verticalFFT(sComplex* pCmpTable );
```

也提供了水面模拟的其他基本方法，例如渲染，初始化水面，等等。

7.4.2.3 阴影特效

阴影特效有两种实现方法：阴影体积（Shadow Volume）和阴影贴图（Shadow Map）

1. 阴影体积（Shadow Volume）

阴影体积指的是光源在物体后方投射出的阴影覆盖的空间，这是一个发散的空间。在阴影体积中的物体表面就会显示出阴影效果，下图中的黄色线框表示的就是阴影体积。

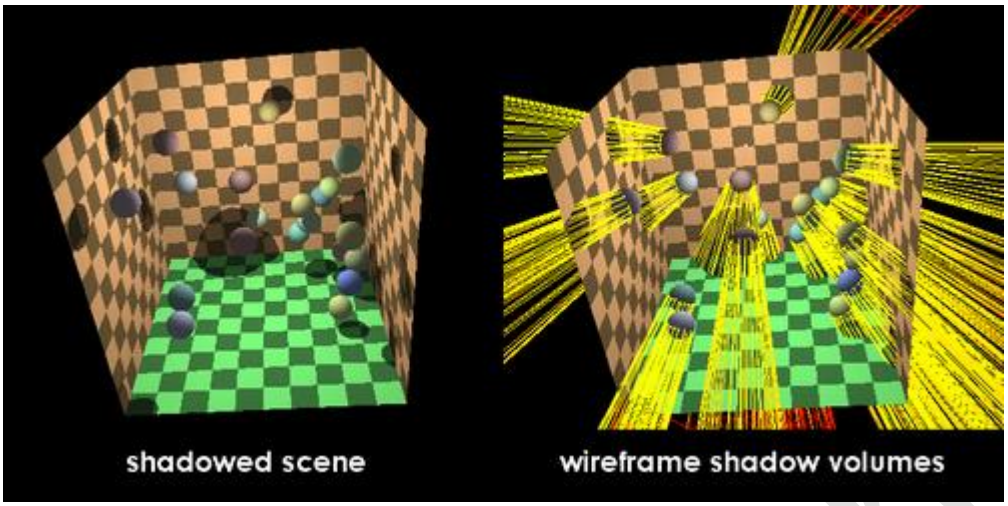

图 7.42 阴影体积

阴影体积使用模板缓冲（stencil buffer）来产生阴影，它类似深度缓冲，存储着屏幕上每个像素对应的一个整数值。首先，计算机会渲染一遍没有阴影的场景，同时填充深度缓冲。然后清空模板缓冲，其中所有像素的值都设为 0。之后渲染阴影体积，如果像素对应着正向面，则模板缓冲的值加一，若对应着背向面，则模板缓冲的值减一。这样在阴影体积外的像素对应的模板缓冲值都为 0，在阴影中的像素的模板缓冲值都为 1。最后我们只对这些值为 1 的像素进行渲染，使其颜色加深。使用这种方式渲染出的阴影无法产生渐变的阴影边缘。

实现阴影体积的类：ShadowVolume

该类保存了阴影体积的所有顶点、光线信息、阴影颜色、阴影方法、遮挡棱锥、当前视口、投射矩阵和视野矩阵，提供渲染方法 Render()

遮挡棱锥是摄像机近平面和近平面的四条边在光线方向延伸出的四个平面构成的空间。

阴影方法有三种，控制阴影的渲染方法（在 SceneObject 的 RenderShadow()方法），如果投影对象在屏幕中太小，则为 SHADOW_NONE，表示不渲染阴影，如果投影对象与遮挡棱锥相交，则为 SHADOW_FAIL，其他情况则为 SHADOW_PASS。

2. 阴影贴图（Shadow Map）

使用阴影贴图时，需要对场景渲染两次。首先，从光源视角渲染场景，把渲染结果的深度缓

冲存储为阴影贴图纹理。然后，以正常方式渲染场景，渲染每个片段时使用阴影贴图判断该片段是否在阴影内。判断方法是，如果该片段自光源的距离比阴影贴图里的对应深度值远，那么边代表该片段被遮挡，也就是位于阴影范围内。这与用深度缓冲判断遮挡的原理一样。

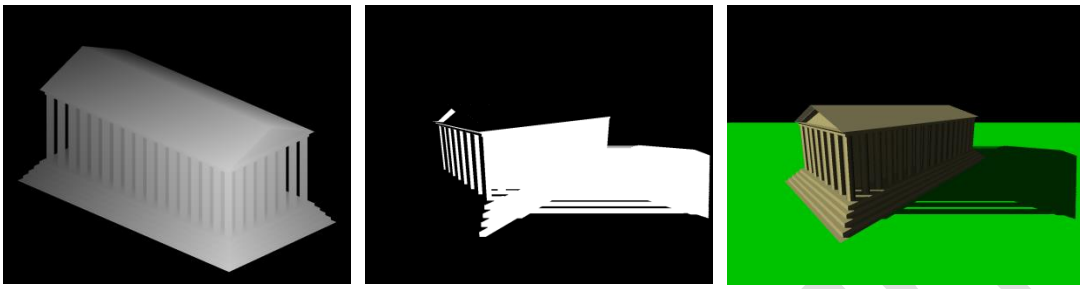

图 7.43　阴影贴图左图是阴影贴图，它是从某点光源十点锁渲染的 z 缓冲内容。中间的图中，黑色像素表示它在光源空间的深度测试失败，即片段在阴影内，白色表示通过测试。右图为最终渲染结果。

阴影贴图相比阴影体积有着更高的效率，而且可是实现渐变过渡的阴影边缘，但是由于受限于纹理的分辨率，它的精确度不如阴影体积。

阴影贴图由 ShadowMap 类实现，它保存了贴图信息、转换矩阵以及其他参数。

重要属性：

m_isShadowType：枚举变量，表示阴影投射方式，有四种取值

SHADOWTYPE_PSM 透视阴影

SHADOWTYPE_LSPSM 光线空间透视阴影

SHADOWTYPE_TSM 梯形阴影

SHADOWTYPE_ORTHO 正交阴影

m_bBlurSMColorTexture：这个值决定是否对全局地形使用模糊的阴影贴图。

m_shadowTexWidth，m_shadowTexWidth：阴影贴图的宽度和高度，有两个默认档位，1024 和 2048，也可以自己设置

重要方法：

BeginShadowPass：构建阴影贴图变换矩阵，设置渲染阴影贴图的渲染状态和效果状态

SetShadowTexture：把阴影贴图传递给指定的 effect file 对象，使对应的物体正常渲染。

7.4.2.4 粒子特效

粒子系统是用来渲染无固定形状的物体的，如烟、火花、雨、雪等动态物体，毛发、草等丝状物体。这些统称为粒子效果。粒子效果有以下几个特点：

- 粒子系统由大量简单的几何物体组成，这些物体通常是四边形片面。
- 这些物体通常是朝向摄像机的，阴影必须确保这些片面的法向量始终朝向摄像机焦点。
- 物体的材质几乎都是半透明的。因此它们有严格的渲染次序。
- 粒子以多种丰富方式表现动画。它们的位置、大小、纹理坐标等许多参数都是于每帧有所变化的。对于静态物体，粒子生命周期中所有形态是一次性全部渲染的，这样能显示出粒子的轨迹，模拟丝状物（见下图）。
- 用于表现动态物体的粒子会不断出生和湮灭。粒子发射器会以设置的速率创造粒子，并将其初始化。粒子湮灭的原因有：碰到预先定义的死亡平面、已超过设定的最大存活时间、或者其他用户定义的条件。

IreamTech

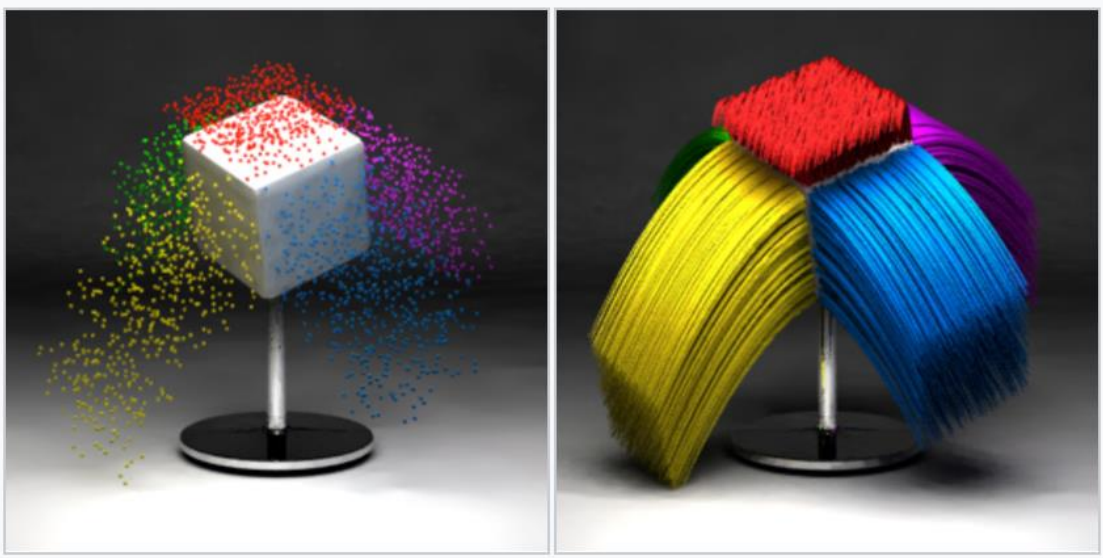

图 7.44　粒子效果，左图表示一个发射 5000 个运动粒子的方块，这些粒子在 y 轴方向遵循引力法则；右图表示用静态粒子渲染的同一个方块

CParticleElement 是所有粒子必须实现的接口，粒子以纹理的形式存储，所有的粒子的实现类必须实现 RenderParticle()和 GetTexture()方法，该接口只声明了一个纹理属性。

实现了 CParticleElement 的 5 个类如下图所示：

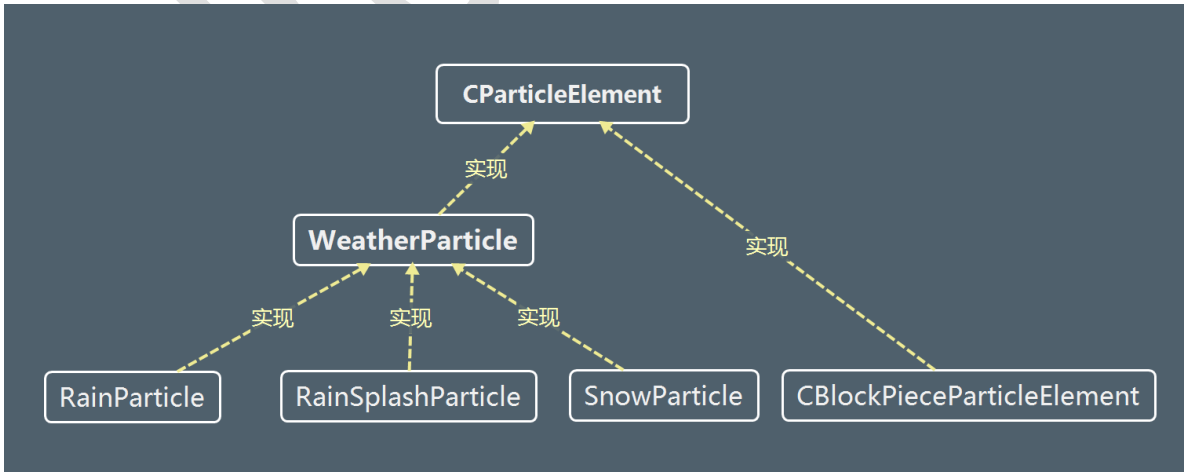

图 7.45　粒子系统

WeatherParticle

表示天气粒子的抽象类，实现了绝大部分的需要方法。WeatherParticle 由 WeatherParticleSpawner（即天气粒子发射器）产生及控制。WeatherPartical 包含了粒子的坐标、速度矢量、粒子大小（长度、宽度）、生命时间、最低高度（y 轴）、重力以及纹理等信息。同一个天气粒子发射器产生的天气粒子共用一个纹理，但这些粒子的纹理坐标不尽相同，这个由发射器在初始化粒子的时候在已有的的几组纹理坐标中随机选择。

重要方法：

Init()：初始化粒子，设置位置、初速度、纹理坐标等。

FrameMove()：每帧需要调用的方法，实现粒子的在 y 轴方向上的移动，以及判断粒子是否死亡（落地或者超过最大存活时间）。

RenderParticle()：将粒子的顶点加入顶点缓存

Draw()：绘制天气粒子

RainParticle

表示在空中的雨滴，初始化的长宽为 0.16 和 0.04。在粒子死亡的时候，会在消失的地方（即地面）创建一个表示水花的粒子（RainSplashParticle）。

RainSplashParticle

表示雨滴落在地面激起的水花，初始化时，定义长宽都是 0.1，y 轴方向初速度为 3，x 及 z 轴方向坐标随机，重力加速度为 9.18x2。FrameMove()方法增加了 x 及 z 轴方向的移动。

SnowParticle

表示一个雪花，这个类多了两个属性：

isFloating：bool，表示雪花是否在飘，即是否在 x 及 z 轴方向有速度

JreamTech

float_speed：float 变量，表示在 x 及 z 轴方向的速度

初始化的时候，isFloat 会随机定义成 true 或 false，float_speed 也被设成一个随机数，重力加速度为 9.18 x 0.03，长宽都为 0.03。FrameMove()方法增加了 x 及 z 轴方向的移动（在 isFloat 为 Ture 时）。

CBlockPieceParticleElement

表示方块世界中的动态粒子，该类只有一个属性，是 CBlockParticle 的指针，所有的功能都由这个 CBlockParticle 实现，CBlockParticle 又继承自 CBlockDynamicObject。

CBlockParticle 包含了纹理、纹理坐标的信息，实现了 Draw()和 RenderParticle()方法和根据淡出时间计算不透明度的方法。

CBlockDynamicObject 是方块世界中动态对象的基类，继承自 CTileObject。它包含了物体所有的物理信息（位置、速度等）、存活时间长度以及淡出（fade out）时间。它的 Animate()方法实现了物体每段时间的动作计算，相当于 WeatherParticle 类的 FrameMove()方法。

7.4 方块引擎

方块世界管理器管理着所有的方块世界，每个方块世界由 64 x 64（x 轴，z 轴）个区域组成，每个区域由 32 x 16 x 32（x 轴，y 轴，z 轴）个块组成，每个块又由 16 x 16 x 16（x 轴，y 轴，z 轴）个方块（Block）组成。

IdreamTech

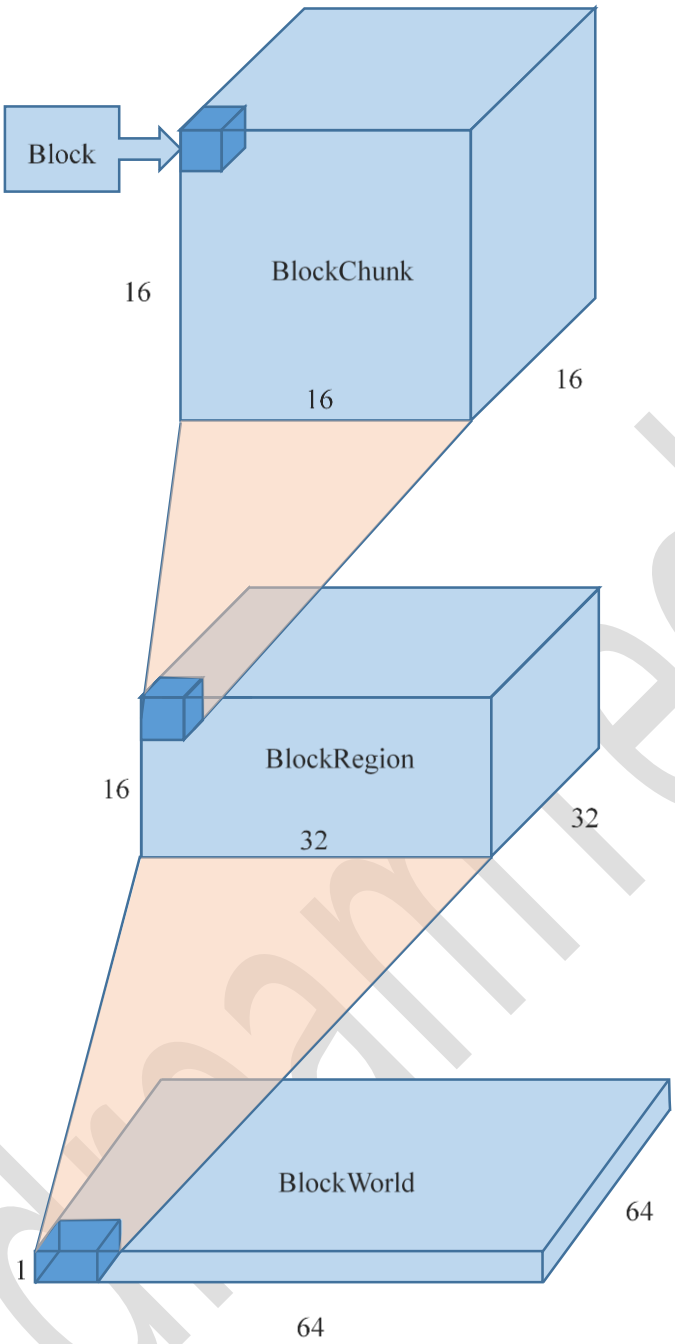

图 7.27　方块世界

方块系统可以分为 4 层来看，同样底层是文件，在方块系统中只有 ParaX 文件， 往上就是方块模型： 5 种模型供选择。有了模型就可以搭建方块和方块世界， 最后便是渲染和显示。

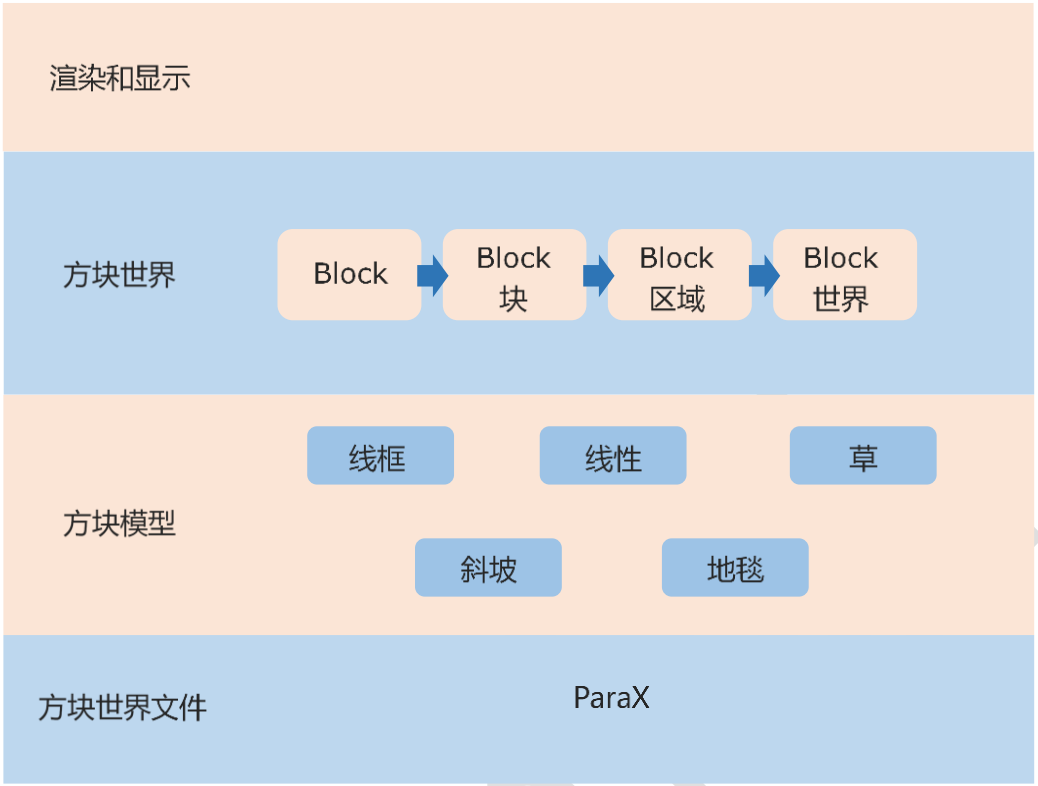

图 7.28 方块系统

7.4.1 BlockWorldManager

用来存放和管理所有创建的 CBlockWorld 实例，提供添加与查找的方法，支持单例模式。该类只有一个成员变量 m_mapBlockWorlds，用于存放所有 CBlockWorld 对象和其名字的对应关系。

7.4.2 CBlockWorld

方块世界，表示一块由 64 x 64 x 1（x 轴，y 轴，z 轴）个 Block 块（Chunk）组成的三维空间。

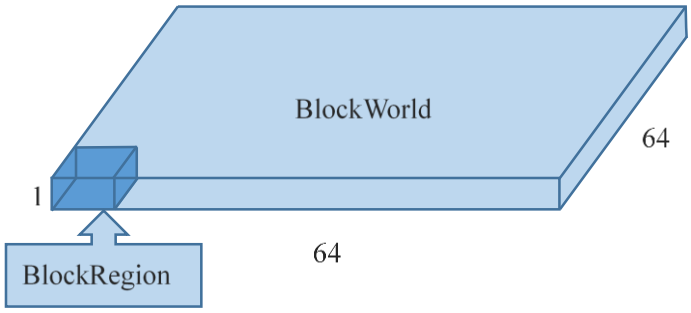

图 7.29 方块世界

它管理世界中所有的方块，负责各种参数的配置，以及相关组件的操作。它保存了当前所有可见 Block 块，即可渲染块（RenderableChunk）的引用，用于渲染操作。方块世界之间是独立的，引擎一次也只能渲染一个世界，也就是说方块世界是最大的三维空间了。CBlockWorld 有一个子类 BlockWorldClient，该类实现了更多关于方块世界渲染的操作，CBlockWorld 相当于是一个借口，实际使用到的是 BlockWorldClient。

重要属性：

m_activeChunks：可渲染块（RenderableChunk）的向量保存了视点（eye position）周围 n 个块的渲染信息。

m_pRegions：64 x 64 固定大小的 BlockRegion*数组，用于存放所有加载了的区域对象。

m_regionCache：一个 int 到 BlockRegion*映射，作为常操作的区域的缓存。

重要方法：

CreateGetRegion()：获取或创建 Block 区域，指定区域不存在时会创建一个 BlockRegion 对象，并将其添加到 CreateGetRegion 数组和 m_regionCache 映射中，再调用 BlockRegion->Load()加载区域。

7.4.2.1 Block 区域

BlockRegion，表示一块由 32 x 16 x 32（x 轴，y 轴，z 轴）个 Block 块（Chunk）组成的三维空间。它保存了这些 Block 块，该区域在方块世界中的坐标，也记录了垂直于 y 轴的平面上每一个位置的方块高度。

dreamTech

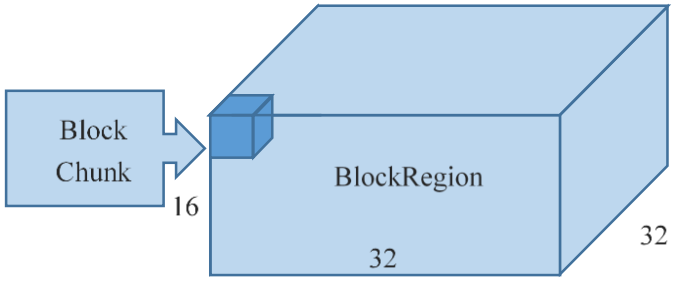

图 7.30 方块区域

该类提供了对其中 Block 和 Block 块的查找、修改等操作，它可从文件中加载 Block 区域，以及之后对区域的更新、卸载。

重要属性：

M_thread：线程变量，用于实现区域的异步加载。

m_bIsLocked：布尔变量，用于实现对区域操作的加锁保护。

m_chunkTimestamp：byte 向量，表示每个 Chunk 列是否已加载。

m_blockHeightMap：保存每个 Chunk 列的高度信息。

重要方法：

Load()：实现 Block 区域的加载，它先执行加载区域前需要执行的相关脚本，然后加载区域，流程如下图。

dreamTech

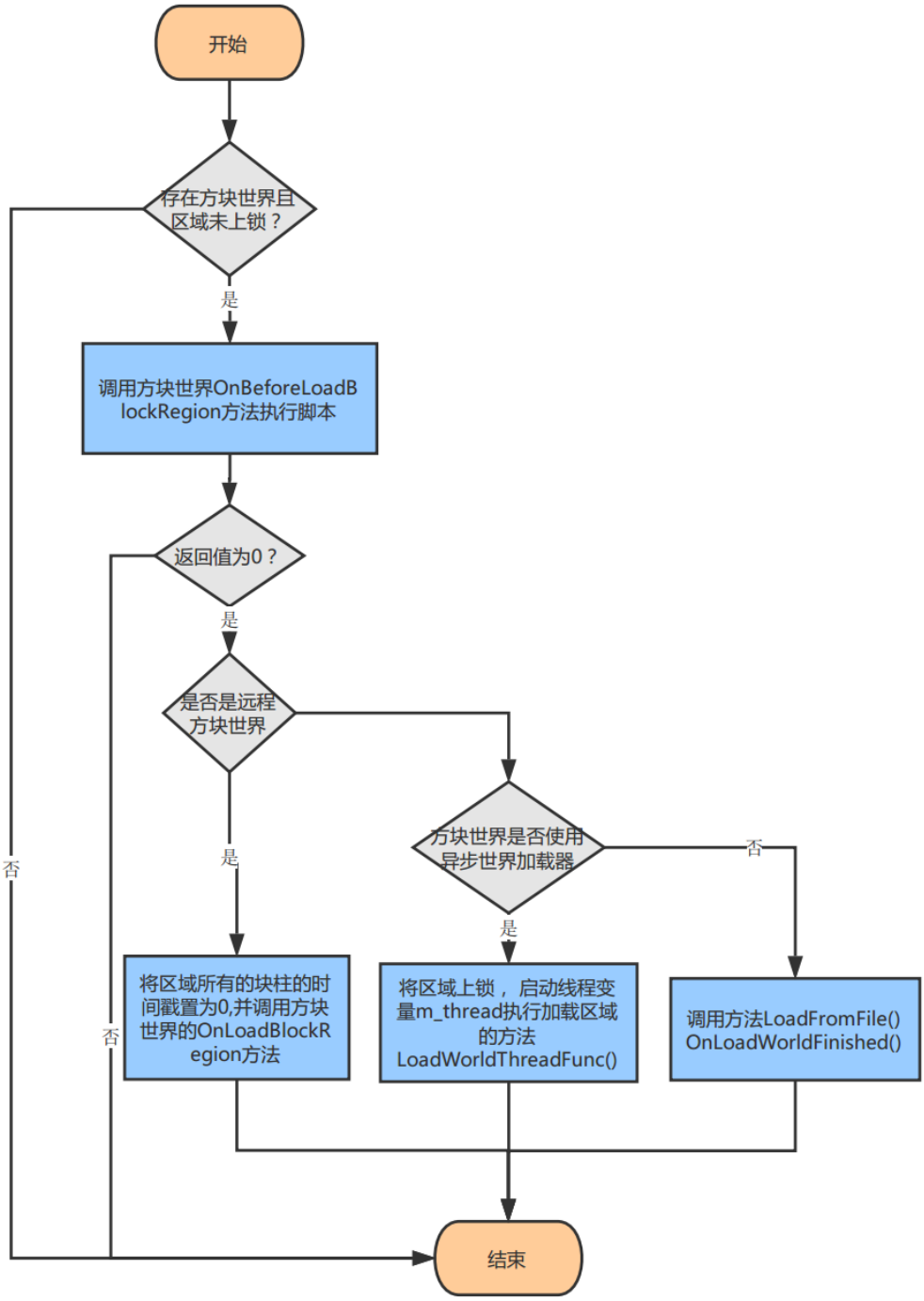

图 7.31　加载 Block 区域流程

LoadFromFile()：从文件加载区域。

SaveToFille()：将修改写入文件。

7.3.2.2 Block 块

表示一块由 16 x 16 x 16（x 轴，y 轴，z 轴）个方块（Block）组成的三维空间。它保存了空间中的方块坐标与对应的方块（Block）对象、光线信息，以及它在方块世界中的位置。

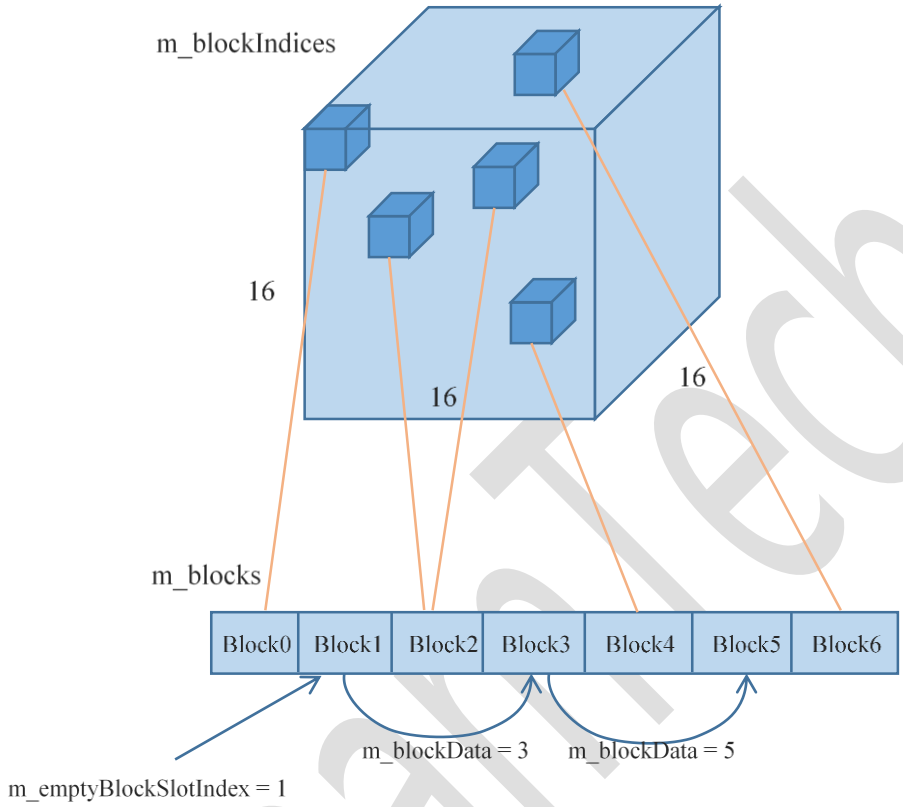

图 7.32 Block 块

方块与 Block 对象的对应关系存放在一个 16 x 16 x 16 的数组 m_blockIndices 中，数组下标的【0-3】【4-7】【8-11】位分别表示 x，y，z 轴坐标，值为该方块对应的 Block 在数组 m_blocks 中的下标，-1 表示该位置不存在方块。类似的，每个方块的光线信息存放在一个 16 x 16 x 16 的 LightData 类型数组中，具有光线属性的方块坐标会放在集合 m_lightBlockIdice 中。

并不是空间中每个存在的方块实体都对应着一个独立的 Block 对象，为了节省空间，相同的方块只对应同一个 Block 对象。这些 Block 对象存在数组 m_blocks 中，当其中某个 Block 对象没有与之对应的方块实体时，系统就会把它重置，但不将其从数组中删除，而是将其放入“空槽栈”，下次需要创建新的 Block 对象时可以重新利用这块空间，这样节省了修改数组和对应表的时间。

7.3.2.3 Block

方块世界最基本的元素，包含一个方块的具体信息，结构比较简单，只有三个成员变量。

- m_pTemplate：保存方块对应的模板（BlockTemplate*），其中包含该方块绝大部分的信息
- m_blockData：int16 类型，方块携带的信息，当方块为空槽（Empty Slot）时，其值表示下一个空槽在 m_blocks 中的下标。

m_nInstanceCount：int16 类型，记录当前方块对象对应的实体方块数目，当它为 0 时，表示其所在的块（BlockChunk）中已经没有对应的方块实体，即为空槽。

7.4.3 方块模型提供形式

方块模板（BlockTemplate）包含了一个 Block 绝大部分的信息，主要是纹理、物理属性和渲染相关的参数信息，它还含有一个方块模型对象（BlockModel）数组，提供方块各个顶点、边界盒子、纹理坐标等更细致的几何信息。

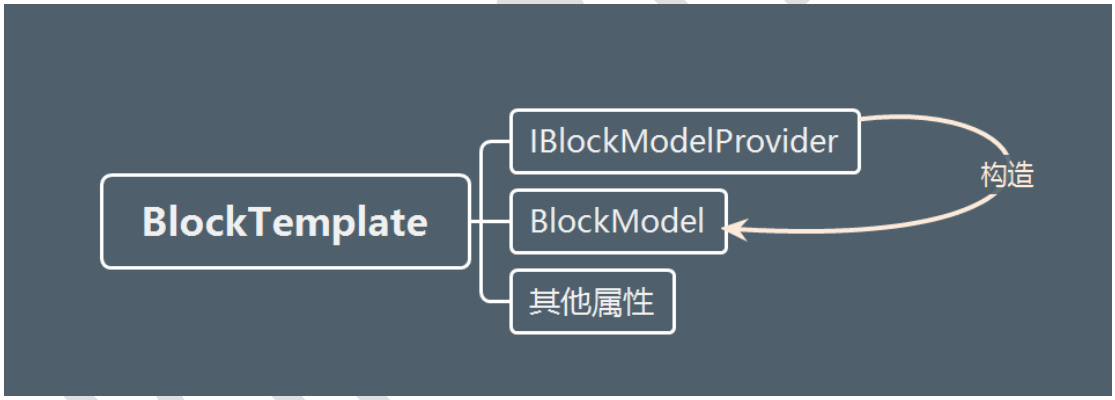

图 7.33 方块模板

对于特定类别的方块模型，我们用 IBlockModelProvider 接口来构造，因此，方块模板还包含一个 IBlockModelProvider 对象，用于生成特定类型的方块模型。方块模板会优先通过这个对象查找方块模型，如果该对象不存在则会使用自带的模型对象。

IBlockModelProvider 有 5 个派生类，它们的继承关系如下图所示：

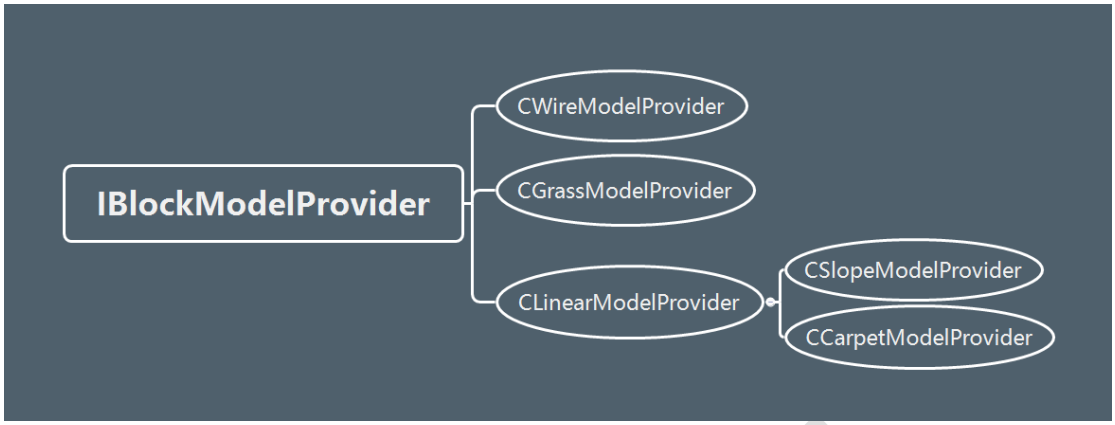

图 7.34　方块模板提供器

7.3.3.1 Wire

WireBlockModel 类用于构造线框模型，实际用于游戏中的藤蔓方块

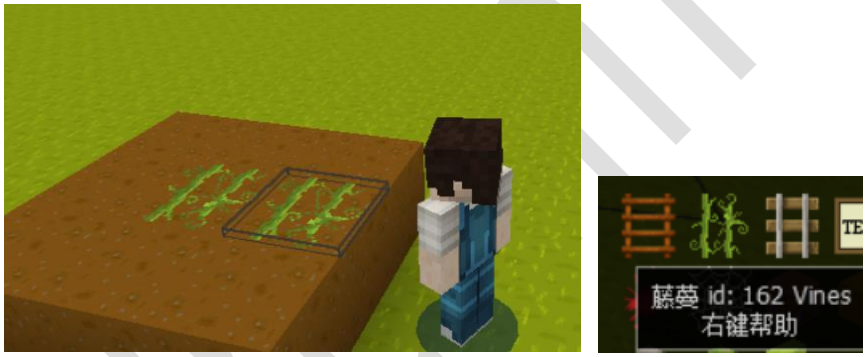

图 7.35　线框模型

7.3.3.2 Grass

CGrassModelProvider 类用于构造草地，它产生的方块会有随机的偏移，从图中可以看出。

图 7.36 草模型

7.3.3.3 Linear

CLinearModelProvider 类用于构造线性模型，绝大多数方块模型都是由这个类提供，比如方块、厚板、交叉平面等。

7.3.3.4 Slope

CSlopeModelProvider 类用于构造斜坡

图 7.37 斜坡模型

7.3.3.5 Carpet

CCarpetModelProvider 类用于构造地毯

图 7.38 地毯模型

7.4.4 ChunkVertexBuilderManager

管理可渲染块（RenderableChunk），含有两个 RenderableChunk 数组，分别用于存放待重构（rebuild）和待上传（upload）的可渲染块。它支持单例模式，提供了处理、上传、增删可渲染块的方法。所有对可渲染块的操作都用一个互斥量保护。

ChunkVertexBuilderManager 与 RenderableChunk 和 BlockRenderTask 的关系如下：

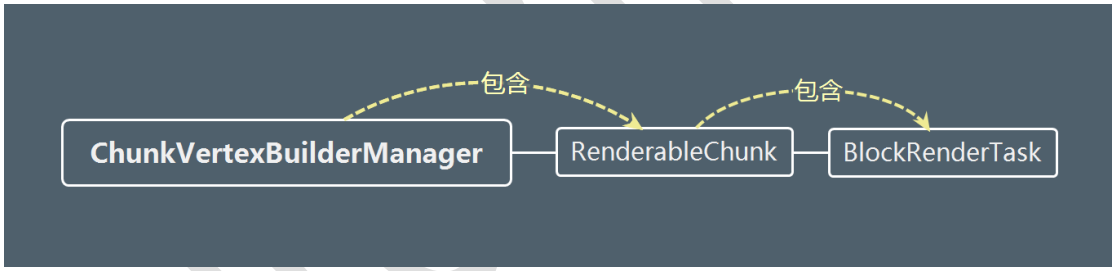

图 7.39 方块渲染任务管理

7.4.4.1 RenderableChunk

可渲染块，保存了该块的所有方块渲染任务，顶点缓冲区，块的位置。它负责渲染缓冲区的更新、重构以及写入内存的操作。

重要属性：

m_renderTasks：BlockRenderTask*的向量，保存块中所有方块渲染任务

m_builder_tasks：BlockRenderTask*的向量，保存构建器任务

m_vertexBuffers：ParaVertexBuffer*的向量，保存块的所有顶点缓存对象

m_memoryBuffers：ParaVertexBuffer*的向量，保存内存缓冲

7.4.4.2 BlockRenderTask

它表示一个方块渲染任务，包含方块模板对象、方块信息、矩形面数、顶点偏移量、该块在方块世界的位置、渲染顺序，和顶点缓存指针。提供了新建、移除渲染任务的方法，和各种属性的 get 和 set 方法。

重要属性：

g_renderTaskPool：静态 BlockRenderTask*向量，作为全局方块渲染任务池，保存所有该类的所有实例。

m_pVertexBuffer：指向顶点缓存的指针变量。

7.5 LOD

7.5.1 介绍

虚拟环境的图形生成的时间耗费大致包括：主机遍历时间，几何处理时间以及绘制时间。几何处理时间由所要处理的图形单元的顶点数目决定，绘制时间由所绘制的面片所覆盖的像素总数决定。虚拟环境的图形生成时间由这三个阶段组成的流水线中时间耗费最长的阶段决定。降低这三个阶段的时间耗费除了利用图形硬件本身外，应当尽量减少几何计算的时间。LOD 技术就是为了加速图形生成而诞生的技术之一。

LOD 全称为 level of detail 多细节层次，指根据物体模型的节点在显示环境中所处的位置和重要度，决定物体渲染的资源分配，降低非重要物体的面数和细节度，从而获得高效率的渲染运算。简单来说就是为每个物体建立多个相似的模型，不同模型对物体细节的描述不同。对物体细节描述越精确，模型越复杂。根据物体在屏幕上所占区域大小及用户视点等因素，为各物体选择或动态生成不同分辨率的三维网格模型，而且同一模型的不同部分也可以使用不同分辨率表示，从而减少需要显示的多边形数目，但图形显示质量不会受到

多大的影响。LOD 技术的基本实现如下：

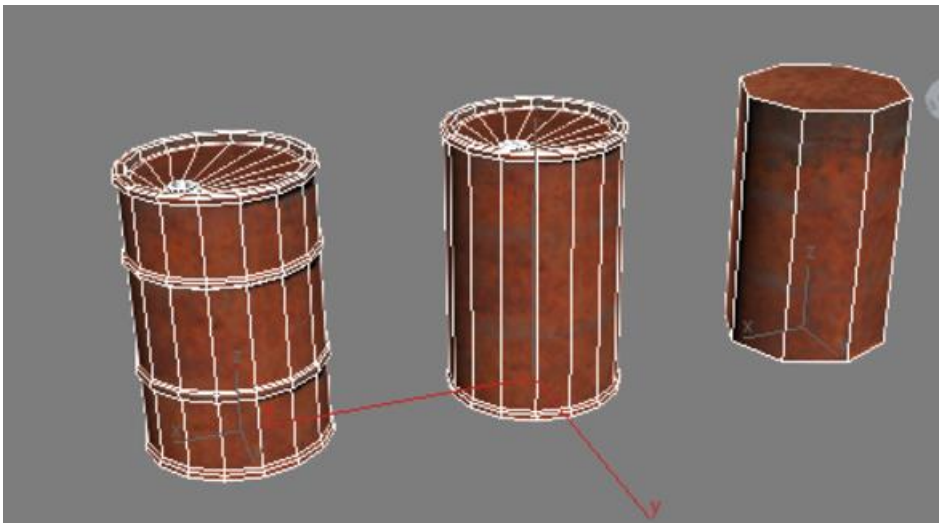

图 7.46　LOD 层级设置

图为面数不同的同种物体模型从左至右依次为高模，中模，低模。为三种模型加上 LOD 编号，并将它们放置在一起

图 7.47　LOD 层级合并

之后设定模型到摄像机或者观察者的距离，在不同距离下生成对应 LOD 编号(LOD0,LOD 1,LOD2.....)的模型就可以了。

LOD 的使用可以在很多方面，例如距离方面：当物体离视点的距离较远时，使用较粗糙的细节层次模型，离得近时使用较细的细节模型。动态方面：当物体处于静止状态时使用细致的细节层次模型，物体处于运动状态时则使用粗糙的细节层次模型。这样总体上物体的三角片总数得到减少，图形渲染速度提升但图像质量几乎不会对用户体验产生影响。

7.5.2 LOD in ParaEngine

在 ParaEngine 中，LOD 的确立主要依赖于两个部分：

1. ParaMeshXMLFile：这个类的作用是在加载 ParaX mesh 或静态网格 Static mesh 文件时起参考作用，包含了 LOD 文件相关的结构和属性定义(LOD 信息，shader 信息)
2. ContentLoader：这个类的核心作用是被用于解析文件并向上层提供数据，为 MeshEntity 和 ParaXEntity 所引用，但是在这个过程中涵盖了 LOD 的创建，分层等功能。

此外，LOD 的创建有时需要依赖 XML 文件，对于 XML 文件的作用，NPLProject.com 给出如下解释：

XML 文件（LOD 网格 XML 文件）是用来把各个.x 模型文件聚合分组。这样当摄像机离观察物体的距离在一个特殊范围内的时候，我们就可以使用定义在 XML 文件里的一个对应的.x 文件。也就是说，LOD 网格 XML 文件就是一种涵括多个.x 文件集合的元文件。当物体在一个特定的半径距离下时它会指定对应情况下该使用哪个文件。

为了生成网格 LOD 文件，要经过以下步骤：

- 把所有的 LOD 网格的模型文件，纹理都放在同一目录下，每个文件命名为 objectName_LOD10.x, objectName_LOD20.x. The number in the trailing

"_LOD[number]"末尾的数字的意思是表示在哪种距离情况（单位米）下这种文件应该被使用

- 在同一个目录下创建一个叫做 objectName.xml 的 XML 文件

例：假设在目录下有 chat.dds, char_LOD5.x, char_LOD10.x, char_LOD30.x. XML 文件就应该命名为 char.xml，然后 LOD 的设置就可以从文件名检索到。

NOTE:所有的 LOD 对于所有动画都必须保证有同样的动画长度。因为游戏引擎会对它所有的 LOD 都使用 LOD0 的动画帧数。

```
<?xml version="1.0" encoding="utf-8"?>
<mesh version="1" type="0">
        <boundingbox minx="-0.336675" miny="0.000000" minz="-0.366781"
maxx="0.312267" maxy="1.247474" maxz="0.335574"/>
        <submesh loddist="5" filename="ElfFemale_LOD5.x"/>
        <submesh loddist="15" filename="ElfFemale_LOD15.x"/>
        <submesh loddist="30" filename="ElfFemale_LOD30.x"/>
</mesh>
```

表 1 The format of xml file

默认情况下，第一个 LOD 包含所有的三角面，第二个 LOD 的三角面不会超过 2000 个，第三个 LOD 三角面不会超过 500 个。对于三角面少于 500 个的网格，不需要任何 LOD。2000 个三角面的网格，2 个 LOD 就够了。三角面多于 4000 的网格，就需要 3 个 LOD 了。但实际上也可以完全由开发者来决定一个模型到底需要多少 LOD。

7.5.3 LOD 的创建

在 ParaEngine，具体的 LOD 创建过程如下所述：

几个重要变量：

m_SubMeshs 表示子网格信息，用于不同的 LOD 信息 **m_sMeshFileName** 网格文件名

m_fToCameraDist 对象到摄像机的距离 **m_Type** 目标文件种类

m_fromDepthSquard LOD 会用到的一种平方值，用于 LOD 列表里的排序

CParaXStaticModelPtr **m_pStaticMesh;** CParaXModelPtr **m_pParaXMesh;**

LOD 创建通过 CreateMeshLODLevel()实现， 流程如图表示，

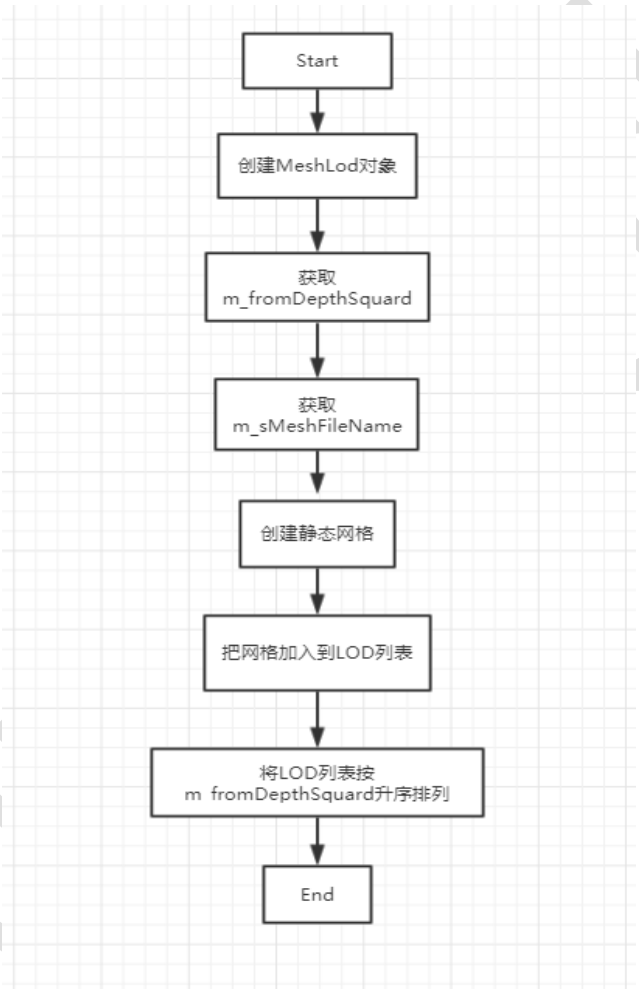

图 7.48 LOD 创建流程图

基本分为 4 步：

1. 获取排序值 depth 的平方 和网格名
2. 创建对应网格名的静态网格
3. 把创建的网格插入 LOD 列表

4. 将 LOD 列表里的元素按 depth 的平方值顺序排列

在 ContentLoader 中，LOD 的配置是文件处理的重要一环，流程为：

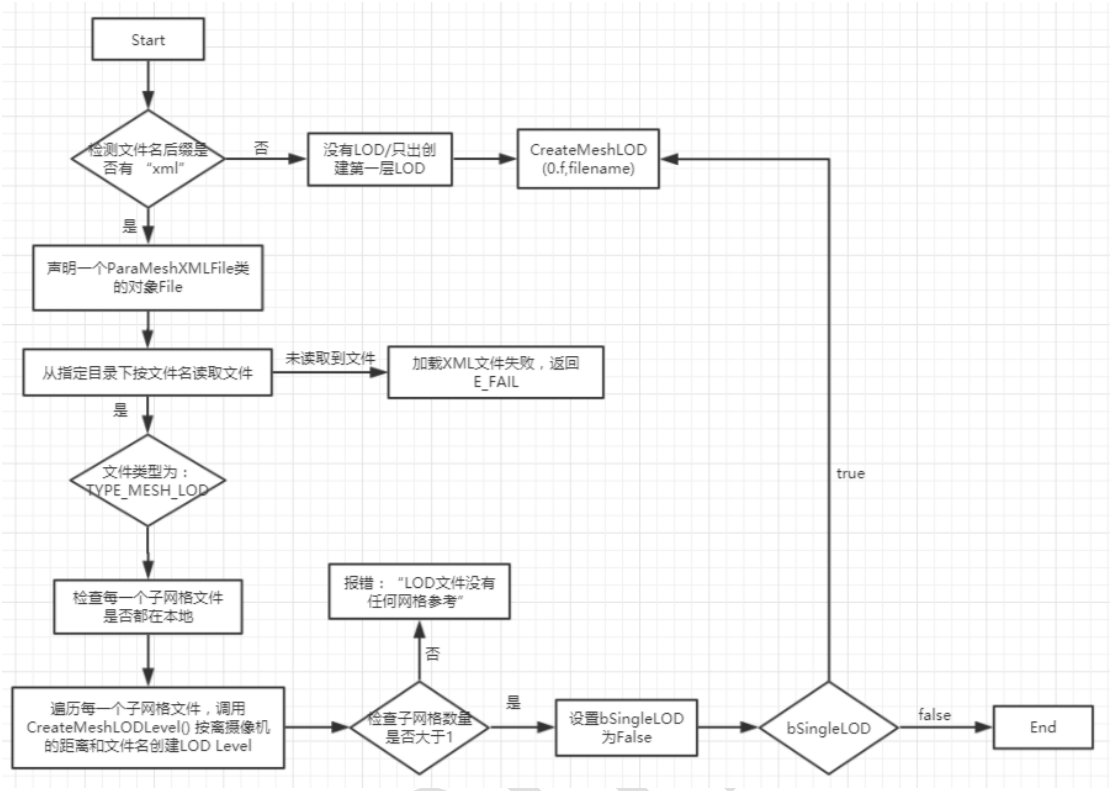

图 7.49 LOD 配置流程图

7.6 动画

动画即计算机 3D 动画，由 3D 模型对象以及动画程序或者说关键帧移动组成。这些模型由有 3D 坐标系下的有几何结构的顶点，面，边构成，所以 3D 模型动画的基本原理是让模型中各顶点位置随时间变化。骨骼/关节动画系统是为了使模型能够进行像人一样的动作的形变而设置的，动画数据可以通过动作捕捉或者动画设计师设计关键帧来实现。主要种类有 Morph(渐变)动画，关节动画和骨骼蒙皮动画，三者均采用关键帧技术，即只给出关键帧的数据，其他帧的数据使用插值得到。就骨骼蒙皮动画来说，其实现流程基本如图：

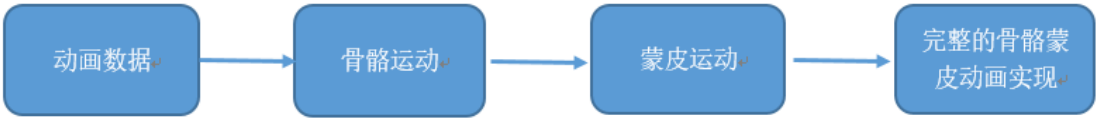

图 7.50　动画实现的基本流程

在这一章中，主要叙述骨骼蒙皮动画，主要分为骨骼和蒙皮两个部分介绍。

7.6.1 骨骼

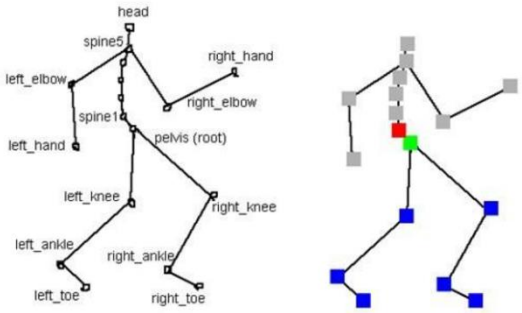

图 7.51　基本的骨骼示例

骨骼的实际理解是一个坐标空间，关节理解为坐标空间的原点，关节位置由它在父骨骼坐标空间中的位置描述。关节既决定了骨骼空间的位置也是骨骼空间的旋转和缩放中心。

骨骼是关节动画的核心，关节动画的模型不是一个整体的 Mesh，而是分成很多部分，通过一个父子层次结构将这些分散的 Mesh 组织在一起如图示。父 Mesh 带动其下子 Mesh 的运动，各 Mesh 定义在自己的坐标系中，所以各 Mesh 是作为一个整体参与运动的。动画帧中设置各子 Mesh 相对于其父 Mesh 的变换(旋转为主，其次是移动和缩放)，通过子到父，一级级的变换累加得到 Mesh 在整个动画模型所在的坐标空间的变换，从而确定每个 Mesh 在世界坐标系中的位置和方向，然后以 Mesh 为单位渲染即可。

关节姿势即骨骼的动作：

局部关节姿势基本可以概括为三类：旋转，平移和放缩，三个动作都是借助于一个仿射变换矩阵实现：

$$P_j=\begin{bmatrix} S_j R_j & 0 \\ T_j & 1 \end{bmatrix}$$

其中 Sj 为 3X3 对角放缩矩阵，Rj 为 3X3 旋转矩阵，Tj 为平移矢量矩阵,j 的范围为 0~N1(N 为骨骼总数)

全局关节姿势就是把关节姿势表示为模型空间或世界空间。某关节的模型空间姿势(j→M)，可通过该关节遍历至根关节，在每个关节乘上其局部姿势算出(j→p(j)).把根关节的父节点定义为模型空间，p(0)=M,关节 J5 的模型空间姿势写为：

$$\boldsymbol{P_{5\to M} = P_{5\to 4}P_{4\to 3}P_{3\to 0}P_{0\to M}}$$

计算流程图如下：

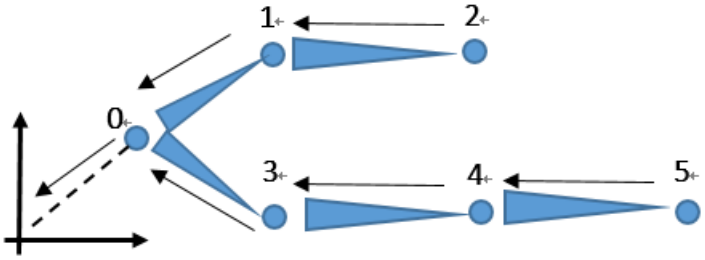

图 7.52 关节矩阵变换

骨骼相关的核心类图如下：

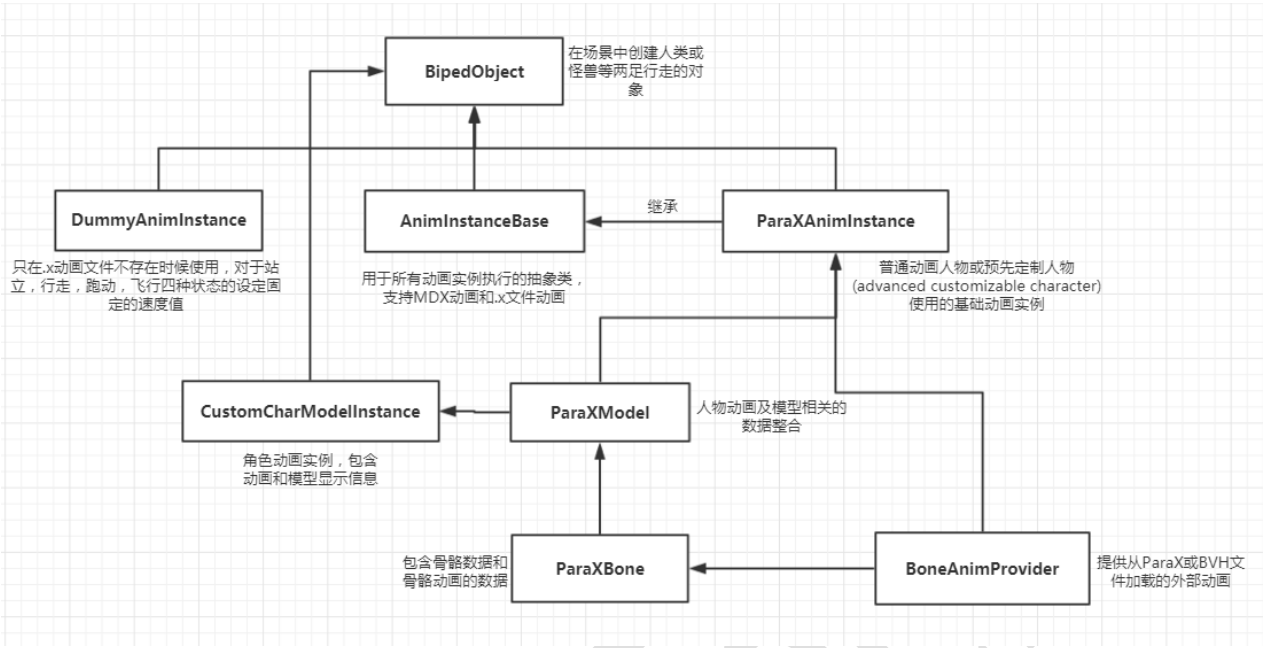

图 7.53 骨骼相关的结构类图

NOTE:

-AnimTable: character animation 人物动画表，定义了各种动画状态及对应序列号，提供动画的映射方法

方法包括依据动画名返回序列号，动画名不存在时返回默认情况序列

-IAnimated: AnimatedVariable 的基类

AnimatedVarible()可以返回插值动画向量帧的帧数量

方法包括：设置获取动画帧数量 / 获取帧索引 / 添加帧 / 获取设置时钟 / 插值算法 (LINEAR 和 Hermite)

-ParaXBone:

相关变量：parent 父骨骼索引 / nBoneID 预定义的骨骼 ID，0 表示未知骨骼，正数表示已知骨骼 / nIndex 骨骼的索引 / mat 最终矩阵 / mrot 最终旋转矩阵 / m_finalTrans. m_finalRot 和 m_finalScalig 是用于组成 mat 的矩阵

相关方法：GetParentIndex() / GetBoneIndex() / GetBoneID() / GetAttachmentId() /

GetPivotRotMatrix() / calcMatrix():该方法是 ParaXBone 的核心方法，包括骨骼搜寻和 ID 获取，骨骼分类和参数获取,骨骼动作变换和相关矩阵计算

-ParaXAnimInstance:

-m_CurrentAnim: current 动画索引，和序号 ID 不同

-m_NextAnim： 下一个动画索引，如果为-1 则动画

-AnimInstanceBase:

-provider：骨骼动画提供者，0 表示本地模型骨骼动画池(local model boane animation pool)，1 表示全局骨骼动画提供者(global bone animation provider)

-looptype:0 表示循环，1 表示不循环，2 表示没确定

-nIndex：current index，-1 表示不可用 -nAnimID：被外部所识别的动画序列 ID

-nCurrentFrame：current frame 当前帧 -nEndFrame：结束帧

方法包括：

动画加载 / 检查模型是否有给定动画 ID / 获取 current 动画 ID / 重设或混合动画 / 设置矩阵 LocalTransform / animated()使模型动起来 / Draw() 开启渲染并推移时钟(advance time) / 获取 Size scale / 设置获取速度 scale / 设置全局时钟 / 使用全局时钟 / 设置获取动画帧数(frame number) / 设置混合因子(动画合成用) / 获取可用动画 ID 等

7.6.2 依附 Attachment

依附就是把物体依附至另一物体之上,可以说是对于骨骼或者对于基本角色模型或动画的扩展。如下图，角色骨骼和角色的大剑武器就是依附关系。

dreamTech

图 7.54 依附示例

最简单的情况，物体对物体的依附涉及物体 A 骨骼关某关节 J_A 的位置或定向，使其与物体 B 骨骼某关节 J_B 重叠。依附通常是斧子关系，父骨骼移动时，子物体应调整以满足约束，其动作情况大致如图：

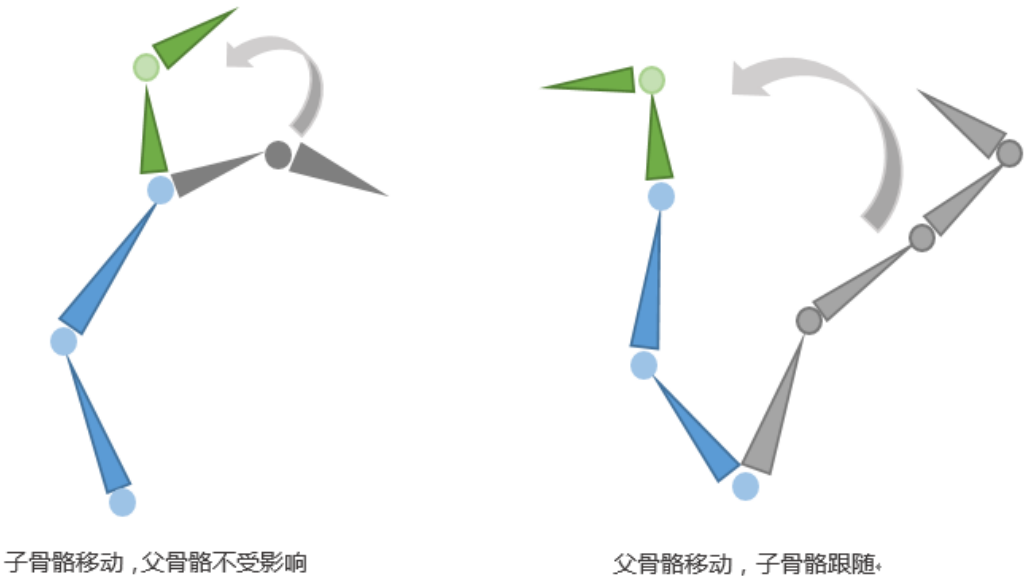

图 7.55 父子骨骼的依附关系

在 ParaEngine 并没有对依附 Attachment 的集中说明的类，依附的实现主要依赖于 addAttachment() / GetAttachment() / HasAttachment()这三个方法，它们基本上在每一个涉及骨骼或动画的类里面都会被使用到。

7.6.3 蒙皮

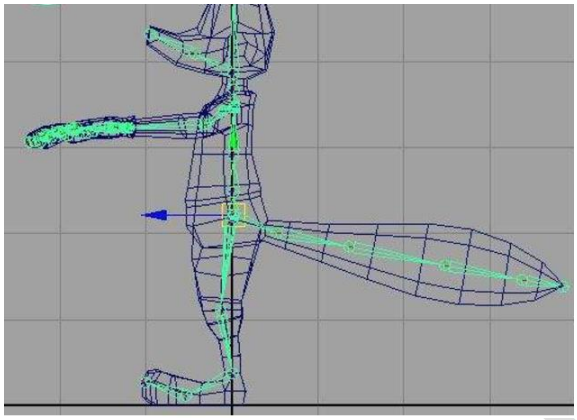

图 7.56 蒙皮示例

蒙皮字面理解就是蒙上一层皮肤，这里的皮肤 skin 其实就是一个 Mesh 网格。蒙皮是指将 Mesh 中的顶点附着在骨骼之上，每个顶点都可被多个骨骼所控制。这层皮肤 Mesh 需要包含蒙皮信息，即 skin 信息。Skin 数据决定顶点如何绑定到骨骼上，它包括顶点受哪些骨骼影响以及骨骼影响该顶点时的权重 Weight，对于每块骨骼还需要骨骼偏移矩阵 BoneOffserMatrix 来将顶点从这层皮肤 Mesh 空间变换到骨骼空间。动画数据控制骨骼的运动，每个关键帧都包含时间和骨骼的运动信息，骨骼再控制蒙皮运动，这样下来整个骨骼蒙皮动画就完成实现了。

蒙皮网格的顶点会追随其绑定的关节移动，数学上就是利用一个蒙皮矩阵，把网格顶点从原来的位置变换至骨骼的当前姿势。变换思路如下图：

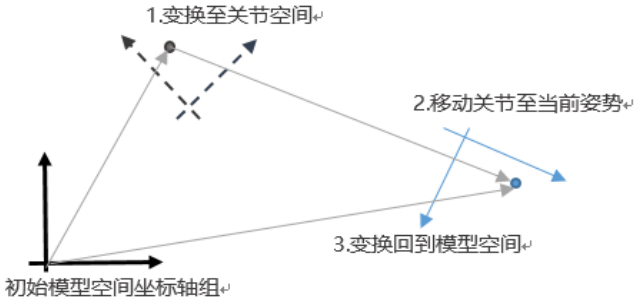

图 7.57 蒙皮矩阵变换思路

只需要两次变动，一次从原模型空间变换至关节空间，一次从关节空间变回至模型空间。蒙

皮矩阵就是这两次变换所用矩阵的乘积。

蒙皮核心类图：

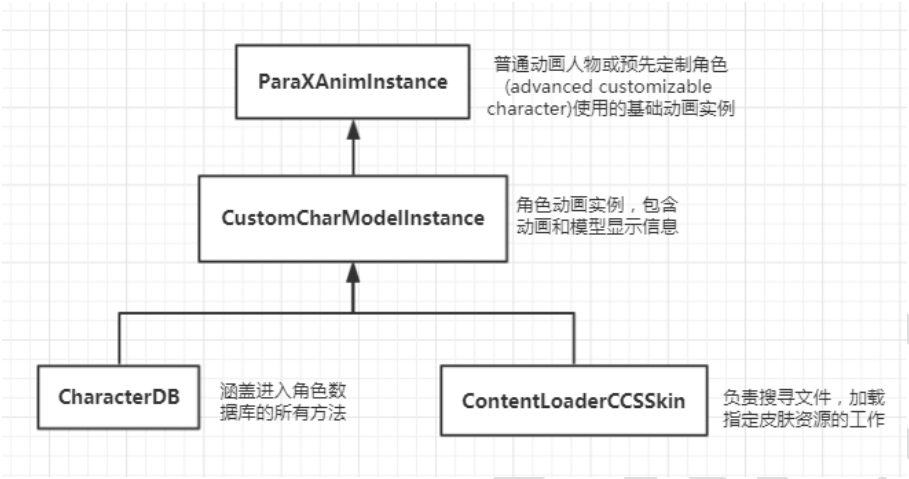

图 7.58 蒙皮相关的结构类图

7.6.4 骨骼蒙皮动画总结

从结构上看：

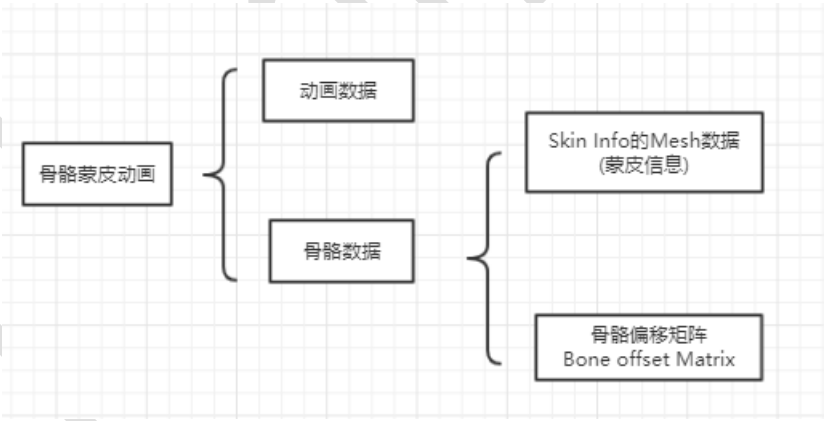

图 7.59 蒙皮动画结构总结

从过程来看：

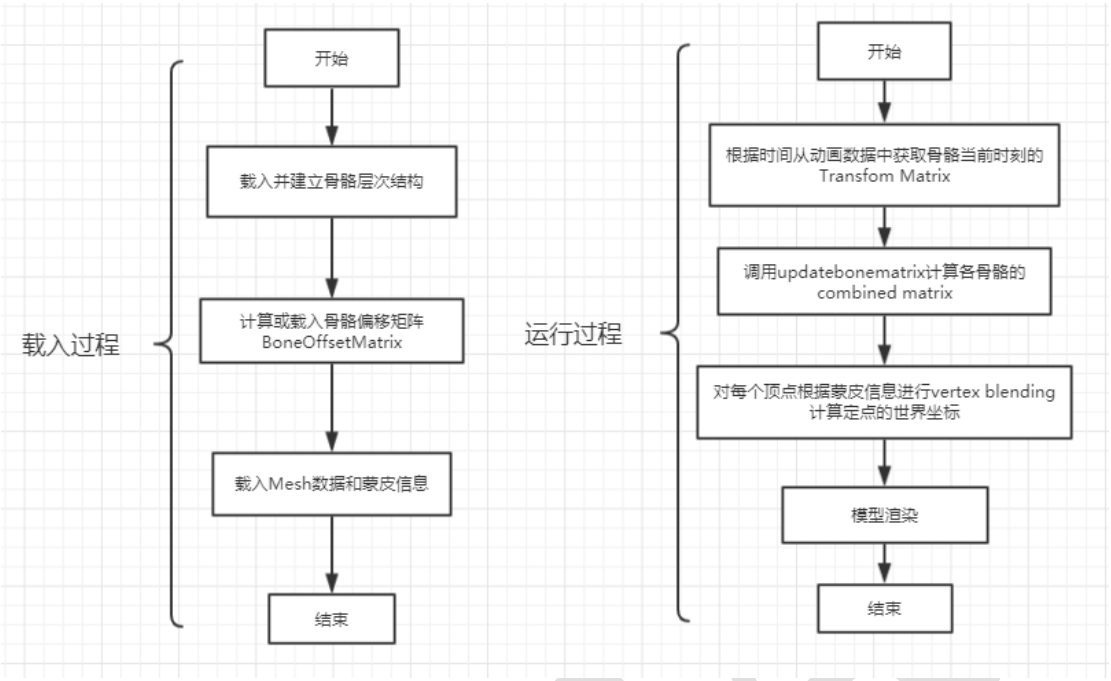

图 7.60　蒙皮动画过程总结

7.6.5 动画管道

底层动画引擎所做的运算，构成了一个把输入(动画片段及混合设置)变换成输出(局部及全局姿势，渲染用的矩阵调色板)的管道。一个动画管道所应具有的阶段如下：

1. 片段解压及姿势提取
2. 姿势混合
3. 全局姿势生成
4. 后期处理
5. 重新计算全局姿势
6. 矩阵调色板生成

ParaEngine 动画管道的数据结构基本也是如此，从资源管理部分里所述依据 ContentLoader 和 Parser 对文件进行解析，到本章前部分介绍的相关动画获取,处理过程，最后进入到 **ParaXModel** 类里依靠 **DrawPass()**方法将之前的数据和相关处理结果生成蒙

皮矩阵调色板呈递给 ParaEngine 的内部渲染器

7.7 事件

7.7.1 概述

事件是游戏过程或操作过程中发生，希望关注的事情。拾取道具，玩家被怪物感知，场景爆炸等都是事件。游戏通常需要一些方法做两件事——当事件发生时通知关注事件的对象，安排对象回应所关注的事件(**事件处理**)。

7.7.2 事件类型与事件参数

事件实际上由两个部分组成：事件类型(受伤，被发现，喝药水等)和事件参数(伤害点数，怪物反击动作，喝药水回复值等)。

在 ParaEngine 中，事件类型主要在 Event_def 和 PE_type 中被声明(Event_def)为主，GUI 相关事件类型被单独声明在 GUIEvent 中，并通过 EventBinding 来获取事件的绑定列表或者说职责链 (binding table)，EventBinding 还有把事件与名称字符串相互转换对应的功能。

```
    enum Event_Mapping{
        EM_NONE,
        //player control
        EM_PL_FORWARD,
        ……
        //game control
        EM_GM_PAUSE,
,       ……
        //camera control
        EM_CAM_LOCK,
        ……
        //GUI control events
        EM_CTRL_SELECT,
        ……
```

dreamTech

```
        //GUI button events
        ……
        EM_BTN_CLICK,
        //GUI scrollbar events
        EM_SB_ACTIONEND,
        ……
        //GUI EditBox Events
        EM_EB_SELECTSTART,
        ……
        //GUI IMEEditBox events
        EM_IME_SELECT,
        ……
        //GUI ListBox events
        EM_LB_ACTIONEND,
        ……
        //GUI Slider events
        EM_KEY_SPACE,
        ……
        //container or window event
        EM_WM_SIZE,
        ……
        };
```

表 7.2 Event_def 代码结构

在 Predefinedevent 中也有一些预先定义的 Events,而保存着事件参数的信息则是在脚本接口中.

此外还有一个继承事件接口 IEvent 的类 EventClass,它主要定义了几种特定的事件结构,,并提供了事件类型参数的获取方法,它会在 ParaEngine 获取用户操作信息时被调用.

```
    /** simple mouse event struct*/
    struct MouseEvent : public IEvent
{WORD m_MouseState;
        int m_x;
        int m_y;
        /** EM_MOUSE_MOVE or EM_MOUSE_CLICK, EM_MOUSE_DOWN, EM_MOUSE_UP,
etc.*/
        int m_nEventType;

    public:
        MouseEvent(DWORD MouseState, int x, int y);
```

DreamTech

```
        MouseEvent(DWORD MouseState, int x, int y, int nEventType);
        /** get event type */
        virtual int GetEventType()  const { return m_nEventType; }
        string ToScriptCode()const;
    };

    /** simple key events struct*/
    struct KeyEvent : public IEvent{…};

    /** system events struct*/
    struct SystemEvent : public IEvent{…};

    /** world editor events, such as scene selection, etc.*/
    struct EditorEvent : public IEvent{…};

    /** touch event */
    struct TouchEvent : public IEvent{…};
```

7.7.3 事件处理与管理

当游戏对象接收到一个事件/消息/命令，它需要以某种方式做出回应，这个过程称为事件处理。对于 ParaEngine,事件处理和事件管理主要依赖于两个类:EventHandler 和 EventCenter。

EventHandler 主要包含了两个方法:

- CEventHandler()用来分析事件种类并设置对应脚本以及

- OnEvent()当事件激活时的反馈方法

EventCenter 是负责事件处理的核心部分，负责对事件进行细化分析并将结果呈递到上层 Globals 以供全局其他接口使用。其涵盖方法大致包括:

添加获取事件处理器 / 登记撤销事件 / 激活事件和待处理事件的分析判定及相关处理 / 事件计数等

dreamTech

整个事件相应管理核心类图如下：

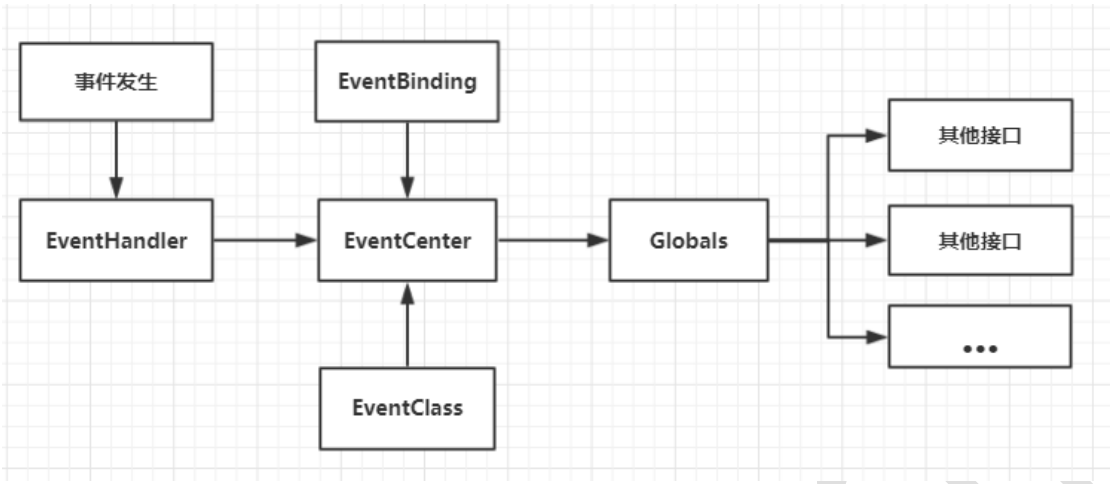

图 7.61　事件相关的结构类图

IdreamTech

8 典型应用

8.1 ParaCraft

Paracraft 创意空间是一款由 NPL 实现，基于 ParaEngine 的软件，其实现了一个 3D 世界，包括方块系统，场景渲染，粒子效果，云雾等等 ParaEngine 的核心效果均在 Paracraft 中展现了出来。作用是用来制作 3D 电影。

通过方块系统来搭建世界：

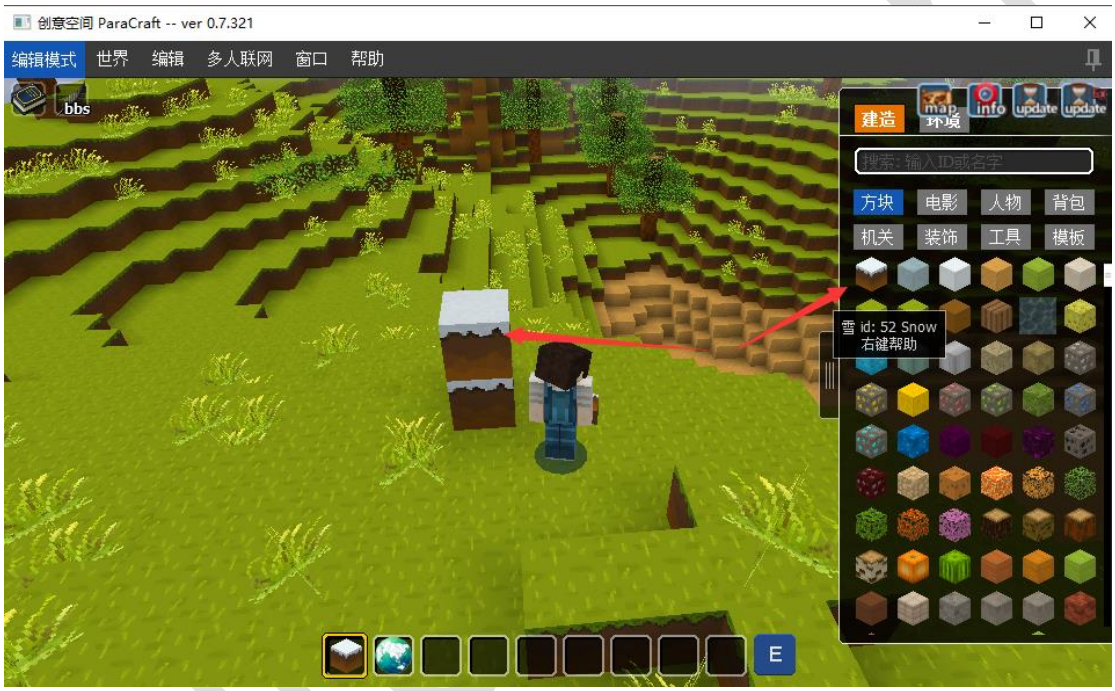

图 8.1 ParaCraft 中方块系统

云雾效果：

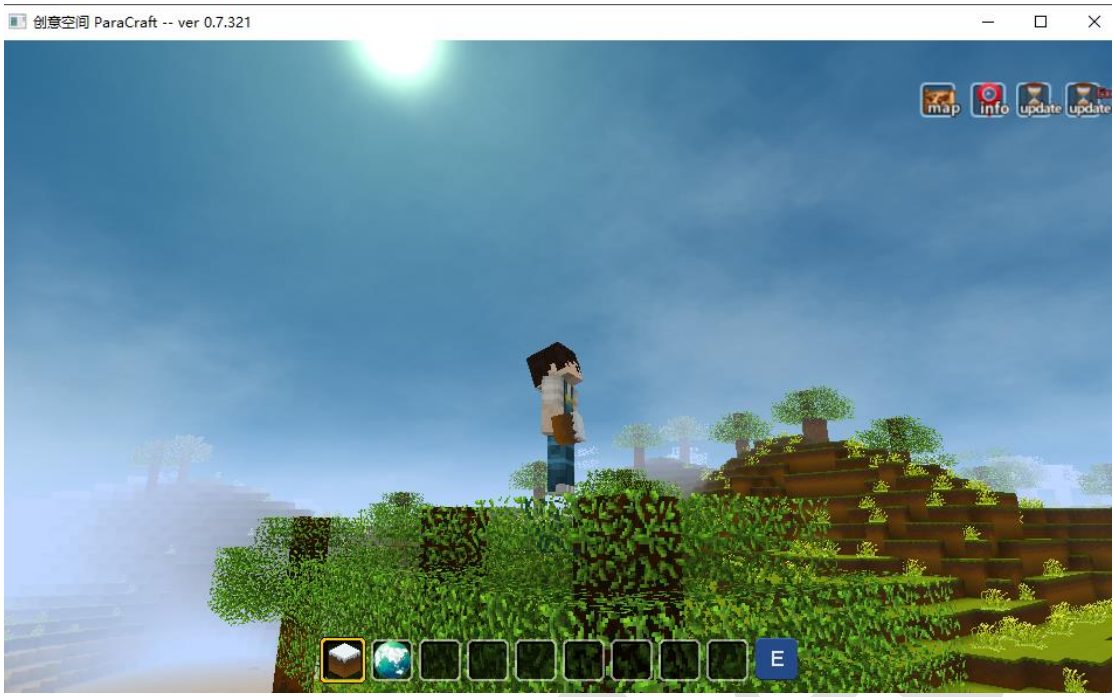

图 8.2 ParaCraft 中的云雾效果

ParaEngine 引擎提供的功能在 Paracraft 中均有体现。

官方网站：http://www.paracraft.cn

8.2 Web Server

NPL Web server 通过 NPL 自己的消息系统实现的，它使用纯 NPL 脚本写成，没有任何外部依赖性。它允许我们通过在后端使用 NPL Runtime，在客户端使用 JavaScript/HTML5 来创建网页。

NPL Web Servers 的优点：

1. 将异步代码转换成同步的代码

许多在服务器端的任务有异步 API，例如数据库操作、远程过程调用、后台任务、定时器等等，异步 API 能将处理线程从 io-bound 或长时间运行的任务中释放。使用异步 API 使服务器非常快，但是很难编写，因为有太多的函数回调会打破其他顺序和相关的编程代码。

在 NPL 页面文件中，任何异步的 API 能够通过 resume/yield 以同步的方式使用。因此，构建 web 页面就像在 PHP 中按顺序编写文档一样。在底层，它仍然是异步 API，同一个工作线程可以每秒处理数千个请求，即使其中一些需要花费很长时间才能完成。

2. 独立的客户端/服务器解决方案

NPL Web server 非常快速且非常容易在各种各样的平台上部署。NPL 提供了丰富的和服务端结构一样的客户端 API，它可以在客户端和服务器代码共享整个运行时环境时提供与专业 web 服务器一样多的请求。

Web Server 源码：

script/apps/WebServer: NPL 中 Web server 实现

script/apps/WebServer/WebServer.lua: 入口文件

script/apps/WebServer/admin: NPL 中的一个类似 php 的 web 站点框架

8.2.1 NPL Admin Site

WebServer/admin 是一个基于 NPL 的开源的网站框架。它所有源代码都可在 NPLCOdeWiki 中查看到。它是一个演示样本，也可调试我们自己的 web 应用。它默认运行在 localhost：8099 上。

运行 NPL Admin Site 方法：

```
npl script/apps/WebServer/WebServer.lua
```

或者可以选择更多的选项：

```
npl bootstrapper="script/apps/WebServer/WebServer.lua"  port="8099"
```

8.2.2 NPL Server Page

NPL server page 是一个混合的 HTML/NPL 文件，通常扩展名为.page。在运行时，网页将会被预处理成纯粹的 NPL 脚本，然后执行。例如：

```
<html><body>
<?npl  for i=1,5 do ?>
   <p>hello</p>
<?npl  end ?>
</body></html>
```

预处理

```
echo ("<html><body>");
for i=1,5 do
 echo("<p>hello</p>")
end
echo ("</body></html>");
```

优点：

当一个 HTTP 请求被发送或重新定向到 NPL 页面处理程序时，会创建一个特殊的沙盒

环境表，所有与该请求相关的页面脚本都在这个新创建的沙盒环境中执行。因此，我们能安全地创建全局变量并且对于每一个页面请求，这些全局变量都是没有被初始化的。

8.2.3 WebSocketServer

NPL websocket server 是一个 tcp 应用程序，它监听着服务器上任何遵循 rfc6455 协议的端口。它是用神经元并行语言实现的，在 web 浏览器和 npl 服务器之间使用 json 来传递信息。

8.2.4 NPL WebSite

NPL WebSite 即网页。关于网页的创建方法可见：https://github.com/LiXizhi/NPLRuntime/wiki/TutorialWebSite

IdreamTech