相关主题 RECOMMEND ARTICLE
- Introduction to 3d game engine design using directx-9 and c#(10)
- Introduction to 3d game engine design using directx-9 and c#(9)
- OGRE 3D 程序设计 (9)
- Introduction to 3d game engine design using directx-9 and c#(8)
- Introduction to 3d game engine design using directx-9 and c#(7)
- Introduction to 3d game engine design using directx-9 and c#(6)
- OGRE 3D 程序设计 (8)
- OGRE 3D 程序设计 (7)

MORE

推荐文章 RECOMMEND ARTICLE
- 策划与程序和美工的沟通
- 龙神传说 原画
- 《LAST online》原画
- 《AION》新原画
- 数据广播方案的优化
- 网络游戏的位置同步
- 游戏音乐制作案例之《战火 红色警戒》音效制作揭秘
- 英雄连Online 原画

MORE

热门文章 HOT ARTICLE
- [电子书下载]游戏设计 — 原理与实践
- [电子书下载]网络游戏开发
- 游戏设计全过程
- [电子书下载]游戏设计技术
- [电子书下载]游戏设计理论
- CS游戏人物模型制作教程
- CG人物插画基本流程
- [转贴]MAX高级人头教程

MORE

您的位置： 游戏引擎

文章标题	OGRE 3D 程序设计 (7)		
来源：	[翻译:邸锐,李旭东]	浏览：	[397]

第七章 资源管理

在Ogre中所有被用来帮助程序进行渲染工作的东西都被称为资源，其中包括模型，骨骼，材质，GPU程序等。而Ogre对这些资源文件和数据载入的组织策略被称为资源管理。

概念总览

在本书的第三章中，已经介绍了一些资源管理系统的API。在这里，让我们做一下简单的回顾。

Ogre可以处理下面将要列出来的所有资源类型。在实际应用的时候，它们可以从磁盘文件中载入，也可以通过其他的办法得到（比如在程序中手动生成）。在这里为了简单其间，我们在这里假设从文件中直接载入值。

·材质资源：在.material文件中包含的材质脚本定义（技术、通路、纹理单元等数据的定义）。

·模型资源：经过优化的二进制网格模型文件，扩展名为.mesh。包含几何信息和一些动画数据。

·骨骼资源：经过优化的二进制骨骼文件，扩展名为.skeleton。包含骨骼动的数据以及相应帧动画的信息。

·字体资源：字体的配置信息，扩展名为.fontdef的文件，其中包括TrueType字体的引用以及其他字体配置数据。

·GPU程序资源：在.program中对GPU程序的声明信息，与材质脚本.material文件有类似的结构，但Ogre保证所有.program文件都会在处理材质脚本之前被载入处理。

·纹理资源：在纹理中使用的2D图片数据。每种类型都有其自身的扩展名，比如JPG文件的.jpg或者.jpeg，Targa文件的.tga，诸如此类。

资源管理

虽然理论上说可以直接从文件中载入资源，但是在实际的执行过程中有很多原因让我们更倾向于是用资源管理策略来处理资源。其中有一些重要的原因分别是，首先：磁盘操作的速度远远慢于内存操作，你基本上不可能在每一帧的时间内都处理磁盘数据。另外一个很重要的原因是很多文件可能需要解压过程，对于其中一些需要被重复使用的文件，使用资源管理可以重复从内存中得到解压过的资源数据。最后的原因，系统可以在空闲时处理对资源的载入，而不用到使用的时候才载入。

资源组

Ogre可以从两个不同的层级来对资源进行管理：对某种类型的资源管理和对一组资源类型的管理。通过资源组进行管理的优点是可以简化对资源整体的操作，当你把资源分成不同的组织后就可以通过组名对整组资源进行载入、卸载以及初始化操作等，这比单独处理每种类型方便很多（这种操作适合同一时间进行很多不同资源处理的时候，比如游戏载入和初始化时）。

是否通过资源组的概念对资源管理完全取决于使用者的决定。在不同的需求情况中可能会有不同的优缺点。资源组的查找算法的复杂度是常数级即O(1)，所以你可以放心的加入任何数量的资源组，这不会对效率有任何影响。如果你没有指定所使用的资源组，那么Ogre会帮助你自动的放入“General”组中，当你不使用资源组进行管理的时候，并不用理会这些具体细节。

资源组与世界地图

在默认的情况下，Ogre通过场景管理器把整个地图载入“General”资源组中。因此你也可以使用与资源管理一样的方法去管理场景/世界地图资源数据。假如需要在载入地图之前载入所有场景对象，你就可以通过资源管理的回调方法来实现这种功能。回调的监听对象是“ResourceGroupListener”类型的子类，你可以在这个章节稍后一点的代码7-5中看到使用的例子。

资源定位（档案）

“资源定位”提供了这样一个储存空间，用户可以通过索引快速搜索到所需资源。你可以把它想象为网络搜索引擎：支持通过关键词映射到资源本身的快速查找。

你并不需要在程序执行前预先定义所有的“资源定位”，你可以随时动态添加或者移除它们。例如可能定义了一组被称为“World”的资源定位用于储存当前关卡的所有信息和资源，你可以在玩家通过了这个关卡之后就卸载它们。这种管理机制对于游戏工具的开发者来说更加重要，因为程序无法在启动时候知道用户需要编辑文件的具体位置，只能靠用户在运行之后提供信息来动态载入。

在Ogre中，这些“资源定位”被称为档案（Archive），也可以把它看作为“一个文件收藏夹”。在使用前需要提供给系统一个文件结构的起始根节点，用于一个新的“资源定位”加入到系统中用来进行搜索，看起来似乎和硬盘上的目录系统很相似。没错，硬盘上的目录系统就是档案的一种具体实现（换句话说，你可以直接把某个目录作为“资源定位”来进行管理）。而另外一种常用的档案的具体实现是ZIP压缩文档。你可以直接把ZIP文件加入到Ogre的“资源定位”中，通过档案Archive的接口来管理它。

你也可以根据自己的需要实现其他形式的档案（比如直接处理网络数据的档案），你只要实现Archive接口就可以很好的工作。这组接口提供的功能包括：

· 需要支持对节点下面资源的枚举（目录列表，类似DOS命令中的“dir”），并且实现通配符的使用（文件名的模式匹配功能，比如“*.gif”）。

· 支持节点递归的结构（子节点也能有自己的子节点，子目录也能有自己的子目录）。

· 能够支持Ogre对文件通过流的方式进行读取。

在Ogre的档案中，对其中资源的管理是以只读的方式提供。虽然Ogre的Archive接口不支持写入数据，不代表不能够实现写入功能。只不过是Ogre并不需要写入功能，所以就没有这些接口了。

Ogre的资源管理器通过档案的枚举文件功能来索引到所需的资源，但是索引之后并不会立即载入这些资源，具体的载入工作将会放在之后执行。

资源生存周期

在图7-1中我们展示了资源各种状态和触发他们转换的“事件”。这些事件可以通过Ogre的API调用到用来触发，我们可以在图中找到这些接口。在通常你没有定义自己独有的资源管理的情况下，并不需要理会这些状态之间的关系。Ogre会通过自己的“帽子戏法”把这些转换神奇的进行良好的管理。

资源管理逻辑

从现在开始，我们对资源管理的关注点从物体的储存转移到资源组织的逻辑层面上来。我们首先要知道，最基本的组织方式是靠不同的资源组来划分。不同名称的资源组中可以储存任意数量任意类型的资源（关于资源类型的相关信息，以在本章前面的“概念总览”中提到）。

每一种资源类型都会有自己独有的管理器类型。它们主要负责载入和卸载所属管理类型的资源。不过Ogre的管理器中除了最基本的工作，并没有实现什么特殊的算法。如果你希望使用一些比较Cool的方式来管理你的资源，比如通过LRU（最近最少使用算法）[1]来作为调配内存的策略，你就需要自己来在管理器中实现具体的代码。不过这里需要提醒你，在显示驱动程序中，已经为大部分重要的资源实现了相应的LRU算法管理了。

资源载入

如果你所使用的资源在之前没有被载入，那么Ogre会自动帮助你在内部执行相应的载入动作。这里有一个常用的例子，当你直接使用一个实体的时候，Ogre会帮助你把它所使用的所有其他资源（可能包括模型以及模型所引用的骨骼、纹理等等）载入进内存。

具体的载入和卸载的实现交给不同类型的资源管理器内部，这是因为即使在同一类型 “档案”中处理，不同类型的资源也可能需要不同的处理策略。

对手动创建的资源载入

与那些通过系统API读取磁盘或者“档案”里面的资源不同，你也可以载入手动创建的资源，Ogre支持“手动资源载入器（manual resource loader）”的概念。因为这些资源的特殊性，在资源管理器的层级上无法实现具体的载入操作（你可能对不同资源有不同构造处理）。解决这个问题的办法是，你需要允许在当前资源层级中定义具体的载入，取得以及装配方法。

注意：这是一种比较常用的处理资源的方法，对任何类型都是可用的。举例来说，你可以根据需要对一个网格模型执行手动或者自动的载入过程。

在通常的情况下，你不用担心那些能持久保持数据的储存介质上的资源消失（比如在光盘或者硬盘上的数据）。所以你尽可以在任何时候再次载入已经卸载的资源数据。

而对于手动资源载入的过程来说，你就不能这么高枕无忧了。因为可能在你下次希望重新载入资源的时候，“手动资源载入器”已经在内存中被释放掉了。所以这时候你要警惕的处理这个问题，在任何需要“载入器”的时候重新构建它。这是手动资源载入唯一需要特别提点的地方，其他的使用和自动的载入方法没有什么大的出入。

后台处理资源载入

不得不提出的是，在默认的情况下Ogre并不是线程安全的。

注意：首先，现代GPU已经在内部实现了高效的并行处理（大多数时候都可以信赖），然而把几何体和材质捆绑而执行渲染的驱动过程是单线程的；并且多线程的共享对象的互斥等问题也会影响整体的渲染效率。因为上面所提到的原因，Ogre并没有对多线程进行支持。不过虽然图形引擎本身并没实强壮的线程安全，但不意味着你的应用程序不能采用为多线程的设计方法（大多数3D应用程序需要并发），你可以把Ogre看作你程序中诸多线程中的某一条来使用（换句话说Ogre本身还是在单独线程中工作）。

不过你可以通过一个编译期的宏定义来启用一个简单的线程保护功能，但要注意这只是简单的针对资源管理系统的后台载入方法而使用的，对你其他的应用并不一定足够。

在实际的应用中，没法在每帧运行中从磁盘载入资源。因为硬盘操作的微小延迟将导致帧率明显的“停顿”。你可能在玩游戏的时候经历过：当磁盘灯开始闪烁的时候，你的游戏就死在某一个画面上。解决这个问题的方法是通过另外的一个线程处理资源载入，它独立于主循环：换句时髦的话说“后台载入”。

后台载入的多线程的资源管理方案

在使用多线程管理后台载入的时候，你必须定义一个这样的宏。

```
#define OGRE_THREAD_SUPORT 1
```

把它写在OgreConfig.h文件中。之后你就可以放心的把资源管理器和Root分别放入不同的线程来处理。这时候Ogre就天然的实现了后台载入的功能。

后台载入的非多线程的资源管理方案

如果你不喜欢多线程的处理方式，Ogre也提供了另外的单线程版本给你。在本书前面提到了Ogre大量的使用了观察者（Observer）模式来监听各种操作。当然资源管理器也包含其中。

包含多个方法的回调接口ResourceGroupListener允许你在一个很细微的层次监听资源载入过程，并且执行你需要的操作。例如，你可能需要每帧只执行十个资源载入过程中的一个，下一个帧执行下一个步，等等。直到资源被载入和初始化，然后处理下一个资源。这样你可以实现一个轻量级的“协同多任务”，简单的资源后台载入而不需要处理多线程。

资源卸载

资源数据载入后一直驻留在内存中直到到应用程序强制卸载（通过资源组管理器或者直接释放资源）。也可以通过资源组管理器在组的层级上卸载资源，意思是说所有同一组中的资源会被组管理器一同释放。Ogre绝不会自动从内存中释放资源，并且你不能够强制卸载还被资源组引用的资源。

3D应用程序的智能资源管理是一个正在讨论的主题，所以我不能说一个系统好于另外一个。只需要提醒你Ogre不能为你管理你的资源。任何未来的智能资源管理（例如，即时的载入/卸载方式）都必须被开发者支持。幸运的，Ogre允许高度的控制资源的生存周期，象你在这章看到的一样，你可以很容易支持智能的资源管理。

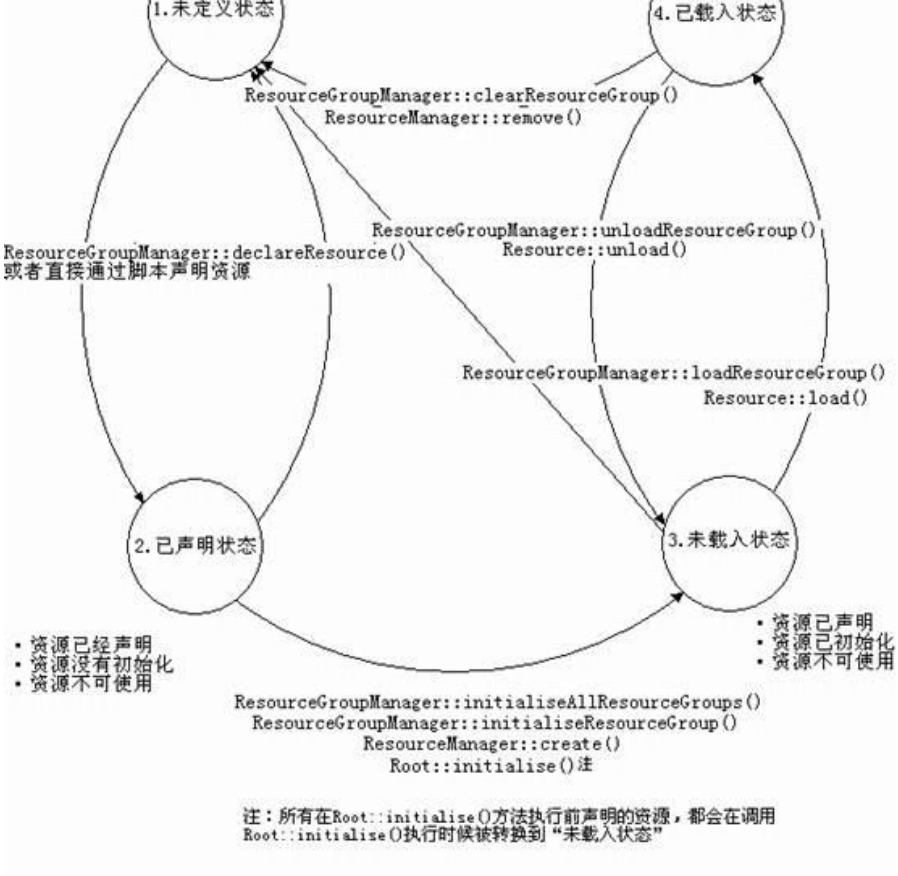

图7-1：资源生存周期示意图

当你决定实现自己的资源内存管理器的话，你就有必要了解上面图中所展示的资源生命期的各种状态，以及它们之间是如何转换的。

·未定义状态：这是在应用程序启动的时候所有资源的默认状态；在资源被定义之前，Ogre是不知道它的，你可以手动使用ResourceGroupManager::declareResource()或者在解析脚本（例如材质脚本）的时候定义资源。在这个状态下，你的资源不使用任何内存。

•已声明状态：当你通知Ogre你要把资源作为要载入内容的一部分（例如，调用ResourceGroupManager::loadResourceGroup()）的时候你就需要向系统声明这个资源。在任何时候都可以执行对资源的声明工作（例如通过ResourceGroupManager::declareResource()方法），甚至可以在你的渲染系统初始化之前就把你需要的资源声明出来。这与资源管理器的创建工作不同，对ResourceManager::create()方法的调用一定要在渲染系统的初始化之后才能执行。

·未载入状态：当资源初始化之后就进入了这个状态。资源的初始化可以通过调用ResourceGroupManager::initialiseAllResourceGroups()方法或者ResourceGroupManager:: initialiseResourceGroup()方法来完成，而当你执行Root::initialize()的时候也会附加的帮你把之前所有声明的资源初都进行始化工作。在进入未载入状态之后，资源会使用一小部分的内存，用于存放定义的实例，但是实际资源数据并没有被载入内存。载入状态的资源也可以通过调用ResourceManager::unload()，ResourceManager::unloadAll()，ResourceManager::unloadAllUnreferencedResources()，ResourceGroupManager::unloadResourceGroup()或者Resource:unload()来释放数据从而转入未载入状态。这些工作虽然不会删除资源的引用，但是真正的数据会被卸载出内存。

•已载入状态：这是你的资源“全部激活”时候的状态：资源的数据（例如，贴图资源中的实际图片数据）在内存中存在，并且可以在你的应用程序中直接使用。当你的应用程序使用它们的时候（例如，使用SceneManager::createEntity()添加一个实体到场景中），如果它们没有被载入（并且资源管理器的内存配额没有超出）Ogre会自动帮你载入它们。当然你也可以手动的调用Resource::load()，Resource::reload()，ResourceManager::load()，ResourceManager::reload()，ResourceManager::reloadAll()，ResourceManager:: reloadAllUnreferencedResources()或者ResourceGroupManager::loadResourceGroup()等方法来强制载入。

资源管理实践

代码7-1提供了Ogre演示程序中用来初始化资源的resource.cfg文件，你可以在Ogre的SDK中找到相应的文件。

代码7-1：resource.cfg文件中对资源管理器的配置信息

```
# Resource locations to be added to the 'boostrap' path

# This also contains the minimum you need to use the Ogre example framework

[Bootstrap]

Zip=../../Media/packs/OgreCore.zip
```

```
# Resource locations to be added to the default path

[General]

FileSystem=../../Media

FileSystem=../../Media/fonts

FileSystem=../../Media/materials/programs

FileSystem=../../Media/materials/scripts

FileSystem=../../Media/materials/textures

FileSystem=../../Media/models

FileSystem=../../Media/overlays

FileSystem=../../Media/particle


FileSystem=../../Media/datafiles/fonts

FileSystem=../../Media/datafiles/imagesets

FileSystem=../../Media/datafiles/layouts

FileSystem=../../Media/datafiles/looknfeel

FileSystem=../../Media/datafiles/schemes

FileSystem=../../Media/datafiles/lua_scripts


FileSystem=../../Media/DeferredShadingMedia

Zip=../../Media/packs/cubemap.zip

Zip=../../Media/packs/cubemapsJS.zip

Zip=../../Media/packs/dragon.zip

Zip=../../Media/packs/fresneldemo.zip

Zip=../../Media/packs/ogretestmap.zip

Zip=../../Media/packs/skybox.zip
```

代码7-1相当的清晰明了。文件中出现的两个资源组分别是“Bootstrap”与“General”。其中“General”是系统默认的组并且始终存在，当你把资源添加到这个组，并不会创建新的资源组。而例子中的Bootstrap组意味着只针对Ogre演示框架ExampleApplication的资源对象；它是一个可以随意起的名字（你也可在你的实现中任意使用），它并不会像General一样要被特殊处理。

FileSystem和Zip指示了所使用“档案”的类型：其中FileSystem表示一个文件系统为基础的资源“档案”；而Zip表示了资源路径是一个ZIP压缩文件的资源“档案”。如果你实现了自定义的“档案”类型，可以在这里调用。

资源定位

一个经常被提出的问题是，“如果没有resource.cfg我该如何设置资源定位？”让我们先撇开这个话题来看看ExampleApplication.h中的代码。代码7-2中，展示了如何读取resources.cfg文件。

代码7-2：ExampleApplication框架中用于解析resources.cfg文件的代码

```
// Load resource paths from config file

ConfigFile cf;

cf.load("resources.cfg");


// Go through all sections & settings in the file

ConfigFile::SectionIterator seci = cf.getSectionIterator();


String secName, typeName, archName;

while (seci.hasMoreElements())

{
```

```
secName = seci.peekNextKey();

ConfigFile::SettingsMultiMap *settings = seci.getNext();

ConfigFile::SettingsMultiMap::iterator i;

for (i = settings->begin(); i != settings->end(); ++i)

{

typeName = i->first;

archName = i->second;

ResourceGroupManager::getSingleton().addResourceLocation(

archName, typeName, secName);

}

}
```

注意：resources.cfg只是为了方便演示程序提供的文件；在Ogre的API中你找不到任何处理resources.cfg文件的方法。ConfigFile类（和它内部的类）是Ogre的一部分；它用来读取和解析脚本使用的格式。你可以借助这个类来处理你自己的配置文件，可以如同处理Ogre自身定义脚本一样简单。

上面这段代码（代码7-2）被Ogre所有的演示程序调用，以用来解析读取资源配置文件resources.cfg。它在外层迭代文件中的每一个片段（每一个片段也是一个组名，Bootstrap和General），然后对每个片断再历遍其中每一个名字/值对。最后对每个名字/值对调用ResourceGroupManager的addResourceLocation()方法来设置资源定位。

代码7-1中脚本所依赖的核心其实只有addResourceLocation()。下面列出通过这个方法实现的与上面配置文件代码相同效果的的硬编码示例（代码7-3）。

代码7-3：直接的，硬编码方式实现与代码7-2一样的效果

```
ResourceGroupManager *rgm = ResourceGroupManager:;getSingletonPtr();

rgm->addResourceLocation(“../../media/packs/OgreCore.zip”,”Zip”,”Bootstrap”);

rgm->addResourceLocation(“../../media/”,”FileSystem”,”General”);

rgm->addResourceLocation(“../../media/fonts”,”FileSystem”,”General”);

// 你可以在后面再添加其他需要的资源定位
```

在这里可以留意一下上面代码中对资源定位的路径设置都是系统相对路径。这并不代表Ogre的资源定位无法支持绝对路径，但是如果你决定使用绝对路径，当程序安装到其他用户的机器上时候，就可能会失效。所以配制各种目录时候尽量使用相对路径是一种很好的习惯。

资源初始化

资源初始化需要一定的顺序支持。在正式添加资源前，你需要首先初始化相应的资源组。而在初始化资源本身之前，你需要创建所需的渲染窗口，这是因为对资源脚本的解析可能会依赖GPU的配置。

警告：请认真阅读上面的一段文字，因为再Ogre论坛中最常出现的问题都是因为上面所提的原因引起的。

代码：7-4 初始化资源定位

```
// 初始化所有先前定义的资源组

ResourceGroupManager::getSingleton().initialiseAllResourceGroup();


// 或者每次交替初始化一个资源组

ResourceGroupManager::getSingleton().intialiseResourceGroup(“General”);
```

```
ResourceGroupManager::getSingleton().intialiseResourceGroup(“Bootstrap”);
```

在代码7-4中其实包含了两种不同的初始化方式，第一种是提供给懒人用来一次初始化所有未初始化资源组的方法。下面的是提供给希望更精确控制初始化过程的人使用的每次初始化一个资源组的方法。

资源卸载

你可以在任何希望的时候从内存中卸载资源（不论是单独的或者一组）。Ogre会负责帮助你维护代码的安全性，当你在其他地方再次使用资源，Ogre会帮助你重新载入。如果当你卸载的时候仍然有系统在使用资源，Ogre会帮助你禁止这个卸载动作（至少是对3D渲染相关的资源）。

下面的代码展示了资源组管理器ResourceGroupManager的全部卸载资源的接口：

```
ResourceGroupManager::getSingleton().unloadResourceGroup(“Bootstrap”, true);

ResourceGroupManager::getSingleton().unloadUnreferencedResourcesInGroup(“Bootstrap”,true);
```

第一行代码提供了卸载一整个资源组的方法，而第二行的方法只会卸载当前没有引用的资源。当资源被卸载之后，其数据被移出内存，但相应的实例和声明信息都还存在（当再次使用时候系统会负责重新载入工作）。

在方法中的true参数告知系统卸载我们允许卸载的资源；换句话说这种调用方法不会卸载我们标记为“不可卸载”的资源。在默认的情况下，方法会使用默认参数false来卸载全部资源。

清理或者销毁资源组

在字面上看来清理和销毁资源组差别并不大，但是在执行中要特别注意这两种操作的细微差别。

清理资源组只是负责卸载和删除资源组里面的资源：

```
ResourceGroupManager::getSingleton().clearResourceGroup(“Bootstrap”);
```

而销毁资源组不单单“清理”了里面的资源，还“销毁”资源组本身：

```
ResourceGroupManager::getSingleton().destroyResourceGroup(“Bootstrap”);
```

卸载单独的资源

如果你希望仔细的管理每一个资源的生存周期，你可以通过调用资源本身的unload方法执行卸载。

```
// 假设pEntity是一个Entity实例的指针

MeshPtr meshPtr = pEntity->getMesh();


meshPtr->unload();
```

载入和重新载入资源组

这里有一个可以载入同一资源组所有资源的方法：

```
ResourceGroupManager::getSingleton().loadResourceGroup(“Bootstrap”);
```

你也可以通过额外的参数来定义载入那部分资源，其中包括“世界地图”和其他“普通”的资源。参看下面两个方法的调用：

```
//例如只载入或者重新载入世界地图，将使用下面的方法调用
```

```
ResourceGroupManager::getSingleton().loadResourceGroup(“Bootstrap”, false, true);


//只载入或者重新载入“普通”组中的资源：

ResourceGroupManager::getSingleton().loadResourceGroup(“Bootstrap”, true, false);
```

资源组载入的回调事件

如果在资源管理其中注册了相应的监听对象，Ogre会载入资源时候通知你各种载入时间。这种功能最通常的应用是UI界面的载入条等很有趣的功能。

代码7-5：展示了一个通过实现资源监听接口ResourceGroupListene来实现的UI窗口空近代码（这里虚构了一个被称为ProgressMeter的类型）

```
// 注意：例子中使用的ProgressMeter是个虚构的类型


class LoadingProgresslistener : public ResourceGroupListener

{

public:

LoadingProgressListener(ProgressMeter& meter) :

m_progressMeter(meter){ m_currentResource = 0;}


// 资源组ResourceGroupListener事件方法


// 当资源组开始解析脚本的时候被调用

void resourceGroupScriptingStarted(const String& groupName,

size”_tscriptCount){}


// 当一个脚本开始被解析的时候调用

void scriptParseStarted(const String& scriptName){}


// 当一个脚本已经被解析完的时候调用

void scriptParseEnded(){}


// 当资源组中脚本被解析完毕的时候被调用

void resourceGroupScriptingEnded(const String& groupName){}


// 当开始载入资源组中资源的时候被调用

void resourceGroupLoadStarted(const String& groupName,size_t resourceCount)

{ m_resCount = resourceCount;}


// 当开始载入一个资源的时候被调用

void resourceLoadStarted(const ResourcePtr& resource){}


// 当一个资源载入结束的时候被调用

void resourceLoadEnded();


// 当世界几何体开始载入的时候被调用

void worldGeometryStageStarted(const String& description){}
```

```
// 当世界地图载入结束的时候被调用

void worldGeometryStageEnded();


// 当资源组的资源全部载入结束的时候被调用

void resourceGroupLoadEnded(const String& groupName){}

private:

int m_resCount;

int m_currentResource;

ProgressMeter &m_progressMeter;

};


void LoadingProgressListener:worldGeometryStageEnded()

{

// 增加当前资源数量

m_currentResource++;


// 根据百分比更新进度条尺寸

m_progressMeter.updateProgress((float)m_currentResource /(float)m_resCount);

}


void LoadingProgressListener::resourceLoadEnded()

{

// 增加当前资源数量

m_currentResource++;


// 根据百分比更新进度条尺寸

m_progressMeter.updateProgress((float)m_currentResource /(float)m_resCount);

}


// 使用ProgressMeter类的实例初始化一个listener

LoadingProgressListener listener(m_progressMeter);


// 在你代码中的ResourceGroupManager注册监听者listener

ResourceGroupManager::getSingleton().addResourceGroupListener(&listener);
```

在上面代码7-5中，我们展示了一个UI中 “ 载入进度条 ” 的实现代码。在代码中，我们只关心载入每个资源之后更新进度条的动作，而不关心具体是世界地图还是单独的资源本身。对于资源管理器回调的其他方法，我们只负责实现了相应的虚函数，并没有做具体工作（因为是对虚接口的实现，它们除了单纯的 “ 存在 ” 之外并没有其他用处）。

我们可以注意到示例代码中的ProgressMeter类，它并不是Ogre框架中的成员。我们假设它是你在自己的应用程序中实现的类型。

Ogre中的 “ 档案（Archive） ”

对Ogre中 “ 档案 ” 的概念最基本的描述是：把一系列文件储存在一起的文件夹。 “ 档案 ” 本身和里面的文件都可以使用压缩格式。而 “ 档案 ” 本身存在的价值是：为Ogre提供一个文件访问的入口。

Ogre对档案中的文件操作是通过Archive接口实现的。你可以在自己的应用中根据需要来实现不同的档案形势（只要继承Archive接口就能很好的被系统支持）。默认的情况下磁盘目录是一种基本的档案实现方式。在Ogre的发行版本中，你会找到基于ZIP文件实现的档案类型，可以直接对PKZIP压缩格式的文件进行操作。

这两种不同的档案实现方式都有自己的优点，在构建和测试应用程序的时候，更适合直接在硬盘操作，因为这样就允许你在测试错误的时候方便快速的改变和转换文件，而不需要进行繁琐的压缩和解压缩操作。而在发行版本中，你可能更需要一个高压缩比例的ZIP文件来在发布的程序中压缩资源。

像在本书前面所提到的一样，各种档案系统也是作为插件“插入”Ogre框架中来的。进而你可以很方面的扩展自己需要的档案实现方式。虽然在大部分情况下操作磁盘文件的FilerSystemArchive和操作压缩格式文件的ZipArchive已经足够使用，但这并不代表你不能扩充，比如实现一个直接在局域网操作传输数据的“网络档案”。

档案管理器（ArchiveManager）

下面介绍实现一个新的“档案”类型的方法。其中我们的档案类型MyArchive继承于Archive接口。然后实现相应的工厂方法MyArchiveFactory，注册到文档管理器ArchiveManager中。

```
class MyArchive : public Archive

{

// 在这省略了具体的实现

}

class MyArchiveFactory : public ArchiveFactory

{

//在这省略了具体的实现ArchiveFactory的一些方法


Archive* createInstance(const String& name){

return new MyArchive(name, “MyArchiveType”);

}

};

void function()

{

MyArchiveFactory *factory = new MyArchiveFactory;

ArchiveManager::getSingleton().addArchiveFactory(factory);

}
```

代码7-6：注册一个新的Archive类型到Ogre中

当你向档案管理器注册了相应的工厂方法之后，就可以放心大胆的在程序中或者脚本中使用新的档案类型了。例如在前面的例子中，系统通过解析resources.cfg文件中的“ZIP”关键字来调用ZipArchive对象处理相应的压缩文件。如果你把其中的“ZIP”改成“MyArchiveType”，就可以通过自定义的档案格式MyArchive来处理文件系统了。

通过自定义“档案”实现自定制资源载入

在这里可以回想一下我们前面提到过手动创建资源载入的概念。在“档案”中我们也可以实现相同的工作。这是因为Ogre只是通过Archive接口来操作资源文件，而并不在意所谓的“资源文件”是一直保存在硬盘上还是刚刚通过程序在内存区中创建。根据这个特性，我们可以通过Archive动态的在内存中生成我们想要的“资源文件”。这里有一个比较实用的例子，在很多游戏中是通过一种被称为“wad”的文件[1]保存游戏数据的。你可以在程序运行时候把需要的wad文件拷贝到内存中来，当应用程序需要具体的文件时候。通过拷贝wad文件映射内存中的所需部分动态创建出一个文件交给应用程序使用。这种处理方法能很好的封装各种不同数据文件格式的区别，进而通过Ogre资源管理系统处理非Ogre支持格式的文件，以达到扩展Ogre的目的。

结语

在3D应用程序中的资源管理器一般都不是一个很复杂的主题，但是它却相当重要。糟糕的资源管理器会导致执行效率下降和用户的抱怨。所以根据这章节所学到的知识，在你的应用程序中应该尽可能灵活的运用资源管理器来维护资源的使用，在必要的时候也可以实现自己的资源管理策略。Ogre提供给你一组通用且灵活的资源管理接口来帮助你的工作。

从下一章节开始，我们将真正的接触到与GPU相关的细节，并逐渐掌握驾驭Ogre高级特性和硬件加速性能的能力。

- 尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规
- 尊重网上道德，遵守中华人民共和国的各项有关法律法规
- 承担一切因您的行为而直接或间接导致的民事或刑事法律责任
- 中国网游研发中心新闻留言板管理人员有权保留或删除其管辖留言中的任意内容
- 您在中国网游研发中心留言板发表的作品，中国网游研发中心有权在网站内转载或引用
- 参与本留言即表明您已经阅读并接受上述条款

昵　　称：

GO

联系EMAIL：