第1章 CEGUI的简介

1.1CEGUI历史和本书使用的版本

CEGUI（Crazy Eddie's GUI http://www.cegui.org.uk）是一个自由免费的GUI库，基于LGPL协议，使用C++实现，完全面向对象设计。CEGUI开发者的目的是希望能够让游戏开发人员从繁琐的GUI实现细节中抽身出来，以便有更多的开发时间可以放在游戏性上。

CEGUI的渲染需要3D图形API的支持，如OpenGL或Direct3D。另外，使用更高级的图形库也是可以的，比如OGRE、Irrlicht和RenderWare等，关键需求可以简化为二点：

纹理（Texture）的支持直接写屏（RHW的顶点格式、正交投影、或者使用shader实现）

本书截止日时，CEGUI的最新版本是0.6.0（本书的讨论也是基于此版本），本书光盘提供了SDK和全部源码的下载。

除此之外，CEGUI还同步提供了官方界面编辑器LayoutEditor和ImageSet编辑器，以方便UI和图像集的制作。作为界面编辑器，它需要系统级界面以提供编辑器操作，0.3.0版是基于MFC实现的；而在0.4.0版本以后，改为基于wxWidgets（跨平台的本地UI框架，这里的UI指Window操作系统底层，如：Windows、Unix和Mac，详见http://www.wxwidgets.org）实现。

目前将CEGUI作为游戏界面库开发的游戏已经有好多种，国内的天龙八部，巨人等游戏就是很好的例子。

CEGUI的功能是非常强大的，而且使用也非常的灵活，可以和脚本配合。可以通过配置文件自定义窗口外观。通过布局文件实现窗口布局等等特性，使得游戏的界面开发更加方便。

1.2 CEGUI的编译和例子介绍

CEGUI是个开源的项目，它和其他开源项目一样，需要依赖一些其他的开源库。因此给编译它带来了一些麻烦。因此我们这一节专门介绍CEGUI的编译和CEGUI例子的编译。

1.2.1CEGUI源代码的简介

CEGUI的官方网站上可以下载CEGUI的全部源代码，包括图像编辑器和布局编辑器。读者可以从以下网址下载http://www.cegui.org.uk/在它的download栏目里有CEGUI各个版本的下载。截至本书写作的时候最新版是0.6.0。本书采用的正是此版。还有一种方法用户可以在随书的光盘上找到CEGUI的各种源代码。源码放在了光盘的CEGUI源代码\目录下。这个目录下的和在CEGUI官网上下载到的内容完全一样。此外我们还需要下载CEGUI的依赖库，可以在光盘的CEGUI依赖库目录下找到，也可以在官方网站上找到。

将CEGUI的源代码解压到本机的某个文件夹下面，下面假设为C:\CEGUI。读者解压后可能出现目录CEGUI目录下面出现两层CEGUI-0.6.0子目录。读者可以把他们拷贝到CEGUI目录下面，这样目录层次比较简单，当然读者也可以不拷贝使用C:\CEGUI\CEGUI-0.6.0\CEGUI-0.6.0目录。不管在那个目录里面，最终CEGUI的源代码内容如图1-1所示。

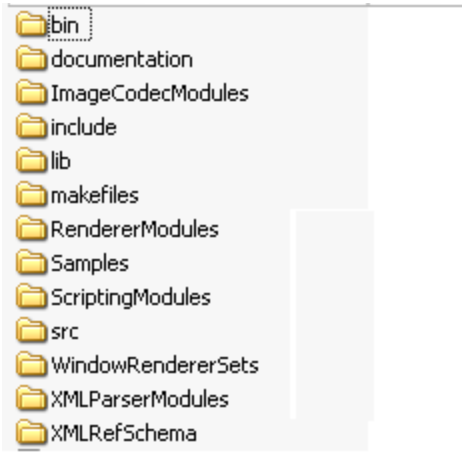

图1-1 CEGUI源代码目录

bin目录是CEGUI编译后的可执行文件和动态库的生成路径，**documentatioin**目录是CEGUI的文档目录，目前这两个目录是空的。bin里的内容还没有生成，documentatioin目录下为空，CEGUI的文档我们专门提供一个光盘目录（读者可以在光盘上的CEGUI文档目录找到所有的文档）来保存。**ImageCodecModules**目录里是CEGUI封装的各种图像解析库。CEGUI封装了FreeImage，SILLY，DevILI，Corona解析库，另外还自己实现了对TGA文件的支持。总的来说这个目录是CEGUI提供的图像解码模块。一般只使用其中一个库就可以了。但CEGUI这里对他们都提供了支持。类似的还有渲染模块和XML解析模块。**include**目录是CEGUIBase（基本的CEGUI库）的所以头文件。这样把头文件和实现文件分开是为了用户使用CEGUI作为动态库是比较方便。lib目录目前为空，编译后会产生一些库文件。**makefiles**目录是CEGUI产生各个平台上编译器文件的目录。这个目录下面有mac和premake两个目录mac目录是为苹果的Mastonish OS 提供的。premake是为Windows操作系统提供的。在这个目录下面有一些bat文件，比如说build_vs2005.bat。它们是用来产生Visual Studio工程文件的。2005对应这visual studio 2005 IDE。**RendererModules**目录是CEGUI提供的渲染模块的集合，里面有directx8 ，directx9，opengl和Irrlicht的支持。有些版本还有对OGRE（目前最火爆的开源游戏引擎）。**Samples**目录里包含了CEGUI提供的所有例子的源代码。如图1-2所示：

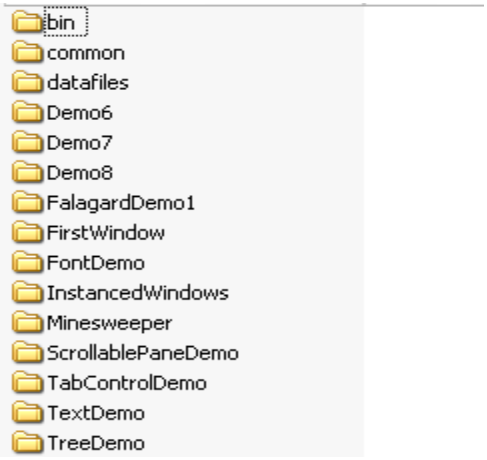

图1-2 例子目录

ScriptingModules目录脚本模块目录，CEGUI提供对脚本的支持，它使用了tolua++（对lua提供封装的一个开源库）提供对lua的支持。笔者认为这个库并不是非常好用，建议使用LuaPlus，这个库对Lua做了简单的封装，而且非常容易使用。以后本书涉及到脚本模块后会使用该库作为Lua的封装库。该库的源代码在光盘的LuaPlus目录下。**src**目录是CEGUIBase的所有源代码的目录。**WindowRendererSets**目录，这个目录下保存了所有的渲染CEGUIBase里窗口的类。（注意：CEGUI以前的版本和本书使用的最新版，这里变化很大，在1.1节有详细介绍）这个目录里面包含有Falagard\include和Falagard\src两个子目录。**XMLParserModules**目录包含了四种XML解析库的源代码，他们分别是expatParser，libxmlParser，TinyXMLParser，XercesParser。这四个子模块分别通过四种XML解析库实现了CEGUI需要的XML解析模块。不论使用那一个模块都可以。**XMLRefSchema**目录里提供了CEGUI各种XML文件的Schema描述文件。如图1-3所示：

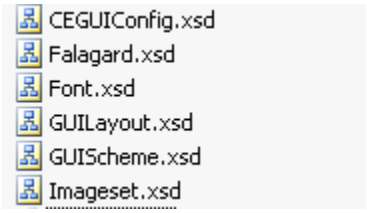

图1-3 模式文件

如果读者对xsd文件不是很熟悉，请阅读相关的XML书籍。这里简单介绍一下这些文件是做什么的。DTD和XSD文件是描述XML文件元素的文件，XSD文件比DTD文件更好用一些。XML解析器通过读取这两种文件后，在读取以他们为标准定义的XML文件的时候，就可以检查该XML文件标记是否合法，元素的属性值，文本是否类型合法，值是否合法。总之就是做

XML语法分析的文件。由于XML解析器需要支持他们，而且他们的规则是统一的，所以XML解析器，通过他们来判断一个XML文件是否合法。

到这里CEGUI目录下的子目录的功能都已经介绍完了。读者可能还是一头雾水，不用担心随着本书慢慢理解吧。

1.2.2CEGUI源代码编译

本书的开发平台是windows xp，visual studio 2005。CEGUI提供了Windows平台下对VC 2002 , VC2003 , VC2005的支持。目前我们还不知道在那里产生VC的工程文件。你可以自己见一个空的工程，然后把源文件添加进去。这种方法是非常麻烦的，你将会遇到许多麻烦的问题，包括依赖库的设定，工程类型的调整，依赖头文件的设定等等。幸好CEGUI为我们已经做好了这一切。还记得1.2.1中介绍的**makefiles**目录吗，该目录下面有个premake子目录，里面有build_vs2005.bat批处理文件。现在我们打开这个目录，执行这个批处理。就会在相同目录里产生CEGUI.sln的工程文件，他就是我们需要的VC2005的解决方案文件。同理执行build_samples_vs2005.bat，可以产生CEGUISamples.sln。这是CEGUI例子的解决方案文件。好了，现在可以打开这两个sln文件，如果你的机器无法打开这两个文件，请安装VC2005。（本书使用VC2005，读者也可以使用其他版本，注意区分批处理文件）。

我们打开CEGUI.sln文件。VC打开解决方案后读者会看到如1-4所示的工程目录。一共有13个工程， 默认的启动工程是CEGUIBase（粗体显示）。工程可以分为以下6类：

1.CEGUI基本库，CEGUIBase，这个库是CEGUI最早的库，第一版的CEGUI就有这个工程。

2.CEGUI的渲染集，CEGUIFalagardWRBase库，这个库在现在这一版改动是最大的。

3.XML的解析库，共有四个库和1.2.1中介绍的四个对应。但CEGUI的解决方案里只使用了expatParser一个工程。

4.图像解码库，所有工程名带codec的都是图像解码的工程。共有四个工程。

5.渲染库，工程里共有两个CEGUI渲染工程，一个是OpenGL的一个是DirectX9的。

6.脚本模块库，共有三个工程，工程名都含有Lua，表明CEGUI通过Lua实现脚本的支持，我们前面已经讲过，本书使用LuaPlus作为CEGUI的Lua封装库。

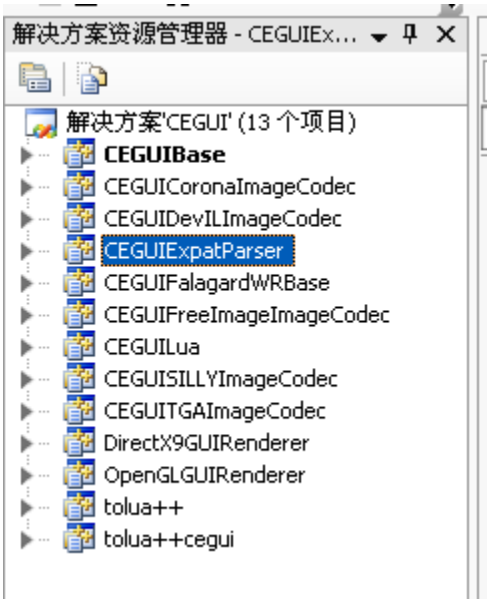

图1-4 工程结构图

读者可能想，是不是每一个使用CEGUI的程序都必须包含这么多的工程呢？其实CEGUI基本库只有先前讲的1和2。其他模块都是通过实现CEGUI提供的接口来和CEGUI挂起钩来的。脚本模块的接口是ScriptModule，图像解码模块的接口是ImageCodec，渲染模块的接口是Renderer和Texture。读者也可以自己实现这些接口，但我认为是没有那个必要的。既然有现成

的为什么不用呢。除非读者自己实现了一个渲染引擎，希望使用CEGUI作为引擎的界面库，哪么读者需要自己实现Renderer接口和Texture接口。或者读者有特殊需求。

CEGUI提供了动态库和静态库两种配置类型，分别提供了Rrelease和Debug版。也就是说读者可以使用CEGUI作为动态链接库，也可以将CEGUI集成到自己的库或者可执行文件中。别人并不知道你使用了CEGUI的库。具体配置如图1-5所示：

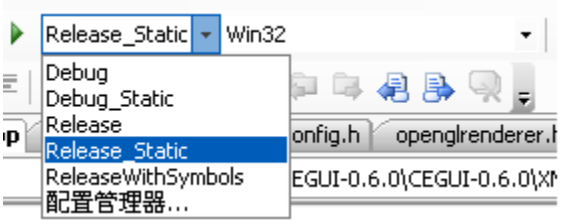

图1-5 CEGUI编译类型配置

ReleaseWithSymbols配置，是在Release版下调试用的。笔者在调试了CEGUI提供的例子程序时候发现，Debug版程序无法启动。这时读者可以使用这个配置项调试你的程序。

现在如果读者编译工程，将会出现无法找到一些头文件和库文件的错误。这是因为CEGUI使用了一些其他的库，所以要把依赖库设置好。依赖库和头文件CEGUI工程已经设置好了，我们只需要将依赖文件拷贝到工程的设置目录。工程设定的目录如何知道呢？打开CEGUIBase的属性对话框（如何打开？不会不知道吧。主菜单->项目->属性，或者在工程上单击右键在弹出的菜单选择 属性）。找到配置属性->C++->常规选项卡。读者会发现附加包含目录。如图1-6所示。

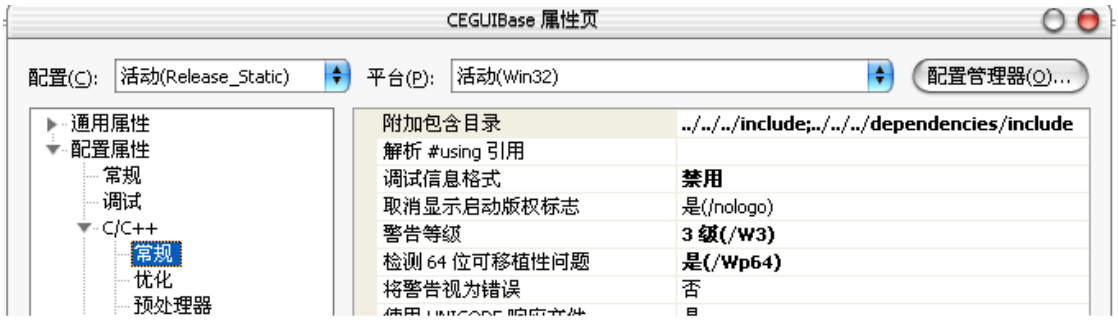

图1-6 常规选项卡

显然可以发现依赖库的目录是dependencies/include它在我们的CEGUI主目录下面。我们现在还没有这个目录，所以我们需要下载CEGUI的依赖库。在光盘的CEGUI依赖库目录里可以找到一个dependencies目录，请读者把它拷贝到CEGUI的主目录下。1.2.1节介绍了主目录的结构，以后我们提起主目录都是指的那个目录（C:\CEGUI）。依赖库里包含了CEGUI需要的所有的动态库（二进制文件.dll），头文件（.h或者.hpp），库文件（.lib），以及一些许可文件或者叫授权文件。

同理用户可以查看链接器选项卡下面的输入子项和常规子项。查看CEGUI用到的库文件和输入库的路径。（也在dependencies目录里面，只要读者拷贝了那个目录，库连接也同时设置好了）。需要注意的是静态库是没有链接器选项卡的。

好的，到现在读者可以编译每一个工程了。如果读者不知道如何编译，或者其他Visual Studio的基本操作，请参考介绍Visual C++和VisualStudio的书籍。

读者可以编译静态库（默认的编译配置类型），也可以设置为动态库类型。静态库的生成文件比较大，动态库生成的库文件和可执行文件都比较小。等待编译完成后将会在CEGUI主目录下的lib目录和bin目录下分别生成库文件和动态库执行文件（.dll）。CEGUI生成文件的命名规则是。静态库：静态库工程名+_Static，Debug版：工程名+ _d，比如CEGUIBase工程的静态库Debug版就会生成名为CEGUIBase_Static_d.lib的静态库文件。静态库不会生成可执行的Dll文件，只会生成一个库文件。

1.3 CEGUI官方编辑器的介绍

CEGUI提供了许多官方编辑器，其中最重要的就是图像集编辑器和布局编辑器。读者可能不理解什么是图像集，什么是布局。不要担心，后文会有详细介绍。具体如何使用这两个编辑器请读者查看光盘上的 教学录像\编辑器简介。如何使用使用文字是无法表达清楚的，所以我们提供了教学录像。

1.3.1 CEGUI布局编辑器

什么是布局呢？我们都知道窗口是由一系列的控件（一些具有某些功能的窗口）组成的，哪么这些控件在父窗口上如何分布呢，他们的大小怎么指定，当父窗口大小变化后子窗口该如何调整呢。窗口的重叠关系如何设置等，这些我们称作窗口的布局。窗口的布局可以在c++代码里指定，创建一个窗口的时候设置子窗口的位置，属性等信息。也可以在布局文件里指定。CEGUI支持这两种方法。布局编辑器就是为在布局文件里指定而设计的。它可以以所见即所得的方式产生布局文件。当CEGUI加载这个布局文件的时候会自动的创建，文件里指定的窗口。从而使得窗口创建更加的方便。

本书提供了布局编辑器的源代码和可执行文件。在光盘的CEGUI编辑器\布局编辑器\目录下有可执行文件和源代码。需要注意的是布局编辑器使用了开源的跨平台窗口库wxWidgets。这个库在CEGUI编辑器\依赖库。编译这个布局编辑器源代码时首先要编译wxWidgets库。这个过程比较复杂，而且不容易设置正确，所以作者提供了已经编译好的库文件和所需的头文件，方便读者的编译。如果不是必须，请读者使用布局编辑器的可执行版本。因为编译这个编辑器是比较困难的。

编辑器如何使用呢？CEGUI编辑器是需要安装的，在光盘的CEGUI编辑器\布局编辑器\可执行，目录下有CEGUI布局编辑器的安装程序。安装完成后启动编辑器，会弹出一个对话框告诉你，没有找到默认的配置文件，单击确定后会出现如图1-7所示的选择具体的数据目录的窗口。

数据文件在那里呢？我们使用CEGUI例子提供的数据。读者可以将CEGUI主目录下面找到Samples目录，读者可以将Samples下的datafiles目录设置为编辑器需要的数据目录。设完后可能会弹出一个，错误对话框，不要管它。下面就可以使用编辑器编辑布局文件了。读者自己练习吧。

注意：
CEGUI无法在含有中文的目录里正确加载各种资源，所以布局编辑器的工作目录必须不包含中文字符。

图1-7 数据目录选择

下面是一段布局文件的XML代码，读者可以对它们有个感性认识。

```
<?xml version="1.0" ?>
<GUILayout>
<Window Type="DefaultGUISheet" Name="">
  <Window Type="TaharezLook/StaticText" Name="Text">
    <Property Name="FrameEnabled" Value="False" />
    <Property Name="HorzFormatting" Value="WordWrapCentred" />
    <Property Name="VertFormatting" Value="CentreAligned" />
    <Property Name="UnifiedAreaRect" Value="{{0.5,0},{0.1,0},{0.9,0},{0.4,0}}" />
    <Property Name="BackgroundEnabled" Value="False" />
```

```
  </Window>
  <Window Type="TaharezLook/Button" Name="Button1">
    <Property Name="Text" Value="- One -" />
    <Property Name="UnifiedAreaRect" Value="{{0.1,0},{0.1,0},{0.3,0},{0.2,0}}" />
  </Window>
  <Window Type="TaharezLook/Button" Name="Button2">
    <Property Name="Text" Value="- Two -" />
    <Property Name="UnifiedAreaRect" Value="{{0.3,0},{0.3,0},{0.5,0},{0.4,0}}" />
  </Window>
  <Window Type="TaharezLook/Button" Name="Button3">
    <Property Name="Text" Value="- Three -" />
    <Property Name="UnifiedAreaRect" Value="{{0.5,0},{0.5,0},{0.7,0},{0.6,0}}" />
  </Window>
  <Window Type="TaharezLook/Button" Name="Button4">
    <Property Name="Text" Value="- Four -" />
    <Property Name="UnifiedAreaRect" Value="{{0.7,0},{0.7,0},{0.9,0},{0.8,0}}" />
  </Window>
</Window>
</GUILayout>
```

这段代码来自刚才的数据目录下的layouts\TabPage.layout文件，是CEGUI已经编辑好的一段代码。具体这段代码的含义，这里我们不做介绍，以后章节专门讲解。

1.3.2 图像集编辑器

何为图像集呢？一个图像集代表一张图片，而图像又代表了这章图片上的某个区域。为什么要这么设计呢？这是和渲染窗口相关的。当你要渲染一个CEGUI窗口，你必须找到窗口所要贴的纹理图片。计算出它们的纹理坐标。如果每个小窗口都单独需要一张图片，哪么会有两个问题。第一，显卡处理2的幂尺寸的图片（纹理）时效率最高，所以图片要是2的幂才行。第二，由于第一的限制，每个窗口一张图片必定导致大量的数据浪费。而且显卡在切换纹理的时候，消耗是很大的。哪么CEGUI的解决方案是用一张大的图比如512*512的图，贴上许多小图。每个小图就代表一个CEGUI窗口需要的贴图。这样就可以避免上面的麻烦了。可以说这个概念在所有的游戏GUI系统里都有。下面是一段ImageSet的XML代码。读者看后应该会明白什么是图像集了。

```
<?xml version="1.0"?>
<Imageset Name="DriveIcons" Imagefile="DriveIcons.png" NativeHorzRes="800" NativeVertRes="600" AutoScaled="true">
    <Image Name="Artic" XPos="0" YPos="0" Width="32" Height="32" />
    <Image Name="Lime" XPos="32" YPos="0" Width="32" Height="32" />
    <Image Name="Sunset" XPos="64" YPos="0" Width="32" Height="32" />
    <Image Name="Black" XPos="96" YPos="0" Width="32" Height="32" />
    <Image Name="Blue" XPos="128" YPos="0" Width="32" Height="32" />
    <Image Name="Silver" XPos="0" YPos="32" Width="32" Height="32" />
    <Image Name="DriveStack" XPos="32" YPos="32" Width="32" Height="32" />
    <Image Name="GreenCandy" XPos="64" YPos="32" Width="32" Height="32" />
    <Image Name="GlobalDrive" XPos="96" YPos="32" Width="32" Height="32" />
</Imageset>
```

Imageset代表一张图像，它有名称DriverIcons，这个名称在XML文件或者程序里用来标识这张图片，Imagefile指定图片相对于ImageSet文件的目录，用来找到这张图像，NativeHorzRes和NativeVertRes指定图像的初始分辨率，AutoScaled指定图像是否能自动缩放。Image描述的就是每个窗口需要的贴图了，它有名字和在整张图像（ImageSet）里的位置信息，大小信息。

图像集编辑器不需要安装，读者在光盘的 CEGUI编辑器\图像集编辑器\可执行，目录下可以找到CEImagesetEditor-0.6.0.exe.zip的压缩包，把它解压到一个安装目录就可以了。它的源代码在光盘上的 CEGUI编辑器\图像集编辑器\源代码， 目录下，同布局编辑器一样它也使用了wxWidgets，编译比较复杂，有兴趣的读者，可以自己尝试修改这个编辑器。

1.4 CEGUI总体架构

CEGUI设计了许多接口，程序可以通过实现接口来为CEGUI提供服务。比如说渲染接口定义好了以后，就使得CEGUI与具体的渲染平台无关。不管是OpenGL还是Direct3D，无论是Windows还是Linux，只要在那个平台上实现了CEGUI的渲染接口。当然CEGUI库也必须在那个平台下编译才行。CEGUI的整体架构图如1-8所示：

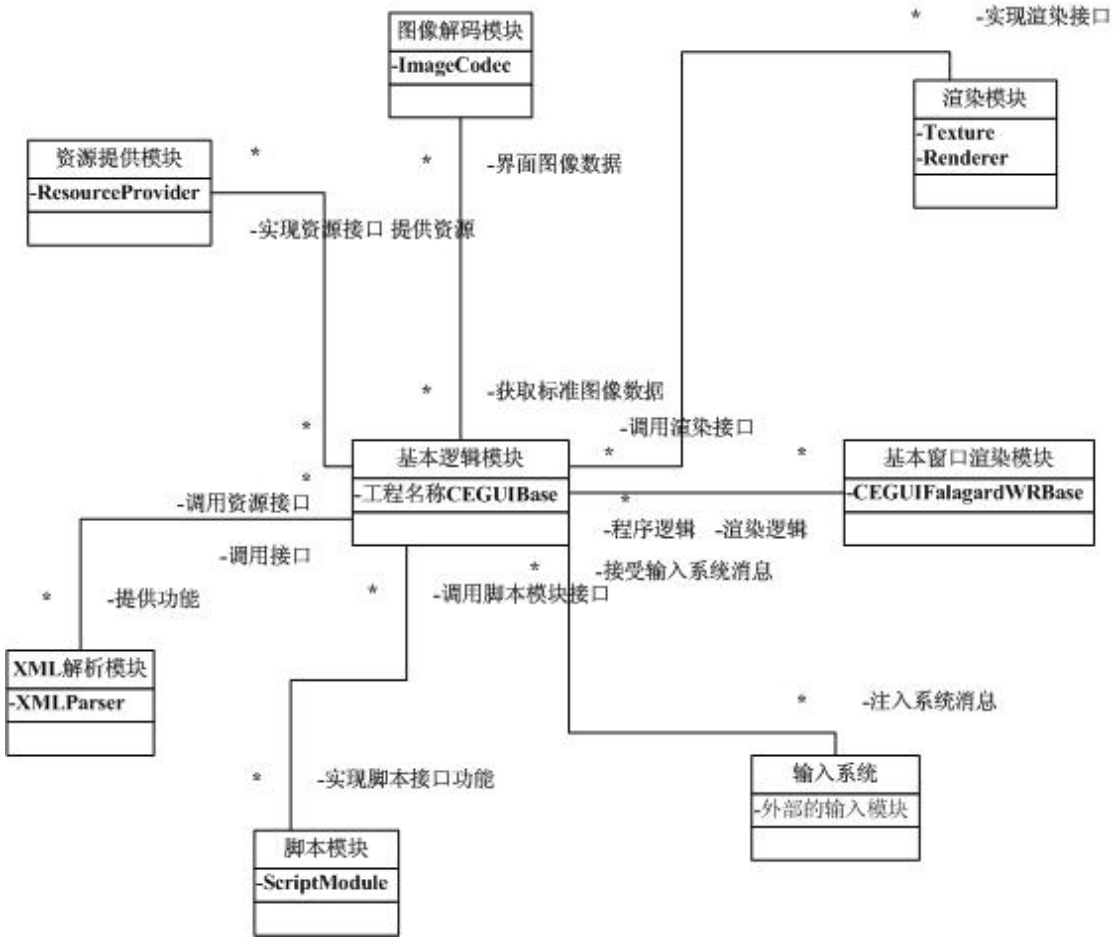

图1-8 CEGUI的整体架构图

CEGUI的核心模块是CEGUIBase，这个模块成为联系其它各个模块的核心模块。要获取资源通过该模块的资源接口可以获得。要执行脚本通过该模块的脚本模块接口可以获得。要获取贴图，可以调用渲染接口的函数。所以说CEGUIBase这个模块读者需要多加注意，深入理解才能真正的理解CEGUI的设计思想和设计方法。

1.5本章小结

为了使读者更快的理解CEGUI，并且可以使用它编写程序。本书设置了每章习题，希望通过作者出的各种习题来提高读者的动手能力。作者始终认为，只有自己亲自动手做一遍，才能体会到其中的许多细节，而且更有成就感。听懂了，看懂了，不代表自己能做出来。所以希望读者能认真的做每一章的习题，自己踏踏实实的学习。

本章主要是讲述了CEGUI的编译方法和CEGUI提供的编辑器的安装使用等并没有设计到CEGUI的核心思想和设计思路。所以本章的习题就相对简单（对与新手来说，也不一定很简单）。

习题1：自己亲自动手在自己的机器上完整的编译一遍CEGUI的所有工程，包括例子工程。虽然本书没有介绍例子工程的编译，但是有了CEGUI工程的编译经验，读者一定能够自己完成。自己看看CEGUI的例子演示，也可以增强自己学学CEGUI的兴趣。

习题2：使用CEGUI的布局编辑器实现一个对话框，并不需要有什么动作（就是界面操作），只要可以在界面中显示出来就可以。但是读者可能不知道怎样才能显示在界面里。没关系读者可以参考例子里的代码，如果还是做不出来，可以先放下等待以后在做。对话框的内容读者自己设计，可以参考例子。同时读者可以打开例子代码里的looknfeel文件，看看它是什么样子，怎么设计的。

习题3：使用CEGUI的图像集编辑器，打开CEGUI例子提供的imageset文件，自己熟悉ImageSet的概念。同时亦可以自己制作一个ImageSet文件。

如果是第一次接触CEGUI的读者可能会感到有困难，请不要放弃，自己试试，你会发现慢慢的你会思考问题和解决问题了。

本章到此结束，下一章介绍CEGUI的最基本的结构，属性和事件。

第2章 CEGUI事件系统和属性系统

CEGUI的基本系统，事件系统和属性系统。他们虽说不是很复杂，但是是非常基础重要的结构。图2-1是CEGUI事件系统，属性系统是如何与窗口基类交互的。从图中可以发现窗口基类是派生自事件集和属性集。因此窗口具备了处理事件和处理属性的能力。读者可能非常的疑惑。什么是属性，什么是事件？

属性是窗口（不完全是窗口，其他任何希望有属性的结构都可以有属性，事件也一样）提供的外部可以操作（设置和获取值）一组数据。这组数据会影响窗口的行为，外观等等。哪么为什么把简单的设置数据和获取数据，搞得这么复杂呢？主要是为了在脚本中使用，在XML文件中描述的方便。如果单纯的使用c++代码来设置一个属性是非常简单的事情，但每次窗口改变后都必须重新编译程序。我们知道窗口的修改是非常频繁的。这样势必效率奇低。哪么为什么增加了属性就可以是XML的定义方便，脚本操作方便呢？因为它统一了接口，在XML中只需要定义一个标记Property，就可以设置所有的变量了。Name标记标识属性的名称，Value标识属性的值。下面这个属性描述的是文本属性（就是窗口要显示的文字），它的值是“Option 1”。

```
<Property Name="Text" Value="Option 1" />
```

下面的属性描述窗口大小的最大值UnifiedMaxSize，它的值是一个格式化好的字符串，这个属性知道怎么把它转化成自己的数据。当获取属性后得到的数据格式也是这个格式的。

```
<Property Name="UnifiedMaxSize" Value="{{1,0},{1,0}}" />
```

提示：
上面的代码是从layout文件中抽取的，CEGUI解析布局文件的时候是区分字母大小写的，Property和property以及PROPERTY等等都是完全不同的，而且布局文件必须是以utf-8格式保存的数据。布局文件，looknfeel文件，schema文件都是XML文件，读者也可以修改这些文件以xml为扩展名，这样方便查看。CEGUI对这些文件的扩展名没有要求，可以是任意的扩展名。例如looknfeel文件可以设置为.looknfeel.xml为它的扩展名。

通过这种方法，属性可以很方便的在XML文件描述了。哪么属性如何在脚本里方便的设置呢？脚本只需要提供两个函数SetProperty和GetProperty就可以获取和设置所有的属性了。例如,下面两个函数就可以很好的设置和获取属性了。当然脚本函数不是这样设计的，这里这是个示意。

```
void SetProperty(String name, String value);
String GetProperty(String name);
```

事件是窗口通知外部（指的是这个窗口以外的任何事物，对象，比如父窗口，其他任何对象，只要它关心这个事件，并且希望处理它）自己内部状态改变的一种机制或者方法。比如Window系统通过消息来通知外部（其实是应用程序本身）系统的状态或者应用程序的窗口状态发生了改变。之所以要有事件通知外部，而不是应用程序轮询（DOS时代的技术），是为了提高效率和满足多进程（线程）并发的需求。CEGUI的事件也是如此，外部的模块希望知道CEGUI内部窗口目前的状态或者状态的改变，不可能是每次遍历所有的窗口来检查窗口的状态是否改变，是否需要做某些处理。回调函数可以很好的解决这个问题，比如我们希望捕获按钮的单击状态，哪么我们可以设一个按钮单击的回调函数，当窗口被单击时，窗口会调用这个回调函数来通知外部，现在按钮被单击了，外部来处理这个事件。这样就很好的避免了轮询。事实上CEGUI提供的事件机制也是一种回调的机制。当事件被激发后事件会调用注册在自己的所有的处理函数。这样的优势是接口统一，处理函数是多种多样的（2.1会详细介绍）。

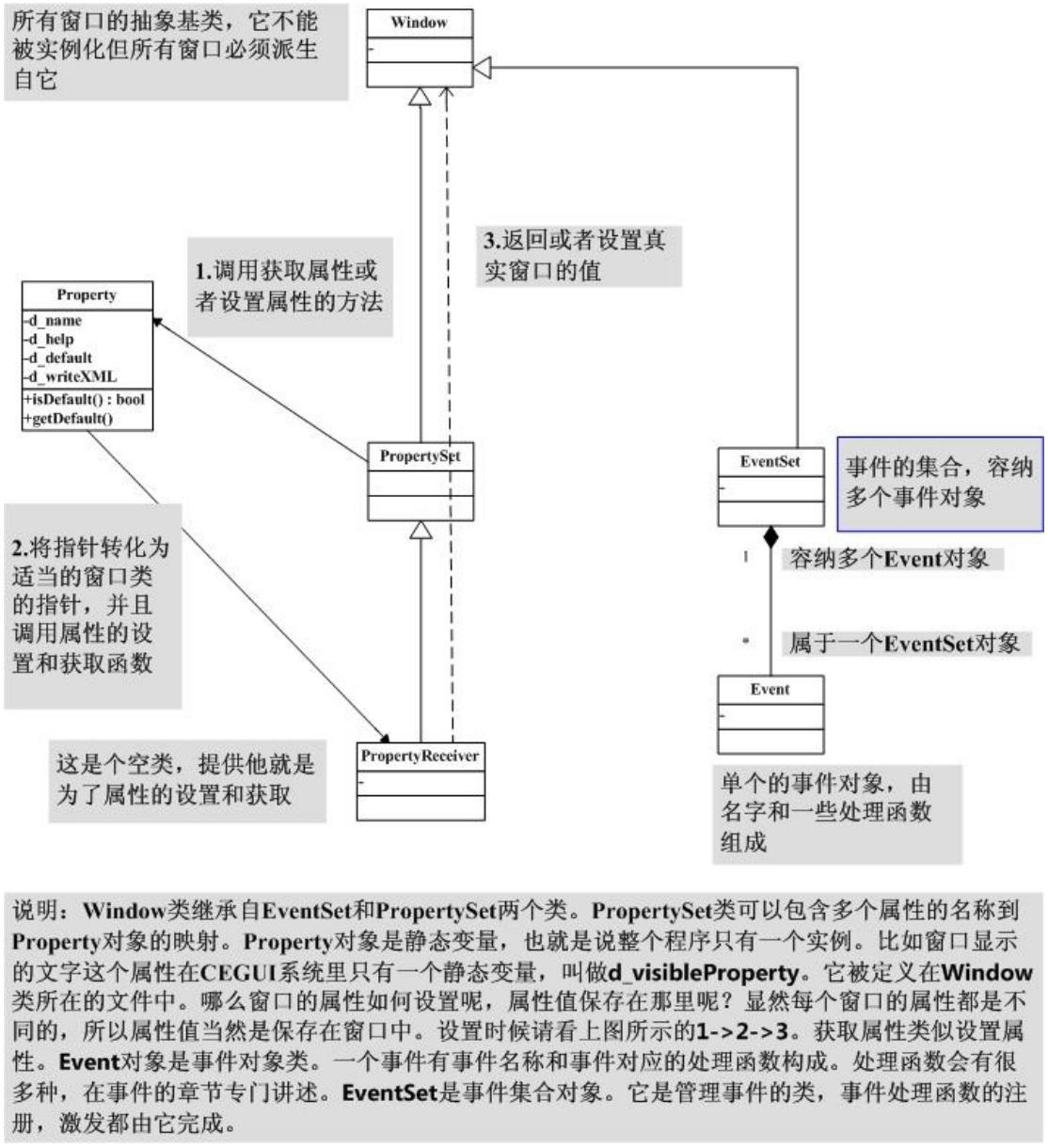

图2-1 属性和事件架构

2.1 CEGUI的事件系统

事件系统无非就是，当这个事件发生时调用一个（些）函数来处理这个事件。所以事件系统主要由一个事件的标识（在CEGUI中是以字符串来标识的）和一个（些）处理函数组成。处理函数可能是多种类型的函数，注要是方便用户设置回调函数。比如，静态函数，可以是全局的也可以是类的静态函数。类的成员函数，注要是为了方便的将一个类的成员函数指定为回调函数，读者可能觉得疑惑，类的成员函数怎么能作为回调函数呢？不要着急，下文有详细阐述。函数指针，定义一种类型的函数，赋值给他一个地址就可以被回调了。

CEGUI为了统一这些函数类型定义了一些类来统一处理各种函数类型。这些类必须有个虚基类才能方便的在一个循环里调用，这个类便是SlotFunctorBase。

```
class SlotFunctorBase
{
public:
   virtual ~SlotFunctorBase() {};
```

```
    virtual bool operator()(const EventArgs& args) = 0;
};
```

可以看到这个类只有两个虚函数。一个是析构函数，一个是重载的操作符（）。操作符用来调用函数（类代表的一类回调函数）。虚的析构函数可以调用到正确的派生类的析构函数。 从这个接口派生出了FreeFunctionSlot，MemberFunctionSlot，FunctorReferenceSlot，FunctorPointerSlot和FunctorCopySlot五个类。这五个类分别代表自由函数（静态函数），类的成员函数，某个类（可以是前两个类，但一般不需要）的引用，某个类的指针，某个类的一份拷贝。还有一个比较特殊它不是派生自SlotFunctorBase，它就是ScriptFunctor，稍后详细介绍。

FreeFunctionSlot类代表的是静态函数，因为这里函数只要知道函数的地址，就可以调用，因为它的没有隐含的参数，所有参数都是明确的指定的。（不像类的成员函数有个隐含的参数this）。因此这个类的定义非常简单如下所示：

```
class FreeFunctionSlot : public SlotFunctorBase
{
public:
    //定义回调函数的格式
    typedef bool (SlotFunction)(const EventArgs&);
    FreeFunctionSlot(SlotFunction* func) :
        d_function(func)
    {}
    virtual bool operator()(const EventArgs& args)
    {
        return d_function(args);
    }
private:
    SlotFunction* d_function;
};
```

从代码中可以看到，这个类首先定义了一个回调函数的格式，然后保存这个回调函数的地址作为自己的变量d_function。调用操作符直接调用这个函数。

MemberFunctionSlot类用来设置类的成员变量为回调函数。类的成员变量有一个隐含的参数this，所以必须保存这个this到MemberFunctionSlot类里面。如下所示

```
template<typename T>
class MemberFunctionSlot : public SlotFunctorBase
{
public:
    //!成员变量的回调函数定义
    typedef bool(T::*MemberFunctionType)(const EventArgs&);
    MemberFunctionSlot(MemberFunctionType func, T* obj) :
        d_function(func),
        d_object(obj)
    {}
    virtual bool operator()(const EventArgs& args)
    {
        return (d_object->*d_function)(args);
    }
private:
    MemberFunctionType d_function;
    T* d_object;
};
```

typedef bool(T::*MemberFunctionType)(const EventArgs&)，定义了一个模板函数，T是某个类，MemberFunctionType是T的某个成员函数类型，就和静态函数类似，它的参数和返回值与自由函数一样。d_object是我们说的this，MemberFunctionType是成员函数的地址。如何调用呢？（d_object->*d_function）（args），相当于this->func(args)。这样成员函数就可以被调用了。

FunctorReferenceSlot是某个类引用的封装，这个类必须重载operator()，类似SlotFunctorBase中的定义。

```
template<typename T>
class FunctorReferenceSlot : public SlotFunctorBase
{
public:
   FunctorReferenceSlot(T& functor) :
      d_functor(functor)
   {}

   virtual bool operator()(const EventArgs& args)
   {
      return d_functor(args);
   }

private:
   T& d_functor;
};
```

这个模板类的参数T可以是任意一个重载了operator()的类，看看d_functor的定义，它是T的引用。所以FunctorReferenceSlot是引用Slot。

FunctorPointerSlot是某个类的指针的封装。

```
template<typename T>
class FunctorPointerSlot : public SlotFunctorBase
{
public:
   FunctorPointerSlot(T* functor) :
      d_functor(functor)
   {}
   virtual bool operator()(const EventArgs& args)
   {
      return (*d_functor)(args);
   }
private:
   T* d_functor;
};
```

这个类和FunctorReferenceSlot相似，它只是把引用换成了指针类型而已。

FunctorCopySlot是某个类的一份拷贝，他和FunctorPointerSlot，FunctorReferenceSlot有区别吗？它们分别保存的是FreeFunctionSlot，MemberFunctionSlot和ScriptFunctor的指针和引用而FunctionCopySlot是这个类的一个副本。

```
template<typename T>
class FunctorCopySlot : public SlotFunctorBase
{
public:
   FunctorCopySlot(const T& functor) :
      d_functor(functor)
   {}
   virtual bool operator()(const EventArgs& args)
   {
      return d_functor(args);
```

```
    }
private:
    T d_functor;
};
```

这里的T任何重载了operator ()的类。d_functor是这个类的实例。就像int a; a是int的实例一样。读者应该清楚了FunctorCopySlot，FunctorPointerSlot，FunctorReferenceSlot的区别了。

ScriptFunctor，这个Functor是比较特殊的，它没有派生自SlotFunctorBase，所以它的操作都要特殊处理，这一点在下文介绍中会详细介绍。这个类只有一个scriptFunctionName的变量，它保存的是脚本调用的函数。

```
class ScriptFunctor
{
public:
    ScriptFunctor(const String& functionName) : scriptFunctionName(functionName) {}
    ScriptFunctor(const ScriptFunctor& obj) : scriptFunctionName(obj.scriptFunctionName) {}
    bool operator()(const EventArgs& e) const;
private:
    // no assignment possible
    ScriptFunctor& operator=(const ScriptFunctor& rhs);
    const String    scriptFunctionName;
};
```

这个Functor的回调函数实现如下：

```
bool ScriptFunctor::operator()(const EventArgs& e) const
{
    ScriptModule* scriptModule = System::getSingleton().getScriptingModule();
    if (scriptModule)
    {
        return scriptModule->executeScriptedEventHandler(scriptFunctionName, e);
    }
    else
    {
        Logger::getSingleton().logEvent("Scripted event handler '" + scriptFunctionName + "'
    could not be called as no ScriptModule is available.", Errors);
        return false;
    }
}
```

容易发现它调用脚本模块执行这段脚本。executeScriptedEventHandler是脚本模块实现的调用脚本函数的方法。

这六个Functor可以分为两类，一类是封装一种函数的类，另一类是封装类似第一类的类。第一类共有三个类分别是FreeFunctionSlot，MemberFunctionSlot和ScriptFunctor，用户可以自己定义其他的类。FreeFunctionSlot封装的是静态函数，MemberFunctionSlot封装的是类的成员函数，ScriptFunctor封装的是脚本函数。第二类也有三个类分别是FunctorCopySlot，FunctorPointerSlot，FunctorReferenceSlot。FunctorCopySlot是第一类的副本，拷贝。FunctorPointerSlot是第一类的指针。FunctorReferenceSlot是第一类的引用。

这六种Functor 形式各异，无法提供一个统一的接口供上层逻辑调用。所以需要有一个类来完成这个工作，这个类便是SubscriberSlot。这个类定义如下：

```
class CEGUIEXPORT SubscriberSlot
{
public:
    //默认构造函数
    SubscriberSlot();
    //FreeFunctionSlot，自由函数的封装类
    SubscriberSlot(FreeFunctionSlot::SlotFunction* func);
```

```
    //标准析构函数
    ~SubscriberSlot();
    //调用函数的（）重载，具体由第一类Functor实现
    bool operator()(const EventArgs& args) const
    {
        return (*d_functor_impl)(args);
    }
    //返回成员是否有效，是否已经连接到一个具体的Functor实现
    bool connected() const
    {
        return d_functor_impl != 0;
    }
   //清除Functor的实例
    void cleanup();
    // 模板构造函数，以成员函数的封装为参数，MemberFunctionSlot。
    template<typename T>
    SubscriberSlot(bool (T::*function)(const EventArgs&), T* obj) :
        d_functor_impl(new MemberFunctionSlot<T>(function, obj))
    {}
    //引用Functor的构造函数，这个类使用了FunctorReferenceBinder类来辅助构造，为什
//么不使用T& binder呢？主要是为了和FunctorCopySlot副本构造函数区分而已。
    template<typename T>
    SubscriberSlot(const FunctorReferenceBinder<T>& binder) :
        d_functor_impl(new FunctorReferenceSlot<T>(binder.d_functor))
    {}
    //副本Functor的构造函数
    template<typename T>
    SubscriberSlot(const T& functor) :
        d_functor_impl(new FunctorCopySlot<T>(functor))
    {}
    //指针Functor的构造函数
    template<typename T>
    SubscriberSlot(T* functor) :
        d_functor_impl(new FunctorPointerSlot<T>(functor))
    {}
private:
    //内部基本Functor的指针，SlotFunctorBase基类的优势在这里用到了
    SlotFunctorBase* d_functor_impl;
};
```

这个类并没有封装ScriptFunctor，这个类的操作是单独的，并没有统一起来。connected用来判断各种函数类型是否有效（就看成员d_functor_impl是否为NULL。）cleanup用来清理资源，()操作符用来调用事件处理函数（内部调用SlotFunctorBase的对应函数）。

d_functor_impl是SlotFunctorBase的指针。

最后一个事件处理函数相关的类是BoundSlot它进一步的封装了事件处理类，是它把事件Event类和事件的处理类（各种Functor）联系起来。它新增了组的概念，新增了成员变量Event的指针。该指针主要是为了反注册自己。在disconnect函数中调用Event的unsubscribe函数完成反注册。

```
class CEGUIEXPORT BoundSlot
{
public:
    typedef unsigned int Group;
```

```
    //拷贝构造函数
    BoundSlot(const BoundSlot& other);
    //析构函数
    ~BoundSlot();
    //检查内部的SubscriberSlot是否连接
    bool connected() const;
    //断开内部SubscriberSlot的连接，断开和事件类的链接
    void disconnect();
    //判断两个BoundSlot是否相等
    bool operator==(const BoundSlot& other) const;
    //判断两个BoundSlot是否不等
    bool operator!=(const BoundSlot& other) const;
private:
    friend class Event;
    // 禁用赋值操作符
    BoundSlot& operator=(const BoundSlot& other);
    Group d_group;                  //给BoundSlot 分组
    SubscriberSlot* d_subscriber;       //实际的处理类指针
    Event* d_event;                 //表明那个Event对应的处理函数
};
```

这个类的几个重要的函数需要在这里介绍。他们是连接和断开连接的函数。

```
bool BoundSlot::connected() const
{
    return (d_subscriber != 0) && d_subscriber->connected();
}
void BoundSlot::disconnect()
{
    //清除事件处理类
    if (connected())
        d_subscriber->cleanup();
    // 断开这个处理类与事件的联系，下次事件发生后不会在调用这个处理类了
    if (d_event)
    {
        d_event->unsubscribe(*this);
        d_event = 0;
    }
}
```

上面是对事件处理函数的说明。事件标识（名称）在CEGUI里其实就是一个字符串，用它来描述不同的事件。

下面看重量级的类Event这个类用来描述一个事件。刚才我们已经说了事件其实就是由事件的名字（字符串）和事件的处理函数组成的。Event类正是由这两个要素组成。不过需要注意的是一个事件可以对应多个处理函数。因此Event类使用mutimap来保存所有的这个事件的处理函数。为什么不使用vector或list或map呢？主要是BoundSlot有个组的概念，需要使用mutimap保存组和处理函数，而组又是可以重复的所以只能使用它，而不是map。Event的成员变量：

```
typedef std::multimap<Group, Connection> SlotContainer;
SlotContainer d_slots;    //    处理函数的集合
const String d_name;    //        事件的名称，标识这个事件
```

d_slots就是我们说到的处理函数映射。d_name就是我们说到的事件的名称。

Connection是这样的定义： typedef RefCounted<BoundSlot> Connection，RefCounted是一个引用的封装模板类。原意可能是一个处理函数可以被多个事件引用。但在Event的实现代码里（析构函数）中直接调用了BoundSlot的cleanup函数而不是。

RefCounted的release函数来清理。可见并没有实现一个事件处理函数被多个事件引用。其实也没有这个必要。Event类的subscribe函数注册一个处理函数，unsubscribe反注册一个处理函数。operator()函数调用所有注册在该事件的处理函数。

下面是Event类几个重要函数的实现，其他函数，请读者参考CEGUIEvent.h和CEGUIEvent.cpp两个文件。subscribe函数是给这个事件注册一个处理函数，operator()当这个事件被激发后会被调用，这个函数遍历所有的处理类，并调用该类的处理函数。unsubscribe函数取消一个处理类和这个事件的关联，BoundSlot的disconnect函数会调用它。

```
Event::Connection Event::subscribe(Event::Group group, const Event::Subscriber& slot)
{
   Event::Connection c(new BoundSlot(group, slot, *this));
   d_slots.insert(std::pair<Group, Connection>(group, c));
   return c;
}
void Event::operator()(EventArgs& args)
{
   SlotContainer::iterator iter(d_slots.begin());
   const SlotContainer::const_iterator end_iter(d_slots.end());
   //执行所以注册在这个事件的函数
   for (; iter != end_iter; ++iter)
      args.handled |= (*iter->second->d_subscriber)(args);
}
void Event::unsubscribe(const BoundSlot& slot)
{
   //在事件处理类集合里找到slot并删除之
   SlotContainer::iterator curr =
      std::find_if(d_slots.begin(), d_slots.end(), SubComp(slot));
   // 如果找到，则删除
   if (curr != d_slots.end())
      d_slots.erase(curr);
}
```

unsubscribe函数中用到SubComp类是辅助multimap查找的类。标准std算法库函数std::find_if，需要一个比较是不是相等的函数（因为这个算法函数并不知道如何比较）。

```
class SubComp
{
public:
   SubComp(const BoundSlot& s) :
      d_s(s)
   {}
      //比较函数
   bool operator()(std::pair<Event::Group, Event::Connection> e) const
   {
      return *(e.second) == d_s;
   }
private:
   const BoundSlot& d_s;
};
```

此外为了方便事件处理函数和事件的自动断开，CEGUI提供了一个超出作用域后自动断开和Event的连接的辅助类ScopedConnection。这个类的实现这里就不做介绍了。

到这里我们并没有介绍事件处理函数的参数。事件的参数：基类是EventArgs，它提供了一个bool型变量表示这个事件是否被处理了。还提供了一个虚的析构函数。然后又从这个类派生出许多的不同的类来表示不同的事件所对应的不同的参数。（这不是说不同的事件不能对应相同的参数）。主要的参数有：WindowEventArgs，派生自EventArgs（EventArgs是所有事件参数的

基类，不论是直接基类还是间接基类），窗口事件的基本参数，内有一个窗口的指针，表示那个窗口发出的事件。MouseEventArgs鼠标事件参数，派生自WindowEventArgs，多了几个相关的参数。MouseCursorEventArgs，鼠标相关事件的事件参数。KeyEventArgs，键盘事件的相关参数。ActivationEventArgs，激活窗口事件用的事件参数。DragDropEventArgs，拖动窗口相关事件使用的事件参数。程序可以根据需要自行定义更多的事件参数。

事件集：有了事件的定义，当然需要有个管理所有事件的类，这个类被称作事件集。定义为EventSet类。这个类显然要包含以下功能(1)添加一个事件(2)删除一个事件(3)激发一个事件，其实就是调用事件相应的处理函数(4)添加事件处理函数(5)查找事件。这五种功能对应的函数分别为：

(1).addEvent

(2).removeEvent，removeAllEvents

(3).fireEvent，fireEvent_impl（内部实现函数）

(4).subscribeEvent（2个重载函数），subscribeScriptedEvent（2个重载函数）

(5).getEventObject，isEventPresent

他的成员变量定义如下：

typedef std::map<String, Event*, String::FastLessCompare>　　EventMap;

EventMap　　d_events;

bool d_muted;　　//如果为false则这个事件集合的所有事件不响应

d_muted决定事件是否被响应。d_events是事件集合。

事件的激发由fireEvent实现，它内部调用fireEvent_impl函数，说白了就是调用事件处理函数。如果容许的话（d_mute == false）。下面详细介绍几个重要函数的实现：

第一组事件从事件集添加，删除，存在检查，事件查找。

```
void EventSet::addEvent(const String& name)
{
    if (isEventPresent(name))
    {
        throw AlreadyExistsException("An event named '" + name + "' already exists in the
        EventSet.");
    }
    d_events[name] = new Event(name);
}
void EventSet::removeEvent(const String& name)
{
    EventMap::iterator pos = d_events.find(name);
    if (pos != d_events.end())
    {
        delete pos->second;
        d_events.erase(pos);
    }
}
void EventSet::removeAllEvents(void)
{
    EventMap::iterator pos = d_events.begin();
    EventMap::iterator end = d_events.end()    ;
    for (; pos != end; ++pos)
    {
        delete pos->second;
    }
    d_events.clear();
```

```
}
bool EventSet::isEventPresent(const String& name)
{
    return (d_events.find(name) != d_events.end());
}
Event* EventSet::getEventObject(const String& name, bool autoAdd)
{
  EventMap::iterator pos = d_events.find(name);
  // 如果事件不存在，添加一个新的事件，并返回它
  if (pos == d_events.end())
  {
    if (autoAdd)
    {
      addEvent(name);
      return d_events.find(name)->second;
    }
    else
    {
      return 0;
    }
  }
  return pos->second;
}
```

提示：
CEGUI大量的使用了C++的STL模板库，如果读者对STL不是很熟悉，请参考STL的书籍，否则读者可能无法理解CEGUI的许多实现。如果读者抱着只是希望知道CEGUI的实现原理，也可以不用完全看懂。

第二组 事件处理函数的注册

```
//脚本函数的注册，看到了吧，在这里特殊处理了
Event::Connection EventSet::subscribeScriptedEvent(const String& name, const String& subscriber_name)
{
    ScriptModule* sm = System::getSingletonPtr()->getScriptingModule();
    if (!sm)
    {
      throw InvalidRequestException("[EventSet::subscribeScriptedEvent] No scripting
          module is available");
    }
    return sm->subscribeEvent(this, name, subscriber_name);
}
//另一个注册函数，提供了对组的支持
Event::Connection EventSet::subscribeScriptedEvent(const String& name, Event::Group group, const String& subscriber_name)
{
  ScriptModule* sm = System::getSingletonPtr()->getScriptingModule();
    if (!sm)
    {
      throw InvalidRequestException("[EventSet::subscribeScriptedEvent] No scripting
          module is available");
    }
    return sm->subscribeEvent(this, name, group, subscriber_name);
}
//一般的可以使用统一的接口来注册函数
Event::Connection EventSet::subscribeEvent(const String& name, Event::Subscriber subscriber)
```

```
{
   return getEventObject(name, true)->subscribe(subscriber);
}
//支持组的注册函数
Event::Connection EventSet::subscribeEvent(const String& name, Event::Group group, Event::Subscriber subscriber)
{
   return getEventObject(name, true)->subscribe(group, subscriber);
}
```

第三组事件的激发，事件对应的处理函数被调用

```
void EventSet::fireEvent(const String& name, EventArgs& args, const String& eventNamespace)
{
   // 全局事件的激发，如果有全局事件处理函数，可以注册到全局事件集，
      //它可以监视和处理所有的事件的激发
   GlobalEventSet::getSingleton().fireEvent(name, args, eventNamespace);
   // 本地事件激发的处理
   fireEvent_impl(name, args);
}
//具体事件激发的处理函数
void EventSet::fireEvent_impl(const String& name, EventArgs& args)
{
   // 找到被激发的事件，通过它的名称
   Event* ev = getEventObject(name);

   // 调用事件处理函数，如果这个事件集允许事件被激发(d_muted == false)
   if ((ev != 0) && !d_muted)
      (*ev)(args);
}
```

也许读者现在对事件的架构关系还不是非常清楚，没关系，图2-2详细的描述了事件的架构。

激发事件的处理流程，首先EventSet激发一个事件（一般是在其子类里调用比如Window类）最终会调用fireEvent_impl函数，这个函数找到Event调用它的事件处理函数。事件处理的操作符（）调用最终传达到FreeFunctionSlot和MemberFunctionSlot两种处理类的（）操作符，他们知道如何调用处理函数（回调）。哪么ScriptFunctor是如何被调用的呢？这个CEGUI没有提供直接的支持，需要用户自己处理。最简单的方法是修改ScriptFunctor，使之派生自SlotFunctorBase（也可以不派生，读者思考原因），然后实现ScriptModule的subscribeEvent函数。这个实现可以借助FunctorCopySlot来实现（当然另外两个也可以），将ScriptFunctor作为模板参数，传递给FunctorCopySlot。下面是简单实现的一个示例。

```
Event::Connection  subscribeEvent(EventSet* target, const String& name, const String& subscriber_name)
{
      ScriptFunctor sfuntor(subscriber_name);
      //这一步是调用那个构造函数呢？显然是FunctorCopySlot。
      SubscriberSlot slot(sfuntor);
      //调用普通的注册函数注册到事件集中
      return target->subscribeEvent(name, slot);
}
```

这样定义以后，脚本函数的调用流程和普通函数的调用流程就统一了。

基于事件的类基本上就是这么多。而且他们之间的关系以及作用已经讲的很清楚了。每一个窗口都是基于属性集和事件集的。下一节介绍属性集。

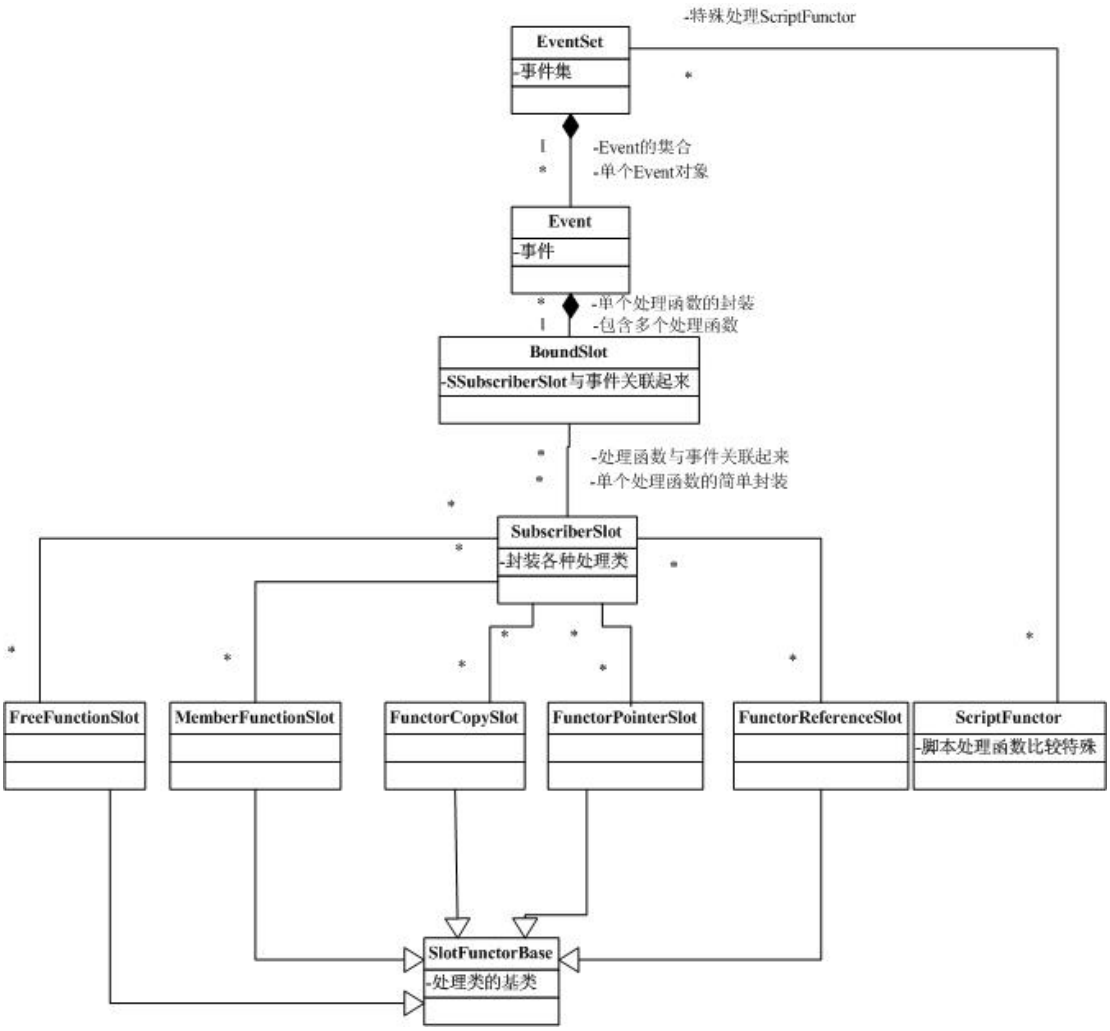

图2-2 事件集的架构

2.2 CEGUI的属性系统

关于为什么要把一些简单的窗口数据信息，比如窗口显示的文字，窗口的高，宽，位置等信息，做的这么复杂呢。在本章开始已经介绍过了。CEGUI中每一个属性都是一个类。而且在CEGUI系统中每个属性类都只有一个实例。读者也许会感到疑惑，每个窗口都有各种属性，为什么整个系统只需要一个属性的实例呢？这和属性的实现有关，我们这里暂不说明。读者带着这个问题，继续。

窗口的属性没有事件哪么复杂，它由一个属性类Property和一个属性集类PropertySet组成。属性类定义了属性的名称，属性的帮助信息，属性的默认值和是否写入到外部流的标志。下面是这四个成员变量的说明。

```
String     d_name;         //   属性的名称
String     d_help;         //   属性的帮助信息
String     d_default;      //   属性的默认值
bool    d_writeXML;  //  属性是否可以被写入到外部的流中的标志
```

属性基类的成员函数，如下所示。最为重要的是get和set函数。

```
const String& getHelp(void) const        {return d_help;}
const String& getName(void) const              {return d_name;}
virtual String   get(const PropertyReceiver* receiver) const = 0;
virtual void     set(PropertyReceiver* receiver, const String& value) = 0;
virtual bool     isDefault(const PropertyReceiver* receiver) const;
```

```
virtual String   getDefault(const PropertyReceiver* receiver) const;
virtual void writeXMLToStream(const PropertyReceiver* receiver, XMLSerializer& xml_stream) const;
```

getHelp获取该属性的帮助信息，getName获取属性的名称，get获取属性值，set设置属性值。isDefault判断是否是属性的默认值，writeXMLToStream将属性写到一个流中，可以非常方便的将属性写入XML文件。可以看到get和set是纯虚函数，也就是说Property是个虚基类无法实例化，必须派生出子类。在CEGUI中所有的属性都是派生自Property类的。

下面还要介绍一个简单的类PropertyReceiver，这个类是PropertySet的基类。可以看出这个类没有任何的成员函数，提供它纯粹是为了方便属性的设置（set）和获取（get）。

```
class CEGUIEXPORT PropertyReceiver
{
public:
    PropertyReceiver() {}
    virtual ~PropertyReceiver() {}
};
```

PropertySet类管理属性，可以添加，删除一个属性。它只有一个成员。那就是一个映射，它映射属性名称String，到属性类的指针（CEGUI系统中的唯一实例）。FastLessCompare比较，map查找属性的时候会用到这个比较函数。

```
typedef std::map<String, Property*, String::FastLessCompare>   PropertyRegistry;
PropertyRegistry    d_properties;
```

这样做有一个空间占用的问题，PropertyRegistry保存了属性的名称，这个名称在所有的属性集里面都要备份，是比较浪费内存的，先前版的CEGUI不是这样实现的。d_properties里面保存了当前属性集中所有的属性，使用属性的名称映射Property的指针。

下面介绍几个重要的函数，首先addProperty向属性集中添加一个属性，我们可以看到是将属性的名称映射到属性的指针。

```
void PropertySet::addProperty(Property* property)
{
    if (!property)
    {
        throw NullObjectException("The given Property object pointer is invalid.");
    }
    if (d_properties.find(property->getName()) != d_properties.end())
    {
        throw AlreadyExistsException("A Property named '" + property->getName() + "'
        already exists in the PropertySet.");
    }
    d_properties[property->getName()] = property;
}
```

其次，获取属性和设置属性的函数。我们看到调用了属性的set和get属性，并且传递this为参数前面我们讲过set和get是以PropertyReceiver的指针为参数的。

```
String PropertySet::getProperty(const String& name) const
{
    PropertyRegistry::const_iterator pos = d_properties.find(name);
    if (pos == d_properties.end())
    {
        throw UnknownObjectException("There is no Property named '" + name + "' available
    in the set.");
    }
    return pos->second->get(this);
}
void PropertySet::setProperty(const String& name,const String& value)
{
    PropertyRegistry::iterator pos = d_properties.find(name);
```

```
    if (pos == d_properties.end())
    {
        throw UnknownObjectException("There is no Property named '" + name + "' available
    in the set.");
    }
    pos->second->set(this, value);
}
```

到这里我们的谜底揭开了，因为属性类并不是包含属性对应的值（数据），具体的数据在Window类或者Window的派生类里面，所以Property类提供了set和get虚函数，在具体的属性类里面通过Window或者Window的派生类的具体函数来设置属性值。比如Window的属性Text，也就是窗口显示的文字。

```
String Text::get(const PropertyReceiver* receiver) const
{
    return static_cast<const Window*>(receiver)->getText().c_str();
}
void Text::set(PropertyReceiver* receiver, const String& value)
{
    static_cast<Window*>(receiver)->setText(value.c_str());
}
```

显而易见，get将receiver转化为Window的指针（为什么转换成Window呢？，因为这是Window的属性，而且Window派生自PropertySet，PropertySet又派生自PropertyReceiver），然后调用Window的getText和setText函数。

最后介绍删除属性的函数，其他函数这里就不介绍了，读者自己阅读源代码。这个函数查找属性映射，然后从属性集中删除。

```
void PropertySet::removeProperty(const String& name)
{
    PropertyRegistry::iterator pos = d_properties.find(name);
    if (pos != d_properties.end())
    {
        d_properties.erase(pos);
    }
}
```

属性集就介绍到这里，下面介绍属性的辅助类PropertyHelper，这个类提供了将属性字符串转换为Window或其派生类所需的值，或者将Window里面的值转化为字符串格式。举两个例子，一个字符串转化为Window需要数据，一个将Window的数据转化为字符串。

```
URect PropertyHelper::stringToURect(const String& str)
{
    using namespace std;
    URect ur;
    sscanf(
        str.c_str(),
        " {{%g,%g},{%g,%g},{%g,%g},{%g,%g}}",
        &ur.d_min.d_x.d_scale, &ur.d_min.d_x.d_offset,
        &ur.d_min.d_y.d_scale, &ur.d_min.d_y.d_offset,
        &ur.d_max.d_x.d_scale, &ur.d_max.d_x.d_offset,
        &ur.d_max.d_y.d_scale, &ur.d_max.d_y.d_offset
    );
    return ur;
}
```

stringToURect，将str里包含的数据转化为窗口需要的URect 。数据的格式是固定的。

```
String PropertyHelper::urectToString(const URect& val)
```

```
{
    using namespace std;
    char buff[512];
    snprintf(buff, sizeof (buff), "{{%g,%g},{%g,%g},{%g,%g},{%g,%g}}",
        val.d_min.d_x.d_scale,val.d_min.d_x.d_offset,
        val.d_min.d_y.d_scale,val.d_min.d_y.d_offset,
        val.d_max.d_x.d_scale,val.d_max.d_x.d_offset,
        val.d_max.d_y.d_scale,val.d_max.d_y.d_offset);
  return String(buff);
}
```

urectToString将val转化为字符串buff。

注意：
属性的字符串格式是固定的，具体格式是由属性确定的，用户可以自己定义一个属性，并且自己设置属性，自己编写属性的转化函数。CEGUI中有许多属性，每种属性都是

属性的架构图如2-3所示，它的结构比较简单，不过涉及到了Window，这个我们下一章重点介绍的类。

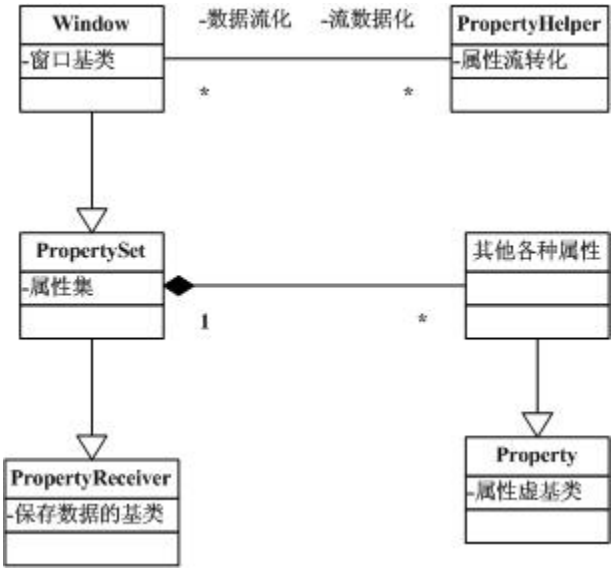

图2-3

用户可以自己定义一个属性，自己设置它的流转化函数。顺便熟悉一些属性的定义，属性和实现。

2.3 属性事件与布局文件

我们前面讲过，CEGUI设计属性和事件有一个原因就是为了方便的在XML文件中设置窗口的数据（属性）和窗口的逻辑操作（事件）。这一节我们详细介绍属性和事件是如何从XML布局文件中获取的。

注意：
CEGUI中的布局文件和其他许多文件并不是以xml为文件扩展名的，布局文件以为扩展名layout，资源管理文件以schema文件为扩展名，控件外观文件是以looknfeel为扩展名的，字体文件是以font为扩展名的等等。虽然不是以xml文件为扩展名，但内部数据是以XML来组织的。所以读者可以修改这些扩展名为xml以便程序打开。目前我们还没有介绍CEGUI的基本类Window，所以CEGUI布局文件中的Window元素读者可能不会非常理解，不过没关系，读者先有个大概的印象，等学习了第3章后在来看这里的介绍会更加清晰。

布局文件的格式如下，首先是XML的标准格式，可以有编码设置，CEGUI所有配置文件都是UTF-8的编码的。其他编码会导致CEGUI抛出异常。

```
<?xml version="1.0" ?>
```

布局文件的第一个元素（或者标记）是GUILayout，所有的其他标记都在这个标记对里。

```
<GUILayout> ...</GUILayout>
```

在下一层就是窗口链了，窗口以Window为标记，窗口中可以嵌套窗口，作为子窗口。

Window是窗口的标记，Type代表窗口的类型，Name代表窗口的名称，这个名称在整个CEGUI系统中都是都必须是唯一的。具体Type的值代表什么，在讲Schema的时候详细介绍。

```
<Window Type="TaharezLook/FrameWindow" Name="Demo8/Window2">
    <Property Name="UnifiedPosition" Value="{{0.55,0},{0.1,0}}" />
    <Property Name="UnifiedSize" Value="{{0.4,0},{0.3,0}}" />
    <Property Name="Text" Value="Demo 8 - Information Window" />
    <Property Name="Tooltip" Value="Contains some StaticText information panels" />
    <Event Name ="MouseClick" Function= "ScriptFunction();"/>
    <Window Type="TaharezLook/StaticText" Name="Demo8/Window2/Info">
     <Property Name="UnifiedPosition" Value="{{0.05,0},{0.1,0}}" />
     <Property Name="UnifiedSize" Value="{{0.9,0},{0.3,0}}" />
     <Property Name="HorzFormatting" Value="WordWrapCentred" />
    </Window>
    <Window Type="TaharezLook/StaticText" Name="Demo8/Window2/Tips">
      <Property Name="UnifiedPosition" Value="{{0.05,0},{0.5,0}}" />
      <Property Name="UnifiedSize" Value="{{0.9,0},{0.4,0}}" />
      <Property Name="HorzFormatting" Value="WordWrapCentred" />
    </Window>
</Window>
```

可以看到，窗口是可以嵌套的，嵌套用来表示内部的窗口是外部窗口的子窗口。好了重点到了Property 标记，它表示一个属性，Name表示它的名称，Value表示它的值。UnifiedPosition是窗口的位置的属性名称，窗口位置为什么不使用两个数（x和y）来表示，而是使用了4个数呢？这个我们稍后在讲。UnifiedSize代表窗口的大小，它也使用了四个数。Text属性代表窗口显示的文字等等。哪么事件呢？事件在的标记是Event，Name表示事件的名称，Function表示事件对应的处理函数。可见布局文件的结构很简单（这都是CEGUI设计的优势）。

布局文件可以修改窗口的数据值，和窗口的处理函数。这给UI系统提供了很大的灵活性。下面介绍布局文件的加载和分析。

注意：
CEGUI所有布局文件里的元素（标记）和属性（XML里的属性的概念，不是CEGUI中的属性，比如Name和Value就是Property的属性）都是大小写敏感的。event不能被CEGUI识别，如果定义了<event Name ="***" Function ="***" />，哪么CEGUI会抛出异常。同理EVENT也不行，只能是Event。Property，Name，Function等等都是一样的。

XML文件的分析是通过CEGUI提供的XML解析库来实现的。解析库读取XML文件并且通过回调函数，通知CEGUI的XML解析接口来处理具体的元素。关于CEGUI的XML解析接口的定义请读者参考第4章的内容。

CEGUI是如何处理XML布局文件的元素的呢？读者会找到GUILayout_xmlHandler.cpp/.h两个文件。从他们的名称可以看出他们是处理布局的。Layout代表布局，xmlHandler代表XML的处理类。类似的还有CEGUIScheme_xmlHandler，CEGUIFont_xmlHandler.h等。每一个CEGUI的描述文件都对应一个类似的处理类。

总的来说，布局文件的处理还是比较复杂的。这里介绍一种阅读代码规律，希望可以帮助读者理解。

XML中出现的标记有两类一类我们称为元素，它主要有以下6种。

```
const String GUILayout_xmlHandler::GUILayoutElement( "GUILayout" );
const String GUILayout_xmlHandler::WindowElement( "Window" );
const String GUILayout_xmlHandler::AutoWindowElement( "AutoWindow" );
const String GUILayout_xmlHandler::PropertyElement( "Property" );
const String GUILayout_xmlHandler::LayoutImportElement( "LayoutImport" );
const String GUILayout_xmlHandler::EventElement( "Event" );
```

他们的名称中带有Element，前缀代表他们具体的含义。比如WindowElement，它代表XML代码里的Window标记。当XML解析库遇到Element的时候它会调用我们编写的处理类中的elementStart函数。什么是Element标记呢？以Window举例，<Window 就是一个Element的开始，这是elementStart会被调用。当XML解析库遇到 </Window>（这里只是那Window来举例，所有的元素都可以这样结束，但为了简单，好多元素使用了简化版）或者它的简化版/>，属性和事件等元素使用简化版。比如属性定义<Property Name="UnifiedSize" Value="{{0.4,0},{0.3,0}}" />，就使用/>来表示元素的结束。当一个元素结束的时候，XML解析库会调用elementEnd函数，这时可以做一些处理工作。如果读者对XML解析库如何回调这两个函数感兴趣可以参考第4章相关内容。

GUILayoutElement是布局文件的根元素，它有LayoutParentAttribute属性用来表示这个布局文件的命名父窗口。WindowElement代表窗口元素，CEGUI遇到这个元素会创建一个窗口作为当前窗口（上一个WindowElement对应的窗口，处理这个窗口后，当前窗口变成这个窗口）的子窗口。AutoWindowElement是这一版的新元素，用来代表窗口的自动孩子，关于什么是自动孩子，后文讲述。PropertyElement元素设置当前窗口的属性，EventElement元素设置当前窗口的事件。LayoutImportElement也是一个新元素，它就像C++ 中的#include，包含一个布局文件到当前布局文件，布局文件返回的窗口，作为当前窗口的子窗口。也可以把它想象成一个定义在另一个文件中的复杂的WindowElement元素。

另一类，我们称作属性（XML中的定义）。他们可以作为元素的属性存在具体如下所示。

```
const String GUILayout_xmlHandler::WindowTypeAttribute( "Type" );
const String GUILayout_xmlHandler::WindowNameAttribute( "Name" );
const String GUILayout_xmlHandler::AutoWindowNameSuffixAttribute( "NameSuffix" );
const String GUILayout_xmlHandler::PropertyNameAttribute( "Name" );
const String GUILayout_xmlHandler::PropertyValueAttribute( "Value" );
const String GUILayout_xmlHandler::LayoutParentAttribute( "Parent" );
const String GUILayout_xmlHandler::LayoutImportFilenameAttribute( "Filename" );
const String GUILayout_xmlHandler::LayoutImportPrefixAttribute( "Prefix" );
const String GUILayout_xmlHandler::LayoutImportResourceGroupAttribute( "ResourceGroup" );
const String GUILayout_xmlHandler::EventNameAttribute( "Name" );
const String GUILayout_xmlHandler::EventFunctionAttribute( "Function" );
```

具体每个属性是什么含义，也可以根据他们的名称来判断，比如EventFunction代表这是事件的函数属性，它包含Attribute来表示它是一个属性。读者也许奇怪为什么EventNameAttribute，WindowNameAttribute以及PropertyNameAttribute都是"Name"，为什么要分成三个呢？主要是方便读者阅读后续的处理代码。他们分别代表不同元素中的名字属性。

WindowTypeAttribute属性代表窗口类型，CEGUI使用窗口类型和窗口名称来创建窗口，其中名称是可选的（CEGUI内部生成唯一的窗口名称来表示这个窗口）。AutoWindowNameSuffixAttribute属性代表窗口的自动子窗口的后缀名。CEGUI自动子窗口的名称是由父窗口的名字+子窗口的后缀构成的。PropertyValueAttribute属性代表属性的值字符串，这个值一般就是我们属性帮助函数设置属性时的参数。stringToURect的参数就是UnifiedAreaRect窗口属性的PropertyValueAttribute值。LayoutImportFilenameAttribute，LayoutImportPrefixAttribute，LayoutImportResourceGroupAttribute是LayoutImportElement的属性，它们提供LayoutImportElement需要的信息，包括导入布局文件的名称，后缀名称，以及资源组。这些属性涉及到了CEGUI的资源提供模块。我们在第4章详细介绍。EventFunctionAttribute属性是EventElement元素对应的属性。它的值就是脚本函数或者脚本片段。

刚才我们已经讲过，当一个元素到来时会调用elementStart，哪么下面来分析这个函数的实现。

```
void GUILayout_xmlHandler::elementStart(const String& element, const XMLAttributes& attributes)
{
   // 处理根元素 GUILayoutElement ， 调用函数elementGUILayoutStart来处理
   if (element == GUILayoutElement)
   {
      elementGUILayoutStart(attributes);
   }
```

```
    //处理Window元素
    else if (element == WindowElement)
    {
        elementWindowStart(attributes);
    }
    //处理自动窗口元素，从当前窗口获取自动孩子
    else if (element == AutoWindowElement)
    {
        elementAutoWindowStart(attributes);
    }
    //处理属性元素，设置当前窗口的属性
    else if (element == PropertyElement)
    {
        elementPropertyStart(attributes);
    }
    // 处理导入布局文件元素，当前窗口接受布局文件返回的窗口为子窗口
    else if (element == LayoutImportElement)
    {
        elementLayoutImportStart(attributes);
    }
    // 处理脚本元素，又称事件元素
    else if (element == EventElement)
    {
        elementEventStart(attributes);
    }
    // 其他未知元素，只写入CEGUI错误日志
    else
    {
        Logger::getSingleton().logEvent("GUILayout_xmlHandler::startElement - Unexpected
        data was found while parsing the gui-layout file: '" + element + "' is unknown.", Errors);
    }
}
```

我们看到CEGUI根据不同元素，都有单独的处理函数，从这个函数无法获得详细信息，下面我们以窗口元素为例，介绍窗口元素的处理过程。在介绍elementWindowStart函数之前必须先介绍XMLAttributes，它作为每个处理函数的参数。这个参数代表我们前面描述的属性集，就是它包含了当前元素的所有的属性。

```
void GUILayout_xmlHandler::elementWindowStart(const XMLAttributes& attributes)
{
    //获取窗口的类型属性
    String windowType(attributes.getValueAsString(WindowTypeAttribute));
    // 获取窗口的名称属性，创建窗口的两个参数
    String windowName(attributes.getValueAsString(WindowNameAttribute));
    // 尝试创建窗口，这个过程会抛出异常
    try
    {
        Window* wnd = WindowManager::getSingleton().createWindow(windowType,
            windowName, d_namingPrefix);
        // 添加这个窗口为当前窗口的子窗口
        if (!d_stack.empty())
            d_stack.back().first->addChildWindow(wnd);
        else
            d_root = wnd;
```

```
        //设置自己为当前窗口
        d_stack.push_back(WindowStackEntry(wnd,true));
        // 通知当前窗口初始化
        wnd->beginInitialisation();
    }
    catch (AlreadyExistsException&)
    {
        // 创建的窗口已经存在，名字重复，清除所有的创建过的窗口
        cleanupLoadedWindows();
        // 抛出异常，告诉调用层，有错误发生了
        throw InvalidRequestException("GUILayout_xmlHandler::startElement - layout loading
        has been aborted since Window named '" + windowName + "' already exists.");
    }
    catch (UnknownObjectException&)
    {
        // 清除所有已经创建的窗口
        cleanupLoadedWindows();
        // 抛出异常，告诉调用层，有错误发生了
        throw InvalidRequestException("GUILayout_xmlHandler::startElement - layout loading
        has been aborted since no WindowFactory is available for '" + windowType + "'
                objects.");
    }
}
```

我们看到遇到WindowElement首先创建一个窗口，然后将窗口作为当前窗口的子窗口，最后设置自己为当前窗口。这样如果窗口元素嵌套就可以自动设置父子关系了。读者对窗口和窗口的创建还不熟悉，别着急第3章将会详细介绍。

其他元素的处理类似，读者自己阅读代码，这里不在赘述。CEGUI处理所有支持的描述文件的读取和处理方法都是一样的。

2.4 本章小结

这一章主要介绍了属性和事件集的架构以及相关的布局文件的介绍。

1.读者自己实现一个事件的处理函数类，并且使用我们说的第二类封装类，封装这个类，使这个类可以被Event注册为事件处理函数（回调函数）。

2.用户自己实现一个CEGUI的属性，给窗口类加一个是否允许调整大小的属性SizeAble。属性的值是bool型的，用户可以尝试的实现，应该不难。

第3章 CEGUI基类的实现

这一章介绍CEGUI最基本的窗口类Window。CEGUI中所有的窗口都必须派生自Window（不管直接还是间接）。在介绍Window类之前，我认为介绍窗口原理比较重要。所以第1节介绍窗口的原理。

3.1 窗口设计原理

游戏中的窗口系统，没有Window窗口系统那么复杂。但他们的结构是类似的。窗口从外表上看是由一些图片的累加而成。从行为上看，它可以接受用户的输入，并做出合适的响应。从结构上看是典型的树状层次结构（窗口有父窗口，子窗口，同级的兄弟窗口）。下面就具体根据这几个方面，详细阐述。

第一，窗口的画面如何组织。CEGUI提供了LookNFeel文件来描述窗口的布局，详细介绍请参考第6章。CEGUI窗口最多可以分割成9个部分，每一部分都有独立的贴图（并不意味着是一张图片，可能是图片的一部分），为什么不用一张贴图呢？不是不可以，用一张贴图就需要美工把贴图做的像个窗口的样子。这样做扩展性差，灵活性也不够。最重要的是不支持窗口的缩放，因为一旦缩放窗口贴图就会被拉伸，图像变的模糊。所以我们在设计外观文件的时候要注意，如果窗口的大小不是固定的一定要使用支持大小变动的分割方法。支持大小变动的格式大概有三种，如图3-1所示一种是支持各向拉升的自适应窗口，它需要把窗口分割成九块。需要注意的是这九块中只有中间的一块必须是支持各向拉伸都不会产生马赛克的。就是说它是背景图片，图片样子单一任何方向上都可以任意扩展。最简单的

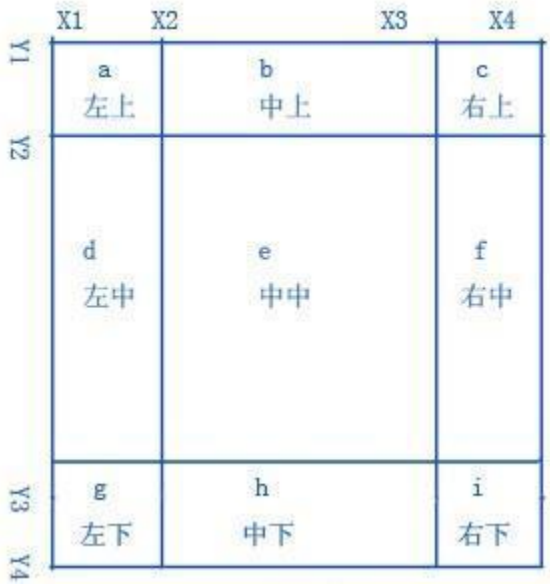

图3-1

这种图片的例子就是单色背景图片了。这种情况还要求b和h是水平方向可拉伸的。d和f是竖直方向可以拉伸的。第二种是水平方向可以拉伸的，这种布局要求中间的部分是可以水平方向拉伸的。这种布局只有三部分左中右。第三种是竖直方向可拉伸的，它要求中间部分是竖直方向可拉伸的。这种布局也只有三部分上中下。后两种不能在各个方向上拉伸，但是也很有用。最普通的一种布局就是一张图片作为窗口的贴图了，这种布局一般不支持拉伸（除非它是第一种布局中的中间图片）。

第二，窗口的用户响应。窗口需要接收用户的输入，以及Window消息输入（主要是IME消息，输入法消息），并且要响应这些输入。CEGUI通过System类来接收这些消息，并且找到处理这些消息的窗口，处理这些消息。除了这些外部消息外，窗口也会产生内部消息，这些消息多是子窗口通知父窗口的通知消息，和父窗口通知子窗口的消息。比如说按钮子窗口接收到了鼠标单击消息，他要通知父窗口，用户鼠标单击。父窗口不能接收到用户消息（因为用户的消息，只能给最上层的窗口处理，父窗口一般不是最上层窗口，一般是子窗口处理消息，然后在通知父窗口）。CEGUI中通过事件机制来发布子窗口的通知消息，如果父窗口希望处理子窗口的通知消息，可以注册子窗口的对应消息的处理函数。当然有时候父窗口也能是最上层窗口，如果鼠标不再所有的子窗口上面，而在父窗口上。比如用户调整窗口大小的时候。父窗口要通知所有字窗口，以便子窗口调整自己

的大小。这一步不是使用事件驱动，而是父窗口直接调用子窗口的事件响应函数，这种函数一般以on开头，比如OnParentSizeChange。

第三，窗口的树状层次结构。窗口显然有父子关系，比如说框架窗口有按钮子窗口，编辑框子窗口等等。这些子窗口是属于父窗口的，他们的位置信息是相对于父窗口来说的。他们是在父窗口的贴图上层覆盖上自己的贴图。当然也有兄弟关系，比如上面的按钮子窗口和编辑框子窗口。他们没有父子关系，但他们有共同的父窗口，所以他们是兄弟窗口的关系。由于窗口有上面的复杂关系所以树形数据结构最为适合描述。CEGUI中使用STL中的vector容器保存窗口所有的子窗口，没有使用常用的链表结构。CEGUI中大量使用了STL容器，这样做似乎会浪费一些内存，但现在内存非常便宜，容量也非常大，所以也不是什么太大的问题了。

3.2 Window类

Window类作为CEGUI的所有窗口的基本类，它在CEGUI中占有重要的地位。它的设计非常复杂，这也给清晰的描述它带来的困难。本书打算分以下几个小节来描述Window类：

第一，Window类的继承关系以及与其相关的函数。第二，窗口的组织结构。第三，窗口位置和大小。第四，窗口渲染。第五，事件处理。第六，窗口状态。第七，窗口与输入系统。第八，其他部分。Window类成员函数和成员变量非常多，所以我们采用分为七个部分的方法详细介绍。

3.2.1 Window类的继承关系以及与其相关的函数。

Window类继承自PropertySet和 EventSet。这两个类第2章已经介绍，分别处理窗口的属性和窗口的事件。这部分在Window类中没有显示的定义任何函数和成员变量。但由于继承的关系，所以Window类可以设置获取窗口的属性。注册和反注册事件处理函数。如果需要注册事件处理函数可以直接使用窗口指针作为EventSet的指针来注册函数，在第2.3节介绍的布局文件就是这样处理的。如果需要设置或者获取窗口的某个属性的值也可以直接使用窗口的指针来调用属性集的对应函数。

3.2.2 窗口的组织结构

简单的说这部分介绍窗口的组织结构相关的函数。这部分涉及到的成员变量有如下几个：

```
typedef std::vector<Window*> ChildList;
//子窗口列表，使用vector来保存子窗口的指针
ChildList d_children;
//保存当前的渲染顺序，用来组织渲染
ChildList d_drawList;
//静态变量，保存当前捕获输入系统输入的窗口（这个和Window系统捕获窗口是不一样的）
static Window* d_captureWindow;
//先前捕获输入的窗口，用来还原旧的捕获模式
Window* d_oldCapture;
//父窗口的指针
Window* d_parent;
```

这部分处理窗口之间的关系。处理窗口直接关系的函数有可以分为两种，处理孩子关系的函数和处理父窗口的函数。

首先，介绍处理孩子的函数。主要有获取一个子窗口和判断一个窗口是不是这个窗口的子窗口两类函数。isChild用来判断是否是窗口的子窗口。它有三个函数分别接受不同的参数，一个接收窗口名称，一个接收窗口ID，窗口ID在父窗口的所有子窗口之间是唯一的，最后一个接收Window的指针。下面是其中一个函数，可以看到它遍历所有子窗口查找是否有名称和给定名称相同的子窗口，如果有自返回true否则返回false。getChildCount用来返回子窗口的个数，它等于d_children中有效元素的个数。

```
bool Window::isChild(const String& name) const
{
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        if (d_children[i]->getName() == name)
```

```
            {
                return true;
            }
        }
        return false;
}
```

提示：
我们只介绍相同功能，但参数不同的函数其中之一，其他函数类似，读者可以自己阅读。

下面这个函数和isChild类似，但它递归查找所有的子窗口。即也查找窗口的间接子窗口。

```
bool Window::isChildRecursive(uint ID) const
{
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        if (d_children[i]->getID() == ID || d_children[i]->isChildRecursive(ID))
        {
            return true;
        }
    }
    return false;
}
```

第二类函数是获取子窗口的函数getChild，这个函数接受两种参数，一种是窗口名称，另一种是子窗口ID。

```
Window* Window::getChild(uint ID) const
{
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        if (d_children[i]->getID() == ID)
        {
            return d_children[i];
        }
    }
    // 抛出未知对象的异常，因为直接子窗口中没有ID对应的窗口
    char strbuf[16];
    sprintf(strbuf, "%X", ID);
    throw UnknownObjectException("Window::getChild - The Window with ID: '" +
    std::string(strbuf) + "' is not attached to Window '" + d_name + "'.");
}
```

获取间接子窗口的函数，也有两种参数的重载函数。

```
//获取间接子窗口，返回子窗口的指针
Window* Window::recursiveChildSearch( const String& name ) const
{
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        String childName = d_children[i]->getName();
        //判断是否窗口名称或者前缀加上窗口名称与name相等
        if(childName == name || childName == d_windowPrefix + name)
        {
            return d_children[i];
```

```
            }
        }
        //递归查找子窗口
        for(size_t i=0;i<child_count;i++)
        {
            Window* temp = d_children[i]->recursiveChildSearch(name);
            if(temp)
                return temp;
        }
        return 0;
}
```

获取当前激活的窗口，这个窗口是最顶层的激活窗口，也就是激活的最上层的子窗口。激活的概念读者应该很熟悉，激活窗口接收当前的用户输入。这个函数唯一的调用类就是System类。

```
const Window* Window::getActiveChild(void) const
{
    // 如果自己没有被激活则子窗口不可能被激活
    if (!isActive())
    {
        return 0;
    }
    size_t pos = getChildCount();
    while (pos-- > 0)
    {
        //如果子窗口被激活则递归返回被激活的子窗口，目的是找到最上层的子窗口
        if (d_drawList[pos]->d_active)
            return d_drawList[pos]->getActiveChild();
    }
    // 没有子窗口被激活的话，返回自己，此时自己就是最上层的激活窗口
    return this;
}
```

其次，介绍父窗口相关的处理函数。父窗口的处理函数主要有，判断一个窗口是否是这个窗口的祖先，获取这个窗口的子窗口，添加和删除子窗口。

isAncestor用来判断是否name对应的窗口是这个窗口的祖先。它还有两个参数不同的重载窗口。

```
bool Window::isAncestor(const String& name) const
{
    // 如果没有父窗口，显然不用判断，直接返回false
    if (!d_parent)
    {
        return false;
    }
    // 检查直接父窗口
    if (d_parent->getName() == name)
    {
        return true;
    }
    // 不是直接父窗口，则递归到父窗口的父窗口
    return d_parent->isAncestor(name);
}
```

获取一个窗口的父窗口，非常简单的返回d_parent变量。

```
Window* getParent(void) const   {return d_parent;}
```

添加一个子窗口，这个函数还有一个带窗口名称的重载函数。

```
void Window::addChildWindow(Window* window)
{
    // 有效性检查，不能添加自己为自己的孩子，不能添加空指针
    if (window == this || window == 0)
    {
        return;
    }
    addChild_impl(window);
    //激发事件
  WindowEventArgs args(window);
    onChildAdded(args);
    //通知窗口Z值改变，窗口的Z值影响窗口的描述顺序
    window->onZChange_impl();
}
```

添加子窗口的具体实现。

```
void Window::addChild_impl(Window* wnd)
{
    // 如果窗口先前有父窗口则先断开与先前窗口的联系
    if (wnd->getParent())
        wnd->getParent()->removeChildWindow(wnd);
    //添加窗口到描绘列表中
  addWindowToDrawList(*wnd);
  // 添加子窗口到列表中
  d_children.push_back(wnd);
    // 设置父窗口
    wnd->setParent(this);
    // 强制更新子窗口区域，因为父窗口改变了
  WindowEventArgs args(this);
    wnd->onParentSized(args);
}
```

有添加必然有删除一个子窗口了，但这里就不做介绍了。

提示：
伴随窗口的一些操作会激发一些事件，有提供给外部使用的比如调用事件集的fireEvent函数。也有内部使用的事件以on开头的一些事件处理函数，当然这些函数也可以激发外部事件，如果需要。总的来说，如果设计窗口的功能的人认为有必要通知外部或者内部一些事件就可以激发需要的事件。

3.2.3 窗口位置和大小

提到位置和窗口，这里要做简单的介绍。CEGUI中位置和大小使用了比较特殊的方法来表示。一般来说窗口的大小和位置使用绝对坐标和相对坐标两者之一。绝对坐标使用的表示数据是以像素为单位的。比如说位置坐标x=10，y=4，表示相对于父窗口在x方向偏移10个像素单位，在y方向上偏移4个单位。相对坐标使用的数据表示的是相对于父窗口的百分比。比如说x=0.2，y=0.3，表示x方向偏移现在父窗口宽度的百分之20，也就是乘以0.2，y方向上偏移父窗口高度的百分之30，也就是乘以0.3。CEGUI采用的是两者的结合，使用相对坐标加偏移来表示位置或者大小。CEGUI称之为统一坐标系统。一个位置信息使用两个数来表示，一个是相对坐标，两个是绝对坐标（偏移量），就是说x = scale*getWidth（） + offset。（这里拿x坐标来举例）如果scale为0且offset不为0则统一坐标变成了绝对坐标系统，如果offset为0且scale不为0则表示相对坐标系统。

提示：
最终用于渲染的坐标必须是绝对坐标，渲染系统只支持绝对坐标系统（而且这个坐标系统是渲染窗口的坐标系统，或者说屏幕坐标系统）。统一坐标转换为绝对坐标的公式就是R = scale*A + offset。其中A可以是高度或者宽度，scale代表相对坐标，

offset是绝对偏移所以不用做转换直接与相对坐标转换成绝对坐标后相加。最终结果R为绝对坐标。在CEGUI中子窗口的永远在父窗口形成的坐标系下，也就是说子窗口的坐标系的原点永远是父窗口的位置。子窗口的坐标数据是相对于父窗口的。比如说子窗口位置x=10，y=5这个10和5表示从父窗口的位置（父窗口的位置是子窗口的原点（0，0））开始偏移10和5个单位。CEGUI中窗口的位置全部使用父窗口坐标系，没有使用过渲染窗口（或者叫做区域，这是操作系统提供的窗口区域）的坐标系，除非窗口没有父窗口。如读者还是不明白各种坐标以及坐标系之间的关系，请参考图3-2。

图3-2描述了父窗口和子窗口坐标之间的关系。从图中可以看到子窗口使用父窗口形成的坐标系来描述自己的位置（O‘的位置是父窗口坐标系下的坐标）。同时自己又形成了一个坐标系，这个坐标系是以自己的位置（O’）为原点的，X，Y轴方向和父窗口相同。子窗口还可以有自己的子窗口（设为B），B使用子窗口形成的坐标系。也就是说在设置窗口的位置的时候，读者需要考虑的是这个位置使用的是父窗口坐标系。窗口的大小和坐标系是独立的，直接设置它就可以了。其实Window的GUI系统也是这样处理子窗口和父窗口之间的坐标关系的。

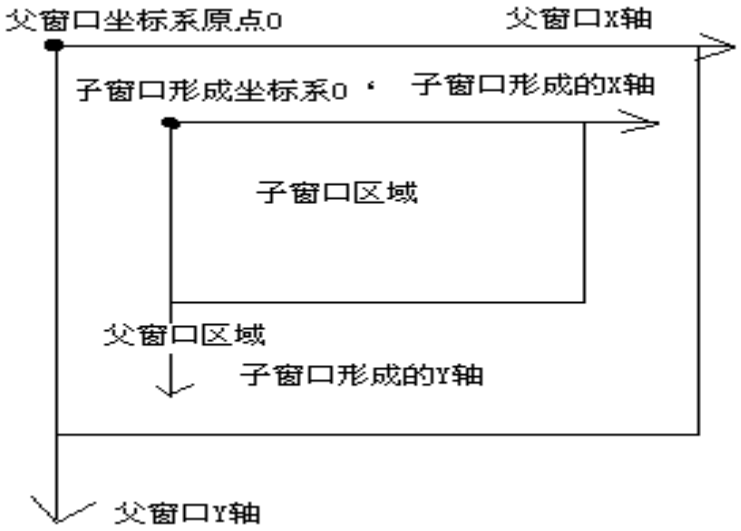

图3-2 坐标系统关系

提示：
没有父窗口的窗口使用那个坐标系呢？它使用Windows应用程序（这里以WindowGUI系统为例，ＣＥＧＵＩ支持Ｌｉｎｕｘ，Ｍａｃ系统）渲染窗口客户区坐标系。Windows应用程序使用屏幕坐标系。显卡驱动程序也使用屏幕坐标系，所以要描述一个CEGUI窗口必须使用屏幕坐标系的位置坐标。

位置和大小涉及到的成员变量如下。

```
//保存窗口的矩形区域，使用统一坐标表示
URect d_area;
//当前窗口的大小，以像素为单位
Size d_pixelSize;
//d_screenUnclippedRect表示窗口在屏幕空间的没有裁剪前的坐d_screenUnclippedRectValid，//表示d_screenUnclippedRect是否有效，从他的名字也可以看出
mutable Rect d_screenUnclippedRect;
mutable bool d_screenUnclippedRectValid;
//表示窗口排除窗口的某些部分的未裁剪矩形，类似WindowsGUI中的客户区区域
mutable Rect d_screenUnclippedInnerRect;
mutable bool d_screenUnclippedInnerRectValid;
//d_screenUnclippedRect的屏幕裁剪后的区域
mutable Rect d_screenRect;
mutable bool d_screenRectValid;
//d_screenUnclippedInnerRect的屏幕裁剪后的区域
```

```
mutable Rect d_screenInnerRect;
mutable bool d_screenInnerRectValid;
//窗口的最小尺寸
UVector2 d_minSize;
//窗口的最大尺寸
UVector2 d_maxSize;
```

WindowGUI的客户区域指的是那一部分区域呢？比如打开记事本程序，客户区如图3-3的黑色部分所示：

图3-3 记事本的客户区

注意：
这里的屏幕指的是Window 应用程序（游戏客户端）客户区区域，而不是电脑的屏幕。

URect 和Size 都是CEGUI定义的类，用来描述窗口的位置矩形和以像素为单位的大小信息，也就是高和宽。前面我们介绍过，CEGUI使用统一坐标，使用两个数表示一个X或者Y坐标，普通的相对坐标和绝对坐标都使用一个数（一般是浮点型的）表示，CEGUI使用UDim类来描述这个值。

这个类有两个成员变量d_scale代表相对量，d_offset代表绝对量。这个类可以提供绝对坐标和相对坐标。通过函数Absolute和asRelative。也就是说统一坐标只是提供了一种全新的计算方法，它并不能直接被渲染系统接受，需要转化为绝对坐标。

```
float d_scale, d_offset;
```

他们的定义如下，可以看出转化为绝对坐标时，将相对分量乘以base（一般为窗口的高或者宽）转化为绝对坐标，然后在加上绝对偏移（它自己已经是绝对坐标了）。转化为相对坐标时，绝对分量处以base然后加上相对偏移。

```
float asAbsolute(float base) const    { return PixelAligned(base * d_scale) + d_offset; }
float asRelative(float base) const    { return (base != 0.0f) ? d_offset / base + d_scale : 0.0f; }
```

UDim类定义了一些操作符，比如加减乘除等一般是绝对部分和绝对部分做相应操作，相对部分和相对部分做操作，这里就不在介绍了。

UDim只能代表一个X或者Y轴上的坐标，所以两个UDim变量才能代表一个二维点。在CEGUI中二维点使用UVector2类来表示。很显然它有两个成员变量。分别代表X轴和Y轴的坐标值。

```
UDim d_x, d_y;
```

这个类的其他成员函数也很好理解这里就不在列出了。

哪么要定义一个矩形，我们知道需要两个点，分别是左上角和右下角。CEGUI定义URect来表示一个矩形区域。它有两个成员变量分别代表左上角和右下角的点。

```
UVector2 d_min, d_max;
```

d_min代表左上角，d_max代表右下角。其实有这个矩形已经知道窗口的大小了哪么何必有个成员变量d_pixelSize来表示窗口的大小呢？我个人认为主要是方便使用。我们知道窗口的大小是使用非常频繁的，如果每次都计算显然不太方便，还不如设置一个变量保存他。

Size类只有两个变量一个表示宽度，一个表示高度。它的成员函数只有==和！=两个操作符。

```
float d_width, d_height;
```

位置相关函数中，最重要的就是setArea_impl，它实现了改变窗口位置和大小等重要的操作，其他的函数比如setArea的几个重载函数都是调用它来实现的。

```
void Window::setArea_impl(const UVector2& pos, const UVector2& size, bool topLeftSizing, bool fireEvents)
{
   // 设置这些变量的无效标志，在相应获取变量时会重新计算
   d_screenUnclippedRectValid = false;
   d_screenUnclippedInnerRectValid = false;
   d_screenRectValid = false;
   d_screenInnerRectValid = false;
   // 标记变量，用来标识是否窗口移动或者大小改变了
   bool moved = false, sized;
   // 保存旧的窗口大小，后边要使用这个值
   Size oldSize(d_pixelSize);
   // 计算当前的最大和最小尺寸，来限制我们窗口大小
   Vector2 absMax(d_maxSize.asAbsolute(System::getSingleton().getRenderer()->getSize()));
   Vector2 absMin(d_minSize.asAbsolute(System::getSingleton().getRenderer()->getSize()));
      //这一步是什么原理，获取父窗口的大小来计算自己的大小，相对值是相对于父窗口的
//或者说子窗口使用的是父窗口形成的坐标系
   d_pixelSize = size.asAbsolute(getParentPixelSize()).asSize();
   // 限制我们计算出的大小不超过最大和最小的限制
   if (d_pixelSize.d_width < absMin.d_x)
      d_pixelSize.d_width = absMin.d_x;
   else if (d_pixelSize.d_width > absMax.d_x)
      d_pixelSize.d_width = absMax.d_x;
   if (d_pixelSize.d_height < absMin.d_y)
      d_pixelSize.d_height = absMin.d_y;
   else if (d_pixelSize.d_height > absMax.d_y)
      d_pixelSize.d_height = absMax.d_y;
      //设置区域的大小
   d_area.setSize(size);
   sized = (d_pixelSize != oldSize);
      //修改位置信息
   if (!topLeftSizing || sized)
   {
      // 如果发生位置改变，则修改位置信息
      if (pos != d_area.d_min)
      {
         d_area.setPosition(pos);
         moved = true;
      }
   }
   // 如果需要则激发事件
   if (fireEvents)
   {
      WindowEventArgs args(this);
```

```
        if (moved)
        {
            onMoved(args);
            // 重设置这个句柄为false方便下面OnSize的调用
            args.handled = false;
        }
        if (sized)
        {
            onSized(args);
        }
    }
}
```

读者可能不能理解当前的最大和最小尺寸限制窗口的含义以及system::getSingleton().getRenderer()->getSize()获取的是什么内容。Window下有个消息WM_GETMINMAXINFO，用来获取窗口的最大和最小值。我们这里的absMax和absMin代表窗口的最大和最下值。窗口的大小不能超过最大值，也不能小于最小值。system::getSingleton().getRenderer()->getSize()，获取渲染窗口的尺寸（一般来说就是游戏客户区的大小）。absMax和absMin的默认值分别是（1.0f，1.0f）和（0.0f和0.0f），也就是说最大是渲染窗口的尺寸，最小值是宽高都是0。setArea设置整个区域包括位置和大小，它共有三个重载函数。setPosition，setXPosition，setYPosition单独设置窗口的位置。setSize设置大小，setWidth设置宽度，setHeight设置高度。setMaxSize，setMinSize设置最大和最下尺寸。这些设置函数还有对应的get函数。

这里还要介绍两个重要的函数onSized和onMoved，分别响应尺寸变化和位置变化。

```
void Window::onSized(WindowEventArgs& e)
{
    //通知所有孩子父窗口尺发生变化
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        WindowEventArgs args(this);
        d_children[i]->onParentSized(args);
    }
    //通知子窗口重新布局，其实最终还是设置子窗口区域（Area）信息
    performChildWindowLayout();
    //请求重画，具体如何描绘请读者参考第4章
    requestRedraw();
    //激发事件，调用事件处理函数EventNamespace用来区分事件来自那类窗口
    fireEvent(EventSized, e, EventNamespace);
}
//窗口移动是被setArea_impl调用
void Window::onMoved(WindowEventArgs& e)
{
  //通知子窗口父窗口已经被移动了
    const size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        d_children[i]->notifyScreenAreaChanged();
    }
  // 因为移动位置不需要全部重画，所以只要重新提交，不需要全部重画
  System::getSingleton().signalRedraw();
    //激发事件
    fireEvent(EventMoved, e, EventNamespace);
```

```
}
```

除了以上介绍的一些位置和尺寸相关的函数外，还有一类比较重要的函数。就是获取窗口矩形的函数，这些矩形不在父窗口坐标系下的结果，而是在屏幕坐标系下的结果。他们的主要目的是方便渲染接口使用。第一个getPixelRect和getPixelRect_impl。前者是调用后者实现的。

```
Rect Window::getPixelRect(void) const
{
    //如果先前计算的值无效了，重新计算
  if (!d_screenRectValid)
  {
    d_screenRect = (d_windowRenderer != 0) ?
      d_windowRenderer->getPixelRect()
      : getPixelRect_impl();
    d_screenRectValid = true;
  }
    //否则直接返回保存的值
  return d_screenRect;
}

Rect Window::getPixelRect_impl(void) const
{
    // 判断是否被父窗口裁剪，getIntersection获取两个矩形的交集
    if (isClippedByParent() && (d_parent != 0))
    {
        return getUnclippedPixelRect().getIntersection(d_parent->getInnerRect());
    }
    // 如果不被父窗口裁剪则被屏幕裁剪
    else
    {
        return getUnclippedPixelRect().getIntersection(System::getSingleton().getRenderer()->g
    etRect());
    }

}
```

读者可能还是不理解这两个函数在做什么？没关系随着本书的深入讲解，读者会陆续理解的。这两个函数返回的是在屏幕空间的窗口矩形。getArea函数返回的是在统一坐标结果，而且不是在屏幕坐标系下是在父窗口的局部坐标系下。下面介绍getUnclippedPixelRect，这个函数也是返回屏幕坐标下的窗口区域，但是它是没有被裁剪过的，getPixelRect返回的是被裁剪过的区域。

```
Rect Window::getUnclippedPixelRect(void) const
{
  if (!d_screenUnclippedRectValid)
  {
    Rect localArea(0, 0, d_pixelSize.d_width, d_pixelSize.d_height);
    d_screenUnclippedRect = CoordConverter::windowToScreen(*this, localArea);
    d_screenUnclippedRectValid = true;
  }
  return d_screenUnclippedRect;
}
```

这个函数首先获取屏幕的局部坐标localArea，然后通过坐标转换函数windowToScreen将局部坐标转换为在屏幕空间的坐标。这个函数的实现是通过窗口的父链递归累加，最终到屏幕空间的，有兴趣的读者可以看具体代码。

除了刚才讲的两个函数外还有另外两个函数getUnclippedInnerRect和getInnerRect，由于篇幅限制这里就不做介绍了。他们也是获取在屏幕空间的区域，不过获取的是内部区域，一般用于框架窗口。效果就像Window GUI中获取窗口的客户区域一样，如图3-3所示。

3.2.4窗口渲染

窗口的渲染比较复杂，涉及到CEGUI的渲染机制和渲染模块。所以这里这是简单的介绍Window类中与渲染相关的函数和成员变量，具体渲染的原理和流程在第4章详细介绍。render函数是窗口渲染入口函数。

```
void Window::render(void)
{
    // 如果窗口被隐藏则不做任何处理，隐藏窗口不可见
    if (!isVisible()) {
        return;
    }
    // 激发渲染开始的事件，这个我觉得没有必要，用户可以根据需要选择是否激发
    WindowEventArgs args(this);
    onRenderingStarted(args);
    // 渲染这个窗口，首先获取系统的渲染模块
    Renderer* renderer = System::getSingleton().getRenderer();
    //调用描绘自己的函数
    drawSelf(renderer->getCurrentZ());
    //递增Z值，Z值基本没用，窗口的层叠关系是通过d_drawList的顺序确定的
    renderer->advanceZValue();
    // 渲染子窗口
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        d_drawList[i]->render();
    }
    // 激发渲染结束的事件
    WindowEventArgs args(this);
    onRenderingEnded(args);
}
```

render函数是窗口的基本渲染函数。由于窗口的父子关系，在CEGUI中只有一个根窗口，它就是DefaultGUISheet（当然它可以是任何一种窗口类型），所有CEGUI其他窗口都是它的直接或者间接子窗口。所以渲染的时候只要它调用了render函数，所有的CEGUI可见窗口都会被渲染。我们看到render函数主要调用了drawSelf函数来渲染自己，所以这个函数就是窗口描绘自己的主函数。

```
void Window::drawSelf(float z)
{
    //如果需要重绘则清理渲染缓冲，重新填充渲染缓冲区
  if (d_needsRedraw)
  {
    // 清除上次重绘的渲染缓冲内容
    d_renderCache.clearCachedImagery();
    // 如果窗口有渲染窗口类则调用它的render方法，否则调用populateRenderCache
    if (d_windowRenderer != 0)
    {
       d_windowRenderer->render();
    }
    else
    {
```

```
            populateRenderCache();
        }
        // 标记下次不用在重绘了，如果窗口需要重绘只需要设置这个变量为true
        d_needsRedraw = false;
    }
    //如果渲染缓冲有内容，则调用渲染缓冲的渲染函数
    if (d_renderCache.hasCachedImagery())
    {
            Point absPos(getUnclippedPixelRect().getPosition());
        // 以自己的屏幕窗口区域作为裁剪器
        Rect clipper(getPixelRect());
        // 如果窗口不是完全裁剪
        if (clipper.getWidth())
        {
            // 发送渲染命令到渲染模块，注意absPos，它是本窗口在屏幕上的偏移
            d_renderCache.render(absPos, z, clipper);
        }
    }
}
```

关于d_renderCache以及CEGUI的渲染机制请参考第4章。populateRenderCache函数Window类提供了空实现。它是个虚函数，如果派生类没有渲染模块，哪么必须重载这个函数实现窗口的描绘，否则窗口将不会被描绘。requestRedraw请求重绘函数，在窗口需要重绘的时候可以调用这个函数。d_needsRedraw 控制窗口是否重新渲染，signalRedraw告诉系统需要重新绘制窗口了。

```
void Window::requestRedraw(void) const
{
    d_needsRedraw = true;
    System::getSingleton().signalRedraw();
}
```

还有几个间接和渲染有关系的函数，他们与窗口的Alpha值有关系。setAlpha设置窗口的Alpha值，setInheritsAlpha设置是否窗口从父窗口取得Alpha值。

```
void Window::setAlpha(float alpha)
{
    d_alpha = alpha;
    WindowEventArgs args(this);
    onAlphaChanged(args);
}
void Window::setInheritsAlpha(bool setting)
{
    if (d_inheritsAlpha != setting)
    {
        // 保存旧的Alpha值，以便和设置后的Alpha值比较
        float oldAlpha = getEffectiveAlpha();
        // 通知设置改变了
        d_inheritsAlpha = setting;
        WindowEventArgs args(this);
        onInheritsAlphaChanged(args);
        // 如果Alpha值发生改变了，通知子窗口
        if (oldAlpha != getEffectiveAlpha())
        {
            args.handled = false;
```

```
                onAlphaChanged(args);
            }
        }
    }
//这个函数产生继承Alpha的效果，自己的Alpha值和父窗口的相乘
float Window::getEffectiveAlpha(void) const
{
    if ((d_parent == 0) || (!inheritsAlpha()))
    {
        return d_alpha;
    }
    return d_alpha * d_parent->getEffectiveAlpha();
}
void Window::onAlphaChanged(WindowEventArgs& e)
{
    // 扫描子窗口列表，调用所以继承Alpha子窗口的onAlphaChanged通知消息
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        if (d_children[i]->inheritsAlpha())
        {
      WindowEventArgs args(d_children[i]);
            d_children[i]->onAlphaChanged(args);
        }
    }
    requestRedraw();
    fireEvent(EventAlphaChanged, e, EventNamespace);
}
```

CEGUI提供了窗口Alpha继承的操作效果，似乎用的也不多。它产生什么效果呢？读者可以思考这个问题。

moveToFront_impl函数貌似和渲染没有关系，但它会调整渲染的层叠次序，所以把它放在这里。这个函数主要的功能是把窗口放在所以窗口的最上面。

```
bool Window::moveToFront_impl(bool wasClicked)
{
  bool took_action = false;
    // 如果窗口没有父窗口，则集合窗口自己，这也是递归结束的标志，一般到根窗口
    if (!d_parent)
    {
        // 如果窗口没有被激活则激活之
        if (!isActive())
        {
      took_action = true;
      ActivationEventArgs args(this);
            args.otherWindow = 0;
            onActivated(args);
        }
        return took_action;
    }
    // 提升父窗口到父窗口的所有兄弟窗口的最上层，这个会开始递归调用
  took_action = wasClicked ? d_parent->doRiseOnClick() :
                d_parent->moveToFront_impl(false);
  // 获取激活的兄弟窗口
```

```
    Window* activeWnd = getActiveSibling();
    // 如果自己没有被激活，则激活自己
    if (activeWnd != this)
    {
        took_action = true;
        // 通知自己被激活
        ActivationEventArgs args(this);
        args.otherWindow = activeWnd;
        onActivated(args);
        // 通知先前被激活的窗口，现在不在激活了
        if (activeWnd)
        {
            args.window = activeWnd;
            args.otherWindow = this;
            args.handled = false;
            activeWnd->onDeactivated(args);
        }
    }
    // 提升自己到所有自己的兄弟窗口的最高层
    if ((d_zOrderingEnabled) && !isTopOfZOrder())
    {
        took_action = true;
        // 从父窗口的渲染窗口列表中移除自己
        d_parent->removeWindowFromDrawList(*this);
        // 重新加入到父窗口的渲染列表中，通过这一步自己变成了父窗口中最高层
        d_parent->addWindowToDrawList(*this);
        // 通知相关的窗口自己的Z值改变了，可见渲染模块提供的Z值是没有用处的
        onZChange_impl();
    }
    return took_action;
}
```

这个函数会沿着父链递归激活所有的父窗口，直到激活调用者窗口。被激活的窗口是Top-Most的，它可以遮盖所有的窗口。窗口有个点击激活窗口的属性，如果窗口被点击了则会调用doRiseOnClick函数，这个函数调用moveToFront_impl并传递true给这个函数。

```
bool Window::doRiseOnClick(void)
{
    // 这个窗口是否提升自己，当窗口被点击后
    if (d_riseOnClick)
    {
        return moveToFront_impl(true);
    }
        //否则继续父链递归其他窗口
    else if (d_parent)
    {
        return d_parent->doRiseOnClick();
    }
    return false;
}
```

addWindowToDrawList，根据at_back参数决定添加到什么位置。窗口在d_drawList中的位置决定了窗口显示的位置，也就是我们平时说的Z值。（为什么不直接设置使用渲染模块提供的Z值呢？因为渲染UI的时候一般要关掉深度测试，所以渲染模块提供的Z值来决定显示的层叠顺序的方法已经无效了。）

```
void Window::addWindowToDrawList(Window& wnd, bool at_back)
{
  //添加到窗口的后面，它不可能被显示在最上面了，一般是没有被激活的窗口
  if (at_back)
  {
    // 计算窗口需要被插入的位置
    ChildList::iterator pos = d_drawList.begin();
    if (wnd.isAlwaysOnTop())
    {
      // 找到第一个TopMost（最高层）的窗口
      while ((pos != d_drawList.end()) && (!(*pos)->isAlwaysOnTop()))
        ++pos;
    }
    // 添加的第一个不是TopMost窗口的后面
    d_drawList.insert(pos, &wnd);
  }
  // 添加到窗口的前面，它一般是激活窗口显示在最上面
  else
  {
    // 计算窗口需要被插入的位置
    ChildList::reverse_iterator     position = d_drawList.rbegin();
    if (!wnd.isAlwaysOnTop())
    {
      // 找到最后一个非最高层窗口
      while ((position != d_drawList.rend()) && ((*position)->isAlwaysOnTop()))
        ++position;
    }
    //添加窗口到列表中
    d_drawList.insert(position.base(), &wnd);
  }
}
```

d_drawList越靠前的窗口越先被渲染，越先被渲染的窗口越容易被覆盖掉（后续渲染覆盖先前渲染）。所以 d_drawList添加到后面的窗口越靠近读者。addWindowToDrawList用来调整窗口的渲染次序也就是它的渲染层次。

CEGUI 0.6.0版将窗口的逻辑和窗口的渲染分开，逻辑还是放在Window类里面，渲染放到了一种新的以WindowRender为基类的窗口渲染类里面。我们从render类就可以看出来，如果窗口有渲染类会调用它的render函数，否则才调用窗口自己的渲染缓冲函数来渲染。setWindowRenderer函数为窗口设置一个渲染窗口类。

```
void Window::setWindowRenderer(const String& name)
{
  WindowRendererManager& wrm = WindowRendererManager::getSingleton();
  if (d_windowRenderer != 0)
  {
    // 容许改变一个窗口的渲染模块
    if (d_windowRenderer->getName() == name)
    {
      return;
    }
```

```
        WindowEventArgs e(this);
        onWindowRendererDetached(e);
        wrm.destroyWindowRenderer(d_windowRenderer);
    }
    if (!name.empty())
    {
        //创建一个新的窗口渲染类，并且和窗口关联起来
        d_windowRenderer = wrm.createWindowRenderer(name);
        WindowEventArgs e(this);
        onWindowRendererAttached(e);
    }
    else
    {
      //抛出异常
    }
}
WindowRenderer* Window::getWindowRenderer(void) const
{
    return d_windowRenderer;
}
void Window::onWindowRendererAttached(WindowEventArgs& e)
{
        //忽略参数的检查，这个函数把一个WindowRender和Window联系起来
    d_windowRenderer->d_window = this;
    d_windowRenderer->onAttach();
    fireEvent(EventWindowRendererAttached, e, EventNamespace);
}
//这个函数断开WindowRender和Window之间的联系
void Window::onWindowRendererDetached(WindowEventArgs& e)
{
    d_windowRenderer->onDetach();
    d_windowRenderer->d_window = 0;
    fireEvent(EventWindowRendererDetached, e, EventNamespace);
}
```

和渲染相关的一个重要的函数就是setLookNFeel。它涉及到LooKNFeel比较多，这里只做简单介绍，更加详细的内容读者参考第6章。

```
void Window::setLookNFeel(const String& look)
{
    if (d_windowRenderer == 0)
    {
      //抛出异常，省略这部分代码
    }
        //获取外观管理器的单件实例
    WidgetLookManager& wlMgr = WidgetLookManager::getSingleton();
        //如果窗口有外观的定义，CEGUI的实现允许重设窗口的外观，读者可以直接抛出异常
//来防止重设窗口外观的发生
    if (!d_lookName.empty())
    {
        d_windowRenderer->onLookNFeelUnassigned();
        const WidgetLookFeel& wlf = wlMgr.getWidgetLook(d_lookName);
        wlf.cleanUpWidget(*this);
```

```
    }
    d_lookName = look;
    // 调用外观的初始化函数对本窗口做一些处理
    const WidgetLookFeel& wlf = wlMgr.getWidgetLook(look);
    // 添加属性定义，应用属性以及创建子窗口等操作
    wlf.initialiseWidget(*this);
    // 作必要的绑定和初始化工作，Window类提供空的默认实现，派生类可以修改
    initialiseComponents();
    // 通知渲染窗口外观定义添加到窗口
    d_windowRenderer->onLookNFeelAssigned();
        //请求重绘
    requestRedraw();
}
```

窗口的外观影响的窗口的渲染，可以说一旦一个窗口的外观被定义后，这个窗口的渲染出的大概样子也就确定了。详细的原理请读者参考第6章。

3.2.5 事件响应与处理

窗口内部响应事件的函数，其实就是一些以on开头的函数。这些函数一般是CEGUI内部处理事件的函数，所以这些函数中一般都要激发一些事件，来调用外部的事件响应函数。这些函数根据事件的不同，功能各异。但有个共同的特点是他们处理内部状态改变。

下面以onParentSized为例介绍这组函数。

```
void Window::onParentSized(WindowEventArgs& e)
{
    //设置自己的区域，并且传递false表示不产生事件
    setArea_impl(d_area.getPosition(), d_area.getSize(), false, false);
        //计算窗口是否被移动或者尺寸是否改变
    bool moved = ((d_area.d_min.d_x.d_scale != 0) || (d_area.d_min.d_y.d_scale != 0));
    bool sized = ((d_area.d_max.d_x.d_scale != 0) || (d_area.d_max.d_y.d_scale != 0));
    // 查看是否被移动，激发事件（外部处理）
    if (moved)
    {
        WindowEventArgs args(this);
        onMoved(args);
    }
    if (sized)
    {
        WindowEventArgs args(this);
        onSized(args);
    }
    //如果窗口没有移动，且没有改变大小则调用子窗口布局函数
    if (!(moved || sized))
        performChildWindowLayout();
        //激发外部事件
        fireEvent(EventParentSized, e, EventNamespace);
}
//处理窗口移动的事件
void Window::onMoved(WindowEventArgs& e)
{
    // 通知子窗口父窗口已经被移动了
```

```
    const size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        d_children[i]->notifyScreenAreaChanged();
    }
  //这里没有调用requestRedraw函数，这是通知系统重新提交每个窗口的缓冲图像
  System::getSingleton().signalRedraw();
    //激发外部处理函数
    fireEvent(EventMoved, e, EventNamespace);
}
//这个函数只是设置以前计算的各种屏幕矩形不在有效，递归调用所有孩子对应函数
void Window::notifyScreenAreaChanged()
{
  d_screenUnclippedRectValid = false;
  d_screenUnclippedInnerRectValid = false;
  d_screenRectValid = false;
  d_screenInnerRectValid = false;
  // 通知所有孩子需要更新内部保存的屏幕矩形的数据
    const size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        d_children[i]->notifyScreenAreaChanged();
    }
}
//响应窗口改变的事件，并激发外部事件
void Window::onSized(WindowEventArgs& e)
{
    //通知子窗口，父窗口大小已经改变了
    size_t child_count = getChildCount();
    for (size_t i = 0; i < child_count; ++i)
    {
        WindowEventArgs args(this);
        d_children[i]->onParentSized(args);
    }
    //重新子窗口布局，这个函数会调整窗口外观LooKNFeel里定义的子窗口的位置
    performChildWindowLayout();
    //因为窗口的大小改变了，所以需要完全重绘，所以调用了这个函数
    requestRedraw();
    //激发外部事件
    fireEvent(EventSized, e, EventNamespace);
}
```

类似的这种函数还很多，这里就不一一介绍了。读者只要理解他们是响应某些事件的函数。比如窗口状态改变，操作状态改变（比如鼠标移入，移出），字体改变等等。这些改变都有对应的事件处理函数。读者也可以根据自己需要添加类似的函数。

3.2.6 窗口状态

窗口有许多状态，比如是否激活，是否可以操作，是否可见，是否总是在最高层等等。这里以setEnabled和setVisible为例。

```
void Window::setEnabled(bool setting)
{
    // 仅仅当设置值和当前值不同时才设置
```

```
    if (d_enabled != setting)
    {
        d_enabled = setting;
    WindowEventArgs args(this);
    if (d_enabled)
    {
      //判断一个窗口是否被可用，取决于它的祖先是否被激活
      if ((d_parent && !d_parent->isDisabled()) || !d_parent)
        onEnabled(args);
    }
    else
    {
      onDisabled(args);
    }
  }
}
//设置窗口的可见性
void Window::setVisible(bool setting)
{
    //只有设置改变是才处理
    if (d_visible != setting)
    {
        d_visible = setting;
    WindowEventArgs args(this);
        d_visible ? onShown(args) : onHidden(args);
    }
}
```

3.2.7 窗口与输入系统

输入来自键盘或者鼠标，键盘输入Window类的处理非常简单，只是简单的激发对应事件，这是因为Window是基类，没有涉及到具体的逻辑，这些处理留给了派生类。鼠标事件基类需要提供处理，以便派生类使用。所以这里主要介绍鼠标输入相关的函数。

```
void Window::onMouseButtonDown(MouseEventArgs& e)
{
  // 当鼠标单击的时候，不管是左键，中键还是右键，这是ToolTips应该消失
  Tooltip* tip = getTooltip();
  if (tip)
  {
    tip->setTargetWindow(0);
  }
    //处理左键，执行doRiseOnClick函数，默认只处理左键
  if (e.button == LeftButton)
  {
    e.handled |= doRiseOnClick();
  }
  // 如果窗口容许自动重复，设置自动重复的状态，自动重复的状态下如果鼠标没有抬起
//则过一定的时间会自动产生鼠标单击的事件，调用鼠标本函数。请参考updateSelf。
  if (d_autoRepeat)
  {
    if (d_repeatButton == NoButton)
      captureInput();
    if ((d_repeatButton != e.button) && isCapturedByThis())
```

```
        {
            d_repeatButton = e.button;
            d_repeatElapsed = 0;
            d_repeating = false;
        }
    }
        //激发鼠标单击的事件
    fireEvent(EventMouseButtonDown, e, EventNamespace);
}
void Window::onMouseButtonUp(MouseEventArgs& e)
{
    // 重设自动重复产生事件的状态
    if (d_autoRepeat && d_repeatButton != NoButton)
    {
        releaseInput();
        d_repeatButton = NoButton;
    }
        //激发鼠标抬起的消息。
        fireEvent(EventMouseButtonUp, e, EventNamespace);
}
```

captureInput捕获鼠标，大家应该对他比较熟悉。当一个窗口捕获输入后，所有的输入都会有捕获窗口的函数处理，直到该窗口释放了输入捕获。

鼠标进入窗口区域和鼠标离开窗口区域和鼠标点击抬起一样都是非常常用的函数，派生类经常需要修改。

```
//鼠标进入窗口的处理函数
void Window::onMouseEnters(MouseEventArgs& e)
{
        // 设置鼠标
        MouseCursor::getSingleton().setImage(getMouseCursor());
    // 控制ToolTip显示自己的信息
    Tooltip* tip = getTooltip();
    if (tip)
    {
        tip->setTargetWindow(this);
    }
        //激发鼠标进入窗口的事件
    fireEvent(EventMouseEnters, e, EventNamespace);
}
//鼠标离开窗口的处理函数
void Window::onMouseLeaves(MouseEventArgs& e)
{
    // 关闭这个窗口的ToolTip的显示
    Tooltip* tip = getTooltip();
    if (tip)
    {
        tip->setTargetWindow(0);
    }
        //激发鼠标离开的消息
    fireEvent(EventMouseLeaves, e, EventNamespace);
}
//鼠标移动的消息处理函数
void Window::onMouseMove(MouseEventArgs& e)
```

```
{
    // 设置ToolTips状态
    Tooltip* tip = getTooltip();
    if (tip)
    {
        tip->resetTimer();
    }
        //激发鼠标移动的消息
    fireEvent(EventMouseMove, e, EventNamespace);
}
```

这个几个函数是如何激发的，被谁调用的呢？是CEGUI的System类，关于它的详细说明在第4章。

3.2.8 窗口的其他功能

CEGUI窗口基类中有一些名称含有XML的函数，他们是负责将窗口写入到XML流中的，为了保存窗口到布局文件而设计的。他们虽然在游戏界面里没有什么用，但如果设计布局编辑器他们就派上用场了。由于篇幅原因这里就不在介绍他们了。除此之外还有字体的相关的函数，设置窗口文本的函数等。这些函数非常简单，就是简单的设置这些值，需要的话激发一些事件。当时需要说明的是字体在CEGUI的文字显示中非常重要。读者可以详细参考第8章的字体部分。

```
//设置窗口显示的文字
void Window::setText(const String& text)
{
        d_text = text;
    WindowEventArgs args(this);
        onTextChanged(args);
}
//设置字体，通过指针
void Window::setFont(Font* font)
{
        d_font = font;
    WindowEventArgs args(this);
        onFontChanged(args);
}
//通过字体的名称设置字体
void Window::setFont(const String& name)
{
        if (name.empty())
        {
                setFont(0);
        }
        else
        {
                setFont(FontManager::getSingleton().getFont(name));
        }
}
```

到这里CEGUI基本窗口类Window就介绍到这里，也许读者还有许多不理解的地方。读者可以自己阅读更多的代码，加深理解。

3.3 窗口类厂和类厂管理

本节共有两小节，第1节介绍窗口的类厂和类厂管理器，第2节介绍渲染窗口的类厂以及类厂管理器。与功能窗口不同的是渲染窗口的类厂管理器和渲染窗口管理器是同一个管理器。

3.3.1 窗口的类厂和类厂管理

CEGUI中窗口的创建和删除都是由类厂控制的，使用设计模式中的类厂的概念来设计的。CEGUI还普遍使用了另一个设计模式就是单件模式。大多数管理器都是单件，它们的特征就是有个getSingleton的函数。

CEGUI的类厂是使用三个宏以及一个工厂基类来创建的，三个宏是CEGUI_DECLARE_WINDOW_FACTORY，CEGUI_DEFINE_WINDOW_FACTORY以及CEGUI_WINDOW_FACTORY。一个基类便是WindowFactory，他们全部定义在CEGUIWindowFactory.h和.cpp中。

首先介绍所有类厂的基类WindowFactory。

```
class CEGUIEXPORT WindowFactory
{
public:
    //删除一个窗口，接口函数强制子类实现
    virtual void    destroyWindow(Window* window) = 0;
    //返回这个类厂的类型，也就是它创建的窗口的类型
    const String& getTypeName(void) const          {return d_type;}
protected:
    //构造函数保护
    WindowFactory(const String& type) : d_type(type) {}
public:
    // 兼容luabind
    virtual ~WindowFactory(void) {}
protected:
    //保存类型的成员变量
    String          d_type;
};
```

哪么CEGUI_DECLARE_WINDOW_FACTORY宏是如何声明一个类厂的呢。

```
#define CEGUI_DECLARE_WINDOW_FACTORY( T )\
class T ## Factory : public WindowFactory\
{\
public:\
    T ## Factory() : WindowFactory( T::WidgetTypeName ) {}\
  Window* createWindow(const String& name)\
  {\
    return new T (d_type, name);\
  }\
  void destroyWindow(Window* window)\
  {\
    delete window;\
  }\
};\
T ## Factory& get ## T ## Factory();
```

c++在宏展开的时候##会作为连接操作，比如T传入GUISheet哪么就会定义一个GUISheetFactory 的类厂。\表示下一行也是宏的展开，只不过为了方便阅读换行的。这下读者应该明白它展开后的样子了。为了直观一些我们以Window为例，展开这个宏。

```
class GUISheetFactory : public WindowFactory
{
public:
```

```
    GUISheetFactory() : WindowFactory( GUISheet::WidgetTypeName ) {}
  Window* createWindow(const String& name)
  {
    return new GUISheet(d_type, name);
  }
  void destroyWindow(Window* window)
  {
    delete window;
  }
};
GUISheetFactory& getWindowFactory();
```

getWindowFactory是全局函数，为了方便类注册而定义的。WidgetTypeName 是每个CEGUI窗口类定义的窗口的类型，它是静态变量。

类厂声明好了后还有一个全局函数没有定义，所以需要实现它。CEGUI_DEFINE_WINDOW_FACTORY完成这个功能。

```
#define CEGUI_DEFINE_WINDOW_FACTORY( T )\
T ## Factory& get ## T ## Factory()\
{\
    static T ## Factory s_factory;\
    return s_factory;\
}
```

它的实现一目了然，这里就不在带入实例了。

还有一个为了方便注册类厂的宏，它就是CEGUI_WINDOW_FACTORY。它调用全局的获取类厂的函数。

```
#define CEGUI_WINDOW_FACTORY( T ) (get ## T ## Factory())
```

显而易见窗口的创建和删除是由类厂的创建和删除函数实现的，创建简单的new一个新的对象，删除delete这个对象。

```
Window* createWindow(const String& name)
{
  return new Window (d_type, name);
}
void destroyWindow(Window* window)
{
  delete window;
}
```

类厂的好处不言而喻，它具有非常好的扩展性。创建和删除一种类型的对象不需要知道这个对象的细节，只需提供对象的类型，就可以窗口对象。而且可以通过在其他模块里注册新的类厂，各个模块里可以共享这些对象。它的好处还有很多这里就不在介绍了。有兴趣的读者可以参考设计模式相关的书籍。

CEGUI所有的类厂的注册和实现分别在CEGUIBaseFactories.h和CEGUIBaseFactories.cp文件中。这里简单列出如何定义和实现一个类厂。（省略的许多定义）

```
namespace CEGUI
{
    CEGUI_DEFINE_WINDOW_FACTORY(GUISheet);
    CEGUI_DEFINE_WINDOW_FACTORY(DragContainer);
}
```

类厂的声明，在CEGUIBaseFactories.h中。

```
namespace CEGUI
{
  CEGUI_DECLARE_WINDOW_FACTORY(GUISheet);
  CEGUI_DECLARE_WINDOW_FACTORY(DragContainer);
}
```

是不是非常简单呢。有了这些宏，实现的类厂即美观有简单。哪么另外一个宏用在那里呢？答案是WindowFactoryManager。

显然类厂需要一个管理器，来统一管理它。CEGUI使用WindowFactoryManager类来管理。它是一个单件类，显然CEGUI中所有的管理器都是单件了，以后就不做特殊说明了。所有的单件类都是使用Singleton模板来实现的。它的具体实现就不做讨论了，相信许多设计模式的书籍都有类似的定义。类厂管理器必须有个类厂名称到类厂指针的映射集，才能方便的通过名称找到类厂。一个窗口的（也可以称作类厂的）名称到窗口内部类型和窗口渲染窗口，以及窗口外观的映射。具体如下所示：

```
//字符串到类工厂指针的映射类型定义
typedef std::map<String, WindowFactory*, String::FastLessCompare> WindowFactoryRegistry;
//定义一种窗口类型别名的映射，就是给窗口类型起个别名，方便读者理解窗口功能
typedef std::map<String, AliasTargetStack, String::FastLessCompare>
TypeAliasRegistry;
//定义标准窗口映射类型的映射定义
typedef std::map<String, FalagardWindowMapping, String::FastLessCompare> FalagardMapRegistry;
//定义的三种映射变量
WindowFactoryRegistry d_factoryRegistry;
TypeAliasRegistry       d_aliasRegistry;
FalagardMapRegistry    d_falagardRegistry;
```

想必读者还是不清楚d_aliasRegistry，以及d_falagardRegistry究竟是什么含义。类AliasTargetStack其实就是一个String类的数组。它的数据成员定义如下：

```
//定义String数组类型
typedef std::vector<String>   TargetTypeStack;
//定义数组类型变量
TargetTypeStack   d_targetStack;
```

TypeAliasRegistry定义了一个字符串到一个字符串数组的映射。alias在英文里是别名的意思。读者可以想象d_aliasRegistry是一个窗口类型别名到窗口类型的映射，这种映射是一对多的，也就是说一个别名可以对应多个实际窗口类型。给窗口类型起别名主要是为了方便理解这类窗口是做什么的。但一个别名对应多个实际类型，就不知是为何设计的了。我认为这个设计完全没有必要，d_aliasRegistry没有什么太大的意义，没有它程序一样可以非常顺利的跑起来，所以读者可以忽略它，我们也不在介绍它相关的几个函数了。

d_falagardRegistry非常重要的一个成员，它定义了窗口类型（创建窗口时使用，外部的窗口类型）到内部窗口类型（具体那个类，也就是前文所说的WidgetTypeName，它是窗口类的静态成员，也是类厂的类型），内部窗口渲染类型，以及窗口外观定义的映射。下面这个结构FalagardWindowMapping定义了这种映射关系。

```
struct CEGUIEXPORT FalagardWindowMapping
{
    //外部的窗口类型名称
    String  d_windowType;
    //窗口外观名称，对应LooKNfeel文件中定义的外观名称
    String  d_lookName;
    //窗口内部类型名称，对应窗口类的静态变量WidgetTypeName
    String  d_baseType;
    //渲染窗口的类型名称，用来创建渲染窗口
    String  d_rendererType;
};
```

有了数据成员必须要有成员函数来操作这些成员变量，下面介绍三个和这个成员相关的重要函数。

```
//添加一个映射到d_falagardRegistry
void WindowFactoryManager::addFalagardWindowMapping(const String& newType, const String& targetType, const String&
lookName, const String& renderer)
{
```

```
    WindowFactoryManager::FalagardWindowMapping mapping;
    mapping.d_windowType = newType;
    mapping.d_baseType   = targetType;
    mapping.d_lookName   = lookName;
    mapping.d_rendererType = renderer;
    //查找是否已经定义了这个名字的类型映射
    if (d_falagardRegistry.find(newType) != d_falagardRegistry.end())
    {
        //这里原本记录了日志，这里省略
    }
        //添加到映射列表中
    d_falagardRegistry[newType] = mapping;
}
//根据外部窗口类型，查找LooKNFeel的类型
const String& WindowFactoryManager::getMappedLookForType(const String& type) const
{
    FalagardMapRegistry::const_iterator iter =
        d_falagardRegistry.find(getDereferencedAliasType(type));
    if (iter != d_falagardRegistry.end())
    {
        return (*iter).second.d_lookName;
    }
    // 没有找到，抛出异常
    else
    {
        throw InvalidRequestException("WindowFactoryManager::getMappedLookForType -
        Window factory type '" + type + "' is not a falagard mapped type (or an alias for one).");
    }
}
//根据外部窗口类型名称找到窗口的渲染类型
const String& WindowFactoryManager::getMappedRendererForType(const String& type) const
{
    FalagardMapRegistry::const_iterator iter =
        d_falagardRegistry.find(getDereferencedAliasType(type));

    if (iter != d_falagardRegistry.end())
    {
        return (*iter).second.d_rendererType;
    }
    //没有找到抛出异常
    else
    {
        throw InvalidRequestException("WindowFactoryManager::getMappedLookForType -
        Window factory type '" + type + "' is not a falagard mapped type (or an alias for one).");
    }
}
//这个函数就是别名的函数之一，它递归查找别名对应的窗口类型（外部），为什么要递归呀//读者可以思考，当没有找到别名的时候，说明没有别名，直接返回原来的类型
String WindowFactoryManager::getDereferencedAliasType(const String& type) const
{
    TypeAliasRegistry::const_iterator alias = d_aliasRegistry.find(type);
    // 这里递归查找，直到找到窗口类型的名称
```

```
    if (alias != d_aliasRegistry.end())
        return getDereferencedAliasType(alias->second.getActiveTarget());
    // 找不到别名的时候返回传入的参数
    return type;
}
```

由getDereferencedAliasType查找不到别名的时候会返回传入的类型可以理解getMappedRendererForType和getMappedLookForType的调用原理了。也就是说没有别名的时候，还是会正常工作，只要窗口外部类型名称的FalagardWindowMapping存在。

读者肯定想知道这些类型定义在那里，谁负责加载他们，这些内容在第4章的Schema相关的章节。读者不用着急，一步一步，扎扎实实的跟着教程走，肯定会有很大收获的。

最后一个成员变量d_factoryRegistry，它保存了类厂类型（也是窗口内部类型）到类厂指针的映射。操作它的成员函数有添加类厂，删除类厂，查找类厂，和是否存在判断四种。

```
void WindowFactoryManager::addFactory(WindowFactory* factory)
{
    // 省略异常判断参数检查以及日志
    // 使用类厂名称(和窗口类型名称相同）映射工厂指针
    d_factoryRegistry[factory->getTypeName()] = factory;
}
void WindowFactoryManager::removeFactory(const String& name)
{
    //移除工厂
    d_factoryRegistry.erase(name);
}
//另一种形式
void WindowFactoryManager::removeFactory(WindowFactory* factory)
{
    if (factory)
    {
        removeFactory(factory->getTypeName());
    }
}
//根据类厂名称获取类厂的指针
WindowFactory* WindowFactoryManager::getFactory(const String& type) const
{
    // 通过别名查找，这个函数已经介绍过
    String targetType(getDereferencedAliasType(type));
    //首先尝试别名查找类厂指针
    WindowFactoryRegistry::const_iterator pos = d_factoryRegistry.find(targetType);
    // 找到后返回
    if (pos != d_factoryRegistry.end())
    {
        return pos->second;
    }
    //没有找到，尝试在映射中寻找
    else
    {
        FalagardMapRegistry::const_iterator falagard = d_falagardRegistry.find(targetType);

        // 找到后，递归调用，通过别名系统的过滤
        if (falagard != d_falagardRegistry.end())
```

```
        {
            return getFactory(falagard->second.d_baseType);
        }
        // 没有找到抛出异常，省略代码
        else
        {
        }
    }
}
//判断一个类厂是否存在
bool WindowFactoryManager::isFactoryPresent(const String& name) const
{
    // 通过别名找到内部窗口类型，d_factoryRegistry保存的是内部窗口类型到类厂的映射
    String targetType(getDereferencedAliasType(name));
    // 尝试查找
    if (d_factoryRegistry.find(targetType) != d_factoryRegistry.end())
    {
        return true;
    }
    // 否则在d_falagardRegistry.中查找
    else
    {
        return (d_falagardRegistry.find(targetType) != d_falagardRegistry.end());
    }
}
```

有了类厂的这些知识，读者应该明白如何如何创建和删除一个窗口了。与窗口类似，渲染窗口也有其类厂和类厂的管理类。图3-3列出了窗口和窗口类厂之间的关系。希望可以加深读者的理解。这里只是那Eidt，Button，Tree三个基本窗口来举例子。并不代表只有这三个基本窗口，实际上所有的窗口类都有一个窗口类工厂，负责他的创建和销毁。读者如果需要添加一个新的窗口，也需要添加它的类厂。不过类厂的添加不需要太多的代码，使用我们介绍过的三个宏就可以轻松解决。

下一节介绍渲染窗口的类厂以及类厂管理器。

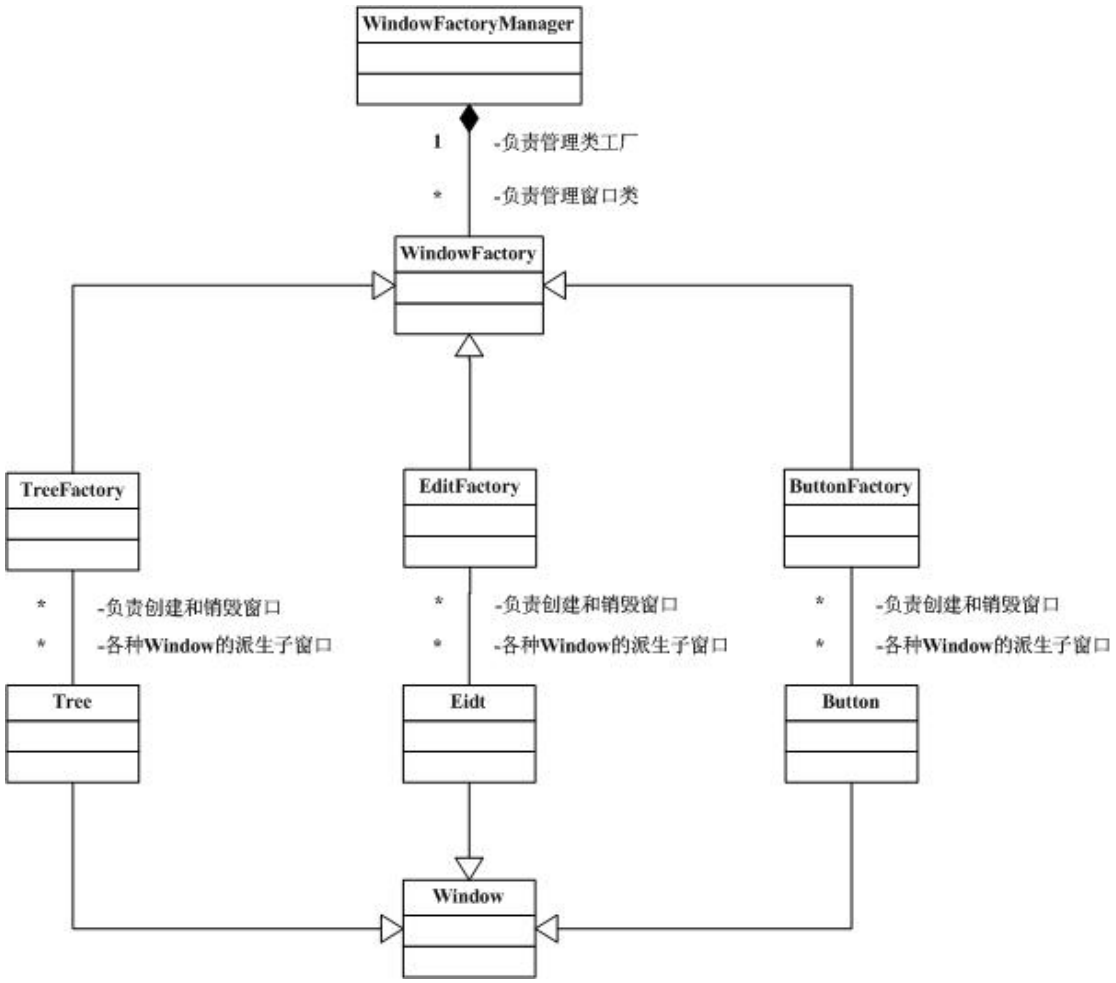

图3-3 类厂以及类厂管理器

3.3.2 渲染窗口的类厂和类厂管理

渲染窗口，读者可能还不是非常明白什么是渲染窗口。它是负责窗口的渲染的，记得在3.2节中介绍的drawSelf函数吗？如果窗口中存在渲染窗口的指针则调用渲染窗口的render函数，否则调用Window提供的populateRenderCache函数。可见渲染窗口是负责渲染工作的。3.3.1中介绍的结构FalagardWindowMapping中d_rendererType就是渲染窗口类厂或者渲染窗口的类型名称。与普通窗口最大的不同是渲染窗口是以模块的形式提供的（编译静态库时例外）。它作为CEGUI的渲染子集出现，之所以要这样设计是因为渲染窗口是可以对应多个普通窗口的，也就是说一个渲染窗口可以负责多个窗口的渲染。但不是说每一个渲染窗口都可以渲染任意一个窗口。只要窗口的LooKNFeel定义了渲染窗口需要的区域和命名对象等，渲染窗口就可以渲染这个窗口。也许比较抽象，读者不好理解，这是因为读者还不了解LooKNFeel的缘故。LooKFeel将在第6章详细介绍。

渲染窗口的类厂也是使用几个宏实现的。具体定义如下：

```
// 定义一个模块，我们说过渲染窗口使用模块来提供
#define CEGUI_DECLARE_WR_MODULE( moduleName )\
\
class CEGUI::WindowRendererFactory;\
\
//这两个函数是导出函数，供外部的程序调用注册单个和多个模块内部的各种渲染窗口类
extern "C" CEGUIWRMODULE_API void registerFactoryFunction(const CEGUI::String& type_name);\
extern "C" CEGUIWRMODULE_API CEGUI::uint registerAllFactoriesFunction(void);\
void doSafeFactoryRegistration(CEGUI::WindowRendererFactory* factory);
```

```
// 类厂定义的宏，相信有了第1节的知识读者可以把它还原回来分析了
#define CEGUI_DEFINE_WR_FACTORY( className )\
namespace CEGUI {\
class className ## WRFactory : public WindowRendererFactory\
{\
public:\
   className ## WRFactory(void) : WindowRendererFactory(className::TypeName) { }\
   WindowRenderer* create(void)\
   { return new className(className::TypeName); }\
   void destroy(WindowRenderer* wr)\
   { delete wr; }\
};\
}\
static CEGUI::className ## WRFactory  s_ ## className ## WRFactory;

// 开始类厂映射，是不是有点类似MFC呢？
#define CEGUI_START_WR_FACTORY_MAP( module )\
struct module ## WRMapEntry\
{\
   const CEGUI::utf8* d_name;\
   CEGUI::WindowRendererFactory* d_factory;\
};\
\
module ## WRMapEntry module ## WRFactoriesMap[] =\
{\
// 工厂映射的结束宏
#define CEGUI_END_WR_FACTORY_MAP {0,0}};
// 映射的入口，定义单独的一个映射
#define CEGUI_WR_FACTORY_MAP_ENTRY( class )\
{CEGUI::class::TypeName, &s_ ## class ## WRFactory},

// 定义模块，实现相应的导出函数
#define CEGUI_DEFINE_WR_MODULE( module )\
void registerFactoryFunction(const CEGUI::String& type_name)\
{\
   module ## WRMapEntry* entry = module ## WRFactoriesMap;\
   while (entry->d_name)\
   {\
      if (entry->d_name == type_name)\
      {\
         doSafeFactoryRegistration(entry->d_factory);\
         return;\
      }\
      ++entry;\
   }\
\
   throw CEGUI::UnknownObjectException("::registerFactory - The window renderer factory f
or type '" + type_name + "' is not known in this module.");\
}\
\
extern "C" CEGUI::uint registerAllFactoriesFunction(void)\
{\
```

```
    CEGUI::uint count = 0;\
    module ## WRMapEntry* entry = module ## WRFactoriesMap;\
    while (entry->d_name)\
    {\
        doSafeFactoryRegistration(entry->d_factory);\
        ++entry;\
        ++count;\
    }\
    return count;\
}\
\
void doSafeFactoryRegistration(CEGUI::WindowRendererFactory* factory)\
{\
    assert(factory != 0);\
\
    CEGUI::WindowRendererManager& wfm = CEGUI::WindowRendererManager::getSingleto
n();\
    if (wfm.isFactoryPresent(factory->getName()))\
    {\
        CEGUI::Logger::getSingleton().logEvent(\
            "WindowRenderer factory '" + factory->getName() + "' appears to be already regist
                ered, skipping.",\
            CEGUI::Informative);\
    }\
    else\
    {\
        wfm.addFactory(factory);\
    }\
}
```

registerFactoryFunction和registerAllFactoriesFunction一般会被CEGUI调用，当CEGUI加载模式的时候会查找所有的外部接口，加载他们然后调用这两个函数（不一定都调用）。这两个函数的功能是完成模块（Window下是Dll模块，Linux下是so模块）内部所有的窗口类的类厂注册工作。渲染窗口和窗口一样，都是只有类厂可以创建和删除一个。需要注意的是由于排版的缘故，发生了一些串行的事情。这几个宏构造了一个静态的数组然后将通过对数组的扫描注册数组中的每一个类厂。实现比较简单，很类似MFC中提供的宏。

哪么这些宏如何使用呢？首先需要在头文件中声明一个模块，通过CEGUI_DECLARE_WR_MODULE宏生成一个模块导出函数的声明。然后在实现文件中调用CEGUI_DEFINE_WR_MODULE宏实现上面声明的几个函数。函数已经完成了声明但是类厂的数组还没有定义。定义一个类厂通过CEGUI_DEFINE_WR_FACTORY，数组以CEGUI_START_WR_FACTORY_MAP开始，以CEGUI_END_WR_FACTORY_MAP收尾，中间还有许多填写数组的宏CEGUI_WR_FACTORY_MAP_ENTRY。具体实现读者参考FalModule头文件和实现文件。

渲染窗口的管理器和渲染窗口类工厂管理器是一个管理器。它就是WindowRendererManager。这个类负责管理类工厂以及渲染窗口。它有唯一的成员变量就是类厂的名称到类厂指针的映射。可以发现这种映射在CEGUI中使用非常广泛。

```
//定义类厂映射类型
typedef std::map<String, WindowRendererFactory*, String::FastLessCompare> WR_Registry;
//定义成员变量
WR_Registry d_wrReg;
```

WindowRendererManager提供了六个主要的函数。四个负责管理类厂，两个负责管理渲染窗口。所谓管理无非就是添加，删除，存在判断，获取等几项。

```
//存在判断函数
```

```
bool WindowRendererManager::isFactoryPresent(const String& name) const
{
    return (d_wrReg.find(name) != d_wrReg.end());
}
//获取类厂函数
WindowRendererFactory* WindowRendererManager::getFactory(const String& name) const
{
    WR_Registry::const_iterator i = d_wrReg.find(name);
    if (i != d_wrReg.end())
    {
        return (*i).second;
    }
    throw UnknownObjectException("There is no WindowRendererFactory named '"+name+"' available");
}
//添加一个新的类厂函数
void WindowRendererManager::addFactory(WindowRendererFactory* wr)
{
    if (wr == 0)
    {
        return;
    }
    if (d_wrReg.insert(std::make_pair(wr->getName(), wr)).second == false)
    {
        throw AlreadyExistsException("A WindowRendererFactory named '"+wr->getName()+"
        ' already exist");
    }
    Logger::getSingleton().logEvent("WindowRendererFactory '"+wr->getName()+"' added.");
}
//删除类厂的函数
void WindowRendererManager::removeFactory(const String& name)
{
    d_wrReg.erase(name);
}
```

添加类厂的函数，上面介绍的模块导出宏中定义的doSafeFactoryRegistration函数调用。总之读者如果希望管理渲染窗口的类厂，就可以调用这个单件的这四个成员函数。下面是创建和销毁渲染窗口的两个渲染窗口函数。他们的实现也非常简单。

```
//创建一个渲染窗口，先找到类厂，然后通过类厂创建
WindowRenderer* WindowRendererManager::createWindowRenderer(const String& name)
{
    WindowRendererFactory* factory = getFactory(name);
    return factory->create();
}
//销毁一个渲染窗口，先找到类厂，然后通过类厂销毁
void WindowRendererManager::destroyWindowRenderer(WindowRenderer* wr)
{
    WindowRendererFactory* factory = getFactory(wr->getName());
    factory->destroy(wr);
}
```

这两个函数没有检查是否能获取渲染窗口的类厂，如果名称不对无法获取渲染窗口的类厂会导致程序崩溃，使用的时候需要注意。

本节介绍了窗口的类厂和渲染窗口的类厂以及渲染窗口的管理。下一节详细介绍窗口的管理类。

3.4 窗口管理系统

CEGUI提供许多种窗口，而且每种窗口都有许多的实例，显然需要一个管理类来管理这些窗口。这个管理类就是WindowManager，它是个单件类，这个系统只有一个。

这个类的主要数据成员如下所示：

```
//定义用来实现窗口注册的类，通过窗口名称来映射窗口的指针
typedef std::map<String, Window*, String::FastLessCompare>
WindowRegistry;
//定义保存窗口指针的数组
typedef std::vector<Window*>    WindowVector;
//定义窗口注册的成员变量
WindowRegistry                  d_windowRegistry;
//窗口的指针数组
WindowVector    d_deathrow;
//定义自动产生名称用到的自增计数器
unsigned long   d_uid_counter;
//默认资源组的名称
static String d_defaultResourceGroup;
```

每个成员的含义已经说明，相信读者完全可以理解。下面介绍重要的成员函数。首先肯定是创建一个窗口了。创建窗口只需要它的名称和窗口类型，以及窗口名称的前缀。

```
Window* WindowManager::createWindow( const String& type, const String& name /*= ""*/, const String& prefix /*= ""*/ )
{
   // 确保不是空的名称传递给类厂创建
   String finalName(prefix + name);
   if (finalName.empty())
   {
      //产生唯一的窗口名称，使用它来确保窗口名称的唯一性
      finalName = generateUniqueWindowName();
   }
      //如果窗口已经存在则抛出异常
      if (isWindowPresent(finalName))
      {
            throw AlreadyExistsException("WindowManager::createWindow - A Window object wit
      h the name '" + finalName +"' already exists within the system.");
      }
      //获取类厂管理器，根据类型获取对应类厂
   WindowFactoryManager& wfMgr = WindowFactoryManager::getSingleton();
   WindowFactory* factory = wfMgr.getFactory(type);
      //调用类厂的创建窗口函数，new一个新的窗口
   Window* newWindow = factory->createWindow(finalName);
      newWindow->setPrefix(prefix);
   // 查看是否需要映射一个外观的定义
   if (wfMgr.isFalagardMappedType(type))
   {
        //获取窗口的外部名称到内部名称，外观定义，以及渲染窗口定义映射
      const WindowFactoryManager::FalagardWindowMapping& fwm = wfMgr.getFalagard
      MappingForType(type);
      //这是一个有外观的定义的窗口，应用窗口的外观定义以及渲染窗口
      newWindow->d_falagardType = type;
      newWindow->setWindowRenderer(fwm.d_rendererType);
      newWindow->setLookNFeel(fwm.d_lookName);
```

```
    }
    //保存到窗口的注册映射集中，并且返回
    d_windowRegistry[finalName] = newWindow;
    return newWindow;
}
```

其次，删除一个窗口。

```
void WindowManager::destroyWindow(const String& window)
{
    //查找窗口，在窗口名称到窗口指针的映射中
    WindowRegistry::iterator wndpos = d_windowRegistry.find(window);
    //窗口找到了
    if (wndpos != d_windowRegistry.end())
    {
    Window* wnd = wndpos->second;
    //从映射中移除
    d_windowRegistry.erase(wndpos);
    // 窗口内部清理
    wnd->destroy();
    // 添加窗口的死亡窗口数组中，CEGUI会每帧清理这个数组
    d_deathrow.push_back(wnd);
    // 通知系统一个窗口被销毁了，系统修改自己的状态
    System::getSingleton().notifyWindowDestroyed(wnd);
    }
}
//调用destroyWindow函数并没调用类厂来销毁窗口，只做了窗口的内部清理，真正的清理
//在cleanDeadPool函数中
void WindowManager::cleanDeadPool(void)
{
  WindowVector::reverse_iterator curr = d_deathrow.rbegin();
  for (; curr != d_deathrow.rend(); ++curr)
  {
        //获取类厂指针，然后删除窗口
    WindowFactory* factory = WindowFactoryManager::getSingleton().getFactory((*curr)
        ->getType());
    factory->destroyWindow(*curr);
  }
  // 清空死亡窗口数组
  d_deathrow.clear();
}
```

第三，加载布局文件的函数，我们知道CEGUI创建窗口可以通过XML格式的布局文件。所以通过加载布局文件可以生成一些窗口，这个函数返回他们的根窗口。

```
//filename是布局文件的路径。name_prefix是创建窗口时的前缀，设计前缀是为了方便布局//文件的导入，就像C++中的
#include命令一样。resourceGroup是资源组，资源提供模块根
//据资源组来加载资源，callback是遇到窗口属性时的回调函数，userdata是callback的参数
Window* WindowManager::loadWindowLayout(const String& filename, const String& name_prefix, const String& resourceGroup,
PropertyCallback* callback, void* userdata)
{
    if (filename.empty())
    {
        //抛出异常，这里省略
```

```
    }
    //创建布局文件的处理类，2.3节详细介绍了布局文件的加载
  GUILayout_xmlHandler handler(name_prefix, callback, userdata);
    //分析布局文件handler具体处理每个元素，创建窗口，窗口事件和窗口属性等
    try
    {
    System::getSingleton().getXMLParser()->parseXMLFile(handler,
        filename, GUILayoutSchemaName, resourceGroup.empty() ? d_defaultResourceGr
            oup : resourceGroup);
    }
    catch(...)
    {
    Logger::getSingleton().logEvent("WindowManager::loadWindowLayout - loading of la
    yout from file '" + filename +"' failed.", Errors);
    throw;
    }
    //返回根窗口的指针
    return handler.getLayoutRootWindow();
}
```

这个函数返回了加载的布局文件的根窗口的指针。它是布局文件中所有窗口的直接或间接父窗口。

第四，窗口的获取和窗口重命名，窗口是否存在，输出一个窗口的布局流到指定的流中。这几个函数比较简单，这里就不做介绍。

3.5 本章小结

1.还原渲染窗口的注册类厂的宏，并且理解它们是如何工作的。

2.尝试着给窗口添加一个新的属性表示是否激发渲染事件，如果这个属性值是True则激发否则不激发事件。

3.尝试着注册一个窗口单击事件的处理函数，并在处理函数里弹出一个对话框。

看懂了，不代表真的懂了。自己动手非常重要。虽然本章并没有第2和第3题的例子，但读者可以思考如何如何实现，如果实在实现不了也不要气馁。后续章节会有例子。

第4章 CEGUI核心控制体系

看到章名，读者可能不是很理解这章讲什么。前三章讲述的都是一个个的零散的结构，这一章要把他们统一起来。第1节，系统控制讲述CEGUI的控制核心类System。第2节，资源管理讲述CEGUI的核心资源管理类Schema，第3节系统接口，讲述CEGUI各种接口的定义，以及他们如何在系统中起作用。总之，这章以后读者会对CEGUI的执行流，资源管理，各种接口非常熟悉。阅读完这一章读者对CEGUI会有整体的概念，有一种找到思路豁然开朗的感觉。

4.1 系统控制

任何库都有一套程序执行路径的控制体系，CEGUI也不例外。CEGUI中System类负责与外部的程序（游戏）通讯，负责内部各个模块之间的联系。总之它是CEGUI控制核心类。

System类派生自EventSet和Sigleton模板。

4.1.1 系统变量

在介绍System之前，我们先分析System类的成员变量，以及他们的用途。从这些变量就可以推断出系统提供的功能。还记得图1-8CEGUI的架构图吗，当时我们说CEGUIBase模块是联系各个子模块的核心模块。其实更加具体的说应该是System类。

```
//渲染接口的指针，对应渲染模块的实现
Renderer*       d_renderer;
//资源提供接口的指针，对应一个资源提供接口
ResourceProvider* d_resourceProvider;
//默认字体，当窗口没有设置字体的时候使用系统提供的默认字体
Font*           d_defaultFont;
//CEGUI界面是否需要重绘的标志变量，如果为true则所有可见窗口将被重绘
bool        d_gui_redraw;
//保存当前鼠标下面的窗口，如果没有窗口捕获输入则这个窗口响应鼠标输入
Window*       d_wndWithMouse;
//根窗口，系统内所有窗口的父窗口
Window*       d_activeSheet;
//系统内的模态窗口，如果模态窗口存在则所有的输入都会输入到这个窗口
Window*       d_modalTarget;
//保存CEGUI的版本
String d_strVersion;
uint        d_sysKeys;          //鼠标各个键的按下状态，抬起是标志被取消
bool        d_lshift;           //左Shift键是否被按下标志
bool        d_rshift;           //右Shift键是否被按下标志
bool        d_lctrl;            //左Ctrl键是否被按下标志
bool        d_rctrl;            //左Ctrl键是否被按下标志
bool        d_lalt;             //左Alt键是否被按下标志
bool        d_ralt;             //左Alt键是否被按下标志
//鼠标单击时间，单位秒
double          d_click_timeout;
//鼠标双击时间，单位秒
double          d_dblclick_timeout;
//鼠标多重点击的限制区域大小，CEGUI支持三击
Size        d_dblclick_size;
//鼠标单击，双击，三击的实现数组
MouseClickTrackerImpl* const    d_clickTrackerPimpl;
//默认鼠标图像，Image代表纹理的一个区域
const Image*  d_defaultMouseCursor;
//脚本接口的指针，由脚本模块实现
```

```
ScriptModule*d_scriptModule;
//CEGUI系统退出的时候会调用的脚本文件名称
String      d_termScriptName;
//鼠标移动时，移动的偏移量乘以这个因子得出最后的鼠标移动量
float d_mouseScalingFactor;
//XML处理接口指针，由XML处理模块提供，注意区分XML解析模块
XMLParser*  d_xmlParser;
//标记是否System创建了XML的解析器，如果是则需要在退出的时候做清理工作
bool        d_ourXmlParser;     //!< true when we created the xml parser.
//动态模块的统一加载器，在Window下可以认为它加载dll文件，在linux下加载so文件
DynamicModule* d_parserModule;
//系统提供的默认Tooltip对象
Tooltip* d_defaultTooltip;
//判断是否System创建了这个Tooltip，类似d_ourXmlParser
bool     d_weOwnTooltip;
//保存System注册到Render的EventDisplaySizeChanged事件的处理函数，在System析构的//时候断开与Render的链接，也就是反注册这个事件的处理函数，否则后果很严重
Event::Connection d_rendererCon;
//默认XML解析器的名称，动态库的名称
static String   d_defaultXMLParserName; //!< Holds name of default XMLParser
```

这些变量可以分为以下几个部分：

1.外部模块的指针（包括XML解析模块，XML处理模块，渲染模块，资源提供模块等）

2.输入相关的变量（包括鼠标和键盘）

3.系统默认的一些变量（包括字体，Tooltip等）

4.当前系统中各种状态的窗口指针（包括模态窗口指针，鼠标上窗口指针等）

5.其他一些变量（不能归为上述4类的其他变量，比如版本等）

这一节介绍了System的各个成员变量的含义。System类在CEGUI中是非常重要的流程控制核心类。它控制系统初始化和退出流程，渲染流程，输入系统的事件派遣流程等。从这个类的名称也可以推断出它的功能。

4.1.2 初始化和退出流程

学习别人写的代码有几个手段，其中有一条就是，跟踪程序的初始化和退出以及其他执行流程。往往初始化的时候要初始化所有的重要的模块，退出的时候要反初始化许多重要的模块。哪么我们先从初始化流程开始。System类的构造函数初始化自己的成员变量到默认值，创建了许多重要的结构。

```
System::System(Renderer* renderer,
               ResourceProvider* resourceProvider,
               XMLParser* xmlParser,
               ScriptModule* scriptModule,
               const String& configFile,
               const String& logFile)

: d_renderer(renderer),
  d_resourceProvider(resourceProvider ? resourceProvider : d_renderer->createResourceProvider()),
//这里省略许多的成员变量简单初始化
{
    bool userCreatedLogger = true;
```

```
    //设置浮点数的小数点的表示符为"."，如果不设置在不同的操作系统上（主要指不同地
//区的操作系统，比如中文和英文等）小数点的设置可能是不一样的，这样可能导致从
//属性字符串中获取的格式化数据变的无效，比如0.5可能会变成0和5两个整形数，
//不是浮点的0.5了
  setlocale(LC_NUMERIC, "C");
  // 第一个创建的对象是打日志的单件对象
  if (!Logger::getSingletonPtr())
  {
    new DefaultLogger();
    userCreatedLogger = false;
  }
  // 设置 CEGUI 的版本信息
  d_strVersion = PropertyHelper::uintToString(CEGUI_VERSION_MAJOR) + "." +
    PropertyHelper::uintToString(CEGUI_VERSION_MINOR) + "." +
    PropertyHelper::uintToString(CEGUI_VERSION_PATCH);
  // 处理模块初始化和解析模块的创建
  setupXMLParser();
  // 定义几个可能从CEGUI的配置XML文件中读取的变量
  String configLogname, configSchemeName, configLayoutName, configInitScript,
    defaultFontName;
  // 如果存在配置文件，加载配置文件，获取配置变量
  if (!configFile.empty())
  {
    // 创建处理配置的模块
    Config_xmlHandler handler;
    // 分析和处理配置文件
    try
    {
      d_xmlParser->parseXMLFile(handler, configFile, CEGUIConfigSchemaName, "");
    }
    catch(...)
    {
      // 清理XML的缓冲，抛出异常，系统退出
      d_xmlParser->cleanup();
      delete d_xmlParser;
      throw;
    }
    // 如果用户没有自己创建Logger对象则设置打日志等级
    if(!userCreatedLogger)
      Logger::getSingleton().setLoggingLevel(handler.getLoggingLevel());
    // 获取先前定义的各种配置变量的值，从字面意义可以理解变量的含义
    configLogname      = handler.getLogFilename();
    configSchemeName   = handler.getSchemeFilename();
    configLayoutName   = handler.getLayoutFilename();
    defaultFontName    = handler.getDefaultFontName();
    configInitScript   = handler.getInitScriptFilename();
    d_termScriptName   = handler.getTermScriptFilename();
    // 设置默认的资源组
    if (!handler.getDefaultResourceGroup().empty())
    {
     d_resourceProvider->setDefaultResourceGroup(handler.getDefaultResourceGroup());
```

```
        }
    }
    // 如果系统创建了日志对象，则设置日志的参数
    if(!userCreatedLogger)
    {
        // 如果预定义了日志的名称，则设置日志名称
        if (!configLogname.empty())
        {
            Logger::getSingleton().setLogFilename(configLogname, false);
        }
        // 没有指定的话使用系统默认值
        else
        {
            Logger::getSingleton().setLogFilename(logFile, false);
        }
    }

    //开始主初始化流程
    Logger::getSingleton().logEvent("---- Begining CEGUI System initialisation ----");
    // 创建系统里所有的单件（各种管理器）
    createSingletons();
    // 添加窗口的核心类厂
    addStandardWindowFactories();
    // 以前版本窗口底板（所有窗口的根窗口）的类厂叫做DefaultGUISheet，现在版本
//改成了DefaultWindow，为了和以前版本兼容，给这个窗口起个别名DefaultGUISheet
        //我们终于看到了别名的用途了！！
    WindowFactoryManager::getSingleton().addWindowTypeAlias("DefaultGUISheet",
        GUISheet::WidgetTypeName);
    // 成输出功创建对象的日志
    outputLogHeader();
    // 注册渲染模块提供的显示区域改变事件的处理函数
    d_rendererCon = d_renderer->subscribeEvent(Renderer::EventDisplaySizeChanged,
        Event::Subscriber(&CEGUI::System::handleDisplaySizeChange, this));
    // 加载基本的模式文件，这个文件管理这CEGUI的各种资源，扩展名.schema
    if (!configSchemeName.empty())
    {
        try
        {
            SchemeManager::getSingleton().loadScheme(configSchemeName,
                                d_resourceProvider->getDefaultResourceGroup());
            //如果指定了默认字体，设置默认字体
            if (!defaultFontName.empty())
            {
                setDefaultFont(defaultFontName);
            }
        }
        catch (CEGUI::Exception&) {}  // catch exception and try to continue anyway
    }
    // 加载初始化的布局文件，如果配置文件里面配置了
    if (!configLayoutName.empty())
    {
```

```
        try
        {
            setGUISheet(WindowManager::getSingleton().loadWindowLayout(
                    configLayoutName));
        }
        catch (CEGUI::Exception&) {}  // catch exception and try to continue anyway
    }
    // 创建脚本模块的绑定
    if (d_scriptModule)
    {
        d_scriptModule->createBindings();
    }
    //执行初始化脚本，如果配置了
    if (!configInitScript.empty())
    {
        try
        {
            executeScriptFile(configInitScript);
        }
        catch (...) {}  // 脚本中出现的异常忽略
    }
}
//System的构造函数调用了几个重要的函数，这里介绍一下
//第一个设置XML解析模块
void System::setupXMLParser()
{
        //如果没有可用的XML解析模块，创建默认的模块
    if (!d_xmlParser)
    {
            //如果CEGUI编译为动态链接库，则动态加载XML模块库
#if !defined(CEGUI_STATIC)
        // 加载动态模块DynamicModule的实现读者自行阅读
        d_parserModule = new DynamicModule(String("CEGUI") +
                                d_defaultXMLParserName);
        // 获取解析模块提供的createParser函数，然后创建该对象
        XMLParser* (*createFunc)(void) =
            (XMLParser* (*)(void))d_parserModule->getSymbolAddress("createParser");
        d_xmlParser = createFunc();
#else
            //静态连接的话，直接调用这个函数创建
            d_xmlParser = createParser();
#endif
        // 设置是谁（CEGUI用户程序，还是CEGUI系统）创建了这个模块
        d_ourXmlParser = true;
    }
    // 调用初始化函数
    d_xmlParser->initialise();
}
//创建系统所有的单件类，如果读者需要添加系统级的单件需要在这里创建在//destroySingletons里销毁，读者可能对有些类还比较陌生
void System::createSingletons()
```

```
{
   new ImagesetManager();                //创建管理图像的单件类
   new FontManager();                    //创建管理字体的单件类
   new WindowFactoryManager();           //创建管理窗口工厂的单件类
   new WindowManager();                  //创建管理窗口的单件类
   new SchemeManager();                  //创建管理模式的单件类
   new MouseCursor();                    //创建管理鼠标的单件类
   new GlobalEventSet();                 //创建全局事件集的单件类
   new WidgetLookManager();              //创建管理外观的单件类
   new WindowRendererManager();          //创建管理渲染窗口类厂的单件类
}
//添加窗口类厂的函数
void System::addStandardWindowFactories()
{
   // 添加类厂到类厂管理器中，这里省略的许多类厂的注册，读者明白原理就行
   WindowFactoryManager& wfMgr = WindowFactoryManager::getSingleton();
      //CEGUI_WINDOW_FACTORY宏我们前面介绍过，这里被使用
   wfMgr.addFactory(&CEGUI_WINDOW_FACTORY(GUISheet));
   wfMgr.addFactory(&CEGUI_WINDOW_FACTORY(DragContainer));
  }
```

脚本接口提供的createBindings函数用来创建脚本函数到窗口的绑定（也就是使用脚本调用窗口函数的一种方法）。这个需要脚本模块来实现。模式管理类的loadScheme函数比较复杂这里不做介绍，读者可以参考第4.2节。读者知道了CEGUI系统的初始化过程，就可以理解System的各个参数，知道传递什么参数到CEGUI中，知道如何开始一个CEGUI的程序了。下面介绍退出流程，这个流程比较简单，在System的析构函数里面。

```
System::~System(void)
{
      Logger::getSingleton().logEvent("---- Begining CEGUI System destruction ----");
      // 执行结束脚本
      if (!d_termScriptName.empty())
      {
            try
            {
                  executeScriptFile(d_termScriptName);
            }
            catch (...) {}  // 忽略所有异常情况
      }
   // 反注册注册到渲染模块中的事件处理函数
   d_rendererCon->disconnect();
   // 清理脚本模块的绑定函数
   if (d_scriptModule)
   {
      d_scriptModule->destroyBindings();
   }

   // 清理XML模块
   cleanupXMLParser();
   // 首先清理所有窗口内部数据
   WindowManager::getSingleton().destroyAllWindows();
      //然后删除所有窗口（调用类厂的删除窗口函数（delete窗口）
```

```
    WindowManager::getSingleton().cleanDeadPool();
    //清除所有类厂
        WindowFactoryManager::getSingleton().removeAllFactories();
        //删除所有单件
    destroySingletons();
        Logger::getSingleton().logEvent("CEGUI::System singleton destroyed.");
        Logger::getSingleton().logEvent("---- CEGUI System destruction completed ----");
        //删除日志单件
        delete Logger::getSingletonPtr();
        //删除鼠标跟踪数据
        delete d_clickTrackerPimpl;
}
```

可以看出退出流程和初始化流程对应，需要注意清除的顺序。

小技巧：
CEGUI中有许多类似的概念。比如单件类，一般都是一类对象的管理器。处理所有的XML格式文件都有特定的处理方法，先定义处理类，然后调用XML解析模块来分析文件。如果读者希望了解某个文件的用途，最后的途径就是查看它的XML处理模块的代码。CEGUI中大量的使用了C++异常，编写代码的时候一般需要使用异常处理块来编写代码。

这一小节介绍了CEGUI系统的初始化和退出流程，希望读者可以理解。下一小节介绍输入系统事件的注入和派遣。

4.1.3 输入系统的事件派遣流程

UI需要和用户交互，显然需要响应用户的各种操作，鼠标输入，键盘输入。但是CEGUI不可能单独提供输入系统来获取玩家的输入，这个输入系统应该是游戏程序提供的。CEGUI提供了一些以Inject为前缀的输入系统注入函数。游戏程序通过调用这些函数来传入用户的操作信息。注入操作函数有3类，一类是鼠标的注入消息，一类是键盘的注入信息，还有一类是注入更新系统内所有窗口的消息。CEGUI目前还不支持游戏杆的输入操作。

第一类，鼠标的注入函数。共有六个函数，鼠标进入游戏程序的窗口（例如Windows窗口），鼠标退出游戏程序的窗口，鼠标按下，鼠标弹起，鼠标位置，鼠标滚动。

```
//注入鼠标移动的消息
bool System::injectMouseMove(float delta_x, float delta_y)
{
    MouseEventArgs ma(0);
    MouseCursor& mouse = MouseCursor::getSingleton();
    //看到d_mouseScalingFactor的作用了吧！
    ma.moveDelta.d_x = delta_x * d_mouseScalingFactor;
    ma.moveDelta.d_y = delta_y * d_mouseScalingFactor;
    ma.sysKeys = d_sysKeys;
    ma.wheelChange = 0;
    ma.clickCount = 0;
    ma.button = NoButton;
    //更新位置，offsetPosition改变mouse的位置信息
    mouse.offsetPosition(ma.moveDelta);
    ma.position = mouse.getPosition();
    //getTargetWindow函数获取鼠标下的目标窗口
    Window* dest_window = getTargetWindow(ma.position);
  // 如果先前的鼠标焦点窗口和目前的不同
  if (dest_window != d_wndWithMouse)
  {
    // 保存先前的鼠标焦点窗口
    Window* oldWindow = d_wndWithMouse;
    // 设置新的鼠标焦点窗口
```

```
        d_wndWithMouse = dest_window;
        // 通知旧的窗口鼠标已经离开了onMouseLeaves是Window类提供的事件处理函数
        if (oldWindow)
        {
            ma.window = oldWindow;
            oldWindow->onMouseLeaves(ma);
        }
        // 通知新窗口鼠标进入
        if (d_wndWithMouse)
        {
            ma.window = d_wndWithMouse;
            ma.handled = false;
            d_wndWithMouse->onMouseEnters(ma);
        }
    }
    // 通知当前的鼠标焦点窗口，鼠标移动了
    if (dest_window)
    {
        // 由于使用了前面的时间参数ma，所以这里需要设置处理标志为false，初始化
        ma.handled = false;
        // 循环处理直到事件被处理为止
        while ((!ma.handled) && (dest_window != 0))
        {
            ma.window = dest_window;
            dest_window->onMouseMove(ma);
            dest_window = getNextTargetWindow(dest_window);
        }
    }

    return ma.handled;
}
//鼠标离开游戏窗口的消息
bool System::injectMouseLeaves(void)
{
    MouseEventArgs ma(0);
    // 如果当前有鼠标上的窗口，则通知它并且设置鼠标上窗口为空
    if (d_wndWithMouse)
    {
        ma.position = MouseCursor::getSingleton().getPosition();
        ma.moveDelta = Vector2(0.0f, 0.0f);
        ma.button = NoButton;
        ma.sysKeys = d_sysKeys;
        ma.wheelChange = 0;
        ma.window = d_wndWithMouse;
        ma.clickCount = 0;
        d_wndWithMouse->onMouseLeaves(ma);
        d_wndWithMouse = 0;
    }
    return ma.handled;
}
//鼠标按下的注入消息
bool System::injectMouseButtonDown(MouseButton button)
```

```
{
    // 这里更新了鼠标点击的标志到d_sysKeys里
    d_sysKeys |= mouseButtonToSyskey(button);
    MouseEventArgs ma(0);
    //获取鼠标位置，这个位置是通过injectMousePosition函数注入系统中的
    ma.position = MouseCursor::getSingleton().getPosition();
    ma.moveDelta = Vector2(0.0f, 0.0f);
    ma.button = button;
    ma.sysKeys = d_sysKeys;
    ma.wheelChange = 0;
  // 找到目标窗口
  Window* dest_window = getTargetWindow(ma.position);
    //产生多重鼠标点击事件，根据按钮获取相应的MouseClickTracker结构
    MouseClickTracker& tkr = d_clickTrackerPimpl->click_trackers[button];
    //累加鼠标点击次数
    tkr.d_click_count++;
  // 重设结构的条件1.超出双击时间，2.鼠标位置不在限定区域内，3.目标窗口改变，
    //4.单件次数大于3，CEGUI最大支持3击
  if ((tkr.d_timer.elapsed() > d_dblclick_timeout) ||
    (!tkr.d_click_area.isPointInRect(ma.position)) ||
    (tkr.d_target_window != dest_window) ||
    (tkr.d_click_count > 3))
  {
    //重设为单击
    tkr.d_click_count = 1;
    // 创建新的点击区域，供下次鼠标按下是检查
    tkr.d_click_area.setPosition(ma.position);
    tkr.d_click_area.setSize(d_dblclick_size);
    tkr.d_click_area.offset(Point(-(d_dblclick_size.d_width / 2), -(d_dblclick_size.d_height /
    2)));
   // 重设目标窗口
    tkr.d_target_window = dest_window;
  }
    // 设置单击次数到事件参数中
    ma.clickCount = tkr.d_click_count;
    // 循环直到消息被处理或者找不到目标窗口
    while ((!ma.handled) && (dest_window != 0))
    {
        ma.window = dest_window;
        //如果窗口需要多重点击，必须调用setWantsMultiClickEvents设置关注
    if (dest_window->wantsMultiClickEvents())
    {
       switch (tkr.d_click_count)
       {
       case 1:
          dest_window->onMouseButtonDown(ma);
          break;
       case 2:
          dest_window->onMouseDoubleClicked(ma);
          break;
       case 3:
          dest_window->onMouseTripleClicked(ma);
```

```
            break;
        }
    }
    // 如果当前窗口不需要多重点击，则全部认为是单击
    else
    {
        dest_window->onMouseButtonDown(ma);
    }
            //查找下一个目标窗口
            dest_window = getNextTargetWindow(dest_window);
    }
    // 重设定时器
    tkr.d_timer.restart();
    return ma.handled;
}
//注入鼠标抬起的事件
bool System::injectMouseButtonUp(MouseButton button)
{
    //取消系统记录的鼠标按下的标志，在鼠标按下时记录的
    d_sysKeys &= ~mouseButtonToSyskey(button);
    MouseEventArgs ma(0);
    ma.position = MouseCursor::getSingleton().getPosition();
    ma.moveDelta = Vector2(0.0f, 0.0f);
    ma.button = button;
    ma.sysKeys = d_sysKeys;
    ma.wheelChange = 0;
  // 获取多重单击的跟踪结构
  MouseClickTracker& tkr = d_clickTrackerPimpl->click_trackers[button];
  // 设置单击次数到事件参数中
  ma.clickCount = tkr.d_click_count;
    //获取目标窗口，根据鼠标位置
  Window* const initial_dest_window = getTargetWindow(ma.position);
    Window* dest_window = initial_dest_window;
    // 循环激发窗口的鼠标抬起事件，指定事件被处理或者找不到目标窗口
    while ((!ma.handled) && (dest_window != 0))
    {
            ma.window = dest_window;
            dest_window->onMouseButtonUp(ma);
            dest_window = getNextTargetWindow(dest_window);
    }
    bool wasUpHandled = ma.handled;
  // 如果满足鼠标单击的条件1.时间小于单击的最长逝去时间，2.鼠标在跟踪区域内，
    //3.目前的跟踪窗口和目标窗口相同
  if ((tkr.d_timer.elapsed() <= d_click_timeout) &&
    (tkr.d_click_area.isPointInRect(ma.position)) &&
    (tkr.d_target_window == initial_dest_window))
  {
            ma.handled = false;
    dest_window = initial_dest_window;
            // 循环直到事件被处理或者找不到目标窗口
            while ((!ma.handled) && (dest_window != 0))
            {
```

```
                ma.window = dest_window;
                dest_window->onMouseClicked(ma);
                dest_window = getNextTargetWindow(dest_window);
            }

        }
        return (ma.handled | wasUpHandled);
}
//注入鼠标滚动消息
bool System::injectMouseWheelChange(float delta)
{
        //设置鼠标滚动事件的参数
        MouseEventArgs ma(0);
        ma.position = MouseCursor::getSingleton().getPosition();
        ma.moveDelta = Vector2(0.0f, 0.0f);
        ma.button = NoButton;
        ma.sysKeys = d_sysKeys;
        ma.wheelChange = delta;
        ma.clickCount = 0;
        Window* dest_window = getTargetWindow(ma.position);
        // 循环执行，直到事件被处理或者没有目标窗口
        while ((!ma.handled) && (dest_window != 0))
        {
            ma.window = dest_window;
            dest_window->onMouseWheel(ma);
            dest_window = getNextTargetWindow(dest_window);
        }
        return ma.handled;
}
//注入鼠标位置信息
bool System::injectMousePosition(float x_pos, float y_pos)
{
        // 看到了吧，这里设置鼠标的位置到MouseCursor中保存起来
        MouseCursor::getSingleton().setPosition(Point(x_pos, y_pos));
        // 注入鼠标移动消息，并传递移动的位置为（0，0），可以不调用这个函数吗？
        return injectMouseMove(0, 0);
}
//上面6个函数使用到来两个获取目标才看到函数，读者一定希望了解他们是如何工作的
//获取鼠标下面的目标窗口的函数
Window*System::getTargetWindow(const Point& pt) const
{
        Window* dest_window = 0;
        // 如果找不到所有窗口的父窗口（CEGUI中称为底板），则无法找到目标窗口，因此这
//个窗口一定要设置，否则CEGUI无法正常工作，它的类型可以是任何Window的子类
        if (d_activeSheet)
        {
            //首先得到捕获窗口输入的窗口
            dest_window = Window::getCaptureWindow();
            //如果找不到则获取鼠标下的最高层的子窗口
            if (!dest_window)
            {
                dest_window = d_activeSheet->getTargetChildAtPosition(pt);
```

```
            //如果还是找不到，则默认窗口底板为鼠标下窗口
            if (!dest_window)
            {
                dest_window = d_activeSheet;
            }
        }
        //如果有捕获鼠标输入的窗口
        else
        {
            //如果自己不出来，而是需要子窗口处理则获取窗口下的目标子窗口
      if (dest_window->distributesCapturedInputs())
      {
        Window* child_window = dest_window->getTargetChildAtPosition(pt);
                //如果找到了则设置，如果没找到目标窗口还是捕获输入的窗口
        if (child_window)
        {
          dest_window = child_window;
        }
      }
        }
        // 如果模态窗口不为空则模态窗口重写目标鼠标
        if (d_modalTarget != 0 && dest_window != d_modalTarget)
        {
            //如果模态窗口不是当前目标窗口的祖先则设置模态窗口为当前目标窗口
            if (!dest_window->isAncestor(d_modalTarget))
            {
                dest_window = d_modalTarget;
            }
        }
    }
    return dest_window;
}
//获取下一个目标窗口
Window* System::getNextTargetWindow(Window* w) const
{
    //如果没有父链递归到模态窗口则继续顺着父链调用下去
    if (w != d_modalTarget)
    {
        return w->getParent();
    }

    // 否则返回空，终止各种inject函数中的循环调用
    return 0;
}
```

getNextTargetWindow函数如果当前没有模态窗口则会沿着第一个目标窗口的父链一直调用下去直到底板窗口为止，如果有模态窗口，则只递归调用到模态窗口就终止。

鼠标注入的消息就介绍完了，消息的派遣在注入函数中已经实现。现在知道Window类的各种鼠标响应消息的驱动源了吧。下面介绍键盘的注入消息。

```
//注入键盘按下，key_code是CEGUI定义的键盘扫描码
bool System::injectKeyDown(uint key_code)
{
```

```
    // 更新系统键的按下状态（Ctrl，Shift，Alt三个键的按下状态）
    d_sysKeys |= keyCodeToSyskey((Key::Scan)key_code, true);
    KeyEventArgs args(0);
    if (d_activeSheet)
    {
        args.scancode = (Key::Scan)key_code;
        args.sysKeys = d_sysKeys;
        //获取键盘的目标窗口
        Window* dest = getKeyboardTargetWindow();
        // 循环调用
        while ((dest != 0) && (!args.handled))
        {
            args.window = dest;
            dest->onKeyDown(args);
            dest = getNextTargetWindow(dest);
        }
    }
    return args.handled;
}
//注入键抬起消息
bool System::injectKeyUp(uint key_code)
{
    // 清除系统键的按下状态
    d_sysKeys &= ~keyCodeToSyskey((Key::Scan)key_code, false);
    KeyEventArgs args(0);
    if (d_activeSheet)
    {
        args.scancode = (Key::Scan)key_code;
        args.sysKeys = d_sysKeys;
        Window* dest = getKeyboardTargetWindow();
        //循环调用
        while ((dest != 0) && (!args.handled))
        {
            args.window = dest;
            dest->onKeyUp(args);
            dest = getNextTargetWindow(dest);
        }
    }
    return args.handled;
}
//注入字符的消息，字符键被按下时需要调用
bool System::injectChar(utf32 code_point)
{
    KeyEventArgs args(0);
    if (d_activeSheet)
    {
        args.codepoint = code_point;
        args.sysKeys = d_sysKeys;
        Window* dest = getKeyboardTargetWindow();
        // 循环执行
        while ((dest != 0) && (!args.handled))
        {
```

```
                args.window = dest;
                dest->onCharacter(args);
                dest = getNextTargetWindow(dest);
            }
        }
        return args.handled;
}
//获取目标键盘窗口的两个函数
//获取第一个键盘目标窗口，如果编游戏的话这个函数可能需要修改
Window* System::getKeyboardTargetWindow(void) const
{
        Window* target = 0;
        //如果没有模态窗口，则获取底板的激活子窗口作为键盘的目标窗口
        if (!d_modalTarget)
        {
            target = d_activeSheet->getActiveChild();
        }
        //否则获取模态窗口的目标子窗口
        else
        {
            target = d_modalTarget->getActiveChild();
            //如果找不到激活子窗口则设置为模态窗口自己
            if (!target)
            {
                target = d_modalTarget;
            }
        }
        return target;
}
//获取下一个目标键盘窗口
Window* System::getNextTargetWindow(Window* w) const
{
        // 类似鼠标的获取下一个目标窗口函数
        if (w != d_modalTarget)
        {
            return w->getParent();
        }
        // 返回空窗口，终止循环调用
        return 0;
}
```

最后一种注入函数是，更新CEGUI系统中所有窗口的消息函数。这个函数给CEGUI所有窗口提供了一种更新机制，如果没有这个机制，则被隐藏的窗口是无法获得执行权的。因为被隐藏的窗口不会被调用渲染函数。窗口的update函数在3.2.7已经介绍过这里就不在赘述。

```
//注入系统时间脉冲，可以用来实现定时更新所有窗口，包括被隐藏的窗口
bool System::injectTimePulse(float timeElapsed)
{
        if (d_activeSheet)
        {
            d_activeSheet->update(timeElapsed);
        }
        return true;
```

```
}
```

这一小节介绍了键盘，鼠标和脉冲更新的系统注入函数。一般这些函数是由游戏程序调用的，通过注入这些数据CEGUI与游戏程序交互。

4.2 资源管理

本节介绍CEGUI中的资源管理，以及图像集。

4.2.1资源管理模式

资源管理也是CEGUI中非常重要的一部分，这里的资源管理指的并不是资源提供模块。而是CEGUI中非常重要的Scheme类以及它的管理器类。Scheme对应一个XML格式的数据文件（.scheme文件）包含了这个系统中用到的各种数据的定义，比如字体信息，图像集信息，窗口外观信息，类型映射等。下面是简化的一段Scheme文件的定义，它来自CEGUI自带例子程序的资源目录。

```
<?xml version="1.0" ?>
<GUIScheme Name="TaharezLook">
    <Imageset Name="TaharezLook" Filename="TaharezLook.imageset" />
    <Font Name="Commonwealth-10" Filename="Commonwealth-10.font" />
    <LookNFeel Filename="TaharezLook.looknfeel" />
    <WindowRendererSet Filename="CEGUIFalagardWRBase" />
    <FalagardMapping WindowType="TaharezLook/Tree" TargetType="CEGUI/Tree"
    Renderer="Falagard/Tree"   LookNFeel="TaharezLook/Tree" />
</GUIScheme>
```

GUIScheme 是根元素，它代表一个Scheme，TaharezLook是它的名称。Imageset 是图像集的定义元素，Name是图像集的名称，TaharezLook.imageset是图像集对应的图片文件全路径。Font是字体的定义元素，Commonwealth-10是字体元素的名称，Commonwealth-10.font是字体定义文件，文件里面定义了系统内可以使用的字体信息。LookNFeel是窗口外观定义的元素，TaharezLook.looknfeel是外观文件的全路径，这个文件非常复杂，第6章会详细介绍。WindowRendererSet是动态渲染模块的名称，还记得前文讲述过的动态模块加载吗？这里定义了动态库的名称CEGUIFalagardWRBase，它是渲染窗口模块的动态链接库。

FalagardMapping 定义一个外部名称到内部名称，渲染类型和外观的映射。WindowType是外部名称的定义元素，TargetType是内部窗口类型的定义元素，Renderer是渲染窗口名称的定义元素，LookNFeel是外观的定义元素。创建窗口只需要外部窗口名称和窗口的名称就可以了。（名称亦可以省略，使用系统的自动名称）。

读者应该发现了，这个文件是联系各个CEGUI功能文件的总控文件，所以它非常重要。这里简单的介绍字体文件。它也是来自CEGUI例子的资源。

```
<?xml version="1.0" ?>
<Font Name="DejaVuSans-10" Filename="DejaVuSans.ttf" Type="FreeType" Size="10"
NativeHorzRes="800" NativeVertRes="600" AutoScaled="true"/>
```

DejaVuSans-10是这个字体的名称，设置字体的时候使用这个名称，DejaVuSans.ttf是这个字体对应的字体文件，FreeType表示这个字体是矢量字体，不是位图字体，10代表字体的大小。800和600代表原始解析度，true代表容许自动缩放。

> 注意：
> Scheme和font文件中的各个元素是可以有许多个的，这里为了简便只列举了其中一个。比如font文件可以有多个Font 的定义，scheme可以有多个FalagardMapping ， Font ， Imageset ， WindowRendererSet ， LookNFeel 的定义。还有font文件里的Font 与Scheme文件里的Font是两个元素，前者定义一个真实的字体，后者定义一个字体文件，里面内容是前者的定义。

资源文件先介绍到这里，下面介绍详细的Scheme的代码。我们说过Scheme类是负责加载系统需要的各种资源的。所以它的成员变量大都是和加载资源相关的。

```
//定义加载一般资源的结构，这个结构可以用来加载图像集，字体，外观文件
struct LoadableUIElement
{
```

```
    String      name;                       //具体元素的名字（比如字体名称）
    String      filename;                   //文件名称（一般是XML格式的配置文件）
  String  resourceGroup;                    //filename所在的资源组
};
//动态库中的类厂名称结构
struct      UIElementFactory
{
    String name;                            //类厂名称
};
//代表一个动态库加载所需要的信息
struct      UIModule
{
    String name;
    FactoryModule*     module;
    std::vector<UIElementFactory>     factories;
};
//别名映射
struct AliasMapping
{
    String aliasName;
    String targetName;
};

//窗口的外部名称到窗口内部名称，渲染窗口类厂名称，和外观定义的映射
struct FalagardMapping
{
  String windowName;
  String targetName;
  String rendererName;
  String lookName;
};
//定义Scheme的名称，系统可以支持加载多个Scheme
String      d_name;
//各种资源的数组定义
std::vector<LoadableUIElement>          d_imagesets;
std::vector<LoadableUIElement>          d_imagesetsFromImages;
std::vector<LoadableUIElement>          d_fonts;
std::vector<UIModule>                   d_widgetModules;
std::vector<UIModule>               d_windowRendererModules;
std::vector<AliasMapping>               d_aliasMappings;
std::vector<LoadableUIElement>          d_looknfeels;
std::vector<FalagardMapping>        d_falagardMappings;
//保存默认的资源组
static String d_defaultResourceGroup;
```

我们前面讲过，所有XML格式文件的处理方法都是类似的，Scheme文件也是XML文件，所以它也有个处理类。Scheme的构造函数，调用解析器分析scheme文件，处理类将各种信息保存在Scheme类中定义的各种资源数组中。当解析完scheme文件后，这些数组里保存了这个shceme文件中定义的所有资源信息。Scheme的构造函数定义如下：

```
Scheme::Scheme(const String& filename, const String& resourceGroup)
{
    // 省略一些参数检查，下面是加载XML文件，然后分析并处理
    Scheme_xmlHandler handler(this);            //创建处理类的实例
```

```
    try
    {
    System::getSingleton().getXMLParser()->parseXMLFile(
        handler, filename, GUISchemeSchemaName,
        resourceGroup.empty() ? d_defaultResourceGroup : resourceGroup);
    }
    catch(...)
    {
   //异常处理省略
    }
    // 尝试加载资源
    loadResources();
}
```

看来Scheme最重要的函数就是各种加载资源的函数了。loadResources函数调用了各种加载资源的函数，这里就不介绍它了，而是直接介绍各种加载资源的函数。

Scheme提供了图像资源的加载，字体资源的加载，窗口类工厂，渲染窗口类工厂，外观定义，窗口类型映射，工程别名等。

首先介绍图资源的加载，CEGUI提供了两种图像资源的定义方法，一种是直接定义一个图像集文件名称和资源组（它的图像集名称在图像集文件中已经指定了），另一种除了第一种的两个参数外还有一个自定义的图像集名称。这里只介绍第二种加载函数loadXMLImagesets，第一种类似（loadImageFileImagesets），读者自己阅读。

```
void Scheme::loadXMLImagesets()
{
    //获取图像集的管理器
  ImagesetManager& ismgr = ImagesetManager::getSingleton();
  std::vector<LoadableUIElement>::const_iterator  pos;
  // 变量所有的图像集资源的定义
  for (pos = d_imagesets.begin(); pos != d_imagesets.end(); ++pos)
  {
    // 如果图像集不存在，则创建
    if (!ismgr.isImagesetPresent((*pos).name))
    {
      // 通过图像集管理器以及图像集定义XML文件名称和资源组创建图像集
      Imageset* iset = ismgr.createImageset((*pos).filename, (*pos).resourceGroup);
      // 检查图像集的名称（imageset中定义的）和（scheme中定义的）是否一致
      String realname = iset->getName();
      if (realname != (*pos).name)
      {
        ismgr.destroyImageset(iset);
        //这里抛出异常
      }
    }
  }
}
```

读者可能无法理解什么是图像集，关于各种资源的具体含义，以及他们的介绍将在后面几个小节中详细介绍。下面介绍字体的加载。

```
void Scheme::loadFonts()
{
    //获取字体管理器
  FontManager& fntmgr = FontManager::getSingleton();
```

```
    std::vector<LoadableUIElement>::const_iterator  pos;
    for (pos = d_fonts.begin(); pos != d_fonts.end(); ++pos)
    {
        // 如果字体还没有加载则加载之
        if (!fntmgr.isFontPresent((*pos).name))
        {
            // 从指定的字体定义XML文件中加载字体，从这里可以看出这个文件可以定
                //义多个字体
            Font* font = fntmgr.createFont((*pos).filename, (*pos).resourceGroup);
            // 检查名字是否一致
            String realname = font->getProperty ("Name");
            if (realname != (*pos).name)
            {
                fntmgr.destroyFont(font);
                //抛出异常，这里省略
            }
        }
    }
}
```

字体是CEGUI非常重要的部分，它担负着文字显示的各种计算任务和显示任务以及字体贴图的管理任务。这是一个非常重要而且复杂的主题，我们将在第8章详细介绍。下面介绍窗口外观定义的加载。

```
void Scheme::loadLookNFeels()
{
    //获取外观定义管理器
    WidgetLookManager& wlfMgr   = WidgetLookManager::getSingleton();
    std::vector<LoadableUIElement>::const_iterator  pos;
    //加载外观定义文件，这个文件里面定义了许多窗口的外观
    for (pos = d_looknfeels.begin(); pos != d_looknfeels.end(); ++pos)
        wlfMgr.parseLookNFeelSpecification((*pos).filename, (*pos).resourceGroup);
}
```

动态库的加载有两类，一类是窗口类厂，一类是渲染窗口类厂。我们这里只介绍渲染窗口的类厂加载。

```
void Scheme::loadWindowRendererFactories()
{
    //获取渲染类厂管理器
    WindowRendererManager& wfmgr = WindowRendererManager::getSingleton();
    std::vector<UIModule>::iterator cmod = d_windowRendererModules.begin();
    for (;cmod != d_windowRendererModules.end(); ++cmod)
    {
        // 创建并且加载动态模块（在Window上是dll文件）
        if (!(*cmod).module)
        {
            (*cmod).module = new FactoryModule((*cmod).name);
        }
            //如果没有指定具体注册那些渲染类厂，则全部注册
        if ((*cmod).factories.size() == 0)
        {
          (*cmod).module->registerAllFactories();
        }
        // 如果指定了一些需要注册的渲染类厂，则只注册这些
        else
```

```
    {
        std::vector<UIElementFactory>::const_iterator   elem =
                                (*cmod).factories.begin();
        for (; elem != (*cmod).factories.end(); ++elem)
        {
            if (!wfmgr.isFactoryPresent((*elem).name))
            {
                (*cmod).module->registerFactory((*elem).name);
            }
        }
    }
  }
}
```

动态模块我们前文讲过，这里就不做介绍了。CEGUI提供了这些模块的加载支持，相当于插件一样，只有读者定义了一个动态库需要导出的函数，就可以把这个动态库当一个插件来使用。最后介绍映射类型的加载。

```
void Scheme::loadFalagardMappings()
{
    //获取管理器
  WindowFactoryManager& wfmgr = WindowFactoryManager::getSingleton();
  std::vector<FalagardMapping>::iterator falagard = d_falagardMappings.begin();
  for (;falagard != d_falagardMappings.end(); ++falagard)
  {
    // 获取类厂映射的迭代器
    WindowFactoryManager::FalagardMappingIterator iter =
                                wfmgr.getFalagardMappingIterator();
    // 检查是否已经注册过了，其实完全没有必要检查
    while (!iter.isAtEnd() && (iter.getCurrentKey() != (*falagard).windowName))
        ++iter;
    // 如果已经注册过了
    if (!iter.isAtEnd())
    {
        // 检查和先前定义的是否一致，如果一致则继续，否则新的取代旧的
        if ((iter.getCurrentValue().d_baseType == (*falagard).targetName) &&
            (iter.getCurrentValue().d_rendererType == (*falagard).rendererName) &&
            (iter.getCurrentValue().d_lookName == (*falagard).lookName))
        {
            // assume this mapping is ours and skip to next
            continue;
        }
    }
    // 创建一个新的，如果存在相同的则新的代替旧的
    wfmgr.addFalagardWindowMapping((*falagard).windowName, (*falagard).targetName,
    (*falagard).lookName, (*falagard).rendererName);
  }
}
```

好了，资源加载就介绍到这里。Scheme类还提供了资源的unload，这里就不做介绍了。还有一些不常用的比如别名加载，这里就不介绍了。既然Scheme在系统里也可以有多个，哪么它就需要一个管理器。CEGUI提供了SchemeManager来管理Scheme，这个类非常简单相信读者完全可以看懂，这里也不介绍了。Scheme是如何被加载的呢？相信读者阅读了4.1.2节应该知道，它是什么时候被加载的，如何被加载的。

4.2.2 图像集

图像集是CEGUI中与渲染接口非常相关的一个部分。图像集（ImageSet）和图像（Image）究竟代表什么呢？图像集可以对应一张图片（这个图片一般是正方形，而且尺寸是2的幂）。Image对应ImageSet上的一块区域。为什么要这么设计呢？每一张图片代表一个Image不可以吗？我们知道显卡处理2的次幂的图片是最有效的，一般贴图都必须是2的次幂，而且一般都是正方形。界面需要的图像经常是长方形，而且大小不一，如果放在一个正方形图片文件上则会浪费许多图片空间（许多空间是无效的）。一般将许多小的窗口需要的图片拼在一张贴图上。也就是说一个窗口对应的图片其实是在一张纹理上的一部分。我们可以认为ImageSet就是一张纹理贴图，Image就是这张贴图上的一块窗口需要的区域。如图4-1所示。图像集和图像之间的关系。图像集A代表整张纹理，图像B代表一个（X1，Y1）（X2，Y2）构成的矩形。在CEGUI提供的例子的数据目录下面有图像集的配置文件。我们从文档里摘出一段来，和读者一起分析。

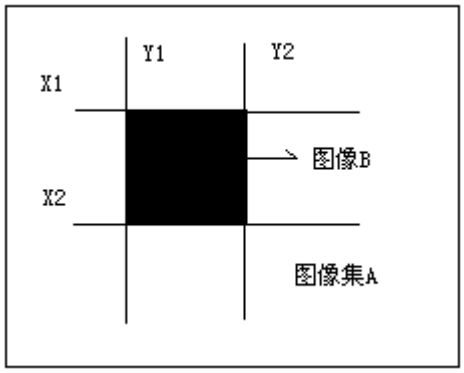

图4-1图像集和图像

imageset文件也是标准的XML格式文件。Imageset 元素标识一个图像集，而且整个文件只能定义一个图像集。这个图像集的名称是TaharezLook，一般和文件名相同（TaharezLook.imageset)。NativeHorzRes和NativeVertRes表示图像的原始分辨率，AutoScaled设置贴图是否可以自动拉伸。Image 元素定义一个图像，它由名称，X，Y坐标以及高和宽组成。高等于图4-1中的（X2-X1），宽等于图4-1中的（Y2-Y1）。有了位置和高宽就可以计算图像的纹理坐标了，在渲染窗口的时候纹理坐标就是通过这四个参数计算而来。一个图像集里定义多少图像是没有限制的，定义多少都可以。

```
<?xml version="1.0" ?>
    <Imageset Name="TaharezLook" Imagefile="TaharezLook.tga" NativeHorzRes="800"
    NativeVertRes="600" AutoScaled="true">
  <Image Name="ClientBrush" XPos="2" YPos="2" Width="64" Height="64" />
  <Image Name="GenericBrush" XPos="9" YPos="71" Width="1" Height="1" />
</Imageset>
```

下面先介绍图像集，从它的成员变量开始。

```
String      d_name;                                  //保存图像集的名称
typedef    std::map<String, Image, String::FastLessCompare>     ImageRegistry;
ImageRegistry d_images;                             //定义图像名称到图像之间的映射变量
Texture*        d_texture;                          //图像集对应的贴图的指针
String      d_textureFilename;        //保存贴图对应的文件全路径
bool d_autoScale;                                   //是否容许自动缩放
float d_horzScaling;                                //竖直缩放时的缩放因子
float d_vertScaling;                                //水平缩放时的缩放因子
float d_nativeHorzRes;                              //元素水平解析度
float d_nativeVertRes;                              //元素竖直解析度
static String d_defaultResourceGroup;  //图像集对应的资源组
```

成员变量简单易懂，可能Texture类读者不熟悉，不过从它的名称可以知道，它代表贴图对象。这是渲染接口必须实现的接口之一，另一个是渲染接口，他们将会在第9章详细介绍。我们只介绍三个成员函数，从imageset文件中加载图像集的函数load，在图像集中定义一个
图像defineImage，描绘这个图像draw。

```
void Imageset::load(const String& filename, const String& resourceGroup)
{
    // unload 旧的数据
    unload();
  // 创建XML的处理器
  Imageset_xmlHandler handler(this);
    //解析XML格式的文件
    try
    {
    System::getSingleton().getXMLParser()->parseXMLFile(
        handler, filename, ImagesetSchemaName,
        resourceGroup.empty() ? d_defaultResourceGroup : resourceGroup);
    }
    catch(...)
    {
        unload();
    throw;
    }
}
//在图像集中定义一块区域作为图像，一般在动态创建图像集时定义图像（比如字体贴图）
void Imageset::defineImage(const String& name, const Rect& image_rect, const Point& render_offset)
{
    // 获取缩放因子
    float hscale = d_autoScale ? d_horzScaling : 1.0f;
    float vscale = d_autoScale ? d_vertScaling : 1.0f;
    // 创建图像，并且保存到映射中
    d_images[name] = Image(this, name, image_rect, render_offset, hscale, vscale);
}
//最重要的可能就是draw描绘函数了
void Imageset::draw(const Rect& source_rect, const Rect& dest_rect, float z, const Rect& clip_rect,const ColourRect& colours,
QuadSplitMode quad_split_mode) const
{
    // 获取裁剪后的目标区域getIntersection函数获取两个Rect的交集
    Rect final_rect(dest_rect.getIntersection(clip_rect));
    // 检查是否最终区域全部被裁剪掉了
    if (final_rect.getWidth() != 0)
    {
    //获取正确的缩放因子
    const float x_scale = d_texture->getXScale();
    const float y_scale = d_texture->getYScale();
        //下面这段计算纹理坐标的算法比较难理解
        float tex_per_pix_x = source_rect.getWidth() / dest_rect.getWidth();
        float tex_per_pix_y = source_rect.getHeight() / dest_rect.getHeight();
        // 纹理坐标计算的复杂性是由裁剪器引起的
        Rect  tex_rect((source_rect.d_left + ((final_rect.d_left - dest_rect.d_left) * tex_per_pix_x)) * x_scale,
            (source_rect.d_top + ((final_rect.d_top - dest_rect.d_top) * tex_per_pix_y)) * y_scale,
```

```
                (source_rect.d_right + ((final_rect.d_right - dest_rect.d_right) * tex_per_pix_x)) * x_scale,
                (source_rect.d_bottom + ((final_rect.d_bottom - dest_rect.d_bottom) * tex_per_pix_y)) * y_scale);
        //PixelAligned宏将一个浮点数转化成整形
        final_rect.d_left      = PixelAligned(final_rect.d_left);
        final_rect.d_right     = PixelAligned(final_rect.d_right);
        final_rect.d_top       = PixelAligned(final_rect.d_top);
        final_rect.d_bottom= PixelAligned(final_rect.d_bottom);
        //添加到渲染接口的渲染队列中
        d_texture->getRenderer()->addQuad(final_rect, z, d_texture, tex_rect, colours,
        quad_split_mode);
    }
}
```

由于裁减器的作用，最终描绘的区域final_rect的区域小于等于dest_rect。如果final_rect等于dest_rect则最终的纹理坐标等于source_rect也就是说图像没有被裁剪，使用图像原始的区域作为纹理坐标。否则计算两个区域的每条边之间的差，比如左边就是final_rect.d_left - dest_rect.d_left。我们以左边举例，这个差记为diff。source_rect是图像（Image）定义的原始区域，dest_rect是目标窗口的区域。tex_per_pix_x计算出了目标窗口一个单位（一般是像素）对应（Image）也就是贴图多少值。diff表示的窗口区域的差值，它乘以tex_per_pix_x可以得到diff对应的Image的多少单位（记为imagediff)。source_rect.d_left表示元素的Image的单位，在加上imagediff，就得出了贴图的纹理坐标的未缩放前的值，最后乘以图像集的缩放因子就可以得到最终的左边的纹理坐标了。当计算右边的时候final_rect.d_right - dest_rect.d_right小于零，所以最终计算出的结果小于source_rect.d_right。

图像（Image）的实现比较简单，这里简单介绍。成员变量如下所示：

```
const Imageset*     d_owner;        //指向图像集的指针，用来访问它提供的方法
Rect                d_area;         //在图像集中的区域
Point               d_offset;       //渲染时的偏移值
float d_scaledWidth;                //缩放图像的宽度
float d_scaledHeight;               //缩放图像的高度
Point       d_scaledOffset;         //渲染偏移的缩放
String      d_name;                 //图像的名称
```

只介绍一个函数draw，这个函数调用图像集的对应函数实现。

```
void Image::draw(const Rect& dest_rect, float z, const Rect& clip_rect, const ColourRect& colours, QuadSplitMode quad_split_mode)
const
{
    Rect dest(dest_rect);
    // 应用渲染偏移
    dest.offset(d_scaledOffset);
    // 调用图像集描绘函数
    d_owner->draw(d_area, dest, z, clip_rect, colours, quad_split_mode);
}
```

图像集也需要管理器，它负责图像集的创建删除等管理操作。管理器在CEGUI中有明显的特点，都是单件类，而且主要负责某种类型的创建，删除，存在判断，从文件中加载等。CEGUI中定义ImagesetManager类管理图像集。我们只介绍创建图像集的函数，读者可以自己阅读其他代码，代码都非常简单清晰。这个类提供了两种创建图像集的方法，前面已经介绍过。createImageset创建一个图像集，图像集的名称必须和XML文件的名称相同（扩展名除外，可以参考4.2.1中loadXMLImagesets函数）。createImagesetFromImageFile则在参数中指定图像集名称和图像集文件的名称没有关系。

```
Imageset* ImagesetManager::createImageset(const String& filename, const String& resourceGroup)
{
    //创建图像集对象
    Imageset* temp = new Imageset(filename, resourceGroup);
    //判断存在性
```

```
    String      name = temp->getName();
    if (isImagesetPresent(name))
    {
        delete temp;
        //抛出异常
    }
    //保存到映射变量中，方便以后的查找和存在性判断等
    d_imagesets[name] = temp;
    return temp;
}
```

图像集和图像集的管理以及图像就介绍到这里，相信读者已经理解了他们的概念和大部分实现。比较困难的部分应该是经过裁剪后的纹理坐标的计算问题。下一节介绍CEGUI中的系统接口。

4.3 系统接口

CEGUI之所以非常灵活是因为它与外部的通讯采用了接口的模式，比如渲染接口，脚本接口，输入系统接口（各种inject函数），XML处理接口以及图像解码接口等。渲染接口将在第9章详细介绍，输入系统接口已经介绍过了。所以本节只介绍其他四个接口。

4.3.1 脚本接口

脚本接口定义在CEGUIScriptModule.h和CEGUIScriptModule.cpp文件中。脚本接口的类名ScriptModule，CEGUI中文件的命名规则是CEGUI+定义的类名，本书中没有提到的文件名称都具有此规则。接口的定义显然也就是定义一些虚函数。哪么首先我们研究CEGUI脚本模块都定义了那些接口，以及为什么定义他们。

```
//定义许多析构函数，派生子类的析构函数可以被正确的调用到
virtual ~ScriptModule(void) {}
//执行一个脚本文件
virtual void executeScriptFile(const String& filename, const String& resourceGroup = "")= 0;
//执行一个全局函数
virtual int executeScriptGlobal(const String& function_name)     = 0;
//执行CEGUI事件的脚本响应函数
virtual      bool executeScriptedEventHandler(const String& handler_name, const EventArgs& e)
      = 0;
//执行一段脚本代码，这段代码已经在内存中了
virtual void executeString(const String& str) = 0;
//创建CEGUI窗口到脚本的绑定
virtual void createBindings(void) {}
//解除CEGUI窗口到脚本的绑定
virtual void destroyBindings(void) {}
//注册CEGUI事件的脚本执行函数
virtual Event::Connection      subscribeEvent(EventSet* target, const String& name, const String& subscriber_name) = 0;
//同上，只不过参数不同而已
virtual Event::Connection      subscribeEvent(EventSet* target, const String& name, Event::Group group, const String&
subscriber_name) = 0;
```

至于为什么这么定义，有些已经写清楚了，有些是可以合并的比如executeScriptGlobal和executeString其实完全可以合并，前者可以在后者中实现。只不过这样设计比较清晰罢了。读者也可以自己增加或者减少一些函数接口的定义。这个类还有一些成员函数和变量这里就不做介绍了，他们的功能已经非常清楚了。本书第15章将实现一个CEGUI的脚本接口的例子。虽然CEGUI提供了脚本模块的实现，但笔者觉得它比较复杂，所以决定自己编写一个小例子。

头文件CEGUIScriptModule.h中还定义了脚本事件处理函数的注册Functor叫做ScriptFunctor，这个类在前文已经介绍过。相信读者已经明白，这里不做介绍了。

4.3.2 XML相关接口

CEGUI中有许多配置文件，他们功能各异，定义的元素也很不相同，但他们都有一个共同的特点，就是处理方法是类似。读者回顾我们前面介绍的几种文件的处理，是不是都非常类似。先创建一个处理类，然后加载文件。

处理XML文件的有两个接口，XML的处理接口 和XML的解析接口。解析接口负责
各种XML文件的解析。处理接口负责处理解析接口分析的元素和属性。分析接口（XMLParser）主要有初始化和反初始化两个虚接口函数以及一个解析主函数。

```
//初始化解析器
virtual bool initialiseImpl(void) = 0;
//反初始化解析器
virtual void cleanupImpl(void) = 0;
//主要的分析函数入口，handler为处理类的引用
virtual void parseXMLFile(XMLHandler& handler, const String& filename, const String& schemaName, const String& resourceGroup) = 0;
```

initialiseImpl和cleanupImpl函数负责初始化和反初始化XML解析器。parseXMLFile是主分析函数，它是暴露给外部使用最重要的函数。

处理接口（XMLHandler）有三个主要的函数。

```
//当一个元素开始的时候会调用
virtual void elementStart(const String& element, const XMLAttributes& attributes);
//当一个元素结束的时候会介绍
virtual void elementEnd(const String& element);
//当一个元素有文本元素的时候会调用，CEGUI中基本没有使用
virtual void text(const String& text);
```

什么是元素呢？前文已经讲过。这里在详细介绍一下，读者参看下面的代码。

```
<这是一个元素 属性1="属性值1" 属性2="属性值2">
这是元素对应的文本
</这是一个元素>
```

当遇到"<这是一个元素"时解析器会调用elementStart函数，当遇到"</这是一个元素>"时解析器会调用elementEnd。需要注意的是元素的结束亦可以这样定义，在<这是一个元素/>。文本信息在开始和结束之间。当遇到文本信息时解析器text函数。XMLAttributes类包含了这个元素对应的属性（在这个例子里面它包含了两个属性和值对）。

哪么如何加载并处理一个文件呢？首先需要建立一个处理这个文件的处理类，这个类必须派生自XMLHandler接口。然后在加载文件的时候先声明一个处理类的实例，然后调用XML解析模块的主处理函数，就可以处理一类文件了。读者可以参看CEGUI提供的各种XML文件的处理类的实现，如果需要读者可以自己定义一类文件，自己创建一个处理类，实现XML的处理。XML的解析接口，CEGUI系统提供4种不同类型的XML分析器的实现。读者不必关心XML是如何解析的只要使用CEGUI提供的4种解析器之一就可以了。

4.3.3 图像解码接口

图像接口定义类是ImageCodec。这个类定义了一个主要的函数，就是load函数。这个函数从一段内存数据，创建一个Texture对象，这个贴图对象在渲染接口中使用。

```
//加载内存中的图像数据，然后返回渲染接口需要的贴图对象
virtual Texture* load(const RawDataContainer& data, Texture* result) = 0;
```

RawDataContainer是内存中的数据容器，它是资源提供类生成的资源的内存存储结构。资源提供接口从文件系统加载资源到内存并且生成RawDataContainer结构供其他类使用。
图像解码接口比较简单。

4.3.4 资源提供接口

资源提供接口定义在CEGUIResourceProvider.h和CEGUIResourceProvider.cpp中。这个接口主要定义了两个接口，一个是加载文件内容到内存的方法，另一个正好相反，它把加载到内存的数据释放掉（但并不写回文件）。下面是这两个函数的具体定义。

```
//资源加载接口，这个函数是个纯虚函数，必须要实现
virtual void loadRawDataContainer(const String& filename, RawDataContainer& output, const String& resourceGroup) = 0;
//释放加载到内存的数据，这个函数是个虚函数，但不是必须实现的函数
virtual void unloadRawDataContainer(RawDataContainer& data) { }
```

从这两个函数可以看出RawDataContainer类的确是内存数据的容器，它负责存储加载到内存中的数据。这个类如何实现这里就不介绍了，读者把它看成是内存的一段数据就可以了。这里解释资源组的概念，一个资源组可以包含一些已经指定的路径作为这个资源组的搜索路径。(CEGUI实现的默认资源管理器一个资源组只对应一个路径)有点类似Window的环境。当从这个资源组加载数据的时候，不需要提供它的全部路径，只要提供它的部分路径。举个例子，比如General资源组定义了两个搜索目录"../General"和"C:\CEGUI\"。如果C:\CEGUI\目录下有一个test.layout文件，哪么可以加载这个文件通过给filename赋值test.layout，resourceGroup赋值General(如果不赋值则使用默认资源组），就可以加载资源了。如果../General目录下有个子目录Scheme，这个目录下有test.scheme文件则可以通过给filename赋值Scheme/test.scheme，资源组赋值General来实现加载。

CEGUI提供了默认的资源提供类，它就是DefaultResourceProvider。当用户没有提供自己的资源提供类时使用默认的资源提供了。具体实现这里就不做介绍了，有兴趣的读者可以自己阅读代码。

4.4渲染机制

CEGUI的渲染机制比较复杂，设计者为了提高效率并没有采用每帧都更新所有可见窗口的做法，而是提出了渲染缓冲的概念缓冲器中存储了一个窗口的渲染结果。如果这个窗口不需要重绘则直接提交这个结果到渲染接口。CEGUI提供了两层缓冲，一层由渲染接口提供，当系统所有的窗口都不需要重绘的时候，使用渲染接口中缓冲的内容直接描绘。另一层由RenderCache支持，它是窗口级的渲染缓冲。读者还记不记得Window类中有一个成员变量d_renderCache，它就是RenderCache的实例。RenderCache缓冲窗口的图像和文本的描绘，当窗口不需要重绘的时候，直接使用窗口的缓冲描绘到渲染接口的渲染缓冲中，当窗口需要重绘的时候，窗口清理渲染缓冲，重新提交窗口描绘结果到渲染缓冲中。读者可以参考图4-2进一步理解CEGUI的渲染机制。

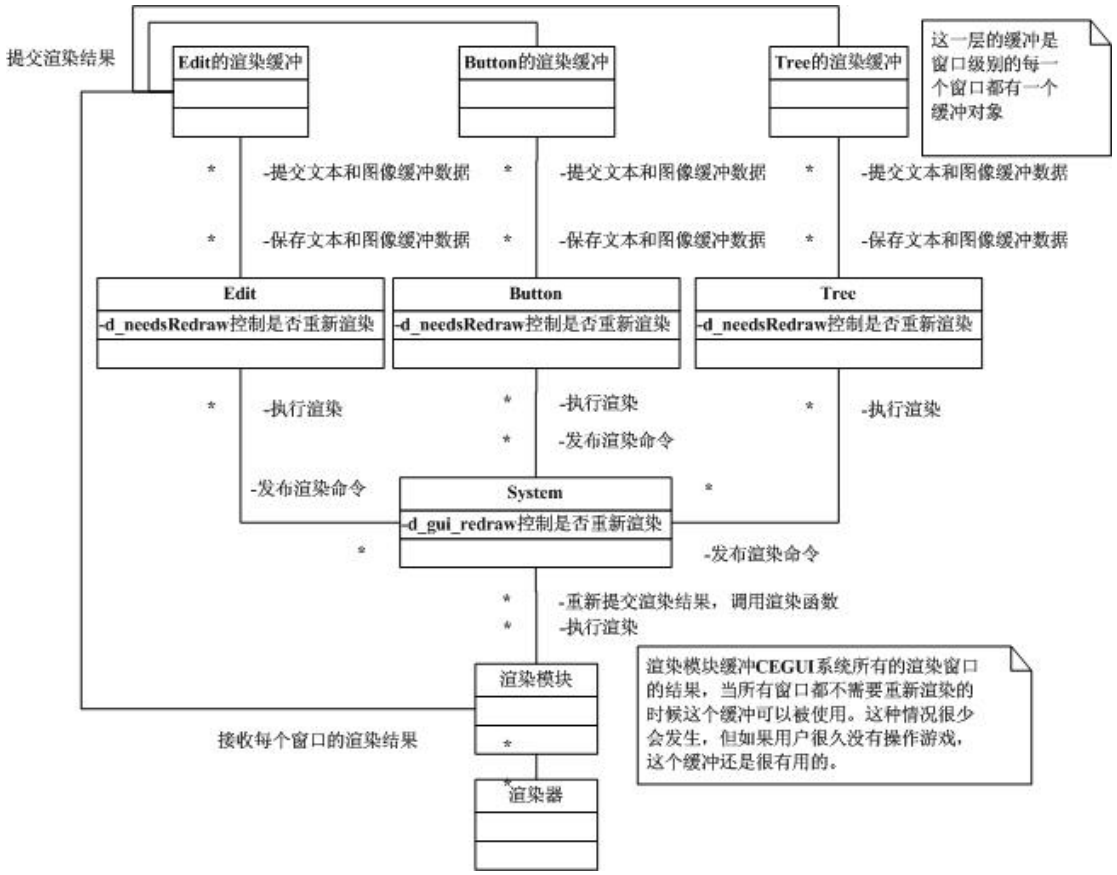

图4-2 渲染机制

图4-2中以三类窗口Edit，Button和Tree为例介绍。所有其他的窗口都和这三个窗口在同一层。而且所作的事情也类似，因为渲染操作是Window窗口提供的。下面首先介绍RenderCache类。这个类的成员变量是两个数组，他们保存了缓冲的内容。

```
//图像缓冲的结构
struct ImageInfo
{
 const Image* source_image;//缓冲的图像的指针
 Rect target_area;          //渲染的目标区域，窗口本地坐标表示的区域
 float z_offset;            //Z值偏移，基本无用，d_drawList（Window）控制的
 ColourRect colours;        //矩形四个顶点的颜色（渲染时被拆分成两个三角形）
 Rect customClipper;        //自定义的裁减器
 bool usingCustomClipper;   //是否使用自定义的裁减器的标志
 bool clipToDisplay;        //是否使用屏幕显示区域裁剪
};
//文本缓冲的结构
struct TextInfo
{
  String text;              //需要渲染的文字
  Font* source_font;        //使用的字体
  TextFormatting formatting; //文本的排列格式
  Rect target_area;         //渲染目标区域也是窗口本地坐标系下的区域
  float z_offset;           //Z值偏移
  ColourRect colours;       //颜色矩形
  Rect customClipper;       //自定义裁减器
```

```
  bool usingCustomClipper;  //是否使用自定义的裁减器
  bool clipToDisplay;               //是否使用屏幕显示区域裁剪
};
//定义两个缓冲数组的类型
typedef std::vector<ImageInfo>  ImageryList;
typedef std::vector<TextInfo>   TextList;
//定义两个数组
ImageryList d_cachedImages;
TextList d_cachedTexts;
```

读者可能会疑惑，target_area保存的是窗口本地坐标系下的位置坐标，不是屏幕坐标系下的位置坐标，而描绘窗口需要屏幕窗口的坐标，CEGUI是怎么实现转化的呢？答案就在
render函数中。下面介绍RenderCache的成员函数。

这个类主要有四个函数。一个渲染函数，提交结果到渲染模块。两个缓冲函数分别缓冲文本和图像。还有一个是渲染缓冲里是否有缓冲内容的判断函数。

```
//渲染缓冲中是否有缓冲的数据
bool RenderCache::hasCachedImagery() const
{
   return !(d_cachedImages.empty() && d_cachedTexts.empty());
}
//渲染函数basePos就是窗口在屏幕坐标系下的偏移，这样窗口本地坐标被转化为屏幕坐标
//clipper是窗口的区域大小，baseZ基本无用
void RenderCache::render(const Point& basePos, float baseZ, const Rect& clipper)
{
      //获取屏幕的渲染区域矩形
   Rect displayArea(System::getSingleton().getRenderer()->getRect());
   Rect custClipper;
   const Rect* finalClipper;
   Rect finalRect;
   // 发送所有缓冲的图像数据到渲染模块
   for(ImageryList::const_iterator image = d_cachedImages.begin(); image !=
      d_cachedImages.end(); ++image)
   {
      if ((*image).usingCustomClipper)
      {
         custClipper = (*image).customClipper;
         custClipper.offset(basePos);
                  //计算裁减器getIntersection是取矩形交集
         custClipper = (*image).clipToDisplay ? displayArea.getIntersection(custClipper) :
            clipper.getIntersection(custClipper);
         finalClipper = &custClipper;
      }
      else
      {
         finalClipper = (*image).clipToDisplay ? &displayArea : &clipper;
      }
            //计算最终在屏幕上的显示区域offset将窗口本地坐标系转化为屏幕坐标
      finalRect = (*image).target_area;
      finalRect.offset(basePos);
            //这一步将图像缓冲到渲染接口
```

```
        (*image).source_image->draw(finalRect, baseZ + (*image).z_offset, *finalClipper,
                (*image).colours);
    }
    //发送所有的文本到渲染接口缓冲
    for(TextList::const_iterator text = d_cachedTexts.begin(); text != d_cachedTexts.end();
        ++text)
    {
        if ((*text).usingCustomClipper)
        {
            custClipper = (*text).customClipper;
            custClipper.offset(basePos);
            custClipper = (*text).clipToDisplay ? displayArea.getIntersection(custClipper) :
                clipper.getIntersection(custClipper);
            finalClipper = &custClipper;
        }
        else
        {
            finalClipper = (*text).clipToDisplay ? &displayArea : &clipper;
        }
        finalRect = (*text).target_area;
        finalRect.offset(basePos);
        (*text).source_font->drawText((*text).text, finalRect, baseZ + (*text).z_offset,
                *finalClipper, (*text).formatting, (*text).colours);
    }
}
//清空渲染缓冲
void RenderCache::clearCachedImagery()
{
    d_cachedImages.clear();
    d_cachedTexts.clear();
}
//缓冲图像到RenderCache的图像数组中
void RenderCache::cacheImage(const Image& image, const Rect& destArea, float zOffset, const ColourRect& cols, const Rect* clipper,
bool clipToDisplay)
{
    ImageInfo imginf;
    imginf.source_image = &image;
    imginf.target_area  = destArea;
    imginf.z_offset     = zOffset;
    imginf.colours      = cols;
    imginf.clipToDisplay = clipToDisplay;
    if (clipper)
    {
        imginf.customClipper = *clipper;
        imginf.usingCustomClipper = true;
    }
    else
    {
        imginf.usingCustomClipper = false;
    }
        //添加到数组
    d_cachedImages.push_back(imginf);
```

```
}
//缓冲文本到文本缓冲数组
void RenderCache::cacheText(const String32& text, Font* font, TextFormatting format, const Rect& destArea, float zOffset, const
ColourRect& cols, const Rect* clipper, bool clipToDisplay)
{
   TextInfo txtinf;
   txtinf.text         = text;
   txtinf.source_font  = font;
   txtinf.formatting   = format;
   txtinf.target_area  = destArea;
   txtinf.z_offset     = zOffset;
   txtinf.colours      = cols;
   txtinf.clipToDisplay = clipToDisplay;
   if (clipper)
   {
      txtinf.customClipper = *clipper;
      txtinf.usingCustomClipper = true;
   }
   else
   {
      txtinf.usingCustomClipper = false;
   }
      //保存到数组
   d_cachedTexts.push_back(txtinf);
}
```

窗口的渲染函数drawSelf（Window的成员函数）具体负责窗口的渲染。它会调用渲染窗口的render函数（如果渲染窗口存在，否则调用populateRenderCache，这个函数里可以调用d_renderCache来缓冲图像和文字。）

System类提供了renderGUI函数，这个函数是渲染的入口函数，也是提供给游戏程序调用的。它负责CEGUI的渲染工作。

```
void System::renderGUI(void)
{
      //如果需要重绘
      if (d_gui_redraw)
      {
            d_renderer->resetZValue();
            //设置渲染模式为缓冲模式，就是现在渲染的所有东西都会被缓冲起来
                  d_renderer->setQueueingEnabled(true);
            //清空渲染接口的所有先前缓冲的内容
            d_renderer->clearRenderList();
            //从根窗口开始递归调用所有的子窗口的渲染函数
            if (d_activeSheet)
            {
                  d_activeSheet->render();
            }
            //下次不在需要重绘，除非窗口需要重绘是通知System
            d_gui_redraw = false;
      }
      //做真正的渲染工作，如果不需要重绘则使用渲染接口模块的缓冲来描绘
      d_renderer->doRender();
      // 描绘鼠标
      d_renderer->setQueueingEnabled(false);
      MouseCursor::getSingleton().draw();
```

```
    // 清理死亡窗口的工作在渲染函数里，主要原因是只有这个函数每帧都会调用
    WindowManager::getSingleton().cleanDeadPool();
}
```

从这个函数的实现上可以发现，如果d_gui_redraw 为false，也就是CEGUI中所有的窗口都不需要重绘则直接调用了doRender函数，这时候描绘的是上次缓冲的内容。也就是第一层的渲染缓冲，它是系统级的缓冲，只要系统里有一个窗口需要重绘则这个缓冲就无效了。显然这个缓冲的效果不会太好。窗口（Window）的render函数在第3章已经详细讲过了。下面介绍渲染窗口，它的基类是WindowRender。它的成员函数非常简单，只有三个。下面先介绍成员变量，知道了类的数据结构再理解实现是非常容易的。

```
Window* d_window;           //保存当前渲染窗口关联的窗口的指针
const String d_name;                //保存渲染窗口（自己）的名称
const String d_class;               //保存窗口的类型，派生类一般使用窗口的事件名字空间
typedef std::vector<Property*> PropertyList;
//定义渲染窗口的属性，当和窗口关联起来时添加到窗口
PropertyList d_properties;
```

最重要的成员函数当然是render了，它负责窗口的渲染工作。这个函数是派生类实现的。其他成员函数大都是为render服务的，他们提供一些基本的和渲染相关的功能。

```
//获取窗口的外观定义类，这个类对于渲染窗口来说非常重要，在第6章详细介绍
const WidgetLookFeel& WindowRenderer::getLookNFeel() const
{
    return WidgetLookManager::getSingleton().getWidgetLook(d_window->getLookNFeel());
}
//获得未剪切的窗口内部矩形，它是屏幕坐标系下的值，类似Window应用程序的客户区
Rect WindowRenderer::getUnclippedInnerRect() const
{
    return d_window->getUnclippedPixelRect();
}
//获得完整的窗口区域，它也是屏幕坐标系下的值，它获取整个窗口的屏幕未裁剪区域
Rect WindowRenderer::getPixelRect() const
{
    return d_window->getPixelRect_impl();
}
//保存渲染窗口的属性值到数组中
void WindowRenderer::registerProperty(Property* property)
{
    d_properties.push_back(property);
}
//当一个渲染窗口和窗口联系起来的时候会被调用，功能是向窗口中添加渲染窗口的属性
void WindowRenderer::onAttach()
{
    PropertyList::iterator i = d_properties.begin();
    while (i != d_properties.end())
    {
        d_window->addProperty(*i);
        ++i;
    }
}
//当一个渲染窗口和窗口断开联系时会被调用，功能是去除窗口内的该渲染窗口注册的属性
void WindowRenderer::onDetach()
```

```
{
   PropertyList::reverse_iterator i = d_properties.rbegin();
   while (i != d_properties.rend())
   {
      d_window->removeProperty((*i)->getName());
      ++i;
   }
}
```

这些函数是WindowRender提供的基本函数，它的派生类会实现它的最主要的虚函数render。这个类还有一些虚函数，需要子类实现（不是必须实现）他们是performChildWindowLayout，子窗口布局函数。onLookNFeelAssigned，onLookNFeelUnassigned外观改变的通知
函数。render函数的实现涉及到了外观的定义，详细参考第6章。

第二层缓冲是窗口级的，每个窗口都有一个RenderCache的成员变量d_renderCache。这个变量可以缓冲这个窗口（注意它并不缓冲子窗口的描绘内容）为了更清晰的介绍，这里再次介绍drawSelf函数。

```
void Window::drawSelf(float z)
{
      //窗口级的是否重新渲染判断
   if (d_needsRedraw)
   {
      // 清理上次渲染的结果
      d_renderCache.clearCachedImagery();
      // 重新缓冲渲染结果，如果有渲染窗口则渲染窗口执行这一步
      if (d_windowRenderer != 0)
      {
         d_windowRenderer->render();
      }
            //否则调用populateRenderCache来缓冲窗口渲染的结果
      else
      {
         populateRenderCache();
      }
      // 重置窗口渲染的标记，下次不需要重绘了
      d_needsRedraw = false;
   }
   // 如果缓冲区里有数据，则发送到渲染接口缓冲，这时候还没有真正送到显卡渲染
   if (d_renderCache.hasCachedImagery())
   {
            //获取渲染的偏移量，这个值用于将窗口坐标的数据转换到屏幕坐标
            Point absPos(getUnclippedPixelRect().getPosition());
      // 计算屏幕坐标的裁减器
      Rect clipper(getPixelRect());
      // 如果窗口没有被完全裁剪则调用绘制函数
      if (clipper.getWidth())
      {
         // 发送到渲染接口缓冲
         d_renderCache.render(absPos, z, clipper);
      }
   }
}
```

哪么什么时候需要重绘呢？如果窗口内容没有改变只是窗口位置改变，则第一层缓冲无效，但第二层缓冲是有效地，因为缓冲的内容没有改变，改变的是窗口描绘的位置。只要重新提交窗口的absPos就可以正确的描绘了。这时候需要调用System的signalRedraw函数设置控制变量d_gui_redraw为真就可以重新描绘窗口了。简单一点的说如果第一层缓冲无效
则调用signalRedraw通知系统重新提交缓冲就可以了。如果窗口的内容改变了，比如窗口的文字改变了，或者窗口的大小改变等，这时窗口内部的缓冲也变得无效了。这就需要重绘这个窗口。这时要调用窗口的请求重绘函数requestRedraw，它定义如下。

```
void Window::requestRedraw(void) const
{
    //设置内部控制是否重绘的变量为真，表示需要重绘（drawSelf中）
  d_needsRedraw = true;
    //请求系统重绘，因为第二层的缓冲改变了，所以第一层肯定无效了
    System::getSingleton().signalRedraw();
}
```

总结，如果窗口内部保存的缓冲有效，也就是窗口的显示数据没有改变，只是位置等其他不影响窗口内部缓冲的改变，则只需要调用系统的signalRedraw就可以了。如果窗口内部的缓冲改变了则只需调用窗口的成员函数requestRedraw就OK了。了解CEGUI的渲染原理对于以后编写控件非常重要，因为控件什么时候重绘以及重绘的深度如何需要理解CEGUI的渲染机制。

4.5 习题

本章介绍了CEGUI的资源管理和CEGUI的各种重要的流程，希望通过本章读者对CEGUI的设计思想以及CEGUI的整体有个比较深入的了解。本章主要介绍的是流程和概念，所以这里只留几道思考题。

1.CEGUI的两层渲染缓冲设计的有没有道理，这样做效率高吗？

2.理解模式(Scheme)类在CEGUI资源管理中的核心地位。

3.理解System类提供的各种功能实现。

4.理解图像集和图像之间的关系，并且可以自己创建一个图像集和图像

5.了解各种CEGUI文件的格式，可以参考CEGUI提供例子程序所附带的资源，在源代码的\Samples\datafiles\目录下。

第5章 CEGUI应用程序框架

读者学习了前四章恐怕还只是了解或者理解的CEGUI中的一些基本原理。如果要求读者编写一个CEGUI的应用程序，我估计大部分读者还是无从下手。所以安排了这一章交给读者如何实现一个简单CEGUI程序。虽然CEGUI提供了许多很好的例子程序。但它的例子程序的结构设计的比较复杂，不易于读者理解。本章将使用OpenGL架设一个简单而且高效的程序框架并且开发一个类似CEGUI例子的小程序。本例的资源使用CEGUI例子附带的。

5.1 OpenGL程序框架

我们的应用程序和CEGUI的库是分开的，应用程序使用单独的工程。CEGUI标准库使用一个工程。如果需要修改CEGUI标准库，则修改这个工程，编译好动态库后在交给应用程序使用。所以应用程序使用的是CEGUI编译好的动态库。其实本书大多数例子都是修改CEGUI库的。

CEGUI提供的例子支持多种应用程序，它设计了应用程序帮助库来辅助应用程序的开发。我们为了更加简便没有采用CEGUI提供的框架，而是自己使用OpenGL设计了一个简单的框架。这一小节我们介绍这个框架。本书所有的例子代码在光盘的"本书例子"目录下。我们修改过的CEGUI工程目录在"本书例子\CEGUI"下，各章的应用程序工程在"本书例子\第X章"目录下，其中X表示第几章。

这个框架主要包括了三个部分，第一，WIN32应用程序框架。第二，OpenGL在Win32下的应用程序框架。第三，CEGUI程序框架。下面分别介绍这三个部分。

5.1.1 WIN32应用程序框架

我们知道C++开发环境下程序的入口点函数是WinMain函数（其他语言也可能是WinMain，事实上可执行文件（PE格式）的入口点并不是WinMain而是C库提供的入口）。搭建一个基本的WIN32应用程序主要需要三步。注册窗口类，创建窗口，然后消息循环等几个步骤。注册窗口类填充一个窗口类的结构，然后调用RegisterClassEx或者RegisterClass就可以完成。创建窗口使用CreateWindowEx或者CreateWindow。消息循环使用GetMessage或者PeekMessage等函数可以实现。注册窗口类的时候有一个非常重要的参数就是窗口过程，这是一个函数的指针，它的定义如下：

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
  LPARAM lParam);
```

在我们的例子中注册窗口类和创建窗口我们在CreateRenderWnd函数中实现。

```
bool CEGUISample::CreateRenderWnd()
{
    //填充注册窗口类的结构
    WNDCLASSEX wd;
    wd.cbClsExtra = 0;
    wd.cbSize = sizeof(wd);
    wd.cbWndExtra = 0;
    wd.hbrBackground = (HBRUSH) GetStockObject(BLACK_BRUSH);
    wd.hCursor = LoadCursor(NULL, IDC_ARROW);
    wd.hIcon   = LoadIcon(NULL, IDI_APPLICATION);
    wd.hIconSm = 0;
    wd.hInstance = g_hInstance;
    //设置窗口过程，WndProc是CEGUISample类的一个静态成员函数
    wd.lpfnWndProc = WndProc;
    //窗口类名称，这个名称要在创建窗口时使用
    wd.lpszClassName = _T("CEGUISampleClass");
    wd.lpszMenuName = NULL;
    wd.style = CS_CLASSDC;
    //注册窗口类，一般来说如果结构填充的正确，这个函数不会失败
    RegisterClassEx(&wd);
    //创建窗口，注意第二个参数，这个函数的参数很多，具体含义参考MSDN
```

```
    g_mainWnd = CreateWindowEx(NULL,_T("CEGUISampleClass"), _T("CEGUI应用程序
框架"),WS_OVERLAPPEDWINDOW | WS_VISIBLE ,0,0,DefaultWidth , DefaultHeight,
NULL,NULL,g_hInstance,NULL);
    //更新窗口
    UpdateWindow(g_mainWnd);
    //显示窗口
    ShowWindow(g_mainWnd, SW_SHOW);
    return g_mainWnd != NULL ? true : false;
}
```

消息循环使用Loop函数实现。

```
void CEGUISample::Loop()
{
    MSG msg;
    //死循环，唯一窗口是如果程序手段WM_QUIT消息
    while(1)
    {
        //获取窗口的消息，从消息队列中移除
        BOOL fMessage = PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE);
        //有消息则处理消息
        if(fMessage)
        {
            if(msg.message == WM_QUIT)
                break;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        //无消息则执行CEGUI的渲染操作，Render函数调用了CEGUI的渲染函数
        else
        {
            Render();
        }
    }
}
```

有了上面的三步窗口可以创建并显示了，但还有一个非常重要的函数窗口过程函数没有介绍。在这个函数中调用了许多CEGUI::System类的输入注入函数。这就是我们第4章介绍的CEGUI的输入接口，游戏应用程序需要通过这些接口来驱动CEGUI。窗口过程函数用来处理所有发送（PostMessage不是SendMessage）到这个窗口上的消息。

```
LRESULT  CEGUISample::WndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg)
    {
    //窗口的大小改变了，需要通知OpenGL框架
    case WM_SIZE:
        {
            if( wParam != SIZE_MAXHIDE && wParam != SIZE_MAXSHOW)
            {
                Height = HIWORD(lParam);            // 窗口的高
                Width  = LOWORD(lParam);            // 窗口的宽
                if(  Height <= 0)  Height = 1;      // 防止被零除
                App.ReSize(Width, Height);          // 通知OpenGL框架
            }
```

```
            }
            break;
        //用户单击了关闭按钮或者系统收到关闭的命令，发送WM_QUIT消息
        case WM_CLOSE:
            PostQuitMessage(0);
            break;
        //注入到CEGUI有字符产生
        case WM_CHAR:
            System::getSingleton().injectChar((utf32)wParam);
            break;
        //注入到CEGUI有键盘键被按下
        case WM_KEYDOWN:
        System::getSingleton().injectKeyDown(ToCEGUIScanCode(wParam, lParam));
            break;
        //注入到CEGUI有键盘的键被抬起
        case WM_KEYUP:
            System::getSingleton().injectKeyUp(ToCEGUIScanCode(wParam, lParam));
            break;
        //当鼠标离开游戏窗口的时候，需要显示或者隐藏鼠标
        case WM_MOUSELEAVE:
            App.mouseLeaves();
            break;
        //当鼠标在非客户端区域移动时，相当于鼠标离开了游戏
        case WM_NCMOUSEMOVE:
            App.mouseLeaves();
            break;
        //鼠标在客户区移动时，注入鼠标移动消息
        case WM_MOUSEMOVE:
            App.mouseEnters();
            System::getSingleton().injectMousePosition((float)(LOWORD(lParam)),
                    (float)(HIWORD(lParam)));
            break;
        //鼠标左键按下，注入到CEGUI系统
        case WM_LBUTTONDOWN:
            System::getSingleton().injectMouseButtonDown(LeftButton);
            break;
        //鼠标左键抬起，注入到CEGUI系统
        case WM_LBUTTONUP:
            System::getSingleton().injectMouseButtonUp(LeftButton);
            break;
        //鼠标右键按下，注入到CEGUI系统
        case WM_RBUTTONDOWN:
            System::getSingleton().injectMouseButtonDown(RightButton);
            break;
        //鼠标右键抬起，注入到CEGUI系统
        case WM_RBUTTONUP:
            System::getSingleton().injectMouseButtonUp(RightButton);
            break;
        //鼠标中键按下，注入到CEGUI系统
        case WM_MBUTTONDOWN:
            System::getSingleton().injectMouseButtonDown(MiddleButton);
            break;
```

```
    //鼠标中键抬起，注入到CEGUI系统
    case WM_MBUTTONUP:
        System::getSingleton().injectMouseButtonUp(MiddleButton);
        break;
    //鼠标滚动的消息WM_MOUSEWHEEL，注入到系统
    case 0x020A:
    System::getSingleton().injectMouseWheelChange(static_cast<float>((short)
    HIWORD(wPara    m)) / static_cast<float>(120));
        break;
    }
    //其他消息调用Windows的默认窗口过程来处理
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

这个函数调用了许多CEGUI的注入函数，来和CEGUI通讯。其中需要注意的是CEGUI定义的各种键盘和鼠标扫描码与Window系统并不相同，所以需要转化为CEGUI的扫描码后才能注入到CEGUI系统中。ToCEGUIScanCode用来做Windows扫描码到CEGUI扫描码的转换。这个函数非常大而且读者可能还要根据需要继续添加其他的扫描码转化代码。这里只简单介绍几个码的转换。

```
CEGUI::Key::Scan CEGUISample::ToCEGUIScanCode(WPARAM wParam, LPARAM lParam)
{
    switch(wParam)
    {
    //退格键，转换为CEGUI定义的退格键
    case VK_BACK:
        return CEGUI::Key::Backspace;
        break;
    //删除键
    case VK_DELETE:
        return CEGUI::Key::Delete;
        break;
    //ESC键
    case VK_ESCAPE:
        return CEGUI::Key::Escape;
        break;
    //这里可以继续添加转换，如果读者需要，比如~号。
    default:
        return (CEGUI::Key::Scan )(CEGUI::utf32) wParam;
        break;
    }
}
```

win32的应用程序框架，就简单的介绍到这里，读者如果有不明白的地方可以参考一些Window的编程书籍。

5.1.2 OpenGL应用程序框架

OpenGL在windows平台下主要有两个动态库一个是Opengl32.dll一个是Glu32.dll。编译程序要用到对应的两个.lib文件（当然亦可以动态链接，直接从动态库里面获取这些函数的指针）。OpenGL的初始化非常简单，而且它的函数调用也非常简单。所以我们采用它来实现我们的程序框架而不是DirectX 。

OpenGL的初始化有三部分。第一，初始化OpenGL。第二，OpenGL的渲染循环。第三，OpenGL的Window事件响应。初始化OpenGL我们在InitializeOpenGL实现，这个函数调用两个函数，一个设置OpenGL的像素格式，另一个初始化OpenGL的各种渲染状态。

```
//这个函数设置OpenGL的像素格式
```

```
int CEGUISample::SetupPixelFormat(void)
{
    hDC = GetDC(g_mainWnd);
    int nPixelFormat;
    PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),    //pfd结构的大小
        1,                                //版本号
        PFD_DRAW_TO_WINDOW |          //支持在窗口中绘图
        PFD_SUPPORT_OPENGL |          //支持 OpenGL
        PFD_DOUBLEBUFFER,// |          // 双缓存模式
        PFD_TYPE_RGBA,                // RGBA 颜色模式
        32,                           // 32 位颜色深度
        0, 0, 0, 0, 0, 0,                 //忽略颜色位
        0,                            //没有非透明度缓存
        0,                            // 忽略移位位
        0,                            // 忽略累加缓存
        0, 0, 0, 0,                       // 忽略累加位
        32,                           // 32 位深度缓存
        8,                            // 8位模板缓存
        0,                            // 无辅助缓存
        PFD_MAIN_PLANE,               // 主层
        0,                          // 保留
        0, 0, 0                       // 忽略层,可见性和损毁掩模
    };
    //选择合适的格式，如果选择失败则返回
    if (!(nPixelFormat = ChoosePixelFormat(hDC, &pfd)))
    {
        return FALSE;
    }
    //设置当前设备的像素格式
    SetPixelFormat(hDC,nPixelFormat,&pfd);  //设置当前设备的像素点格式
    hRC = wglCreateContext(hDC);        //获取渲染描述句柄
    wglMakeCurrent(hDC, hRC);           //激活渲染描述句柄
    return TRUE;
}
```

另一个初始化OpenGL的函数InitGL这里就不做介绍了，读者可以自己阅读。主渲染函数为Render。当窗口没有消息的时候调用主渲染循环。这个函数非常简单，它调用OpenGL渲染。

```
void CEGUISample::Render()
{
    //设置背景清除颜色为黑色
    glClearColor(0.0f, 0.0f, 0.3f, 1.0f);
    //设置当前颜色为白色
    glColor3i(255,255,255);
    //清除背景，使用背景颜色
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //重置当前矩阵为单位阵
    glLoadIdentity();
    //调用CEGUI的渲染入口函数
    CEGUI::System::getSingleton().renderGUI();
    //交换缓冲区内容，将后背缓冲区内容显示在屏幕上
```

```
    SwapBuffers(hDC);
}
```

这个函数调用了CEGUI的renderGUI函数执行CEGUI的渲染。其实读者可能还需要调用System类的injectTimePulse函数，更新所有的窗口。但作者认为这是不必要的，如果使用了CEGUI提供的Tooltip类则必须调用这个函数，才能显示提示信息。

第三部分只有一个函数就是当窗口大小改变的时候通知OpenGL。这个函数在窗口过程中被调用。

```
void CEGUISample::ReSize(int Width, int Height)
{
    glViewport(0,0,Width,Height);                   // 设置OpenGL视口大小。

    glMatrixMode(GL_PROJECTION);                    // 设置当前矩阵为投影矩阵。
    glLoadIdentity();                               // 重置当前指定的矩阵为单位矩阵
    gluPerspective                                  // 设置透视图
        ( 45.0f,                                    // 透视角设置为 54 度
        (GLfloat)Width/(GLfloat)Height,             // 窗口的宽与高比
        0.1f,                                       // 视野透视深度:近点0.1f
        2500.0f                                     // 视野透视深度:始点0.1f远点2500.0f
        );
    glMatrixMode(GL_MODELVIEW);                     // 设置当前矩阵为模型视图矩阵
    glLoadIdentity();                               // 重置当前指定的矩阵为单位矩阵
}
```

这个函数重新设置了OpenGL的投影矩阵。

通过以上三个部分OpenGL的基本框架就搭建起来了。

5.1.3 CEGUI应用程序框架

CEGUI的输入接口和渲染接口与游戏程序的交互在5.1.1和5.1.2中已经出现，想必读者已经理解。这里主要介绍CEGUI的初始化和窗口的创建等。

CEGUI的初始化是从System类的创建开始的。我们知道System类可以说是CEGUI的流程和控制流的核心类。首先创建System，然后设置资源组，最后创建窗口。创建窗口将在第5.2节介绍。这一节主要介绍System的创建和资源组的设置。

创建System类非常简单，先创建OpenGL的渲染接口，然后创建System类。

```
new CEGUI::System(new CEGUI::OpenGLRenderer(1024));
```

设置资源组，如果资源组不设置则CEGUI无法正确的加载到各种资源，笔者在编写这节例子的时候，就因为没有正确的设置资源组导致无法加载各种资源。

```
//设置资源组信息
void CEGUISample::SetResourceGroup()
{
    char resourcePath[MAX_PATH];
    //设置资源文件夹的前缀
    const char* dataPathPrefix = "..\\datafiles";
    //获取CEGUI的默认资源提供器
    CEGUI::DefaultResourceProvider* rp = static_cast<CEGUI::DefaultResourceProvider*>
        (CEGUI::System::getSingleton().getResourceProvider());
    //设置schemes的资源组
    sprintf(resourcePath, "%s\\%s","..\\datafiles", "schemes\\");
    rp->setResourceGroupDirectory("schemes", resourcePath);
    //设置imagesets的资源组
    sprintf(resourcePath, "%s\\%s", dataPathPrefix, "imagesets\\");
    rp->setResourceGroupDirectory("imagesets", (resourcePath));
    //设置fonts的资源组
```

```
    sprintf(resourcePath, "%s\\%s", dataPathPrefix, "fonts\\");
    rp->setResourceGroupDirectory("fonts", (resourcePath));
    //设置layouts的资源组
    sprintf(resourcePath, "%s\\%s", dataPathPrefix, "layouts\\");
    rp->setResourceGroupDirectory("layouts", (resourcePath));
    //设置looknfeels的资源组
    sprintf(resourcePath, "%s\\%s", dataPathPrefix, "looknfeel\\");
    rp->setResourceGroupDirectory("looknfeels", (resourcePath));
    //设置lua_scripts的资源组
    sprintf(resourcePath, "%s\\%s", dataPathPrefix, "lua_scripts\\");
    rp->setResourceGroupDirectory("lua_scripts", (resourcePath));
    // 下面这段非常重要，设置各个主要资源的默认资源组
    CEGUI::Imageset::setDefaultResourceGroup("imagesets");
    CEGUI::Font::setDefaultResourceGroup("fonts");
    CEGUI::Scheme::setDefaultResourceGroup("schemes");
    CEGUI::WidgetLookManager::setDefaultResourceGroup("looknfeels");
    CEGUI::WindowManager::setDefaultResourceGroup("layouts");
    CEGUI::ScriptModule::setDefaultResourceGroup("lua_scripts");
}
```

有了上面两步的设置，下一步就可以加载CEGUI的资源，创建窗口了。

5.2 CEGUI例子程序

这一节，使用第5.1节创建的框架和CEGUI提供的例子来介绍。

在5.1.3中介绍了CEGUI框架以System的创建开始，然后需要设置默认资源组。如果不设置资源组则需要在各种资源文件中指定，否则很可能CEGUI无法找到资源。CEGUI提供的默认资源管理器，一个资源组只能对应一个路径，不像有些资源管理器（比如OGRE提供的）可以一个资源组下有多个路径，当从这个资源组加载资源的时候，按照顺序匹配。

我们的框架提供了两个类一个是CEGUISample，这个类负责搭建Win32框架和OpenGL框架，以及一部分的CEGUI的框架。它的内容在第5.1节做了详细介绍。另一个类是UILogic，这个类主要负责资源加载和窗口创建以及窗口事件处理函数。本节的内容主要介绍UILogic类的内容。以后章节中的代码都使用这个框架，一般来说CEGUISample不需要改变，只改变UILogic就可以实现。

5.2.1 加载资源和创建窗口

第4章介绍过，CEGUI的资源是由Scheme管理的。所以资源管理非常简单，只要加载Scheme文件就可以完成资源的加载。至于Scheme文件中各个元素的含义，在第4章已经详细介绍，不清楚的读者可以参考第4章。CEGUISample会调用UILogic两个成员函数，来初始化和创建窗口。他们是createDemoWindows和initGUI。这里主要介绍这两个函数的实现。initGUI函数，在第5章的例子中没有用到，就不做介绍了。createDemoWindows函数的主要内容也是从CEGUI官方提供的例子Sample_FalagardDemo1中选取的。

```
void UILogic::createDemoWindows()
{
    // 获取窗口管理器的单件引用
    WindowManager& winMgr = WindowManager::getSingleton();
    // 加载模式，各种资源在这里加载
    SchemeManager::getSingleton().loadScheme("VanillaSkin.scheme");
    // 设置默认鼠标的图像，CEGUI中鼠标也是一个图像，这时必须隐藏Windows鼠标
    System::getSingleton().setDefaultMouseCursor("Vanilla-Images", "MouseArrow");
    // 直接从一个文件加载背景图像集，这一步会定义一个full_image的图像
    ImagesetManager::getSingleton().createImagesetFromImageFile("BackgroundImage",
    "GPN-2000-001437.tga");
    // 创建一个静态图像窗口作为窗口的根目录，当然也可以是其他窗口，根据需要
```

```
    Window* background = winMgr.createWindow("Vanilla/StaticImage");
    // 设置窗口的区域大小，使用相对模式，可以在不知道窗口的高宽的时候设置占用整个
//渲染窗口区域
    background->setArea(URect(cegui_reldim(0), cegui_reldim(0), cegui_reldim(1),
    cegui_reldim(1)));
    // 设置两个属性
    background->setProperty("FrameEnabled", "false");
    background->setProperty("BackgroundEnabled", "false");
    // 设置背景图片信息，这个图像是我们刚才从文件直接创建的
    background->setProperty("Image", "set:BackgroundImage image:full_image");
    //设置background为窗口底板，所有窗口的根目录
    System::getSingleton().setGUISheet(background);
    //加载字体
    if(!FontManager::getSingleton().isFontPresent("Iconified-12"))
        FontManager::getSingleton().createFont("Iconified-12.font");
    // 加载布局文件，并且作为根窗口的子窗口
    background->addChildWindow(winMgr.loadWindowLayout("VanillaWindows.layout"));
    // 创建控制台，这个类下文介绍
    d_console = new DemoConsole("Demo");
    // 注册根窗口的键盘按下处理事件
    background->subscribeEvent(Window::EventKeyDown,
                        Event::Subscriber(&UILogic::handleRootKeyDown, this));
    // 激活根窗口，也可以不激活它，这一步可选
    background->activate();
}
```

特别说明，笔者在使用CEGUI库编译Debug版的应用程序的时候，可以编译成功，但如果执行则在注册窗口事件处理函数的时候，程序总会崩溃。而且CEGUI提供的官方例子也是如此。可见CEGUI库在Debug配置下编译的程序有Bug。

CEGUI加载资源可以同时加载多个Scheme文件，但这些文件中必须没有重复的资源定义。加载资源的唯一入口方法就是SchemeManager::getSingleton().loadScheme。创建窗口的唯一入口是WindowManager::getSingleton.createWindow，这个函数有两个参数一个是Scheme文件中定义的外部窗口类型一个是窗口的名称。FalagardMapping元素将外部窗口类型和内部窗口类型，渲染窗口以及窗口外观联系起来。注册窗口的事件处理函数的唯一入口就是事件集的subscribeEvent函数。加载窗口布局的唯一入口是WindowManager::getSingleton.loadWindowLayout，这个函数需要窗口布局的文件名称。CEGUI所有的操作都是通过各个管理器来实现的，读者可以根据这个原理自己尝试加载各种资源，创建窗口。窗口创建以后就可以设置它的属性和注册事件处理函数。这些功能和游戏UI界面的逻辑息息相关，不同的逻辑需要设置的值不同。布局文件提供了一种通过XML文件设置属性，注册事件处理函数的方法。这种方法的优势非常明显，一般建议采用这种方法来创建窗口。

DemoConsole这个类完全从CEGUI的Sample_FalagardDemo1例子中拷贝过来，它负责创建一个可以输入的控制台。它在构造函数里加载了控制台的布局文件。并且注册了几个事件处理函数。这里就不详细介绍了。

5.2.2 窗口的逻辑处理

窗口的逻辑主要就是一些事件的响应和窗口属性的调整。事件的响应通过注册事件响应函数到CEGUI窗口中，窗口属性设置和获取通过属性集提供的获取和设置函数就可以完成。这是在C++ 程序中的实现，在脚本和布局文件也可以修改。而且后者的修改更加灵活，方便。窗口的逻辑是非常不同的，因为它和功能相关。窗口的功能往往是经常变化的，因此建议采用布局文件加上脚本处理来创建窗口和处理事件而不是使用C++代码。本例中用到了布局文件来创建窗口，但逻辑事件的处理还是在C++中处理的。本书第15章将介绍如何在CEGUI中使用脚本。

这节主要介绍以下几个逻辑事件的处理函数。

控制台的文本提交逻辑。

```
bool DemoConsole::handleSubmit(const CEGUI::EventArgs& args)
```

```
{
    //使用CEGUI名字空间，免得每次都多写CEGUI
    using namespace CEGUI;
    // 获取CEGUI中的窗口，这个窗口定义在布局文件中，布局文件设置了它的ID属性
    Editbox* editbox = static_cast<Editbox*>(d_root->getChild(EntryBoxID));
    // 获取编辑器中的文字
    String edit_text(editbox->getText());
    // 如果文字内容不为空，则添加到多行文本框中
    if (!edit_text.empty())
    {
        // 添加编辑框中的内容到历史基类的向量中（数组）
        d_history.push_back(edit_text);
        //重设历史位置
        d_historyPos = d_history.size();
        // 末尾加上\n表示换行
        edit_text += '\n';
        // 获取历史窗口，这个窗口也是以ID来获取的
        MultiLineEditbox* history = static_cast<MultiLineEditbox*>(d_root->
                getChild(HistoryID));
        // 添加内容到控件
        history->setText(history->getText() + edit_text);
        // 设置光标到最后
        history->setCaratIndex(static_cast<size_t>(-1));
        // 清除输入框的内容
        editbox->setText("");
    }
    //输入框激活，接受输入
    editbox->activate();
    return true;
}
```

这些逻辑和CEGUI的具体控件的实现非常相关，读者可能不明白为什么这么做。读者可以参考CEGUI的函数和类参考手册，或者直接看相应的控件的实现。本书在第7章将会介绍一部分窗口控件的实现，但显然不可能介绍所有的控件。读者可以自己阅读这些控件的实现，其实只要知道了基本的Window的各个函数以及CEGUI整体的架构，就可以随心所欲做自己希望做的修改了。如果不懂得CEGUI的架构原理，以及各个模块之间的关系，各种配置文件之间的关系等等。哪么希望灵活的设计自己的控件，基本上也是很难得。这就是本书为什么花哪么多章节介绍CEGUI的各种原理，各种文件和架构信息等。

第二个事件处理函数handleKeyDown，处理四个方向键，上下翻历史，左右移动光标。

```
bool DemoConsole::handleKeyDown(const CEGUI::EventArgs& args)
{
    using namespace CEGUI;
    // 获取编辑框的文字内容
    Editbox* editbox = static_cast<Editbox*>(d_root->getChild(EntryBoxID));

    switch (static_cast<const KeyEventArgs&>(args).scancode)
    {
    //鼠标上键，则弹出历史作为当前的输入，d_history记录了所有的输入提交历史
    case Key::ArrowUp:
        d_historyPos = ceguimax(d_historyPos - 1, -1);
        if (d_historyPos >= 0)
        {
            editbox->setText(d_history[d_historyPos]);
```

```
                editbox->setCaratIndex(static_cast<size_t>(-1));
            }
            else
            {
                editbox->setText("");
            }
            //激活输入框，继续接受输入
            editbox->activate();
            break;
        //向下弹出历史
        case Key::ArrowDown:
            d_historyPos = ceguimin(d_historyPos + 1, static_cast<int>(d_history.size()));
            if (d_historyPos < static_cast<int>(d_history.size()))
            {
                editbox->setText(d_history[d_historyPos]);
                editbox->setCaratIndex(static_cast<size_t>(-1));
            }
            else
            {
                editbox->setText("");
            }
            editbox->activate();
            break;
        default:
            return false;
        }
        return true;
}
```

以前一直没有介绍，CEGUI处理函数返回true表示这个事件被处理和，返回false表示这个事件没有被处理。但是需要注意CEGUI会调用所有的注册到一个事件的处理函数，即使已经有函数返回true。这个返回值只影响这个事件是否被处理过，如果所有的处理函数中有一个处理了这个事件则事件将会表标记为已经处理。具体实现在Event::operator()(EventArgs& args)函数中。

本文的例子有点像一个简单的聊天栏，但是它并不支持中文的输入和显示。本书第13和第14章将会详细介绍CEGUI的中文显示和输入，而且会制作一个简单例子程序。

窗口的逻辑处理一般来说比较繁琐而且也没有什么规律，所以是比较容易出错的地方。这里举这个例子所示的逻辑还比较简单，但是它很有代表性。读者只要熟悉了这种方法，以后制作复杂的程序也不是难事。

5.3 本章小结

本章主要介绍了如何编写CEGUI的客户程序。介绍了一个简单的CEGUI游戏程序的框架，以后本书的各章例子都使用这个框架来实现。第5.2节介绍了CEGUI的窗口逻辑和加载资源创建窗口等内容。这些内容都有它的方法和步骤，读者熟悉后就会灵活的使用了。

本章习题：

1.熟悉本章的例子，自己编译和运行一遍，读懂例子代码。

2.自己仿照例子，改写一个CEGUI提供的官方的例子程序。

第6章 外观系统

CEGUI最早期的版本是没有外观系统的，后来在一个论坛上有个网友建议在CEGUI中添加这么一个系统。外观系统可以提供XML文件指定窗口的图像布局，子窗口的布局，默认属性，属性定义等。此前只能通过C++代码或者脚本语言来实现。Falagard这个名称就是当初在论坛上建议新增这个系统的网友使用的网名。外观系统的XML定义文件在CEGUI的例子数据目录下的looknfeel子目录下，文件以.looknfeel为扩展名。处理这类文件的C++元素的代码在CEGUIBase工程目录下的falagard过滤器下。由于外观的定义和渲染窗口密切相关，所以渲染窗口中定义的各种命名区域，子窗口，需要的图像等在外观定义文件中必须要存在。CEGUI实现的各种官方控件（在CEGUIBase工程目录下的elements过滤器下）以及各种控件的官方渲染窗口（在CEGUIFalagardWRBase工程里定义）都定义了一些特定的必须在looknfeel文件中指出具体值的元素。CEGUI团队为了使使用者不必阅读代码就可以方便的知道在各个控件必须要指定的元素，所以提供了一个文档叫做FalagardSkinning.pdf它可以在光盘"CEGUI文档"目录下找到。本章结合这个文档，介绍外观系统。本章分为两部分，一部分介绍looknfeel文件的各个元素的含义以及如何使用他们。另一部分介绍对应元素在CEGUI中的具体实现。前者参考CEGUI的文档，但毕竟篇幅有限不能完全介绍，读者可以自己阅读这个文档。

6.1 LookNFeel文件

looknfeel文件可以说是CEGUI最复杂和ＣＥＧＵＩ具体实现最相关的配置文件了。控件和渲染窗口中定义了那些命名的区域，子窗口等等都需要在looknfeel文件中有相应的定义。本节以CEGUI提供的FalagardSkinning.pdf说明文件为基础，介绍CEGUI的外观定义。

6.1.1 统一坐标系统

何为统一坐标系统？窗口的坐标一般有两种，一种是绝对坐标，另一种是相对坐标。绝对坐标需要知道窗口的位置的绝对像素值，大小的绝对像素值。而且渲染系统只认识绝对坐标，在最终渲染的时候相对坐标也必须转化为绝对坐标。相对坐标记录的其实是实际窗口的百分比。比如窗口的X坐标位置是屏幕宽度的0.25，窗口的宽度是屏幕宽度的1/2等。相对坐标不需要知道具体的屏幕位置和具体的大小，在需要渲染的时候根据传递给它的参数（高度或者宽度）乘以自己保存的百分比值，就可以计算出屏幕的绝对坐标值了。所以相对坐标在游戏窗口大小变化的时候能很好的跟随它变化。统一坐标是这两种坐标的合并，它既包含一个相对坐标量也包含一个绝对坐标量。相对坐标量叫做scale（缩放因子），绝对坐标量叫做offset（绝对偏移）。CEGUI中使用UDim表示一个绝对坐标系的坐标值（就如X轴上的x值一样）。

UDim(scale, offset)，其中scale值范围在[0.0f, 1.0f]之间，百分比值的范围，offset值是在绝对坐标系统下的偏移量，它的取值范围取决于窗口的大小。当scale等于0的时候，UDim表示绝对坐标，当offset等于0的时候UDim表示相对坐标。比如UDim（0.5f, -30)值，如果表示窗口的X位置哪么它表示的就是父窗口宽度的一半减去30像素。计算公式0.5*parentWidth + (-30)。这个公式是UDim转化为绝对坐标的公式，转化为相对坐标的公式为:
0.5+(-30)/parentWdith。这里以具体例子举例，将具体数值变成scale和offset就可以得到普适的公式了。UDim需要在XML文件中配置，它的属性格式是{scale, offset}。

UDim只能表示一个坐标上的值，如果要表示一个点必须有两个坐标（在二维平面上）。CEGUI中定义了UVector2来表示点，它是由两个UDim构成，分别表示两个坐标轴上对应的值。举个简单的例子，如下所示：

```
UVector2( UDim(0, 25), UDim(0.2f, 12) )
```

这个例子中指定了一个在X轴为25，Y轴为1/5的父窗口高度加上12像素。

UVector2的属性格式为{{scale1，offset1},{scale2，offset2}}。

有了点的定义就可以定义一个矩形了，窗口显然都是矩形的，定义矩形就显得很重要。矩形的定义可以使用四条交点定义法，也可以使用位置加高宽定义法。其实都一样使用两个点来定义。CEGUI中使用URect定义一个矩形。URect有两个构造方法分别对应矩形的两种定义方法。URect（left udim, top udim, right udim, bottom udim）四条线包裹起来的矩形或者URect（tl uvec2, br uvec2）两个点构成，如果br_uvec2保存的是窗口的高宽则就是第二种矩形定义方法，如果保存的是右和下的值则定义的是第一种矩形。

属性格式：{{sl,ol},{st,ot},{sr,or},{sb,ob}}。

6.1.2 模式中的外观定义

模式（Scheme）是CEGUI中资源的集中管理器。模式的外观自然要在模式中有定义。模式中首先必须定义一个（可以有多个）渲染窗口的模块CEGUI官方的模块是CEGUIFalagardWRBase。通过<WindowRendererSet Filename="CEGUIFalagardWRBase" />来指定。渲染窗口中定义了许多状态和皮肤图像，命名区域，子窗口等等需要在looknfeel文件中明确定义的元素。所以这个模块的定义被放在这一章介绍。

LookNFeel元素描述一个外观文件。

```
<LookNFeel Filename="FunkyWidgets.looknfeel" />
```

模式中可以包含多个这样的定义，不同窗口的外观定义可以在不同的文件中。

外观映射FalagardMapping，相信读者对这个元素已经很熟悉了。在第4章曾经介绍过它定义一个窗口的外部类型，映射为内部类型，内部渲染窗口，以及一个外观定义。

一个外观定义的例子。

```
<FalagardMapping
WindowType="FunkyLook/Button"
TargetType="CEGUI/PushButton"
Renderer="Falagard/Button"
LookNFeel="MyButtonSkin"
/>
```

外部类型名称是FunkyLook/Button，它可以用来创建窗口。内部类型名称是CEGUI/PushButton，它对应一个内部的控件，内部渲染窗口名称是Falagard/Button，它对应模块CEGUIFalagardWRBase中的一个渲染窗口，外观定义名称是MyButtonSkin，它是某个外观文件中定义的外观对象。

一个创建窗口的例子。

```
// 获取窗口管理器
CEGUI::WindowManager wMgr& = CEGUI::WindowManager::getSingleton();
// 创建窗口，第一个参数是外部窗口类型，第二个参数是窗口名称（可以省略）
Window* wnd = wMgr.createWindow("FunkyLook/Button","myFunkyButton");
```

模式中和外观相关的内容就这么多，下一小节介绍各种外观元素。

6.2 外观元素介绍

外观元素一共有37个，它是树形结构，最顶层的根元素是WidgetLook。所有的其他元素都必须是它的间接或直接子元素。

6.2.1 WidgetLook元素

外观定义的元素，最顶层的是WidgetLook，这个元素代表一个外观定义，FalagardMapping中的LookNFeel的值就是它的名字。它的下一层的元素可以是ImagerySection，Section，PropertyDefinition，Property，PropertyLinkDefinition，NamedArea，Child，StateImagery这个几个元素。ImagerySection用来指定一个一种图像的布局或者一种文字的布局。Section是为了重复使用ImagerySection而定义的元素。就是说这个元素可以指定在其他的WidgetLook中找到ImagerySection而不需要多次的定义同样的ImagerySection。PropertyDefinition可以定义一个属性，这个属性和其他属性一样，可以使用属性的通用接口来操作。不同之处是它是在外观文件中定义的。Property则是初始化使用这个外观的窗口的一些属性。PropertyLinkDefinition代表定义在子窗口的属性在使用这个外观的窗口中的链接。NamedArea代表一个命名区域，在渲染窗口中可以通过名称找到这个区域。Child外观定义一个子窗口，最好的例子便是滚动条。注意PropertyLinkDefinition链接到这些子窗口中的属性。StateImagery定义一种状态的显示状态。最好的例子便是按钮的四种状态，正常（Normal），高亮（Hover），受击（Clicked），禁用（Disabled）。渲染窗口可以通过程序获得这些定义的状态，然后根据窗口的当前状态，决定使用那个来渲染。下面是一个简单的例子，为了说明结构这里省略了具体实现，而且也不是所有的元素都在里面。

```
<WidgetLook name="TaharezLook/FrameWindow">
  <PropertyDefinition name="NormalTextColour" initialValue="FFFFFFFF" />
    <NamedArea name="TextArea">
    </NamedArea>
```

```
    <ImagerySection name="container_normal">
    </ImagerySection>
  <StateImagery name="Enabled">
    <Layer>
      <Section section="container_normal" />
    </Layer>
  </StateImagery>
    <Child type="TaharezLook/Titlebar" nameSuffix="__auto_titlebar__">
    <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
     <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
     <Dim type="Height" ><FontDim type="LineSpacing" padding="8" /></Dim>
    </Area>
   <Property name="AlwaysOnTop" value="False" />
  </Child>
</WidgetLook>
```

上面的8种元素而且只有他们才能放在WidgetLook元素的内部，用来描述WidgetLook的详细信息，具体这8个元素的含义，后文将有详述。一些元素读者可能还不太清楚，比如Dim，Area等。这些虽然不能放在WidgetLook的第一层（直接元素），但他们可以放在WidgetLook的子元素中。外观的各种元素可以组成一个元素树。

6.2.2 ImagerySection元素

ImagerySection元素，描述一种可以被渲染的基本结构。它可以描述一个各向可以拉伸的窗口结构，也可以描述只有竖直或水平可拉伸的窗口结构，也可以描述一张背景图片或者一个文字显示。在第3.1章，图3.1中我们介绍了各向可拉伸窗口。这个元素可以组合多个
ImageryComponent或者TextComponent来描述一个渲染结构。ImageryComponent元素描述各种窗口渲染结构，TextComponent元素描述文本显示。ImagerySection包含且只能包含这两个元素。比如下面的ImagerySection 名称是withtitle_frame。外观文件中的定义各种名称都是为了找到对应的元素。

```
<ImagerySection name="withtitle_frame">
  <FrameComponent>
    <Area>
    </Area>
    <Image type="BottomLeftCorner" imageset="TaharezLook"
                                image="WindowBottomLeft" />
    <Image type="BottomRightCorner" imageset="TaharezLook"
                           image="WindowBottomRight" />
    <Image type="LeftEdge" imageset="TaharezLook" image="WindowLeftEdge" />
    <Image type="RightEdge" imageset="TaharezLook" image="WindowRightEdge" />
    <Image type="BottomEdge" imageset="TaharezLook" image="WindowBottomEdge" />
  </FrameComponent>
    <ImageryComponent>
    <Area>
    </Area>
    <Image imageset="TaharezLook" image="GenericBrush" />
    <ColourProperty name="ClientAreaColour" />
    < type="Stretched" />
    <HorzFVertFormatormat type="Stretched" />
  </ImageryComponent>
</ImagerySection>
```

由于ImagerySection 只能包含三个元素，这里就不做过的介绍。而是详细介绍FrameComponent和ImageryComponent，以及TextComponent。FrameComponent可以描述我们前
面说过的三种窗口的布局类型。Area对象描述一个区域。Image 对象描述一个图像，它的属性type描述这个图像是窗口分成9分的时候，这个Image对象描述的是那个部分。比如
BottomLeftCorner描述图3-1中的左下（g），也可以根据英文名称来想象。imageset和image两个属性配合找到一个图像。图像是通过图像集名称和图像名称唯一确定的。FrameComponent元素下最多可以有9个窗口图像，对应图3-1中的9个位置。如果指定3个水平方向的就是水平可以拉伸的，同理如果3个竖直方向的就是竖直可拉伸的。下面是一个FrameComponent的定义，可以看到它定义了一个框架的渲染区域，然后定义了8个图像，中间的背景图像没有定义，哪么中间的背景区域就是透明的了。
VertFormat 和HorzFormat 表示当图像高度或者宽度小于它要覆盖的窗口的区域的时候采用什么方法来处理，Stretched表示拉伸，还有CentreAligned表示居中显示，BottomAligned表示底部对齐显示（主要对应VertFormat ），Tiled表示重复图像，把图像看做瓷砖一样平铺。Area元素描述一个非命名区域，它和NameArea最大的区别就是后者可以通过名称找到。区域的定义比较复杂，后文有详述。

```
<FrameComponent>
 <Area>
    <Dim type="LeftEdge">
       <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge">
       <AbsoluteDim value="11" />
    </Dim>
    <Dim type="RightEdge">
       <UnifiedDim scale="1" offset="0" type="RightEdge" />
    </Dim>
    <Dim type="BottomEdge">
       <UnifiedDim scale="1" offset="0" type="BottomEdge" />
    </Dim>
 </Area>
 <Image type="TopLeftCorner" imageset="TaharezLook" image="StaticTopLeft" />
 <Image type="TopRightCorner" imageset="TaharezLook" image="StaticTopRight" />
 <Image type="BottomLeftCorner" imageset="TaharezLook" image="StaticBottomLeft" />
 <Image type="BottomRightCorner" imageset="TaharezLook" image="StaticBottomRight" />
 <Image type="LeftEdge" imageset="TaharezLook" image="StaticLeft" />
 <Image type="TopEdge" imageset="TaharezLook" image="StaticTop" />
 <Image type="RightEdge" imageset="TaharezLook" image="StaticRight" />
 <Image type="BottomEdge" imageset="TaharezLook" image="StaticBottom" />
 <VertFormat type="Stretched" />
 <HorzFormat type="Stretched" />
</FrameComponent>
```

ImageryComponent描述一个单个的可渲染的结构，它就类似一个图像。而FrameComponent则描述的是最多9个可以渲染的结构或者叫区域。

```
<ImageryComponent>
 <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
 </Area>
 <Image imageset="TaharezLook" image="ProgressBarLitSegment" />
```

```
 <VertFormat type="Stretched" />
 <HorzFormat type="Tiled" />
</ImageryComponent>
```

各个元素的含义和FrameComponent里的一样。它只有一个图像，而且图像是没有类型的。

TextComponent描述文本元素，它描述了一个文字渲染的区。在渲染的时候它获取窗口的文本，然后缓存到窗口的渲染缓冲中。

```
<ImagerySection name="label">
 <TextComponent>
    <Area>
       <Dim type="LeftEdge">
          <AbsoluteDim value="15" />
       </Dim>
       <Dim type="TopEdge">
          <AbsoluteDim value="0" />
       </Dim>
       <Dim type="RightEdge">
          <UnifiedDim scale="1" offset="0" type="RightEdge" />
       </Dim>
       <Dim type="BottomEdge">
          <UnifiedDim scale="1" offset="0" type="BottomEdge" />
       </Dim>
    </Area>
    <VertFormat type="TopAligned" />
    <HorzFormat type="LeftAligned" />
 </TextComponent>
</ImagerySection>
```

元素名称中包含Component的都是描述具体的绘制对象的比如窗口框架，文本和简单图像。

ImagerySection元素可以包含一个或者多个Component。它是Component的容器，也是可以渲染的，渲染时它调用所有容器中的Component元素来渲染。

6.2.3 StateImagery元素

StateImagery元素可以看做是ImagerySection的各种状态的集合。它描述一些状态的ImagerySection。比如按钮有正常，高亮，点击和禁用四种状态（当然可以有其他的定义的状态，但一般来说就是这四种）就可以使用StateImagery来描述，如下所示。

```
<StateImagery name="Normal">
 <Layer>
    <Section section="normal" />
    <Section section="label">
       <ColourProperty name="NormalTextColour" />
    </Section>
 </Layer>
</StateImagery>
<StateImagery name="Hover">
 </StateImagery>
<StateImagery name="Pushed">
</StateImagery>
<StateImagery name="Disabled">
</StateImagery>
```

StateImagery只有一个直接子元素就是Layer，它只能作为WidgetLook的直接子元素。StateImagery元素可以包含任意多个Layer元素，它是Layer元素的集合。Layer元素，从它的名字就可以理解，它是表示渲染的层，多层的时候是按顺序层叠的，出现在最前面的最后被渲染。Layer元素只能包含Section 元素，Section 元素我们已经介绍过，它是链接到一个ImagerySection的，目的是为了ImagerySection的重用，它可以重用其他WidgetLook内部的ImagerySection，不过一般也不会使用其他外观定义里的ImagerySection。Layer元素可以包含多个Section元素，渲染顺序也是先出现的后渲染。上面的例子里面normal显然是一个图像或者框架的ImagerySection的名字而label则是一个文字的ImagerySection的名字。

在一个外观定义中可以出现任意多个StateImagery 的定义，这个定义在渲染窗口渲染时根据窗口的当前状态决定使用那个StateImagery来渲染。

6.2.4 属性相关的三个元素

与属性相关的有三个元素分别是Property，PropertyLinkDefinition和PropertyDefinition。Property是用来初始化使用这个外观的窗口的属性，没有子元素，可以出现在WidgetLook中为应用这个外观的窗口指定属性初始值，也可以出现在Child元素内为子窗口初始化属性值。PropertyLinkDefinition链接外观中定义的子窗口的属性到应用这个外观的窗口中，在这个窗口中可以使用这个属性。这个元素没有子元素，而且必须出现在WidgetLook元素中。PropertyDefinition为应用这个外观的窗口定义一个属性，它只能出现在WidgetLook元素也没有子元素。

下面是Property元素使用的一个例子，从CEGUI资源文件中截取的一段。

```
<WidgetLook name="TaharezLook/Titlebar">
<Property name="CaptionColour" value="FFFFFFFF" />
<ImagerySection name="main">
<ImageryComponent>
<Area>
<Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
...
</Area>
...
</ImageryComponent>
</ImagerySection>
...
</WidgetLook>
```

这里定义了名为CaptionColour的属性，并且初始的值为不透明的白色。

PropertyDefinition和PropertyLinkDefinition有两个个共同的属性redrawOnWrite和layoutOnWrite。前者表示当这个属性被写入新值的时候是不是引发窗口的重绘，后者表示当新值写入的时候会不会引发重新布局。比如说定义了一个或者链接一个文字颜色的属性，哪么应该指定当这个值发生变化的时候会引发重绘。

下面是这段是PropertyDefinition的例子，它定义了滚动条宽度的属性，并指定当属性值被修改的时候重新布局。什么是布局呢，就是当一个窗口的子窗口的位置或者大小改变的时候需要重新调整这个窗口的所有子窗口的位置和属性，这个调整的过程就是布局。滚动条的宽度改变了，哪么它就需要重新布局了。

```
<WidgetLook name="PropertyDefExample">
<PropertyDefinition
name="ScrollbarWidth"
initialValue="12"
layoutOnWrite="true"
/>
...
</WidgetLook>
```

PropertyLinkDefinition链接一个属性到子窗口，目的是在应用外观的窗口中可以通过一个名字访问这个窗口的自动子窗口（也就是在外观中定义的子窗口）的属性。

```
<WidgetLook name="PropertyLinkExample">
```

```
<PropertyLinkDefinition
name="CaptionTextColour"
widget="__auto_titlebar__"
targetProperty="CaptionColour"
initialValue="FFFF3333"
layoutOnWrite="true"
/>
...
<Child type="TaharezLook/Titlebar" nameSuffix="__auto_titlebar__">
  <Property name="AlwaysOnTop" value="False" />
</Child>
</WidgetLook>
```

这个例子中容易发现"__auto_titlebar__"这个自动子窗口必须出现，而且这个窗口中必须有名为CaptionColour的属性。定义后应用这个外观的窗口可以通过CaptionTextColour这个名称来访问子窗口的属性，就像访问自己的属性一样。

CEGUI中可以使用操作属性的各种方法直接操作PropertyLinkDefinition和PropertyDefinition定义或链接的属性。

6.2.5 自动子窗口

自动子窗口读者可能已经比较熟悉了，它是在外观中定义的窗口的子窗口，子窗口的名称是应用这个外观的窗口名称加上子窗口的后缀名做为窗口名。自动子窗口也是窗口可以通过窗口名称来查找，这样就可以找到这些子窗口了。外观文件中使用Child元素来定义一个子窗口。它有三个属性一个是type对应模式文件中定义的外部窗口类型名称。一个是nameSuffix定义子窗口的后缀名称。还有一个look用来指定子窗口应用的外观，这个属性可选而且不常用。比如下面的定义。

```
<WidgetLook name="TaharezLook/FrameWindow">
<Child type="TaharezLook/Titlebar" nameSuffix="__auto_titlebar__">
 <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><FontDim type="LineSpacing" padding="8" /></Dim>
 </Area>
 <Property name="AlwaysOnTop" value="False" />
</Child>
</WidgetLook >
```

指定了窗口的外部创建类型是TaharezLook/Titlebar，窗口的自动名称后缀是__auto_titlebar__。创建一个窗口需要的两个要是在这里都定义好了，所以肯定可以创建这个窗口了。

子窗口必须指定一个Area对象，表示这个窗口的位置。可以通过Property 元素指定许多的属性初始值。可以可选的指定窗口的竖直排列格式VertAlignment和水平排列格式HorzAlignment。

自动窗口的获取，假如有个叫Test_Frame的窗口使用了TaharezLook/FrameWindow这个窗口外观，哪么它可以通过以下代码获取子窗口__auto_titlebar__。

```
WindowManager::getSingleton().getWindow("Test_Frame__auto_titlebar__");
```

自动子窗口和布局文件中定义的窗口或者c++代码创建的窗口没有什么不同，它可以执行像他们一样操作。

6.2.6 区域的定义

区域可以简单的分为命名和非命名的区域两种。命名区域（NamedArea）有个名字属性，代表这个区域的名称，有个区域子元素代表具体的区域。这个元素只能出现在WidgetLook元素下，而且必须包含一个Area元素。如下所示。也就是说只有区域元素才真正的定义一个区域。

```
<NamedArea name="ClientWithTitleNoFrame">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
```

```
    <Dim type="TopEdge" ><WidgetDim widget="__auto_titlebar__"
                                dimension="BottomEdge" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="BottomEdge" ><WidgetDim dimension="BottomEdge" /></Dim>
  </Area>
</NamedArea>
```

区域也可以叫做矩形，前面我们讲过有两种类型，一种是四条线围成的区域，一种是位置和尺寸决定的区域。如果第一种定义的矩形通过left，right，top，bottom四条线定义。哪么它对应的第二种表示就是left，right-left（表示宽度），top，bottom-top（表示高度）。这两种矩形是可以转化的。

窗口都是矩形的哪么如何绘制出各种各样的样子呢，比如说一个圆形的按钮。答案是使用Alpha值来控制，不是圆形的部分Alpha值为0它透明。

区域使用Area元素来定义。它必须包含四个Dim子元素或者一个AreaProperty元素。前者指定矩形区域的四个数据，或者表示矩形区域的矩形数据在窗口的属性里面，这个属性通过AreaProperty来指定。Area只可以出现在Child，ImageryComponent，NamedArea，TextComponent，FrameComponent这五个元素中。

Dim元素表示一个坐标，比如X坐标或者Y坐标。属性type表示这个Dim是描述一条边还是高度宽度等。LeftEdge左边，TopEdge上边，当然还可以是RightEdge和BottomEdge。这四个可以指定第一种矩形。当后两个换成Width和Height时指定第二种矩形。当然也可以任意指定，只要知道这两种矩形的关系，可以算出矩形区域就可以。

Dim元素可以包含AbsoluteDim，FontDim，ImageDim，PropertyDim，UnifiedDim，WidgetDim这六个元素之一。这六个元素有共同的使用方法，他们都可以包含DimOperator元素，都可以被DimOperator包含，都可以被Dim元素包含。

DimOperator元素用来操作两个*Dim（可以是这六种元素之一）元素，它提供了加减乘除四种基本方法。有时候无法简单的使用这六种元素中的如何一种描述，只能通过某种计算才能的到矩形的某个数据。这时候就可以使用DimOperator元素来计算这个值。

AbsoluteDim描述一个绝对数值，对应像素为单位。下面例子使用AbsoluteDim定义了一个从父窗口的（10,50）开始，高250，宽290的一个矩形。如果使用另一种矩形表示就是（10，50，300，300）分别是左边，上边，右边和下边。

```
<Area>
<Dim type="LeftEdge" >
<AbsoluteDim value="10" />
</Dim>
<Dim type="TopEdge" >
<AbsoluteDim value="50" />
</Dim>
<Dim type="Width" >
<AbsoluteDim value="290" />
</Dim>
<Dim type="Height" >
<AbsoluteDim value="250" />
</Dim>
</Area>
```

FontDim元素，可以应用字体来计算或者获取一些数据来作为矩形的数据。这个元素使用的并不多，它可以用来获取字体的行距。

```
<Dim type="Height">
<FontDim type="LineSpacing" />
</Dim>
```

可以用它来计算一段字符串的描绘长度。使用目标窗口的字体和文字计算出水平宽度在加上10像素。

```
<Dim type="Width">
```

```
<FontDim type="HorzExtent" font="Roman-14" padding="10" />
</Dim>
```

ImageDim元素获取图像的数据（一般来说是高和宽）作为矩形的某个数据。

```
<Area>
<Dim type="LeftEdge">
<ImageDim imageset="myImages" image="leftImage" dimension="width" />
</Dim>
...
</Area>
```

这段代码中ImageDim返回图像"set:myImages image:leftImage"的宽度。

PropertyDim 元素，从指定的窗口中获取属性作为输出值。"__auto__button__"，是这个外观定义的子窗口，这个值也可以不指定，使用目标窗口。AbsoluteWidth是属性的名称，也就是窗口的绝对宽度的属性值。

```
<Area>
<Dim type="LeftEdge">
<PropertyDim widget="__auto__button__"  name="AbsoluteWidth" />
</Dim>
...
</Area>
```

这个例子是PropertyDim 的典型用途，获取子窗口的宽度，用于矩形的计算。

UnifiedDim元素，表示一个统一坐标值，以这个值作为输出。type属性系统用于确定到底使用高度还是宽度来参与计算。

```
<Area>
    <Dim type="LeftEdge">
        <UnfiedDim scale="0.5" offset="-8" type="LeftEdge" />
    </Dim>
...
</Area>
```

读者应该记得统一坐标系的计算公式吧，只要读者理解了统一坐标系的含义相信这些公式自动就会在你的脑海里出现的。

WidgetDim元素，获取子窗口或者自己的某个维度作为输出值。

```
<Area>
    <Dim type="LeftEdge" >
    <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge" >
        <WidgetDim widget=" auto titlebar " dimension="BottomEdge" />
    </Dim>
    <Dim type="Width" >
        <UnifiedDim scale="1" type="Width" />
    </Dim>
    <Dim type="BottomEdge" >
        <WidgetDim dimension="BottomEdge" />
    </Dim>
</Area>
```

当指定widget的时候使用子窗口的维度，当没有指定的时候使用自己的维度。

DimOperator元素，前文已经讲过有时候单一的六种Dim是无法计算出所需的数值的。所以要借助DimOperator元素执行一些四则运算。它只有一个属性op，表示操作的类型，比如加减乘除。下面一个DimOperator的例子。

```
<AbsoluteDim value="10">
    <DimOperator op="Multiply">
        <AbsoluteDim value="4" />
    </DimOperator>
```

```
</AbsoluteDim>
```

这个例子中具体操作是10*4等于40。

下一个例子，解开DimOperator优先级之谜。

```
<AbsoluteDim value="2">
    <DimOperator op="Multiply">
        <AbsoluteDim value="4">
            <DimOperator op="Add">
                <FontDim type="LineSpacing" />
            </DimOperator>
        </AbsoluteDim>
    </DimOperator>
</AbsoluteDim>
```

设字体的行距为L则上面这段操作是(2*4)+L还是2*(4+L)呢。答案是后者，DimOperator是内部先结合的。

区域(Area)的定义，一种是通过四个数定义一个矩形，另一种是定义一个区域属性，这个属性是URect类型的矩形数据。第一种定义需要上面介绍的六种Dim和DimOperator结合起来。算出一个值作为矩形的某个值。下面是一个比较综合的例子。

```
<Area>
  <Dim type="LeftEdge" >
        <ImageDim imageset="TaharezLook" image="WindowTopLeft" dimension="Width" />
    </Dim>
  <Dim type="TopEdge" >
      <ImageDim imageset="TaharezLook" image="WindowTopEdge" dimension="Width" />
    </Dim>
  <Dim type="RightEdge" >
    <UnifiedDim scale="1" type="Width">
      <DimOperator op="Subtract">
        <ImageDim imageset="TaharezLook" image="WindowTopRight"
                  dimension="Width" />
      </DimOperator>
    </UnifiedDim>
  </Dim>
  <Dim type="BottomEdge" >
    <UnifiedDim scale="1" type="Height">
      <DimOperator op="Subtract">
        <ImageDim imageset="TaharezLook" image="WindowBottomEdge"
                  dimension="Height" />
      </DimOperator>
    </UnifiedDim>
  </Dim>
</Area>
```

左边定义为一个图片的宽度，这个图片是框架的左边图片。上边定义为一个图片的宽度，这个图像是顶部的图像，这个不知是出于什么考虑（这段是从CEGUI例子的TaharezLook.looknfeel中拷贝的），我觉得取这个图片的高度似乎更加合适。右边使用父窗口的宽度减去右边边界图像的宽度。底边使用父窗口的高度减去底边窗口边界图像的高度。

介绍到这里大家应该对区域比较清楚了。本节并没有介绍所有的37种元素，但已经介绍了大部分常用而且非常重要的元素。读者如果有不清楚的地方可以查看CEGUI提供的官方介绍。

下一节将以一个例子详细介绍如何编写looknfeel文件。

提示： 由于篇幅所限这里不可能详细介绍每一个元素，如果读者在使用中有疑惑或者不知道这个元素都有那些属性可以参考CEGUI官方提供的手册（在光盘的CEGUI文档目录下，压缩包内的FalagardSkinning.pdf文件）。读者也可以参考CEGUI例子资源附带的一些外观定义的文件。

6.3 外观定义的例子

本节介绍如何从零开始，编写一个外观的定义。一个文件可以定义任意多的外观，但如果定义的多了可能会是文件变的非常庞大，不容易阅读。如果可以还是多分几个文件，这样便于阅读。读者知道这些文件要在那里注册吗？资源管理的总控文件scheme文件里啊。

外观文件也是XML文件，下面是一个空的外观文件的定义。

```
<?xml version="1.0" ?>
<Falagard>
</Falagard>
```

标准的XML文件，Falagard作为这个文件的根元素。所有其他的元素都必须在它的内部。其实根元素的内部只应该出现WidgetLook元素。这个元素定义一个外观。本书以一个按钮的外观定义，介绍如何编写外观文件。下面是空外观的定义。

```
<WidgetLook name="TaharezLook/Button">
</WidgetLook>
```

WidgetLook内部只容许6.2中介绍的7种元素。在Falagard/Button的渲染窗口中用到了四个StateImagery，分别是Normal，Hover，Pushed和Disabled。我们填写这四个状态图像。

```
<WidgetLook name="TaharezLook/Button">
    <StateImagery name="Normal">
    </StateImagery>
    <StateImagery name="Hover">
    </StateImagery>
    <StateImagery name="Pushed">
    </StateImagery>
    <StateImagery name="Disabled">
    </StateImagery>
</WidgetLook>
```

我们知道StateImagery 其实是ImagerySection的集合。真正可以描绘的还是ImagerySection。所以我们添加几个。目前的状态如下。

```
<WidgetLook name="TaharezLook/Button">
    <ImagerySection name="normal imagery">
    </ImagerySection>
    <ImagerySection name="hover imagery">
    </ImagerySection>
    <ImagerySection name="pushed imagery">
    </ImagerySection>
    <StateImagery name="Normal">
    </StateImagery>
    <StateImagery name="Hover">
    </StateImagery>
    <StateImagery name="Pushed">
    </StateImagery>
    <StateImagery name="Disabled">
    </StateImagery>
</WidgetLook>
```

下一步就是定义每个ImagerySection，按钮应该不需要太复杂的定义，我们这里选择ImageryComponent一个单一的图像来表示一个按钮。其中一个定义如下。

```
...
<ImagerySection name="normal imagery">
    <ImageryComponent>
    </ImageryComponent>
```

```
</ImagerySection>
...
```

ImageryComponent需要定义图像和矩形区域的位置，所以先添加一个区域对象，这个对象不需要是命名的。

```
<ImageryComponent>
    <Area>
    </Area>
</ImageryComponent>
```

添加了区域自然要定义区域的位置了，第6.2.6节介绍了如何定义一个区域。

```
<ImageryComponent>
<Area>
    <Dim type="LeftEdge">
        <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge">
        <AbsoluteDim value="0" />
    </Dim>
    <Dim type="Width">
     <ImageDim imageset="TaharezLook" image="ButtonLeftNormal" dimension="Width"/>
    </Dim>
    <Dim type="Height">
        <UnifiedDim scale="1.0" type="Height" />
    </Dim>
</Area>
</ImageryComponent>
```

这个矩形定义的位置为按钮的原点位置，因为它定义了两个绝对像素都是0。区域的宽度是图像的宽度，高度是窗口的高度。下一步就是给元素添加图像的定义了。

```
...
<Image imageset="TaharezLook" image="ButtonLeftNormal" />
...
```

图像的定义非常简单。上面定义矩形的时候宽度是等于图片的宽度的，但高度是等于窗口的高度，有可能窗口的高度和图像的高度不相同，所以要设置竖直方向的图像格式。

```
...
<VertFormat type="Stretched" />
...
```

设置竖直方向拉伸。最终所得的代码如下。

```
<ImageryComponent>
    <Area>
        <Dim type="LeftEdge">
            <AbsoluteDim value="0" />
        </Dim>
        <Dim type="TopEdge">
            <AbsoluteDim value="0" />
        </Dim>
        <Dim type="Width">
            <ImageDim imageset="TaharezLook" image="ButtonLeftNormal"
                    dimension="Width"/>
        </Dim>
        <Dim type="Height">
            <UnifiedDim scale="1.0" type="Height" />
        </Dim>
    </Area>
```

```
    <Image imageset="TaharezLook" image="ButtonLeftNormal" />
    <VertFormat type="Stretched" />
</ImageryComponent>
```

这一段定义了正常状态下的ImageryComponent，其他状态我们这里就不举例了，他们大同小异，可能就是区域的算法不太一样而已。到这里应该已经定义了三个ImagerySerction，按理说应该是四个，因为是四种状态的按钮吗？但CEGUI资源没有提供禁用状态的图像，所以这里只有三种。但是StateImagery却必须是四种，因为这是渲染窗口的需要。虽然没有禁用的图像，但是可以使用其他的图像来代替，比如正常状态的图像。

下面定义一个StateImagery ，这是正常的状态。需要注意的是这里并没有定义文字的显示的section。

```
<StateImagery name="Normal">
    <Layer>
        <Section section="normal imagery" />
    </Layer>
</StateImagery>
```

其他四个也类似，这里就不做介绍了。下面介绍给按钮添加文字的显示定义。文字也需要定义一个ImagerySection ，只不过使用的是TextComponent元素而已。

```
...
<ImagerySection name="label">
    <TextComponent>
    </TextComponent>
</ImagerySection>
...
```

添加好框架，就可以开始定义TextComponent，首先还是要定义区域。

```
<Area>
<Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
<Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
<Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
<Dim type="Height"><UnifiedDim scale="1" type="Height"/></Dim>
</Area>
```

显然这个区域是真个的按钮的大小。文本的定义还有一些非常重要的地方，比如文字对齐格式的定义。如果这里不定义则使用窗口的默认定义。

```
<VertFormat type="CentreAligned" />
<HorzFormat type="WordWrapCentreAligned" />
```

这两个定义，竖直方向居中显示，水平方向居中换行显示。最终的文字显示定义如下。

```
<ImagerySection name="label">
    <TextComponent>
        <Area>
            <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
            <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
            <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
            <Dim type="Height"><UnifiedDim scale="1" type="Height"/></Dim>
        </Area>
    <VertFormat type="CentreAligned" />
    <HorzFormat type="WordWrapCentreAligned" />
    </TextComponent>
</ImagerySection>
```

定义好后需要修改StateImagery，使之可以显示文字。

```
<StateImagery name="Normal">
    <Layer>
        <Section section="normal imagery" />
```

```
            <Section section="label" />
        </Layer>
</StateImagery>
```

另外三个StateImagery 的定义类似，这里就不做介绍了。读者可以自己编写其他的几个StateImagery 和ImagerySection 。

6.4 外观的程序实现

第6.2节介绍了各种外观定义的元素，这一节介绍他们在C++中的实现。外观文件的处理类是Falagard_xmlHandler，这个类是外观定义的入口类。这个处理类非常庞大，不过思路还是和其他的处理类一样，读者有兴趣可以自己阅读。本节将按照第6.2节的顺序，介绍元素在C++中的实现。外观定义的C++类和外观的处理类定义在CEGUIBase工程中的include和Src过滤器中的Falagard过滤器下面。

6.4.1 WidgetLookFeel类

这个类中包含七种元素的集合，这七种元素对应七种类。我们先从类的数据成员开始。

```
//对应Property元素，属性的初始化集合的定义
typedef std::vector<PropertyInitialiser>  PropertyList;
//对应PropertyDefinition元素，属性定义集合的定义
typedef std::vector<PropertyDefinition>   PropertyDefinitionList;
//对应PropertyLinkDefinition元素，属性的链接集合的定义
typedef std::vector<PropertyLinkDefinition> PropertyLinkDefinitionList;
//对应StateImagery元素，图像状态集合的定义
typedef std::map<String, StateImagery, String::FastLessCompare>    StateList;
//对应ImagerySection元素，图像集合的定义
typedef std::map<String, ImagerySection, String::FastLessCompare>  ImageryList;
//对应NamedArea元素，命名区域的集合定义
typedef std::map<String, NamedArea, String::FastLessCompare>       NamedAreaList;
//对应Child元素，自动子窗口的元素集合定义
typedef std::vector<WidgetComponent>      WidgetList;
//保存外观的名称
CEGUI::String   d_lookName;
//保存图像集
ImageryList     d_imagerySections;
//子窗口的集合
WidgetList      d_childWidgets;
//状态图像的集合
StateList       d_stateImagery;
//初始化的属性集合
PropertyList    d_properties;
//命名区域的集合
NamedAreaList   d_namedAreas;
//属性定义的集合
mutable PropertyDefinitionList  d_propertyDefinitions;
//属性链接的定义集合
mutable PropertyLinkDefinitionList d_propertyLinkDefinitions;
```

这些数据大多是一些集合，这些集合都是WidgetLook元素的子元素的集合。在解析外观文件的时候会将每个定义的元素添加到这些集合里面，供以后使用。

这个类的成员函数无非是向这些集合里添加对应的元素或者清空这些集合，集合元素存在性判断等。这些都不是非常难理解，这里不做介绍了，只介绍外观是如何映射到一个窗

口上的。窗口的基类Window里有一个setLookNFeel函数，它会调用WidgetLookFeel里的initialiseWidget，这个函数会应用外观定义到目标窗口。

```
//这个函数应用外观定义里的内容到widget窗口中
void WidgetLookFeel::initialiseWidget(Window& widget) const
{
    // 添加外观定义的子窗口到应用外观的窗口，遍历子窗口集合
    for(WidgetList::const_iterator curr = d_childWidgets.begin(); curr != d_childWidgets.end();
++curr)
    {
        (*curr).create(widget);
    }
    // 添加新的属性到目标窗口，遍历属性定义集合
    for(PropertyDefinitionList::iterator propdef = d_propertyDefinitions.begin(); propdef !=
        d_propertyDefinitions.end(); ++propdef)
    {
        // 添加属性到目标窗口
        widget.addProperty(&(*propdef));
        //设置属性的初始值
        widget.setProperty((*propdef).getName(), (*propdef).getDefault(&widget));
    }
    // 添加属性的定义到目标窗口，遍历属性链接集合
    for(PropertyLinkDefinitionList::iterator linkdef = d_propertyLinkDefinitions.begin();
        linkdef != d_propertyLinkDefinitions.end(); ++linkdef)
    {
        //添加属性到窗口
        widget.addProperty(&(*linkdef));
        // 设置属性的默认值
        widget.setProperty((*linkdef).getName(), (*linkdef).getDefault(&widget));
    }
    // 设置属性的初始值到目标窗口，遍历属性初始化集合
    for(PropertyList::const_iterator prop = d_properties.begin(); prop != d_properties.end();
        ++prop)
    {
        (*prop).apply(widget);
    }
}
```

这个函数应用外观定义的所有内容，包括子窗口，属性定义，属性初始化，属性链接等。这个操作是在设置窗口的外观（setLookNFeel）的时候执行的。其他的函数就不介绍了。

每当窗口附加一个外观的时候，外观中定义的各种内容会附加到目标窗口上，如果先前窗口有外观，则先前的外观被反加载，然后在重新加载现在的外观。

6.4.2 ImagerySection类

ImagerySection元素对应的类也叫ImagerySection。它是一些可以重用的可渲染组件的集合。这些可渲染的组件有ImageryComponent，TextComponent，FrameComponent三种。第6.2节也介绍了它们如何使用。它的重用性体现在StateImagery可以使用他们，外观之间可以重用ImagerySection的定义（通过Section来重用指定）。通过上面的分析，读者应该可以想到这个类的成员变量了。

```
//定义三个集合类型
typedef std::vector<ImageryComponent> ImageryList;
typedef std::vector<TextComponent> TextList;
typedef std::vector<FrameComponent> FrameList;
```

```
//保存这个ImagerySection的名称，通过名称可以重用（外部重用的时候需要外观名称）
CEGUI::String      d_name;
//ImagerySection内部的颜色，如果没有定义颜色则使用这个颜色
CEGUI::ColourRect   d_masterColours;
//框架的集合
FrameList          d_frames;
//图像组件集合
ImageryList        d_images;
//文本组件集合
TextList           d_texts;
//如果颜色值从属性中获取则这个变量保存属性的名称
String             d_colourPropertyName;
//是否颜色属性是颜色矩形（就是四个颜色值，对应每个区域的四个顶点的颜色值）
bool               d_colourProperyIsRect;
```

这个类只介绍渲染函数，有两个渲染函数，大体上是相同的只是参数不太相同。这里介绍其中的一个。

```
void ImagerySection::render(Window& srcWindow, const Rect& baseRect, float base_z, const CEGUI::ColourRect* modColours, const Rect* clipper, bool clipToDisplay) const
{
   // 获取最终的描绘颜色
   ColourRect finalCols;
   initMasterColourRect(srcWindow, finalCols);
   if (modColours)
      finalCols *= *modColours;
   ColourRect* finalColsPtr = (finalCols.isMonochromatic() &&
finalCols.d_top_left.getARGB() == 0xFFFFFFFF) ? 0 : &finalCols;

   //渲染这个类中的所有的框架组件（Component）
   for(FrameList::const_iterator frame = d_frames.begin(); frame != d_frames.end(); ++frame)
   {
      (*frame).render(srcWindow, baseRect, base_z, finalColsPtr, clipper, clipToDisplay);
   }
   //渲染所有的图像组件
   for(ImageryList::const_iterator image = d_images.begin(); image != d_images.end();
      ++image)
   {
      (*image).render(srcWindow, baseRect, base_z, finalColsPtr, clipper, clipToDisplay);
   }
   //渲染所有的文字组件
   for(TextList::const_iterator text = d_texts.begin(); text != d_texts.end(); ++text)
   {
      (*text).render(srcWindow, baseRect, base_z, finalColsPtr, clipper, clipToDisplay);
   }
}
```

这个类是所有的组件的集合，它本身并不是可以渲染的单元，单独的渲染单元是这三个组件。这三个框架类非常复杂，这里就不做介绍了，有兴趣的读者自己阅读。

6.4.3 StateImagery类

这个类是一些层的集合，它保存所有的层。渲染的时候调用所有的层的渲染函数。下面是它的成员变量，非常简单只有层的一个集合。

```
typedef std::multiset<LayerSpecification> LayersList;
//状态的名称
```

```
CEGUI::String   d_stateName;
//层集合的变量
LayersList      d_layers;
//是否裁剪到目标窗口
bool            d_clipToDisplay;
```

这个类有主要的函数，渲染函数。函数中用到了层类的渲染函数。

```
void StateImagery::render(Window& srcWindow, const Rect& baseRect, const ColourRect* modcols, const Rect* clipper) const
{
   float base_z;
   //渲染所有在这个状态中的所有层
   for(LayersList::const_iterator curr = d_layers.begin(); curr != d_layers.end(); ++curr)
   {
      // 计算层的Z值，我们曾经说过Z值是基本没有的
      base_z = -0.0000001f * static_cast<float>((*curr).getLayerPriority());
      (*curr).render(srcWindow, baseRect, base_z, modcols, clipper, d_clipToDisplay);
   }
}
```

这个函数主要是调用LayerSpecification的渲染函数。下面介绍这个类。

LayerSpecification是StateImagery元素的子元素。它可以包含多个SectionSpecification对象（对应Section对象）。它的数据成员同样简单，只包含一个SectionList的变量一个层渲染的优先级。

变量的定义如下。

```
typedef std::vector<SectionSpecification> SectionList;
//定义层中的ImagerySection的集合变量
SectionList d_sections;
//这个层的优先级，决定层的渲染次序
uint        d_layerPriority;
```

这个类对应的渲染函数。

```
void LayerSpecification::render(Window& srcWindow, float base_z, const ColourRect* modcols, const Rect* clipper, bool
clipToDisplay) const
{
   // 渲染层中的所有Section对象
   for(SectionList::const_iterator curr = d_sections.begin(); curr != d_sections.end(); ++curr)
   {
      (*curr).render(srcWindow, base_z, modcols, clipper, clipToDisplay);
   }
}
```

到这里渲染流转到了SectionSpecification中，顺便我们介绍这个类。

```
//保存拥有这个ImagerySection的外观的名称
String          d_owner;
//ImagerySection的名称，在d_owner中定义的ImagerySection
String          d_sectionName;
//保存的颜色矩阵，当d_usingColourOverride为true的时候会使用这个值作为段的颜色
ColourRect      d_coloursOverride;
//是否使用 d_coloursOverride
bool            d_usingColourOverride;
//颜色属性的名称，如果颜色是从属性中获取的则这个是颜色的属性名称
String          d_colourPropertyName;
//判断是否颜色属性给的值是完整的颜色矩形
bool            d_colourProperyIsRect;
```

```
//是否渲染这个段的标志，如果为true则渲染这个段
String        d_renderControlProperty;
```

它的渲染函数会调用ImagerySection类的渲染函数。下面是段的渲染函数。

```
void SectionSpecification::render(Window& srcWindow, const Rect& baseRect, float base_z, const ColourRect* modcols, const Rect*
clipper, bool clipToDisplay) const
{
  try
  {
  // 获取ImagerySection的指针，这个对象在外观d_owner中的d_secti
onName图像层
  const ImagerySection* sect =
            &WidgetLookManager::getSingleton().getWidgetLook(d_owner).getImagerySection(d_secti
onName);
  //决定最终使用的颜色
  ColourRect finalColours;
      initColourRectForOverride(srcWindow, finalColours);
      finalColours.modulateAlpha(srcWindow.getEffectiveAlpha());
      if (modcols)
          finalColours *= *modcols;
      // 渲染这个图像段
      sect->render(srcWindow, baseRect, base_z, &finalColours, clipper, clipToDisplay);
  }
  // 捕获可能发生的异常，比如找不到这个图像段，这个异常不致命，所以不用处理
  catch (Exception&)
  {}
}
```

到这里图像状态的渲染流程就清楚了。StateImagery到LayerSpecification到SectionSpecification最终到了ImagerySection。从这里也可以看出SectionSpecification类（对应Section元素）的作用。

6.4.4 属性相关的三个元素

属性相关的类有PropertyInitialiser（对应Property元素），PropertyDefinition（对应PropertyDefinition元素），PropertyLinkDefinition（对应PropertyLinkDefinition元素）。后两个类派生自PropertyDefinitionBase类，PropertyDefinitionBase类继承自Property类。现在应该明白在widgetLookFeel类中initialiseWidget函数直接将PropertyLinkDefinition和PropertyDefinition的对象作为属性来使用了吧。

下面介绍PropertyInitialiser的apply函数，这个函数设置目标窗口的属性。

```
void PropertyInitialiser::apply(CEGUI::PropertySet& target) const
{
 try
 {
  target.setProperty(d_propertyName, d_propertyValue);
 }
 // 容忍不存在属性的异常，也就是可以设置不存在的属性
 catch (UnknownObjectException&)
 {}
}
```

这个函数也是在initialiseWidget中被调用了。

PropertyLinkDefinition元素我们说过是链接到子窗口的属性，现在我们介绍它是怎么做到的。

```
//获取函数，这是个虚函数，也就是说PropertyLinkDefinition定义的属性会调用这个函数
//这个函数获取目标窗口（某个子窗口）获取目标窗口的属性值
```

```
String PropertyLinkDefinition::get(const PropertyReceiver* receiver) const
{
   return getTargetWindow(receiver)->getProperty(d_targetProperty.empty() ? d_name :
      d_targetProperty);
}
//设置属性的值，这个函数也是个虚函数，它设置目标窗口的属性值
void PropertyLinkDefinition::set(PropertyReceiver* receiver, const String& value)
{
   getTargetWindow(receiver)->setProperty(d_targetProperty.empty() ? d_name :
      d_targetProperty, value);
   PropertyDefinitionBase::set(receiver, value);
}
//获取两个目标窗口
const Window* PropertyLinkDefinition::getTargetWindow(const PropertyReceiver* receiver) const
{
      //如果不是子窗口，则使用自己作为目标窗口
   if (d_widgetNameSuffix.empty())
      return static_cast<const Window*>(receiver);
      //查找子窗口并且返回
   return WindowManager::getSingleton().getWindow(
         static_cast<const Window*>(receiver)->getName() + d_widgetNameSuffix);
}
Window* PropertyLinkDefinition::getTargetWindow(PropertyReceiver* receiver)
{
   return const_cast<Window*>(static_cast<const PropertyLinkDefinition*>(this)->
getTargetWindow(receiver));
}
```

类PropertyLinkDefinition派生自Property类，它实现了Property的两个虚函数get和set，这样当获取和设置PropertyLinkDefinition链接的属性的时候自动的获取目标窗口的属性。

PropertyDefinition定义一个属性，它提供了两个函数get和set。窗口从外观文件中定义的属性是通过UserString实现的。Window有一个d_userStrings的字符串映射，一个字符串映射到另一个字符串（属性名映射到属性值）。

```
//从用户字符串中获取属性值
String PropertyDefinition::get(const PropertyReceiver* receiver) const
{
   return static_cast<const Window*>(receiver)->getUserString(d_userStringName);
}
//设置属性值到用户字符串中
void PropertyDefinition::set(PropertyReceiver* receiver, const String& value)
{
   static_cast<Window*>(receiver)->setUserString(d_userStringName, value);
   PropertyDefinitionBase::set(receiver, value);
}
```

PropertyDefinitionBase的set函数激发重绘和重新布局。

```
void PropertyDefinitionBase::set(PropertyReceiver* receiver, const String& value)
{
   //如果这个定义的属性影响布局，则激发重新布局
      if (d_writeCausesLayout)
      static_cast<Window*>(receiver)->performChildWindowLayout();
      //如果这个定义的属性影响描绘，则激发重新描绘
   if (d_writeCausesRedraw)
```

```
    static_cast<Window*>(receiver)->requestRedraw();
}
```

当定义属性的时候，设置了redrawOnWrite="true"或者redrawOnLayout="true"的时候才会激发这两个调用。

6.4.5 自动子窗口

自动子窗口也是一个某种类型（具体类型有外部窗口类型决定）窗口。它的实现类是WidgetComponent首先我们介绍它的成员变量。

```
typedef std::vector<PropertyInitialiser> PropertiesList;
//目标区域对象，相对于它的父窗口的区域
ComponentArea   d_area;
//窗体的外部类型，创建窗口时需要使用的类型
String   d_baseType;
//WidgetLookFeel 的名称，子窗口使用的外观名称
String   d_imageryName;
//自动子窗口的后缀名
String   d_nameSuffix;
//渲染窗口的类型，用来查找渲染窗口
String   d_rendererType;
//竖直对齐方式
VerticalAlignment    d_vertAlign;
//水平对齐方式
HorizontalAlignment  d_horzAlign;
//属性集定义，用来初始化子窗口的属性
PropertiesList  d_properties;
```

这个函数有个create函数负责创建子窗口。

```
void WidgetComponent::create(Window& parent) const
{
   // 生成最终的子窗口名称，它等于父窗口名称加上自己的窗口后缀名
   String widgetName = parent.getName() + d_nameSuffix;
   Window* widget = WindowManager::getSingleton().createWindow(d_baseType,
      widgetName);
   // 设置窗口的渲染窗口
   if (!d_rendererType.empty())
      widget->setWindowRenderer(d_rendererType);
   // 设置窗口的外观，会调用到外观的初始化函数（initialiseWidget）
   if (!d_imageryName.empty())
      widget->setLookNFeel(d_imageryName);
   //作为子窗口添加到父窗口
   parent.addChildWindow(widget);
   // 设置对齐方式
   widget->setVerticalAlignment(d_vertAlign);
   widget->setHorizontalAlignment(d_horzAlign);
   // 应用初始的属性（Property指定的初始值）
   for(PropertiesList::const_iterator curr = d_properties.begin(); curr != d_properties.end();
      ++curr)
   {
      (*curr).apply(*widget);
   }
}
```

另一个重要的函数就是布局函数了。这个函数设置子窗口的区域

```
void WidgetComponent::layout(const Window& owner) const
```

```
{
  try
  {
        //获取子窗口的位置
    Rect pixelArea(d_area.getPixelRect(owner));
    URect window_area(cegui_absdim(pixelArea.d_left),
                      cegui_absdim(pixelArea.d_top),
                      cegui_absdim(pixelArea.d_right),
                      cegui_absdim(pixelArea.d_bottom));
        //获取子窗口根据名称
    Window* wnd = WindowManager::getSingleton().getWindow(owner.getName() +
        d_nameSuffix);
        //设置窗口的区域信息
    wnd->setArea(window_area);
    wnd->notifyScreenAreaChanged();
  }
  catch (UnknownObjectException&)
  {}
}
```

子窗口的创建和布局是最重要的，其他的函数这里就不做介绍了。

6.4.6 区域的定义

区域相关各个元素都定义在CEGUIFalDimensions.h和CEGUIFalDimensions.cpp两个文件中。区域（Area）元素对应类ComponentArea。前面讲过区域可以有四个维度（Dimension）元素或者区域的值从属性获取，这个属性必须是URect类型的数值。各种Dim类型的基类是BaseDim，这个类是个虚基类，它设计了一些接口，派生自它的各种维度描述类都必须实现这个接口。加减乘除四种操作使用DimensionOperator宏来定义，不同的类型执行不同的运算。下面先介绍BaseDim类。这个类有两个成员变量一个是操作类型，一个是操作数。

```
//操作类型，只有加减乘除四种
DimensionOperator   d_operator;
//和自己操作的另一个维度数
BaseDim*            d_operand;
```

看来这个基类就已经支持维度的操作了。这个类提供了三个重要的纯虚虚函数，子类必须实现。他们是两个参数不同的获取值的函数，一个拷贝函数。

```
//两个获取值的函数，各个子类的获取方法是不同的
virtual float getValue_impl(const Window& wnd, const Rect& container) const = 0;
virtual float getValue_impl(const Window& wnd) const = 0;
//一个拷贝函数的实现，用来创建一个对象
virtual BaseDim* clone_impl() const = 0;
```

这个类实现了两个维度的运算，函数getValue实现了具体的操作。

```
//获取维度的值，如果有操作符则执行操作
float BaseDim::getValue(const Window& wnd) const
{
  // 获取子类的具体实现值
  float val = getValue_impl(wnd);
  // 如果我们有操作数则执行操作，加减乘除四种运算
  if (d_operand)
  {
    switch (d_operator)
    {
    case DOP_ADD:
```

```
            val += d_operand->getValue(wnd);
            break;
        case DOP_SUBTRACT:
            val -= d_operand->getValue(wnd);
            break;
        case DOP_MULTIPLY:
            val *= d_operand->getValue(wnd);
            break;
        case DOP_DIVIDE:
            val /= d_operand->getValue(wnd);
            break;
        default:
            // 默认不执行如何操作
            break;
        }
    }
        //返回最终的结果
    return val;
}
```

下面介绍两个维度的实现，一个是AbsoluteDim另一个是ImageDim其余的读者自己阅读代码。AbsoluteDim和我们说的一个坐标的值一样，所以它只有一个浮点型的成员，表示它的值。下面两个是实现的虚接口（省略一个获取值的接口）。

```
//获取值的具体实现，直接返回保存的数值
float AbsoluteDim::getValue_impl(const Window& wnd, const Rect& container) const
{
    return d_val;
}
//生成一个新的元素并返回
BaseDim* AbsoluteDim::clone_impl() const
{
    AbsoluteDim* ndim = new AbsoluteDim(d_val);
    return ndim;
}
```

ImageDim的通过获取图像的数据来返回维度值，它的数据成员定义如下。

```
String d_imageset;          //图像集的名称
String d_image;             //图像的名称，这两个可以唯一确定一个图像
DimensionType d_what;       //决定获取图像的那个值作为输出数据
```

它的两个接口实现。

```
float ImageDim::getValue_impl(const Window& wnd) const
{
        //第一步，获取图像
    const Image* img = &ImagesetManager::getSingleton().getImageset(d_imageset)->
getImage(d_image);
        //第二步，根据获取的类型，返回获取的值
    switch (d_what)
    {
                //这四个类型是图像的四种属性
        case DT_WIDTH:
            return img->getWidth();
            break;
        case DT_HEIGHT:
```

```
            return img->getHeight();
            break;
        case DT_X_OFFSET:
            return img->getOffsetX();
            break;
        case DT_Y_OFFSET:
            return img->getOffsetY();
            break;
        // 下面的几种是图像在图像集上的相关数据（位置数据）
        case DT_LEFT_EDGE:
        case DT_X_POSITION:
            return img->getSourceTextureArea().d_left;
            break;
        case DT_TOP_EDGE:
        case DT_Y_POSITION:
            return img->getSourceTextureArea().d_top;
            break;
        case DT_RIGHT_EDGE:
            return img->getSourceTextureArea().d_right;
            break;
        case DT_BOTTOM_EDGE:
            return img->getSourceTextureArea().d_bottom;
            break;
        default:
            throw InvalidRequestException("ImageDim::getValue - unknown or unsupported
                DimensionType encountered.");
            break;
    }
}
//简单的创建一个新的对象并返回
BaseDim* ImageDim::clone_impl() const
{
    ImageDim* ndim = new ImageDim(d_imageset, d_image, d_what);
    return ndim;
}
```

由于篇幅限制，这里不在介绍其他的类，读者可以自己学习。

外观定义中的Dim元素对应Dimension类。下面介绍它的成员。

```
BaseDim*        d_value;    //这个维度的值，注意这里使用的是基类的指针
DimensionType   d_type;     //维度的类型
```

这个类是各种维度的包装，没有什么太重要的函数，这里不做介绍。最后介绍ComponentArea类，它描述一个区域，其实它像是一个URect的数。成员变量如下所示。

```
//左边，相当于X轴位置
Dimension d_left;
//右边，相当于Y轴的位置
Dimension d_top;
//右边或者宽度，对应两种类型的矩形
Dimension d_right_or_width;
//底边或者高度，对应两种类型的矩形
Dimension d_bottom_or_height;
//属性的名称，这个属性对应的值必须是URect类型的
String  d_areaProperty;
```

这个类有非常重要的函数getPixelRect，获取这个区域。这个函数有两个重载函数，这里介绍其中之一。

```
Rect ComponentArea::getPixelRect(const Window& wnd) const
{
   Rect pixelRect;
   // 是否从属性中获取
   if (isAreaFetchedFromProperty())
   {
      pixelRect =  PropertyHelper::stringToURect(wnd.getProperty(d_areaProperty)).
            asAbsolute(wnd.getPixelSize());
   }
   // 通过使用维度来计算矩形
   else
   {
      // 先做检查，是否里面的数据可以代表一个矩形
      assert(d_left.getDimensionType() == DT_LEFT_EDGE ||
            d_left.getDimensionType() == DT_X_POSITION);
      assert(d_top.getDimensionType() == DT_TOP_EDGE ||
            d_top.getDimensionType() == DT_Y_POSITION);
      assert(d_right_or_width.getDimensionType() == DT_RIGHT_EDGE
            ||d_right_or_width.getDimensionType() == DT_WIDTH);
      assert(d_bottom_or_height.getDimensionType() == DT_BOTTOM_EDGE
            || d_bottom_or_height.getDimensionType() == DT_HEIGHT);
            //首先获取位置信息
      pixelRect.d_left = d_left.getBaseDimension().getValue(wnd);
      pixelRect.d_top = d_top.getBaseDimension().getValue(wnd);
            //其次获取宽度信息或者右边信息
      if (d_right_or_width.getDimensionType() == DT_WIDTH)
         pixelRect.setWidth(d_right_or_width.getBaseDimension().getValue(wnd));
      else
         pixelRect.d_right = d_right_or_width.getBaseDimension().getValue(wnd);
            //最后获取高度信息或者底边信息
      if (d_bottom_or_height.getDimensionType() == DT_HEIGHT)
         pixelRect.setHeight(d_bottom_or_height.getBaseDimension().getValue(wnd));
      else
         pixelRect.d_bottom = d_bottom_or_height.getBaseDimension().getValue(wnd);
   }
      //返回获取的矩形
   return pixelRect;
}
```

这个函数根据维度的类型，计算出最终的矩形。读者需要明白两种矩形的不同，最终返回的都是四条边构成的矩形。读者只要明白最终返回的是屏幕的矩形坐标，就可以了。

这小节就介绍到这里，这一节主要介绍了外观文件中的各种元素在CEGUI中的实现，读者完全可以不懂这些，就可以编写出正确的外观文件，但如果读者明白了其中的原理，然后在设计外观定义就会得心应手了。

6.5 本章小结

本章介绍了外观文件的定义，介绍了按钮外观定义的例子，最后还介绍了C++的具体实现。这章的内容比较复杂，而且很重要，如果读者希望自己编写个控件则必须熟悉这章。读者需要好好消化。

1.读者自己编写一个框架的外观定义，要求这个框架支持滚动条。读者可以参考CEGUI提供的例子中的外观定义。

2.了解外观定义的CEGUI实现，明白其中的原理，一般来说读者并不需要非常的清楚具体的细节，只要知道原理就可以了。

3.学习本章没有涉及到的相关的代码（有兴趣的读者可以阅读）。

第7章 CEGUI控件介绍

CEGUI实现了大多数常用的控件，控件的实现代码都在CEGUIBase工程下的include或者Src过滤器下面的element过滤器下。通过学习这些控件的实现，对读者以后编写控件来说非常有好处。所以这里设计了一章，希望能够提高读者控件的设计能力。CEGUI实现的控件非常的多，这里不可能一一说明，我们挑选了几个有代表性，比较简单的控件介绍一下。读者务必自己阅读其他的控件实现，如果读者希望自己设计控件的话。所有的控件都必须派生自Window类，各个控件根据功能的不同可能会重载一些重要的虚函数，比如鼠标操作相关的，渲染相关的，状态更新相关的等。总之所有的更改都是在读者已经明白Window类的各个函数的功能，以及窗口之间的关系的基础之上的。这一章主要介绍控件逻辑关于这些窗口是如何被渲染的将在第10章详细介绍。

7.1 按钮控件

按钮是UI中的最基本的控件了，这里安排在第一节介绍，也就是体现它的重要性。这一节将会介绍CEGUI官方实现的所有的按钮。

7.1.1 按钮基类

按钮包含许多种，有CheckBox，TabButton，PushButton，RadioButton等。这些按钮都有许多相同的特点，所以CEGUI提供了按钮的基类ButtonBase类。哪么我们就以ButtonBase开始。首先看它的数据成员，它只有两个成员变量，表示两种按钮状态。

```
bool d_pushed;                  //当用户单击这个按钮的时候为true
bool d_hovering;                //当鼠标在按钮上的时候为true
```

按钮主要是处理鼠标消息，它不接受键盘消息，所以这个类重载了一些鼠标相关的消息。

```
//当鼠标在窗口上移动的时候
void ButtonBase::onMouseMove(MouseEventArgs& e)
{
    //基类的处理
    Window::onMouseMove(e);
    //更新按钮的内部状态
    updateInternalState(e.position);
    e.handled = true;
}
//更新内部状态，鼠标是否在按钮上的检测
void ButtonBase::updateInternalState(const Point& mouse_pos)
{
    bool oldstate = d_hovering;
    // 假设鼠标不在窗口上
    d_hovering = false;
    //如果我们不是捕获鼠标事件（哪么我们怎么会收到鼠标事件的消息呢？可能我们是
//捕获鼠标的父窗口。）的窗口，但鼠标在我们上面，而且在我们的区域内部（IsHit）
    const Window* capture_wnd = getCaptureWindow();
    if (capture_wnd == 0)
    {
        System* sys = System::getSingletonPtr();
        if (sys->getWindowContainingMouse() == this && isHit(mouse_pos))
        {
            d_hovering = true;
        }
  }
    //否则我们是捕获的窗口，如果鼠标在窗口的区域内则鼠标高亮
  else if (capture_wnd == this && isHit(mouse_pos))
```

```
    {
        d_hovering = true;
    }
        // 如果状态改变了则重绘，这会导致两层渲染缓冲都失效
        if (oldstate != d_hovering)
        {
            requestRedraw();
        }
}
//当鼠标按钮按下的时候会调用
void ButtonBase::onMouseButtonDown(MouseEventArgs& e)
{
        // 默认处理
        Window::onMouseButtonDown(e);
        //如果鼠标左键按下
        if (e.button == LeftButton)
        {
            //捕获鼠标
            if (captureInput())
            {
                //设置为按下状态，并更新内部状态
                d_pushed = true;
                updateInternalState(e.position);
                requestRedraw();
            }
            //设置事件被处理过的标志
            e.handled = true;
        }

}
//鼠标抬起是会被调用
void ButtonBase::onMouseButtonUp(MouseEventArgs& e)
{
        Window::onMouseButtonUp(e);
        if (e.button == LeftButton)
        {
            //释放捕获，鼠标已经抬起了，releaseInput会引发onCaptureLost函数的调用
            releaseInput();
            e.handled = true;
        }
}
//当捕获丢失的时候会调用
void ButtonBase::onCaptureLost(WindowEventArgs& e)
{
        Window::onCaptureLost(e);
        //按下的状态解除
        d_pushed = false;
        updateInternalState(MouseCursor::getSingletonPtr()->getPosition());
        requestRedraw();
        e.handled = true;
}
//当鼠标离开按钮的时候被调用
```

```
void ButtonBase::onMouseLeaves(MouseEventArgs& e)
{
    Window::onMouseLeaves(e);
    //高亮显示状态取消
    d_hovering = false;
    requestRedraw();
    e.handled = true;
}
```

按钮基类重载了鼠标的几个事件函数，主要目的是设置按钮的两种状态，一种高亮，一种按下的状态。ButtonBase不支持按钮的单击事件，它只设置状态，具体做什么各个派生类自己来决定。

7.1.2 普通按钮控件

下面介绍PushButton，一个可以单击的按钮。单击就要产生单击的事件，我们说过CEGUI的事件其实就是一个字符串，当事件激发的时候，会调用这个事件对应的事件函数。所以要定义一个事件就是定义一个字符串，这个字符串应该是常量而且是静态的，因为一个类只需要一个事件的定义。

```
const String PushButton::EventNamespace("PushButton");
const String PushButton::WidgetTypeName("CEGUI/PushButton");
const String PushButton::EventClicked( "Clicked" );
```

EventNamespace表示事件的空间，就是这个事件是从何发出的。WidgetTypeName表示这个控件的类型，它也是类厂的类型，在FalagardMapping中表示内部类型。EventClicked表示单击的事件，当用户单击按钮的时候会激发这个事件，调用所有的关心这个事件的函数。

这个类派生自ButtonBase，重载了onMouseButtonUp事件，新增了一个onClicked的单击处理函数。具体定义如下。

```
//单击事件，激发事件调用所有的事件处理函数
void PushButton::onClicked(WindowEventArgs& e)
{
    fireEvent(EventClicked, e, EventNamespace);
}
//重载鼠标抬起的事件
void PushButton::onMouseButtonUp(MouseEventArgs& e)
{
    if ((e.button == LeftButton) && isPushed())
    {
        Window* sheet = System::getSingleton().getGUISheet();
        if (sheet)
        {
            //如果鼠标在这个按钮上释放则激发单击事件
            if (this == sheet->getTargetChildAtPosition(e.position))
            {
                WindowEventArgs args(this);
                onClicked(args);
            }
        }
        e.handled = true;
    }
    // ButtonBase的默认处理，设置按钮的内部状态
    ButtonBase::onMouseButtonUp(e);
}
```

这个类的功能大家应该很清楚了，就是添加了当用户单击的时候的单击事件。

7.1.3 Tab控件

Tab读者应该非常的清楚，标签的切换。CEGUI提供了一个Tab的例子，读者可以在光盘上找到。这个类有三个事件，分别代表按钮被单击，拖到按钮和鼠标在按钮上滚动。

```
const String TabButton::EventClicked( "Clicked" );
const String TabButton::EventDragged( "Dragged" );
const String TabButton::EventScrolled( "Scrolled" );
```

这个类有三个成员变量，两个状态变量，一个目标窗口的指针。

```
bool    d_selected;           //按钮是否被选中
bool    d_dragging;           //按钮是否正在被拖动
Window* d_targetWindow;       //这个Tab对应的目标窗口
```

Tab按钮也是重载了一些鼠标消息并适时的激发了一些消息。从setTargetWindow函数可以看出Tab按钮显示的文本是目标窗口的文本。

```
void TabButton::setTargetWindow(Window* wnd)
{
   d_targetWindow = wnd;
   setText(wnd->getText());
}
//鼠标按下，当中键按下的时候开始拖动，究竟如何决定拖动，读者可以自己设计
void TabButton::onMouseButtonDown(MouseEventArgs& e)
{
   if (e.button == MiddleButton)
   {
      captureInput ();
      e.handled = true;
      d_dragging = true;
      fireEvent(EventDragged, e, EventNamespace);
   }
      ButtonBase::onMouseButtonDown(e);
}
//鼠标抬起，左键抬起的时候激发Clicked事件，中键抬起时取消拖动
void TabButton::onMouseButtonUp(MouseEventArgs& e)
{
      if ((e.button == LeftButton) && isPushed())
      {
            Window* sheet = System::getSingleton().getGUISheet();
            if (sheet)
            {
                  if (this == sheet->getTargetChildAtPosition(e.position))
                  {
                        WindowEventArgs args(this);
                        onClicked(args);
                  }
            }
            e.handled = true;
   }
   else if (e.button == MiddleButton)
   {
      d_dragging = false;
      releaseInput ();
      e.handled = true;
   }
      ButtonBase::onMouseButtonUp(e);
```

```
}
//鼠标移动事件，如果中键处以按下状态则激发拖动事件
void TabButton::onMouseMove(MouseEventArgs& e)
{
   if (d_dragging)
   {
      fireEvent(EventDragged, e, EventNamespace);
      e.handled = true;
   }
      ButtonBase::onMouseMove(e);
}
//当鼠标在Tab按钮上滚动的时候，激发滚动消息
void TabButton::onMouseWheel(MouseEventArgs& e)
{
   fireEvent(EventScrolled, e, EventNamespace);
      ButtonBase::onMouseMove(e);
}
```

究竟什么时候激发什么事件都是由用户决定的，用户根据自己的逻辑自己决定。比如拖动可以设计为左键单击按下的时候，鼠标移动时激发。这里顺便介绍一个函数，这个函数每种控件都有，它用来决定这个控件的名称或者类型。当通过Window类的指针来测试一个控件的类型的时候非常有用，也可以看做是动态类型确定的函数。

```
virtual bool     testClassName_impl(const String& class_name) const
{
      if (class_name=="TabButton")      return true;
      return ButtonBase::testClassName_impl(class_name);
}
```

Tab控件和PushButton按钮其实是比较类似的。

7.1.3 单选控件

单选按钮大家应该是比较熟悉的了。RadioButton类对应单个的单选按钮。一个组内的所有单选按钮同一时刻只能选中一个。这个组内的按钮必须在一个父窗口内。这个类有两个属性，这是我们首次给控件加属性，我们介绍属性的添加方法。CEGUI中每个属性都是一个类，这个类必须派生自Property，而且必须的实现get和set函数。单选按钮提供了两个属性分别是单选按钮所在的组和单选按钮当前是否被选择。每个属性在整个CEGUI系统中只有一个实例，就是在对应控件内部定义的静态变量。

```
RadioButtonProperties::Selected   RadioButton::d_selectedProperty;
RadioButtonProperties::GroupID  RadioButton::d_groupIDProperty;
```

我们主要介绍Selected属性。Property构造函数的三个参数代表属性的名称，属性的帮助字符串和属性默认值。

```
class Selected : public Property
{
public:
      Selected() : Property(
            "Selected",
            "Property to get/set the selected state of the RadioButton.  Value is either \"True\" or
      \"False\".", "False")
      {}
      String      get(const PropertyReceiver* receiver) const;
      void set(PropertyReceiver* receiver, const String& value);
};
```

set和get函数都是虚函数必须每个属性必须实现，他们一般是将receiver转化为对应的控件指针，然后调用控件对应的函数来设置属性的值和获取属性的值。

```
//属性的名字控件
namespace RadioButtonProperties
{
    //获取是否被选中的属性，这个属性是个布尔值，转化成字符串返回
    String Selected::get(const PropertyReceiver* receiver) const
    {
        return PropertyHelper::boolToString(static_cast<const RadioButton*>(receiver)->
        isSelected());
    }
    //设置是否被选中，通过将receiver转化为RadioButton然后设置内部属性
    void Selected::set(PropertyReceiver* receiver, const String& value)
    {
     static_cast<RadioButton*>(receiver)->setSelected(PropertyHelper::stringToBool(value));
    }
}
```

窗口是派生自属性集的，它可以通过addProperty函数添加属性。

```
void RadioButton::addRadioButtonProperties(void)
{
    addProperty(&d_selectedProperty);
    addProperty(&d_groupIDProperty);
}
```

如果读者需要给控件添加属性可以按照上文介绍的步骤。下面介绍设置这两个属性具体值的函数。

```
//设置单选按钮选中状态
void RadioButton::setSelected(bool select)
{
    //如果设置的状态和我们目前的状态不一致
    if (select != d_selected)
    {
        d_selected = select;
        requestRedraw();
        //如果设置当前状态为选中状态，则反选所有其他同组的单选按钮
        if (d_selected)
        {
            deselectOtherButtonsInGroup();
        }
        //激发事件
        WindowEventArgs args(this);
        onSelectStateChanged(args);
    }
}
//设置组ID
void RadioButton::setGroupID(ulong group)
{
    d_groupID = group;
    //如果当前的是选中的状态则更新选中状态
    if (d_selected)
    {
        deselectOtherButtonsInGroup();
    }
}
```

反选所有同一组的按钮，当自己选中的时候。保证同一组只有一个被选中。

```
void RadioButton::deselectOtherButtonsInGroup(void) const
{
    // 如果我们没有父窗口则什么都不做
    if (d_parent)
    {
        size_t child_count = d_parent->getChildCount();
        // 扫描所有的子窗口
        for (size_t child = 0; child < child_count; ++child)
        {
            // 如果这个子窗口也是一个单选按钮
            if (d_parent->getChildAtIdx(child)->getType() == getType())
            {
                RadioButton* rb = (RadioButton*)d_parent->getChildAtIdx(child);
                // 如果这个窗口选中状态而且不是我们自己，而且和我们同组则反选它
                if (rb->isSelected() && (rb != this) && (rb->getGroupID() == d_groupID))
                {
                    rb->setSelected(false);
                }
            }
        }
    }
}
```

当设置自己为选中状态的时候显然需要反选所有其他同组的单选按钮，当自己的组ID改变的时候也需要反选其他的按钮，这时候主要是使自己不处于选中状态。下面要介绍的函数是getSelectedButtonInGroup，它返回当前这个组的被选中元素。

```
RadioButton* RadioButton::getSelectedButtonInGroup(void) const
{
    // 当我们有父窗口的时候才能操作，没有父窗口的单选按钮是没有功能的
    if (d_parent)
    {
        size_t child_count = d_parent->getChildCount();
        // 扫描所有的直接子窗口
        for (size_t child = 0; child < child_count; ++child)
        {
            //如果都是单选按钮
            if (d_parent->getChildAtIdx(child)->getType() == getType())
            {
                RadioButton* rb = (RadioButton*)d_parent->getChildAtIdx(child);
                // 是否是同一组的而且被选中，如果是则返回，结果可能是自己
                if (rb->isSelected() && (rb->getGroupID() == d_groupID))
                {
                    return rb;
                }
            }
        }
    }
    return 0;
}
```

这个函数用于获取当前选中的单选按钮。讲了这么多，读者一定希望知道单选按钮如何被选中，显然是当鼠标单击按钮的时候被选中了。CEGUI也的确是这么做的。

```
void RadioButton::onMouseButtonUp(MouseEventArgs& e)
{
```

```
    //如果是鼠标左键，而且当前鼠标处于按下的状态，则可能会被选中，继续判断
    if ((e.button == LeftButton) && isPushed())
    {
        Window* sheet = System::getSingleton().getGUISheet();
        if (sheet)
        {
            // 如果鼠标在单选按钮上空释放，则设置这个按钮被选中
            if (this == sheet->getTargetChildAtPosition(e.position))
            {
                // 调用选中函数，这个函数前面讲过，他会反选其他的窗口
                setSelected(true);
            }
        }
        e.handled = true;
    }
    ButtonBase::onMouseButtonUp(e);
}
```

单选按钮也是非常简单的，只要鼠标在按钮上抬起，其实也就是单击事件（Clicked），就设置这个按钮为选中状态。派生自ButtonBase的还有多选控件CheckBox，它与单选控件的区
别是它没有组的概念，只有内部的状态选中还是没有选中。下一小节介绍它。

7.1.4 多选控件

多选控件对应的类是CheckBox，这个类非常简单，只有一个成员变量标识窗口是否被选中，因为多选窗口不必关心其他的窗口是否被选中，它只关心自己是否被选中。这个类还提供了一个事件，当控件的选中状态发生变化的时候会激发一个事件。这两个变量定义如下。

```
bool        d_selected;
const String Checkbox::EventCheckStateChanged( "CheckStateChanged" );
```

同时这个类提供了一个属性，这个属性表示是否窗口被选中。属性对应多选按钮的数据成员就是d_selected。下面介绍这个类的主要的三个函数。

```
//设置选择状态
void Checkbox::setSelected(bool select)
{
    if (select != d_selected)
    {
        d_selected = select;
        requestRedraw();
        //激发事件
    WindowEventArgs args(this);
        onSelectStateChange(args);
    }
}
//这个函数主要就是激发按钮状态改变的事件
void Checkbox::onSelectStateChange(WindowEventArgs& e)
{
    fireEvent(EventCheckStateChanged, e, EventNamespace);
}
//操作入口函数和单选按钮的类似
void Checkbox::onMouseButtonUp(MouseEventArgs& e)
{
    if ((e.button == LeftButton) && isPushed())
```

```
    {
        Window* sheet = System::getSingleton().getGUISheet();
        if (sheet)
        {
            // 当鼠标单击这个控件的时候，设置为选中的反状态
            if (this == sheet->getTargetChildAtPosition(e.position))
            {
                // 切换选中状态
                setSelected(d_selected ^ true);
            }
        }
        e.handled = true;
    }
    ButtonBase::onMouseButtonUp(e);
}
```

多选控件就是这么简单，这里就不多做介绍了。第7.1节主要介绍了四种按钮，他们也是非常常用的按钮。他们的实现都比较简单，特别适合作为例子。通过这四个控件的学习读者应该对编写控件有了简单的了解。应该已经明白实现一个控件需要重载那些函数，需要做那些逻辑判断。具体的控件的实现完全是根据用户的要求来设计的，需要什么样的控件就想办法实现它。当然有些控件完全可以通过组合现有的控件来实现，这就到布局这一层了。布局这一层是比控件这一层更加高一级的。控件作为布局需要的各种基本元素。总之实现什么样的控件，如何实现它完全是根据用户的需要和控件作者的想法了。CEGUI提供了许多官方的控件，其中大部分都是常用控件。对于编写一个游戏来说是不够的，游戏有他特殊的要求，所以学习编写控件还是非常必要而且有用的，如果读者可以非常轻松的编写控件，相信读者可以非常容易的找到一份工资不菲的工资。毕竟我们学习这些知识的目的还是能更好的工资，能有更好的发展。这里说这些就是希望读者能自己多做练习，实践出真知，有些事情自己不亲自做是永远不会知道的。下一节我们介绍重量级的控件单行编辑框控件。

7.2 单行编辑框控件

编辑框控件不像按钮控件，它是非常复杂的控件。这里分为几个小节介绍CEGUI提供的EditBox控件。按照控件的功能可以分为鼠标消息处理，编辑框内容处理，键盘消息处理和编辑框的数据，事件和属性。首先介绍编辑框控件的属性事件和成员变量。

7.2.1 编辑框的实现数据

编辑框的实现数据包括成员变量，属性和事件。属性操作的数据其实就是成员变量，但有的成员变量不对应某个属性。下面是11个成员变量。

```
bool d_readOnly;                //如果编辑框是只读模式，则为真
bool d_maskText;                //是否显示的时候全部显示为掩码，例如密码框
utf32       d_maskCodePoint;    //当显示为掩码的时候使用的字符（d_maskText为真）
size_t      d_maxTextLen;       //编辑框的最大字符数
size_t      d_caratPos;         //当前输入光标的位置（在字符串中的位置）
size_t      d_selectionStart;   //文字被选中的起始位置
size_t      d_selectionEnd;     //文字被选中的结束位置
String      d_validationString; //正则表达式的字符串缓冲
RegexValidator*     d_validator;//字符串有效性的校验结构
bool d_dragging;                //当选择文字的时候标识开始选择文本
size_t      d_dragAnchorIdx;    //!< Selection index for drag selection anchor point.
```

CEGUI的输入框支持正则表达式，当用户设置了正则表达式的时候，可以调用函数isStringValid来判断字符串是否符合正则表达式的要求。CEGUI使用的正则表达式库是PCRE全称（Perl-Compatible Regular Expressions ）是剑桥大学编写的正则表达式库。读者如果不清楚正则表达式可以参阅一些书籍或者上网查询。RegexValidator简单的封装了prce的环境。它有一个prce的指针成员变量，并提供了release函数释放这个环境，定义如下。

```
struct RegexValidator
{
    RegexValidator(void) : d_regex(0) {}
    ~RegexValidator(void) { release(); }
    void release()
    {
        if (d_regex != 0)
        {
            //是否pcre环境
            pcre_free(d_regex);
            d_regex = 0;
        }
    }
    pcre* d_regex;
};
```

utf32被定义成整型，CEGUI中使用整型变量来表示一个字符，哪怕这个字符是一个简单的英文字符。它的String类的数据都是保存为整型的。大家现在只需要了解就可以了，我们将会在第8章详细介绍。

编辑框有11个事件，定义如下。

```
//只读模式发生变化的通知事件
const String Editbox::EventReadOnlyModeChanged( "ReadOnlyChanged" );
//是否显示为掩码的模式改变
const String Editbox::EventMaskedRenderingModeChanged( "MaskRenderChanged" );
//掩码显示的字符改变时激发的事件（比如密码框的掩码字符是'*'，现在改为'@'时）
const String Editbox::EventMaskCodePointChanged( "MaskCPChanged" );
//正则表达式的校验字符串发生改变的事件
const String Editbox::EventValidationStringChanged( "ValidatorChanged" );
//编辑框的最大字符串长度发生改变时的通知事件
const String Editbox::EventMaximumTextLengthChanged( "MaxTextLenChanged" );
//如果当前的字符串不符合正则表达式校验字符串的规则则激发事件
const String Editbox::EventTextInvalidated( "TextInvalidated" );
//当用户输入一个字符时会做正则表达式的检查，如果检查失败则激发这个事件
const String Editbox::EventInvalidEntryAttempted( "InvalidInputAttempt" );
//输入光标移动会激发的事件
const String Editbox::EventCaratMoved( "TextCaratMoved" );
//当前选中的字符串发生变化是激发的事件
const String Editbox::EventTextSelectionChanged( "TextSelectChanged" );
//当前编辑框的字符串等于最大字符串时激发的事件
const String Editbox::EventEditboxFull( "EditboxFull" );
//这个事件对应用户按回车或者Tab键的时候激发，表示输入框输入完成
const String Editbox::EventTextAccepted( "TextAccepted" );
```

CEGUI只定义了这些事件，如果读者有需要可以自己定义其他的事件，并在适合的时候激发它们。激发事件就是调用事件的所有处理函数，相信读者应该非常清楚了。

下面介绍最后的数据，编辑框的属性。

```
//只读模式属性，对应的数据成员d_readOnly
static EditboxProperties::ReadOnly                  d_readOnlyProperty;
//掩码模式属性，对应数据成员d_maskText
static EditboxProperties::MaskText                  d_maskTextProperty;
//掩码字符串属性，对应数据成员d_maskCodePoint
static EditboxProperties::MaskCodepoint             d_maskCodepointProperty;
```

```
//正则表达式的校验字符串，对应数据成员d_validationString
static EditboxProperties::ValidationString          d_validationStringProperty;
//输入光标对应的位置（其实就是文本串中的位置），对应成员变量d_caratPos
static EditboxProperties::CaratIndex                d_caratIndexProperty;
//选中文本的起始位置，对应成员变量d_selectionStart
static EditboxProperties::SelectionStart            d_selectionStartProperty;
//选中文本的起始长度，没有直接对应的成员变量
static EditboxProperties::SelectionLength           d_selectionLengthProperty;
//编辑框允许的最大文本长度，对应成员变量d_selectionStart
static EditboxProperties::MaxTextLength             d_maxTextLengthProperty;
```

属性的提供就是为了修改和获取控件内部的数据。SelectionStart虽然没有直接对应成员数据但它和两个成员（d_selectionStart和d_selectionEnd）有关。

7.2.2 操作控制

什么是操作控制呢？简单的说就是用户的操作比如键盘操作，鼠标操作对文本编辑的控制，就是用户发生这些控制指令来控制文本框的字符，字符的选择，删除，光标的控制等。我们把控制当作一层，把具体的逻辑处理算作另一层。这一小节介绍控制，下一小节介绍逻辑处理。

编辑框需要处理的键盘消息有字符消息，控制消息，方向键消息等。字符消息输入字符，方向键控制输入光标，后退键删除字符等。这些消息在onKeyDown中捕获。

```
void Editbox::onKeyDown(KeyEventArgs& e)
{
    // 基类处理
    Window::onKeyDown(e);
    if (hasInputFocus() && !isReadOnly())
    {
        WindowEventArgs args(this);
        switch (e.scancode)
        {
        //Shift键设置文本选择的起始光标位置（d_dragAnchorIdx变量保存）
        case Key::LeftShift:
        case Key::RightShift:
            if (getSelectionLength() == 0)
            {
                d_dragAnchorIdx = getCaratIndex();
            }
            break;
        //向前删除一个字符
        case Key::Backspace:
            handleBackspace();
            break;
        //向后删除一个字符
        case Key::Delete:
            handleDelete();
            break;
        case Key::Tab:
        case Key::Return:
        case Key::NumpadEnter:
            // 激发输入接收事件，读者可以修改键盘事件Tab和Enter以及NumpadEnter
            onTextAcceptedEvent(args);
            break;
        //移动光标到左边或者向左选择文本
```

```
            case Key::ArrowLeft:
                if (e.sysKeys & Control)
                {
                    handleWordLeft(e.sysKeys);
                }
                else
                {
                    handleCharLeft(e.sysKeys);
                }
                break;
            //移动光标到右边或者向右选择文本
            case Key::ArrowRight:
                if (e.sysKeys & Control)
                {
                    handleWordRight(e.sysKeys);
                }
                else
                {
                    handleCharRight(e.sysKeys);
                }
                break;
            //切换光标到最前面
            case Key::Home:
                handleHome(e.sysKeys);
                break;
            //切换光标到最后面
            case Key::End:
                handleEnd(e.sysKeys);
                break;
        default:
           return;
            }
            e.handled = true;
        }
}
```

onKeyDown函数主要是捕获各种控制字符的消息。这个函数里调用了许多handle函数，用来处理各种用户控制。这里只介绍一个，剩余的读者自己阅读。其实这些控制也是非常好理解的。我们以回退键为例介绍这些函数。这个函数的主要功能是向前删除一个字符。

```
void Editbox::handleBackspace(void)
{
    //非只读模式才处理
    if (!isReadOnly())
    {
        String tmp(d_text);
        //如果当前有选中的文本则删除之
        if (getSelectionLength() != 0)
        {
            tmp.erase(getSelectionStartIndex(), getSelectionLength());
            //删除完了在做正则表达式的检查
            if (isStringValid(tmp))
            {
```

```
                    //检查通过则取消选择并且设置新的文本作为文本框的文本
                    eraseSelectedText(false);
                    setText(tmp);
                }
                else
                {
                    // 正则表达式检查失败则激发事件
                    WindowEventArgs args(this);
                    onInvalidEntryAttempted(args);
                }
            }
            //如果没有选中文本，则向前删除一个字符
            else if (getCaratIndex() > 0)
            {
                //在当前光标的位置向前删除一个字符，然后还是做正则表达的检查
                tmp.erase(d_caratPos - 1, 1);
                if (isStringValid(tmp))
                {
                    setCaratIndex(d_caratPos - 1);
                    setText(tmp);
                }
                else
                {
                    WindowEventArgs args(this);
                    onInvalidEntryAttempted(args);
                }
            }
        }
    }
```

字符消息通过重载onCharacter函数来捕获。当游戏注入injectChar的时候会调用这个函数。

```
void Editbox::onCharacter(KeyEventArgs& e)
{
    Window::onCharacter(e);
    // 处理条件，我们是当前的激活的键盘焦点窗口，字体支持这个字符，我们可写
    if (hasInputFocus() && getFont()->isCodepointAvailable(e.codepoint) && !isReadOnly())
    {
        //备份当前的文本框文本
        String tmp(d_text);
        //删除当前选中的文本
        tmp.erase(getSelectionStartIndex(), getSelectionLength());
        //判断是否达到字符的最大值，如果不是才继续执行
        if (tmp.length() < d_maxTextLen)
        {
            //将新的字符追加的原有字符串的当前光标后面
            tmp.insert(getSelectionStartIndex(), 1, e.codepoint);
            //正则表达式检查，如果没有设置正则表达式则肯定会成功
            if (isStringValid(tmp))
            {
                // 这个函数传回false只取消文本选择，传回true才删除选中的文本
                eraseSelectedText(false);
        // 递增光标位置
```

```
            d_caratPos++;
            // 设置新的编辑框字符串
                    setText(tmp);
            //设置事件已经被处理的标志
            e.handled = true;
                }
                else
                {
                    // 激发无效更改的事件
                    WindowEventArgs args(this);
                    onInvalidEntryAttempted(args);
                }

            }
            else
            {
                // 激发文本已满的消息
                WindowEventArgs args(this);
                onEditboxFullEvent(args);
            }
        }
}
```

这些函数的设计完全是根据用户的处理习惯来做的。比如当有选中文本的时候后退键删除选中文本，否则删除当前光标的前一个字符。追加一个字符的时候如果有选中文本则使用追加的字符替换选中的文本，如果没有选中文本则在当前输入光标的位置向后添加一个字符。如果输入框没有键盘焦点，或者输入框只读，或者字符不可渲染（字体不支持）则不会将字符添加到输入框。所有这些规则都是在键盘和鼠标控制逻辑中实现的。读者可以根据不同的需要来自己修改。比如激发文本接受事件的时候可以不支持Tab键，也就是只有Enber键（包括小键盘的）可以激发文本接受事件。

鼠标的操作也会影响输入框，比如修改当前光标的位置，选中文本等。文本框重载以下几个鼠标消息虚函数。

```
void Editbox::onMouseButtonDown(MouseEventArgs& e)
{
    Window::onMouseButtonDown(e);
    if (e.button == LeftButton)
    {
        // 左键单击的时候设置捕获鼠标消息
        if (captureInput())
        {
            // 上边点下的时候需要取消当前的文本选择
            clearSelection();
            d_dragging = true;
            //通过鼠标位置获取选中的文本在文本框字符串中的位置，需要字体支持
            d_dragAnchorIdx = getTextIndexFromPosition(e.position);
            //设置当前的鼠标光标位置为鼠标点击的位置
            setCaratIndex(d_dragAnchorIdx);
        }
        e.handled = true;
    }
}
void Editbox::onMouseButtonUp(MouseEventArgs& e)
{
    Window::onMouseButtonUp(e);
    if (e.button == LeftButton)
```

```
    {
        //如果是鼠标左键则是否鼠标的捕获
        releaseInput();
        e.handled = true;
    }
}
//鼠标双击需要选中一些文本
void Editbox::onMouseDoubleClicked(MouseEventArgs& e)
{
    Window::onMouseDoubleClicked(e);
    if (e.button == LeftButton)
    {
        //如果显示的文字是掩码文字，则设置选中所有的文本
        if (isTextMasked())
        {
            d_dragAnchorIdx = 0;
            setCaratIndex(d_text.length());
        }
        //不显示掩码则选中一些文本，决定选中那些文本的函数是getWordStartIdx和
        //getNextWordStartIdx函数
        // not masked, so select the word that was double-clicked.
        else
        {
            d_dragAnchorIdx = TextUtils::getWordStartIdx(d_text, (d_caratPos ==
                d_text.length()) ? d_caratPos : d_caratPos + 1);
            d_caratPos              = TextUtils::getNextWordStartIdx(d_text, d_caratPos);
        }
        // 执行真正的选中操作
        setSelection(d_dragAnchorIdx, d_caratPos);
        e.handled = true;
    }
}
//鼠标三击函数处理，这个函数选中所有的当前编辑框中的文本
void Editbox::onMouseTripleClicked(MouseEventArgs& e)
{
    Window::onMouseTripleClicked(e);
    if (e.button == LeftButton)
    {
        d_dragAnchorIdx = 0;
        setCaratIndex(d_text.length());
        setSelection(d_dragAnchorIdx, d_caratPos);
        e.handled = true;
    }
}
//鼠标移动的消息，用来选中文本
void Editbox::onMouseMove(MouseEventArgs& e)
{
    Window::onMouseMove(e);
    //如果当前正在选择文本则，设置选择状态
    if (d_dragging)
    {
        setCaratIndex(getTextIndexFromPosition(e.position));
```

```
            setSelection(d_caratPos, d_dragAnchorIdx);
        }
        e.handled = true;
}
```

编辑框的控制函数就介绍这几个。这些控制逻辑是可以随用户的需要改变的。CEGUI的实现和Window的实现非常类似，这也是为了方便用户的操作。如果读者需要新的操作方法可以自己设计。设计的思路就是重载一些键盘和鼠标消息的虚函数，然后处理自己的逻辑。

7.2.3 数据控制

数据的控制和用户的操作无关，它是数据控制的逻辑。只设计如何处理文本数据。这部分使用了许多的String类的函数。但这个类我们还没有介绍可能读者有许多的还不能理解不过如果读者使用个std::string类则应该对CEGUI提供的String类就比较熟悉了。这个类的接口和std::string的接口有好的是一样的。

编辑框最重要的数据就是它保存的文本了。它使用父类提供的d_text成员变量来保存窗口的文本数据。它是String类型的，String类可以理解为保存的Unicode的字符串，不同的是它每个字符使用四个自己来表示，而不是两个字节。具体String类的信息这里不做介绍，读者可以参考第8章。本小节介绍的函数都是处理成员变量的函数，他们一般也比较简单，就是些成员变量的赋值和获取函数，但它们这些数据是控制渲染的数据，渲染窗口需要根据他们来渲染。setSelection函数设置当前的选中起始和结束位置。

```
void Editbox::setSelection(size_t start_pos, size_t end_pos)
{
    // 确保选中的开始位置有效
    if (start_pos > d_text.length())
    {
        start_pos = d_text.length();
    }
    // 确保选中结束位置有效
    if (end_pos > d_text.length())
    {
        end_pos = d_text.length();
    }
    // 确保选中的起始位置在结束位置的前面
    if (start_pos > end_pos)
    {
        size_t tmp = end_pos;
        end_pos = start_pos;
        start_pos = tmp;
    }
    // 修改选中的起始或结束位置
    if ((start_pos != d_selectionStart) || (end_pos != d_selectionEnd))
    {
        d_selectionStart = start_pos;
        d_selectionEnd       = end_pos;
        // 激发选择改变的消息
        WindowEventArgs args(this);
        onTextSelectionChanged(args);
    }
}
```

清除选中的函数。有设置选中的自然有清除选中的，处理它还有另一个函数eraseSelectedText它出入false的时候也是清除选中文本的选中状态。

```
void Editbox::clearSelection(void)
```

```
{
    if (getSelectionLength() != 0)
    {
        setSelection(0, 0);
    }
}
//传入false的时候不会修改文本
void Editbox::eraseSelectedText(bool modify_text)
{
    if (getSelectionLength() != 0)
    {
        // 设置光标的位置，并且清除选中状态
        setCaratIndex(getSelectionStartIndex());
        clearSelection();
        if (modify_text)
        {
            //删除选中文本，并且激发需要的事件
            d_text.erase(getSelectionStartIndex(), getSelectionLength());
            WindowEventArgs args(this);
            onTextChanged(args);
        }
    }
}
```

设置最大的编辑框字符数函数setMaxTextLength。

```
void Editbox::setMaxTextLength(size_t max_len)
{
    //只有数据不同的时候才需要修改
    if (d_maxTextLen != max_len)
    {
        //设置成员变量的值
        d_maxTextLen = max_len;
        //激发对应的事件
        WindowEventArgs args(this);
        onMaximumTextLengthChanged(args);
        // 裁剪文本，确保当前的文本小于最大文本
        if (d_text.length() > d_maxTextLen)
        {
            d_text.resize(d_maxTextLen);
            onTextChanged(args);
            // 检查是否符合正则表达式的需要
            if (!isTextValid())
            {
                onTextInvalidatedEvent(args);
            }
        }
    }
}
```

正则表达式的相关函数这里就不介绍了，它们是setValidationString（设置正则表达式的规则字符串，这时候会编译正则表达式，供以后的检查使用），isStringValid（检查一个字符串是否符合正则表达式的要求），isTextValid（检查编辑框中的文本是否符合正则表达式的要求，会调用isStringValid函数）。读者如果不清楚正则表达式可以参考其他书籍，不过一般来说也不

会给文本框设置正则表达式。但也不是没有用，比如如果文本框只需要数字不需要其他的文本就可以通过使用正则表达式来完成。

设置光标位置的函数setCaratIndex，光标的位置绝对字符插入的位置，选择文本的位置。

```
void Editbox::setCaratIndex(size_t carat_pos)
{
    //确保光标位置的有效性
    if (carat_pos > d_text.length())
    {
        carat_pos = d_text.length();
    }
    // 如果新位置和旧位置不同才执行操作
    if (d_caratPos != carat_pos)
    {
        d_caratPos = carat_pos;
        // 激发光标移动的事件
        WindowEventArgs args(this);
        onCaratMoved(args);
    }
}
```

最后介绍一个函数根据当前鼠标的位置获取光标的位置的函数。

```
size_t Editbox::getTextIndexFromPosition(const Point& pt) const
{
  if (d_windowRenderer != 0)
  {
        //渲染窗口计算这个值，其实最终还是通过字体来算出的
    EditboxWindowRenderer* wr = (EditboxWindowRenderer*)d_windowRenderer;
    return wr->getTextIndexFromPosition(pt);
  }
  else
  {
    //抛出异常
  }
}
```

EditboxWindowRenderer类派生自WindowRenderer（渲染窗口的基类）。这里申明了这个接口目的就是调用getTextIndexFromPosition函数。而这个函数其实最终还是调用字体的对应函数来实现的。哪么为什么要这么设计呢，直接在这里调用字体的函数不也可以吗？我想这主要是满足渲染和逻辑分开的要求。其实在控件逻辑类里实现也是完全可以的。

```
class CEGUIEXPORT EditboxWindowRenderer : public WindowRenderer
{
public:
  EditboxWindowRenderer(const String& name);
  virtual size_t getTextIndexFromPosition(const Point& pt) const = 0;
};
```

这个类的声明非常简单，只声明了虚接口函数。到这里文本框的逻辑控件就介绍完了，这里并没有涉及到如何渲染它。控件的渲染在这一版的CEGUI中通过渲染窗口来实现的。渲染窗口的介绍在第10章。

7.3 框架控件

框架控件主要功能有拖动，调整大小，关闭控件等。它一般来说是作为其他控件的容器（作为其他控件的父窗口）出现。它也是非常重要的一种控件，内部包含一个标题栏控件和关闭按钮控件。当拖动标题栏的时候会拖动整个框架窗口，当关闭按钮被单击的时候会关闭这个控件（一般是隐藏起来而不是摧毁这个控件）。

7.3.1 控件控件的数据成员

框架控件也包含三部分的数据，属性数据，事件数据和基本的成员数据。

```
//控件是否激活状态，用于决定是否调整大小
bool d_frameEnabled;
//查了词典roll-up是自制雪茄烟的意思，在渲染窗口中发现d_rolledup用来控制是否渲染
//而 d_rollupEnabled是前者的控制变量，是否允许修改前者
bool d_rollupEnabled;
bool d_rolledup;
//拖动修改大小的数据
bool d_sizingEnabled;   //是否允许修改窗口的大小
bool d_beingSized;      //当前是否正在修改窗口的大小
float d_borderSize;     //修改大小的检测区域大小
Point      d_dragPoint; //开始修改区域时的鼠标位置
//拖动修改窗口大小的时候使用的四种鼠标图像，这四种图像读者应该明白是什么
const Image*   d_nsSizingCursor;
const Image*   d_ewSizingCursor;
const Image*   d_nwseSizingCursor;
const Image*   d_neswSizingCursor;
//是否允许拖动标题栏的时候拖动整个框架窗口
bool d_dragMovable;
```

属性的定义，在CEGUI系统中一个属性只有一个实例。

```
//是否允许调整框架的大小
static FrameWindowProperties::SizingEnabled          d_sizingEnabledProperty;
//框架控件是否可操作
static FrameWindowProperties::FrameEnabled           d_frameEnabledProperty;
//框架控件的标题栏是否可操作
static FrameWindowProperties::TitlebarEnabled  d_titlebarEnabledProperty;
//关闭按钮是否可操作
static FrameWindowProperties::CloseButtonEnabled d_closeButtonEnabledProperty;
//卷起状态
static FrameWindowProperties::RollUpState       d_rollUpStateProperty;
//卷起状态的控制变量
static FrameWindowProperties::RollUpEnabled          d_rollUpEnabledProperty;
//是否允许拖动大小
static FrameWindowProperties::DragMovingEnabled  d_dragMovingEnabledProperty;
//拖动的检测半径
static FrameWindowProperties::SizingBorderThickness d_sizingBorderThicknessProperty;
//上下拖动时的鼠标图像
static FrameWindowProperties::NSSizingCursorImage   d_nsSizingCursorProperty;
//左右拖动时的鼠标图像
static FrameWindowProperties::EWSizingCursorImage   d_ewSizingCursorProperty;
//右下（左上）拖动时的鼠标图像（\）
static FrameWindowProperties::NWSESizingCursorImage d_nwseSizingCursorProperty;
//左下（右上）拖动时的鼠标图像（/）
static FrameWindowProperties::NESWSizingCursorImage d_neswSizingCursorProperty;
```

属性总是对应一些成员变量或者对应成员变量的操作值（比如两个成员变量的差）。

框架框架有两个事件，如下所示。

```
//rollup的状态切换事件
static const String EventRollupToggled;
//关闭按钮被单击的事件，应该关闭窗口啦
static const String EventCloseClicked;
```

框架的数据成员就介绍到这里，下一小节介绍框架控件的外观。

7.3.2 窗口框架的外观

读者还记得第6章介绍的外观定义吗？外观文件中可以定义自动子窗口，框架窗口的标题栏和关闭按钮就是通过在外观定义中定义自动子窗口来实现的。框架控件的外观几乎使用了所有的外观定义的重要元素。它可以作为控件外观的典型代表，前两节没有介绍对应控件的外观，是因为他们的外观比较简单。这里介绍外观是希望读者能够明白如何去定义一个控件的外观。首先我们看外观定义中属性相关元素的定义。

```
<PropertyDefinition name="ClientAreaColour" initialValue="FF141B38"
redrawOnWrite="true" />
<PropertyLinkDefinition name="TitlebarFont" widget="__auto_titlebar__"
targetProperty="Font" />
<Property name="NSSizingCursorImage" value="set:TaharezLook image:MouseNoSoCursor" />
```

外观中定义了属性ClientAreaColour，表示客户区的颜色。这个属性可以想其他的属性一样通过getProperty和setProperty函数来获取和设置。他们可以和在C++中定义的属性一样操作。只不过他们是在外观文件中定义的而已。属性链接，这里框架窗口定义了TitlebarFont这个属性，它链接到框架标题窗口的Font属性。就是说在框架窗口中使用TitlebarFont属性的时候它的获取数据和修改数据其实是对标题窗口的Font属性来操作的。属性的初始化，NSSizingCursorImage表示上下拖动时的鼠标图像，这里指定了默认值。

标题栏窗口的定义，使用自动子窗口来实现。

```
<Child type="TaharezLook/Titlebar" nameSuffix="__auto_titlebar__">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><FontDim type="LineSpacing" padding="8" /></Dim>
  </Area>
  <Property name="AlwaysOnTop" value="False" />
</Child>
```

窗口类型TaharezLook/Titlebar的映射如下。从中可以知道标题栏的实现窗口类是Titlebar类。这个类我们就不介绍了，它主要就是支持拖动。

```
<FalagardMapping WindowType="TaharezLook/Titlebar"    TargetType="CEGUI/Titlebar"    Renderer="Falagard/Titlebar"
LookNFeel="TaharezLook/Titlebar" />
```

后缀名，"__auto_titlebar__"可以用来定位这个子窗口，前面的属性链接就使用这个名称来定义标题栏窗口。区域定义和属性初始值的指定读者应该已经明白了吧。框架窗口的另一
个自动子窗口关闭按钮，它的区域计算比较复杂，也比较有代表性，这里详细介绍。

```
<Area>
  <Dim type="LeftEdge" >
    <UnifiedDim scale="1" type="LeftEdge">
      <DimOperator op="Subtract">
        <ImageDim imageset="TaharezLook" image="SysAreaRight"
                                dimension="Width">
          <DimOperator op="Add">
            <ImageDim imageset="TaharezLook" image="SysAreaMiddle"
                            dimension="Width" />
          </DimOperator>
        </ImageDim>
```

```
        </DimOperator>
      </UnifiedDim>
    </Dim>
    <Dim type="TopEdge" >
      <AbsoluteDim value="0.5">
        <DimOperator op="Multiply">
          <WidgetDim widget="__auto_titlebar__" dimension="Height">
            <DimOperator op="Subtract">
              <ImageDim imageset="TaharezLook"
                                image="NewCloseButtonNormal" dimension="Width" />
            </DimOperator>
          </WidgetDim>
        </DimOperator>
      </AbsoluteDim>
    </Dim>
    <Dim type="Width" ><ImageDim imageset="TaharezLook"
image="NewCloseButtonNormal" dimension="Width" /></Dim>
    <Dim type="Height" ><ImageDim imageset="TaharezLook"       image="NewCloseButtonNormal" dimension="Width" /></Dim>
</Area>
```

关闭按钮在框架窗口标题栏的右边，所以它的位置比较难确定，大小就是关闭按钮图像的大小（高和宽）。左边（X轴）的位置使用（TotalWidth-（FrameRightWidth+FrameMiddleWidth））这个公式来计算。其中TotalWidth是整个框架的宽度，FrameRightWidth是框架使用的右上的图像的宽度（共分9个部分，前文讲过），FrameMiddleWidth是框架中上的图像宽度。上边（Y轴）的位置使用（0.5*（TitleHeight - CloseButtonHeight））来计算。其中TitleHeight 表示标题栏的高度，CloseButtonHeight表示关闭按钮的图像高度。这样算出的结果会产生关闭按钮在标题栏居中显示的效果。

除了上面介绍的部分还有几个命名区域的定义，各个StateImagery和ImagerySection的定义，这里就不做介绍了，读者可以自己分析。窗口的外观定义功能非常强大，读者只有理解了它的各个元素的功能和窗口坐标系统的关系才能够灵活的使用。

7.3.3 窗口框架的逻辑

我们说过框架窗口的标题栏和关闭按钮都是定义在外观文件中的，哪么在程序中如何获取他们呢？当然是通过名字了，它的名字是由父窗口的名称加上自己的名字后缀构成的。

```
Titlebar* FrameWindow::getTitlebar() const
{
  return static_cast<Titlebar*>(WindowManager::getSingleton().getWindow(
                        getName() + TitlebarNameSuffix));
}
```

这个函数获取标题栏窗口，同样还有一个函数用来获取关闭按钮窗口指针。

标题窗口的初始化，获取标题栏和关闭按钮做相应的处理。

```
void FrameWindow::initialiseComponents(void)
{
  // 获取两个子窗口的指针
  Titlebar* titlebar = getTitlebar();
  PushButton* closeButton = getCloseButton();
  // 配置标题栏子窗口
  titlebar->setDraggingEnabled(d_dragMovable);
  titlebar->setText(d_text);
  // 绑定关闭按钮的单击事件
  closeButton->subscribeEvent(PushButton::EventClicked,          Event::Subscriber(&CEGUI::FrameWindow::closeClickHandler,
this));
    //重新布局子窗口，布局其实就是设置子窗口的位置和大小
  performChildWindowLayout();
```

```
}
```

框架控件的一个重要功能就是支持调整大小。首先需要捕获鼠标的移动消息，然后决定是否到了拖动的范围内，最后决定拖动的方向，调整框架窗口的大小。

```
void FrameWindow::onMouseMove(MouseEventArgs& e)
{
    Window::onMouseMove(e);
    // 如果我们鼠标不在窗口的上方则什么也不做
    if (System::getSingleton().getWindowContainingMouse() != this)
    {
        return;
    }
    if (isSizingEnabled())
    {
        //通过鼠标的屏幕坐标获得它在窗口上的局部坐标
        Point localMousePos(CoordConverter::screenToWindow(*this, e.position));
        if (d_beingSized)
        {
            SizingLocation dragEdge = getSizingBorderAtPoint(d_dragPoint);
            //计算鼠标的移动偏移量
            float deltaX = localMousePos.d_x - d_dragPoint.d_x;
            float deltaY = localMousePos.d_y - d_dragPoint.d_y;
            //左边或者右边拖动
            if (isLeftSizingLocation(dragEdge))
            {
                moveLeftEdge(deltaX);
            }
            else if (isRightSizingLocation(dragEdge))
            {
                moveRightEdge(deltaX);
            }
            //上面或者下面拖动
            if (isTopSizingLocation(dragEdge))
            {
                moveTopEdge(deltaY);
            }
            else if (isBottomSizingLocation(dragEdge))
            {
                moveBottomEdge(deltaY);
            }
        }
        //如果还没开始拖动则先决定拖动方向和开始拖动
        else
        {
            setCursorForPoint(localMousePos);
        }
    }
    e.handled = true;
}
```

setCursorForPoint函数根据鼠标的位置决定鼠标的图像。

```
void FrameWindow::setCursorForPoint(const Point& pt) const
{
```

```
    //判断当前的鼠标拖动类型，根据不同的类型设置相应的鼠标图像
    switch(getSizingBorderAtPoint(pt))
    {
    case SizingTop:
    case SizingBottom:
        MouseCursor::getSingleton().setImage(d_nsSizingCursor);
        break;
    case SizingLeft:
    case SizingRight:
        MouseCursor::getSingleton().setImage(d_ewSizingCursor);
        break;
    case SizingTopLeft:
    case SizingBottomRight:
        MouseCursor::getSingleton().setImage(d_nwseSizingCursor);
        break;
    case SizingTopRight:
    case SizingBottomLeft:
        MouseCursor::getSingleton().setImage(d_neswSizingCursor);
        break;
    default:
        //默认显示正常光标
        MouseCursor::getSingleton().setImage(getMouseCursor());
        break;
    }
}
```

getSizingBorderAtPoint根据鼠标的位置决定拖动的类型。

```
FrameWindow::SizingLocation FrameWindow::getSizingBorderAtPoint(const Point& pt) const
{
    Rect frame(getSizingRect());
    // 框架可操作，允许拖动操作时才判断
    if (isSizingEnabled() && isFrameEnabled())
    {
        //判断鼠标是否在窗口内部
        if (frame.isPointInRect(pt))
        {
            //调整框架大小，缩小整个框架矩形，每边缩小d_borderSize
            frame.d_left    += d_borderSize;
            frame.d_top     += d_borderSize;
            frame.d_right  -= d_borderSize;
            frame.d_bottom  -= d_borderSize;
            // 检查每个方向是否在这个矩形的外面（为什么呢，读者思考）
            bool top   = (pt.d_y < frame.d_top);
            bool bottom = (pt.d_y >= frame.d_bottom);
            bool left   = (pt.d_x < frame.d_left);
            bool right= (pt.d_x >= frame.d_right);
            // 根据鼠标的位置返回对应的类型
            if (top && left)
            {
                return SizingTopLeft;
            }
            else if (top && right)
            {
```

```
                    return SizingTopRight;
                }
                else if (bottom && left)
                {
                    return SizingBottomLeft;
                }
                else if (bottom && right)
                {
                    return SizingBottomRight;
                }
                else if (top)
                {
                    return SizingTop;
                }
                else if (bottom)
                {
                    return SizingBottom;
                }
                else if (left)
                {
                    return SizingLeft;
                }
                else if (right)
                {
                    return SizingRight;
                }
            }
        }
        return SizingNone;
}
```

现在可以总结改变大小的过程了。首先是鼠标移动时检查是否处在拖动状态，如果是则根据拖动类型执行拖动操作，如果不是则根据鼠标的位置决定拖动类型，为下一次拖动做准备。两个辅助函数setCursorForPoint设置鼠标图像，调用getSizingBorderAtPoint函数获取鼠标的拖动类型。哪么拖动操作究竟如何实现呢？我们这里介绍其中之一，左边拖动函数moveLeftEdge，其他函数类似。

```
void FrameWindow::moveLeftEdge(float delta)
{
   float orgWidth = d_pixelSize.d_width;
   URect area(d_area);
      //获取框架限制的最大和最小窗口，拖动时不应该超出他们的限制
   float maxWidth(d_maxSize.d_x.asAbsolute(System::getSingleton().getRenderer()->
getWidth()));
   float minWidth(d_minSize.d_x.asAbsolute(System::getSingleton().getRenderer()->
getWidth()));
      //计算拖动后新的宽度，并修正尺寸的增加因子delta
   float newWidth = orgWidth - delta;
      //如果新窗口的尺寸大于最大尺寸则delta变为最大尺寸和目前尺寸的差（最终结果就是
//最大尺寸），对于最小尺寸也类似
   if (newWidth > maxWidth)
      delta = orgWidth - maxWidth;
   else if (newWidth < minWidth)
      delta = orgWidth - minWidth;
```

```
    float adjustment = PixelAligned(delta);
        //根据不同的对齐方式，设置框架窗口的区域
    if (d_horzAlign == HA_RIGHT)
    {
        area.d_max.d_x.d_offset -= adjustment;
    }
    else if (d_horzAlign == HA_CENTRE)
    {
        area.d_max.d_x.d_offset -= adjustment * 0.5f;
        area.d_min.d_x.d_offset += adjustment * 0.5f;
    }
    else
    {
        area.d_min.d_x.d_offset += adjustment;
    }
        //最后设置框架窗口的区域属性
    setArea_impl(area.d_min, area.getSize(), d_horzAlign == HA_LEFT);
}
```

框架窗口的拖动位置实现在标题栏中。有兴趣的读者可以自己阅读。这一小节就介绍到这里。下一小节介绍滚动条控件。

7.4滚动条控件

要理解滚动条的实现必须理解几个概念。什么是文档，什么是页。文档就是所有需要被显示的内容，页就是当前被显示的内容。当文档的内容大于页的大小时需要出现滚动条来辅助显示。滚动条有个位置变量保存开始显示的位置，显示的范围就是从这个位置开始，显示一页的大小的内容。滚动条有一个向上增加位置的按钮，一个向下增加位置的按钮和一个中间表示滚动位置的按钮。滚动条一般还是作为其他的窗口的子窗口出现的，它的作用就是辅助其他窗口的显示，比如一个多行显示文本的窗口就需要一个滚动条子窗口。

7.4.1 滚动条的数据成员

有了上面的介绍，相信下面几个数据成员就很好理解了。

```
float d_documentSize;          //文档的大小
float d_pageSize;              //页的大小
float d_stepSize;              //增加和减少的步进
float d_overlapSize;                //调页时的步进
float d_position;              //当前滚动条的位置
```

滚动条有三个子窗口的名称，他们都是在外观文件中定义的。有三个函数来获取他们的指针，如何获取一个窗口的指针呢？读者应该知道了吧。

```
const String Scrollbar::ThumbNameSuffix( "__auto_thumb__" );
const String Scrollbar::IncreaseButtonNameSuffix( "__auto_incbtn__" );
const String Scrollbar::DecreaseButtonNameSuffix( "__auto_decbtn__" );
```

滚动条激发的事件有以下四个。

```
//滚动条的位置改变了
const String Scrollbar::EventScrollPositionChanged( "ScrollPosChanged" );
//当滚动条中间的按钮被单击的时候激发
const String Scrollbar::EventThumbTrackStarted( "ThumbTrackStarted" );
//当滚动条中间的按钮被是否时激发，这是用户的拖动结束了
const String Scrollbar::EventThumbTrackEnded( "ThumbTrackEnded" );
//滚动条的配置信息改变了，滚动条的数据成员发生改变的时候会激发
const String Scrollbar::EventScrollConfigChanged( "ScrollConfigChanged" );
```

滚动条有以下几个属性，对应滚动条的数据成员。

```
//滚动条的文档尺寸
```

```
static ScrollbarProperties::DocumentSize     d_documentSizeProperty;
//滚动条的页尺寸
static ScrollbarProperties::PageSize         d_pageSizeProperty;
//滚动条的步进
static ScrollbarProperties::StepSize         d_stepSizeProperty;
//滚动条的页进
static ScrollbarProperties::OverlapSize      d_overlapSizeProperty;
//滚动条的位置信息
static ScrollbarProperties::ScrollPosition   d_scrollPositionProperty;
```

滚动条的数据成员就介绍到这里，它的外观定义就不做介绍了，读者可以自己阅读，希望读者可以读懂。下一小节介绍滚动条的Thumb子窗口。

7.4.2 滚动条的子窗口

滚动条有三个子窗口，其中两个对应类PushButton，他们是向上和向下的按钮，他们使用渲染窗口是Falagard/Button，外观定义是TaharezLook/ImageButton。另一个是Thumb窗口，就是中间的可以拖动的子窗口。

```
<FalagardMapping WindowType="TaharezLook/ImageButton" TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
LookNFeel="TaharezLook/ImageButton" />
```

向上和向下按钮使用的窗口类型是TaharezLook/ImageButton，而在模式文件中我们还发现了类似下面的定义。

```
<FalagardMapping WindowType="TaharezLook/Button"  TargetType="CEGUI/PushButton"
Renderer="Falagard/Button"      LookNFeel="TaharezLook/Button" />
<FalagardMapping WindowType="TaharezLook/SystemButton" argetType="CEGUI/PushButton"  Renderer="Falagard/SystemButton"
LookNFeel="TaharezLook/Button" />
```

他们只是外观定义或者渲染窗口不同，其他全部相同。从这里或许可以看出CEGUI为什么把窗口的渲染和逻辑分开的原因。分开后可以通过组合生成许多新的功能的控件。如果需要的逻辑是相同的只是显示的内容不同则不需要生成多个控件，只需要生成多个外观的定义或者多个渲染窗口就可以了。

这节的重点是Thumb类，这个类激发一些事件。滚动条父窗口会捕获这些消息做相应的处理。这些消息如下所示。

```
//Thumb的位置发生改变
const String Thumb::EventThumbPositionChanged( "ThumbPosChanged" );
//Thumb被单击，开始追踪位置的改变
const String Thumb::EventThumbTrackStarted( "ThumbTrackStarted" );
//Thumb单击后鼠标抬起，追踪位置结束
const String Thumb::EventThumbTrackEnded( "ThumbTrackEnded" );
```

追踪用户的鼠标事件，方法相信大家应该掌握了，就是重载Window类的鼠标虚函数接口来实现自己的逻辑。

```
void Thumb::onMouseButtonDown(MouseEventArgs& e)
{
    PushButton::onMouseButtonDown(e);
    if (e.button == LeftButton)
    {
        //开始拖动了
        d_beingDragged = true;
        d_dragPoint = CoordConverter::screenToWindow(*this, e.position);
        // 激发开始追踪事件
        WindowEventArgs args(this);
        onThumbTrackStarted(args);
        e.handled = true;
    }
}
```

接下来，鼠标移动的时候激发Thumb位置的改变。

```
void Thumb::onMouseMove(MouseEventArgs& e)
```

```
{
    PushButton::onMouseMove(e);
    //只有单击了Thumb才可能发生位置的拖动改变
    if (d_beingDragged)
    {
    Size parentSize(getParentPixelSize());
        Vector2 delta;
        float hmin, hmax, vmin, vmax;
    delta = CoordConverter::screenToWindow(*this, e.position);
    hmin = d_horzMin;
    hmax = d_horzMax;
    vmin = d_vertMin;
    vmax = d_vertMax;
        //计算鼠标移动的偏移量
        delta -= d_dragPoint;
    delta.d_x /= parentSize.d_width;
    delta.d_y /= parentSize.d_height;
        // Calculate new (pixel) position for thumb
        UVector2 newPos(getPosition());
        if (d_horzFree)
        {
            newPos.d_x.d_scale += delta.d_x;
            // 限制新位置在允许的最大和最小之间（含边界，水平方向）
            newPos.d_x.d_scale = (newPos.d_x.d_scale < hmin) ? hmin :
            (newPos.d_x.d_scale > hmax) ? hmax : newPos.d_x.d_scale;
        }
        if (d_vertFree)
        {
            newPos.d_y.d_scale += delta.d_y;
            // 限制新位置在允许的最大和最小之间（含边界，竖直方向）
            newPos.d_y.d_scale = (newPos.d_y.d_scale < vmin) ?
            vmin : (newPos.d_y.d_scale > vmax) ? vmax : newPos.d_y.d_scale;
        }
        //如果需要更新Thumb的位置
        if (newPos != getPosition())
        {
            setPosition(newPos);
            // 如果是跟踪位置才激发位置的改变
            if (d_hotTrack)
            {
                WindowEventArgs args(this);
                onThumbPositionChanged(args);
            }
        }
    }
    e.handled = true;
}
```

自然还有重载鼠标抬起的函数，用来激发追踪结束的事件。不过这里没有重载onMouseButtonUp函数，而是重载了丢失焦点的函数onCaptureLost，这个函数当用户是否了鼠标消息捕获后会被调用，而是否鼠标消息捕获（releaseInput）会在onMouseButtonUp函数中调用，所以也可以认为是鼠标抬起会调用的事件。

```
void Thumb::onCaptureLost(WindowEventArgs& e)
```

```
{
    PushButton::onCaptureLost(e);
    d_beingDragged = false;
    //激发追踪结束和位置改变的消息
    WindowEventArgs args(this);
    onThumbTrackEnded(args);
    onThumbPositionChanged(args);
}
```

有了这三步就可以实现拖动Thumb发生三个事件了，而滚动条或者其他使用这个控件的父控件就可以捕获这三个消息，并做相应的处理了。这个类的其他函数就不多做介绍了。下一小节介绍滚动条的各种处理函数。

7.4.3 滚动条的逻辑处理

首先看滚动条的初始化函数，他会获取子窗口并注册关心的事件。

```
void Scrollbar::initialiseComponents(void)
{
    // 获取Thumb子窗口并且注册关心的事件，这三个事件比较熟悉吧
    Thumb* thumb = getThumb();
    thumb->subscribeEvent(Thumb::EventThumbPositionChanged,
Event::Subscriber(&CEGUI::Scrollbar::handleThumbMoved, this));
    thumb->subscribeEvent(Thumb::EventThumbTrackStarted,
Event::Subscriber(&CEGUI::Scrollbar::handleThumbTrackStarted, this));
    thumb->subscribeEvent(Thumb::EventThumbTrackEnded,      Event::Subscriber(&CEGUI::Scrollbar::handleThumbTrackEnded,
this));
    //获取向下子窗口并且注册关心的事件，注册鼠标单击的事件
    PushButton* increase = getIncreaseButton();
    increase->subscribeEvent(PushButton::EventMouseButtonDown,
    Event::Subscriber(&CEGUI::Scrollbar::handleIncreaseClicked, this));
    // 获取向上的子窗口并且注册关心的事件，注册鼠标单击的事件
    PushButton* decrease = getDecreaseButton();
    decrease->subscribeEvent(PushButton::EventMouseButtonDown,
    Event::Subscriber(&CEGUI::Scrollbar::handleDecreaseClicked, this));
    //初始化布局
    performChildWindowLayout();
}
```

滚动条的最重要的函数当然是设置滚动条位置的函数了。

```
void Scrollbar::setScrollPosition(float position)
{
    float old_pos = d_position;
    // 最大位置是(docSize - pageSize), 但是必须大于0(这种情况下文档大小非常小)
    float max_pos = ceguimax((d_documentSize - d_pageSize), 0.0f);
    // 限制位置到有效的位置:  0 <= position <= max_pos
    d_position = (position >= 0) ? ((position <= max_pos) ? position : max_pos) : 0.0f;
    //更新Thumb的位置
    updateThumb();
    //如果位置改变了，则激发相应的消息
    if (d_position != old_pos)
    {
        WindowEventArgs args(this);
        onScrollPositionChanged(args);
    }
}
```

下面介绍向下按钮被单击的处理事件。它只是简单的调整滚动条的位置而已。

```
bool Scrollbar::handleDecreaseClicked(const EventArgs& e)
{
    if (((const MouseEventArgs&)e).button == LeftButton)
    {
        // 调整滚动条的位置
        setScrollPosition(d_position - d_stepSize);
        return true;
    }
    else
    {
        return false;
    }
}
```

CEGUI中函数起名还是有规律可寻的，比如以on开头的都是事件处理或者激发的函数。以handle开头的都是处理CEGUI事件（窗口激发的各种事件）的函数。

下面两个函数是鼠标按钮按下和鼠标移动时的消息。

```
//这个函数主要处理用户单击滚动条的空白部分需要向上或向下滚动
void Scrollbar::onMouseButtonDown(MouseEventArgs& e)
{
    Window::onMouseButtonDown(e);
    if (e.button == LeftButton)
    {
        //这个函数获取滚动的方向，实现在滚动条的渲染窗口中
        float adj = getAdjustDirectionFromPoint(e.position);
        if (adj != 0)
        {
            setScrollPosition(d_position + ((d_pageSize - d_overlapSize) * adj));
        }
        e.handled = true;
    }
}
//滚动条吗，当鼠标滚轮滚动的时候显然要改变滚动条的位置
void Scrollbar::onMouseWheel(MouseEventArgs& e)
{
    Window::onMouseWheel(e);
    // 滚动的位置计算公式是wheelChange * stepSize，滚动的量乘以步进
    setScrollPosition(d_position + d_stepSize * -e.wheelChange);
    //确保父窗口不会有机会再处理这个消息了
    e.handled = true;
}
```

下面这个函数是从滚动条的渲染窗口中找到的，它实现了当鼠标单击空白处的时候返回滚动量。

```
 float FalagardScrollbar::getAdjustDirectionFromPoint(const Point& pt) const
  {
    Scrollbar* w = (Scrollbar*)d_window;
    Rect absrect(w->getThumb()->getUnclippedPixelRect());
        //如果在Thumb的下面或者右边则返回1
    if ((d_vertical && (pt.d_y > absrect.d_bottom)) ||
       (!d_vertical && (pt.d_x > absrect.d_right)))
    {
       return 1;
```

```
    }
        //如果在Thumb的上边或者左边则返回-1
    else if ((d_vertical && (pt.d_y < absrect.d_top)) ||
        (!d_vertical && (pt.d_x < absrect.d_left)))
    {
        return -1;
    }
        //否则返回0，不改变滚动条的位置
    else
    {
        return 0;
    }
}
```

其实滚动条最终要的就是要知道文档大小和页大小之间的关系，以及步进。如何调整滚动条的位置（操作方法，这里的实现和各种操作系统的UI操作是一样的）。滚动条是分水平滚动条和竖直滚动条的。一般来说不肯一个滚动条即使水平滚动条又是竖直滚动条。但一个控件（比如多行的文本框控件）上是可以既出现水平又出现竖直滚动条的。那样就需要两个滚动条控件了。一般来说滚动条控件都是配合其他控件来使用的。它很少单独使用，其他控件可能会根据文档的大小和页的大小来决定器子窗口的位置信息等。由于篇幅原因不可能在介绍其他的控件了比如树形控件，ListBox控件，多行文本控件，提示信息控件（Tooltips）等。没有介绍这些控件并不是说他们不重要，相反他们也非常的重要。读者如果有时间笔者建议还是经常阅读他们。这样对可以很快的提高读者的设计能力和分析问题解决问题的能力。

7.5 本章小结

这一章主要介绍了CEGUI官方提供的各种控件的逻辑部分，具体的实现比较复杂。而且由于控件的数量繁多不可能一一讲述，所以我们挑选了几个比较有代表性的控件作为例子。通过他们的学习希望读者可以知道如何使用CEGUI提供的各种接口设计一个控件。本章提供一个习题，希望读者可以自己完成一个控件。可以根据自己的需要自行设计一个
控件，如果不知道设计什么控件，那就设计一个定时器控件吧。如果做的出来则很好，做不出了也不要气馁，因为在第12章，笔者将实现一个定时器控件，读者可以比较一下你的实现和笔者的实现，看看有什么不同。实现方法是多种多样的但需要结果是相同的。

第8章 字体

字体在显示文字的时候非常重要。文本是如何显示的呢？在CEGUI里文本都是一个个的小图像，文字的显示就是这些图像拼成的。文本的显示的格式也是字体控制的，字体还计算文本的渲染长度，宽度等。同一种字体渲染同样的文本的渲染尺寸是一致的。字体还需要支持通过鼠标位置获取文本的位置。这个特性被用在鼠标点击后设置光标的位置（比如编辑框控件）。这里说的字体指的是CEGUI的字体类，而不是真正的字体文件。CEGUI中有两种字体一种是点阵型的字体，它的大小固定不可以变化（指的是设计时多大就多大，不肯能动态的改变）。另一种是TrueType 类型的字体，对应Windows系统提供的TTF字体文件。这种字体是矢量字体，也就是说这种字体里面保存了每个字的生成算法，当需要多大的字时，就计算出多大的字的点阵。总之字体的文本的渲染和计算中非常重要，字体的实现也非常的复杂。

8.1 字符编码以及CEGUI的字符串类

8.1.1 字符编码和字体的关系

字体和字符编码有什么关系呢？我们知道Windows的字体文件内码都是Unicode的也就是说一个short型的数来表示一个字符，它对应一个点阵图像（或者一个矢量算法）。我们的字符串中保存的都是字符的内码，每个内码对应一个可以显示它的数据（就是点阵图像或者矢量算法）。我们的字符串中保存的都是字符对应的内码（比如char*类型保存多字节内码数据，wchar_t*类型保存Unicode宽字节数据）。字符的编码多种多样各个地区的标准往往相互冲突，比如大陆地区的中文编码有基本集GB2312，全集 GB18030，不是国家标准但被广泛使用的GBK。再比如台湾地区使用的big5编码。虽然大陆地区的各种编码是相互兼容的但是台湾的Big5编码和大陆的就不兼容了。为了解决类似这个问题国际标准化组织制定了Unicode字符集，它收集了世界上绝大多数字符，它可以唯一的表示一个字符不会发生冲突。所以字体（指真正的字体文件，比如宋体的字体文件）的内码采用的是Unicode编码。我们要从一个编码（内码）对应一个图像（字体的点阵）最好使用Unicode字符串。CEGUI也的确是这样做的，它的String类中保存的就是Unicode字符编码，只不过它没有使用short（两字节）型来保存，而是使用int（四字节）型来保存的。这里还要介绍一种编码utf8编码。它也是Unicode的一种编码，全称是（Unicode Transfer Format），主要用于在网络的传输，又因为它的编码最小单位是字节（8位）所以又叫utf8。总之utf8编码也是一种Unicode编码，它适合在网络上传输（因为它的编码中永远不可能出现0。

我们可以简单的认为字体是存储字的描绘信息的对象，而编码是用来代表这个描述信息的一些数字。也就是说从编码到字的描绘信息之间建立了一种映射关系。CEGUI的字体就是使用标准的Unicode编码映射到字的贴图信息的。所有的字体文件也是通过Unicode编码的数字来映射大他们对应的描绘信息的。在这一点上他们是一致的。

CEGUI的字符串类String保存的就是Unicode数据，根据它来映射到数据对应的字体贴图，当渲染一个String字符串的时候就会渲染出对应的字符了。

8.1.2 String类介绍

字符串类String的接口和std::string非常类似，而且许多实现的方法也有相似之处。只不过在构造函数方面，String提供了更多的构造函数。最特别的就是utf8*类型的构造函数。

utf8的定义如下，它被定义成了uint8，utf32被定义成了uint32。

```
typedef     uint8     utf8;
typedef     uint32    utf32;
```

哪么uint8和uint32又是什么呢，相信读者应该已经猜到了。

```
typedef unsigned char   uint8;
typedef unsigned int    uint32;
```

没错uint8被定义成了无符号的字符型，uint32被定义成了无符号的整型。可以看出utf8类型和普通的char类型还是比较相似的，他们都是以字节为单位来保存数据的。（unicode的标准编码是以两个字节为单位来保存数据的，也就是说每个字符都必须使用两个字节来表示）。utf32才是String的保存数据的基本单位，即每个字符使用四个字节来保存。我是如何得知的呢？看下面的定义。

```
typedef     utf32               value_type;    //String类使用的基本字符类型
typedef     size_t              size_type;     //String类使用的尺寸的类型
```

```
typedef        utf32&                reference;          //String基本类型的引用
```

这种定义在STL里非常常见，是典型的泛型编程的代码。String类中使用了这些类型定义。（这里并没有完全的列出）我们接着看String类的数据成员。

```
//保存String类中保存的字符数不包含'\0'
size_type d_cplength;
//当前申请的内存大小（单位是value_type，整型）
size_type d_reserve;
//保存utf8类型的数据，当调用c_str函数时生成
mutable utf8*        d_encodedbuff;
//保存utf8编码数据的大小，一般小于utf32缓冲去(d_buffer)的大小
mutable size_type   d_encodeddatlen;
//保存d_encodedbuff缓冲的大小，这个缓冲区可能比保存的字符要大
mutable size_type   d_encodedbufflen;
//utf32数据的快速缓冲，但String保存的数据大小小于STR_QUICKBUFF_SIZE时使用它
utf32              d_quickbuff[STR_QUICKBUFF_SIZE];
//utf32的数据缓冲，当数据大于STR_QUICKBUFF_SIZE是前者无效，这个缓冲区有效
utf32*             d_buffer;
```

字符串的数据成员比较容易理解，它的快速缓冲的设计非常消耗内存，每个String对象都有这段区域不管它使不使用。建议将它设计的小一些。从d_buffer的类型读者应该可以理解为什么说String类使用四个字节来表示一个字符了吧。为什么CEGUI使用四个字节而不是两个字节来表示一个unicode字符呢？（两个字节完全可以表示啊）。四个字节的前两个字节可以作为标志来出现，这样CEGUI就可以实现复合文档的定义了。比如说改变某个字的颜色，透明度等等。这样设计渲染起来速度比较快，因为这些标记的解析非常快（前
两个字节大于0就表示这四个字节是一个标记了）。

这个字符串类提供了大量的方法，大部分是比较，赋值，操作符的重载等函数。这些函数这里就不做介绍了，他们简单易懂，而且大部分的字符串类都有类似的实现。这里主要介绍CEGUI的几个重要的构造函数。从他们读者可以发现CEGUI的确是保存的unicode数据。String类有许多构成函数，主要提供了这char*，utf8*，std::string和String四种类型的构造函数。下面分别介绍他们。

拷贝构造函数，从String复制。还有另外一个重载的构造函数。

```
String(const String& str, size_type str_idx, size_type str_num = npos)
{
    init();
    assign(str, str_idx, str_num);
}
```

init函数初始化String类的成员变量，assign函数同样有许多重载的函数，负责具体的构造实现。

```
void init(void)
{
    d_reserve             = STR_QUICKBUFF_SIZE;
    d_encodedbuff         = 0;
    d_encodedbufflen   = 0;
    d_encodeddatlen       = 0;
  d_buffer          = 0;
    setlen(0);
}
```

这个函数初始化String类的成员变量。assign函数在String类末尾追加新的数据，这个函数有许多构造函数，读者要区分究竟使用那个函数。

```
String&   assign(const String& str, size_type str_idx = 0, size_type str_num = npos)
{
```

```
    //超出范围了，抛出异常
    if (str.d_cplength < str_idx)
        throw std::out_of_range("Index was out of range for CEGUI::String object");
    //计算需要增长的大小，如果空间不够的话
    if ((str_num == npos) || (str_num > str.d_cplength - str_idx))
        str_num = str.d_cplength - str_idx;
    //增大缓冲区
    grow(str_num);
    //设置新的缓冲长度
    setlen(str_num);
    //拷贝内容到末尾
    memcpy(ptr(), &str.ptr()[str_idx], str_num * sizeof(utf32));
    return *this;
}
```

下面介绍char*类型的构造函数。这个函数同样有一个重载的构造函数。

```
String(const char* chars, size_type chars_len)
{
    init();
    assign(chars, chars_len);
}
```

assign函数，将数据直接转换为utf32类型，这个函数主要是将英文等西方文字转化为utf32类型的，英文的Ascii编码和unicode的编码是相同的，所以可以直接转化为utf32类型。

```
String&   assign(const char* chars, size_type chars_len)
{
    //增加缓冲区的长度
    grow(chars_len);
    utf32* pt = ptr();
    //将字符直接赋值到utf32的缓冲区
    for (size_type i = 0; i < chars_len; ++i)
    {
        *pt++ = static_cast<utf32>(static_cast<unsigned char>(*chars++));
    }
    setlen(chars_len);
    return *this;
}
```

std::string的构造函数，这个函数也有一个重载的构造函数。

```
String(const std::string& std_str, size_type str_idx, size_type str_num = npos)
{
    init();
    assign(std_str, str_idx, str_num);
}
```

assign函数功能和char* 对应的函数类似。

```
String&   assign(const std::string& std_str, size_type str_idx = 0, size_type str_num = npos)
{
    if (std_str.size() < str_idx)
        throw std::out_of_range("Index was out of range for std::string object");
    if ((str_num == npos) || (str_num > (size_type)std_str.size() - str_idx))
        str_num = (size_type)std_str.size() - str_idx;
    grow(str_num);
    setlen(str_num);
    while(str_num--)
```

```
    {
        ((*this)[str_num]) = static_cast<utf32>(static_cast<unsigned char>
        (std_str[str_num + str_idx]));
    }
    return *this;
}
```

最后一个utf8类型的构造函数，这个是重头戏啊。中文的显示需要通过这个构造函数来实现。

```
String(const utf8* utf8_str, size_type chars_len)
{
    init();
    assign(utf8_str, chars_len);
}
```

我们说过utf8编码的数据也是Unicode编码数据，它使用字节为单位保存。哪么utf8究竟是如何保存Unicode的标准编码的呢？看看下面的函数就知道了。

```
String&   assign(const utf8* utf8_str, size_type str_num)
{
    if (str_num == npos)
        throw std::length_error("Length for utf8 encoded string can not be 'npos'");
    size_type enc_sze = encoded_size(utf8_str, str_num);
    grow(enc_sze);
    encode(utf8_str, ptr(), d_reserve, str_num);
    setlen(enc_sze);
    return *this;
}
```

这个函数调用了encode函数将utf8格式的数据转化为标准的Unicode数据。这个函数的实现就可以解释utf8数据到Unicode标准数据的转化了。

```
size_type encode(const utf8* src, utf32* dest, size_type dest_len, size_type src_len = 0) const
{
    // 如果外部没有指定长度，则计算utf8的字节串的长度作为长度
    if (src_len == 0)
    {
        src_len = utf_length(src);
    }
    size_type destCapacity = dest_len;
    // 分析每个字节，根据不同的阀值获取utf8保存的Unicode编码数据
    for (uint idx = 0; ((idx < src_len) && (destCapacity > 0));)
    {
        utf32       cp;
        utf8  cu = src[idx++];
        //简单的英文字符，他和Unicode的编码相同，直接赋值到utf32中
        if (cu < 0x80)
        {
            cp = (utf32)(cu);
        }
        //Unicode编码范围0x0080-0x07FF
        else if (cu < 0xE0)
        {
            cp = ((cu & 0x1F) << 6);
            cp |= (src[idx++] & 0x3F);
        }
        //Unicode编码范围0x0800- 0xFFFF
```

```
            else if (cu < 0xF0)
            {
                cp = ((cu & 0x0F) << 12);
                cp |= ((src[idx++] & 0x3F) << 6);
                cp |= (src[idx++] & 0x3F);
            }
            //这个范围一般不会用到，这里就不考虑了
            else
            {
                cp = ((cu & 0x07) << 18);
                cp |= ((src[idx++] & 0x3F) << 12);
                cp |= ((src[idx++] & 0x3F) << 6);
                cp |= (src[idx++] & 0x3F);
            }
            //utf32的串赋值
            *dest++ = cp;
            --destCapacity;
        }
        return dest_len - destCapacity;
}
```

读者可能对encode函数所做的事情非常不理解，这是因为读者不理解utf8的缘故。如果读者希望彻底理解字符编码，字符集以及字体等内容，请参考专门的书籍或者上网查找相关的帖子。这里不可能详细介绍，下面只简单的介绍什么是Unicode，什么是utf8，以便读者深入的理解CEGUI的实现。

什么是Unicode呢，第8.1节已经介绍过了，它全称"Universal Multiple-Octet Coded Character Set"，简称 UCS, 俗称 "UNICODE"。Unicode废了所有的地区性编码方案，重新搞一个包括了地球上所有文化、所有字母和符号的编码！这种编码可以确保唯一性，从而可以在不同的地区的计算机上都可以正确的显示。总之Unicode编码将世界上绝大部分的字符都编入了，这个从编码到字符的映射是一一对应的。所以只有它才可以作为CEGUI映射字符贴图的编码。CEGUI将Unicode编码映射到这个编码对应的显示数据（贴图），在CEGUI的字体类中实现。

什么是utf8呢？Unicode来到时，一起到来的还有计算机网络的兴起，Unicode如何在网络上传输也是一个必须考虑的问题，于是面向传输的众多 UTF（UCS Transfer Format）标准出现了，顾名思义，UTF8就是每次8个位传输数据，而UTF16就是每次16个位，只不过为了传输时的可靠性，从Unicode到UTF时并不是直接的对应，而是要过一些算法和规则来转换。

受到过网络编程加持的计算机僧侣们都知道，在网络里传递信息时有一个很重要的问题，就是对于数据高低位的解读方式，一些计算机是采用低位先发送的方法，例如我们PC机采用的 INTEL 架构，而另一些是采用高位先发送的方式，在网络中交换数据时，为了核对双方对于高低位的认识是否是一致的，采用了一种很简便的方法，就是在文本流的开始时向对方发送一个标志符——如果之后的文本是高位在位，那就发送"FEFF"，反之，则发送"FFFE"。不信你可以用二进制方式打开一个UTF-X格式的文件，看看开头两个字节是不
是这两个字节？

到这里读者应该对utf8有了大概的了解。utf8到Unicode的转化如何实现呢？读者对照表8-1 理解他们之间的转化。

Unicode编码范围	utf8编码格式	说明
0000 - 007F	0xxxxxxx	英文范围和Ascill编码一致可以直接转化为Unicode值。是不是这个范围的阀值是0x80。
0080- 07FF	110xxxxx 10xxxxxx	这个范围的阀值是0xE0（11100000），因为它是下一种类型的开始部分。

0800- FFFF	1110xxxx 10xxxxxx 10xxxxxx	这个范围的阀值是 0xF0（11110000）这个值刚好 是大于oxE0可以用来判断。

表8-1 Unicode和utf8之间的转化

表8-1中的x表示Unicode对应的值举个例子，例如"汉"字的Unicode编码是6C49。6C49在0800-FFFF之间，所以要用3字节模板：1110xxxx 10xxxxxx 10xxxxxx。将6C49写成二进制是：0110 1100 0100 1001，将这个比特流按三字节模板的分段方法分为0110 ，11000，1 001001，依次代替模板中的x，得到：1110-0110 10-110001 10-001001，即E6 B1 89，这就是其UTF8的编码。

到这里读者应该明白encode函数的功能了吧。如果还是不明白请参考Unicode以及utf8相关的书籍。

下面这三个函数获取一个utf8编码的长度，即它是几个字节的编码，我们知道utf8是便字节的编码，英文字符占一个字节，中文字符少部分占两个字节，大部分占三个字节。

```
//计算一个utf32字符utf8编码所占的字节数
size_type encoded_size(utf32 code_point) const
{
    if (code_point < 0x80)
        return 1;
    else if (code_point < 0x0800)
        return 2;
    else if (code_point < 0x10000)
        return 3;
    else
        return 4;
}
// 计算一定长度的utf32缓冲编码成utf8时所占的字节数
size_type encoded_size(const utf32* buf, size_type len) const
{
    size_type count = 0;
    //循环调用第一个函数，获取总的字节数
    while (len--)
    {
        count += encoded_size(*buf++);
    }
    return count;
}
//计算一个utf32流编码成utf8所需的字节数
size_type encoded_size(const utf32* buf) const
{
    return encoded_size(buf, utf_length(buf));
}
```

在CEGUI中String类是非常庞大的类，它提供了各种字符串操作的函数，这里就不在介绍了。我们主要介绍了和中文显示相关的几个构造函数以及utf8和Unicode之间的关系。不知道读者现在知不知道如何显示中文了。不知道也没关系在第8.4节详细介绍。

8.2 CEGUI字体

字体是CEGUI中文本显示和计算的核心部分。简单的说它根据String类提供的Unicode数据流，映射到对应的字符的显示图像，在描绘的时候将字符对应的图像描绘到游戏窗口内。描绘的位置，对齐方式，描绘的大小等等，这些难题都是字体来解决的。同时字体还支持根据鼠标的位置计算字符的位置，计算一个字符串在特定的格式下的描绘后的宽度，尺寸，等信息。

8.2.1 字体的数据结构

在CEGUI中字体的基类是Font，它派生出了两个类一个是FreeTypeFont处理TTF格式的字体文件，另一个是PixmapFont处理字体的点阵数据，组织结构如图8-1所示。字体基类主要实现了两类功能，一类是描绘一段文字，另一类是计算字描绘这段文字的高度，宽度，根据鼠标的位置计算出鼠标在字符串中的位置等。派生类实现不同的数据的加载，管理等工作。哪么我们先介绍字体基类，然后介绍字体的派生了。

字体基类派生自PropertySet，因此它具备了处理属性的能力，也就是它也可以有属性。字体的属性非常简单，有字体的名称，字体的资源组，字体的对应的数据文件名，字体的分辨率（主要指字体图像的分辨率），字体的自动缩放等。这些属性基本和数据成员一一对应这里就不做介绍了。下面着重介绍字体的各个数据成员，

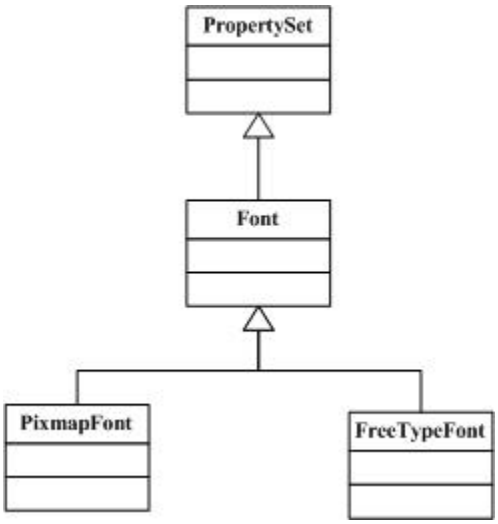

图8-1 字体组织结构

我们还是从类的数据成员开始介绍。从数据入手看一段代码还是很好的方法。

```
//定义字符（utf32）到字体数据（FontGlyph）的映射
typedef std::map<utf32, FontGlyph> CodepointMap;
//定义映射的成员变量
CodepointMap d_cp_map;
//这个字体的名称，设置字体的时候需要根据名称来区分不同的字体
String d_name;
//创建这个字体使用的数据文件，可能是TTF格式的字体文件或者用户定义的图像数据
String d_fileName;
//加载这个资源文件的资源组
String d_resourceGroup;
//保存默认的资源组当d_resourceGroup为空是使用这个资源组
static String d_defaultResourceGroup;
//基线上方的像素数目
float d_ascender;
//基线下方的像素数目
float d_descender;
// 计算公式(ascender - descender) + linegap，表示字体的高度，每个字都一样高
float d_height;
//是否自动缩放
bool d_autoScale;
//水平缩放因子
float d_horzScaling;
//竖直缩放因子
float d_vertScaling;
```

```
//保存字体图像的图像集的元素水平分辨率
float d_nativeHorzRes;
//保存字体图像的图像集的元素竖直分辨率
float d_nativeVertRes;
//字体支持的最大的字符数目
utf32 d_maxCodepoint;
```

我们说过utf32保存的字符的Unicode数据，字体使用它映射到字符的显示数据可以一一映射。在这里得到了证实，CodepointMap就是我们所说的映射了。顺便介绍字体数据保存的类FontGlyph，它保存单个字符的渲染数据。下面是这个类的数据成员。

```
//字符对应的图像指针，用来直接渲染
const Image* d_image;
//渲染一个字符后应该在水平方向移动的基本单位
float d_advance;
```

一个字的渲染后高度和宽度不一定就是字符对应的图像的高度和宽度，可能还有其他的各种复杂的计算。

另一个重要的问题就是文字的排版问题，文字排版有多种格式可以组合。CEGUI的例子中有一个例子Sample_TextDemo，它就是演示文字排版的，读者不妨先看看它的演示。CEGUI中定义了一个枚举来描述文字的排版格式。

```
enum TextFormatting
{
    //所有的文字描绘在一行，而且左对齐
    LeftAligned,
    //所有的文字描绘在一行，而且右对齐
    RightAligned,
    //也是显示在一行，但居中对齐
    Centred,
    //描绘在一行，简单的说就是两端对齐，中间空格和Tab被加宽了
    Justified,
    //当一行显示不下所有的字符的时候，自动换行显示，对齐方式为左对齐
    WordWrapLeftAligned,
    //当一行显示不下所有的字符的时候，自动换行显示，对齐方式为右对齐
    WordWrapRightAligned,
    //当一行显示不下所有的字符的时候，自动换行显示，对齐方式为居中对齐
    WordWrapCentred,
    //显示多行，两端对齐
    WordWrapJustified
};
```

这些对齐方式中前三种可以用到水平和竖直两个方向上，后面几种只能用在水平方向上。读者如果对这些对齐方式还是不太清楚，请打开Word或者WPS，写上几个字然后使用各种对齐方式对齐看看效果应该就会明白了。

好了这一小节就介绍到这里。学习别人写的东西其实是非常枯燥而且容易使人失去耐心的事情。如果读者可以看到这里，恭喜你，你已经完成了长征的一半，马上就会见到彩虹了。程序的学习其实最主要的是它的设计原理，设计思想，以及程序流程。只要抓住了这三点，做起事情来就会得心应手。不会出现似乎看明白了，但让自己做的时候却无从下手。如果读者目前还处于这种状态，则需要在细心的阅读前几章以及CEGUI的源代码了。

8.2.2 文字的描绘

字体负责窗口文字的描绘，大家应该记得在Window类里用过字体基类Font的指针成员。它就是这个窗口渲染文字的字体。哪么文字是如何描绘的呢？大家还记不记得渲染缓冲，文本通过cacheText函数被缓冲起来。在渲染缓冲的描绘函数（render）中恰恰是调用了对应字体的文本渲染函数（drawText）。所以我们才说文字的描述是由字体实现的。这个就是流程的重要性，分析清楚了流程就会理清楚各个类，各个模块之间的关系。也就会清楚这个类或者模块描述什么，应该如何使用他们。笔者不是希望罗列许多的代码到本书上，而是希望读者通过作者的分析从中学习到如何自己分析问题，如何自己下去仔细阅读代码。

CEGUI的代码量这么大，一本书根本不可能把它完全的讲清楚，只能找最重要而且最根本的部分做详细介绍，大量的细节还需要读者自己去体会，去练习。这里要特别的强调练习的重要性，实践出真知，不经过实际的检验，就无法找到自己的不足，就无法真正的理解CEGUI。而且只是看懂了非常容易遗忘。这里讲了一些题外话，主要目的是希望读者多做练习，多思考，多问自己为什么。如果读者能够非常精通CEGUI，可以熟练的制作控件，我相信读者一定可以找到一份薪水颇高的工作。为了将来的美好生活，我们从现在开始认真的学习吧。

要实现一个游戏界面部分，CEGUI提供的字体类还是不够强大。由于CEGUI是英国人设计的，所以它对像汉字这样庞大的字库，支持的并不好。主要的问题是从字体文件（TTF）中获取文字图像（Image）数据会占用大量的内存。而且字体也不支持各种特效，比如说自定义文本颜色，文本闪烁，文本边框等。要实现这些也不是特别复杂，只要读者理解了字体以及CEGUI渲染原理，字体的数据格式等就可以制作了。本书并不打算介绍这些，毕竟这些属于锦上添花的东西。

下面我们首先介绍描绘文本的接口函数drawText，CEGUI中提供了许多重载的drawText函数，我们这里只介绍最终被调用到的具体实现函数。首先介绍这个函数的参数，text要描绘的文本串，draw_area文本被描绘的区域，z表示描绘的深度信息，一般来说没有用，在将渲染的时候曾经讲过。clip_rect裁剪矩形，超出这个矩形的部分将不会显示出来，fmt就是我们前面介绍的排版的格式，colours颜色矩形分别代表矩形四个点的颜色值，适当的设置它的值可以实现不同的文字显示效果，比如从左上角到右下角的渐变颜色，x_scale和
y_scale是在水平和竖直方向的缩放因子。返回描绘的行数。这个函数并不是真正的负责渲染的函数，它只是根据各种格式，调用不同的具体实现函数来描绘的。

```
size_t Font::drawText(const String& text, const Rect& draw_area, float z, const Rect& clip_rect, TextFormatting fmt, const ColourRect& colours, float x_scale, float y_scale)
{
   size_t thisCount;
   size_t lineCount = 0;
      //什么是基线呢，参见下文介绍
   float    y_base = draw_area.d_top + getBaseline(y_scale);
   Rect tmpDrawArea(
      PixelAligned(draw_area.d_left),
      PixelAligned(draw_area.d_top),
      PixelAligned(draw_area.d_right),
      PixelAligned(draw_area.d_bottom)
   );
   size_t lineStart = 0, lineEnd = 0;
   String  currLine;
   while (lineEnd < text.length())
   {
            //找到\n换行符，这是强制换行，如果Wrap格式的话可能这里边还会自动换行
      if ((lineEnd = text.find_first_of('\n', lineStart)) == String::npos)
         lineEnd = text.length();
            //得到当前的这行的文本
      currLine = text.substr(lineStart, lineEnd - lineStart);
      lineStart = lineEnd + 1;  // +1 跳过字符 '\n'
            //根据不同的格式，调用不同的函数，并且计算行数和下一列的高度(y_base)
      switch(fmt)
      {
      case LeftAligned:
         drawTextLine(currLine, Vector3(tmpDrawArea.d_left, y_base, z), clip_rect,
                  colours, x_scale, y_scale);
         thisCount = 1;
         y_base += getLineSpacing(y_scale);
         break;
```

```
        case RightAligned:
            drawTextLine(currLine, Vector3(tmpDrawArea.d_right -
                    getTextExtent(currLine, x_scale), y_base, z), clip_rect, colours, x_scale, y_scale);
            thisCount = 1;
            y_base += getLineSpacing(y_scale);
            break;
        case Centred:
            drawTextLine(currLine, Vector3(PixelAligned(tmpDrawArea.d_left +
                        ((tmpDrawArea.getWidth() - getTextExtent(currLine, x_scale)) / 2.0f)), y_base, z),
                    clip_rect, colours, x_scale, y_scale);
            thisCount = 1;
            y_base += getLineSpacing(y_scale);
            break;
        case Justified:
            drawTextLineJustified(currLine, draw_area,
                    Vector3(tmpDrawArea.d_left, y_base, z), clip_rect, colours, x_scale, y_scale);
            thisCount = 1;
            y_base += getLineSpacing(y_scale);
            break;
        case WordWrapLeftAligned:
            thisCount = drawWrappedText(currLine, tmpDrawArea, z, clip_rect,
                    LeftAligned, colours, x_scale, y_scale);
            tmpDrawArea.d_top += thisCount * getLineSpacing(y_scale);
            break;
        case WordWrapRightAligned:
            thisCount = drawWrappedText(currLine, tmpDrawArea, z, clip_rect, RightAligned,
                colours, x_scale, y_scale);
            tmpDrawArea.d_top += thisCount * getLineSpacing(y_scale);
            break;
        case WordWrapCentred:
            thisCount = drawWrappedText(currLine, tmpDrawArea, z, clip_rect, Centred,
                colours, x_scale, y_scale);
            tmpDrawArea.d_top += thisCount * getLineSpacing(y_scale);
            break;
        case WordWrapJustified:
            thisCount = drawWrappedText(currLine, tmpDrawArea, z, clip_rect, Justified,
                colours, x_scale, y_scale);
            tmpDrawArea.d_top += thisCount * getLineSpacing(y_scale);
            break;
        default:
            throw InvalidRequestException("Font::drawText - Unknown or unsupported
                    TextFormatting value specified.");
        }
            //递增行数
        lineCount += thisCount;
    }
    // 最少返回一行
    return ceguimax(lineCount, (size_t)1);
}
```

这个函数根据当前的文本排版格式调用不同的函数，如果是单行而且不是Justified格式的就调用drawTextLine函数，如果是自动换行的就调用drawWrappedText函数，如果是Justified单行格式就调用drawTextLineJustified函数。函数getLineSpacing获取当前字体的行高。

什么是字体的基线呢？汉字的基线在字符的最底下，而西方字符的就不同了。还记得拼音的写法吗？里面有三行，有的字母写在上两行，有的写在下两行，有的三行都有。基线可以理解为第二道线。

下面介绍这三个文本描绘函数。为了和以前drawTextLine兼容，提供了drawTextLineJustified函数来描绘Justified格式的文本。首先介绍drawTextLine函数，这个函数描绘一整行文本，这行文本不会自动换行，即使文本的描绘长度大于描绘区域。

```
void Font::drawTextLine(const String& text, const Vector3& position, const Rect& clip_rect, const ColourRect& colours, float x_scale, float y_scale)
{
   Vector3      cur_pos(position);
   const FontGlyph* glyph;
   float base_y = position.d_y;
      //遍历字符串中所有的Unicode字符，并且渲染他们
   for (size_t c = 0; c < text.length(); ++c)
   {
            //获取字符对应的图像数据，从我们说的映射中获取
      glyph = getGlyphData(text[c]);
      if (glyph)
      {
                  //获取字符对应数据的图像
         const Image* img = glyph->getImage();
                  //计算新的文本高度
         cur_pos.d_y = base_y - (img->getOffsetY() - img->getOffsetY() * y_scale);
                  //描绘字符对应的图像
         img->draw(cur_pos, glyph->getSize(x_scale, y_scale), clip_rect, colours);
                  //递增当前的文本X位置，增量是这个字符的宽度，为下一个字符描绘做准备
         cur_pos.d_x += glyph->getAdvance(x_scale);
      }
   }
}
```

另一个描绘单行Justified格式的函数。从这个函数的实现也可以看出Justified格式的含义。这个函数首先计算左对齐和Justified格式渲染同样文本的宽度之差，然后计算文本中空白和TAB的数量，计算每个空白和TAB多占用的空间。最后在描绘的时候遇到空白和TAB就增加X方向的位置。最终的描绘结果就是文本整个覆盖了描绘区域

```
void Font::drawTextLineJustified (const String& text, const Rect& draw_area, const Vector3& position, const Rect& clip_rect, const ColourRect& colours, float x_scale, float y_scale)
{
   Vector3      cur_pos(position);
   const FontGlyph* glyph;
   float base_y = position.d_y;
   size_t char_count = text.length();
   //计算Justified格式和LeftAligned格式的文本之间描绘的大小之差
   float lost_space = getFormattedTextExtent(text, draw_area, Justified, x_scale) -
      getTextExtent(text, x_scale);
   //保存TAB和空格字符的个数
   uint space_count = 0;
   size_t c;
   for (c = 0; c < char_count; ++c)
      if ((text[c] == ' ') || (text[c] == '\t'))
         ++space_count;
   //计算每个空白和TAB占用的空间
   float shared_lost_space = 0.0;
   if (space_count > 0)
```

```
        shared_lost_space = lost_space / (float)space_count;
        //遍历每个字符然后描绘它
    for (c = 0; c < char_count; ++c)
    {
        glyph = getGlyphData(text[c]);
        if (glyph)
        {
            const Image* img = glyph->getImage();
            cur_pos.d_y = base_y - (img->getOffsetY() - img->getOffsetY() * y_scale);
            img->draw(cur_pos, glyph->getSize(x_scale, y_scale), clip_rect, colours);
            cur_pos.d_x += glyph->getAdvance(x_scale);
            // 调整空白和TAB的宽度
            if ((text[c] == ' ') || (text[c] == '\t'))
                cur_pos.d_x += shared_lost_space;
        }
    }
}
```

getFormattedTextExtent函数获取这个字体描绘这个文本在特定的格式下的占用的空间。这个函数将会在第8.2.3节介绍。最后一个描绘函数，描绘可以自动换行的文本，如果文本的描绘宽度在一行内描绘不下则自动换行。

```
size_t Font::drawWrappedText(const String& text, const Rect& draw_area, float z, const Rect& clip_rect, TextFormatting fmt, const ColourRect& colours, float x_scale, float y_scale)
{
    size_t  line_count = 0;
    Rect    dest_area(draw_area);
    float   wrap_width = draw_area.getWidth();
    String  whitespace = TextUtils::DefaultWhitespace;
    String  thisLine, thisWord;
    size_t  currpos = 0;
    //获取第一个字符
    currpos += getNextWord(text, currpos, thisLine);
    // while there are words left in the string...
    while (String::npos != text.find_first_not_of(whitespace, currpos))
    {
        // get next word of the string...
        currpos += getNextWord(text, currpos, thisWord);
        //如果文本一行描绘不下则分为多行描绘，找到最后一个画不下的字符
        if ((getTextExtent(thisLine, x_scale) + getTextExtent(thisWord, x_scale)) > wrap_width)
        {
            //描绘这一行
            line_count += drawText(thisLine, dest_area, z, clip_rect, fmt, colours, x_scale,
                y_scale);
            //移除换行符
            thisWord = thisWord.substr(thisWord.find_first_not_of(whitespace));
            //重设新的行内容
            thisLine.clear();
            //更新高度信息
            dest_area.d_top += getLineSpacing(y_scale);
        }
        //如果一行可以描绘的下则将这个字符添加到这行
        thisLine += thisWord;
    }
```

```
    //描绘最后一行，取消最后一行的Justified格式
    TextFormatting last_fmt = (fmt == Justified ? LeftAligned : fmt);
    //描绘最后一行
    line_count += drawText(thisLine, dest_area, z, clip_rect, last_fmt, colours, x_scale, y_scale);
    return line_count;
}
```

描绘函数介绍到这里。下面介绍一个字体描绘数据的函数getGlyphData，这个函数首先检查是否字符对应的图像是否加载了，如果没有加载就加载对应的图像，最后返回对应图像。其中d_maxCodepoint变量保存字体中最大的字符编号，可以作为字符是否存在的标志，如果字符大于这个值则字符肯定不存在。

```
const FontGlyph *Font::getGlyphData (utf32 codepoint)
{
    if (codepoint > d_maxCodepoint)
        return 0;
    if (d_glyphPageLoaded)
    {
        //检查codepoint对应的图像是否已经被创建和加载了，如果没有就创建和加载
        uint page = codepoint / GLYPHS_PER_PAGE;
        uint mask = 1 << (page & (BITS_PER_UINT - 1));
        if (!(d_glyphPageLoaded [page / BITS_PER_UINT] & mask))
        {
                    //设置已经加载的标志，保证下次不会在被加载
            d_glyphPageLoaded [page / BITS_PER_UINT] |= mask;
                    //加载字体的图像
            rasterize (codepoint & ~(GLYPHS_PER_PAGE - 1),
                    codepoint | (GLYPHS_PER_PAGE - 1));
        }
    }
    CodepointMap::const_iterator pos = d_cp_map.find (codepoint);
    return (pos != d_cp_map.end()) ? &pos->second : 0;
}
```

函数rasterize是派生类必须实现的加载字体数据到CEGUI图像的方法。这一节就主要介绍了3个字符串的描绘函数，字体描绘的统一接口叫做drawText函数，他会调用其他三个具体实现函数，最终将图像描绘到窗口对应的位置上。CEGUI描绘必须使用图像来进行，所以字符描绘数据必须写入到图像中。下一小节介绍字体提供的各种计算函数。

8.2.3 字符串计算

字体提供了计算字符串描绘宽度的函数getTextExtent，getWrappedTextExtent和getFormattedTextExtent，第一个函数获取非格式化的字符串的描绘宽度，第二个函数获取自动换行文本的最大宽度，最后一个获取格式化的字符串描绘宽度。提供了获取描绘行数的计算函数getFormattedLineCount，这个函数返回对应字符串描绘后的长度。提供了根据鼠标的位置计算鼠标在字符串中位置的函数getCharAtPixel，这个函数返回鼠标在字符串中那个字符附近。本节将详细介绍这四个函数。首先介绍getTextExtent函数。

```
float Font::getTextExtent(const String& text, float x_scale)
{
    const FontGlyph* glyph;
    float cur_extent = 0, adv_extent = 0, width;
    for (size_t c = 0; c < text.length(); ++c)
    {
        glyph = getGlyphData(text[c]);
        if (glyph)
        {
                    //或者对应字符图像的渲染后宽度，图像的宽度加上偏移
```

```
            width = glyph->getRenderedAdvance(x_scale);
                    //递增当前宽度
            if (adv_extent + width > cur_extent)
                cur_extent = adv_extent + width;
                    //递增字体的真正宽度
            adv_extent += glyph->getAdvance(x_scale);
        }
    }
        //返回两个中最大的作为渲染后的宽度
    return ceguimax(adv_extent, cur_extent);
}
```

这个函数最终返回的是所有字符宽度之和与所有字体宽度之和加上最大一个图像偏移中大的一个，作为描绘这个字符的最终宽度。这个函数可以说非常有用。在设计控件的时候常常要计算一个字符串描绘后的宽度，最终用来决定窗口的宽度。Window系统中ＧＤＩ函数也有类似的函数，名称是GetTextExtentPoint。下面介绍这个函数的另一个兄弟函数，它支持可换行的字符串描绘的宽度获取。

```
float Font::getWrappedTextExtent(const String& text, float wrapWidth, float x_scale)
{
    String  whitespace = TextUtils::DefaultWhitespace;
    String  thisWord;
    size_t  currpos;
    float   lineWidth, wordWidth;
    float   widest = 0;
    currpos = getNextWord (text, 0, thisWord);
    lineWidth = getTextExtent (thisWord, x_scale);
    //循环扫描所有的行，这个步骤和drawText里类似
    while (String::npos != text.find_first_not_of (whitespace, currpos))
    {
        currpos += getNextWord (text, currpos, thisWord);
        wordWidth = getTextExtent (thisWord, x_scale);
        //如果新增一个函数就太宽了
        if ((lineWidth + wordWidth) > wrapWidth)
        {
                    //修改最大宽度值
            if (lineWidth > widest)
                widest = lineWidth;
            //移除无效字符
            thisWord = thisWord.substr (thisWord.find_first_not_of (whitespace));
            wordWidth = getTextExtent (thisWord, x_scale);
            // 重设行宽值为0
            lineWidth = 0;
        }
        //添加下一个字符到当前行，如果它增加它不会使这一行超出wrapWidth
        lineWidth += wordWidth;
    }
        //修改最大宽度值
    if (lineWidth > widest)
        widest = lineWidth;
    return widest;
}
```

这个函数返回所有行文本中宽度最大的哪一行。下面介绍格式化字符串的宽度获取函数，它将会调用这两个单行和多行获取宽度的函数。

```
//这个函数有些像drawText函数
float Font::getFormattedTextExtent(const String& text, const Rect& format_area, TextFormatting fmt, float x_scale)
{
   float lineWidth;
   float widest = 0;
   size_t lineStart = 0, lineEnd = 0;
   String  currLine;
   while (lineEnd < text.length())
   {
      if ((lineEnd = text.find_first_of('\n', lineStart)) == String::npos)
      {
         lineEnd = text.length();
      }
      currLine = text.substr(lineStart, lineEnd - lineStart);
      lineStart = lineEnd + 1;  //跳过 '\n'字符
      switch(fmt)
      {
      case Centred:
      case RightAligned:
      case LeftAligned:
         lineWidth = getTextExtent(currLine, x_scale);
         break;
      case Justified:
         // 确保返回的值是区域宽度和文本描绘宽度中最大的
         lineWidth = ceguimax(format_area.getWidth(), getTextExtent(currLine, x_scale));
         break;
      case WordWrapLeftAligned:
      case WordWrapRightAligned:
      case WordWrapCentred:
         lineWidth = getWrappedTextExtent(currLine, format_area.getWidth(), x_scale);
         break;
      case WordWrapJustified:
         //和Justified类似
         lineWidth = ceguimax(format_area.getWidth(), getWrappedTextExtent(currLine,
            format_area.getWidth(), x_scale));
         break;
      default:
         throw InvalidRequestException("Font::getFormattedTextExtent - Unknown or
                  unsupported TextFormatting value specified.");
      }
         //循环分析所有的行，返回最大的那行的宽度值
      if (lineWidth > widest)
      {
         widest = lineWidth;
      }
   }
   return widest;
}
```

这个函数是getTextExtent以及getWrappedTextExtent函数的包装，它根据不同的格式返回最终的宽度。接着我们介绍getCharAtPixel函数。

```
size_t Font::getCharAtPixel(const String& text, size_t start_char, float pixel, float x_scale)
{
   const FontGlyph* glyph;
   float cur_extent = 0;
   size_t char_count = text.length();
   // 处理最简单的情况，如果超出了字符串的范围直接返回
   if ((pixel <= 0) || (char_count <= start_char))
      return start_char;

      //循环计算从start_char开始每增加一个字符是不是鼠标的位置小于这个字符增加后的
//宽度，如果是就返回这个字符的位置
      for (size_t c = start_char; c < char_count; ++c)
   {
      glyph = getGlyphData(text[c]);
      if (glyph)
      {
         cur_extent += glyph->getAdvance(x_scale);
         if (pixel < cur_extent)
            return c;
      }
   }
   return char_count;
}
```

这个函数也非常有用，比如单行文本框当鼠标单击的时候需要计算当前光标的位置，就需要通过这个函数获取光标在字符串之中的位置，然后重新设置光标的位置。最后一个函数getFormattedLineCount，它获取字符串描绘行数。这个函数我们就不做介绍了，读者可以自己阅读它的实现，我们将它作为一个习题。

有关字符串计算的函数我们就介绍到这里。需要注意的是这几个函数都非常有用，在设计控件的时候经常需要用到他们。

8.2.4 字体文件

字体也是一种资源他会在CEGUI加载模式的时候加载到系统。定义字体的也是一些XML文件，CEGUI官方的扩展名是".font"，可以在模式文件中找到字体文件的引用。下面是从一个模式文件中摘抄出来的。

```
<Font Name="Commonwealth-10" Filename="Commonwealth-10.font" />
```

这一句定义了一个字体文件的位置和字体的名称。字体名叫Commonwealth-10，定义在叫做Commonwealth-10.font的文件里。哪么我们看看这个字体文件的内容。

```
<?xml version="1.0" ?>
<Font Name="Commonwealth-10" Filename="Commonv2c.ttf" Type="FreeType" Size="10" NativeHorzRes="800"
NativeVertRes="600" AutoScaled="true"/>
```

这里面定义了一个叫做Commonwealth-10的字体，它的字体数据文件是Commonv2c.ttf，它的类型是FreeType的（TTF），不是位图格式的，尺寸是10像素（可以自己调整），水平和竖直分辨率是800和600，支持自动缩放。这个文件是TTF格式字体的文件，如果是位图字体的文件则还需要多一些元素的定义。格式如下所示，只是大概的示例。

```
<Mapping      Codepoint = "" Image ="" HorzAdvance = "" />
```

这段的意义是将一个字符（Codepoint）映射到一个图像（Image），也就是说定义图像的工作转移到字体文件里了。TTF格式定义字符到图像的映射是动态进行的。

CEGUI在加载这个字体的时候会自动创建TTF类型的字体对象，并且调用它的load函数执行具体的加载字体工作。字体文件的解析和其他各种文件解析类似，处理类名叫Font_xmlHandler，读者可以自己看看，比较简单。读者可以思考，如果需要显示中文是不是要对字体的定义做一些修改呢？具体如何修改请参考第8.4节。

8.2.5 位图字体

位图字体（PixmapFont）保存一张图像集。这张图像集保存所有的字符的图像，显然只有西方的文字才能在一张图像集上保存的下。中文显然无法全部保存在一张图像集上，必须
像TTF字体一样有个映射才行。下面是位图字体的数据成员。

```
//保存字符图像的图像集指针
Imageset *d_glyphImages;
//X轴的缩放因子
float d_origHorzScaling;
//如果图像集是我们创建的则为true
bool d_imagesetOwner;
```

这个类重载了基类Font的两个纯虚函数updateFont和load函数。下面我们着重介绍他们。首先介绍load函数，这个函数会在加载字体文件的时候调用。

```
void PixmapFont::load ()
{
   // 记录开始加载字体的日志
   Logger::getSingleton ().logEvent ("Started creation of Pixmap Font:");
   Logger::getSingleton ().logEvent ("---- CEGUI font name: " + d_name);
   Logger::getSingleton ().logEvent ("----     Source file: " + d_fileName +
   " in resource group: " + (d_resourceGroup.empty () ? "(Default)" : d_resourceGroup));
      //关键的一步，更新字体
   updateFont ();
   char tmp [50];
   snprintf (tmp, sizeof (tmp), "Succsessfully loaded %d glyphs", d_cp_map.size ());
   Logger::getSingleton ().logEvent (tmp);
}
```

我们看到load函数，其实就是记录了日志，最终的工作还是交给了updateFont函数了，哪么现在我们就介绍这个函数。

```
void PixmapFont::updateFont ()
{
   float factor = (d_autoScale ? d_horzScaling : 1.0f) / d_origHorzScaling;
   d_ascender = 0;
   d_descender = 0;
   d_height = 0;
   d_maxCodepoint = 0;
      //设置图像集的各个属性
   d_glyphImages->setAutoScalingEnabled (d_autoScale);
   d_glyphImages->setNativeResolution (Size (d_nativeHorzRes, d_nativeVertRes));
      //循环遍历所有的字符集，设置每个字符图像的信息
   for (CodepointMap::const_iterator i = d_cp_map.begin (); i != d_cp_map.end (); ++i)
   {
      if (i->first > d_maxCodepoint)
         d_maxCodepoint = i->first;
      ((FontGlyph &)i->second).setAdvance (i->second.getAdvance () * factor);
      const Image *img = i->second.getImage ();
      if (img->getOffsetY () < d_ascender)
         d_ascender = img->getOffsetY ();
      if (img->getHeight () + img->getOffsetY () > d_descender)
         d_descender = img->getHeight () + img->getOffsetY ();
   }
      //计算这个字体的各种信息
   d_ascender = -d_ascender;
```

```
    d_descender = -d_descender;
    d_height = d_ascender - d_descender;
    if (d_autoScale)
        d_origHorzScaling = d_horzScaling;
    else
        d_origHorzScaling = 1.0;
}
```

可以看到位图字体使用已经定义好的图像集来描绘字符，哪么这个图像集的图像是如何与字符对应的呢？第8.2.4节介绍Mapping元素，它映射字符到图像集。所以这里才可以直接根据字符获取到图像，然后设置图像的各种信息。哪么关联这两项的函数是那个呢？它就是defineMapping 函数了。

```
void PixmapFont::defineMapping (String image_name, utf32 codepoint, float horzAdvance)
{
    const Image *image = &d_glyphImages->getImage (image_name);
    //如果没有指定宽度则在这里计算
    if (horzAdvance == -1.0f)
        horzAdvance = (float)(int)(image->getWidth () + image->getOffsetX ());
    if (d_autoScale)
        horzAdvance *= d_origHorzScaling;
    //创建字符到图像数据的映射
    d_cp_map [codepoint] = FontGlyph (horzAdvance, image);
}
```

这个函数会在Font_xmlHandler::elementStart函数中被调用（调用的是另一个重载的函数，但最终会调用这个实现函数），当遇到Mapping元素的时候。

从图像字体的实现我们可以看出CEGUI支持自定义的图像作为字符的显示数据，但它只能支持一个图像集，也就是说最大支持256（每个图像集最多的图像个数）个字符的显示。这样显然是不支持中文的显示了，哪么中文如何使用位图字体呢？这就需要自己定义一种字体，使用这种字体可以保存多个图像集。

8.2.6 TTF字体

经过第8.2.5节的介绍大家已经知道图像字体是无法支持中文的显示的，哪么CEGUI能否直接支持中文的显示呢？答案是肯定的，那就是通过TTF字体（FreeTypeFont类）可以实现中文的显示。TTF字体将TTF字体文件作为它的图像的数据源。使用开源TTF
字体支持库来实现字体的读取。这个开源库就是FreeType库，它支持TTF格式字体库的字符对应数据的读取。这里不打算介绍这个库，读者可以参考CEGUI，TTF字体的实现或者可以阅读FreeType库的说明等。下面介绍这个类的成员变量。

```
typedef std::vector<Imageset*> ImagesetVector;
//定义图像集数组，它可以保存任意多个图像集
ImagesetVector d_glyphImages;
//字体的大小，单位像素
float d_ptSize;
//字符是否被渲染为抗锯齿
bool d_antiAliased;
//字体库FreeType的字体句柄
FT_Face d_fontFace;
//字体文件的数据，通过资源提供器来加载的原始数据
RawDataContainer d_fontData;
```

通过这个类的图像集数组就可以理解为什么这个字体类可以支持中文显示了。CEGUI使用RGBA格式的纹理来渲染字符，显然这个还是很浪费内存空间的。读者可以尝试着修改为R4G4B4A4格式的纹理。下面介绍字体的更新函数，加载函数和位图字体类似这里就不做介绍了。

```
void FreeTypeFont::updateFont ()
```

```
{
    //free函数是否所有的图像集和FreeType库的字体句柄
  free ();
    //加载字体的原始资源
  System::getSingleton ().getResourceProvider ()->loadRawDataContainer (
    d_fileName, d_fontData, d_resourceGroup.empty () ?
    getDefaultResourceGroup () : d_resourceGroup);
  //创建一个新的FreeType库的句柄，通过原始数据，创建失败的话抛出异常
  if (FT_New_Memory_Face (ft_lib, d_fontData.getDataPtr (),
      static_cast<FT_Long>(d_fontData.getSize ()), 0, &d_fontFace) != 0)
    throw GenericException ("FreeTypeFont::load - The source font file '" +
          d_fileName +"' does not contain a valid FreeType font.");
  //检查是否默认的Unicode字符映射可用，如果不可用则释放字体句柄并抛出异常
  if (!d_fontFace->charmap)
  {
    FT_Done_Face (d_fontFace);
    d_fontFace = 0;
    throw GenericException ("FreeTypeFont::load - The font '" + d_name +
          "' does not have a Unicode charmap, and cannot be used.");
  }
    //获取水平和竖直DPI（点或者像素每英寸，一般显示器都是96）
  uint horzdpi = System::getSingleton ().getRenderer ()->getHorzScreenDPI ();
  uint vertdpi = System::getSingleton ().getRenderer ()->getVertScreenDPI ();

  float hps = d_ptSize * 64;
  float vps = d_ptSize * 64;
  if (d_autoScale)
  {
    hps *= d_horzScaling;
    vps *= d_vertScaling;
  }
    //设置字体的大小，究竟如何设置以及参数的含义，请参考FreeType的教程
  if (FT_Set_Char_Size (d_fontFace, FT_F26Dot6 (hps), FT_F26Dot6 (vps), horzdpi, vertdpi))
  {
    //执行到这里说明字体不支持设置的大小，这里尝试设置最接近的尺寸
    float ptSize_72 = (d_ptSize * 72.0f) / vertdpi;
    float best_delta = 99999;
    float best_size = 0;
    for (int i = 0; i < d_fontFace->num_fixed_sizes; i++)
    {
      float size = d_fontFace->available_sizes [i].size * float(FT_POS_COEF);
      float delta = fabs (size - ptSize_72);
      if (delta < best_delta)
      {
        best_delta = delta;
        best_size = size;
      }
    }
    if ((best_size <= 0) ||
      FT_Set_Char_Size (d_fontFace, 0, FT_F26Dot6 (best_size * 64), 0, 0))
    {
      char size [20];
```

```
            snprintf (size, sizeof (size), "%g", d_ptSize);
            throw GenericException ("FreeTypeFont::load - The font '" + d_name +
                    "' cannot be rasterized at a size of " + size + " points, and cannot be used.");
        }
    }
        //字体是否支持缩放
    if (d_fontFace->face_flags & FT_FACE_FLAG_SCALABLE)
    {
        float y_scale = d_fontFace->size->metrics.y_scale * float(FT_POS_COEF) *
                    (1.0f/65536.0f);
        d_ascender = d_fontFace->ascender * y_scale;
        d_descender = d_fontFace->descender * y_scale;
        d_height = d_fontFace->height * y_scale;
    }
    else
    {
        d_ascender = d_fontFace->size->metrics.ascender * float(FT_POS_COEF);
        d_descender = d_fontFace->size->metrics.descender * float(FT_POS_COEF);
        d_height = d_fontFace->size->metrics.height * float(FT_POS_COEF);
    }
    //创建空的FontGlyph
    FT_UInt gindex;
    FT_ULong codepoint = FT_Get_First_Char (d_fontFace, &gindex);
    FT_ULong max_codepoint = codepoint;
        //循环遍历所有的字体中的字符
    while (gindex)
    {
        if (max_codepoint < codepoint)
            max_codepoint = codepoint;
        //查询字体的信息，不是渲染字体
        if (FT_Load_Char(d_fontFace, codepoint,
                    FT_LOAD_DEFAULT | FT_LOAD_FORCE_AUTOHINT))
            continue; // 出错的话继续
        float adv = d_fontFace->glyph->metrics.horiAdvance * float(FT_POS_COEF);
        //创建空的FontGlyph映射，字体图像现在为空
        d_cp_map[codepoint] = FontGlyph (adv);
        //处理下一个字符
        codepoint = FT_Get_Next_Char (d_fontFace, codepoint, &gindex);
    }
    setMaxCodepoint (max_codepoint);
}
```

需要注意的是FreeType的大多数函数都是出错返回不等于0的值（一般是大于0的值）。updateFont函数只是设置字体的大小，创建所有空的字体图像映射，真正的图像数据还没有被加载。真正的加载图像数据的函数是rasterize函数。

```
void FreeTypeFont::rasterize (utf32 start_codepoint, utf32 end_codepoint)
{
    CodepointMap::const_iterator s = d_cp_map.lower_bound(start_codepoint);
    if (s == d_cp_map.end ())
        return;
    CodepointMap::const_iterator orig_s = s;
    CodepointMap::const_iterator e = d_cp_map.upper_bound (end_codepoint);
```

```
  while (true)
  {
    uint texsize = getTextureSize (s, e);
    // 如果所有的字符都已经渲染到图像中则直接退出
    if (!texsize)
      break;
        //创建一个新的图像集用来保存图像
    Imageset *is = ImagesetManager::getSingleton ().createImageset (
      d_name + "_auto_glyph_images_" + int (s->first),
      System::getSingleton ().getRenderer ()->createTexture ());
    d_glyphImages.push_back (is);
    //创建渲染的内存，用于渲染字符显示数据到这里
    argb_t *mem_buffer = new argb_t [texsize * texsize];
    memset (mem_buffer, 0, texsize * texsize * sizeof (argb_t));
    uint x = INTER_GLYPH_PAD_SPACE, y = INTER_GLYPH_PAD_SPACE;
    uint yb = INTER_GLYPH_PAD_SPACE;
    // 是否所有的字符都渲染完成的标志变量
    bool finished = false;
    bool forward = true;
    //渲染所有的从s到e的字符，可能会渲染超出这个范围但不能小于这个范围
    while (s != d_cp_map.end())
    {
      // 检查是否我们已经完成了所有字符的渲染
      finished |= (s == e);
      // Check if glyph already rendered
      if (!s->second.getImage())
      {
        // Render the glyph
        if (FT_Load_Char (d_fontFace, s->first, FT_LOAD_RENDER |
                          FT_LOAD_FORCE_AUTOHINT |
              (d_antiAliased ? FT_LOAD_TARGET_NORMAL :
                               FT_LOAD_TARGET_MONO)))
        {
          // 渲染出错这里记录错误日志（省略）并且创建空的图像
          Rect area(0, 0, 0, 0);
          Point offset(0, 0);
          String name;
          name += s->first;
          is->defineImage(name, area, offset);
          ((FontGlyph &)s->second).setImage(&is->getImage(name));
        }
        else
        {
          uint glyph_w = d_fontFace->glyph->bitmap.width +
                                       INTER_GLYPH_PAD_SPACE;
          uint glyph_h = d_fontFace->glyph->bitmap.rows +
                                  INTER_GLYPH_PAD_SPACE;
          //检查是否右边超过了字体的宽度
          uint x_next = x + glyph_w;
          if (x_next > texsize)
          {
            x = INTER_GLYPH_PAD_SPACE;
```

```
                x_next = x + glyph_w;
                y = yb;
            }
            // 检查是否超出图像集的高度
            uint y_bot = y + glyph_h;
            if (y_bot > texsize)
                break;
            //拷贝数据，数据格式是RGBA
            drawGlyphToBuffer (mem_buffer + (y * texsize) + x, texsize);
            //创建新的图像
            Rect area (float(x), float(y), float(x + glyph_w -
                                    INTER_GLYPH_PAD_SPACE),
            float(y + glyph_h - INTER_GLYPH_PAD_SPACE));
            Point offset (d_fontFace->glyph->metrics.horiBearingX *
                                    float(FT_POS_COEF),
            -d_fontFace->glyph->metrics.horiBearingY * float(FT_POS_COEF));
            String name;
            name += s->first;
            is->defineImage (name, area, offset);
            ((FontGlyph &)s->second).setImage (&is->getImage (name));
            // 宽度递增到下一位置
            x = x_next;
            if (y_bot > yb)
            {
                yb = y_bot;
            }
        }
    }
    // 到下一个数据
    if (forward)
        if (++s == d_cp_map.end ())
        {
            finished = true;
            forward = false;
            s = orig_s;
        }
    //向前
    if (!forward)
        if (--s == d_cp_map.begin ())
            break;
  }
  //拷贝我们的数据内存到图像集对应贴图的内存空间，并且删除我们的缓冲
  is->getTexture ()->loadFromMemory (mem_buffer, texsize, texsize,
                          Texture::PF_RGBA);
  delete [] mem_buffer;
  if (finished)
    break;
 }
}
```

这个函数很大，而且比较凌乱，非常难理解，最终要的是还有许多的FreeType的函数调用，就更加增大了阅读的难度。不过读者可以把握这样一条基本规律。这个函数，加载从

start_codepoint到 end_codepoint的字符的渲染数据到这些字符对应的图像中。而updateFont只是创建了对应的映射，图像都是空的，这个函数创建了图像并且拷贝字符显示数据到图像中（通过转化将FreeType提供的数据，也就是点阵数据或者灰度数据，转化为纹理的数据格式，实现这一步的函数是drawGlyphToBuffer）。下面我们就介绍这个转化数据格式的函数。

```
void FreeTypeFont::drawGlyphToBuffer (argb_t *buffer, uint buf_width)
{
  FT_Bitmap *glyph_bitmap = &d_fontFace->glyph->bitmap;
    //遍历字符图像的数据的所有行
  for (int i = 0; i < glyph_bitmap->rows; ++i)
  {
    uchar *src = glyph_bitmap->buffer + (i * glyph_bitmap->pitch);
    switch (glyph_bitmap->pixel_mode)
    {
            //灰度格式的数据
      case FT_PIXEL_MODE_GRAY:
        {
          uchar *dst = reinterpret_cast<uchar*> (buffer);
          for (int j = 0; j < glyph_bitmap->width; ++j)
          {
            // RGBA
            *dst++ = 0xFF;
            *dst++ = 0xFF;
            *dst++ = 0xFF;
            *dst++ = *src++;
          }
        }
        break;
            //点阵格式的数据，如果这个点是1则为白色否则为黑色
      case FT_PIXEL_MODE_MONO:
        for (int j = 0; j < glyph_bitmap->width; ++j)
          buffer [j] = (src [j / 8] & (0x80 >> (j & 7))) ? 0xFFFFFFFF : 0x00000000;
        break;
      default:
        //异常情况，抛出异常
        break;
    }
    buffer += buf_width;
  }
}
```

读者应该知道字体在显示的时候都是点阵的格式，TTF格式字体只不过是保存了字体生成点阵的算法和基本数据。最终渲染的时候还是使用点阵来渲染的。所以我们看到FreeType库在设置的字体大小后，渲染出的结果还是点阵型的。而在渲染的时候贴图一般都不是单色的点阵贴图，所以必须做转化。CEGUI在这里转化为RGBA格式的真彩色格式，显然有些浪费内存，因为其实字符显示数据只有两色，在一个像素处显示还是不显示两种情况。为了节省内存，其实可以使用其他更小的格式比如R4G4B4A4。这种格式比RGBA小一半。

这一小节就介绍到这里，第8.2节主要介绍了各种字体以及字体文件。

8.3 字体管理器

在CEGUI中各种管理器都是单件，而且都是负责对应事物的创建，删除等操作。字体管理器也是一样，它保存了系统内所有的字体。它的数据成员很简单只有一个字体的映射。

```
typedef std::map<String, Font*, String::FastLessCompare> FontRegistry;
```

```
FontRegistry d_fonts;
```

从字体的名称映射到字体的具体实现类的指针。可能包括位图字体和TTF字体。

成员函数我们只介绍两个createFont和destroyFont，其他函数读者自己阅读。createFont函数被字体文件的处理类（Font_xmlHandler）调用。这个函数定义如下：

```
Font *FontManager::createFont (const String &type, const XMLAttributes& attributes)
{
   Font *temp;
   // 根据不同的字体类型创建不同的字体类实例
   if (type == FontTypeFreeType)
      temp = new FreeTypeFont (attributes);
   else if (type == FontTypePixmap)
      temp = new PixmapFont (attributes);
   // 错误的类型，抛出异常
   else
      throw FileIOException ("FontManager::createFont - The value for the Font:Type
            attribute '" + type + "' is unknown.");
      //检查字体是否存在
   String name = temp->getProperty ("Name");
   if (isFontPresent (name))
   {
      delete temp;
      throw AlreadyExistsException ("FontManager::createFont - A font named '" + name + "'
      already exists.");
   }
      //添加到字体映射中
   d_fonts [name] = temp;
   return temp;
}
```

创建过程非常简单，具体如何加载字体的数据都在两种字体的load中实现，这个函数会被字体的XML解析类自动调用。load函数又调用了updateFont函数来实现字体的更新。这些内容第8.2节有详细的介绍。createFont函数还有几个重载的函数，功能都类似，这里就不在介绍了。

第二个函数删除一个字体，destroyFont函数。

```
void FontManager::destroyFont(const String& name)
{
   FontRegistry::iterator pos = d_fonts.find(name);
      //如果找到了对应的字体
      if (pos != d_fonts.end())
      {
            String tmpName(name);
            //删除字体实现类
            delete pos->second;
            //从映射中移除
            d_fonts.erase(pos);
         //写入日志
            Logger::getSingleton().logEvent("Font '" + tmpName +"' has been destroyed.");
      }
}
```

这个函数同样有重载的函数。字体管理器就介绍到这里，如果读者需要处理一些资源一般的步骤是先找到资源管理器的单件对象，然后调用管理器对应的函数。只要掌握了这个基本规律，在CEGUI中实现代码就变的比较容易了。

8.4 CEGUI显示中文

终于到了大家最关心的问题了？CEGUI支不支持中文的显示呢？答案是肯定的，CEGUI原生支持中文的显示，但中文的输入是不支持的，这个需要读者自己来实现。本书第14章将会制作一个支持中文输入的控件。哪么在CEGUI中显示中文需要那些步骤呢？

步骤1，准备可以显示中文的字体。

步骤2，准备显示中文的字符串转换函数。

步骤3，显示中文了。

8.4.1 中文字体的创建

前文已经提及到显示中文需要修改字体文件。其实修改这个文件非常简单。读者可以到光盘的CEGUI例子\datafiles\fonts\目录下找到chinese.font的文件，这个文件里就定义了一个中文字体。下面是它的具体内容。

```
<?xml version="1.0" ?>
<Font Name="chinese" Filename="LiShu.ttf" Type="FreeType" Size="24" NativeHorzRes="800" NativeVertRes="600"
AutoScaled="true"/>
```

字体的名称是chinese，字体的数据文件是LiShu.ttf，这个文件是笔者从Window系统目录下拷贝过来的。在同一个目录下可以找到这个文件。字体大小笔者设为24，比较大，显
示的比较清楚。其实中文体只要这样简单的设置就可以了。注意设置完了以后CEGUI并不知道这个字体的存在，哪么如何加载这个字体呢？相信聪明的读者应该知道了，需要在模式文件中加字体的定义。但读者可能在所有的模式文件中都没有发现字体的定义，哪么在我们修改过的CEGUI字体的例子中是如何显示中文的呢？在那里我们在程序里动态的加载了字体资源，如何加载呢？只要获取字体管理器调用createFont函数和load函数不就可以了吗。在CEGUI的那个例子里面的确是这样做的，不信读者可以自己看源代码。不够CEGUI的官方例子并不支持中文，笔者做了简单的修改后可以支持中文的显示了。

8.4.2 中文字符显示

前面我们讲过CEGUI的字体使用Unicode字符映射到字符显示数据。在显示字符串的时候就可以描绘这个字符对应的显示数据，从而实现字符的显示。哪么我们的本地GBK数据如何转化为Unicode数据呢？String类提供了utf8的接口，我们可以将中文先转化为utf8格式的数据，然后传递给String类，这个类就可以自动的转化为Unicode数据了。这样显然比较麻烦，哪么有没有自动将GBK数据转化为String类的Unicode数据的方法呢？CEGUI的String类没有提供这样的方法，如过要实现就必须自己实现。还有既然CEGUI的String类保存的是Unicode数据，哪么我们可以给String类提供一个wchar_t的构造函数，也就是说直接给String类的就是Unicode数据，String类直接可以转化为utf32的字符串了。下面提供从GBK到utf8的转换函数。

```
utf8* mbcs_to_utf8(const char* pMbcs)
{
    static utf8 g_buf[1024] = {0};
    static  wchar_t g_Unicode[1024] = {0};
    memset(g_Unicode, 0, sizeof(g_Unicode));
    memset(g_buf, 0, sizeof(g_buf));
    //首先将GBK的数据转化为Unicode数据
    MultiByteToWideChar(936, 0, pMbcs, strlen(pMbcs), g_Unicode, 1024);
    //然后将Unicode数据转化为utf8数据
    WideCharToMultiByte(CP_UTF8, 0, g_Unicode, wcslen(g_Unicode),
    (char*)g_buf, 1024, 0, 0);
    //最后返回utf8数据
    return g_buf;
}
```

在显示中文之前调用这个函数将GBK的数据转化为utf8的数据，然后将这个数据赋值给String类，这样就可以显示中文了。具体操作可以类似下面。

```
showChinesWindow->setText (mbcs_to_utf8((const char*)"中文测试"));
```

其中showChinesWindow是一个窗口。字符串"中文测试"，是我们测试的中文GBK数据。

CEGUI窗口的设置字符串的函数使用的参数都是类String，哪么就会调用String类的utf8的构造函数，最终将GBK数据还是转化为了Unicode数据。读者可能会像哪么为什么不直接提供Unicode的构造函数，我们把中文转化为Unicode数据直接赋值给String类不也可以吗？的确可以我们可以提供这样的构造函数。

构造函数的定义如下：

```
String(const wchar_t* str)
{
    init();
    assign(str);
}
```

我们还需要实现assign函数，这个函数实现数据的转化。

```
String&   assign(const wchar_t* chars)
{
    //首先获取Unicode字符串的长度信息
    int chars_len = wcslen(chars);
    //申请内存
    grow(chars_len);
    //得到当前的数据指针
    utf32* pt = ptr();
    //遍历整个Unicode串，一个一个的赋值过去
    for (size_type i = 0; i < chars_len; ++i)
    {
        *pt++ = static_cast<utf32>(*chars++);
    }
    //设置字符的长度
    setlen(chars_len);
    return *this;
}
```

这个函数将Unicode数据直接赋值给utf32的字符串，应该是没有问题的，只要读者明白了String类使用四个字节保存的还是Unicode数据，就可以明白为什么这么做也可以，而且效率更高了。

最后一步，如何将这两个步骤整合起来从而可以显示中文。第一步定义好了字体，将这个字体应用到需要显示中文的窗口，第二步将中文转化为utf8格式后传递给窗口的setText函数，这样最终会转化为Unciode的中文字符，从而正确的显示中文。当然我们还提供了另外一种方法，就是扩充String类，提供Unicode字符串的构造函数。这样我们也可以在设置窗口文本的时候使用L开头的字符串（在编译的时候会转化为Unicode字符串）比如L"中文显示"，这段文字在编译后会转化为Unicode，这样String类的Unicode构造函数就可以工作了。

好，这一小节到这里就结束了。相信读者应该明白如何在CEGUI中显示中文了吧。如果不明白也不要气馁，在以后的章节我们会有详细的介绍。

8.5 本章小结

本章主要介绍了字体的具体实现，以及字体和编码，String了以及中文显示的方法等内容。字体的具体实现比较复杂，而且比较难以描述，读者尽可能理解，如果实在无法理解也不必太在意，也不是每个人都需要修改字体的。读者只要明白它的原理就可以了。

本章习题：

1.阅读getFormattedLineCount函数的具体实现，理解实现原理。

2.自己修改一个字体文件，并且要修改CEGUI的字体代码实现中文的显示。

有了这章的介绍相信读者应该可以自己实现中文的显示了吧。如果无法实现可以参考第14章的源代码。

第9章 CEGUI渲染插件

CEGUI是专门为游戏设计的界面库。官方提供了对当前流行的3D基础API的支持，包括OpenGL，Direct3D的两个版本。当然并不是说其他高级的渲染引擎就不可用使用CEGUI了，CEGUI官方还提供了一个开源的3D引擎Irrlicht的渲染插件。读者可能会感到疑惑，为什么没有提供目前最流行的3D开源引擎OGRE的渲染插件呢？其实OGRE官方已经实现了OGRE的CEGUI渲染器，所以CEGUI官方就没有提供。值得一提的是OGRE官方强烈推荐使用CEGUI作为界面库，他们自己实现的简单的UI在以后的版本（1.5版以后）将不会被支持了。本章我们介绍OpenGL的渲染插件和DirectX9的渲染插件，其他的插件读者可以试着阅读，其实也是大同小异。

9.1 CEGUI渲染接口

我们知道实现接口最常用的方法就是使用虚函数来定义接口的方法。CEGUI也是采用这种做法，它定义了两个虚基类。第一个是Texture类，这个类代表一个纹理对象，第二个是Renderer类，代表渲染对象。下面我们详细介绍这两个接口。

9.1.1 纹理接口

纹理其实存储的是一定格式的图像的数据。比如RGBA格式（真彩色）的图像数据。他们被用做三角形的贴图（当然OpenGL还支持其他的多边形的渲染比如四边形等）。更多纹理的介绍请参考3D相关的书籍。

下面介绍Texture接口的定义，首先看他的成员变量（啊，接口怎么会有成员变量呢？），只有唯一的一个成员变量，就是渲染接口的指针。

```
Renderer* d_owner;              //渲染接口的指针
```

有了这个指针就可以使用渲染接口提供的各种功能了。

CEGUI定义的纹理格式的定义，这两种格式显然对于文字贴图来说太浪费空间了，我们可以尝试加上一种PF_R4G4B4A4的格式，它占用两个字节，而且还包含Alpha通道。当然要添加这样一种格式，不是简单的在下面加上就可以实现的。它需要修改Renderer类的纹理创建和渲染才能实现。

```
enum PixelFormat
{
   //每个像素占用三位（BYTE）
   PF_RGB,
   //每个像素占四位（BYTE）
   PF_RGBA
};
```

重点是Texture提供的虚接口，下面我们一个一个介绍。它的接口函数可以分为两类，一类是获取纹理相关属性的函数，另一类是加载纹理的函数。首先介绍获取纹理属性的相关函数。属性主要是高和宽，其中只有getWidth和getHeight函数是虚函数，其他函数都是通过这两函数实现的内联函数

```
//宽度相关的函数
virtual     ushort  getWidth(void) const = 0;
//这个函数是为了兼容以前版本而提供的
virtual     ushort  getOriginalWidth(void) const { return getWidth(); }
//X方向的缩放因子
virtual     float   getXScale(void) const { return 1.0f / static_cast<float>(getOriginalWidth()); }
//高度相关的函数
virtual     ushort  getHeight(void) const = 0;
//这个函数也是为了兼容以前版本而提供的
virtual     ushort  getOriginalHeight(void) const { return getHeight(); }
//Y方向的缩放因子
virtual     float   getYScale(void) const { return 1.0f / static_cast<float>(getOriginalHeight()); }
```

这几个函数非常简单，这里就不过多的介绍了。

两个加载函数，一个从内存中读取数据，另一个从文件中加载。

```
//从文件中读取，需要指定文件名和资源组
virtual void     loadFromFile(const String& filename, const String& resourceGroup) = 0;
//从内存中读取，需要指定数据地址，宽度和高度，以及图像数据的格式
virtual void     loadFromMemory(const void* buffPtr, uint buffWidth, uint buffHeight, PixelFormat pixelFormat) = 0;
```

这两个函数是负责加载纹理的数据的。纹理接口就介绍到这里，下一小节介绍渲染接口。

9.1.2 渲染接口

渲染接口负责纹理接口的创建和销毁，最终要的就是负责CEGUI图像的渲染工作了。界面的渲染非常的简单，因为每个窗口都可以看做是由一些四边形构成的。为什么不是一个四边形呢？因为有的时候窗口是有许多部分构成的，我们前面也介绍过最多可以是9个部分构成的。而且他们都是正对着屏幕的，这种投影方式叫做正交投影。当然Direct3D也可以使用RHW格式的顶点来实现。OpenGL是直接支持四边形的渲染的（但CEGUI的官方的渲染插件还是拆分成两个三角形来渲染的），Direct3D则必须拆分成2个三角形来渲染。拆分的方法有两种，CEGUI提供了一个枚举来定义他们。如图9-1所示。

```
enum QuadSplitMode
{
    TopLeftToBottomRight,       //从左上到右下（a图）
    BottomLeftToTopRight        //从左下到右上（b图）
};
```

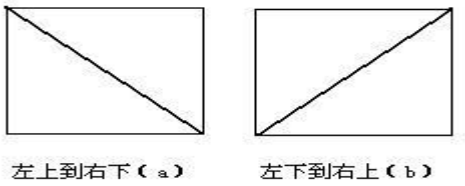

图9-1 四边形拆分

渲染接口派生自EventSet，这样它就具备了事件的处理和激发能力。它只定义了一个事件当显示尺寸发生改变的时候会激发这个事件。如下所示。

```
static const String EventDisplaySizeChanged;
```

此外这个接口还有以下成员变量。

```
static const float    GuiZInitialValue;          //每帧的初始Z值
static const float    GuiZElementStep;           //每个GUI元素的Z递进值
static const float    GuiZLayerStep;             //每个 GUI层的Z递进值
float d_current_z;                               //保存当前的Z坐标值
ResourceProvider* d_resourceProvider;    //资源提供接口的指针，用来加载纹理
String d_identifierString;               //保存这个渲染接口信息的字符串
```

读者应该还记得，我们曾经说过CEGUI中Z值是没有用的，真正决定窗口的层叠顺序的是窗口的渲染数组和窗口的天生的渲染先后关系（先渲染父窗口然后渲染子窗口）。好了下面介绍渲染接口的各种接口函数。它大致可以分为渲染模块的信息提供函数，渲染相关的函数，纹理管理函数以及其他函数。

首先，介绍各种信息的获取函数。他们不全是虚接口函数，有一部分是内敛的简单函数。

```
//获取这个渲染插件的身份信息
const String& getIdentifierString() const;
//获取层的Z值
float getZLayer(uint layer) const    {return d_current_z - ((float)layer * GuiZLayerStep);}
//获取当前Z值
float getCurrentZ(void) const              {return d_current_z;}
//Z值步进
void advanceZValue(void)                   {d_current_z -= GuiZElementStep;}
```

```
//重设Z值
void resetZValue(void)                    {d_current_z = GuiZInitialValue;}
//获取竖直的DPI（每英寸的点数）
virtual     uint  getVertScreenDPI(void) const      = 0;
//获取水平的DPI
virtual     uint  getHorzScreenDPI(void) const      = 0;
//获取纹理的最大尺寸（正方形的所以只返回了一个值）
virtual     uint  getMaxTextureSize(void) const     = 0;
//渲染的区域大小（可能是游戏程序的客户区域，或者整个屏幕，全屏的时候）
virtual Rect     getRect(void) const        = 0;
//当前渲染区域的大小
virtual Size     getSize(void) const        = 0;
//当前渲染区域的高度
virtual float     getHeight(void) const     = 0;
//当前渲染区域的宽度
virtual float     getWidth(void) const      = 0;
```

这些函数都是获取渲染相关的信息的函数。

其次，渲染相关的函数。既然是渲染接口当然他们是最重要的了。

```
//添加四边形，如果允许渲染缓冲就添加到渲染缓冲中，否则直接渲染
irtual       void addQuad(const Rect& dest_rect, float z, const Texture* tex, const Rect& texture_rect, const ColourRect& colours,
QuadSplitMode quad_split_mode) = 0;
//开始渲染，如果允许渲染缓冲则开始真正的渲染，否则没什么可以做的
virtual     void doRender(void) = 0;
//清空渲染缓冲
virtual     void clearRenderList(void) = 0;
//设置是否开启渲染缓冲
virtual void     setQueueingEnabled(bool setting) = 0;
```

这几个接口都是虚函数，他们是渲染模块的最重要的接口之一。

再次，介绍纹理相关的接口。纹理相关的函数主要有创建和删除一个纹理。

```
//创建一个空的纹理对象
virtual     Texture*  createTexture(void) = 0;
//从文件中加载一张图片作为纹理
virtual     Texture*  createTexture(const String& filename, const String& resourceGroup) = 0;
//创建一个指定大小的纹理对象
virtual     Texture*  createTexture(float size) = 0;
//删除一个纹理对象
virtual     void        destroyTexture(Texture* texture) = 0;
```

纹理的管理，简单而高效。

最后，介绍一个非常简单的函数，就是创建资源提供接口的函数。这个函数会创建CEGUI提供的默认资源管理器。

```
ResourceProvider* Renderer::createResourceProvider(void)
{
   d_resourceProvider = new DefaultResourceProvider();
   return d_resourceProvider;
}
```

以及渲染接口的构造函数，参数是资源提供接口的指针和身份字符串。如果资源提供指针为空，当需要加载资源的时候就会调用上面的函数创建一个CEGUI提供的默认资源管理器。

```
Renderer::Renderer(void)
   : d_resourceProvider(0),
     d_identifierString("Unknown renderer (vendor did not set the ID string!)")
```

```
{
    //添加标准的事件到事件集
    addEvent(EventDisplaySizeChanged);
    resetZValue();
}
```

这节就介绍到这里，9.2和9.3两个小节将详细介绍两个渲染接口的实现。

9.2 OpenGL渲染插件

本书的例子都是使用OpenGL的渲染插件来实现的。OpenGL在初始化，还是它的操作都是非常简单的，用他来做例子程序非常的容易而且好理解。DirectX的初始化比较复杂，而且还会出现设备丢失。每一个版本的兼容型也比较差（从CEGUI提供了两个版本的DirectX的渲染插件就可以看出）。它的成功和OpenGL的止步不前和内讧是有关系的，当然微软的推广能力也是不容小视的。这里都是个人观点，这个问题也没有什么好争论的，仁者见仁智者见智，但从API的设计上，笔者认为还是OpenGL设计的更加合理，更加有条理。

CEGUI的渲染插件由两部分组成，一部分是纹理的接口，另一部分是渲染接口。本章就从这两个方面来介绍，每种插件都分这两个部分。OpenGL渲染插件工程是OpenGLGUIRenderer，在CEGUI的解决方案里可以找到。

9.2.1 OpenGL纹理

OpenGL的纹理是它自己管理的，它提供了一个类似句柄一样的整型数，用它来操作纹理对象。这样客户程序完全不需要知道纹理的具体实现，只需要保存这个句柄，然后调用可以操作这些句柄的函数就可以了。这样的程序设计方法其实非常常见，比如Window的内核对象，GDI对象等等。都是给外部程序一个代表这个对象的数，和一些可以操作这些对象的函数。

我们先介绍OpenGL纹理类（OpenGLTexture）的成员变量。

```
//这个变量就是我们说的OpenGL提供的纹理对象的句柄
GLuint          d_ogltexture;
//纹理的高度
ushort          d_width;
//纹理的宽度
ushort          d_height;
//保存图像的数据以便以后还原纹理
uint8*          d_grabBuffer;
//缓存的图像数据的高度
ushort d_orgWidth;
//缓存的图像数据的宽度
ushort d_orgHeight;
//缓存的图像的X方向缩放因子
float d_xScale;
//缓存的图像的Y方向缩放因子
float d_yScale;
```

这些变量的含义都非常简单，这里就不在介绍了。下面介绍几个重要的成员变量。第一个是构造函数，构造和析构函数被声明为私有函数，一般的函数无创建纹理对象，只有OpenGLRenderer类的创建和删除纹理函数可以创建和删除纹理对象（通过友元函数实现）。

```
OpenGLTexture::OpenGLTexture(Renderer* owner) :
    Texture(owner),
    d_grabBuffer(0),
  d_xScale(1.0f),
  d_yScale(1.0f)
{
    // 获取一个空的OpenGL纹理对象，并保存到纹理句柄中。
    glGenTextures(1, &d_ogltexture);
    // 先绑定这个纹理，然后设置纹理对应的参数
```

```
    glBindTexture(GL_TEXTURE_2D, d_ogltexture);
    //设置放大的时候使用线性采样函数
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    //缩小的时候也使用线性采样函数
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    //纹理卷曲的时候在边缘被截断（GL_CLAMP_TO_EDGE定义为0x812F）
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, 0x812F);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, 0x812F);
    //设置纹理环境，设置为顶点的颜色和纹理的颜色相乘作为结果颜色
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
}
```

阅读9.2和9.3章需要读者对OpenGL和Direct3D比较熟悉，否则读者可能无法读懂。建议读者阅读一些3D相关的书籍。

第二个函数是从文件加载图片作为纹理对象的函数。

```
void OpenGLTexture::loadFromFile(const String& filename, const String& resourceGroup)
{
    //获取渲染接口的指针
  OpenGLRenderer* renderer =  static_cast<OpenGLRenderer*>(getRenderer());
    glBindTexture(GL_TEXTURE_2D, d_ogltexture);
    //通过资源提供器加载图片资源
    RawDataContainer texFile;
    System::getSingleton().getResourceProvider()->loadRawDataContainer(filename, texFile,
resourceGroup);
    //图像解码接口加载图片并且调用从内存加载的函数实现加载
    Texture* res = renderer->getImageCodec().load(texFile, this);
    //释放内存图片数据
    System::getSingleton().getResourceProvider()->unloadRawDataContainer(texFile);
    if (res == 0)
  {
    // 发生错误抛出异常
        throw RendererException("OpenGLTexture::loadFromFile - " +
                    renderer->getImageCodec().getIdentifierString() +
                    " failed to load image '" + filename + "'.");
  }
}
```

通过这个函数，读者可以理解图像解码器的作用了。

下一个函数是通过内存数据生成纹理对象的函数。

```
void OpenGLTexture::loadFromMemory(const void* buffPtr, uint buffWidth, uint buffHeight, Texture::PixelFormat pixFormat)
{
    GLint comps;
    GLenum format;
    switch (pixFormat)
    {
      case PF_RGB:
        comps = 3;
        format = GL_RGB;
        break;
      case PF_RGBA:
        comps = 4;
        format = GL_RGBA;
        break;
```

```
    };
  //决定纹理的尺寸而且给纹理赋初值，如果先前有纹理则先删除这个纹理
  setOGLTextureSize(ceguimax(buffWidth, buffHeight));
  //保存原始的高度和宽度
  d_orgWidth = buffWidth;
  d_orgHeight = buffHeight;
  //更新缩放因子
  updateCachedScaleValues();
  //设置图片数据到纹理对象，先绑定然后设置数据
  glBindTexture(GL_TEXTURE_2D, d_ogltexture);
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, buffWidth, buffHeight, format,
    GL_UNSIGNED_BYTE, buffPtr);
}
```

这里不得不介绍setOGLTextureSize函数了。这个函数获取合适的纹理大小，并且初始化纹理，如果存在旧的纹理则删除之。

```
void OpenGLTexture::setOGLTextureSize(uint size)
{
    //这个函数获取最接近size的2的幂的纹理大小（当然不能小于size）
  size = getSizeNextPOT(size);
    //确保我们获取的纹理大小小于OpenGL支持的最大纹理
    int maxSize;
    glGetIntegerv(GL_MAX_TEXTURE_SIZE, (GLint*)&maxSize);
    if (size > (uint)maxSize)
    {
        throw RendererException("OpenGLTexture::setOGLTextureSize - size too big");
    }
    //申请一块内存，目的是作为缓冲
    uchar* buff = new uchar[size * size * 4];
    //加载空的数据到纹理对象，如果先前这个纹理对象有纹理数据则先前的被销毁
    glBindTexture(GL_TEXTURE_2D, d_ogltexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, size, size, 0, GL_RGBA ,
GL_UNSIGNED_BYTE, buff);

    //删除缓冲，这个缓冲可以理解是真正图片数据的站位缓冲
    delete[] buff;
    //设置纹理内部宽度和高度信息，原始高度和宽度会在loadFromMemory函数中重置
    d_orgWidth = d_orgHeight = d_height = d_width = static_cast<ushort>(size);
    //更新内部因子
  updateCachedScaleValues();
}
```

纹理的创建主要的函数就这些，下面介绍纹理的数据获取和纹理的重置。

纹理的数据获取函数。

```
void OpenGLTexture::grabTexture(void)
{
  //绑定纹理对象，在操作纹理是首先就是要绑定纹理
  glBindTexture(GL_TEXTURE_2D, d_ogltexture);
  //申请接收数据的内存
  d_grabBuffer = new uint8[4*d_width*d_height];
    //获取数据到内存
  glGetTexImage(GL_TEXTURE_2D,0,GL_RGBA,GL_UNSIGNED_BYTE,d_grabBuffer);
```

```
  //删除这个纹理对象
  glDeleteTextures(1, &d_ogltexture);
}
```

重置纹理数据的函数。

```
void OpenGLTexture::restoreTexture(void)
{
  //重新创建纹理对象
    glGenTextures(1, &d_ogltexture);
    //设置纹理的各种状态和构造函数类似
    glBindTexture(GL_TEXTURE_2D, d_ogltexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, 0x812F);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, 0x812F);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    //设置纹理的真实数据
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, d_width, d_height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, d_grabBuffer);
  //释放保存纹理的内存缓冲数据
  delete [] d_grabBuffer;
  d_grabBuffer = 0;
}
```

这两个函数会被渲染接口的同名函数调用。当用户希望全部重新加载纹理对象的时候可以使用渲染接口提供的同名函数。但这几个函数不常用，很少有全部重置纹理的需求。

好了，OpenGL的纹理类就介绍到这里。不熟悉OpenGL的读者可能对这里边的操作无法理解，没关系一般来说读者不需要修改这章的内容。读者只要理解他们就可以了。

9.2.2 OpenGL渲染模块

OpenGL的渲染接口比较庞大，我们不可能介绍全部的函数，我们选择和渲染最相关的，渲染插件必须要实现的虚接口函数，作为介绍的重点函数。其他函数读者自己阅读。

首先，我们要介绍渲染用到的几个结构体。QuadInfo代表一个四边形，它提供了渲染需要的基本信息。

```
struct QuadInfo
{
 GLuint        texid;              //对应OpenGL纹理对象的句柄
 Rect          position;           //描绘的位置，屏幕坐标系下
 float         z;                  //Z值
 Rect          texPosition;        //纹理坐标
 uint32        topLeftCol;         //左上角的顶点颜色
 uint32        topRightCol;        //右上角的顶点颜色
 uint32        bottomLeftCol;      //左下角的顶点颜色
 uint32        bottomRightCol;     //右下角的顶点颜色
 QuadSplitMode   splitMode;        //顶点的拆分模式，有两种
 //两个四边形的比较函数，决定那个先被渲染，其实没有用到
 bool operator<(const QuadInfo& other) const
 {
    return z > other.z;
 }
};
```

MyQuad结构体，渲染的时候会用到的辅助结构体，交叉数组的基本结构。

```
struct MyQuad
{
  float tex[2];                    //纹理坐标
  uint32 color;                    //颜色
  float vertex[3];                      //顶点的三个值
};
```

下面介绍渲染插件的成员变量。

```
//定义类型
typedef std::multiset<QuadInfo>          QuadList;
//定义保存所有的渲染矩形的变量，也就是我们曾经讲过的第一层渲染缓冲了
QuadList d_quadlist;
//显示区域
Rect       d_display_area;
//定义交叉数组，用于大批量的传送顶点渲染数据到显卡的数据结构
MyQuad  myBuff[OGLRENDERER_VBUFF_CAPACITY];
//当前的渲染模式，是缓冲模式还是直接渲染的模式
bool       d_queueing;
//当前渲染使用的纹理对象
uint       d_currTexture;
//交叉数组的游标，指向下一个可以使用的MyQuad结构
int        d_bufferPos;
//是否渲染经过排序了，其实没有用到
bool       d_sorted;
//所有渲染插件创建的纹理的集合
std::list<OpenGLTexture*>   d_texturelist;
//保存OpenGL显卡支持的最大纹理尺寸
GLint      d_maxTextureSize;
//图像解码插件的指针，从文件加载纹理的时候会用到
ImageCodec* d_imageCodec;
//图像解码模块（一般是动态库）的指针
DynamicModule* d_imageCodecModule;
//默认图像解码模块的名称
static String d_defaultImageCodecName;
```

图像插件会使用到图像解码器来加载图像数据，图像解码器负责从文件中加载图片数据并且转化为可以使用的各种格式。图片为了压缩大小会编码，比如有名的JEPG编码。这些数据是要解码后才能被程序使用的，所以需要用到图像解码器。

下面介绍OpenGL渲染插件的渲染相关的函数。我们已经说过CEGUI有两层渲染缓冲一层在渲染插件中支持，另一层在每个窗口中支持（RenderCache）。后者前文已经介绍过了，这里就要介绍渲染插件提供的渲染缓冲和渲染函数了。

对于第一层缓冲来说它的效果并不是很好，因为如果第二层缓冲中有一个发生变化则第一层缓冲就失效了。当用户没有任何界面操作而且界面也没有自动更新的时候，它才有作用。所以它的作用不是非常明显。因此CEGUI提供了不使用第一层缓冲直接执行渲染的函数接口。这个接口就是setQueueingEnabled，它虽然不负责渲染工作，但设置它的值会影响渲染插件是直接渲染呢，还是缓冲起来等到doRender函数调用时在渲染。下渲染缓冲添加渲染矩形的函数是addQuad，它根据是否缓冲调用不同的函数。具体实现如下所示。

```
void OpenGLRenderer::addQuad(const Rect& dest_rect, float z, const Texture* tex, const Rect& texture_rect, const ColourRect& colours, QuadSplitMode quad_split_mode)
{
    //如果不是缓冲状态的话直接渲染
    if (!d_queueing)
```

```
    {
        renderQuadDirect(dest_rect, z, tex, texture_rect, colours, quad_split_mode);
    }
    //否则保存渲染矩形的各种渲染参数到渲染缓冲中
    else
    {
        QuadInfo quad;
        quad.position       = dest_rect;
        quad.position.d_bottom  = d_display_area.d_bottom - dest_rect.d_bottom;
        quad.position.d_top     = d_display_area.d_bottom - dest_rect.d_top;
        quad.z              = z;
        quad.texid          = ((OpenGLTexture*)tex)->getOGLTexid();
        quad.texPosition    = texture_rect;
        quad.topLeftCol     = colourToOGL(colours.d_top_left);
        quad.topRightCol    = colourToOGL(colours.d_top_right);
        quad.bottomLeftCol  = colourToOGL(colours.d_bottom_left);
        quad.bottomRightCol = colourToOGL(colours.d_bottom_right);
    //矩形拆分成三角型的拆分模式
    quad.splitMode = quad_split_mode;
        //添加到队列中
        d_quadlist.insert(quad);
    }
}
```

添加矩形的函数非常简单，就是根据不同的状态分别做了处理。重要的函数当然是渲染函数了。由于renderQuadDirect函数和doRender函数的工作非常类似，我们这里就只介绍
doRender函数，另外一个函数读者自己阅读。这个函数使用了OpenGL提供的交叉数组来提供渲染的效率（因为它可以最大限度的减少函数的调用开销，比较类似DirectX中的FVF格式的渲染方式）。

```
void OpenGLRenderer::doRender(void)
{
    d_currTexture = 0;
    //初始化OpenGL的渲染状态，主要是设置OpenGL正交投影，禁用深度等
    initPerFrameStates();
    //这一步指定交叉数组的内存位置和数组中各个成员的排列信息
    glInterleavedArrays(GL_T2F_C4UB_V3F , 0, myBuff);
    //遍历整个缓冲，如果缓冲为空显然什么都不做
    for (QuadList::iterator i = d_quadlist.begin(); i != d_quadlist.end(); ++i)
    {
        const QuadInfo& quad = (*i);
        //如果缓冲中使用不同的纹理则不能同时渲染更多的三角形了，必须立即渲染
        if(d_currTexture != quad.texid)
        {
            renderVBuffer();
            glBindTexture(GL_TEXTURE_2D, quad.texid);
            d_currTexture = quad.texid;
        }
        //下面是根据拆分模式的不同将四边形拆分为两个三角形的过程
        //第一个顶点的拆分
        myBuff[d_bufferPos].vertex[0]    = quad.position.d_left;
        myBuff[d_bufferPos].vertex[1]    = quad.position.d_top;
        myBuff[d_bufferPos].vertex[2]    = quad.z;
        myBuff[d_bufferPos].color        = quad.topLeftCol;
```

```
            myBuff[d_bufferPos].tex[0]      = quad.texPosition.d_left;
            myBuff[d_bufferPos].tex[1]      = quad.texPosition.d_top;
            ++d_bufferPos;
            //第二个顶点
            myBuff[d_bufferPos].vertex[0]   = quad.position.d_left;
            myBuff[d_bufferPos].vertex[1]   = quad.position.d_bottom;
            myBuff[d_bufferPos].vertex[2]   = quad.z;
            myBuff[d_bufferPos].color       = quad.bottomLeftCol;
            myBuff[d_bufferPos].tex[0]      = quad.texPosition.d_left;
            myBuff[d_bufferPos].tex[1]      = quad.texPosition.d_bottom;
            ++d_bufferPos;
        //根据不同的拆分模式执行不同的拆分方法
        if (quad.splitMode == TopLeftToBottomRight)
        {
            myBuff[d_bufferPos].vertex[0] = quad.position.d_right;
            myBuff[d_bufferPos].vertex[1] = quad.position.d_bottom;
            myBuff[d_bufferPos].vertex[2] = quad.z;
            myBuff[d_bufferPos].color     = quad.bottomRightCol;
            myBuff[d_bufferPos].tex[0]    = quad.texPosition.d_right;
            myBuff[d_bufferPos].tex[1]    = quad.texPosition.d_bottom;
        }
        else
        {
            myBuff[d_bufferPos].vertex[0] = quad.position.d_right;
            myBuff[d_bufferPos].vertex[1] = quad.position.d_top;
            myBuff[d_bufferPos].vertex[2] = quad.z;
            myBuff[d_bufferPos].color     = quad.topRightCol;
            myBuff[d_bufferPos].tex[0]    = quad.texPosition.d_right;
            myBuff[d_bufferPos].tex[1]    = quad.texPosition.d_top;
        }
            ++d_bufferPos;
            //第四个顶点
            myBuff[d_bufferPos].vertex[0]   = quad.position.d_right;
            myBuff[d_bufferPos].vertex[1]   = quad.position.d_top;
            myBuff[d_bufferPos].vertex[2]   = quad.z;
            myBuff[d_bufferPos].color       = quad.topRightCol;
            myBuff[d_bufferPos].tex[0]      = quad.texPosition.d_right;
            myBuff[d_bufferPos].tex[1]      = quad.texPosition.d_top;
            ++d_bufferPos;
            //第五个顶点
        if (quad.splitMode == TopLeftToBottomRight)
        {
            myBuff[d_bufferPos].vertex[0] = quad.position.d_left;
            myBuff[d_bufferPos].vertex[1] = quad.position.d_top;
            myBuff[d_bufferPos].vertex[2] = quad.z;
            myBuff[d_bufferPos].color     = quad.topLeftCol;
            myBuff[d_bufferPos].tex[0]    = quad.texPosition.d_left;
            myBuff[d_bufferPos].tex[1]    = quad.texPosition.d_top;
        }
        else
        {
            myBuff[d_bufferPos].vertex[0] = quad.position.d_left;
```

```
        myBuff[d_bufferPos].vertex[1] = quad.position.d_bottom;
        myBuff[d_bufferPos].vertex[2] = quad.z;
        myBuff[d_bufferPos].color     = quad.bottomLeftCol;
        myBuff[d_bufferPos].tex[0]    = quad.texPosition.d_left;
        myBuff[d_bufferPos].tex[1]    = quad.texPosition.d_bottom;
    }
        ++d_bufferPos;
        //第六个顶点
        myBuff[d_bufferPos].vertex[0]  = quad.position.d_right;
        myBuff[d_bufferPos].vertex[1]  = quad.position.d_bottom;
        myBuff[d_bufferPos].vertex[2]  = quad.z;
        myBuff[d_bufferPos].color      = quad.bottomRightCol;
        myBuff[d_bufferPos].tex[0]     = quad.texPosition.d_right;
        myBuff[d_bufferPos].tex[1]     = quad.texPosition.d_bottom;
        ++d_bufferPos;
        //如果当前交叉数组的已经填满了则必须立即渲染
        if(d_bufferPos > (VERTEXBUFFER_CAPACITY - VERTEX_PER_QUAD))
        {
            renderVBuffer();
        }
    }
    //最后的渲染，如果交叉数组未被填满，则会退出，这里定义渲染函数确保全部渲染
    renderVBuffer();
    //退出OpenGL的界面渲染模式
    exitPerFrameStates();
}
```

解释OpenGL交叉数组的格式定义GL_T2F_C4UB_V3F，这个定义告诉OpenGL使用纹理是两个纹理坐标，数据格式是浮点型（T2F），颜色是四个颜色（RGBA），数据格式是无符号字节，顶点是三个顶点数据（XYZ），格式也是浮点型。读者应该知道它和MyQuad机构体的对应关系了吧。顶点的拆分读者可以参考我们图9-1的介绍。下面介绍初始化状态的函数initPerFrameStates，从这个函数中读者可以明白为什么Z值是没有用到的。

```
void OpenGLRenderer::initPerFrameStates(void)
{
    //保存当前OpenGL的所有属性状态
    glPushClientAttrib(GL_CLIENT_ALL_ATTRIB_BITS);
    glPushAttrib(GL_ALL_ATTRIB_BITS);

    //多边形的填充模式是只填充前面（默认顶点构成多边形的逆时针方向为正面）
    glPolygonMode(GL_FRONT, GL_FILL);
    //设置投影矩阵为正交投影，这里没有深度信息的指定
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0.0, d_display_area.d_right, 0.0, d_display_area.d_bottom);
    //设置模型变换矩阵，设成简单的单位阵
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    //禁用一些OpenGL状态，包括光照，深度测试，雾，纹理的坐标的自动生成
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_FOG);
```

```
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    glDisable(GL_TEXTURE_GEN_R);
        //设置正面为逆时针方向，与默认值相同
        glFrontFace(GL_CCW);
        //裁剪背面
        glCullFace(GL_BACK);
        //打开OpenGL的裁剪面操作
        glEnable(GL_CULL_FACE);
        //打开混合操作，支持alpha效果
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        //设置纹理环境我相乘，并且打开2D纹理
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
        glEnable(GL_TEXTURE_2D);
}
```

为什么说虽然顶点中指定了Z值但，它不能起到排序的作用呢？因为这里的深度测试是关闭的，而且指定的正交投影矩阵Z方向没有深度（指定了最大最小Z值都为0）。哪么窗口的层叠顺序是如何控制的呢？在Window类里有d_drawList的窗口指针数组变量，它的顺序保存的窗口的渲染顺序，哪么先渲染的窗口就可能被后渲染的窗口覆盖了。而CEGUI保证父窗口一定比子窗口先渲染。有了这中保证就可以决定窗口的层叠顺序了。

从上面的实现可以看出如果使用同一张纹理的所有的多边形可以排在一起就可以最大限度的提高效率，因为它不涉及到纹理的切换。但他会破坏层叠顺序，所以不能这么做。虽然CEGUI提供了这种排序的接口。

OpenGL的渲染插件还有一些其他的函数比如创建和销毁纹理的函数，图像解码器相关的函数等。由于篇幅所限这里就不在介绍了。读者可以认真阅读他们，虽然他们不是最重要的。

9.3 DirectX9渲染插件

渲染插件的各种实现其实都是大同小异，读者如果理解的OpenGL的渲染插件，哪么DirectX9的渲染插件也基本上理解了。不过读者必须具备D3D的基本知识，否则不可能看的懂这里的实现。本书就简单的介绍d3d的插件。

9.3.1 DirectX9纹理

纹理相关的成员这里不在介绍，只简单的介绍纹理的创建函数。

```
void DirectX9Texture::loadFromMemory(const void* buffPtr, uint buffWidth, uint buffHeight, Texture::PixelFormat pixFormat)
{
    using namespace std;
    freeD3DTexture();
    uint tex_size = ceguimax(buffWidth, buffHeight);
    //转换CEGUI的纹理格式到D3D的纹理格式
    D3DFORMAT pixfmt;
    switch (pixFormat)
    {
    case PF_RGB:
        pixfmt = D3DFMT_R8G8B8;
        break;
    case PF_RGBA:
        pixfmt = D3DFMT_A8B8G8R8;
        break;
    default:
        throw RendererException("Failed to load texture from memory: Invalid pixelformat.");
    }
    //插件D3D的纹理对象
```

```
    HRESULT hr = D3DXCreateTexture(((DirectX9Renderer*)getRenderer())->getDevice(),
tex_size, tex_size, 1, 0, pixfmt, D3DPOOL_MANAGED, &d_d3dtexture);
    //如果失败抛出异常
    if (FAILED(hr))
    {
        throw RendererException("Failed to create D3D Texture failed.");
    }
    else
    {
    d_orgWidth = buffWidth;
    d_orgHeight = buffHeight;
    obtainActualTextureSize();
        //锁住D3D纹理，以便将图像数据写入
        D3DLOCKED_RECT  rect;
        hr = d_d3dtexture->LockRect(0, &rect, 0, 0);
        //如果失败则抛出异常（省略），释放创建的纹理对象
        if (FAILED(hr))
        {
            d_d3dtexture->Release();
            d_d3dtexture = 0;
        }
        else
        {
            //拷贝纹理数据到我们锁住的纹理
            ulong* dst = (ulong*)rect.pBits;
            ulong* src = (ulong*)buffPtr;
      if (pixFormat == PF_RGBA)
      {
        for (uint i = 0; i < buffHeight; ++i)
            {
                for (uint j = 0; j < buffWidth; ++j)
                {
                  //在Window上一般都是small endian的机器这不用关心
                    uchar r = (uchar)(src[j] & 0xFF);
                    uchar g = (uchar)((src[j] >> 8) & 0xFF);
                    uchar b = (uchar)((src[j] >> 16)  & 0xFF);
                    uchar a = (uchar)((src[j] >> 24) & 0xFF);
                    dst[j] = D3DCOLOR_ARGB(a, r, g, b);
                }
                dst += rect.Pitch / sizeof(ulong);
                src += buffWidth;
            }
      }
      else
      {
        for (uint i = 0; i < buffHeight; ++i)
            {
                for (uint j = 0; j < buffWidth; ++j)
                {
                  dst[j] = src[j];
                }
                dst += rect.Pitch / sizeof(ulong);
```

```
                        src += buffWidth;
                    }
                }
                d_d3dtexture->UnlockRect(0);
            }
        }
}
```

这个函数创建纹理并且为纹理填充数据，这些操作都是纹理的基本操作，在任何介绍D3D的书籍上都可以找到。创建纹理还有一个从文件中创建的函数，它直接使用D3DX提供的辅助函数，直接从文件中加载并且返回创建的纹理对象。

9.3.2 DirectX9渲染模块

渲染模块我们也只介绍两个函数，一个是设置状态的函数，另一个是直接渲染的函数。

首先看设置渲染状态的函数，其实3D的渲染流程确定了，设置状态在各种3D基本API的步骤也是类似，设置的值也是类似的。

```
void DirectX9Renderer::initPerFrameStates(void)
{
    // 设置顶点流并且设置FVF的格式
    d_device->SetStreamSource(0, d_buffer, 0, sizeof(QuadVertex));
    d_device->SetFVF(VERTEX_FVF);
    //禁用Shader
    d_device->SetVertexShader( 0 );
    d_device->SetPixelShader( 0 );
    //设置设备的渲染状态，参数和值和OpenGL的设置非常类似
    d_device->SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE);
    d_device->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
    d_device->SetRenderState(D3DRS_ALPHATESTENABLE, FALSE);
    d_device->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
    d_device->SetRenderState(D3DRS_FOGENABLE, FALSE);
  d_device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    //设置纹理的参数
    d_device->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
    d_device->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
    //设置顶点最终的颜色是纹理颜色乘以顶点的漫反射颜色
    d_device->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
    d_device->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
    d_device->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
    //设置Alpha值的操作方法
    d_device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
    d_device->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);
    d_device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
    //设置纹理放大和缩小的过滤函数
    d_device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
    d_device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
  //禁用第二层的颜色操作
  d_device->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_DISABLE);
    //设置Alpha值的混合功能（最终效果就是根据Alpha值决定透明度）
    d_device->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
    d_device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    d_device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
}
```

读者可以发现，这里设置的状态和OpenGL设置的状态大致是一样的。就是没有设置投影矩阵，哪么可以使用D3D中的特殊格式的自定义顶点（FVF），我们看它的VERTEX_FVF的定义。它定义为下面的组合。

```
D3DFVF_XYZRHW|D3DFVF_DIFFUSE|D3DFVF_TEX1
```

D3DFVF_XYZRHW告诉D3D这里的顶点已经经过了模型投影变换了，可以直接对应屏幕上的位置了，D3DFVF_DIFFUSE告诉D3D这个格式中包含漫反射颜色，D3DFVF_TEX1告诉D3D这个格式中包含一对纹理坐标。这个格式定义和OpenGL的交叉数组的格式定义非常类似，所不同的就是这里使用D3DFVF_XYZRHW而不是正交投影来描绘2D多边形。

下面介绍第二个函数，renderQuadDirect函数，这个函数直接渲染不缓冲。我们这里将省略四边形分割成两个三角型的过程。

```
void DirectX9Renderer::renderQuadDirect(const Rect& dest_rect, float z, const Texture* tex, const Rect& texture_rect, const ColourRect& colours, QuadSplitMode quad_split_mode)
{
    //做合适的坐标变换
    Rect final_rect(dest_rect);
    final_rect.offset(Point(-0.5f, -0.5f));
    QuadVertex*  buffmem;
    //初始化渲染状态
    initPerFrameStates();
    //设置纹理
    d_device->SetTexture(0, ((DirectX9Texture*)tex)->getD3DTexture());
    //锁定缓冲区，写入需要渲染的数据
    if (SUCCEEDED(d_buffer->Lock(0, VERTEX_PER_QUAD * sizeof(QuadVertex),
    (void**)&buffmem, D3DLOCK_DISCARD)))
    {
        //这里省略分割四边形为两个三角型的过程

        //解除缓冲区的锁定
        d_buffer->Unlock();
        d_bufferPos = VERTEX_PER_QUAD;
        //渲染整个缓冲
        renderVBuffer();
    }
}
```

这个函数和OpenGL的也非常类似，为了节省篇幅我们这里省略了四边形拆分的代码。我们介绍渲染函数。

```
void DirectX9Renderer::renderVBuffer(void)
{
    //如果没有需要渲染的数据则返回
    if (d_bufferPos == 0)
    {
        return;
    }
    //渲染所有缓冲中的三角形
    d_device->DrawPrimitive(D3DPT_TRIANGLELIST, 0, (d_bufferPos /
    VERTEX_PER_TRIANGLE));
    //重置渲染缓冲，方便以后的使用
    d_bufferPos = 0;
}
```

我们看到渲染函数非常简单，只是简单的调用了D3D提供的描绘函数。

关于D3D渲染接口就介绍到这里，如果读者不理解其中相关D3D的函数，请参考任意一本介绍D3D的书籍。

9.4 本章小结

本章主要介绍了CEGUI提供的官方的两个渲染插件，以及CEGUI的渲染接口。这一章的内容需要读者懂一些OpenGL和D3D的知识。设置这一章的目的一是为了读者理解第一层缓冲是如何实现的，二是希望读者理解CEGUI中渲染插件的工作原理。读者应该还记得在System类和Imageset类中有对渲染插件doRender和addQuad函数的调用。同时读者也理解了图像解码接口在CEGUI中的应用。这一章的具体习题大都是阅读和理解的，没有需要读者实现的。如果读者有兴趣可以尝试这实现一个CEGUI的渲染插件，比如读者实现一个DirectX7的渲染插件，虽然这个版本的DirectX已经不常用了。或者实现一个OGRE的插件（虽然OGRE官方已经实现了一版）。

1.阅读本书没有介绍的OpenGL和DirectX9的渲染插件的具体实现。

2.尝试自己实现一个CEGUI的渲染插件。

第10章 渲染窗口

本章的内容和第7章对应，第7章介绍了CEGUI官方实现的一些控件这里简单的介绍他们对应的渲染窗口类是如何实现的。在详细介绍渲染窗口之前我们先看Scheme中的窗口外部类型到窗口内部类型，窗口渲染窗口和窗口外观定义的映射。

```
<FalagardMapping WindowType="TaharezLook/Button"  TargetType="CEGUI/PushButton"
Renderer="Falagard/Button"      LookNFeel="TaharezLook/Button" />
<FalagardMapping WindowType="TaharezLook/Checkbox"   TargetType="CEGUI/Checkbox"   Renderer="Falagard/ToggleButton"
LookNFeel="TaharezLook/Checkbox" />
<FalagardMapping WindowType="TaharezLook/ImageButton" TargetType="CEGUI/PushButton"  Renderer="Falagard/Button"
LookNFeel="TaharezLook/ImageButton" />
<FalagardMapping WindowType="TaharezLook/SystemButton TargetType="CEGUI/PushButton"
Renderer="Falagard/SystemButton" LookNFeel="TaharezLook/Button" />
```

前面我们已经介绍过这里，渲染窗口和窗口控件是可以重用的，这四类按钮的定义中非常明显。首先看内部控件，只有Checkbox使用的是CheckBox作为内部控件，其他三个都是用的是PushButton。再看渲染窗口，一共使用了三个渲染窗口类型，分别是Falagard/Button，Falagard/ToggleButton以及Falagard/SystemButton。这四类控件的外观定义都是不同的，如果相同哪么他们显示的样子也是一样的，所以一般来说控件外观定义都是不相同的。

从这些映射中可以发现控件的实现和控件渲染的实现是可以重用的。如果读者需要的只是外观不同而逻辑都相同的多个类似的控件，完全没有必要为每个控件都开发一个对应类，使用现在已有的就可以了。

> 注意
> 渲染窗口的类型和渲染窗口类是两码事，渲染窗口类有个代表它的类型名称就是渲染窗口类型或者叫做渲染窗口类厂的名称。

所有的渲染窗口都必须派生自WindowRenderer类，这个类是个虚基类，派生类需要实现其定义的各种接口，其中最重要的当然是render接口了。所以我们每个渲染窗口都必须介绍这个函数。WindowRenderer类已经在第3.2.4节介绍过，这里就不在赘述了。

下面介绍第一节各种按钮的渲染窗口类。

10.1 按钮的渲染窗口

按钮基类是没有渲染窗口的，因为它根本就不会被实例化。它是各种按钮的基类，不会被直接使用所以就没有必要开发渲染窗口了。此外有些按钮只是外观的定义不同，没有必要为每个控件开发对应的渲染窗口。我们完全可以使用相同的渲染窗口，只是外观定义不同就可以了。比如RadioButton和Checkbox，这两个控件都使用FalagardToggleButton渲染窗口。外观定义中会定义渲染时使用的图像。

10.1.1 FalagardButton渲染类

首先看这个渲染窗口类的数据成员，它没有什么数据成员除了一个静态的代表类型的字符串外。

```
static const utf8   TypeName[];
```

这个字符串的值如下所示，它正好是Scheme文件中映射的渲染窗口的类型名称。

```
const utf8 FalagardButton::TypeName[] = "Falagard/Button";
```

它也只实现了render函数，其他函数使用基类默认。下面是render的具体实现。

```
void FalagardButton::render()
{
   ButtonBase* w = (ButtonBase*)d_window;
   const WidgetLookFeel& wlf = getLookNFeel();
   bool norm = false;
   String state;
      //是否是禁用状态
      if (w->isDisabled())
      {
         state = "Disabled";
```

```
    }
    //是否是按下状态
    else if (w->isPushed())
    {
    state = w->isHovering() ? "Pushed" : "PushedOff";
    }
    //是否鼠标在按钮上
  else if (w->isHovering())
    {
      state = "Hover";
    }
    //都不是则是正常状态
    else
    {
      state = "Normal";
      norm = true;
    }
    //如果上面的某些状态不存在则修改状态为正常状态，正常状态必须要存在
  if (!norm && !wlf.isStateImageryPresent(state))
  {
    state = "Normal";
  }
    //获取这个状态并渲染
  wlf.getStateImagery(actualStateName(state)).render(*w);
}
```

读者可能不知道这些状态在那里定义，其实他们都是在外观定义中定义的。像上面的这个渲染窗口就必须定义一个名为Normal的StateImagery元素。其他的名称的元素是可选的。也就是说要使用这个渲染窗口就必须在外观定义中定义名为Normal的StateImagery元素。这就是为什么CEGUI要专门提供一个手册来说明每个渲染窗口都需要对应的外观定义中必须实现那些命名区域，状态图像等元素。

10.1.2 FalagardTabButton渲染类

这个渲染窗口同样非常简单，它的成员变量和FalagardButton一样，都只有一个静态的代表渲染窗口类型的变量。每个渲染窗口都必须有这样一个而且名称必须是TypeName的类型静态变量。下面我们介绍它的渲染函数。

```
void FalagardTabButton::render()
{
  TabButton* w = (TabButton*)d_window;
  // 获取外观定义
  const WidgetLookFeel& wlf = getLookNFeel();
    TabControl* tc = static_cast<TabControl*>(w->getParent()->getParent());
  String state;
    String prefix((tc->getTabPanePosition() == TabControl::Top) ? "Top" : "Bottom");
    //根据控件窗口的不同状态，设置对应的渲染状态名称
    if (w->isDisabled())
      state = "Disabled";
    else if (w->isSelected())
      state = "Selected";
    else if (w->isPushed())
      state = "Pushed";
    else if (w->isHovering())
      state = "Hover";
    else
```

```
        state = "Normal";
    //如果状态不存在
  if (!wlf.isStateImageryPresent(prefix + state))
  {
    state = "Normal";
          if (!wlf.isStateImageryPresent(prefix + state))
                prefix = "";
  }
    //获取这个状态对应的元素并渲染它
  wlf.getStateImagery(prefix + state).render(*w);
}
```

实现起来就是这么简单，我们发现渲染窗口的逻辑比控件的逻辑简单多了，它主要是根据控件的状态，从外观定义中获取对应的可以渲染的元素并且渲染它。可渲染的元素是知道如何渲染自己的。

10.1.3 FalagardToggleButton渲染类

这个类就更加简单了，它派生自FalagardButton类。只是实现了前者的虚函数actualStateName，改变了渲染状态而已。这个函数的具体实现如下。

```
String FalagardToggleButton::actualStateName(const String& name) const
{
    //如果当前是选中状态则在原来的名字前面加上Selected
    bool selected = PropertyHelper::stringToBool(d_window->getProperty("Selected"));
  return selected ? "Selected"+name : name;
}
```

这个函数的外观定义中必须还要多实现一个StateImegery元素SelectedNormal。它可以使用的外观定义是FalgardButton类的两倍（在每个状态前面加上Selected作为一个新的状态）。

10.1.4 按钮对应的外观定义

外观定义的介绍其实在第6章已经详细说过，尤其以按钮作为例子做了介绍。但是这里为了巩固，再次介绍按钮的外观定义。我们以CEGUI官方的例子的数据为例。使用的外观定义是TaharezLook/Button。这里我们不一一列出所有的实现，只是简单的介绍正常状态的实现，其他的实现类似读者可以自己阅读。

要定义一个状态图像元素，就必须先要定义一些ImageSection（或者使用其他外观中定义的，使用Section元素来声明它定义在那个外观中，具体的名称）。我们首先看正常状态的图像段的定义。

```
<ImagerySection name="normal">
  <FrameComponent>
    <Area>
      <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
    </Area>
    <Image type="LeftEdge" imageset="TaharezLook" image="ButtonLeftNormal" />
    <Image type="RightEdge" imageset="TaharezLook" image="ButtonRightNormal" />
    <Image type="Background" imageset="TaharezLook" image="ButtonMiddleNormal" />
  </FrameComponent>
  <ImageryComponent>
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
```

```
    </Area>
    <ImageProperty name="NormalImage" />
    <VertFormatProperty name="VertImageFormatting" />
    <HorzFormatProperty name="HorzImageFormatting" />
  </ImageryComponent>
</ImagerySection>
```

这个图像段中使用了两个Component元素，一个框架组件对象，一个普通图像组件元素对象。两个组件的区域都是控件窗口的整个大小。ImagerySection元素可以有三个组件元素，他们的渲染顺序是先渲染框架组件元素，然后是图像组件元素，最后是文本组件元素。

其次，我们看按钮的文本定义。

```
<ImagerySection name="label">
  <TextComponent>
    <Area>
      <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
    </Area>
    <VertFormatProperty name="VertLabelFormatting" />
    <HorzFormatProperty name="HorzLabelFormatting" />
  </TextComponent>
</ImagerySection>
```

这个文本显示的区域也是整个按钮，并且指定了水平和竖直格式的属性名称。也就是说水平和竖直文本显示格式是通过这两个属性指定的。修改这两个属性就可以改变水平和竖直的文本格式。

最后，有了这两个显示段，就可以定义一个正常的状态了。

```
<StateImagery name="Normal">
  <Layer>
    <Section section="normal" />
    <Section section="label">
      <ColourProperty name="NormalTextColour" />
    </Section>
  </Layer>
</StateImagery>
```

这个状态段名叫Normal是不是和程序里指定的一样。这个段中有图片背景和显示文字构成。文字还指定了显示颜色，通过属性NormalTextColour来获取。

有了Normal状态的介绍相信其他状态读者应该也可以理解了。这里就不在介绍了。

10.2 编辑框的渲染窗口

编辑框的渲染窗口比较复杂，但还是和其他的渲染窗口类似，主要是它的render函数实现比较复杂。编辑框的显示包含了是否选中，光标的显示等问题。

10.2.1 渲染窗口的实现

渲染窗口的实现类是FalagardEditbox，它派生自EditboxWindowRenderer，而EditboxWindowRenderer又派生自WindowRenderer，所以说FalagardEditbox还是派生自WindowRenderer接口的（只不过是间接派生）。中间类EditboxWindowRenderer只是定义了一个接口getTextIndexFromPosition，这个接口函数返回鼠标位置对应的编辑框中的文本中的那一个。我们前文已经讲过这个函数最终还是会调用控件字体的对应函数来实现，不知道为什么CEGUI的设计者把它放在了渲染窗口中实现。下面我们首先看它的实现。

```
size_t FalagardEditbox::getTextIndexFromPosition(const Point& pt) const
{
```

```
    Editbox* w = (Editbox*)d_window;
    //将屏幕坐标转换为窗口的局部坐标
    float wndx = CoordConverter::screenToWindowX(*w, pt.d_x);
        //位置减去文本的偏移，可能这是一个在渲染窗口中定义这个函数的原因
    wndx -= d_lastTextOffset;
    //根据不同的显示状态通过字体计算鼠标在显示文本中的位置
    if (w->isTextMasked())
    {
        return w->getFont()->getCharAtPixel(String(w->getText().length(), w->
                            getMaskCodePoint()), wndx);
    }
    else
    {
        return w->getFont()->getCharAtPixel(w->getText(), wndx);
    }
}
```

通过这个函数的实现，我们发现将它定义在渲染窗口中是有道理的。首先有个文本偏移在控件中是不容易知道的，其次放在这里应该也是比较符合CEGUI的设计思路的。下面介绍最重要的函数。

```
void FalagardEditbox::render()
{
        Editbox* w = (Editbox*)d_window;
    const StateImagery* imagery;
    //获取外观定义并且描绘编辑框的背景
    const WidgetLookFeel& wlf = getLookNFeel();
    //根据编辑框的状态选中对应的外观状态
    imagery = &wlf.getStateImagery(w->isDisabled() ? "Disabled" : (w->isReadOnly() ?
        "ReadOnly" : "Enabled"));
    //渲染背景框
    imagery->render(*w);
    //获取文本显示的区域，外观中必须定义这个命名区域
    const Rect textArea(wlf.getNamedArea("TextArea").getArea().getPixelRect(*w));
    //获取窗口对应的字体，字体类是渲染文本的核心类
    Font* font = w->getFont();
    if (!font)
        return;
        //最终显示文本指针，根据当前是否显示掩码它的值不同
    String* editText;
    //是否是掩码状态
    String maskedText, windowText;
    if (w->isTextMasked())
    {
        maskedText.insert(0, w->getText().length(), w->getMaskCodePoint());
        editText = &maskedText;
    }
    //不是掩码状态
    else
    {
        windowText = w->getText();
        editText = &windowText;
    }
        //计算合适的渲染文本位置确保当前光标总是可见的
```

```
float textOffset;
float extentToCarat = font->getTextExtent(editText->substr(0, w->getCaratIndex()));
//获取光标的显示图像段（ImagerySection）
const ImagerySection& caratImagery = wlf.getImagerySection("Carat");
//保存图像段的宽度
float caratWidth = caratImagery.getBoundingRect(*w, textArea).getWidth();
//如果编辑框是非激活的
if (!w->hasInputFocus())
{
    textOffset = d_lastTextOffset;
}
//如果光标在左边而且现在无法显示在编辑框的可视区域内
else if ((d_lastTextOffset + extentToCarat) < 0)
{
    textOffset = -extentToCarat;
}
//如果光标在右边而且现在无法显示在编辑框的可视区域内
else if ((d_lastTextOffset + extentToCarat) >= (textArea.getWidth() - caratWidth))
{
    textOffset = textArea.getWidth() - extentToCarat - caratWidth;
}
//否则光标还在可视区域内，哪么使用上次的偏移
else
{
    textOffset = d_lastTextOffset;
}
ColourRect colours;
float alpha_comp = w->getEffectiveAlpha();
//开始描绘文本了
Rect text_part_rect(textArea);
//扩大描绘区域，容纳看不到的文字（textOffset一般是小于等于0的）
text_part_rect.d_left += textOffset;
//竖直方向居中对齐
text_part_rect.d_top += (textArea.getHeight() - font->getFontHeight()) * 0.5f;

//获取非高亮状态下的文本显示颜色
colour unselectedColour(getUnselectedTextColour());
//描绘高亮前面的非高亮文本（可能什么也没有描绘）
String sect = editText->substr(0, w->getSelectionStartIndex());
colours.setColours(unselectedColour);
colours.modulateAlpha(alpha_comp);
    //缓冲起来，第二层缓冲
w->getRenderCache().cacheText(sect, font, LeftAligned, text_part_rect, 0, colours,
    &textArea);
//调整矩形的大小，准备描绘第二段
text_part_rect.d_left += font->getTextExtent(sect);
//描绘高亮状态的文本（也可能什么也没描绘）
sect = editText->substr(w->getSelectionStartIndex(), w->getSelectionLength());
colours.setColours(getSelectedTextColour());
colours.modulateAlpha(alpha_comp);
```

```
    w->getRenderCache().cacheText(sect, font, LeftAligned, text_part_rect, 0, colours,
        &textArea);
    //调整矩形位置，为最后一部分做准备
    text_part_rect.d_left += font->getTextExtent(sect);
    //描绘高亮文本后面的非高亮文本
    sect = editText->substr(w->getSelectionEndIndex());
    colours.setColours(unselectedColour);
    colours.modulateAlpha(alpha_comp);
    w->getRenderCache().cacheText(sect, font, LeftAligned, text_part_rect, 0, colours,
        &textArea);
    //记住这次的偏移为下一次描绘使用
    d_lastTextOffset = textOffset;
    //是否当前的编辑框是激活的
    bool active = (!w->isReadOnly()) && w->hasInputFocus();
    //渲染选中文本的选中区域
    if (w->getSelectionLength() != 0)
    {
        //计算选中文本在文本中的位置起始和结束
        float selStartOffset = font->getTextExtent(editText->substr(0, w->
                                getSelectionStartIndex()));
        float selEndOffset   = font->getTextExtent(editText->substr(0, w->
                                getSelectionEndIndex()));
        //计算选中区域
        Rect hlarea(textArea);
        hlarea.d_left += textOffset + selStartOffset;
        hlarea.d_right = hlarea.d_left + (selEndOffset - selStartOffset);
        // 获取选中区域的状态元素并渲染之
        wlf.getStateImagery(active ? "ActiveSelection" : "InactiveSelection").render(*w, hlarea,
                0, &textArea);
    }
    //渲染光标，只有激活的编辑框才具有光标
    if (active)
    {
            //计算光标的位置，并且渲染之
        Rect caratRect(textArea);
        caratRect.d_left += extentToCarat + textOffset;
        caratImagery.render(*w, caratRect, 0, 0, &textArea);
    }
}
```

这个函数非常庞大，但思路是清晰的。读者可能不是非常理解文本偏移和为什么划分为三段来渲染。偏移一般是负值或者0，它代表从文本开始到最后一个不可以显示的字符的宽度的负值。为什么会有不可以显示的文本呢？因为玩家输入的文本的宽度可能已经超过了编辑框的文本区域的宽度，无法显示下，但是输入光标必须要显示，因此就要向左滚动，导致一部分文本无法显示。文本分为三段也很好理解，它以选中文本为中心两侧都有可能出现没有被选中的文本。因此分为了三部分来渲染。最后一步描绘选中文本的选中区域和光标。

10.2.2 编辑框的外观

编辑控件需要两个自定义属性，他们是 NormalTextColour和SelectedTextColour外观定义中必须定义这两个属性。读者是否还记得如何在外观定义中定义属性吗？（使用PropertyDefinition元素）。必须指定光标属性（MouseCursorImage）的初始值。必须指定以下几种状态：Enabled，ReadOnly ，Disabled ，ActiveSelection，InactiveSelection。必须指定一个命名区域（TextArea），它是显示文本的区域。必须指定一个图像段Carat，用来显示光标。

每个渲染窗口都需要那些状态，命名属性，图像段等这些信息都在渲染窗口类的头文件中有详细的说明。读者也可以在CEGUI官方提供的说明文档中找到。这个文档在光盘的CEGUI文档目录下的压缩文件包中，名叫FalagardSkinning.pdf这个文件我们已经提到过多次了。

我们介绍了这么多关于外观定义的内容，相信读者应该理解如何定义一个外观了，以后的章节我们将不会在介绍外观的定义。本节我们也是简单的介绍编辑框定义中需要的元素。首先介绍文本显示区域的定义。

```
<NamedArea name="TextArea">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="5" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="5" /></Dim>
    <Dim type="RightEdge" ><UnifiedDim scale="1.0" offset="-5" type="RightEdge" />
    </Dim>
    <Dim type="BottomEdge" ><UnifiedDim scale="1.0" offset="-5" type="BottomEdge"
    /></Dim>
  </Area>
</NamedArea>
```

读者知道这个命名区域的范围吗？它是编辑框控件窗口区域各向向内侧偏移5个像素形成的区域。下面介绍一个图像段，它被选中状态用到。

```
<ImagerySection name="selection">
  <ImageryComponent>
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="RightEdge" ><UnifiedDim scale="1.0" type="RightEdge" /></Dim>
      <Dim type="BottomEdge" ><UnifiedDim scale="1.0" type="BottomEdge"
          /></Dim>
    </Area>
    <Image imageset="TaharezLook" image="TextSelectionBrush" />
    <VertFormat type="Stretched" />
    <HorzFormat type="Stretched" />
  </ImageryComponent>
</ImagerySection>
```

非常简单，覆盖整个编辑框，渲染的图像是set:TaharezLook image:TextSelectionBrush，水平和竖直方向都是拉伸显示。下面是一个激活选中状态的定义。

```
<StateImagery name="ActiveSelection">
  <Layer>
    <Section section="selection">
      <ColourProperty name="ActiveSelectionColour" />
    </Section>
  </Layer>
</StateImagery>
```

激活状态使用的颜色是ActiveSelectionColour属性的值，这个属性在外观定义中定义过了。它的定义如下。

```
 <PropertyDefinition name="ActiveSelectionColour" initialValue="FF607FFF"
redrawOnWrite="true" />
```

当然编辑框的外观定义还有其他的状态和图像段的定义，这里就不一一介绍了。

10.3 框架窗口的渲染窗口

框架窗口有两个重要的函数需要介绍，一个就是render函数，另一个是获取客户区域的函数getUnclippedInnerRect函数。首先介绍前者。

```
void FalagardFrameWindow::render()
```

```
{
   FrameWindow* w = (FrameWindow*)d_window;
   //如果是卷起状态则不渲染
   if (w->isRolledup())
      return;
   //生成状态名称
   String stateName(w->isDisabled() ? "Disabled" : (w->isActive() ? "Active" : "Inactive"));
   stateName += w->getTitlebar()->isVisible() ? "WithTitle" : "NoTitle";
   stateName += w->isFrameEnabled() ? "WithFrame" : "NoFrame";
   const StateImagery* imagery;
      //尝试获取对应的状态，如果失败返回不报错
   try
   {
      // get WidgetLookFeel for the assigned look.
      const WidgetLookFeel& wlf = getLookNFeel();
      // try and get imagery for our current state
      imagery = &wlf.getStateImagery(stateName);
   }
   catch (UnknownObjectException&)
   {
      return;
   }
   //真正的渲染操作
   imagery->render(*w);
}
```

什么是卷起状态呢？前文作者也不是非常明白，但看到这个函数作者明白了。当用户双击标题栏的时候框架窗口是会在显示和不显示之间切换的，哪么卷起状态就是代表这个状态。如果是卷起状态则不显示直接返回。如果外观中没有定义框架渲染窗口需要的状态则什么也描绘不出，但不会报错。下一个重要的函数。

```
Rect FalagardFrameWindow::getUnclippedInnerRect(void) const
{
   FrameWindow* w = (FrameWindow*)d_window;
   if (w->isRolledup())
      return Rect(0,0,0,0);
   //生成客户区的命名区域名字
   String areaName("Client");
   areaName += w->getTitlebar()->isVisible() ? "WithTitle" : "NoTitle";
   areaName += w->isFrameEnabled() ? "WithFrame" : "NoFrame";
   //获取外观定义
   const WidgetLookFeel& wlf = getLookNFeel();
      //获取区域的大小并且返回
   return wlf.getNamedArea(areaName).getArea().getPixelRect(*w,
      w->getUnclippedPixelRect());
}
```

这个函数返回客户区域的大小，这个大小是在外观定义中决定的。

10.4 滚动条的渲染窗口

滚动条的渲染窗口有四个重要的函数，其中render是继承字WindowRender接口，其余三个继承自ScrollbarWindowRenderer接口。下面还是首先介绍render函数。

```
void FalagardScrollbar::render()
{
   const StateImagery* imagery;
```

```
    const WidgetLookFeel& wlf = getLookNFeel();
    //获取对应状态的外观状态
    imagery = &wlf.getStateImagery(d_window->isDisabled() ? "Disabled" : "Enabled");
    //渲染
    imagery->render(*d_window);
}
```

这个函数比较简单，只是简单的获取外观文件并且渲染。

下面介绍子窗口的布局函数。滚动条由三个子窗口，一个向上按钮，一个向下按钮和一个中间的Thumb。这个函数就是更新Thumb的位置。

```
void FalagardScrollbar::updateThumb(void)
{
    Scrollbar* w = (Scrollbar*)d_window;
    const WidgetLookFeel& wlf = getLookNFeel();
        //Thumb可以滚动的区域
    Rect area(wlf.getNamedArea("ThumbTrackArea").getArea().getPixelRect(*w));
    Thumb* theThumb = w->getThumb();
    float posExtent = w->getDocumentSize() - w->getPageSize();
    float slideExtent;
    if (d_vertical)
    {
        slideExtent = area.getHeight() - theThumb->getPixelSize().d_height;
        theThumb->setVertRange(area.d_top / w->getPixelSize().d_height, (area.d_top +
                slideExtent) / w->getPixelSize().d_height);
                //设置新的竖直位置
        theThumb->setPosition(UVector2(cegui_absdim(area.d_left),
        cegui_reldim((area.d_top + (w->getScrollPosition() * (slideExtent / posExtent))) /
                w->getPixelSize().d_height)));
    }
    else
    {
        slideExtent = area.getWidth() - theThumb->getPixelSize().d_width;
        theThumb->setHorzRange(area.d_left / w->getPixelSize().d_width, (area.d_left +
                slideExtent)  / w->getPixelSize().d_width);
                //设置新的水平位置
        theThumb->setPosition(UVector2(cegui_reldim((area.d_left + (w->getScrollPosition() *
        (slideExtent / posExtent))) / w->getPixelSize().d_width),
         cegui_absdim(area.d_top)));
    }
}
```

读者或许不明白设置位置的计算方法。slideExtent是滚动区域减去Thumb的大小。posExtent是文档的长度减去每一页的长度。哪么slideExtent / posExtent（设为p）就是文档长度的每一个单位对应的Thumb显示的多少，getScrollPosition获取的是当前文档的单位数

量（设为t），t乘以p就是当前文档的位置换算为Thumb显示区域的位置了，在加上显示区域的顶部得到最终的绝对位置，最后处以高度或者宽度就是对应的相对数值。

下面这个函数获取当前Thumb代表的位置，计算后返回Thumb位置对应的文档位置。

```
float FalagardScrollbar::getValueFromThumb(void) const
{
    Scrollbar* w = (Scrollbar*)d_window;
    const WidgetLookFeel& wlf = getLookNFeel();
    Rect area(wlf.getNamedArea("ThumbTrackArea").getArea().getPixelRect(*w));
```

```
    Thumb* theThumb = w->getThumb();
    float posExtent = w->getDocumentSize() - w->getPageSize();
        //如果是竖直滚动条则返回对应文档的高度位置
    if (d_vertical)
    {
        float slideExtent = area.getHeight() - theThumb->getPixelSize().d_height;
        return (theThumb->getYPosition().asAbsolute(w->getPixelSize().d_height) - area.d_top)
        / (slideExtent / posExtent);
    }
        //否则是水平滚动条返回文档对应的宽度位置
    else
    {
        float slideExtent = area.getWidth() - theThumb->getPixelSize().d_width;
        return (theThumb->getXPosition().asAbsolute(w->getPixelSize().d_width) - area.d_left)
        / (slideExtent / posExtent);
    }
}
```

这个函数中的slideExtent和posExtent与updateThumb中的含义相同，只不过这个函数返回的是文档的位置，前者计算的是Thumb的位置。

另一个函数根据鼠标是在Thumb的上面还是下面返回-1或者1，以便滚动条控件计算如何滚动。

```
float FalagardScrollbar::getAdjustDirectionFromPoint(const Point& pt) const
{
    Scrollbar* w = (Scrollbar*)d_window;
    Rect absrect(w->getThumb()->getUnclippedPixelRect());
    if ((d_vertical && (pt.d_y > absrect.d_bottom)) ||
        (!d_vertical && (pt.d_x > absrect.d_right)))
    {
        return 1;
    }
    else if ((d_vertical && (pt.d_y < absrect.d_top)) ||
        (!d_vertical && (pt.d_x < absrect.d_left)))
    {
        return -1;
    }
    else
    {
        return 0;
    }
}
```

这个函数中如果是竖直滚动条而且鼠标的位置在Thumb下面返回1，否则返回-1如果水平滚动条而且鼠标位置Thumb的右面返回1否则返回-1。这个值被滚动条控件使用决定是向前还是向后滚动。

到这里所有的渲染窗口都介绍完了。当然还有许多的其他控件的渲染窗口没有介绍到。但他们也都非常重要。之所以要介绍控件的实现和控件的渲染窗口主要是告诉读者要实现一个控件都需要做那些工作，以及如何去做。

10.5 本章小结

本章主要介绍了几个渲染窗口，他们和第7章中介绍的窗口控件对应。在这一版的CEGUI中将控件的逻辑和控件的渲染分开了，先前版本的CEGUI是在一起实现的。这样做有它的好处，就是渲染窗口和控件都可以重用了。而且结构也变的比较清晰了。

这一章的习题就是读者继续阅读其他没有介绍过的渲染窗口，读者详细阅读渲染窗口相关的外观定义。希望读者能够理解外观和渲染窗口之间的对应关系。只有理解了它们将来才可以很轻松的完成控件的编写。

第11章 CEGUI控件的实现步骤

前10章都是介绍CEGUI的各种概念和各种关键类的实现。从这章开始以后的章节都会介绍CEGUI的各种应用。主要介绍编写一些控件的实例。本章起到抛砖引玉的作用，它将前10章与后5章联系起来。本章主要介绍添加控件的基本步骤以及其他一些使用CEGUI的技巧。

11.1 添加新控件的步骤

在决定添加一个新控件之前，读者应该先考虑，不添加新控件只是修改外观定义是否可以实现需要的效果。如果可以哪么就没有必要添加控件。只要添加新的外观定义使用原有的控件来定义一种新的窗口类型就可以了。如何定义？在Scheme文件中增加一个映射，前文我们还专门讲过类似的情况。

如果必须通过添加新控件来实现，哪么我们需要思考如何实现。一般就是重载Window类的一些消息函数来实现功能。

好，下面我们介绍添加一个控件的基本步骤。

第一步，添加一个新类，这个类必须派生自Window或者它的派生类。

我们创建一个TestWindow的类，它派生自Window类，声明文件和实现文件分别是TestWindow.h和TestWindow.cpp，申明文件需要我们注意的有。

1.必须要有头文件包含的保护宏，如下所示。

```
#ifndef _CEGUITestWindow_h_
#define _CEGUITestWindow_h_
//这里是头文件的内容
#endif
```

其实还可以是另外一种方法就是使用预处理指令，但是这个指令有些编译器是不支持的。这个指令如下所示，它必须放在头文件的最开始。

```
#pragma once
```

2.头文件必须包含两个文件，当然需要那些包含那些就可以了，但这两个必须被包含。

```
#include "CEGUIBase.h"
#include "CEGUIWindow.h"
```

3.声明CEGUI的命名空间。

```
namespace CEGUI
{  }
```

4.定义TestWindow类。其中CEGUIEXPORT宏声明这个类为动态库的导出类。当然如果编译的是静态库，哪么它什么也不是了。

```
class CEGUIEXPORT  TestWindow : public Window
{
public:
    Combobox(const String& type, const String& name);
    virtual ~Combobox(void);
}
```

5.添加成员变量，事件变量，成员函数，属性类的静态变量等。这些都是根据需要添加，并不是那个必须添加，那个肯定不能添加。有些变量和成员函数是必须实现的。必须存在的成员变量有如下两个。

```
    static const String EventNamespace;                    //全局事件的事件空间
  static const String WidgetTypeName;          //窗口类厂的名称
```

窗口类厂的名称就是Scheme文件中的FalagardMapping元素中的内部控件名称。就是TargetType指定的名称。这里我们实现如下。

```
const String TestWindow::EventNamespace("TestWindow")
const String TestWindow::WidgetTypeName("CEGUI/TestWindow")
```

有一个必须实现的虚函数是testClassName_impl，用它来在运行时确定类的类型。也就是RTTI（Run-Time Type Information），运行时的类型信息。

```
virtual bool     testClassName_impl(const String& class_name) const
{
    if (class_name=="Combobox")     return true;
    return Window::testClassName_impl(class_name);
}
```

其他的变量和成员函数可以根据需要添加。

第二步，创建所需的属性类。

这里举一个属性类的例子，假设属性为SelectedTextColor，选择状态的文本颜色。

1.创建属性类，派生自Property类，定义如下。

```
namespace TestWindowProperties
{
class SelectedTextColor: public Property
{
public:
   SelectedTextColor() : Property(
         "SelectedTextColor",
         "设置或者获取选择状态下的文字颜色",
         "FFFFFFFF")
   {}
      //设置和获取属性的函数
   String  get(const PropertyReceiver* receiver) const;
   void    set(PropertyReceiver* receiver, const String& value);
};
}
```

属性类值需要两个函数一个设置函数一个获取函数。当然还有个构造函数，它有四个参数第一个是属性的名称，第二个是属性的帮助信息，第三个是属性的默认值，第四个是是否写入XML文件的标志。

属性一般要在控件名加上Property的命名空间内。这个属性的命名空间是TestWindowProperties，这个命名空间应该包含所有的TestWindow的属性。

2.实现获取和设置函数。

这两个函数首先要在控件里实现，然后属性里只是简单的调用控件的实现函数。

```
void TestWindow::setSelectedTextColour(colour& c)
{
    d_selectTextColor = c;
}
colour TestWindow::getSelectTextColour()
{
    return d_selectTextColor;
}
```

其中d_selectTextColor为colour类型的成员变量，它对应这个属性。哪么属性的设置和获取函数就可以实现了。

```
String SelectedTextColor::get(const PropertyReceiver* receiver) const
{
     return static_cast<TestWindow*>(receiver)->getSelectTextColour();
}
void  SelectedTextColor::set(PropertyReceiver* receiver, const String& value)
{
    static_cast<TestWindow*>(receiver)->setSelectedTextColour(
                                        PropertyHelper::stringToColour(value));
```

```
}
```

3.定义全局的静态属性变量，这个变量定义在TestWindow类中，作为它的静态变量。

```
static TestWindowProperties::SelectedTextColor        d_selectedTextColourProperty;
```

注意属性在整个CEGUI系统中只有一个实例，就是这个定义在对应类中的实例。

4.添加属性到属性集中。

```
void TestWindow::addTestWindowProperties(void)
{
    addProperty(&d_selectedTextColourProperty);
}
```

这个函数应该在TestWindow的构造函数中调用。

第三步，添加所需的事件，事件其实非常简单，可以看做它就是一个字符串和一些处理函数构成的。处理函数不是我们提供的而是关心这个事件的其他模块提供的。所以事件只需要定义一个代表这个事件名称的字符串就可以了。

1.定义一个事件。

我们这里举个鼠标单击的事件（这个事件在Window类里已经定义，这里只是举个例子而已）。在TestWindow中声明这个事件。

```
static const String EventLeftClicked;
```

然后在TestWindow.cpp中实现它。

```
const String TestWindow::EventLeftClicked( "LeftClicked" );
```

2.激发一个事件。当这个事件需要激发的时候，调用fireEvent函数激发它。究竟什么时候激发这个事件是根据控件的设计了。

```
fireEvent(EventLeftClicked, e, EventNamespace);
```

激发这个事件的效果是所有关心这个事件的处理函数都会被调用。

第四步，重载需要的Window类提供的消息函数。

控件的基类Window类提供了许多的虚函数，这些函数派生类可以重载以实现特殊的功能。最经常重载的有鼠标相关的消息函数，键盘相关的函数。比如下面这些函数。

```
    virtual void    onMouseButtonDown(MouseEventArgs& e);
    virtual void    onMouseButtonUp(MouseEventArgs& e);
    virtual void    onMouseDoubleClicked(MouseEventArgs& e);
    virtual void    onMouseTripleClicked(MouseEventArgs& e);
    virtual void    onMouseMove(MouseEventArgs& e);
    virtual void    onCaptureLost(WindowEventArgs& e);
    virtual void    onCharacter(KeyEventArgs& e);
    virtual void    onKeyDown(KeyEventArgs& e);
    virtual void    onTextChanged(WindowEventArgs& e);
```

这些函数都是被CEGUI系统自动调用的，只要重载他们就可以了。

第五步，具体实现控件的逻辑。

这一步根据控件的需求各异，这里就不在举例了。

一般来说添加一个控件就是这么五步。其实读者完全可以拷贝一份已经实现好的控件到自己创建的控件文件中，然后替换原来的控件的名称为自己控件的名称。然后删除不需要的接口，添加一些需要的接口。修改宏的名称等步骤，就可以实现一个控件的基本框架了。

11.2 添加渲染窗口的步骤

有了控件这个控件还不能显示，必须有渲染窗口的支持。同控件一样，如果现存的渲染窗口已经可以支持我们的控件的渲染了，哪么没有必要专门写一个渲染窗口。直接使用它，然后提供不同的外观定义就可以了。渲染窗口都必须直接或者间接的派生自WindowRender接口，而且必须实现render接口函数。渲染窗口也可以有属性，这些属性在控件使用这个渲染窗口的时候被添加到控件中，当控件和渲染窗口脱离的时候这些属性被取消掉。

渲染窗口的实现步骤和控件的大致类似，只是它一般没有事件的处理。

第一步，实现渲染窗口类。

我们这里还是举TestWindow的例子，我们要实现它的渲染窗口FalagardTestWindow。下面是这个类的具体定义。

```
#ifndef _FalTestWindow_h_
#define _FalTestWindow_h_
#include "FalModule.h"
#include "elements/CEGUITestWindow.h"
#include "FalTestWindowProperties.h"
#if defined(_MSC_VER)
#     pragma warning(push)
#     pragma warning(disable : 4251)
#endif
//开始命名空间
namespace CEGUI
{
class FALAGARDBASE_API FalagardTestWindow: public WindowRenderer
{
public:
   static const utf8   TypeName[];    //渲染窗口类型（类厂的名字）
      //构造函数
   FalagardScrollbar(const String& type);
};
}
```

渲染窗口类型的定义如下。

```
const utf8 TestWindow::TypeName[] = "Falagard/TestWindow";
```

第二步，实现属性，如果这个渲染窗口需要属性的话。

实现的方法和控件的实现方法非常类似。只是在添加属性的时候有些不同。在构造函数中不是调用addProperty函数，因为渲染窗口类没有派生自PropertySet所以无法调用这个函数。但WindowRender类提供了一个函数registerProperty，它负责保存属性。

```
void WindowRenderer::registerProperty(Property* property)
{
   d_properties.push_back(property);
}
```

其中d_properties成员变量是一个Property指针的数组定义如下。

```
typedef std::vector<Property*> PropertyList;
PropertyList d_properties;
```

这样在构造函数中可以调用这个函数保存属性。在和控件链接和脱离的时候会调用这两个函数。

```
//链接的时候添加属性到控件
void WindowRenderer::onAttach()
{
   PropertyList::iterator i = d_properties.begin();
   while (i != d_properties.end())
   {
      d_window->addProperty(*i);
      ++i;
   }
}
//脱离是移除控件的对应属性
void WindowRenderer::onDetach()
```

```
{
   PropertyList::reverse_iterator i = d_properties.rbegin();
   while (i != d_properties.rend())
   {
      d_window->removeProperty((*i)->getName());
      ++i;
   }
}
```

第三步，实现render函数。

这个函数为必须实现的函数，当然根据渲染窗口的复杂性还可能会有其他的函数。这些函数一般是控件需要调用的，也就是说这些函数是提供给控件的接口函数。比如第10章介绍的EditBox的渲染窗口就实现了getTextIndexFromPosition函数。

到这一步渲染窗口的具体功能已经实现了，哪么它还需要注册才能使用。下一节介绍控件和渲染窗口的注册。

11.3 其他步骤

在本书介绍的这一版CEGUI的实现中，类厂的实现已经可以通过几个宏来完成了。本书在第3.3.2节已经介绍了这几个宏。这里就不在介绍了。需要告诉读者的渲染窗口的类厂定义统一在FalModule.cpp文件中。控件类厂的定义统一在CEGUIWindowFactory.cpp中。下面我们以TestWindow为例说明如果简单的添加类厂。

第一步，添加控件的类厂。

在CEGUIWindowFactory.cpp文件中添加如下一行。

```
CEGUI_DECLARE_WINDOW_FACTORY(TestWindow)
```

只需这么一行，这个类厂会被自动的创建了。

第二步，添加渲染窗口的类厂。

在FalModule.cpp文件中添加如下几行。

```
CEGUI_DEFINE_WR_FACTORY( FalagardTestWindow)
CEGUI_START_WR_FACTORY_MAP( Falagard )
   CEGUI_WR_FACTORY_MAP_ENTRY( FalagardTestWindow)
CEGUI_END_WR_FACTORY_MAP
```

其中第2和第4行是已经存在的，我们的第3行必须添加在它的中间。当然它中间会有许多其他渲染窗口的定义。

只需简单的几步就可以完成两个类厂的定义和实现，实在是方便。

实现了控件和渲染窗口，哪么还差外观的定义。外观定义是必须存在的，如果读者不需要具体外观的实现可以指定一个空的外观，但不能不指定外观。关于如何实现外观的定义以及它的步骤我们不在介绍了，前几章已经做了详细的介绍。这里我们假设读者已经实现了外观定义，外观的名称是TaharezLook/TestWindow，下面我们会使用到这个名称。

最后需要在模式文件中添加的内容。很简单就是一个映射。

```
<FalagardMapping WindowType="TaharezLook/TestWindow"    TargetType="CEGUI/TestWindow"
Renderer="Falagard/TestWindow"    LookNFeel="TaharezLook/TestWindow" />
```

这里面的各个元素的含义已经介绍了不止一遍了，相信读者应该知道他们的含义了。

当然如果外观中使用到了一些图像集哪么还要在模式文件中添加图像集的定义。类似下面的定义。

```
<Imageset Name="TaharezLook" Filename="TaharezLook.imageset" />
```

如果还使用到了其他的资源也可以在模式文件里添加，比如新添加了一个外观定义的文件来定义我们的TaharezLook/TestWindow外观。哪么可以新增一个外观定义的描述。类似下面的定义。

```
      <LookNFeel Filename="TaharezLook.looknfeel" />
```

总之，只要是和控件相关的资源都需要在模式文件中指定，如果这个先前并没有指定过的话。

11.4 本章小结

本章主要介绍了控件和渲染窗口的具体实现步骤。为后面章节的例子实现打下坚实的基础。也可提供一种制作的方法。免得读者不知如何下手。

本章的习题只有一个就是按照本章介绍的步骤实现一个控件。控件的具体功能读者自己设计，但希望读者能够练习。只有多多练习才能取得更快的进步。

第12章 定时器控件

从这一章开始将正式进入本人的例子部分。理解了原理还不够，还需要大量的练习才能真正的会修改CEGUI会更好的使用CEGUI。本书的所有例子都是笔者自己编写的，可以自由使用。包括商业和非商业的使用。本书以后章节将会详细介绍每个例子的实现原理以及实现代码。

倒计时控件，在游戏中使用还是非常频繁的。这章提供了一个简单的实现。但它具备了倒计时的功能。具备一定的实用价值。

本章的代码是基于第5章介绍的程序框架，本书所有的代码都是基于它的。如果读者有些遗忘则可以在阅读第5章，或者直接阅读本章的代码。

12.1 定时器控件的实现

定时器是设置一个时间后就会开始倒计时的控件。游戏中使用比较常见。

12.1.1 定时器逻辑

介绍控件的基本方法，本书采用的是先介绍成员变量，然后介绍重要的成员函数。其中穿插的介绍为什么要这么实现，它的原理和好处是什么。哪么，首先介绍定时器的成员变量。

```
float d_lastTime;                    //定时器设置的时间，倒计时持续多久的属性变量
char d_timeFormat[32];               //倒计时器时间的显示格式的属性变量
bool d_isTimerEnd;                       //是否是倒计时结束的标志
bool d_showBackground;               //是否显示背景的属性变量
int d_timeEndCount;                  //激发多少次倒计时结束消息的属性变量
int d_curTimeEndCount;                   //当前激发了多少次倒计时事件
colour d_textColor;                  //文本显示的颜色的属性变量
```

这几个变量有的是内部变量，有的是属性变量。所谓属性变量就是有一个属性与他对应的变量。还有一些属性我们没有设置对应的变量，而是在使用的时候直接从属性中获取。比如倒计时器的前缀文字和后缀文字等。

属性的定义分为两种一种是在C++中使用代码来定义属性还有一种简单的方法是在外观文件中定义属性。这两种的各种操作完全相同。后者定义起来非常方便，但是它并不对应一个C++的属性类，它只是被保存成一个名称到值的映射，用它来模拟属性而已。

代码中属性定义如下。

```
static TimerWindowProperties::TimeFormat d_timeFormatProperty;
static TimerWindowProperties::TimerEndEventCount d_timeEndEventCountProperty;
static TimerWindowProperties::TimerValue d_timerValueProperty;
static TimerWindowProperties::TextColor d_textColorProperty;
static TimerWindowProperties::ShowTimerBackground d_showBackgroundProperty;
```

这五个属性对应五个成员变量，具体那个对应那个根据属性的名字就可以看出这里就不在介绍了。下面看看控件的事件定义。

```
static const String EventNamespace;
static const String EventTimerEnd;
```

其中EventNamespace是全局事件的名称，它用来区分事件来自那个控件。另一个EventTimerEnd就是定时器倒计时结束的时候激发的事件。这个事件可以激发多次，根据不同的用户设置决定。

最后一个变量就是类厂的名称或者叫做控件的类型。它用来区分控件，CEGUI根据它来获取类厂然后创建对应的控件。

```
static const String WidgetTypeName;
```

这个成员必须是静态而且名称也不可改变，必须是这个名称。因为类厂相关的宏需要用到这个变量。

下面介绍定时器的成员函数。和属性相关的获取和设置函数非常简单我们就不做介绍了。只是提醒读者一点，获取函数必须申明为常函数。下面介绍第一个函数计算倒计时文本的函数。我们设置倒计时的单位是秒，就是设置100就是100秒。

```
void TimerWindow::calcText(unsigned int times)
{
```

```
    //计算小时数，一小时有3600秒
    int hour = times/3600;
    //计算分钟，去掉小时的数量，得到分钟的秒数，每分钟有60秒
    int minute = (times%3600)/60;
    //计算秒，去掉小时和分钟对应的秒数，剩余的就是秒数
    int second =  (times%3600)%60;
    char buf[100];
    //根据使用者提供的字符串格式生成时间字符串，默认格式%d:%02d:%02d
    sprintf(buf, d_timeFormat, hour, minute, second);
    //获取文本前缀属性值
    String pre = getProperty("TextPrefix");
    //获取文本后缀属性值
    String suff = getUserString("TextSuffix_fal_auto_prop__");
    //设置窗口的文本
    setText(pre + buf + suff);
    //渲染缓冲无效，请求重绘
    requestRedraw();
}
```

这个函数结构清晰，读者唯一可能不明白的就是获取前缀和后缀属性的时候为什么不同，而且获取后缀的方法非常奇怪。这两个属性是在外观文件中定义的，他们和普通的属性不太相同，虽然他们可以使用属性的统一接口来访问，比如getProperty和setProperty函数。但是他们真正的值保存在窗口基类Window的成员变量d_userStrings中，这个变量是字符串映射到字符串的映射。映射属性的名称就是属性的定义名称加上后缀字符串"_fal_auto_prop__"，笔者是如何得知呢？当然是看源码了，在外观定义文件中PropertyDefinition元素对应的C++类是PropertyDefinition本书在6.4.4节已经详细介绍了这个类，读者可以自己温习。

另一个重要的函数就是updateSelf函数，这个函数被调用需要做一些处理。第五章提供的框架是无法调用到这个函数的。这里要新增一个CEGUI系统函数injectTimePulse的调用。
它向CEGUI注入时间脉冲，CEGUI会调用所有窗口的update函数，而这个函数会调用我们的updateSelf函数。这个函数的调用是没有条件的，只要injectTimePulse函数被调用就可以。我们在框架渲染函数中加入了这个函数的调用。

```
void CEGUISample::Render()
{
    glClearColor(0.0f, 0.0f, 0.3f, 1.0f);
    glColor3i(255,255,255);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    static unsigned long begin = timeGetTime();
    unsigned long now = timeGetTime();
    //注入调用
    CEGUI::System::getSingleton().injectTimePulse((float)(now - begin));
    begin = now;
    CEGUI::System::getSingleton().renderGUI();
    SwapBuffers(hDC);
}
```

下面介绍updateSelf函数。这个函数根据当前的逝去时间计算当前显示的文本内容。需要注意的是elapsed是以毫秒为单位的。

```
void TimerWindow::updateSelf(float elapsed)
{
    //计算剩余的时间
    float last = (d_lastTime - elapsed/1000.0f);
```

```
    //如果倒计时结束则激发结束事件
    if (last <= 0.00001f)
    {
        onTimerEnd();
        d_lastTime = 0.0f;
    }
    //否则更新倒计时器显示的文本
    else
    {
        d_lastTime = last;
        if (isVisible())
        {
            calcText(d_lastTime);
        }
    }
}
```

最后一个函数倒计时结束的事件处理。这个函数根据用户设计的激发次数产生对应的事件。

```
void TimerWindow::onTimerEnd()
{
    if (d_isTimerEnd == false)
    {
        WindowEventArgs e(this);
        fireEvent(EventTimerEnd, e, EventNamespace);
        d_curTimeEndCount--;
        if (d_curTimeEndCount  <= 0)
        {
            d_isTimerEnd = true;
        }
    }
    if (isVisible())
    {
        String pre = getProperty("TextPrefix");
        String suff = getUserString("TextSuffix_fal_auto_prop__");
        setText(pre + "N/A" + suff);
    }
}
```

定时器的逻辑比较简单不需要重载任何的鼠标和键盘消息就可以实现。

12.1.2 定时器属性

定时器的属性在第12.1.1节中以及做了简单介绍，本节详细介绍他们的实现，其实他们的实现非常类似，这里举两个例子。

属性的定义和实现非常简单，就是先从Property基类派生然后实现get函数set虚函数，最后一步在对应控件中申明静态变量，在控件实现文件中定义这些变量并且调用控件的函数
addProperty函数将属性添加到控件的属性集中。下面我们介绍TimerEndEventCount和TimeFormat两个属性。他们的声明如下。

```
//倒计时激发事件次数的属性
class TimerEndEventCount : public Property
{
public:
    TimerEndEventCount() : Property(
        "TimerEndEventCount",
        "设置或者获取，多重计时器结束事件",
```

```
            "1")
    {}
    String      get(const PropertyReceiver* receiver) const;
    void set(PropertyReceiver* receiver, const String& value);
};
//时间格式的属性
class TimeFormat : public Property
{
public:
    TimeFormat() : Property(
            "TimeFormat",
            "设置或者获取，定时器的字符串格式默认hh:mm:ss，设置的值是%d:%02d:%02d",
            "%d:%02d%02d")
    {}
    String      get(const PropertyReceiver* receiver) const;
    void set(PropertyReceiver* receiver, const String& value);
};
```

Property基类的三个参数分别是属性的名称，属性的帮助信息以及属性的初始值，其实还有一个默认的参数就是属性是否写入的XML的输出流中。get和set函数的格式和参数固定，每个属性都一样。每个控件的属性都必须在指定的命名空间中。

这两个类的函数实现。

```
String TimerEndEventCount::get(const PropertyReceiver* receiver) const
{
    return PropertyHelper::intToString(static_cast<const TimerWindow*>(receiver)->
getTimerEndCount());
}
void TimerEndEventCount::set(PropertyReceiver* receiver, const String& value)
{
    static_cast<TimerWindow*>(receiver)->      setTimerEndCount(PropertyHelper::stringToInt(value));
}

String TimeFormat::get(const PropertyReceiver* receiver) const
{
    return static_cast<const TimerWindow*>(receiver)->getTimerFormat();
}
void TimeFormat::set(PropertyReceiver* receiver, const String& value)
{
    static_cast<TimerWindow*>(receiver)->setTimerFormat(value.c_str());
}
```

实现一目了然，获取函数获取控件内部对应属性变量的值然后调用属性帮助函数转化为字符串返回。设置函数将字符串的属性值转化为对应格式的数据值并设置到控件中。这里说明为什么控件里的设置函数必须是常函数。因为获取函数将控件的指针转化为长指针导致的（指针指向的值不可用被修改）。

属性在控件中的声明第12.1.1节已经介绍了，下面只介绍添加属性到控件属性集的函数。

```
void TimerWindow::registerProperty()
{
    addProperty(&d_timeFormatProperty);
    addProperty(&d_timeEndEventCountProperty);
    addProperty(&d_timerValueProperty);
    addProperty(&d_textColorProperty);
    addProperty(&d_showBackgroundProperty);

}
```

这个函数也没有什么好说的，一切都在不言中。

另外一种属性的定义在外观实现中介绍，读者只需明白他们是可以当作在C++中定义的属性使用就可以了。

12.2 定时器渲染窗口的实现

定时器的渲染窗口实现也非常简单，只是简单的获取外观定义中的各种图像段，然后渲染之。我们实现了必须实现的render函数。

```
void FalgardTimerWindow::render()
{
    TimerWindow* w = (TimerWindow*)d_window;
    const WidgetLookFeel& wlf = getLookNFeel();
    //获取文本的显示颜色
    colour c = w->getTextColor();
    //获取是否要渲染背景的属性变量的值
    bool showBK = ("True" == w->getProperty("ShowTimerBackground"));
    //显示背景
    if(showBK)
    {
        Rect clipRect = getPixelRect();
        if(w->getFont() == NULL)
        {
            return;
        }
        //通过字体得到文本的渲染宽度
        int width = w->getFont()->getTextExtent(w->getText()) + 2;
        //渲染背景到指定的宽度，背景刷是单色图像
        wlf.getImagerySection("Background").render(*w,Rect(0, 0, width,
                        clipRect.getHeight()),0);
    }
    //显示文本
    wlf.getImagerySection("Text").render(*w, 0,&ColourRect(c,c,c,c));
}
```

实现简单清晰，读者可能不知道控件中的文本是怎么被渲染的呢？我们的文本段的渲染过程中会检查各种定义的字符串是否为空，如果都为空就使用附加这个窗口的窗口文本来显示。我们没有设置其他文本所以就使用控件的文本来显示了。CEGUI官方提供的例子中所有文本显示都是白色的，其实CEGUI原生支持彩色文本的渲染，甚至支持变色文本的渲染。只要ColourRect的各个顶点的颜色不一样就可以实现变色文本了，读者可以自己试验。

背景的描绘使用一个图像段来实现，在图像段中指定了渲染背景的图像。

另一个需要注意的地方可能就是渲染窗口的类厂名称TypeName的设置了。

12.3定时器外观的实现

外观和渲染窗口关系最为紧密。我们的外观定义在exampl.looknfeel文件中，我们所有例子的外观都将定义在这个文件中。外观的定义如下：

首先是四个属性的定义。

```
<PropertyDefinition name="VertLabelFormatting" initialValue="CentreAligned" />
<PropertyDefinition name="HorzLabelFormatting" initialValue="LeftAligned" />
<PropertyDefinition name="TextPrefix" initialValue="" redrawOnWrite="true" />
<PropertyDefinition name="TextSuffix" initialValue="" redrawOnWrite="true" />
```

读者应该对后两个属性比较熟悉，这里定义了他们，而且指定值改变后需要重绘窗口。另外两个属性描述文本的显示格式，分别表示竖直和水平方向的文本的显示格式。下面介绍文本段的定义。

```
<ImagerySection name="Text">
```

```
  <TextComponent>
    <Area>
      <Dim type="LeftEdge"><AbsoluteDim value="3" /></Dim>
      <Dim type="TopEdge"><AbsoluteDim value="3" /></Dim>
      <Dim type="Width"><UnifiedDim scale="1" offset="-3" type="Width" /></Dim>
      <Dim type="Height"><UnifiedDim scale="1" offset="-3" type="Height" /></Dim>
    </Area>
    <VertFormatProperty name="VertLabelFormatting" />
    <HorzFormatProperty name="HorzLabelFormatting" />
  </TextComponent>
</ImagerySection>
```

这个文本段的竖直和水平显示格式是在两个属性中获取，而这两个属性恰恰是我们刚才定义的属性。它的显示区域是控件各向向内偏移3像素。这些值可以更改，如果读者认为显示区域就是整个控件当然也可以了。

最后一个图像段，背景刷图像段。

```
<ImagerySection name="Background">
 <ImageryComponent>
   <Area>
     <Dim type="LeftEdge"><AbsoluteDim value="3" /></Dim>
      <Dim type="TopEdge"><AbsoluteDim value="3" /></Dim>
      <Dim type="Width"><UnifiedDim scale="1" offset="-3" type="Width" /></Dim>
      <Dim type="Height"><UnifiedDim scale="1" offset="-3" type="Height" /></Dim>
   </Area>
   <Image imageset="Vanilla-Images" image="GenericBrush" />
   <VertFormat type="Stretched" />
   <HorzFormat type="Stretched" />
 </ImageryComponent>
</ImagerySection>
```

这个图像段的实现和文本段非常类似只是它指定了渲染的图像，并且非常重要的一点是它指定了图像的水平和竖直显示格式都是拉伸模式，如果不指定这个模式则只能看到这个图像大小的背景。

外观的名称定义如下：

```
<WidgetLook name="Vanilla/TimerWindow">
</WidgetLook>
```

这个名称会在模式文件中用到。下一小节将详细介绍。

12.4 定时器扩展到CEGUI中

有了前3节的介绍一个定时器控件已经接近完工了。这一节介绍如何将这个控件添加到CEGUI中。在第11章节我们介绍了，如何将控件添加到CEGUI中的方法。但本章不使用那种需要修改CEGUI源文件的方法，而是采用向CEGUI控件（窗口）类厂注册新的控件和向CEGUI渲染窗口类厂注册新的控件的方法。

控件类厂的声明，我们使用CEGUI提供的简单宏来实现，具体如下。

```
CEGUI_DECLARE_WINDOW_FACTORY(TimerWindow);
```

实现同样使用一个宏来实现。

```
CEGUI_DEFINE_WINDOW_FACTORY(TimerWindow)
```

这两个宏在前文都已经详细介绍过。读者只要知道如何使用便可以了。

定义好了控件的类厂，下一步就是将它注册到CEGUI的类厂管理器中。我们提供了一个函数来完成这个工作。

```
void CEGUISample::RegisterClassFactory()
{
    //注册控件类厂
    WindowFactoryManager::getSingleton().addFactory(&
    (CEGUI_WINDOW_FACTORY(TimerWindow)));
```

```
    //注册渲染窗口类厂
    CEGUI::WindowRendererManager& wfm =          CEGUI::WindowRendererManager::getSingleton();
    wfm.addFactory( &s_FalgardTimerWindowWRFactory);
}
```

添加控件类厂到类厂管理器，需要借助宏CEGUI_WINDOW_FACTORY来实现。这里还实现了渲染窗口类厂的注册工作。

渲染窗口类厂没有声明的宏只有定义的宏，如下所示：

```
CEGUI_DEFINE_WR_FACTORY( FalgardTimerWindow );
```

这个宏会定义一个渲染窗口的类厂，并且定义一个静态的渲染窗口类厂的实例。这个宏的定义如下所示：

```
#define CEGUI_DEFINE_WR_FACTORY( className )\
namespace CEGUI {\
class className ## WRFactory : public WindowRendererFactory\
{\
public:\
   className ## WRFactory(void) : WindowRendererFactory(className::TypeName) { }\
   WindowRenderer* create(void)\
   { return new className(className::TypeName); }\
   void destroy(WindowRenderer* wr)\
   { delete wr; }\
};\
}\
static CEGUI::className ## WRFactory  s_ ## className ## WRFactory;
```

我们在RegisterClassFactory函数中使用的渲染窗口类厂的变量就是这个宏定义的。还有一个重要的问题就是注册函数在什么时候调用。我们知道在System创建完成以后就会创建CEGUI系统中所有的单件类，两个类厂管理器也属于单件类，所以只要System类创建以后就可以调用这个函数了。

代码部分完成上面所述的内容就已经把这个控件加入到CEGUI中了，但是还没有定义模式文件中的类型映射，还是不能创建定时器控件窗口。下面我们在Scheme文件中添加了控件的映射定义。我们程序框架使用的是VanillaSkin.scheme文件定义的模式，所以我们在这个文件中添加了如下映射。

```
<FalagardMapping WindowType="Vanilla/TimerWindow" TargetType="CEGUI/TimerWindow" Renderer="Falagard/TimerWindow"
LookNFeel="Vanilla/TimerWindow" />
```

这个映射定义了一个Vanilla/TimerWindow外部类型的窗口，可以使用它来创建窗口。

此外我们新添加了一个外观定义的文件，所以也要在模式文件中有所表现。

```
<LookNFeel Filename="example.looknfeel" />
```

添加这行以后我们的外观被加载了。

有了前4节的介绍我们可以创建定时器窗口了。下一节将详细介绍如何使用这个控件。其实它的使用和使用其他CEGUI官方控件一样。

12.5 控件的使用

我们提供了两种使用这个控件的方法，一种是在CEGUI的布局文件中定义一个子窗口，这个子窗口是定时器控件。另一种方法是在C++源代码里添加定时器控件的创建和使用代码。

我们选择例子程序使用到的VanillaWindow.layout文件，并在合适的位置添加了如下一段代码。

```
<Window Type="Vanilla/TimerWindow" Name="Vanilla/Console/Timer">
    <Property Name="AlwaysOnTop" Value="True" />
    <Property Name="TextColor" Value="FFFFFF00" />
    <Property Name="TimerValue" Value="40" />
    <Property Name="ShowTimerBackground" Value="False" />
    <Property Name="UnifiedMaxSize" Value="{{1,0},{1,0}}" />
    <Property Name="UnifiedPosition" Value="{{0.0,20},{0.5,0}}" />
```

```
    <Property Name="UnifiedSize" Value="{{1,0},{0,30}}" />
    <Property Name="TextPrefix" Value="After " />
    <Property Name="TextSuffix" Value=" The Window Will Be Closed" />
</Window>
```

布局文件可以使用CEGUI官方的布局编辑器来编辑，但我们这个似乎不能直接使用，因为官方编辑器使用的动态库中没有我们定时器窗口的实现。如果要使用那个编辑器还需要将我们的控件添加到CEGUI的工程中，然后按照第11章介绍的方法注册。生成动态库后替换官方编辑器使用的动态库。这样就可以使用官方编辑器了。

手动调节窗口的位置需要读者完全理解CEGUI的坐标系统才行。作者就是手动调整的，读者可以自己试试手动调整，看看自己有没有掌握CEGUI的坐标系统。

这段布局定义中设置了我们定义的属性，两种属性的设置方法完全一样。使用的时候可以不区分他们。

我们注册了定时器的结束函数并且关闭的定时器窗口的父窗口。首先注册结束事件的处理函数。

```
Window* pWinTimer = winMgr.getWindow("Demo_NamespaceViewer_Timer");
pWinTimer->subscribeEvent(TimerWindow::EventTimerEnd, Event::Subscriber(&UILogic::handleCloseParent, this));
```

然后实现关闭父窗口的函数。

```
bool UILogic::handleCloseParent (const EventArgs& args)
{
    Window* pWinTimer =        WindowManager::getSingleton().getWindow("Demo/NamespaceViewer");
    if (pWinTimer)
    {
        pWinTimer->hide();
    }
    return true;
}
```

在代码中我们创建了背景定时器，实现代码如下。

```
Window* pTimer = winMgr.createWindow("Vanilla/TimerWindow","Timer");
if (pTimer)
{
    pTimer->setArea(UVector2(UDim(0, 50), UDim(1, -200)),
    UVector2(UDim(0,300), UDim(0,30)));
    pTimer->setProperty("TimerValue", "3700");
    pTimer->setProperty("TextColor", "FF00FF00");
    pTimer->setProperty("TextPrefix", "This is a background Timer ");
    pTimer->show();
    background->addChildWindow(pTimer);
}
```

实现也非常简单，通过窗口管理器创建对应的窗口，然后设置窗口的各种属性，显示它，最后添加到根窗口。

如果读者发现创建了一个窗口后这个窗口没有显示出来，最可能的问题是位置和大小设置的不对。父窗口没有设置或者设置的不对导致的。没有显示这个窗口（默认创建后是显示的）。

本章的源代码在光盘上的例子目录里第12章子目录下。本书所有的例子都在例子目录下，以后就不在说明。

12.6 本章小节

本章主要实现了一个定时器控件，详细介绍了这个控件的实现步骤，并且介绍了如何使用这个控件。读者可以打开例子程序查看具体的实现。

本章习题：

1.认真学习这个控件的实现，自己创建一个新的定时器窗口，实现倒计时结束后关闭整个例子程序。

2.自己在CEGUI的代码中实现定时器控件，要求自己实现最好不要使用作者的实现。并且编译好动态库后替换官方编辑器的动态库，尝试是否能在编辑器中编辑定时器窗口。

第13章 中文输入

CEGUI可以显示中文，前文已经简单的介绍过。哪么如何在CEGUI中输入中文呢？计算机原生支持英文的输入，但要输入其他的文字则需要输入法IME（Input Method Editor）的支持。我们前文已经介绍过CEGUI的String类其实保存的是Unicode字符串。所以CEGUI其实是可以支持任何字符的显示的，只要有对应的字体支持就可以。既然CEGUI支持中文的显示，哪么其实输入中文已经支持了。只不过我们需要输入Unicode字符到CEGUI系统中而已。CEGUI注入的字符本来都是Unicode的字符。为什么英文字符可以直接注入也能显示呢？英文的本地编码（ASCII编码）字符与它在Unicode中的编码完全相同。所以我们注入ASCII编码的英文也可以正常的显示。但是我们的中文编码GBK等是多字节的编码，它和Unicode编码是不同的，所以就需要获取多字节编码的对应Unicode字符然后在注入到CEGUI中。当然还要有对应的中文字体支持才能正确的显示。我们的中文字体采用隶书字体，笔者是从Window系统的字体文件夹中拷贝出来的。

在介绍CEGUI中文输入之前我们首先简单介绍IME。以便读者可以理解Window是如何支持非英文的输入的。

13.1 IME简介

什么是IME呢？IME全称Input Method Editor，输入法编辑器的意思。其实就是读者熟悉的输入。现在比较流行的有搜狗输入法，紫光，五笔，智能ABC等。作者以前使用微软拼音输入法比较多，但最近发现搜狗输入法非常智能。输入常常只需要输入想输入的词的第一个字母就可以第一个输入这个词。

相信读者对输入法应该非常的熟悉了。中文的输入有几个步骤，首先启用一种支持中文的输入法。然后就可以通过这种输入法支持的输入方法通过输入英文或者数字来最终输入中文了。输入法涉及到2个窗口，一个是输入法窗口，它包含了输入的状态，名称，设置等等功能。另一个是单词选择窗口，当用户输入的拼音（这里以拼音编码的中文输入法为例）对应多个字或者词的时候需要有个字符选则窗口来供用户选择。

游戏里往往都需要输入法的编程，当然不是需要游戏自己编写一个输入法编辑器。而是一般游戏都需要自己实现选择文本的窗口，这样才能符合游戏的风格。这就要涉及到一小部分的IME的编程了。本节主要介绍，如果希望自己实现一个输入法的选择窗口需要做些什么。第14章还要实现一个输入法的选择窗口的控件，并且完整的实现一个输入法的界面。

13.1.1 输入法的Window消息介绍

输入法有很多消息，下面分别介绍。

消息WM_IME_CHAR，当输入法混合字符串完成后，就是说输入法已经确定用户已经输入一些字符的时候会发送这个消息给窗口。这个消息wParam参数包含的输入字符的编码，如果窗口是Unicode窗口则发送的是Unicode编码的字符，如果不是则发送的是本地代码页的字符编码。（中文就是GBK编码）。这个消息和WM_CHAR消息比较类似，如果窗口不是Unicode窗口默认处理函数DefWindowProc会将这个字符的两个字节分成两个字符生成
两个WM_CHAR消息，每个消息对应字符的一个字节。

消息WM_IME_COMPOSITION，就是我们说的混合字符的消息。这个消息的lParam参数值代表许多种状态。其中比较重要的有GCS_RESULTSTR，表示一次输入已经完成，可以获取结果字符串了，这时候可以调用Imm函数来获取这个字符串。GCS_CURSORPOS代表当前输入法光标的位置。GCS_COMPSTR代表目前用户输入的混合字符串。比如输入"ni"希望得到"你"这里的ni就是GCS_COMPSTR状态可以获取的字符。而GCS_RESULTSTR状态获取的字符则是"你"。

消息WM_IME_ENDCOMPOSITION会在一次混合结束后发送的，可以用来确定隐藏混合字符的窗口。

消息WM_IME_NOTIFY会在IME的各种状态改变的时候发送，比如输入法的全角和半角状态，进入选字状态，退出选字状态，关闭选字表等。它的wParam参数的值代表着这些不同的状态。

消息WM_IME_STARTCOMPOSITION会在混合开始的时候被发送。收到这个消息时可以准备显示选词窗口了。

这里只介绍了一部分我们会用到的IME消息，其他用不到的我们就不在介绍，有兴趣的读者可以详细阅读MSDN的相关部分。

还有一个不是IME消息，但它和IME有关，这个消息就是WM_INPUTLANGCHANGE输入法改变的消息。要处理这个消息获取新的输入法的信息，并且需要阻止默认窗口过程处理这个函数。

如果窗口不处理这些消息，Window默认窗口过程DefWindowProc函数会处理他们。他会负责调用输入法的默认选字窗口，显示用户的选字并最终生成一些WM_IME_CHAR消息。我们要做的就是截获其中一些消息，然后获取其中的选字列表和当前的输入编码（比如前面说过的"ni"）。

总结，我们需要处理的消息有WM_INPUTLANGCHANGE，WM_IME_STARTCOMPOSITION，WM_IME_ENDCOMPOSITION，WM_IME_NOTIFY和WM_IME_COMPOSITION这5个消息，因为这些消息没有传递给默认窗口过程因此并不会产生WM_IME_CHAR消息。如果用户不需要实现选字窗口，可以通过截获WM_CHAR或者WM_IME_CHAR消息，然后找到对应的Unicode编码并注入到CEGUI系统也可以实现中文输入的支持。但似乎这两个函数并不好实现，更好的方法是截获WM_IME_COMPOSITION消息，然后获取最终的混合后的Unicode编码的字符串，然后一个字符一个字符的注入到CEGUI系统中。典型的代码如下所示：

```
static wchar_t buf[1024];
if (lParam & GCS_RESULTSTR)
{
    //获取IME句柄
    HIMC hIMC = ImmGetContext(g_mainWnd);
    //获取Unicode结果字符串的长度，这个长度怎么也不会比1024还长
    LONG buflen = ImmGetCompositionStringW(hIMC,GCS_RESULTSTR,NULL,0);
    //重置字符串长度的缓冲区为0，否则会出现先前的字符
    memset(buf,0, buflen*sizeof(wchar_t));
    //获取Unicode结果字符串
    ImmGetCompositionStringW(hIMC,GCS_RESULTSTR,buf,buflen);
    //逐个字符注入到CEGUI系统中
    for (int i=0; i<buflen; i++)
    {
        System::getSingleton().injectChar((CEGUI::utf32)buf[i]);
    }
    //释放IME句柄
    ImmReleaseContext(g_mainWnd, hIMC);
}
```

上面的代码通过两个Unicode函数就获得了Unicode的字符串。然后逐个注入到系统中。

下一节详细介绍需要用到的输入法相关的函数。

13.1.2 输入法函数介绍

在截获了IME的消息后可以通过一些以Imm开头的函数来获取当前窗口的输入法的各种信息。

- ImmGetContext函数，这个函数获取指定窗口的IME的句柄（或者叫环境）。其他的Imm函数都需要通过这个环境来获取输入法在当前窗口的信息。
- GetKeyboardLayout函数，获取某个线程当前的激活的键盘布局。传递0作为参数是获取当前线程激活的句柄。
- ActivateKeyboardLayout函数，激活某一个键盘布局作为当前线程的键盘布局。
- ImmIsIME函数，判断一个键盘布局是不是具有输入法。
- ImmEscape函数，获取输入法的各种信息，可以通过这个函数获取输入法的名称。
- ImmGetConversionStatus函数，获取混合的各种字符串。
- ImmReleaseContext函数，释放指定的IME环境。与ImmGetContext配对使用。
- ImmGetConversionStatus函数，获取某个输入法的一些状态信息，比如半角，全角信息。
- ImmGetCandidateList函数，获取混合的字符列表。

我们用到的函数大概就有这些。这些函数的详细使用方法读者可以查阅MSDN或者查看我们的源代码。

这些函数的使用一般是在截获对应的消息后，调用他们获取或者设置输入法的字符串或者状态等。

13.2 CEGUI中文输入支持

前文已经说过，如果只是希望支持CEGUI的中文输入其实非常简单，只要在游戏窗口过程中加入第13.1.1节中指定的代码就可以实现。（本章就是暂时这样实现的）但是我们希望自己定义一个选词窗口，因此必须做一些其他的处理工作。

要显示选词列表最重要的是获取选词列表，然后显示它。另外一般选词窗口需要显示当前的输入名称以及各种种状态。本章代码中定义了一个新类IME，它提供了对选词列表的
支持。下面我们着重介绍这个类，首先还是介绍它的成员变量。

```
wchar_t          d_imeName[64];          //输入法的名称
BOOL             d_bAlphaNumeric;        //英文模式
BOOL             d_bSharp;               //全角标志
BOOL             d_bSymbol;              //中文标点标志;
BYTE*            d_bufCandidate ;        //选字缓冲
DWORD            d_candidateLength;      //缓冲区大小
LONG             d_cursorPos;            //当前光标的位置
wchar_t          *d_pResultStr;          //结果字符串缓冲
LONG             d_ResultStrBuflen;      //这个缓冲的长度
wchar_t          *d_pCompStr;            //编码字符串缓冲
LONG             d_CompStrBuflen;        //这个缓冲的大小
HWND             d_hWnd;                 //对应的窗口
std::vector<STRING> d_candidateArray;    //选词列表
```

本节的字符串变量都是wchar_t类型的，就是说他们保存的都是Unicode数据，这是为了方便注入到CEGUI而设计的。选字缓冲是一个结构体，所以这里设计成了BYTE类型的，但在获取选词列表的时候做了wchar_t类型的转化。STRING类定义为std::wstring它保存的是Unicode字符。

成员函数有两个比较重要，他们是MessageProc用来处理游戏窗口过程中消息处理，另一个其实也是处理消息的只不过它处理的消息WM_IME_NOTIFY，这个消息的处理比较复杂所以单独写了一个函数OnNotify函数来处理。下面着重介绍这两个函数。MessageProc函数我们按照每个消息的处理单独讲解。第一个消息输入法改变的处理。

```
case WM_INPUTLANGCHANGE:
{
    //获取当前激活的键盘布局
    HKL hKL = GetKeyboardLayout( 0 );
    //当前激活的键盘布局是否有输入法窗口
    if(ImmIsIME(hKL))
    {
        //获取环境
        HIMC hIMC = ImmGetContext(d_hWnd);
        //获取输入法的名称
        if(!ImmEscapeW(hKL,hIMC,IME_ESC_IME_NAME,d_imeName))
        {
            //出错的话激活下一个键盘布局
            ActivateKeyboardLayout((HKL)HKL_NEXT,0);
            break;
        }
        //更新各个输入法内部状态
        DWORD dwConversion, dwSentence;
        ImmGetConversionStatus(hIMC,&dwConversion,&dwSentence);
        if(dwConversion & IME_CMODE_NATIVE) d_bAlphaNumeric = FALSE;
        else d_bAlphaNumeric = TRUE;
        if(dwConversion & IME_CMODE_FULLSHAPE) d_bSharp = TRUE;
```

```
            else d_bSharp = FALSE;
            if(dwConversion & IME_CMODE_SYMBOL) d_bSymbol = TRUE;
            else d_bSymbol = FALSE;
            ImmReleaseContext(d_hWnd,hIMC);
        }
        else
        {
            d_imeName[0] = d_imeName[1] =0;
        }
        //激发输入法改变的事件
        onInputMethodChange();
    }
    break;
```

下面两个消息，只是简单的激发了对应的事件。

```
    case WM_IME_STARTCOMPOSITION:
        onCompositionBegin();
        break;
    case WM_IME_ENDCOMPOSITION:
        onCompositionEnd();
        break;
```

使用这个类的其他模块可能希望截获这两个事件，因此我们提供了响应函数。

消息WM_IME_COMPOSITION包含许多子状态，我们分别介绍他们，下面是这个消息处理的公共部分获取当前游戏窗口的输入法环境。

```
    case WM_IME_COMPOSITION:
    {
        HIMC hIMC = ImmGetContext(d_hWnd);
        ...
        ImmReleaseContext(d_hWnd,hIMC);
        onCompositionStr((LONG)lParam);
    }
    break;
```

第一个子状态，获取输入的编码的字符串获取。使用Unicode格式的函数获取的结果就是Unicode的字符串结果。

```
    if(lParam & GCS_COMPSTR)
    {
        LONG buflen = ImmGetCompositionStringW(hIMC,GCS_COMPSTR,NULL,0);
        if(buflen > 0)
        {
            buflen += sizeof(WCHAR);
            if(buflen > d_CompStrBuflen)
            {
                if(d_pCompStr)
                {
                    delete d_pCompStr;
                }
                d_pCompStr = new wchar_t [buflen];
                    d_CompStrBuflen = buflen;
            }
            memset(d_pCompStr,0,buflen);
            ImmGetCompositionStringW(hIMC,GCS_COMPSTR,d_pCompStr,buflen);
        }
        else if(d_pCompStr)
```

```
    {
        d_pCompStr[0] = d_pCompStr[1]=0;
    }
}
```

第二个子状态，获取结果字符串的状态。同理这个函数获取的也是Unicode字符串。

```
if(lParam & GCS_RESULTSTR)
{
    LONG buflen = ImmGetCompositionStringW(hIMC,GCS_RESULTSTR,NULL,0);
    if(buflen > 0)
    {
        buflen += sizeof(wchar_t);
        if(buflen > d_ResultStrBuflen)
        {
            if(d_pResultStr)
                {
                    delete[] d_pResultStr;
                }
            d_pResultStr = new wchar_t [buflen];
            d_ResultStrBuflen = buflen;
        }
        memset(d_pResultStr,0, buflen);
        ImmGetCompositionStringW(hIMC,GCS_RESULTSTR,d_pResultStr,buflen);
    }
}
```

第三个状态，获取当前光标的位置。

```
if(lParam & GCS_CURSORPOS)
{
    d_cursorPos = ImmGetCompositionStringW(hIMC,GCS_CURSORPOS,NULL,0);
}
```

下面介绍消息WM_IME_NOTIFY这个消息的处理比较复杂我们专门提供了一个函数来处理。

```
case WM_IME_NOTIFY:
{
    return OnNotify(wParam,lParam);
}
break;
```

下面介绍OnNotify函数。它获取选择文本，修改输入法的状态。这个函数的介绍也分为两部分，一部分是状态改变，另一部分是选词。首先介绍OnNotify的结构，主要功能都省略了，下面单独介绍。

```
BOOL    IME::OnNotify(WPARAM wParam, LPARAM lParam)
{
    switch(wParam)
    {
        //全角/半角，中/英文标点改变
        case IMN_SETCONVERSIONMODE:
            break;
        //进入选字状态
        case IMN_OPENCANDIDATE:
        //选字表翻页
        case IMN_CHANGECANDIDATE:
            break;
        //关闭选字表，清理选词的向量
        case IMN_CLOSECANDIDATE:
```

```
                {
                    ClearCandidateString();
                }
                break;
            case IMN_PRIVATE:
                {
                    if(lParam == 193) return FALSE;
                }
                break;
        }
        return TRUE;
}
```

下面这段代码获取当前输入法的各种状态。

```
case IMN_SETCONVERSIONMODE:
{
    HIMC hIMC = ImmGetContext(d_hWnd);
    DWORD dwConversion, dwSentence;
    ImmGetConversionStatus(hIMC,&dwConversion,&dwSentence);
    //英文模式的标志
    if(dwConversion == IME_CMODE_ALPHANUMERIC)
    {
        d_bAlphaNumeric = TRUE;
    }
    //全角半角标志
    if(dwConversion & IME_CMODE_FULLSHAPE)
    {
        d_bSharp = TRUE;
    }
    else d_bSharp = FALSE;
    //中文标点标志
    if(dwConversion & IME_CMODE_SYMBOL)
    {
        d_bSymbol = TRUE;
    }
    else d_bSymbol = FALSE;
    ImmReleaseContext(d_hWnd,hIMC);
    //激发状态改变的消息
    onIMEStatusChanged();
}
break;
```

下面这段代码获取选词列表。

```
case IMN_OPENCANDIDATE:
case IMN_CHANGECANDIDATE:
{
    //清理上次的选词列表
    ClearCandidateString();
    HIMC hIMC = ImmGetContext(d_hWnd);
    //获取需要的缓冲区大小
    DWORD buflen = ImmGetCandidateListW(hIMC,0,NULL,0) * sizeof(wchar_t);
    if(buflen)
    {
```

```
            //修正缓冲区到合适的大小
            if(buflen > d_candidateLength)
            {
                if(d_bufCandidate)
                {
                    delete[] d_bufCandidate;
                }
                d_bufCandidate = new BYTE [buflen];
                d_candidateLength = buflen;
            }
            //获取缓冲区的数据
                                ImmGetCandidateListW(hIMC,0,(LPCANDIDATELIST)d_bufCandidate,buflen);
            CANDIDATELIST *pList = (CANDIDATELIST*)d_bufCandidate;
            STRING wstr = L"";
            //循环生成每个选词行
            for(DWORD i = 0; (i < pList->dwCount - pList->dwSelection) && (i < pList->
                dwPageSize);  i ++)
            {
                wstr = L"";
                if(i >9) wstr = L"\0";
                //数字0
                else if(i == 9) wstr =  (wchar_t)(0x31);
                //只是数字1到9的Unicode编码生成
                else wstr = (wchar_t)(0x31 + i);
                wstr += L":";
                //这一句非常重要，注意指针的转换
                wstr += (wchar_t*)((char*)pList+pList->dwOffset[pList->dwSelection+i]);
                //添加到选词列表中
                d_candidateArray.push_back(wstr);
            }
        }
        ImmReleaseContext(d_hWnd,hIMC);
        //激发选词列表改变的事件
        onCandidateListChanged();
    }
    break;
```

选词列表的获取根据MSDN介绍CANDIDATELIST 结构的各个成员的含义编写。选词列表的每行词的格式是行号：这一行的词。比如0:你好。

有关Window的输入法消息的函数就介绍完了，这个类还有一些以on开头的事件函数，这些函数目前什么都不干，他们是为使用这个类提供的事件函数。使用者可以在这些函数中做一些处理来更新选词窗口的状态和选词列表等。这些工作将在下一章进行。

13.3 本章小结

第14章 IME选词控件

在游戏中经常可以看到输入法的选词界面。我想读者应该非常想知道如何实现它。这一章我们结合第13章实现一个输入法选词控件。他的行为和普通的输入法提供的界面非常类似。

第13章实现了输入法的各种事件的处理并激发对应的输入法事件。我们这一章主要是响应这些事件，并将对应的内容输入到我们这一章实现的选词控件中。本章先介绍输入法控件的实现，然后介绍如何使用这个控件。读者可以先运行这章的例子，以便更好的学习本章。

14.1 选词控件

选词控件由三部分构成，第一部分是读者输入的编码，第二部分是选词列表，第三部分是输入法的信息。第一和第三部分使用CEGUI的静态文本来实现。第二部分使用CEGUI的ListBox控件来实现。本书所有的例子都是基于Vanilla外观的，它定义在Vanilla.looknfeel和VanillaSkin.scheme等文件中。

这三部分的控件以子窗口的形式定义在外观文件中，在控件中需要获取这些子窗口，并且调用这些子窗口的函数来实现控件的功能。我们介绍一个子窗口的外观定义。本书例子的外观定义都在example.looknfeel文件中，本章控件的外观名称是Vanilla/IMEWindow。下面是外观中定义的第一部分的子窗口。

```
<Child  type="Vanilla/StaticText" nameSuffix="__auto_inputcode__">
  <Area>
    <Dim type="LeftEdge"><AbsoluteDim value="5" /></Dim>
    <Dim type="TopEdge"><AbsoluteDim value="3" /></Dim>
    <Dim type="Width"><UnifiedDim scale="1" offset="-10" type="Width" /></Dim>
    <Dim type="Height"><AbsoluteDim value="25" /></Dim>
  </Area>
  <VertFormatProperty name="VertLabelFormatting" />
  <HorzFormatProperty name="HorzLabelFormatting" />
  <Property name="Font" value="FZYT" />
</Child>
```

这个子窗口类型是静态文本框，名称后缀是__auto_inputcode__，位置在父窗口的左边5像素，上边3像素，宽度是父窗口的宽度减去10像素（两边边框各5像素），高度是25像素。指定的字体是方正姚体，这个字体是本章新加的一个字体。增加字体的方法读者应该知道了，如果还不知道，请参考datafiles/fonts目录下的字体文件定义，以及datafiles/schemes中VanillaSkin.scheme的定义。其他两个子窗口的外观定义类似。

哪么如何获取外观中定义的子窗口的指针呢？（获取指针后就可以操作这个控件了）在CEGUI中获取窗口的方法是通过窗口管理器通过窗口的名称来获取这个窗口。获取子窗口的函数必须在Window基类提供的一个虚函数initialiseComponents中。

```
d_wordList = (Listbox*)WindowManager::getSingleton().getWindow(getName() + "__auto_woldlist__");
d_inputCodeWindow = WindowManager::getSingleton().getWindow(getName() + "__auto_inputcode__");
d_imeNameWindow = WindowManager::getSingleton().getWindow(getName() + "__auto_imename__");
```

这段代码分别获取了第二部分，第一部分以及第三部分的子窗口。这个获取的顺序是无关紧要的。除了定义子窗口外观中还定义了几个属性，我们这里获取控件关心的三个属性，他们分别是选词ListBox控件的文本颜色，控件的默认宽度，控件的边框宽度。边框宽度是外观中计算出来的，这里获取后在控件中会用到。

```
d_wordTextColor = PropertyHelper::stringToColour(getProperty("WordListColor"));
d_maxWidth  =  PropertyHelper::stringToFloat(getProperty("DefaultWidth"));
d_borderWidth = PropertyHelper::stringToFloat(getProperty("BorderWidth"));
```

下面分别介绍控件提供的函数。对于玩家输入的英文编码和输入法的名称以及状态只需提供设置和获取函数就可以了。

```
//设置输入法名称
void IMEShowWindow::setInputName(const String& name)
{
    //保存在控件的变量中
    d_inputName = name;
```

```
    //设置对应子窗口的文本
    d_imeNameWindow->setText(name);
    //计算设置文本的宽度，如果宽度大于目前的宽度则调整控件的宽度
    if(d_imeNameWindow->getFont())
    {
        //这里通过字体获取设置文本的宽度，然后加上控件的边框宽度
        float w  = d_imeNameWindow->getFont()->getTextExtent(name) + d_borderWidth;
        //如果这个值大于目前最大的控件宽度则设置控件宽度为这个值
        if (w > d_maxWidth)
        {
            d_maxWidth = w;
            setWidth(cegui_absdim(d_maxWidth));
        }
    }
    //文本改变了请求重绘
    requestRedraw();
}
//设置输入的英文编码字符，处理类似setInputName
void IMEShowWindow::setInputCode(const String& code)
{
    d_inputCode = code;
    d_inputCodeWindow->setText(code);
    if(d_inputCodeWindow->getFont())
    {
        float w  = d_inputCodeWindow->getFont()->getTextExtent(code) + d_borderWidth;
        if (w > d_maxWidth)
        {
            d_maxWidth = w;
            setWidth(cegui_absdim(d_maxWidth));
        }
    }
    requestRedraw();
}
```

选词窗口的文本清除和设置。清除使用Listbox函数的清除函数，添加一个新词将创建一个文本ListBoxItem然后添加到Listbox子控件中。

```
//清除所有的列表子项
void IMEShowWindow::clearWordList()
{
    d_wordList->resetList();
    requestRedraw();
}
//添加一个列表项
void IMEShowWindow::addWord(const String& word)
{
    //创建一个文本子项
    ListboxTextItem* pItem = new ListboxTextItem(word);
    //设置文本子项的文本颜色
    pItem->setTextColours(d_wordTextColor);
    //添加到列表框中
    d_wordList->addItem(pItem);
    //计算文本宽度，并适时调整控件的宽度
```

```
    if(pItem->getFont())
    {
        float w  = pItem->getFont()->getTextExtent(word) + d_borderWidth;
        if (w > d_maxWidth)
        {
            d_maxWidth = w;
            setWidth(cegui_absdim(d_maxWidth));
        }
    }
    //请求重绘
    requestRedraw();
}
```

我们的控件会根据文本的宽度自己调整控件的宽度以适用窗口的宽度。除了这项功能外，选词窗口还需要根据输入框的位置来确定自己的位置，它的基本原理是获取屏幕区域，控件自己的区域以及目标输入框的区域，然后计算合适的区域保证整个控件都会被显示在屏幕上。下面介绍位置的控制函数。

```
void   IMEShowWindow::TraceWindow(Window* inputWindow)
{
    //如果传入的窗口是输入框
    if (inputWindow && inputWindow->testClassName("Editbox"))
    {
        d_traceWindow = inputWindow;
        //获取输入框在屏幕坐标系下的区域
        Rect rect = inputWindow->getPixelRect();
        //获取屏幕的区域大小
        Rect scrRect = System::getSingleton().getRenderer()->getRect();
        //获取控件自己在屏幕坐标系下的区域，这个区域没有经过父窗口的裁剪
        Rect selfRect = getUnclippedPixelRect();
        Rect resultRect ;
        //首先计算输入框左边加上控件的宽度，在加上控件留下的空白（10）
        resultRect.d_left = rect.d_left +  selfRect.getWidth() + 10;
        //其次计算输入框的高度减去控件的高度
        resultRect.d_top = rect.d_top -  selfRect.getHeight();
        //如果紧靠着输入框无法完全的显示控件则控件向输入法的左边移动
        if (resultRect.d_left >= scrRect.d_right)
        {
            resultRect.d_left = scrRect.d_right - selfRect.getWidth() -10;
        }
        //否则计算真正的输入法的控件位置
        else
        {
            resultRect.d_left = rect.d_left -10;
        }
        //如果控件无法在在输入框的上边完全显示则显示在输入框的下边
        if (resultRect.d_top < 0)
        {
            resultRect.d_top = rect.d_bottom;
        }
        //设置最终控件的高度和宽度
        resultRect.setWidth(selfRect.getWidth());
        resultRect.setHeight(selfRect.getHeight());
        //设置控件的区域，这个控件的父窗口占据这个屏幕，所以这个控件使用屏幕坐标
```

```
            setArea(URect(cegui_absdim(resultRect.d_left), cegui_absdim(resultRect.d_top),
            cegui_absdim(resultRect.d_right), cegui_absdim(resultRect.d_bottom)));
        }
}
```

这个函数的算法比较复杂，读者细细体会应该会明白。我们希望在输入法下次显示的时候，控件的宽度是默认的宽度。因为上次的输入过程可能导致控件的宽度过宽。所以我们设置输入法在显示的时候为默认宽度。在显示的时候CEGUI窗口基类会调用一个名叫onShown的虚函数，我们重载它实现我们的功能。

```
void IMEShowWindow::onShown(WindowEventArgs& e)
{
    //首先基类处理
    Window::onShown(e);
    //然后获取默认宽度值作为最大宽度
    d_maxWidth = PropertyHelper::stringToFloat(getProperty("DefaultWidth"));
    //最后设置控件的宽度为默认宽度
    setWidth(cegui_absdim(d_maxWidth));
}
```

读者可能对cegui_absdim宏不太熟悉，它设置一个绝对值作为一个UDim变量。还有一个类似的是cegui_reldim宏，它设置一个相对值作为UDim量。

控件的注册以及导出和第12章介绍的一样这里就不在介绍了。

14.2 选词控件的渲染

这个控件由于使用了三个子窗口来实现功能，所以它的渲染窗口实现非常简单，只是负责描绘背景。代码如下。

```
void FalgardIMEShowWindow::render()
{
    IMEShowWindow* w = (IMEShowWindow*)d_window;
    const WidgetLookFeel& wlf = getLookNFeel();
    wlf.getImagerySection("background").render(*w, 0);
}
```

读者可能非常奇怪，为什么就这么简单的描绘背景，就会描绘出这么复杂的选词窗口呢。因为我们这里只是负责渲染自己，子窗口的渲染在他们自己的对应渲染窗口中进行。他们的已经被CEGUI实现了，他们是比较复杂的。有兴趣的读者可以自己阅读相关的代码。

下面简单介绍选词控件的外观定义。（第14.1节已经介绍了一部分）。

我们的控件使用了属性的链接的定义。由于我们的需要通过设置控件的属性就能够设置子窗口的属性，所以我们需要定义属性链接。定义如下。

```
<PropertyLinkDefinition name="IMENameColors" widget="__auto_imename__" targetProperty="TextColours"
initialValue="tl:FFFFFF00 tr:FFFFFF00 bl:FF00FF00 br:FF00FF00" redrawOnWrite="true" />
<PropertyLinkDefinition name="InputNameColors" widget="__auto_inputcode__" targetProperty="TextColours"
initialValue="tl:FFFFFF00 tr:FFFFFF00 bl:FF00FFFF br:FF00FF00" redrawOnWrite="true" />
```

这是两个颜色的属性链接。控件中定义了一个名为IMENameColors的属性，它指向子窗口后缀名为"__auto_imename__"的子窗口中的TextColours属性。也就是说如果操作了控件的IMENameColors属性实际上是在操作这个子窗口中的TextColours属性。而这个属性正式子窗口文本的颜色矩形。我们说过CEGUI支持文本的过渡色。只要设置控件的这个属性就可以实现不同的过渡色。读者可以自己试验。tl表示左上，t表示top，l表示left，其他类似。指定redrawOnWrite为真，当这个值被修改后重绘控件。另外一个属性和我们介绍的非常类似，这里就不在介绍了。

背景图片的外观的定义是从外观Vanilla/Shared中的BackgroundOnly图像段中拷贝过来的。它定义了一个图像的框架作为背景。这个框架支持各向的拉伸操作。因此非常时候我们的控件，因为我们需要随时的调整控件的大小。

14.3 使用选词控件

控件设计好了，哪么如何使用他呢。我们说过选词控件需要响应第13章实现的IME消息。把IME类获取的各种文本添加到控件中。

首先，我们要定义IME的实例，并且处理各种IME消息。我们定义了一个名为g_IME的全局变量，并且在Windows窗口过程中调用这个实例的MessageProc函数来处理IME消息。定义和消息处理如下所示。

```
//全局变量的声明
IME g_IME;
//处理消息，如果处理的话就返回，不在被其他函数处理
if (g_IME.MessageProc(hwnd, uMsg, wParam, lParam))
{
    return 0;
}
```

其次，创建我们的选词控件，我们在程序里创建。代码如下。

```
//创建一个选词窗口
Window* pIME = winMgr.createWindow("Vanilla/ImeWindow", "IMEShow");
if (pIME)
{
    //设置其实位置，其实这一步也没什么用，控件或自己调整位置和大小
    pIME->setArea(UVector2(UDim(0, 150), UDim(1, -400)),
    UVector2(UDim(0,200), Dim(0,260)));
    //添加到背景窗口
    background->addChildWindow(pIME);
    //默认隐藏窗口
    pIME->hide();
    //保存到全局变量中，在IME类中使用
    g_IMEShower = (IMEShowWindow*)pIME;
}
```

再次，我们需要修改IME类，响应IME类提供的各种函数来控制选词窗口。

当开始文本混合的时候我们显示选词窗口。

```
void IME::onCompositionBegin()
{
    if (g_IMEShower == NULL)
    {
        return;
    }
    //显示窗口
    g_IMEShower->show();
    //获取当前的激活窗口
    CEGUI::Window* pWnd = CEGUI::System::getSingleton().getGUISheet()->
getActiveChild();
    //设置这个窗口为跟踪窗口
    g_IMEShower->TraceWindow(pWnd);
}
```

当混合结束后我们隐藏选词窗口。

```
void IME::onCompositionEnd()
{
    if (g_IMEShower == NULL)
    {
        return;
    }
```

```
    g_IMEShower->hide();
}
```

当在混合过程中的时候，我们响应各种操作。

```
void  IME::onCompositionStr(LONG flag)
{
    if (g_IMEShower == NULL)
    {
        return;
    }
    //如果混合结束后，我们注入结果字符串到CEGUI中
    if (flag & GCS_RESULTSTR)
    {
        for (int i=0; i<wcslen(d_pResultStr); i++)
        {
            CEGUI::System::getSingleton().injectChar(d_pResultStr[i]);
        }
    }
    //如果混合编码变化我们修改混合编码
    if (flag&GCS_COMPSTR )
    {
        if (wcslen(d_pCompStr) <= 0)
        {
            return;
        }
        g_IMEShower->setInputCode(ucs_to_utf8(d_pCompStr));
    }
    //暂不处理这个消息
    if (flag&GCS_CURSORPOS )
    {
    }

    获取激活的窗口
    CEGUI::Window* pWnd = CEGUI::System::getSingleton().getGUISheet()->
getActiveChild();
    //追踪这个窗口
    g_IMEShower->TraceWindow(pWnd);
}
```

当输入法的状态发生变化的时候。修改输入法的名称和全角半角状态。

```
void  IME::onIMEStatusChanged()
{
    if (g_IMEShower == NULL)
    {
        return;
    }
    STRING str = d_imeName ;
    str += L"(" ;
    str += (d_bSharp ? L"全角" : L"半角") ;
    str += L")";
    g_IMEShower->setInputName(ucs_to_utf8(str.c_str()));
    g_IMEShower->hide();
}
```

当选词列表发生变化的时候。

```
void IME::onCandidateListChanged()
{
    if (g_IMEShower == NULL)
    {
        return;
    }
    //先清空列表
    g_IMEShower->clearWordList();
    //然后获取选词窗口的文本，然后注入到CEGUI系统中
    for (int i=0; i<d_candidateArray.size(); i++)
    {
        g_IMEShower->addWord(ucs_to_utf8(d_candidateArray[i].c_str()));
    }
}
```

读者可能不明白ucs_to_utf8函数的作用。它将Unicode的字符串转化为utf8的字符串。我们说过String类接受的Unicode编码格式是UTF8格式。原始的Unicode编码不直接支持。读者可以写一个支持Unicode格式的构造函数，哪么这里就不需要这一步转化了。转化函数如下所示。

```
const CEGUI::utf8* ucs_to_utf8(const wchar_t* pucs)
{
    static char tmpbuf[1024];
    memset(tmpbuf, 0, sizeof(tmpbuf));
    WideCharToMultiByte(CP_UTF8, 0, pucs, (int)wcslen(pucs), tmpbuf, 1023,0,0);
    return (CEGUI::utf8*)tmpbuf;
}
```

这个函数调用了Window提供的转化函数将Unicode编码转化为UTF8编码。静态缓冲tmpbuf有1024个字节一般来说不可能超过它，所以不必检查缓冲区的溢出。本书的很多代码都是为例子程序而写，可能不是很注重效率以及安全性。读者在使用的时候需要自己考虑。

14.4 本章小结

本章介绍了选词控件的实现，并且介绍了如何使用这个控件。结合第13章介绍的输入法的支持，本章完整的实现了一个输入法的选词控件。读者可以打开例子程序体验，感觉一定很不错哦。

本章习题

第15章 CEGUI和脚本的交互

界面的修改是非常频繁的。因此如果在C++代码中实现界面逻辑，哪么将会频繁的编译程序，对于小工程也可以，但如果编译一个大的工程就比较费事了。使用脚本来实现界面逻辑是非常常见的。CEGUI也提供了对脚本的支持。应该说CEGUI应该是支持任何脚本的，因为它定义了脚本接口，只要实现了这个接口，不管什么脚本语言都可以。但CEGUI推荐使用Lua作为脚本语言。CEGUI实现了Lua的脚本接口，在三个工程中可以看到源代码，这三个工程分别是tolua++cegui，tolua++和CEGUILua。本书并不打算介绍这三个工程，因为直接使用Lua来编写脚本接口是非常繁琐而且不易理解。所以我们推荐使用Lua的各种包装工程。最著名的有LuaBind，LuaPlus等。LuaBind功能非常强大但是要依赖Boost库，所以不太适合介绍，因此我们使用LuaPlus来实现脚本模块。LuaPlus是Lua语言的面向对象的包装。它使用非常方便，而且功能也非常强大。本章首先介绍简单LuaPlus的使用，然后介绍如何使用LuaPlus来实现，最后通过脚本实现第14章的功能。

15.1 LuaPlus介绍

Lua使用起来有点类似汇编语言，使用比较晦涩。LuaPlus封装了Lua语言，提供了方便的方法来使用Lua的功能。在本书的随书光盘上LuaPlus目录下有LuaPlus的源代码，例子以及帮助文件。LuaPlus工程可以编译动态库，也可以编译成静态库，本书使用后者。虽然我们介绍的是LuaPlus但读者也必须对Lua有一定的了解。

LuaPlus有三个重要的类，第一个是LuaState类，它封装了Lua的lua_State的功能。另一个类LuaObject描述一类值。在脚本语言里是没有数据类型，所有的值都是一个类型。但是在C++中是有类型的。LuaStackObject和LuaStack这两个类操作Lua的堆栈。下面我们就分别简单的介绍如何使用这几个类。

15.1.1 LuaState

LuaState是C++格式的lua_State。LuaState通过内敛函数的方式封装大多数的存取lua_State的函数。Lua的功能是通过一系列的API来提供的，这些API都有一个共同的参数就是一个lua_State的指针。据LuaPlus开发者说，使用LuaPlus的效率甚至比用Lua来开发还要高。

- 开始LuaPlus程序

  LuaState* state = LuaState::Create();

  创建一个LuaState对象。销毁一个对象也非常简单，调用下面的静态函数。

  LuaState::Destroy(state);

  可以使用另外两个类LuaStateAuto和LuaStateOwner。这两个类自动管理LuaState的创建和销毁。

  LuaStateAuto支持LuaState的自动删除。

```
//自动设置LuaState为NULL
LuaStateAuto stateOwner;
//创建新对象
stateOwner = LuaState::Create();
//使用功能
stateOwner->PushNil();
//超出作用域后自动删除
```

  LuaStateOwner派生自LuaStateAuto它自动创建LuaState并且超出自己的生存期的时候自动删除LuaState对象。典型使用如下。

```
//自动调用LuaState::Create()
LuaStateOwner stateOwner;
//这里可以使用Lua的功能
stateOwner->PushNil();
//超出作用域自动调用删除函数
```

  这种封装可以称作智能指针，比较好理解。

- LuaState封装的各种功能。

1.执行Lua脚本。分为从内存执行和从文件执行两种。

（1）函数DoFile(const char* fileName)，从文件加载脚本文件并执行之。它镜像Lua中函数lua_dofile() 的功能。

（2）函数DoString()和DoBuffer()从内存执行一段Lua脚本。

（3）函数LoadString()和LoadBuffer()加载一段Lua脚本（内存中）但并不执行，需要调用函数Call()或者PCall()才会执行。

（4）对应宽字符版的函数DoWString(), LoadWString()等类似的命名方法。

2.全局变量的存取。

（1）获取全局变量的方法。我们说过LuaObject封装Lua中的各种变量，以下假设state是一个LuaState对象的指针。

方法1：通过函数GetGlobal获取指定的全局对象。

```
LuaObject obj = state->GetGlobal("MyGlobalVariable");
```

方法2：获取全局变量表，然后通过[]操作符获取全局变量的值。

```
LuaObject globalsObj = state->GetGlobals();
LuaObject obj = globalsObj["MyGlobalVariable"];
```

方法3：在方法2的基础上不开单独的对象而是直接通过操作符获取。

```
LuaObject obj = state->GetGlobals()["MyGlobalVariable"];
```

（2)设置全局变量的方法。获取全局变量表然后设置全局变量值。

方法1：通过获取全局LuaObject对象，然后通过这个对象创建对象（一般是表）。

```
LuaObject globalsObj = state->GetGlobals();
LuaObject tbl = globalsObj .CreateTable("God");
```

方法2：将一个LuaObject对象设置成全局对象。

```
LuaObject obj;
LuaObject globalsObj = state->GetGlobals();
globalsObj.SetObject("globalObj",obj);
```

严格的说全局变量的设置不是LuaState的封装的功能而是LuaObject封装的功能。因为Lua中的全局变量是保存在一个叫做_G的Lua表中。所以在LuaPlus中全局变量是一个LuaObject对象。

3.堆栈存取函数。

（1）向堆栈上设置值的函数。

PushNil() - 映射 lua_pushnil().

PushInteger(int n) - 映射 lua_pushinteger().

PushNumber(lua_Number n) - 映射 lua_pushnumber().

PushLString(const char* s, size_t len) - 映射 lua_pushlstring().

PushLWString(const lua_WChar* s, size_t len) - 映射 lua_pushlwstring().

PushString(const char* s) - 映射 lua_pushstring().

PushWString(const lua_WChar* s) - 映射 lua_pushwstring().

PushBoolean(bool value) - 映射 lua_pushboolean().

PushLightUserData(void* p) - 映射 lua_pushlightuserdata().

这些函数直接支持C++中的类型，使用很方便。这些函数用的最多的是当我们编写给Lua调用的C++函数的时候。他们的功能可以从他们的名称得知，比如PushNumber函数将一个整形值压到当前堆栈上。

这个类还包装了一些其他的Lua操作堆栈的函数，但并不推荐使用这里就不在介绍了。

一个LuaState代表一个Lua语言的执行环境，一个程序里可以同时存在多个Lua的环境（如果需要的话），本书的例子中只使用一个Lua环境。

下一节介绍非常重要的类LuaObject它包装了Lua的各种类型的值。

15.1.2 LuaObject

LuaObject提供了高于Lua的高层次的函数来操作操作Lua的数值，使用者甚至可以不知道Lua有堆栈。LuaObject提供的操作函数大部分根本就不涉及堆栈的概念。

- 判断一个值的类型

这类函数非常多，可以根据他们的名称来理解他们的功能。这里举几个典型的例子。

（1）函数IsTable()判断这个对象是不是表。

（2）函数IsCFunction()判断这个对象是不是一个C函数对象。

（3）函数IsWString() 判断这个对象是不是一个宽字符的字符串对象。

（4）函数IsConvertibleToInteger() 判断这个对象是否可以转化为整形。

（5）函数IsUserData() 判断这个对象是否是一个用户数据。

如果读者对Lua的各种数据类型还不太了解，请参考Lua的相关教程。笔者推荐一本很好的教程<<Programming in Lua>>，这本书有电子版，读者可以在网上下载。

- 获取对象的值

这类函数和Lua在C++中的类型一一对应，大概有7种左右。这里举几个例子。

（1）函数int GetInteger() 获取整形值。

（2）函数float GetFloat() 获取浮点型值。

（3）函数const char* GetString()获取字符串值。

（4）函数void* GetUserData()获取用户数据值。

这类函数也非常好理解，需要注意的是在获取这种值之前读者必须知道它一定是这种类型的值或者可以转化为这种类型的值。否则调用会导致Lua错误。

- 设置对象的值

一共有如下一些函数。他们设置不同的值到LuaObject中。

```
AssignNil(LuaState* state)
AssignBoolean(LuaState* state, bool value)
AssignInteger(LuaState* state, int value)
AssignNumber(LuaState* state, double value)
AssignString(LuaState* state, const char* value)
AssignWString(LuaState* state, const lua_WChar* value)
AssignUserData(LuaState* state, void* value)
AssignLightUserData(LuaState* state, void* value)
AssignObject(LuaState* state, LuaObject& value)
AssignNewTable(LuaState* state, int narray = 0, int nhash = 0)
```

我们可以定义一个LuaObject然后通过上面的函数设置为不同的值，然后在设置为全局变量。具体方法如15.1.1中所述。

- 表对象的处理

为什么要单独将表对象列出来呢？因为它是Lua中唯一的数据结构，而且它的功能非常强大，用它可以实现各种面向对象的支持。它可以实现复杂的算法，比如实现链表等结构。而且本书通过它来保存个种函数，用它来描述窗口类的继承关系。

1.创建一个表，有两种方法，如下所述。

方法1：通过全局表对象来创建一个在全局表中的表对象。

```
LuaObject globalsObj = state->GetGlobals();
LuaObject myArrayOfStuffTableObj = globalsObj.CreateTable("MyArrayOfStuff");
```

方法2：创建一个独立的表对象，首先声明一个对象，然后通过先前介绍的给LuaObject对象赋值的方法创建一个空表。

LuaObject aStandaloneTableObj;
aStandaloneTableObj.AssignNewTable(state);

2.获取表中元素的个数。

有两个函数可用，第一个获取表中所有元素的个数，第二个获取表中从1开始顺序排开的元素的个数。

（1）函数GetN()获取表中所有元素的个数，它使用table.getn()函数返回结果。

（2）函数GetCount() 获取从1开始的连续下标的元素的个数。

这里说的元素其实也是LuaObject对象。

3.表数据获取

有两种方法，一是通过操作符[]获取，另一种方法是通过一些函数获取。

下面是第一种方法的示例：

```
state->DoString("MyTable = { WindSpeed = 50, Value = 'Hello', 10, 20, 30 }");
LuaObject myTableObj = state->GetGlobals()["MyTable"];
LuaObject windSpeedObj = myTableObj["WindSpeed"];
LuaObject is20Obj = myTableObj[2];
LuaObject keyObj;
keyObj.AssignString(state, "Value");
LuaObject valueObj = myTableObj[keyObj];
```

第二种方法是通过以下函数来获取。

```
GetByName(const char* key)
GetByIndex(int index)
GetByObject(const LuaObject& obj)
GetByObject(const LuaStackObject& obj)
```

4.表数据的设置

可以通过以下函数来设置表对象数据。

```
SetNil(key)
SetBoolean(key, bool value)
SetInteger(key, int value)
SetNumber(key, double value)
SetString(key, const char* value)
SetWString(key, const lua_WChar* value)
SetUserData(key, void* value)
SetLightUserData(key, void* value)
SetObject(key, LuaObject& value)
```

这里的key是保存到这个表中的对象名称，一般是字符串，整形值（索引）或者另一个对象。每个函数设置一种数据类型。

5.表中注册导出函数

向表中注册C++函数，可以在脚本里调用他们。从而实现脚本和C++代码的交互。有了这些导出函数脚本可以方便的调用各种控件的函数。

LuaPlus支持三种函数的导出，第一种静态或者全局函数。这类函数没有数据依赖，他们可以认为是完全的和数据独立，数据只通过参数和返回值和他们交互。第二种是类的成员函数，LuaPlus支持直接将类成员函数导出。这种函数对数据有依赖性，因为这些函数默认传递的第一个参数就是对应类的指针（this）。第三类是直接的C++函数和C++类的成员函数。前两类都有固定的格式（返回整形，参数必须只有一个而且是LuaState的指针）。而这一类可以直接导出任何格式的C++函数。

下面我们通过一些例子来分别介绍这三类函数的导出。

第一类导出静态或者全局函数。

这里函数的格式已经确定，标准格式是 int FunctinoName(LuaState* state)。

```
//定义一个静态函数，标准格式
static int LS_LOG(LuaState* state)
{
   printf("In static function\n");
   return 0;
}
//定义一个类，它导出一些静态函数
class Logger
{
public:
   static int LS_LOGMEMBER(LuaState* state)
   {
      LuaStack args(state);
      printf("In member function.  Message: %s\n", args[1].GetString());
      return 0;
   }

   static virtual int LS_LOGVIRTUAL(LuaState* state)
   {
      printf("In virtual member function\n");
      return 0;
   }
};
//获取全局变量表，导出静态函数
LuaObject globalsObj = state->GetGlobals();
globalsObj.Register("LOG", LS_LOG);
state->DoString("LOG()");
//导出类静态成员函数
globalsObj.Register("LOGMEMBER", &Logger::LS_LOGMEMBER);
state->DoString("LOGMEMBER('The message')");
//导出虚的类静态成员函数
globalsObj.Register("LOGVIRTUAL", &Logger::LS_LOGVIRTUAL);
state->DoString("LOGVIRTUAL()");
```

这是静态函数相关的例子，其实Register可以直接支持类的成员函数的导出，这个和数据相关了属于第二类。下面是Register函数的三种定义。

```
void Register( const char* funcName, lua_CFunction function, int nupvalues = 0 );
void Register( const char* funcName, int (*func)(LuaState*), int nupvalues = 0 );
void Register( const char* funcName, const Callee& callee, int (Callee::*func)(LuaState*), int nupvalues = 0 );
```

最后种可以导出C++类的成员函数，我们看到它的第二个参数是一个C++类对象的引用可见它是数据依赖的。虽然Register函数可以导出类的成员函数，但它无法表示C++类的继承关系，第二类函数则可以轻松的完成这个任务。

第二类导出成员类函数

因为类的成员函数都和一个this变量有关，所以他们按理来说是无法导出到一个表中作为独立函数集的。但我们可以把它导出到一个表中（这个表叫做元表），将这个表里的成员函数需要的this变量保存在另一用户数据对象或者表的__object成员里。然后通过这个表或者用户数据来调用元表中的方法。比如this:hide()这里的this是一个用户数据它代表一个c++对象的this指针，操作符':'是一种调用方法，它表示调用的hide函数的第一个参数就是this对象的用户数据。通过这种方法可以调用类的成员函数了。下面看一个例子。

首先定义一个C++类。这个类定义了两个函数，一个导出函数。

```
class MultiObject
{
```

```
public:
    MultiObject(int num) :
        m_num(num)
    {
    }
    int Print(LuaState* state)
    {
        printf("%d\n", m_num);
        return 0;
    }
protected:
    int m_num;
};
```

其次，我们创建一个元表，并且将这个函数导入到元表中。

```
LuaObject metaTableObj = state->GetGlobals().CreateTable("MultiObjectMetaTable");
metaTableObj.SetObject("__index", metaTableObj);
metaTableObj.RegisterObjectFunctor("Print", &MultiObject::Print);
```

创建一个元表，它保存在全局表中。然后设置元表的__index元素为它自己。当Lua在查找一个元素找不到的时候会看这个表里有没有__index元素如果有则看这个元素的类型，如果是函数就调用这个函数通过它获取要查找的值，如果是表就这这个表中继续查找刚才查找的元素。所以这里设置表的__index元素为元表自己。就是为了方便其他表在查找的找不到的时候来查找元表。最后一步将类的成员函数导出到元表中。这里只举了一个导出函数的例子，读者可以导出更多的函数，方法相同。

现在的元表相当于一个算法表，但这些导出函数都要依赖一个他们要操作对象的指针也就是this作为他们的第一个参数。所以直接调用他们是不可能的，必须通过一个对象来调用他们。哪么这个对象如何产生呢？

第三，创建调用元表函数的用户数据LuaPlus对象。创建的方法非常简单，首先必须要有一个MultiObject对象的指针。然后调用BoxPointer函数就可以创建一个用户数据对象。

```
//定义一个C++对象
MultiObject obj1(10);
//创建一个LuaPlus用户数据对象
LuaObject obj1Obj = state->BoxPointer(&obj1);
//设置这个对象的元表是刚才我们创建的函数表
obj1Obj.SetMetaTable(metaTableObj);
//保存这个用户数据对象到全局表中
state->GetGlobals().SetObject("obj1", obj1Obj);
//同理这是另一个对象的创建
MultiObject obj2(20);
LuaObject obj2Obj = state->BoxPointer(&obj2);
obj2Obj.SetMetaTable(metaTableObj);
state->GetGlobals().SetObject("obj2", obj2Obj);
```

第四，创建好了调用对象就可以调用成员变量了。

```
state->DoString("obj1:Print()");
state->DoString("obj2:Print()");
```

obj1和obj2是C++对象obj1和obj2的this指针的LuaPlus用户数据对象。然后在这里调用Print函数就知道操作那个对象了。

还有另外一种处理方法。创建一个表，然后设置轻量级的用户数据。上面一种处理方法是创建了一个用户数据。

```
//创建一个空表
LuaObject table1Obj = state->GetGlobals().CreateTable("table1");
//设置轻量级的数据注意__object名称不能改变
table1Obj.SetLightUserData("__object", &obj1);
//设置元表
```

```
table1Obj.SetMetaTable(metaTableObj);

LuaObject table2Obj = state->GetGlobals().CreateTable("table2");
table2Obj.SetLightUserData("__object", &obj2);
table2Obj.SetMetaTable(metaTableObj);
//调用成员函数
state->DoString("table1:Print()");
state->DoString("table2:Print()");
```

这两种方法都很方便，读者选择自己喜欢的方法。

第三类函数，直接导出C++ 的函数到Lua脚本中。

这类函数没有格式的限制一般普通的C++类型都支持，它最多支持7个参数。支持导出静态和成员函数。下面是一个简单的例子。

```
float Add(float num1, float num2)
{
   return num1 + num2;
}
LuaStateOwner state;
//这个注册函数很强大
state->GetGlobals().RegisterDirect("Add", Add);
//直接调用
state->DoString("print(Add(10, 5))");
```

导出类成员函数的例子。

```
class Logger
{
public:
   void LOGMEMBER(const char* message)
   {
      printf("In member function: \n", message);
   }
   virtual void LOGVIRTUAL(const char* message)
   {
      printf("In virtual member function: %s\n", message);
   }
};
//定义C++对象
Logger logger;
//注册成员函数
globalsObj.RegisterDirect("LOGMEMBER", logger, &Logger::LOGMEMBER);
globalsObj.RegisterDirect("LOGVIRTUAL", logger, &Logger::LOGVIRTUAL);
//调用脚本中的导出函数
state->DoString("LOGMEMBER('Hello')");
state->DoString("LOGVIRTUAL('Hello')");
```

到这里就导出函数就介绍完了。本书大量使用了第一和第二类的导出函数，没有使用第三类。读者可以尝试使用。下一小节介绍堆栈对象。

15.1.3 LuaStack

这个类用的并不是很多，只是在C++导出函数中使用非常多。用它可以方便的获取导出函数的参数。它内部使用LuaStackObject对象来获取当前LuaState的参数。我们现在介绍如何使用这个类来获取参数。下面的loadScheme函数是一个静态导出函数，它需要一个字符串类型的参数。首先定义一个LuaStack的对象，然后通过操作符[]来获取各个参数的值。由于这是

个静态函数，所以第一个参数就是脚本里传入的参数。如果是成员变量则第一个参数是this对象，第二个参数才是从脚本传入的参数。这一点一定要牢记。

```
int LSchemeMgr::loadScheme(LuaState* L)
{
    LuaStack arg(L);
    if (arg[1].IsString())
    {
        SchemeManager::getSingleton().loadScheme(arg[1].GetString());
    }
    return 0;
}
```

下面在举个成员函数的导出。这个函数注册一个控件事件的脚本处理函数。它需要两个参数一个是事件的名称，另一个是脚本处理函数。

```
int LWindow::subscribeEvent(LuaState* L)
{
    LuaStack  arg(L);
    if (arg[2].IsString() && arg[3].IsString())
    {
        ScriptFunctor functor(arg[3].GetString());
        d_window->subscribeEvent(arg[2].GetString(), Event::Subscriber(functor));
    }
    return 0;
}
```

注意这里是从第二个参数获取的，因为第一个参数是调用这个函数的用户数据对象它代表这这个成员函数的this参数。

到此本书涉及到的LuaPlus相关的知识基本就介绍完了。下一节介绍脚本模块的实现。

15.2 脚本模块

前文已经讲过CEGUI的支持脚本是通过脚本接口来支持的，任何脚本只要实现了脚本接口都可以方便的和CEGUI挂钩。CEGUI使用Lua作为脚本语言。本书也一样，不同的是本书使用的是LuaPlus封装库。读者还记得需要实现那个接口吗？没错它就是ScriptModule接口。本节介绍一个名叫ScriptSystem的类，它派生自ScriptModule并实现了它的所有虚函数。

首先介绍类的成员变量。这个类有两个成员变量（不包括基类的）一个是LuaPlus的状态智能指针类，另一个是脚本系统的单件指针。

```
static LuaStateAuto m_plusState;
static ScriptSystem* s_pOnlyYou;
```

类LuaStateAuto不会自动创建Lua执行环境，它只会释放它。LuaPlus的初始化在如下函数中进行。

```
bool ScriptSystem::Initialize()
{
    //创建一个新的LuaState对象
    m_plusState = LuaState::Create();
    //打开Lua的系统库，如果不开则许多Lua系统函数无法执行
    m_plusState->OpenLibs();
    //创建一个控件功能管理器的对象
    new LUA_CEGUI::ControlMgr();
    return true;
}
```

控件管理器将在下一节介绍。它管理控件功能导出类。

其次，我们逐个讲述脚本接口的虚函数。

第一个函数，执行一个文件。非常简单调用LuaPlus的DoFile函数来执行。

```
void ScriptSystem::executeScriptFile(const String& filename, const String& resourceGroup)

{
    try
    {
        if( m_plusState->DoFile((resourceGroup + filename).c_str()) != 0)
        {
            char tmp[512];
            sprintf(tmp, "%s", m_plusState->Stack(-1).GetString());
            Logger::getSingleton().logEvent(tmp , Errors);
        }
    }
    catch (LuaException& e)
    {
        MessageBox(NULL, "executeScriptFile Error", e.GetErrorMessage(), MB_ICONSTOP
    | MB_OK);
    }
}
```

读者也可以先自己加载脚本资源然后在调用LuaPlus的DoString或者DoBuffer函数来执行脚本。在本章代码中提供了这样的实现。为什么要这么做呢？因为游戏资源一般是打包的，LuaPlus（其实是Lua实现的，但我们以后就这样介绍了）一般来说是无法获取这些资源。

第二个函数executeScriptGlobal执行全局函数，我们也是使用DoString函数来实现。

```
int   ScriptSystem::executeScriptGlobal(const String& function_name)
{
    try
    {
        if(  m_plusState->DoString(function_name.c_str()) != 0)
        {
            char tmp[512];
            sprintf(tmp, "executeScriptGlobal  %s", m_plusState->Stack(-1).GetString());
            Logger::getSingleton().logEvent(tmp , Errors);
        }
    }
    catch (LuaException& e)
    {
        MessageBox(NULL, "executeScriptGlobal Error", e.GetErrorMessage(),
                MB_ICONSTOP | MB_OK);
        return 0;
    }
    return 1;
}
```

第三个函数executeString执行一段字符串。

```
void ScriptSystem::executeString(const String& str)
{
    try
    {
        if(  m_plusState->DoString(str.c_str()) != 0)
        {
            char tmp[512];
            sprintf(tmp, "executeString  %s", m_plusState->Stack(-1).GetString());
            Logger::getSingleton().logEvent(tmp , Errors);
```

```
            }
        }
        catch (LuaException& e)
        {
            MessageBox(NULL, "executeString Error", e.GetErrorMessage(),
            aMB_ICONSTOP | MB_OK);
        }
}
```

第四个函数executeScriptedEventHandler执行一段字符串。这个函数用来调用脚本函数。可以参考ScriptFunctor的操作符()的实现。

```
bool ScriptSystem::executeScriptedEventHandler(const String& handler_name, const EventArgs& e)
{
    try
    {
        if(  m_plusState->DoString(handler_name.c_str()) != 0)
        {
            char tmp[512];
            sprintf(tmp, "executeScriptedEventHandler  %s",
            m_plusState->Stack(-1).GetString());
            Logger::getSingleton().logEvent(tmp , Errors);
        }
    }
    catch (LuaException& e)
    {
        MessageBox(NULL, "executeScriptedEventHandler Error", e.GetErrorMessage(),
        MB_ICONSTOP | MB_OK);
        return false;
    }
    return true;
}
```

第五个函数createBindings这个函数负责导出CEGUI控件的功能到脚本中。如何导出呢？注册一些C++函数到Lua脚本中，这些函数用来调用CEGUI控件的一些功能函数。这个功能我们在下一节详细介绍。

```
void ScriptSystem::createBindings(void)
{
    LUA_CEGUI::ControlMgr::RegisterAll(m_plusState.Get());
}
```

第六个函数destroyBindings与第五个函数配合，销毁对应的函数导出。

```
void ScriptSystem::destroyBindings(void)
{
    LUA_CEGUI::ControlMgr::ReleaseAll();
}
```

第七个函数subscribeEvent这个函数当CEGUI在解析布局文件发现Event元素的时候会调用这个函数来注册事件的脚本处理函数。这里通过ScriptFunctor来实现脚本的调用。这个函数还有一个不同参数的重载函数，功能类似这里就不在介绍了。

```
Event::Connection  ScriptSystem::subscribeEvent(EventSet* target, const String& name, const String& subscriber_name)
{
    if(target)
    {
        ScriptFunctor functor(subscriber_name);
        Event::Connection con = target->subscribeEvent(name, Event::Subscriber(functor));
        return con;
    }
```

```
    throw ;
}
```

读者可能对为什么在Layout文件中定义Event元素后就可以在事件发生的时候调用脚本函数呢？首先布局文件中发现Event元素的时候CEGUI的布局文件处理类就会调用这个函数注册一个ScriptFunctor到对应的事件上。在事件发生的时候会调用ScriptFunctor的操作符()，这个操作符就会调用executeScriptedEventHandler函数，最终调用到脚本函数。

脚本模块必须实现的接口函数就这么多。

在这里我们介绍CEGUI调用脚本模块的方法的顺序。在CEGUI的System类的构造函数中会检查是否有配置文件传入，如果有的话就分析配置文件。如果配置文件中有全局脚本文件则CEGUI在构造函数中会调用脚本模块的executeScriptFile函数。

```
if (!configInitScript.empty())
{
  try
  {
      executeScriptFile(configInitScript);
  }
  catch (...) {}  //忽略所有异常
}
```

本章例子就添加了一个名为ceguiconfig.xml的CEGUI配置文件，并在其中设置了全局的脚本文件。配置文件的格式如下，其实还可以设置默认的资源组。

```
<CEGUIConfig InitScript="../datafiles/lua_scripts/Global.lua" />
```

在创建System类之前，先创建一个脚本系统的对象然后传送脚本系统的指针到System类中参与构造。具体的调用代码如下所示：

```
new ScriptSystem();
new CEGUI::System(new CEGUI::OpenGLRenderer(1024), NULL, NULL, ScriptSystem::onlyYou(), "ceguiconfig.xml");
```

创建一个脚本系统对象，然后传入脚本模块的指针以及全局配置文件的名称。这样当CEGUI的System类构造的时候就会调用全局的脚本文件了。第15.4节将介绍这个脚本如何实现第14章例子的界面逻辑。

15.3 控件功能导出

本节将使用LuaPlus导出Window类以及我们实现的倒计时控件的功能。其他控件的功能没有导出，读者需要的时候自己导出。除此之外我们还导出了System类，WindowManager类，Scheme类，ImagesetManager类，FontManager类的部分功能。

在脚本系统里我们初始化了LuaPlus并且创建了ControlMgr对象。哪么我们先介绍它。这个类负责管理所有的控件导出类。什么是控件导出类呢？控件的功能要导出必须有个类来代理，这个代理类包含控件的指针，定义一些Lua的C++函数，然后导出到Lua中。控件的继承关系需要在Lua中表现出来，哪么如何表现呢？通过元表来实现，把派生类的元表设置为基类的函数表（导出函数）。这样当在派生类中函数表中找不到调用的函数的时候，就会查找元表（基类的导出函数表），这样就可以实现类似继承的关系了。

我们首先介绍ControlMgr类。它管理所有的控件导出代理类。注册函数到Lua中的函数RegisterAll这个函数会在脚本模块的createBindings中调用。

```
void ControlMgr::RegisterAll(LuaState* L)
{
    LWindow::Register(L);
    LTimerWindow::Register(L);
    LApp::RegisterFunctor();
    LSystem::RegisterFunctor();
    LWindowMgr::RegisterFunctor();
    LSchemeMgr::RegisterFunctor();
    LFontMgr::RegisterFunctor();
    LImagesetMgr::RegisterFunctor();
}
```

这个函数调用所有代理类的脚本注册函数。LWindow类是Window类的代理，另一个类LTimerWindow是TimerWindow类的代理类。其他类除了LApp外，都是CEGUI单件的功能导出类。他们导出的是静态类，属于我们介绍的第一类。而两个控件类则是属于第二类。下面介绍导出每个窗口功能的函数。由于窗口是树形结构，因此这个函数递归调用所有的窗口，为每个窗口和它的子窗口导出功能。释放控件绑定的函数我们就不介绍了。

```
void ControlMgr::createControl(CEGUI::Window* pWindow)
{
    if (pWindow == NULL)
    {
        return;
    }
    int cnt = pWindow->getChildCount();
    //递归创建所有的子窗口对象
    for (int i=0; i<cnt; i++)
    {
        Window* pChild = pWindow->getChildAtIdx(i);
        //这句非常重要，否则会导致堆栈崩溃
        if (!pChild->isAutoWindow())
        {
            createControl(pChild);
        }
    }
    LWindow* d_pWindow = newLWindow(pWindow);
    d_allControls.push_back(d_pWindow);
    d_pWindow->RegisterFunctor();
}
```

其中的newLWindow函数根据窗口的类型创建对应的窗口代理类。创建代理后将它保存起来，在清除绑定的时候销毁。然后调用窗口的注册函数，将代理类的函数注册到函数表中。下面是newLWindow的实现。

```
LWindow* ControlMgr::newLWindow(CEGUI::Window* pWindow)
{
    if (pWindow)
    {
        if (pWindow->testClassName("Timer"))
        {
            return new LTimerWindow(pWindow);
        }
        else
        {
            return new LWindow(pWindow);
        }
    }
    return new LWindow(pWindow);
}
```

这个函数根据不同的窗口类型创建对应的代理类，我们为了简单只实现了两个代理类。读者可以自己实现其他的类。

下一个函数当然是控件代理基类（LWindow）的Register函数。这个是静态函数。它创建一个函数导出表（这个表可以作为派生类的元表）。

```
void LWindow::Register(LuaState* L)
{
    if (d_metaTable || L == NULL)
    {
        return;
```

```
    }
    //获取全局变量表
    LuaObject objGlobal = L->GetGlobals();
    //创建一个LuaObject对象
    d_metaTable = new LuaObject;
    //创建一个全局表并保存在导出表中
    *d_metaTable = objGlobal.CreateTable("metaTblWindow");
    //设置这个表为元表
    d_metaTable->SetObject("__index", *d_metaTable);
    //注册导出函数
    d_metaTable->RegisterObjectFunctor("show", &LWindow::show);
    d_metaTable->RegisterObjectFunctor("hide", &LWindow::hide);
    d_metaTable->RegisterObjectFunctor("getText", &LWindow::getText);
    d_metaTable->RegisterObjectFunctor("setText", &LWindow::setText);
    d_metaTable->RegisterObjectFunctor("getProperty", &LWindow::getProperty);
    d_metaTable->RegisterObjectFunctor("setProperty", &LWindow::setProperty);
    d_metaTable->RegisterObjectFunctor("addChildWindow", &LWindow::addChildWindow);
    d_metaTable->RegisterObjectFunctor("active", &LWindow::active);
    d_metaTable->RegisterObjectFunctor("subscribeEvent", &LWindow::subscribeEvent);
}
```

注册完成以后这个表就是一个函数导出表了，它导出了控件的功能。读者可以看到我们使用的是第二类函数。

下一个函数RegisterFunctor，这个函数创建一个LuaObject对象并且这个对象是一个用户数据对象。用它来调用函数集中的函数，就想用this来调用类的成员函数一样。

```
void LWindow::RegisterFunctor()
{
    if (d_metaTable == NULL)
    {
        throw std::exception(" LWindow::RegisterFunctor MetaTable is not ready!");
    }
    //这一步创建一个用户数据对象，把this指针传递给它
    LuaObject objSys = ScriptSystem::State()->BoxPointer(this);
    //设置函数集为这个用户对象的元表
    objSys.SetMetaTable(*d_metaTable);
    //生成这个用户对象的名称，使用窗口控件的名称
    STRING name = Helper::Encode::utf8_to_mbcs(d_window->getName().c_str());
    //设置全局变量，使用窗口名称为变量的名称
    ScriptSystem::State()->GetGlobals().SetObject( name.c_str() , objSys);
}
```

如果有个窗口名叫Login哪么可以通过"Login:show()"来调用窗口的函数show了。

下面介绍LTimerWindow的函数表注册函数。这个函数注册TimerWindow的Lua导出函数，并且设置Window类的导出集为TimerWindow类导出集的元表。

```
void LTimerWindow::Register(LuaState* L)
{
    if (d_metaTableTimer || L == NULL)
    {
        return;
    }
    //创建一个派生类的导出集
    LuaObject objGlobal = L->GetGlobals();
    d_metaTableTimer = new LuaObject;
```

```
    *d_metaTableTimer = objGlobal.CreateTable("metaTblTimer");
    //设置派生类导出集对象为元表
    d_metaTableTimer->SetObject("__index", *d_metaTableTimer);
    //设置父类的导出函数集为自己的元表
    d_metaTableTimer->SetMetaTable(*LWindow::d_metaTable);
    //导出自己的Lua函数
    d_metaTableTimer->RegisterObjectFunctor("setTimer", &LTimerWindow::setTimer);
    d_metaTableTimer->RegisterObjectFunctor("getTimer", &LTimerWindow::getTimer);
    d_metaTableTimer->RegisterObjectFunctor("setTimeFormat",
&LTimerWindow::setTimeFormat);
    d_metaTableTimer->RegisterObjectFunctor("getTimeFormat",
                &LTimerWindow::getTimeFormat);
}
```

这个函数实现了导出代理类的继承关系，其实也是控件类的继承关系。LTimerWindow类的父类是LWindow类，代理类的继承关系和控件类的继承关系完全相同。

下面这个函数是LTimerWindow的用户数据注册函数。

```
void LTimerWindow::RegisterFunctor()
{
    if (d_metaTableTimer == NULL)
    {
        throw std::exception(" LTimerWindow::RegisterFunctor MetaTable is not ready!");
    }
    //创建用户数据对象
    LuaObject objSys = ScriptSystem::State()->BoxPointer(this);
    //设置元表为LTimerWindow的函数集，并不是LWindow的函数集
    objSys.SetMetaTable(*d_metaTableTimer);
    STRING name = Helper::Encode::utf8_to_mbcs(d_window->getName().c_str());
    //设置这个对象为全局对象
    ScriptSystem::State()->GetGlobals().SetObject( name.c_str() , objSys);
}
```

如果这里有一个名为User的TimerWindow窗口，哪么可以通过"User:hide()"来调用基类的函数，也可以通过"User:getTimer()"来调用LTimerWindow的导出函数。

前面介绍了第二类导出函数，下面介绍第一类导出函数的导出过程。这类函数没有数据依赖，可以独立的导出，它没有第二类哪么复杂，它只需要将函数导入到一个表中就可以了，如果导入的表示全局变量表，则这个导出函数就是全局函数，可以直接通过函数名来调用，否则导出到其他表中就需要加上表名来调用。

我们举WindowManager类的注册函数为例。

```
void LWindowMgr::RegisterFunctor()
{
    //获取全局变量表
    LuaObject objGlobal = ScriptSystem::State()->GetGlobals();
    //创建一个名为WinMgr的表
    LuaObject system = objGlobal.CreateTable("WinMgr");
    //注册所有的导出函数
    system.Register("loadLayout", &LWindowMgr::loadLayout);
    system.Register("isWindowExist", &LWindowMgr::isWindowExist);
    system.Register("getWindow", &LWindowMgr::getWindow);
    system.Register("destoryWindow", &LWindowMgr::destoryWindow);
    system.Register("createWindow", &LWindowMgr::createWindow);
}
```

这里不需要设置什么元表，因为它只是为了更加合理才把它放到一个表中。因为他们都是一个单件的导出功能。我们介绍的这个函数就是WindowManager类的导出函数。表名就是WinMgr，可以在脚本里通过"WinMgr.getWindow()"函数。注意这里使用的是点而不是冒号。其他的导出类都是一样这里就不在介绍了。

读者如果希望添加新的控件代理类或者新的单件的导出代理类都可以仿照我们的实现自己实现，具体的框架已经设计出来，在添加新的内容就比较简单了。

函数框架已经介绍完了，下面介绍具体的导出函数。导出函数比较多我们不打算一一介绍，只是挑选一些有代表性的介绍。

首先介绍一个获取属性的函数。前文已经讲过可以通过脚本来设置和获取属性，现在我们就具体介绍如何通过Lua脚本来获取属性。下面是获取属性的C++导出函数的定义。

```
int LWindow::getProperty(LuaState* L)
{
    //脚本调用参数的LuaStack对象
    LuaStack  arg(L);
    //注意参数是从第2个开始，第一个是this
    if (arg[2].IsString())
    {
        STRING propName = arg[2].GetString();
        STRING code = "mbcs";
        if (arg[2].IsString())
        {
            code = arg[2].GetString();
        }
        try
        {
            CEGUI::String value = d_window->getProperty(propName.c_str());
            if (code == "unicode")
            {
                L->PushWString( (lua_WChar*)Helper::Encode::utf8_to_ucs(value));
            }
            else
            {
                L->PushString(Helper::Encode::utf8_to_mbcs(value));
            }
        }
        catch (...)
        {
            L->PushString("catch getProperty exception");
        }
    }
    return 1;
}
```

这个函数根据是需要Unicode字符串还是MBCS字符串决定返回的字符串类型。如果希望获取Unicode字符串则先把utf8字符串转化为Unicode字符串，然后调用LuaPlus提供的宽字符函数将字符串压入堆栈返回给Lua脚本的调用函数。

LuaPlus导出函数标准形式是参数是一个LuaState的指针，返回整形。当然LuaPlus是支持直接调用C++格式的函数的，但本书没有这样做。返回值代表这个导出函数将返回给Lua脚本中调用函数的参数个数，如果返回-1表示函数调用出错。大于等于零的返回表示函数返回的值的个数。哪么如何返回值给Lua脚本的调用函数呢？只需要在Lua的堆栈上压入返回的值就可以了。获取调用导出函数的参数也非常简单。只需要声明一个LuaStack对象然后就可以通过[]操作符来获取第几个参数了，在获取具体类型的值之前要先检查这个参数是否存在，是否是需要的类型。

第二个函数是loadLayout它是窗口管理器的导出函数，它导出后脚本可以通过它加载一个布局文件。这个函数之所以特殊，是因为它需要返回一个窗口的用户数据对象。

```
int LWindowMgr::loadLayout(LuaState* L)
{
    LuaStack arg(L);
    //不是第二类函数则传入的参数从1开始
    if (arg[1].IsString())
    {
        try
        {
            //加载窗口布局文件
            Window* pRoot =
                    WindowManager::getSingleton().loadWindowLayout(arg[1].GetString());
            //创建窗口到脚本的代理类
            ControlMgr::onlyYou()->createControl(pRoot);
            //将刚刚创建的脚本管理类返回
            L->GetGlobal(pRoot->getName().c_str()).Push();;
        }
        catch (...)
        {
            L->PushString("LoadWindwoLayout Error");
        }
    }
    //返回一个参数给脚本
    return 1;
}
```

这个函数需要注意的是createControl函数是递归调用的，他将每个窗口的名称作为代理类在脚本系统中注册的名称。也就是说窗口的名称对应脚本里的窗口的导出代理类的对象。

第三个函数setGUISheet它将一个窗口设置为默认的窗口底板。这个函数的关键部分是获取要被设置为底板的窗口。如果传入的是窗口的名称就通过窗口管理器获取指定的窗口然后设置为底板窗口。如果传入的是一个代理类的对象（这个对象可能是createWindow或者是loadLayout函数返回的）就获取代理对象的窗口指针，然后获取对应窗口的指针，最后设置为底板窗口。

```
int LSystem::setGUISheet(LuaState*L)
{
    LuaStack arg(L);
    //如果传入的是窗口的名称
    if (arg[1].IsString())
    {
        Window* pWin = WindowManager::getSingleton().getWindow(arg[1].GetString());
        System::getSingleton().setGUISheet(pWin);
    }
    //如果传入的是代理类的对象
    else if (arg[1].IsUserData())
    {
        //首先获取用户数据，用户数据是LWindow类型的指针
        LWindow** pWin = (LWindow**)(arg[1].GetUserData());
        if(pWin && *pWin)
        {
            System::getSingleton().setGUISheet((*pWin)->getCEGUIWindow());
        }
    }
```

```
    return 0;
}
```

需要注意的是GetUserData的是用户数据的地址，而不是用户数据本身。这个函数不返回任何数据给脚本使用。

第四个函数subscribeEvent这个函数注册一个事件的脚本处理函数。同过这个函数可以在脚本里动态的注册事件的脚本处理函数，本书并没有实现反注册函数，读者可以尝试自己实现。

```
int LWindow::subscribeEvent(LuaState* L)
{
    LuaStack  arg(L);
    if (arg[2].IsString() && arg[3].IsString())
    {
        ScriptFunctor functor(arg[3].GetString());
        d_window->subscribeEvent(arg[2].GetString(), Event::Subscriber(functor));
    }
    return 0;
}
```

最后一个函数是handleSubmit这个函数的特殊之处是它并不是和CEGUI相关，而是和逻辑相关的导出函数。

```
int LApp::handleSubmit(LuaState* L)
{
    if (UILogic::getDemoConsole())
    {
        EventArgs e;
        UILogic::getDemoConsole()->handleSubmit(e);
    }
    return 0;
}
```

列出这个函数的目的是告诉读者，导出函数不一定非要CEGUI的控件或者是CEGUI的各种管理器的功能函数。只要读者认为需要这个函数需要导出就可以导出它。

本节只介绍了一小部分的导出函数，但是不论有多少的导出函数，他们的原理都是一样的，而且框架也已经定义好了。下节将介绍使用Lua脚本实现我们第14章例子的逻辑功能。

15.4 脚本逻辑

第14章的例子逻辑是通过C++实现的。这一节将通过脚本来实现同样的逻辑。需要的导出函数已经在上一节介绍了。

本章的Lua脚本文件名为Global.lua从它的名字可以看出它是CEGUI系统类会调用的一个全局的脚本。关于如何它是如何被调用的在第15.2节已经介绍。在Lua脚本中可以定义函数，定义变量。可以调用函数，函数和变量的定义已经Lua的语法等信息读者可以阅读Lua相关的教程。在脚本里调用函数，直接调用即可。比如在脚本里定义了createWindow函数则调用它，只要createWindow()就可以了。

我们首先介绍在这个全局脚本文件中定义的函数。函数configResourceGroup配置脚本的资源组，以及设置默认的资源组。这个函数的功能在C++函数SetResourceGroup定义，这个函数目前已经不会被调用了。

```
function configResourceGroup()
    System.setResourceGroup("schemes","../datafiles/schemes/")
    System.setResourceGroup("imagesets","../datafiles/imagesets/")
    System.setResourceGroup("fonts","../datafiles/fonts/")
    System.setResourceGroup("layouts","../datafiles/layouts/")
    System.setResourceGroup("looknfeels","../datafiles/looknfeel/")
    System.setResourceGroup("lua_scripts","../datafiles/lua_scripts/")
    System.setDefaultResourceGroup("scheme","schemes")
    System.setDefaultResourceGroup("font","fonts")
    System.setDefaultResourceGroup("imageset","imagesets")
    System.setDefaultResourceGroup("layout","layouts")
```

```
    System.setDefaultResourceGroup("looknfeel","looknfeels")
    System.setDefaultResourceGroup("lua_scripts","lua_scripts")
end
```

调用System类的定义设置资源函数，以及设置默认资源组函数。

初始化函数，这个函数加载模式，设置默认字体，创建一个字体，设置鼠标，创建背景图片。

```
function initialize()
    SchemeMgr.loadScheme("VanillaSkin.scheme")
    System.setDefaultFont("FZYT");
    if FontMgr:isFontPresent("Iconified-12") then
        FontMgr:createFont("Iconified-12.font");
    end
    System.setDefaultMouseCursor("Vanilla-Images", "MouseArrow");
    ImagesetMgr.createImagesetFromImageFile("BackgroundImage", "GPN-2000-001437.tga");
end
```

这些函数都是在对应的表中定义的，具体实现是通过CEGUI对应的函数来实现的。

第三个函数createWindow这个函数实现了createDemoWindows函数的功能。

```
function createWindow()
    --创建背景窗口，并设置属性
    local background = WinMgr.createWindow("Vanilla/StaticImage");
    background:setProperty("FrameEnabled", "False");
    background:setProperty("BackgroundEnabled", "False");
    background:setProperty("RiseOnClick", "False");
    background:setProperty("AlwaysOnTop", "True");
    background:setProperty("Image", "set:BackgroundImage image:full_image");
    --设置这个窗口为默认窗口底板
    System.setGUISheet(background);
    --加载子窗口布局文件
    local child = WinMgr.loadLayout("VanillaWindows.layout")
    background:addChildWindow(child);
    --创建IME显示窗口
    local ime = WinMgr.createWindow("Vanilla/ImeWindow", "IMEShow");
    ime:setProperty("UnifiedAreaRect", "{{0.0,150.0},{1.0,-400.0},{0.0,350.0},{1.0,-140.0}}");
    background:addChildWindow(ime);
    ime:hide();
    App.setImeWindow(ime);background:active();
    --创建背景计时器
    local bgtimer = WinMgr.createWindow("Vanilla/TimerWindow", "BGTimer");
    bgtimer:setProperty("UnifiedAreaRect","{{0.0,50.0},{1.0,-200.0},{0.0,350.0},
    {1.0,-170.0}}");
    bgtimer:setProperty("TimerValue", "3700");
    bgtimer:setProperty("TextColor", "FF00FF00");
    bgtimer:setProperty("Font", "FZYT");
    bgtimer:setProperty("TextPrefix", "这是背景计时器");
    background:addChildWindow(bgtimer);
    background:active();
    --注册倒计时结束的事件处理函数
    Demo_NamespaceViewer_Timer:subscribeEvent("TimerEnd","Timer_OnEnd()");
end
```

第四个函数Console_Submit处理用户输入内容提交。

```
function Console_Submit()
    local txt = Vanilla_Console_Editbox:getText();
```

```
    App.PrintDebugMessage(txt);
    --如果是退出命令则退出
    local s,e = string.find(txt,"/quit")
    if s ~= nil and s >= 0 then
        App.quitApp();
    end
    --原来的C++代码处理
    App.handleSubmit();
    --设置为空
    Vanilla_Console_Editbox:setText("");
end
```

这个函数获取输入框的文字，然后查找是否是退出命令，如果是则退出程序，否则交给第14章的默认函数处理。最后设置输入框的文本为空。这个函数是通过在布局文件中添加事件元素来实现。具体的实现如下。

在输入框窗口中注册为TextAccepted的处理函数。

```
<Event    Name="TextAccepted"  Function ="Console_Submit()" />
```

在提交按钮窗口中注册为鼠标单击的处理函数。

```
<Event    Name="MouseClick"    Function ="Console_Submit()" />
```

有了这个注册当用户提交输入的时候就会调用这个脚本函数了。

最后一个函数 Timer_OnEnd，当定时器倒计时结束的时候会调用它。它将会隐藏父窗口。隐藏时就是调用父窗口的hide函数。这个函数实现原来handleCloseParent实现的内容。

```
function Timer_OnEnd()
    Demo_NamespaceViewer:hide();
end
```

它是如何被调用呢？通过在创建窗口的时候注册窗口的事件处理函数来实现。相信细心的读者已经知道它定那里实现了。没错是在createWindow函数里实现的。

在定义了函数之后。我们应该调用这些函数了。下面是调用的代码。

```
--首先配置资源组
configResourceGroup()
--初始化
initialize()
--创建窗口，等c++初始化完成后再调用
--createWindow()
```

这里首先调用了配置资源组的脚本函数，然后调用了初始化函数。创建窗口的函数我们放在了C++代码初始化完毕后在调用。调用代码如下：

```
void CEGUISample::createDemoWindows()
{
    System::getSingleton().getScriptingModule()->executeScriptGlobal("createWindow()");
    d_uiLogic.createDemoWindows();
}
```

调用脚本模块的执行全局脚本函数的接口。现在的d_uiLogic.createDemoWindows()函数中已经没有什么逻辑内容了，只是创建了DemoWindow类。

15.5 本章小节

本章介绍了LuaPlus的使用方法，实现了一套导出CEGUI功能的脚本框架。并且用脚本实现了第14章例子逻辑部分。实现了许多导出函数。

本章提供的练习题如下：

1.读者实现导出一个控件的功能。比如ListBox的功能。

2.读者将CEGUI的某个例子的逻辑脚本化。

附录

附录1 STL简单介绍

CEGUI代码大量使用了c++库的STL，如果读者对它不太熟悉可以看下面的简单介绍。

STL包含大量的模板类和模板化算法，我们只使用了一些简单的模板类和算法。模板类有许多共同的特征，他们大都有相关名称的成员函数，而且功能也极为相似。比如说vector（向量类），List（列表类），map（映射类1对1），multimap（映射类1对多），deque（双向队列）等，他们基本上都有push_back（插入到最后面），push_front（插入到最前面），pop_front（删除最前面的元素），pop_back（删除最后面的元素），insert（插入元素），erase（元素删除），begin(第一个元素），end（最后一个元素）等函数。例如：

```
std::vector<int> a , std::list<float> l;
void test（）
{
    a.push_back(1) ;  l.push_back(1.0f);
}
```

等等，注意std是命名空间如果没有指令#using std，则必须加上std::否则无法找到对应的类。说的简单点命名空间就是为了防止名字冲突，而限定变量，函数，类，结构，枚举等的作用域的。例如：

```
int a ;
void fun1() {}
namespace  nsp
{
    int a;
    void fun1() {}
}
```

这里不会引发重定义错误，因为a，和 nsp::a是不相干的，fun1和 nsp::fun1同理，如果在第一行加上#using namespace nsp;则会引发重定义错误，可见他的作用。

其次stl的类都重载了[]操作符，所有[]内可以不是数字，比如std::map<std::string , int> mp;，mp是一个将字符串映射到整形的map，我们可以mp.insert(make_pair("one" , 1))插入一个元素，并可以使用mp["one"]来访问该元素，例如mp["one"] = 2;例子中的make_pair就是算法函数，他返回一个map可以插入的结构。

第三 元素的查找，有的类中带有find函数，则可以直接调用，如果不带该函数比如std::vector<int> v;，我们可以使用find(v.begin()，v.end()，value)来查找，该函数返回一个迭代器类型std::vector<int>::iterator it;，注意迭代器的定义类型::iterator，我们可以把std::vector<int>看做是一个新类，这也符合模板的含义。比如：

```
v.push_back(1) ; v.push_back(2); v.push_back(3);
it = find(v.begin() , v.end() , 2);
if(it != v.end() )
{
    printf("找到元素%d\n" , *it);
}
```

可见it和指针类似可以直接调用it++来得到下一个元素，it--得到上一个元素。同样所有的stl类都有这样的重载操作符。另外可以遍历一遍来比较元素，例如：

```
for(it = v.begin(); it != v.end() it++)
{
    if(*it == 2)
    {
        printf("找到元素%d\n" , *it);
        break；
    }
}
```

第四 元素删除，可以调用stl类的erase函数，注意该函数将导致迭代器失效，所以要重置迭代器为erase的返回值，例如：it = v.erase（it），删除it指向的元素，并更新it为有效值。

第五 元素的遍历，方法如上元素查找的第二中方法，那个例子中遍历是为了查找元素。

这里只是简单的STL类的介绍，如要了解更多，请阅读STL教程。

附录2 关键词到CEGUI文件的映射表