OGRE 分析之设计模式（一）

Mythma
http://www.cppblog.com/mythma
Email: mythma@163.com

OGRE 的设计结构十分清晰，这得归功于设计模式的成功运用。

一、 Singleton

Singleton 模式是 OGRE 中用的最广泛的设计模式，若干类都使用了该模式。Singleton 模式本身很简单，实现方式多种多样，简单的如 GoF 提到的实现方式，复杂的有 Loki 库中解决方案，而 OGRE 的实现方法也是十分简单。

OGRE 提供了一个模板类——**Ogre::Singleton<T>**，作为所有 singleton 类的基类：

```
template <typename T> class Singleton
{
protected:
    static T* ms_Singleton;
public:
    Singleton( void )
    {
        assert( !ms_Singleton );
        ms_Singleton = static_cast< T* >( this );
    }
    ~Singleton( void )
        { assert( ms_Singleton );  ms_Singleton = 0;  }
    static T& getSingleton( void )
    {   assert( ms_Singleton );  return ( *ms_Singleton ); }
    static T* getSingletonPtr( void )
    { return ms_Singleton; }
};
```

需要使用 Singleton 的类，只需要从它继承即可：

```
class MyClass : public Singleton<MyClass>{//……..}
```

但在 OGRE 的源码中我们可以发现，从 Singleton 继承的类都 Override 了 Singleton 的成员 getSingleton 和 getSingletonPtr。OGRE 中是这样解释的：

Singleton 的**实现**都在 OgreSingleton.h 文件中，这意味着任何包含 OgreSingleton.h 头文件的文件编译后都有 Singleton 的一份实现，这符合模板机制。但是从别的 dll 中使用基于 Singleton 的类时，就会产生连接错误。子类 Override 一下，把实现放在.cpp 中，就会避免该问题。

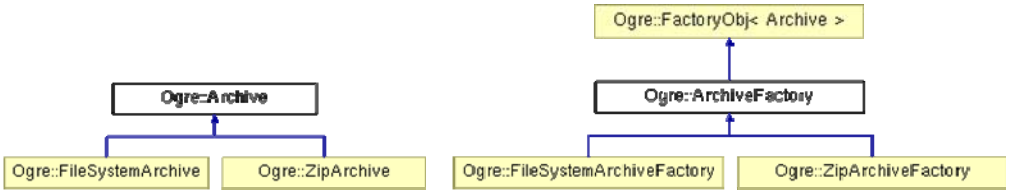
二、 Factory Method

Factory Method 本质上代理某个类的实例化过程，封装了一个 new 的语句而已。C++中 new 和 delete 通常都要成对出现，因此 Factory Method 也应该负责一个对象的 delete（GoF 没有提到由 Factory Method 构造的对象的析构问题）。另外，当一个类的构造函数的参数很多，其中有可以提供默认值的时候，用 Factory Method 还可以简化对象的创建。

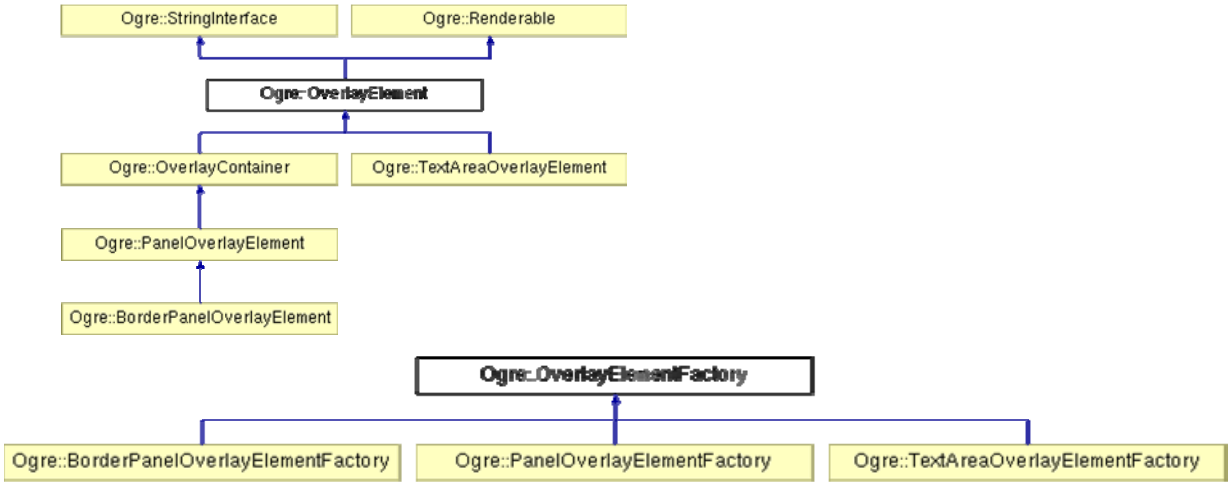
OGRE 中用到的 Factory Method 的地方很多。为满足最普遍的情况，OGRE 有一个模板化的 Factory Method——**Ogre::FactoryObj< T >**。因为具体类的构造函数的参数并不都相同，因此 Ogre::FactoryObj< T >并不是 OGRE 所有工厂方法的基类。

```
template< typename T > class FactoryObj
  {
public:
    virtual ~FactoryObj() {};
    virtual const String& getType() const = 0;
    virtual T* createInstance( const String& name ) = 0;
    virtual void destroyInstance( T* ) = 0;
};
```

OGRE 大部分 Factory Method 的使用效果都是连接平行的类层次，如：



不使用 Ogre::FactoryObj< T >的 Factory Method 很多，如：



还有 ParticleEmitter 和 ParticleEmitterFactory 两个比较大的类层次等。

三、 Abstract Factory

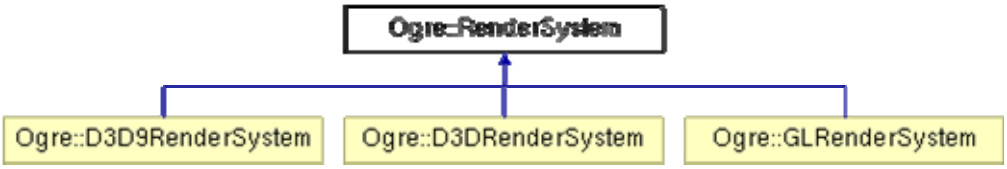
Abstract Factory 提供了用来创建一系列相关或相互依赖对象的接口。对使用者而言，无需关心使用的是何种具体的对象。OGRE 中，该模式体现在它的 Plug-in 机制。

OGRE 提供了 Plug-in 机制。利用 Plug-in，一方面把与平台相关的部分和与平台无关的部分分离开来，如底层渲染系统既可以使用 OpenGL，又可以使用 DirectX；另一方面便于系统功能的扩充，如场景既可以使用 BSP 方式管理，又可以用 Octree 管理。

OGRE 的 Plug-in 实现方式有两种，自此只讨论一下用 Abstract Factory 模式实现的 Plug-in 。以此方式实现的主要有两大类 Plug-in ： RenderSystem_x 和 Plugin_xSceneManager。

1、RenderSystem

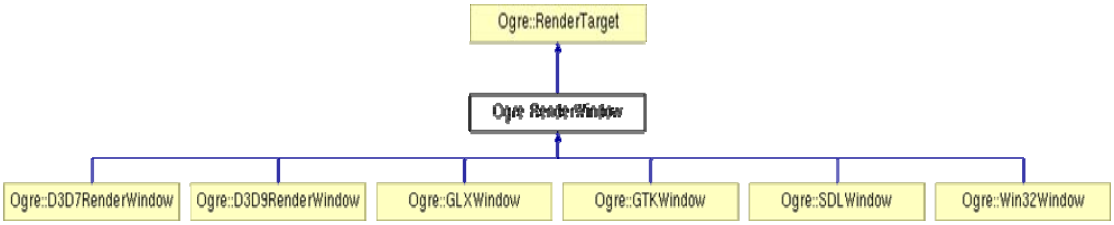
RenderSystem 是与渲染引擎相关的部分，如 RenderSystem_Direct3D7.dll、RenderSystem_Direct3D9.dll、RenderSystem_GL.dll：



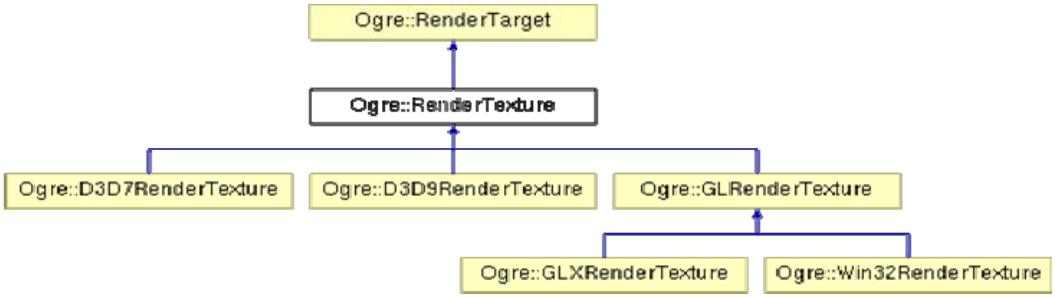
与渲染相关的对象都是通过具体的子 RenderSystem 创建的，这样就形成了不同类别的 Render 产品。在使用 OGRE 的时候，只需关心 RenderSystem，而无需关心究竟是那种具体的 RenderSystem。

RenderSystem 创建的产品主要有：

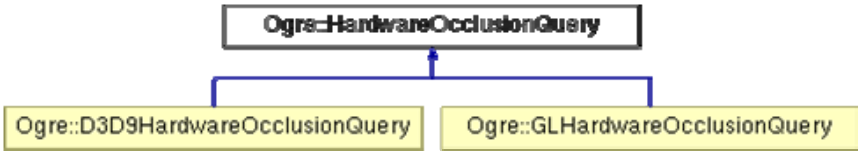
1）RenderWindow



2) RenderTexture

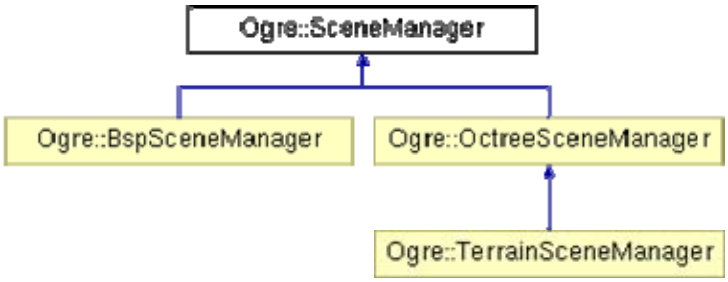


3) HardwareOcclusionQuery



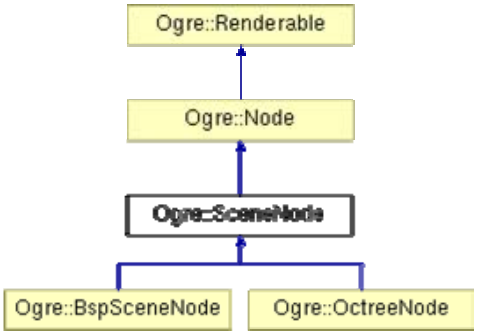
2、SceneManager

SceneManager 是与场景内数据的组织方式相关的部分，如：Plugin_OctreeSceneManager.dll、Plugin_BSPSceneManager.dll 等：

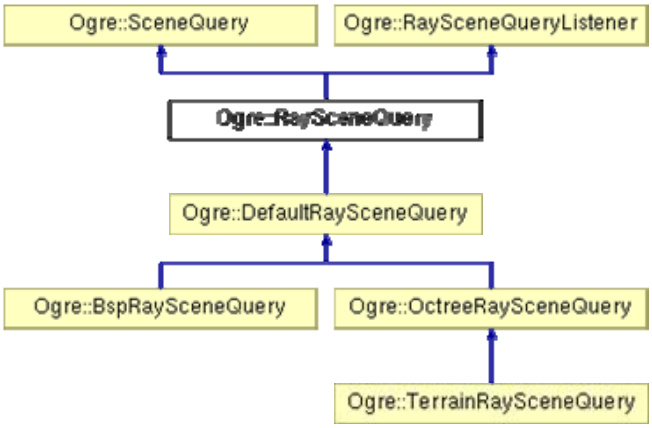


同样，SceneManager 只是个接口，具体的场景管理是由子类完成的。它的产品系主要有：

1）SceneNode



2）RaySceneQuery



SceneManager 中的很多 Products 是相同的，因此由 SceneManager 就完成创建过程。所以说 SceneManager 不是个纯粹的 Abstract Factory。

OGRE 分析之设计模式（二）

Mythma
http://www.cppblog.com/mythma
Email: mythma@163.com

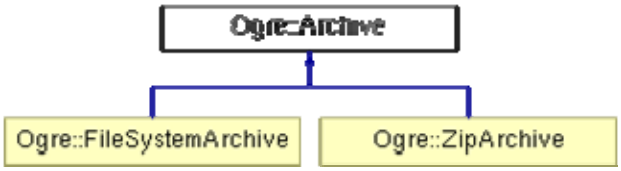
OGRE 的设计结构十分清晰，这得归功于设计模式的成功运用。

四、Adapter

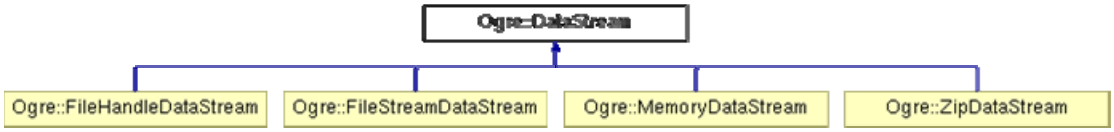
Apdater 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。当使用第三方工具包时候，这种模式经常用到。OGRE 使用了其他的几个开源项目，如 Cg、freetype、ode、zzip 等，与其他接口肯定会有不兼容的情况，这就需要 Adapter。

看一下 OGRE 的文件系统与 zzip：OGRE 是面向对象的，zzip 提供的都是结构体和函数。为了处理压缩文件和普通文件使用相同的接口，就需要 Adapter。

OGRE 中处理文件目录使用 Ogre::Archive，zzip 用 ZZIP_DIR 来表示压缩文件目录。因此在 Ogre::ZipArchive 中加入一个 ZZIP_DIR 对象，以使 zzip 的接口适应 Archive 的接口——使用的是对象适配器。



文件的处理，OGRE 使用 Ogre::DataStream 接口，对应 zzip 的 ZZIP_FILE。同样，OGRE 使用的是对象适配器。

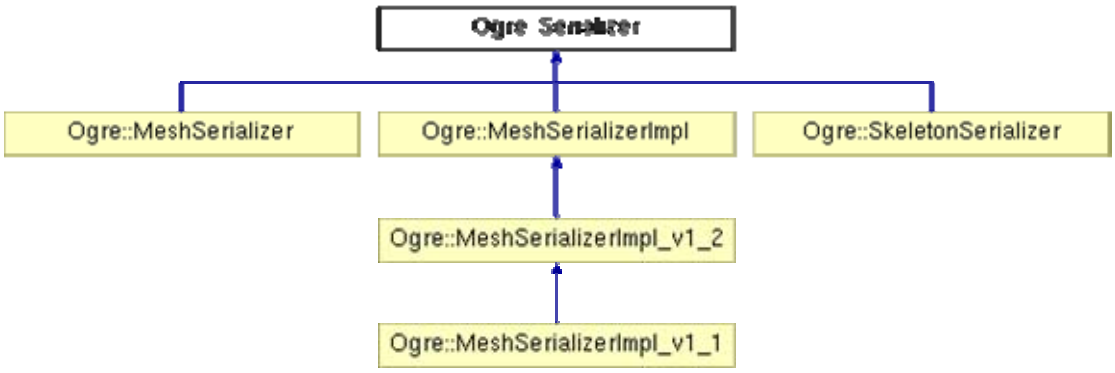


可以看出，Adapter 模式与 Abstract Factory 模式可以实现相同的效果——让别的接口兼容（回顾一下用 Abstract Factory “适配”不同的 RenderSystem）,但是两者的区别在于 Abstract Factory 是用于创建不同类别的对象；而 Adapter 没有创建别的对象的功能。所以 GoF 把 Abstract Factory 分在创建型模式类别中，而 Adapter 在结构型模式中。可以发现，这两种模式，OGRE 的文件系统都使用了。

五、Bridge

Bridge 的意图是使得抽象部分与它的实现部分分离，这样就使得两部分都可以独立变化提高可扩充性。该模式可以在多种情况下使用，如实现跨平台、程序换肤、文件版本演化等。当然实现这些功能也可以使用别的设计模式，同一个问题有不同的解决方案，模式的使用也是仁者见仁，智者见智。

在分析 OGRE 二进制文件的序列化时，提到了 Ogre::Serializer 类层次，如下图。



可以看到序列化 Mesh 文件的 Serializer 既有 Ogre::MeshSerializer 又有 Ogre::MeshSerializerImpl 。OGRE 在读写 Mesh 文件的时候使用的是 Ogre::MeshSerializerImpl 以及它的子类，而 Ogre::MeshSerializer 是 Ogre::MeshSerializerImpl 系列对外的“接口”，但二者是平辈关系。再看一下 MeshSerializer 的实现一切将会真相大白：

```
MeshSerializer::MeshSerializer()
{
    // Set up map
    mImplementations.insert(
        MeshSerializerImplMap::value_type("[MeshSerializer_v1.10]",
        new MeshSerializerImpl_v1_1() ) );

    mImplementations.insert(
        MeshSerializerImplMap::value_type("[MeshSerializer_v1.20]",
        new MeshSerializerImpl_v1_2() ) );

    mImplementations.insert(
        MeshSerializerImplMap::value_type(msCurrentVersion,
        new MeshSerializerImpl() ) );
}
//---------------------------------------------------------------------
void MeshSerializer::exportMesh(const Mesh* pMesh, const String& filename)
{
    MeshSerializerImplMap::iterator impl = mImplementations.find(msCurrentVersion);
    if (impl == mImplementations.end())
    {
        OGRE_EXCEPT(...);
    }
    impl->second->exportMesh(pMesh, filename);
}
```

可见具体的 Mesh 文件读取 MeshSerializer 并未实现，而是由合适的 Implementor 实现的，其他组件若要使用序列化功能，只需调用 MeshSerializer 即可。

从实现上，还可以发现，与 GoF 提到的实现有很大的差别，这说明模式不是“死的”，同一模式的实现也多种多样，只要能满足需求就可以了。

六、Proxy

Proxy 为其他对象提供一种代理以控制对该对象的访问。GoF 提到四种常见的情况：Romote Proxy，Virtual Proxy、Protection Proxy 以及 Smart Reference。在这里我只分析一下 Smart Reference。

Smart Pointer 在 STL 中有 std::auto_ptr，在 BOOST 中有 boost::shared_ptr、boost::shared_array 、 boost::scoped_ptr 、 boost::scoped_array 、boost::weak_ptr、boost::intrusive_ptr，在 Loki 中则有 Loki::SmartPtr。各种 Smart Pointer 都有不同的功能，适用的地方又各不相同。加上有的 Smart Pointer 的行为又有点诡异，尤其是 std::auto_ptr，所以实际应用中一向对之退而却步。

OGRE 虽不是模板库，却也有个 Smart Pointer——Ogre::SharedPtr。SharedPtr 是一个引用计数的共享指针。下面是 OGRE 对 SharedPtr 作的使用说明：

> This is a standard shared pointer implementation which uses a reference count to work out when to delete the object. OGRE does not use this class very often, because it is usually more efficient to make the destruction of objects more intentional (in blocks, say). **However in some cases you really cannot tell how many people are using an object, and this approach is worthwhile** (e.g. ControllerValue)

除了加上 mutex 外，其实现手法没有什么特别之处。

附上 Ogre::SharedPtr 的实现：

```
template<class T> class SharedPtr {
protected:
    T* pRep;
    unsigned int* pUseCount;
public:
    OGRE_AUTO_SHARED_MUTEX // public to allow external locking
    /** Constructor, does not initialise the SharedPtr.
     @remarks
        <b>Dangerous!</b> You have to call bind() before using the SharedPtr.
    */
    SharedPtr() : pRep(0), pUseCount(0) {}
    explicit SharedPtr(T* rep) : pRep(rep), pUseCount(new unsigned int(1))
    {
     OGRE_NEW_AUTO_SHARED_MUTEX
    }
    SharedPtr(const SharedPtr& r)
    {
     // lock & copy other mutex pointer
     OGRE_LOCK_MUTEX(*r.OGRE_AUTO_MUTEX_NAME)
     OGRE_COPY_AUTO_SHARED_MUTEX(r.OGRE_AUTO_MUTEX_NAME)
     pRep = r.pRep;
     pUseCount = r.pUseCount;
```

```
        // Handle zero pointer gracefully to manage STL containers
        if(pUseCount)
        {
            ++(*pUseCount);
        }
    }
    SharedPtr& operator=(const SharedPtr& r) {
        if (pRep == r.pRep)
            return *this;
        release();
        // lock & copy other mutex pointer
        OGRE_LOCK_MUTEX(*r.OGRE_AUTO_MUTEX_NAME)
        OGRE_COPY_AUTO_SHARED_MUTEX(r.OGRE_AUTO_MUTEX_NAME)
        pRep = r.pRep;
        pUseCount = r.pUseCount;
        if (pUseCount)
        {
            ++(*pUseCount);
        }
        return *this;
    }
    virtual ~SharedPtr() {
        release();
    }


    inline T& operator*() const { assert(pRep); return *pRep; }
    inline T* operator->() const { assert(pRep); return pRep; }
    inline T* get() const { return pRep; }

    /** Binds rep to the SharedPtr.
        @remarks
            Assumes that the SharedPtr is uninitialised!
    */
    void bind(T* rep) {
        assert(!pRep && !pUseCount);
        OGRE_NEW_AUTO_SHARED_MUTEX
        OGRE_LOCK_AUTO_SHARED_MUTEX
        pUseCount = new unsigned int(1);
        pRep = rep;
    }

    inline bool unique() const { assert(pUseCount); OGRE_LOCK_AUTO_SHARED_MU
TEX return *pUseCount == 1; }
```

```
        inline unsigned int useCount() const { assert(pUseCount); OGRE_LOCK_AUTO_S
HARED_MUTEX return *pUseCount; }
        inline unsigned int* useCountPointer() const { return pUseCount; }

        inline T* getPointer() const { return pRep; }

        inline bool isNull(void) const { return pRep == 0; }

        inline void setNull(void) {
          if (pRep)
            {
              // can't scope lock mutex before release incase deleted
              release();
              pRep = 0;
              pUseCount = 0;
              OGRE_COPY_AUTO_SHARED_MUTEX(0)
          }
      }

    protected:

        inline void release(void) {
          bool destroyThis = false;
            {
              // lock own mutex in limited scope (must unlock before destroy)
              OGRE_LOCK_AUTO_SHARED_MUTEX
              if (pUseCount)
                {
                  if (--(*pUseCount) == 0)
                    {
                      destroyThis = true;
                  }
              }
          }
          if (destroyThis)
              destroy();
      }

      virtual void destroy(void)
        {
          // IF YOU GET A CRASH HERE, YOU FORGOT TO FREE UP POINTERS
          // BEFORE SHUTTING OGRE DOWN
          // Use setNull() before shutdown or make sure your pointer goes
          // out of scope before OGRE shuts down to avoid this.
```

```
            delete pRep;
            delete pUseCount;
            OGRE_DELETE_AUTO_SHARED_MUTEX
        }
    };

    template<class T, class U> inline bool operator==(SharedPtr<T> const& a, SharedPtr
<U> const& b)
    {
        return a.get() == b.get();
    }

    template<class T, class U> inline bool operator!=(SharedPtr<T> const& a, SharedPtr
<U> const& b)
    {
        return a.get() != b.get();
    }
```

OGRE 分析之设计模式（三）

Mythma
http://www.cppblog.com/mythma
Email: mythma@163.com

OGRE 的设计结构十分清晰，这得归功于设计模式的成功运用。

七、Chain of Responsibility

Chain of Responsibility 是对象行为型模式，它把请求或消息以链的方式传送给对象处理者，避免了请求的发送者和接收者之间的耦合关系。该模式普遍用于处理用户事件和处理图形的更新。

OGRE 的消息传递也是使用 Chain of Responsibility 模式，体现在处理用户事件（鼠标消息和键盘消息）和图形的更新。首先看 OGRE 是如何传递处理用户事件的消息。

1、用户事件的消息

在《OGRE 分析之消息机制》中分析了 OGRE 中消息的产生、处理和传递，得到如下的传递顺序：

InputReader
→EventProcessor → EventDispatcher → EventTarget
→ EventListener

这只是一个总体的过程，现在从代码上看一下是如何使用 Chain of Responsibility 的。

1）消息的获取

InputReader 产生的用户消息是如何进入消息传递链的？从代码上分析：

OGRE 的入口是 Root::startRendering，然后进入系统循环：

```
void Root::startRendering(void)
  {//.....
   mQueuedEnd = false;
   while( !mQueuedEnd )
     {
      MSG  msg;
      while( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
        {
         TranslateMessage( &msg );
         DispatchMessage( &msg );
        }
      if (!renderOneFrame())
         break;
    }
}
```

然后进入 Root::renderOneFrame()：

```
bool Root::renderOneFrame(void)
  {
```

```
    if(!_fireFrameStarted())
        return false;
    _updateAllRenderTargets();      //进行图形重画
    return _fireFrameEnded();
}
```

```
bool Root::_fireFrameStarted()
{
    unsigned long now = mTimer->getMilliseconds();
    FrameEvent evt;
    evt.timeSinceLastEvent = calculateEventTime(now, FETT_ANY);
    evt.timeSinceLastFrame = calculateEventTime(now, FETT_STARTED);
    return _fireFrameStarted(evt);
}
```

从上面可以看出，构造出了 FrameEvent 消息并传递，但这并不是我们需要的 InputEvent，继续：

```
bool Root::_fireFrameStarted(FrameEvent& evt)
{
    // ……略
    // Tell all listeners
    for (i= mFrameListeners.begin(); i != mFrameListeners.end(); ++i)
    {
        if (!(*i)->frameStarted(evt))    //FrameListener::frameStated(evt)
            return false;
    }
    return true;

}
```

可见 FrameEvent 消息被送往 FrameListener 去了。**在初始化 OGRE 的时候，都需要创建一个 EventProcessor 并自动注册到 Root::mFrameListeners 中**，而 EventProcessor 属于 FrameLIstener 的子类，所以消息**必定**会传递给 EventProcessor：

```
bool EventProcessor::frameStarted(const FrameEvent& evt)
{
    mInputDevice->capture();
    while (mEventQueue->getSize() > 0)
    {
        InputEvent* e = mEventQueue->pop();
        processEvent(e);
        delete e;
    }
    return true;
}
```

现在终于出现 InputEvent（OGRE 中的鼠标消息封装在 MouseEvent，键盘消息封装在 KeyEvent 中，它们都是 InputEvent 的子类）。开始进入消息的传递阶段。

2）消息的传递

```
void EventProcessor::processEvent(InputEvent* e)
{
    // try the event dispatcher list
    for (DispatcherList::iterator i = mDispatcherList.begin();
      i != mDispatcherList.end(); ++i )
    {
      (*i)->dispatchEvent(e);  //EventDispatcher:: dispatchEvent(e)
    }

    // try the event target list
    if (!e->isConsumed())
    {
        //自己处理，略
    }
}
```

可见，InputEvent 先传递给 EventDispatcher 处理，若仍未被处理，就由 EventProcessor 自己处理（EventProcessor 本身也是个 Target，有处理用户消息的能力）。看消息在 EventDispatcher 中是如何处理的：

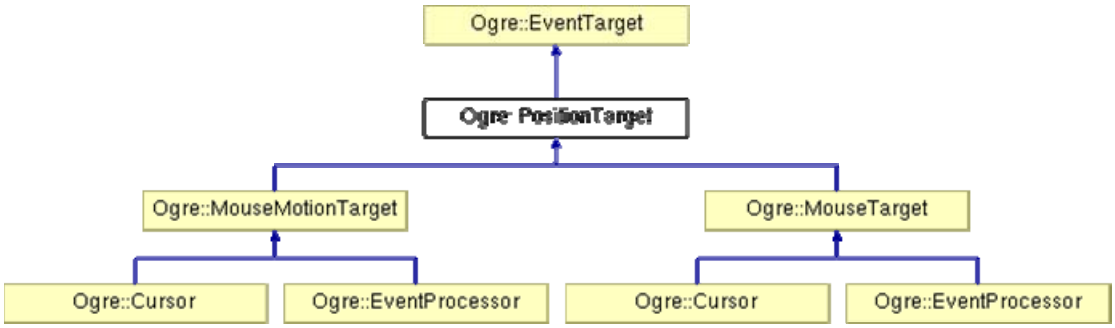
```
bool EventDispatcher::dispatchEvent(InputEvent* e)
{
    bool ret = false;
    if (e->isEventBetween(MouseEvent::ME_FIRST_EVENT,
              MouseEvent::ME_LAST_EVENT))
    {
      MouseEvent* me = static_cast<MouseEvent*> (e);
      ret = processMouseEvent(me);
    }
    else if (e->isEventBetween(KeyEvent::KE_FIRST_EVENT,
              KeyEvent::KE_LAST_EVENT))
    {
      KeyEvent* ke = static_cast<KeyEvent*> (e);
      ret = processKeyEvent(ke);
    }
    return ret;
}
```

至此，EventDispatcher 把 InputEvent 消息分成 MouseEvent 和 KeyEvent 分别处理。

先看键盘消息：

```
bool EventDispatcher::processKeyEvent(KeyEvent* e)
{
    if (mKeyCursorOn != 0)
    {
        mKeyCursorOn->processEvent(e); //PositionTarget:: processEvent(e)
    }
    return e->isConsumed();
}
```

键盘消息被分发给 Cursor 当前所在的 PositionTarget，由 PositionTarget 处理。而 PositionTarget 是个抽象类，需要其子类才能处理消息：



再看鼠标消息：

```
bool EventDispatcher::processMouseEvent(MouseEvent* e)
{
    PositionTarget* targetOver;

    mMouseX = e->getX();
    mMouseY = e->getY();

    targetOver = mTargetManager->getPositionTargetAt(e->getX(), e->getY());
    trackMouseEnterExit(targetOver, e);

    switch (e->getID())
    {
    case MouseEvent::ME_MOUSE_PRESSED:
        mDragging = true;
        if (mDragDropOn)
            mDragDropActive = true;
        mMouseDragSource = targetOver;
        retargetMouseEvent(targetOver, e);
        trackKeyEnterExit(targetOver, e);
        break;

    case MouseEvent::ME_MOUSE_RELEASED:
```

```
        if (targetOver != 0)
          {
           if (targetOver == mMouseDragSource)
             {
              retargetMouseEvent(mMouseDragSource, MouseEvent::ME_MOUSE_CLICK
ED, e);
              retargetMouseEvent(mMouseDragSource, e);
           }
           else // i.e. targetOver != mMouseDragSource
             {
              if (mDragDropActive)
                 retargetMouseEvent(targetOver, MouseEvent::ME_MOUSE_DRAGDROPP
ED, e);
              retargetMouseEvent(mMouseDragSource, e);
              retargetMouseEvent(targetOver, MouseEvent::ME_MOUSE_ENTERED, e);
           }
        }
        else
           retargetMouseEvent(mMouseDragSource, e);

        mDragging = false;
        mDragDropActive = false;
        mMouseDragSource = 0;
        break;

      case MouseEvent::ME_MOUSE_MOVED:
      case MouseEvent::ME_MOUSE_DRAGGED:
        if (!mDragging || targetOver == mMouseDragSource)
          {
           retargetMouseEvent(targetOver, e);
        }
        else // i.e. mDragging && targetOver != mMouseDragSource
          {
           retargetMouseEvent(mMouseDragSource, MouseEvent::ME_MOUSE_DRAGGE
D, e, true);
           if (mDragDropActive)
              retargetMouseEvent(targetOver, MouseEvent::ME_MOUSE_DRAGMOVE
D, e);
        }
        break;
     }

     return e->isConsumed();
  }
```

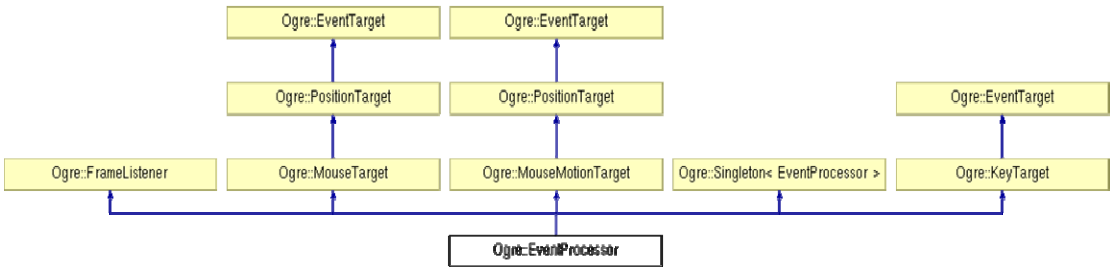
```
void EventDispatcher::retargetMouseEvent(PositionTarget* target, MouseEvent* e)
{
    if (target == NULL)
    {
        return;
    }

    MouseEvent* retargeted = new MouseEvent(target,
                                e->getID(),
                                e->getButtonID(),
                                e->getWhen(),
                                e->getModifiers(),
                                e->getX(),
                                e->getY(),
                                e->getZ(),
                                e->getClickCount());

    target->processEvent(retargeted);
    delete retargeted;

    e->consume();
}
```

可见所有的消息经过 EventDispatcher 分发后都是由各种 target 处理的。若 target 处理不了，则由 EvenetProcessor 自行处理。而 EvenetProcessor 本身是也是 Target 的一种：



所有，用户消息默认的情况下（在没有其他的 Target 的时候），是由 EvenetProcessor 处理的，再看 EventProcessor::processEvent(InputEvent* e)：

```
void EventProcessor::processEvent(InputEvent* e)
{
    // try the event dispatcher list
    for (DispatcherList::iterator i = mDispatcherList.begin();
            i != mDispatcherList.end(); ++i )
    {
        (*i)->dispatchEvent(e);
    }
```

```
        // try the event target list
    if (!e->isConsumed())
      {
        EventTargetList::iterator i, iEnd;

        iEnd = mEventTargetList.end();
        for (i = mEventTargetList.begin(); i != iEnd; ++i )
          {
           (*i)->processEvent(e);
        }
    }

    if (!e->isConsumed())
      {
        switch(e->getID())
          {
        case MouseEvent::ME_MOUSE_PRESSED:
        case MouseEvent::ME_MOUSE_RELEASED:
        case MouseEvent::ME_MOUSE_CLICKED:
        case MouseEvent::ME_MOUSE_ENTERED:
        case MouseEvent::ME_MOUSE_EXITED:
        case MouseEvent::ME_MOUSE_DRAGENTERED:
        case MouseEvent::ME_MOUSE_DRAGEXITED:
        case MouseEvent::ME_MOUSE_DRAGDROPPED:
            processMouseEvent(static_cast<MouseEvent*>(e));
            break;
        case MouseEvent::ME_MOUSE_MOVED:
        case MouseEvent::ME_MOUSE_DRAGGED:
        case MouseEvent::ME_MOUSE_DRAGMOVED:
            processMouseMotionEvent(static_cast<MouseEvent*>(e));
            break;
        case KeyEvent::KE_KEY_PRESSED:
        case KeyEvent::KE_KEY_RELEASED:
        case KeyEvent::KE_KEY_CLICKED:
            processKeyEvent(static_cast<KeyEvent*>(e));
            break;
        }
    }
}
```

可见，对不同的消息，EventProcessor 分别进行了处理。而上面三个高亮的函数，是 EventProcessor 继承其父类而来的。以 processKeyEvent 为例（其他的两个类似），它是由 KeyTarget 处理的：

```
void KeyTarget::processKeyEvent(KeyEvent* e)
{
    // Remove all marked listeners
    std::set<KeyListener*>::iterator i;
    for (i = mRemovedListeners.begin(); i != mRemovedListeners.end(); i++)
    {
        mKeyListeners.erase(*i);
    }
    mRemovedListeners.clear();

    // Tell all listeners
    for (i= mKeyListeners.begin(); i != mKeyListeners.end(); i++)
    {
        KeyListener* listener = *i;
        if (listener != 0)
        {
            int id = e->getID();
            switch(id)
            {
            case KeyEvent::KE_KEY_PRESSED:
                listener->keyPressed(e);
                break;
            case KeyEvent::KE_KEY_RELEASED:
                listener->keyReleased(e);
                break;
            case KeyEvent::KE_KEY_CLICKED:
                listener->keyClicked(e);
                break;
            }
        }
    }
}
```

由此可见，经过长途跋涉用户消息最终分发给了 Listener。若要处理键盘和鼠标消息，就得向 EventProcessor 注册相应的 Listeners。

2、图形更新消息

在《OGRE 分析之场景渲染》中，已经分析过，得到更新消息是按如下路径传递的：

Root →RenderSystem → RenderTarget → Viewport → Camera → SceneManager →RenderSystem

OGRE 分析之设计模式（四）

Mythma
http://www.cppblog.com/mythma
Email: mythma@163.com

OGRE 的设计结构十分清晰，这得归功于设计模式的成功运用。

八、Iterator

说到 Iterator，让人首先想到的是 STL 中各种 iterators。OGRE 源码中广泛用到了 STL，尤其是容器 map。但 OGRE 大部分情况下并没有直接使用与容器配套的迭代器，而是在 iterator 上包了一层。对序列式容器的 iterator，OGRE 包装为 VectorIterator<T>，其 const 形式为 ConstVectorIterator；对关联式容器（map），包装为 MapIterator<T>，其 const 形式为 ConstMapIterator。所以从另一个角度看，使用的是 Adapter 模式。

OGRE 的包装本身没有什么复杂，看一下 map 的 iterator 封装就清楚了：

```
template <class T>
class MapIterator
{
private:
    typename T::iterator mCurrent;
    typename T::iterator mEnd;
    /// Private constructor since only the parameterised constructor should be used
    MapIterator() {};
public:
    typedef typename T::mapped_type MappedType;
    typedef typename T::key_type KeyType;
    /** Constructor.
    @remarks
        Provide a start and end iterator to initialise.
    */
    MapIterator(typename T::iterator start, typename T::iterator end)
        : mCurrent(start), mEnd(end)
    {
    }
    /** Returns true if there are more items in the collection. */
    bool hasMoreElements(void) const
    {
        return mCurrent != mEnd;
    }
    /** Returns the next value element in the collection, and advances to the nex
t. */
    typename T::mapped_type getNext(void)
    {
        return (mCurrent++)->second;
    }
```

```
        /** Returns the next value element in the collection, without advancing to the ne
xt. */
        typename T::mapped_type peekNextValue(void)
        {
            return mCurrent->second;
        }
        /** Returns the next key element in the collection, without advancing to the nex
t. */
        typename T::key_type peekNextKey(void)
        {
            return mCurrent->first;
        }
        /** Required to overcome intermittent bug */
        MapIterator<T> & operator=( MapIterator<T> &rhs )
        {
            mCurrent = rhs.mCurrent;
            mEnd = rhs.mEnd;
            return *this;
        }
        /** Returns a pointer to the next value element in the collection, without
            advancing to the next afterwards. */
        typename T::pointer peekNextValuePtr(void)
        {
            return &(mCurrent->second);
        }
        /** Moves the iterator on one element. */
        void moveNext(void)
        {
            mCurrent++;
        }
};
```

九、Observer

Observer 模式“定义对象间一对多的依赖关系，当一个对象的状态发生变化时，所有依赖他的对象都得到通知并自动更新”。回想一下 OGRE 的消息机制，用的正是该模式。

为了得到 OGRE 的各种消息（更新、鼠标、键盘），在初始化 EventProcessor 后需要向它添加各种 Listeners：KeyListener、MouseListener、MouseMotionListener。而 EventProcessor 本身又是个 FrameListener，在它 startProcessingEvents 的时候，又以 FrameListener 的身份注册到 Root 中。可以看出，Root 是消息的**发布者**，EventProcessor 是个**代理**，它把消息分发给各种**订阅者** KeyListener、MouseListener 或 MouseMotionListener。

至于消息是如何分发的，可以参考 Chain of Responsibility 模式或消息机制分析。

十、Strategy

Strategy 模式在于实现算法与使用它的客户之间的分离，使得算法可以独立的变化。

回想一下 Bridge 模式，可以发现，两者之间有些相似性：使得某一部分可以独立的变化。只不过 Bridge 是将抽象部分与它的实现部分分离。从两者所属的类别来看，Bridge 强调静态结构，而 Strategy 强调更多的是行为——算法的独立性。

同样是 Bridge 模式中的例子，若把 Mesh 各版本文件读取的实现看作是算法，把 MeshSerializer 看作是算法的客户，那么该例也可以看作是 Strategy 模式。具体参考 Bridge 模式。

从上面可以看出，模式之间本没有绝对的界限，从不同的角度看可以得到不同的结论；另一方面，模式的实现也是随机应变，要与具体的问题想结合。

十一、Template Method

Template Method 比较简单的一个模式，属于类行为模式。可以用“全局与细节”、“步骤与实现”来概括，具体就是基类定义全局和步骤，子类来实现每一步的细节。

OGRE 给的 Example 框架使用了该模式，并具代表性。看一下 ExampleApplication 的 setup()成员：

```
bool setup(void)
{
    mRoot = new Root();

    setupResources();

    bool carryOn = configure();
    if (!carryOn) return false;

    chooseSceneManager();
    createCamera();
    createViewports();

    // Set default mipmap level (NB some APIs ignore this)
    TextureManager::getSingleton().setDefaultNumMipmaps(5);

    // Create any resource listeners (for loading screens)
    createResourceListener();
    // Load resources
    loadResources();

    // Create the scene
    createScene();

    createFrameListener();
```

```
        return true;

    }
```

该成员函数调用的其他 virtual 成员函数都有默认的实现，若不满足需求，子类可以自行实现。而 setup（）只是定义了一个设置顺序。