# OGRE 分析之设计模式（二）

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

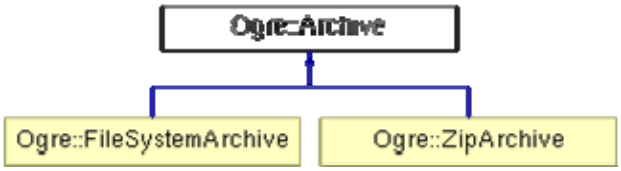
OGRE 的设计结构十分清晰，这得归功于设计模式的成功运用。

## 四、Adapter

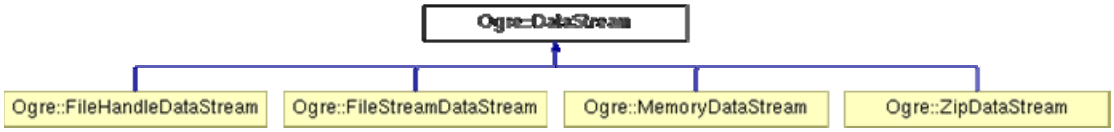
Apdater 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。当使用第三方工具包时候，这种模式经常用到。OGRE 使用了其他的几个开源项目，如 Cg、freetype、ode、zzip 等，与其他接口肯定会有不兼容的情况，这就需要 Adapter。

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。

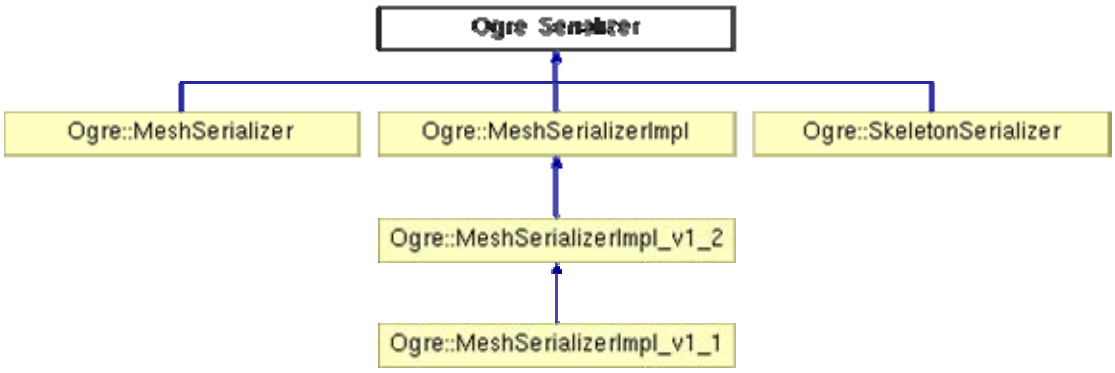



可以看出，Adapter 模式与 Abstract Factory 模式可以实现相同的效果——让别的接口兼容（回顾一下用 Abstract Factory “适配”不同的 RenderSystem）,但是两者的区别在于 Abstract Factory 是用于创建不同类别的对象；而 Adapter 没有创建别的对象的功能。所以 GoF 把 Abstract Factory 分在创建型模式类别中，而 Adapter 在结构型模式中。可以发现，这两种模式，OGRE 的文件系统都使用了。

## 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();
    }
```