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：

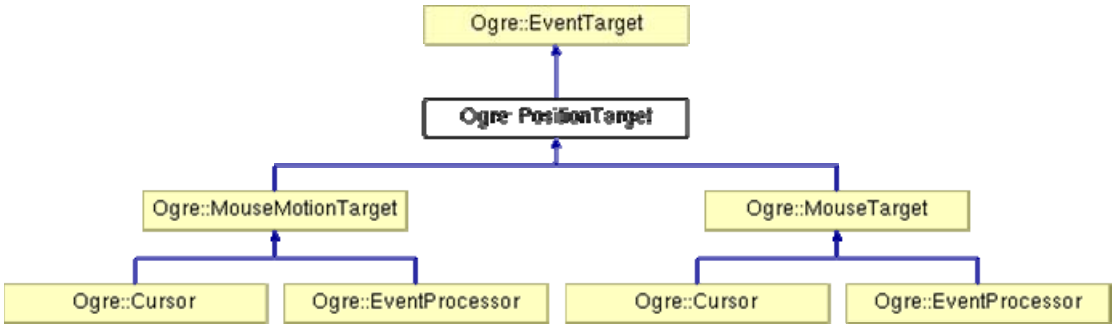
```
bool EventDispatcher::dispatchEvent(InputEvent* e)
{
    bool ret = false;
    if (e->isEventBetween(MouseEvent::ME_FIRST_EVENT,
              MouseEvent::ME_LAST_EVENT))
    {
      MouseEvent* me = static_cast<MouseEvent*> (e);
      ret = processMouseEvent(me);
    }
    else if (e->isEventBetween(KeyEvent::KE_FIRST_EVENT,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

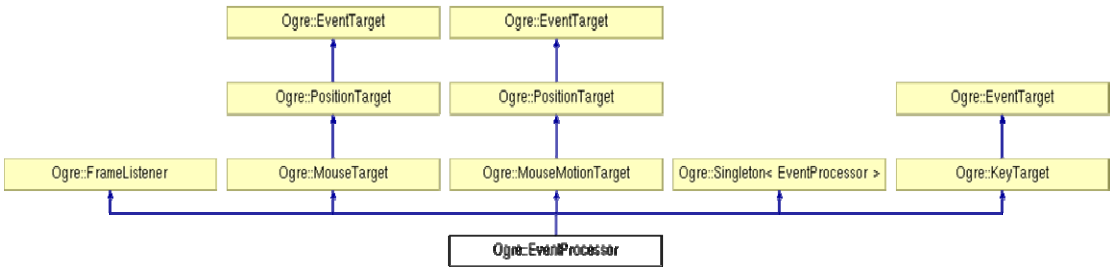
```
void EventDispatcher::retargetMouseEvent(PositionTarget* target, MouseEvent* e)
{
    if (target == NULL)
    {
        return;
    }

    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