

OGRE 的消息机制

盛崇山

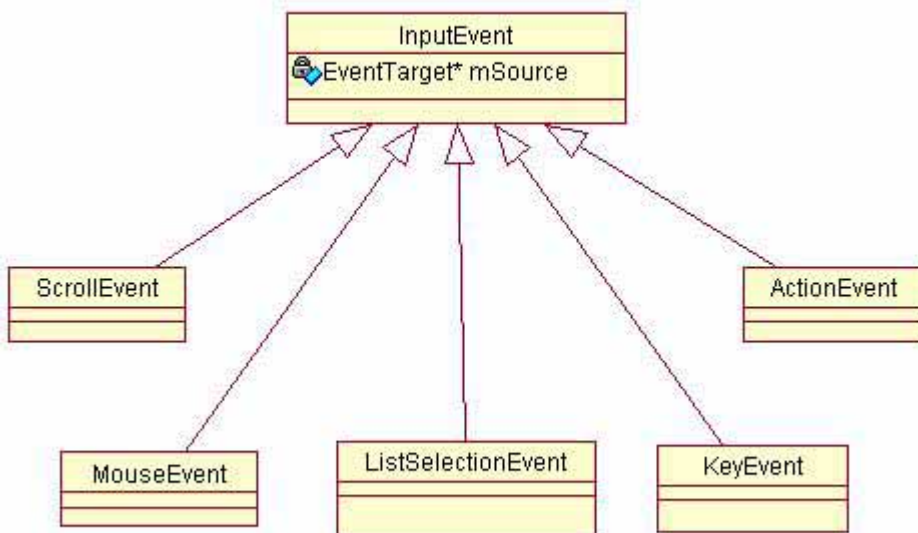
<http://antsam.blogone.net>

AntsamCGD@hotmail.com

消息机制的设计一般总要设计到三个部分：消息的产生、消息的传递和消息的处理。这里主要分析 OGRE 中的消息机制。

OGRE 中的消息处理者的抽象类主要是 listener 类，而 listener 必须是对应特定的 target 的，所以可以认为是由 listener 和 target 两个抽象类组成。而消息传递这是有 Dispatcher 和 Processor 组成。那么消息的产生有那些呢？暂时只分析出 Input Reader。下面介绍大概的结构。

Event 类的 class view 如下所示：

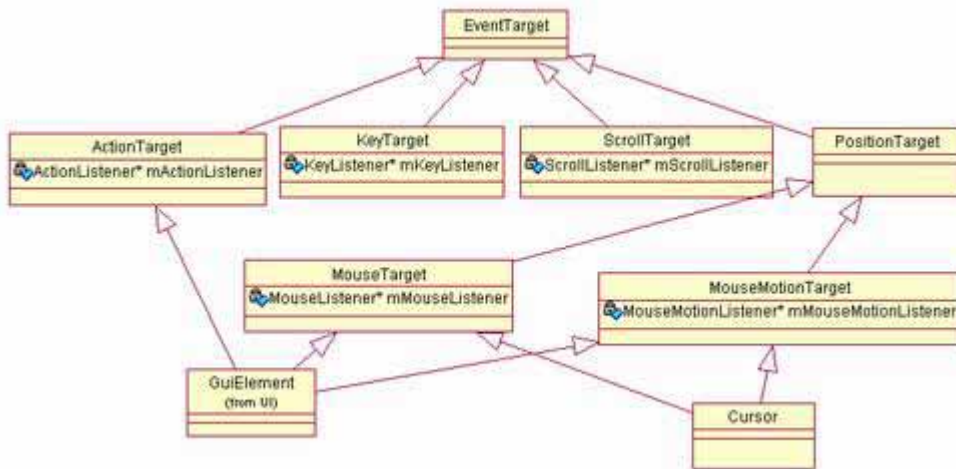


图一：Event Class view

Input Event 是基类，主要包括了一些辅助键（如：ctrl、alt、shift 等）。这些辅助键同鼠标和键盘结合就可以产生一些复合操作。而子类则根据具体的对象定义一些特有的属性，例如：鼠标则具有屏幕坐标属性一些各个按键的状态；而键盘则包括按键的 ascoll 码。其它则包括了各类 GUI 所特有的属性。

同时注意 Event 类中都包含了事件所对应的目标对象指针 EventTarget。

下面我们就介绍 EventTarget 类的 Class View :

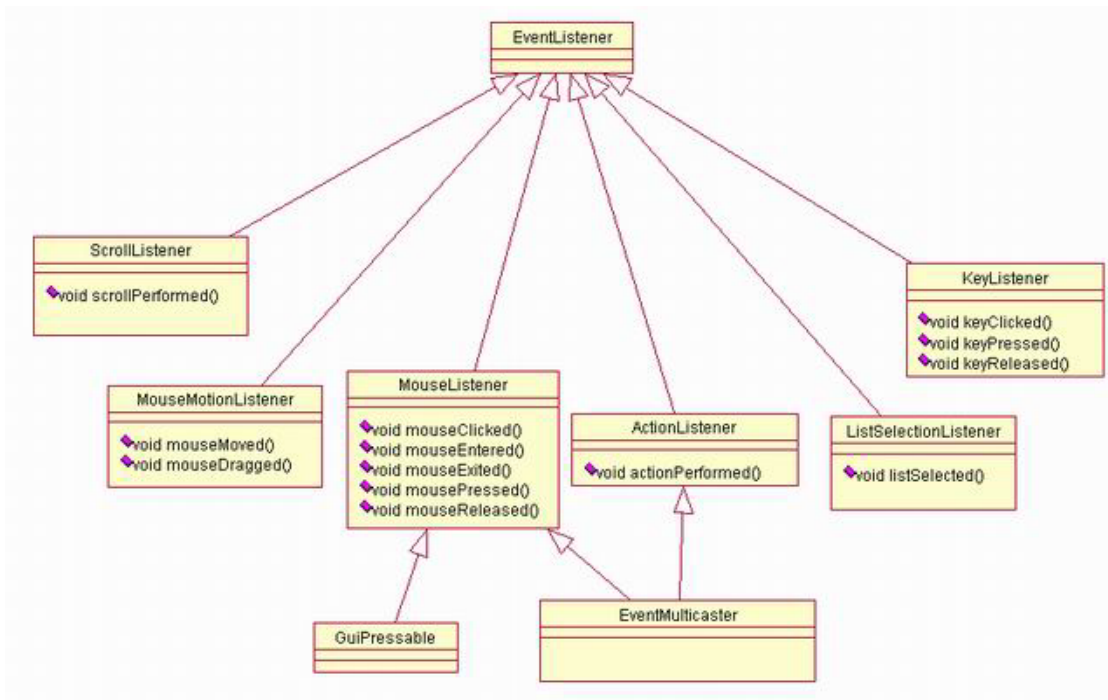


图二： Event Class View

图中上面两层都是抽象对象的共同属性,例如:第二层中的 Key Target 和 Position Target 描述了同键盘和位置相关的 target。第三层中的 MouseEvent、 MouseMotionTarget 描述的更为具体的类。

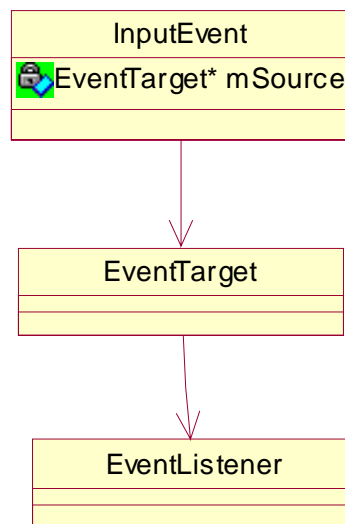
Listener 类 class view 如下所示

Listener 类 class view



上图所定义类都是纯抽象类,如果想实现某些功能必须继承上述类实现自己的消息处理函数, listener 有点像网络中的侦听消息。同时对一些你不感兴趣的消息可以采用默认的处理函数或者直接丢弃。

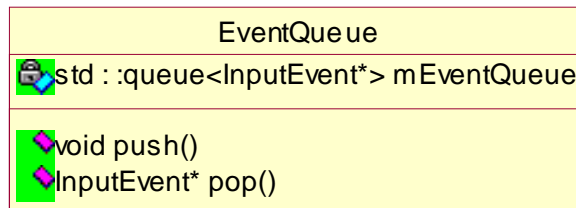
那么上面三者的关系是如何的呢? 它们的关系图:



其实这三者的关系就是设计模式中的 Command 模式。Event Target 就是 Command, Listener 就是 Receiver。而 Input Event 的产生者正好是 Invoker。

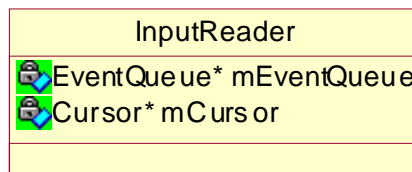
现在分析一下它们是在程序中如何组织以及如何产生和传递的。

Input Event 主要是一 Queue 的形式保存, 如下所示:



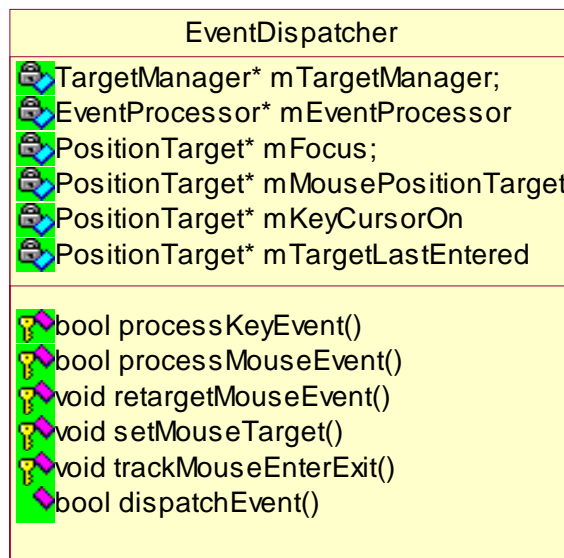
而Target则以 Target Manager 保存 ,注意这个 Target Manager 是一个抽象类 ,不同属性的 target 需要实现不同的 Target Manager ;例如 : Overlay 就有 Overlay Manager (继承了 Target Manager) 如果还有一些 3D 的物体可以作为 Target 的话 ,就需要定义一个 3D Object Manager 了。(注 : 这些 target 在游戏中就表现为具有 visibility 的 ,如 : overlay 就表现为菜单、按钮等一些控件。而 3D 中的 object 的话表现为一些在游戏中的 trigger 或者动态的物体。Listener 的动作是在 Target 内部实现的 ,可以认为是一一对应的 (可能有点不准确) 。但必须注意的是 Input Event 中有 get Source 的接口 ,而在 Event Target 中是没有 get listener 的接口的。

前面提到过 Input Reader 是可以产生消息的 :



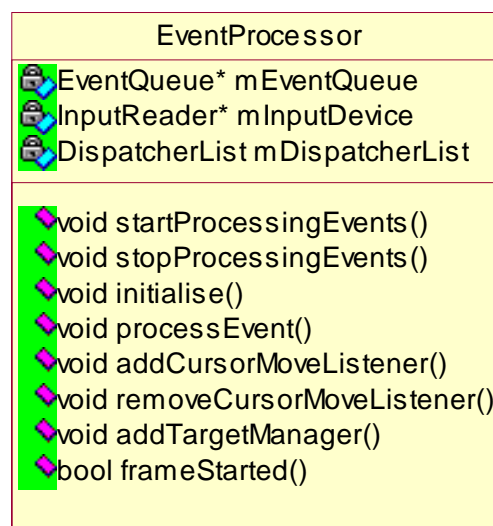
先不分析这个。

消息的传递是由 Dispatcher 和 Processor 组成的 :



从图中我们分析一下 Dispatcher 的组成：Key Cursor 是同鼠标对应的图片（其实对应的对象是 Cursor 而不是 position target），Target Last Enter 是前一帧所交互过的 target，Mouse Position Target 主要是当前鼠标交互的 target，定义这两个主要是为了处理一些鼠标的复杂消息（如：拖动效果）。特别注意的是 Dispatcher 同 Target Manager 是一一对应的。也就是说一个 dispatcher 只是处理一个特定的 Target set 的，主要的处理函数就是 dispatcher Event ()。下面看 Processor 就可以看出为什么 Processor 中有 dispatcher 的 list 了。

Processor 的类结构如下所示：



Processor 中主要的函数是 frame Start：

```
//消息处理
bool EventProcessor::frameStarted(const FrameEvent& evt)
{
    //读取输入设备当前状态
    mInputDevice->capture();
    while (mEventQueue->getSize() > 0)
```










```





{
    InputEvent* e = mEventQueue->pop();
    bool blsConsumed = false;
    //各类target处理当前消息
    //有点类似广播的消息
    for(DispatcherList::iterator i = mDispatcherList.begin();
        i != mDispatcherList.end(); ++i
    {
        blsConsumed |= (*i)->dispatchEvent(e);
    }
    //默认处理
    if (!blsConsumed)
    {
        processEvent(e);    // nothing used it, so use default processing
    }
    delete e;    // created from dispatch;
}
return true;
}

```

必须注意的是 Processor 是从 Mouse Target 、 Key Target、 Mouse Motion Target 等继承而来，所以上面函数中的默认函数其实就是父类中的处理函数（有内部成员变量 listener 来处理）。但是只要你看 listener 的代码就可以知道，里面的处理函数全部是纯虚函数。也就是说 processor 只是帮你把具体的消息发送到相应的 listener 中，而不做任何处理，如果想处理，必须从相应的 listener 中继承实现一个类。同时必须在初始化的时候把这些 listener 挂靠到相应的 target 中,后面会分析一个具体的例子来说明这一点。

我们还有必要在这里分析另外两个类 Cursor 和 Input Reader。

InputReader	
	EventQueue* mEventQueue
	Cursor* mCursor
	void getMouseState()
	bool getMouseButton()
	void mouseMoved()
	void createMouseEvent()
	void triggerMouseButton()
	void createKeyEvent()
	void keyChanged()

Cursor	
	Real mMouseX, mMouseY, mMouseZ
	Real mRelX, mRelY, mRelZ
	Real mXLowLimit, mXHighLimit, mYLowLimit,
	Real mYHighLimit, mZLowLimit, mZHighLimit

Cursor 鼠标图片的属性，

它继承了 Mouse target 和 mouse motion target。所以也会有一个 listener (这个 target 其实就是 Cursor GUI element)。

Input Reader 前面已经说过是可以产生 event 的，有一个函数我们必须分析一下：

```
void InputReader::useBufferedInput(EventQueue* pEventQueue,
                                   bool keys, bool mouse)
{
    mEventQueue = pEventQueue;
    if (mCursor)
        delete mCursor;
    mCursor = new Cursor();
    // initial states of buffered don't call setBufferedInput
    // because that can be overridden (in the future) to save releasing and
    // acquiring unchanged inputs
    // if we ever decide to release and acquire devices
    mUseBufferedKeys = keys;
    mUseBufferedMouse = mouse;
}
```

注意 Input Reader 产生的 Event 存放的 Queue 是通过指针初始化的，而并不是在内部申请内存得到的。那继续跟踪一下：

```
//初始化成员变量
void EventProcessor::initialise(RenderWindow* ren)
{
    cleanup();
    mEventQueue = new EventQueue();
    mInputDevice = PlatformManager::getSingleton().createInputReader();
    mInputDevice->useBufferedInput(mEventQueue);
}
```

```

        mInputDevice->initialise(ren,true, true, false);
    }

```

也就是说，Input Reader 产生的消息是存放在 processor 的 event queue 中的，也就是说在 frame start 中将被所有的 dispatcher 发送到 target，然后到所有 listener。

下面具体分析一个 GUI demo 来理解整个消息机制的实现，可以参看 Demo_Gui。

现在分析一下它所有的 event、target、listener。

- Target：三个按钮、一个 list、还有一个是 cursor GUI。
- Event：主要有 mouse event、mouse motion event、以及 action event（主要是用来处理空间类的高级消息的）。
- Listener 除了一些默认的消息外，就只有 Example Frame Listener、GUI Frame Listener、GUI Application。

下面这部分代码说明了如何把 target 同 listener 联系在一起的。

```

        ActionTarget* at =
static_cast<BorderButtonsGuiElement*>(GuiManager::getSingleton().getGuiElement("SS/Setup/HostScreen/Join"));
        at->addActionListener(this);
        at =
static_cast<BorderButtonsGuiElement*>(GuiManager::getSingleton().getGuiElement("SS/Setup/HostScreen/Exit"));
        at->addActionListener(this);
.....
        ListChanger* list =
static_cast<ListGuiElement*>(GuiManager::getSingleton().getGuiElement("SS/Setup/HostScreen/AvailableGamesList"));
        (GuiManager::getSingleton().getGuiElement("Core/CurrFps"))->addMouseListener(this);

```

那么为什么会要联系在一起呢？

我们下面分析一下消息的处理过程你就会知道为什么，前面分析过一个 EventProcessor::frameStarted 里面主要是调用了两个函数：dispatcher 和默认的 processor event。现在先看看 dispatcher：

```

bool EventDispatcher::dispatchEvent(InputEvent* e)
{
.....
{
    MouseEvent* me = static_cast<MouseEvent*>(e);
    ret = processMouseEvent(me);
}
}

```

```

    }
.....
    {
        KeyEvent* ke = static_cast<KeyEvent*>(e);
        ret = processKeyEvent(ke);
    }
.....

```

调用了 processMouseEvent，函数如下所示：

```

bool EventDispatcher::processMouseEvent(MouseEvent* e)
{.....
    mMousePositionTarget->processEvent(e);
    targetOver = mTargetManager->getPositionTargetAt(e->getX(), e->getY());
.....
    setMouseEvent(targetOver, e);
.....
}

```

现在我们分析上面三个函数，第一个 process event，但是如果你查询一下这个函数，你会发现，这是一个纯虚函数，那我们先来看第二个会返回一个 GUI 类，那第三个函数如下所示：

```

void EventDispatcher::setMouseEvent(PositionTarget* target, MouseEvent* e)
{
    if (target != mMousePositionTarget)
    {
        mMousePositionTarget = target;
    }
}

```

所以第一个调用的是 GUI 类的 process event，那现在让我们看看 GUI 中做了些什么：

```

void GuiElement::processEvent(InputEvent* e)
{
    if (!mEnabled || e->isConsumed())
    {
        return;
    }
    switch(e->getID())
    {
        case ActionEvent::AE_ACTION_PERFORMED:

```

```

        processActionEvent(static_cast<ActionEvent*>(e));
        break;
    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:
        processMouseEvent(static_cast<MouseEvent*>(e));
        break;
    case MouseEvent::ME_MOUSE_MOVED:
    case MouseEvent::ME_MOUSE_DRAGGED:
        processMouseMotionEvent(static_cast<MouseEvent*>(e));
        break;
    }
}
}

```

上面用红色标出的三个函数都是直接调用父类中的函数的。

```

void ActionTarget::processActionEvent(ActionEvent* e)
{
    ActionListener* listener = mActionListener;
    if (listener != NULL)
    {
        int id = e->getID();
        switch(id)
        {
            case(ActionEvent::AE_ACTION_PERFORMED:
                listener->actionPerformed(e);
                break;
            }
        }
    }
}

void MouseTarget::processMouseEvent(MouseEvent* e)
{
    MouseListener* listener = mMouseListener;
    if (listener != NULL)
    {
        int id = e->getID();
        switch(id)
        {
            case(MouseEvent::ME_MOUSE_PRESSED:
                listener->mousePressed(e);
                break;
            case(MouseEvent::ME_MOUSE_RELEASED:
                listener->mouseReleased(e);

```

```

        break;
    case MouseEvent::ME_MOUSE_CLICKED:
        listener->mouseClicked(e);
        break;
    case MouseEvent::ME_MOUSE_EXITED:
        mMouseWithin = false;
        listener->mouseExited(e);
        break;
    case MouseEvent::ME_MOUSE_ENTERED:
        mMouseWithin = true;
        listener->mouseEntered(e);
        break;
    }
}

void MouseMotionTarget::processMouseEvent(MouseEvent* e)
{
    MouseMotionListener* listener = mMouseMotionListener;
    if (listener != NULL)
    {
        int id = e->getID();
        switch(id)
        {
            case MouseEvent::ME_MOUSE_MOVED:
                listener->mouseMoved(e);
                break;
            case MouseEvent::ME_MOUSE_DRAGGED:
                listener->mouseDragged(e);
                break;
        }
    }
}

```

但是如果查询上面这些 listener 调用的函数，你会发现全是纯虚的，如果再往上面想想，我们就会发现，在把 Target 和 listener 联系在一起的时候。已经把 GuiApplication 赋值给 listener 了，那在查看一下 GuiApplication 中的函数，你会发现都定义了上述 listener 调用的函数：

```

void mouseClicked(MouseEvent* e) {}
void mouseEntered(MouseEvent* e)
{
    int i =5;
}

```

```
void mouseExited(MouseEvent* e) {}
void mousePressed(MouseEvent* e) {}
void mouseReleased(MouseEvent* e) {}
void actionPerformed(ActionEvent* e)
{
    // Think about doing something here
    std::string action = e->getActionCommand();
    LogManager::getSingleton().logMessage("Got event: " + action);

    if (action == "SS/Setup/HostScreen/Exit")
    {
        // Queue a shutdown
        static_cast<GuiFrameListener*>(mFrameListener)->requestShutdown();
    }
}
```

所以实际调用的都是 `GuiApplication` 定义的函数。

最后一个问题就是如何调用 `processor` 中的 `frame start`