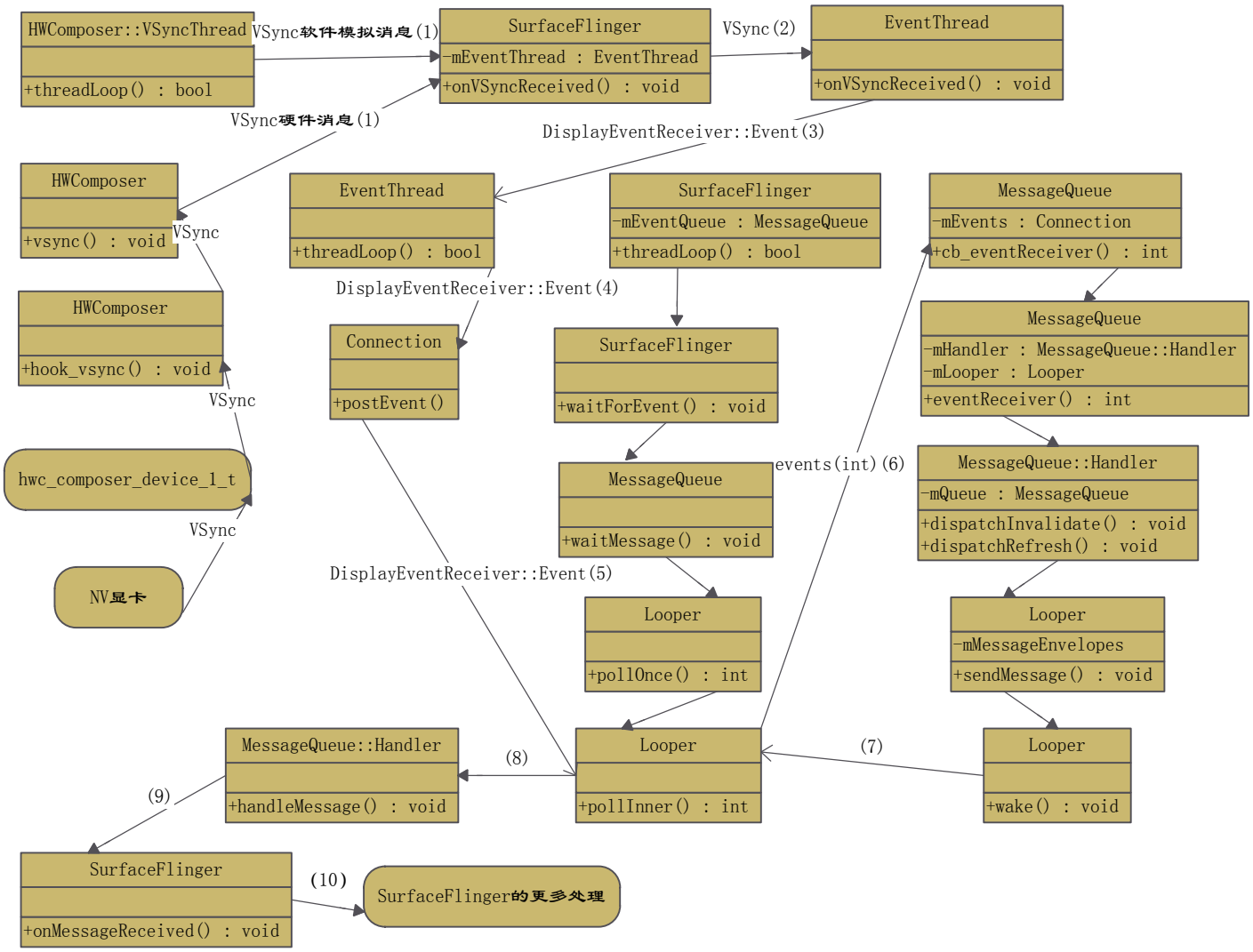

从扮演的角色上来看，EventThread是Server，不断地往BitTube中写入数据;而MessageQueue是Client,负责读取数据。MessageQueue如何得知有Event到来，然后去读取它呢？答案就是它们之间的数据读写模式采用的是Socket(AF_UNIX域)。在MessageQueue::setEventThread这个函数的末尾，通过Looper添加了一个fd，这实际上就是Socket pair中的接收端。然后Looper将这个fd与其callback函数(即MessageQueue::cb_eventReceiver)加入全局的mRequests进行管理。

KeyedVector<int, Request> mRequests;

这个Vector会集中所有需要监测的fd，这样当Looper进行pollInner时，只要有事件需要处理，它就可以通过回调函数通知“接收者”。

总结：当EventThread通过了一个注册了的连接Connection送来的VSync事件(使用DisplayEventReceiver::Event来描述)，将唤醒睡眠等待消息的SurfaceFlinger线程，于是指定的回调函数MessageQueue::cb_eventReceiver将会在pollInner中被调用，这将最终调用Looper的成员方法sendMessage发送一条消息到Vector<MessageEnvelope>，这是按照处理消息时间的大小来进行排序的，这条消息将再次唤醒睡眠在pollOnce上的SurfaceFlinger线程，于是SurfaceFlinger线程再次醒来，将在pollInner中调用MessageQueue::Handler的handleMessage去处理消息，从而调用SurfaceFlinger的成员方法onMessageReceived来进行更多的处理。

从以上可以看出，在收到VSync事件通知后，做两次处理：

1. 第一次只是简单地发送了一个消息，引发第二次对它进一步处理，因为第一次的操作非常简单，故可以快速的响应收到的事件。
2. 第二次将会将相对耗时的操作任务添加到队列中留待第二次处理，若第二次的操作相对较慢，而又快速地收到很多事件，则会在队列中存放多个待处理的任务。

注意：这个过程类似于Linux驱动程序对中断处理分为上半部和下半部两个部分进行处理。这样做的好处就是可以防止程序阻塞EventThread线程使其无法接受新的消息，SurfaceFlinger线程首先快速的响应完消息后告诉EventThread线程赶快工作，而它自身能够将这些事件按照处理时间的优先级进行排序逐个的处理这些耗时操作。