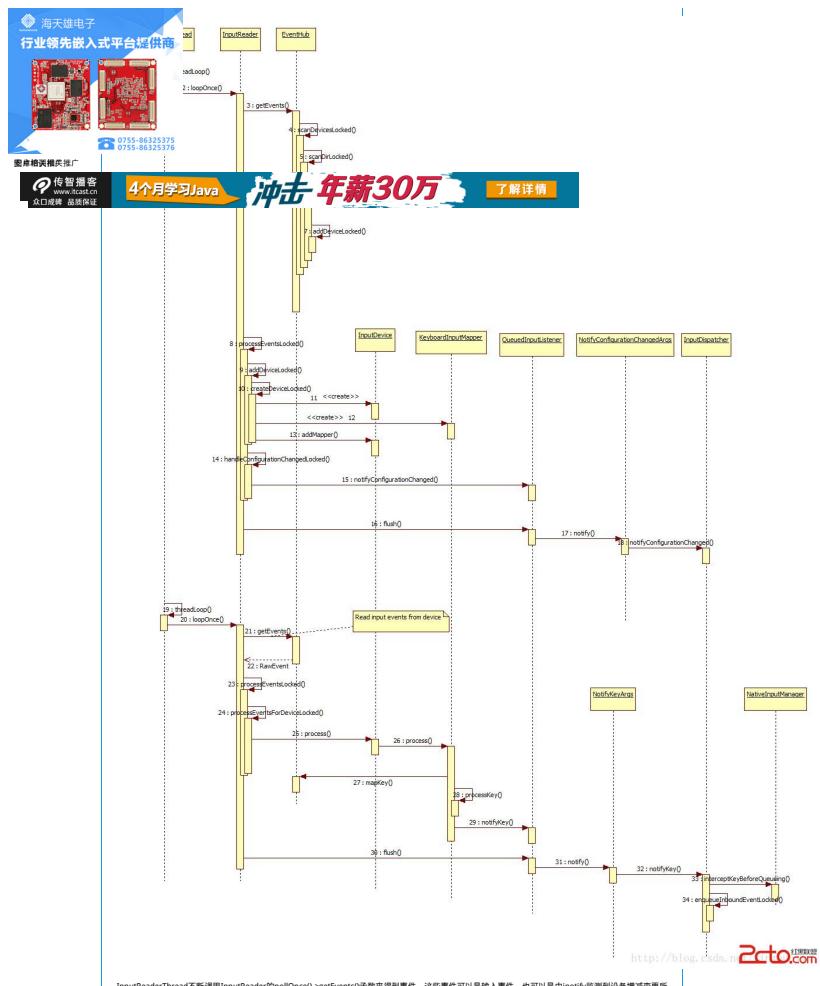


福利 ★
紧急求助
文章推荐



<http://blog.csdn.net> 2cto.com

InputReaderThread不断调用InputReader#pollOnce()>getEvents()函数来得到事件。这些事件可以是输入事件，也可以是由notify监听到设备变更而更新所触发的事件。第一次进入时会扫描/dev/input目录来建立设备列表，存在mDevice成员变量中(EventHub中有设备列表keyedVector mDevices；对应的是，InputReader中也有设备列表keyedVector mDevices，这里先添加前者，然后会在InputReader#addDeviceLocked()中添加后者。)。同时将增加的InputDevice对象放入队列中。在InputReader#pollOnce()方法中，如果需要处理输入，同时会调用InputDispatcher#processEventsLocked()方法。从InputDispatcher#processEventsLocked()方法中，getEvent()方法调用InputReader#processEventsForDeviceLocked()方法。对于设备变更，会根据实际情况调用InputDispatcher#addDeviceLocked()、removeDeviceLocked()和HandleConfigurationChangedLocked()。对于其它设备中的输入事件，会调用processEventsForDeviceLocked()进一步处理。其中会根据当时注册的InputMapper对事件进行处理，然后将事件处理请求放入缓冲队列（QueuedInputListener中的mArgsQueue）。

159 void QueuedInputListener.notifyKey(const NotifyKeyArgs& args) {

160 mArgsQueue.push(mInnerListener.getNotifyKeyArgs(args));

161 }

在InputReader的loopOnce()的结尾会调用QueuedInputListener::flush()统一将缓冲队列中各元素的notify()接口：

171 void QueuedInputListener::flush() {

172 size_t count = mArgsQueue.size();

173 for (size_t i = 0; i < count; i++) notify(mInnerListener);

174 delete args;

175 }

176 mArgsQueue.clear();

177 }

178 }

179 }

以按键事件为例，最后会调用InputDispatcher的notifyKey()函数中。这里先将参数封装成KeyEvent：

2416 KeyEvent event;

2417 eventInitiateArgs->deviceId, args->source, args->action,

2418 args->flags, keyCode, args->scanCode, metaState, 0,

2419 args->downTime, args->eventTime);

然后把它作为参数调用NativeInputManager的interceptKeyBeforeQueuing()函数，顾名思义，就是在放到待处理队列前看看是不是需要系统处理的系统按键，它会通过InputManager的InterceptKeyBeforeQueuing()方法，最终调到PhoneWindowManager的InterceptKeyBeforeQueuing()。然后，基于输入事件信息创建KeyEntry对象，调用enqueueInboundEventLocked()将之放入队列等待InputDispatcher线程来处理。

2439 KeyEntry* newEntry = new KeyEventArgs->eventTime,

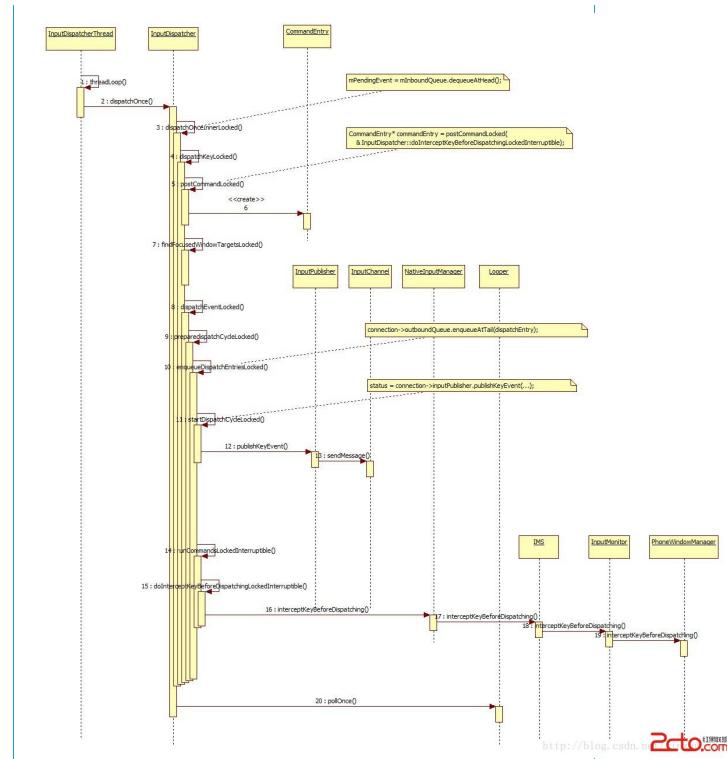
2440 args->deviceId, args->source, flags,

2441 args->action, flags, keyCode, args->scanCode,

2442 metaState, repeatCount, args->downTime,

2443 needWake = enqueueInboundEventLocked(newEntry);

下面该InputDispatcher线程登场了。



可以看到，InputDispatcher的主要任务是将前面收到的输入事件发送到PVM及App端的焦点窗口，前面提到在InputReaderThread中收到事件后会调用notifyKey来通知InputDispatcher，也就是说在mInboundQueue中，在InputDispatcher的dispatchOnce()函数中，会从这个队列拿出处理。

```

234     if (!haveCommandsLocked()) {
235         dispatchOnceInnerLocked(&nextWakeUpTime);
236     }
237     ...
238     if (!runCommandsLocked(interruptible)) {
239         nextWakeUpTime = LONG_LONG_MIN;
240     }
241 }
242 }

其中dispatchOnceInnerLocked()会根据拿出来的EventEntry类型调用相应的处理函数，以Key事件为例会调用dispatchKeyLocked():

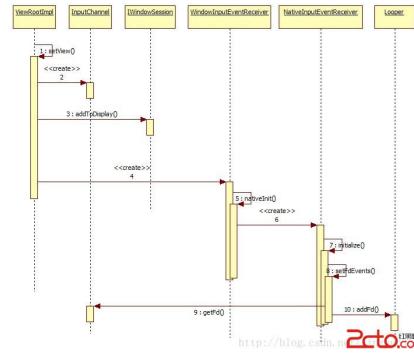
767     CommandEntry* commandEntry = postCommandLocked(
768             &InputDispatcher::dispatchKeyBeforeDispatchingLocked,
769             if (mFocusedWindowHandle != NULL) {
770                 commandEntry->inputWindowHandle = mFocusedWindowHandle;
771             }
772         );
    commandEntry->keyEntry = entry;

791 // Id: #if targets,
792 Vector<InputTarget> targets;
793 int32_t injectionResult = findFocusedWindowTargetsLocked(currentTime,
794     entry, inputTargets, nextWakeUpTime);
795
804 addMonitoringTargetsLocked(inputTargets);
805
806 // Dispatch the key.
807 dispatchEventLocked(currentTime, entry, inputTargets);

```

它会找到目标窗口，然后通过之前和App间建立的连接发送事件。如果是个需要系统处理的Key事件，这里会封装成CommandEntry插入到m_commands队列中，然后调用postCommandLocked方法，该方法会调用dispatchKeyBeforeDispatchingLocked方法让PVM窗口机进行处理。最后dispatchOnce()调用polOnce()和app连接上接收处理器完成消息。那么，InputDispatcher是怎么确定要在哪个窗口中发事件呢？这里的成员变量mFocusedWindowHandle展示了焦点窗口，然后findFocusedWindowTargetsLocked()会调用一系列函数（handleTargetsNotReadyLocked(), checkInjectionPermission(), checkWindowReadyForMoreInputLocked()等）检查mFocusedWindowHandle是否能接收输入事件。如果可以，将之以InputEvent的形式插入到m_commands队列中，然后会调用InputDispatcher::dispatchEventLocked()进行发送。那么，这个mFocusedWindowHandle是如何保护的呢？为了更好地理解，这里部分分析一下窗口连接的管理及焦点窗口的管理。

在App端，新的珀斯窗口需要被注册到VMS中，这是在ViewRootImpl::setView()中的。



其中与输入相关的主要有以下几步：

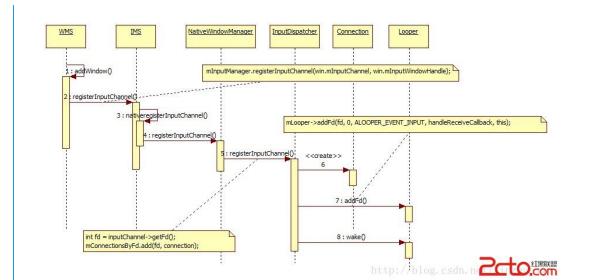
先创建InputChannel，注意还没初始化。

```

521     mInputChannel = new InputChannel();
522     初始化InputChannel，通过VMS的相应接
523     glAddToDisplay();
524     res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
525         getHostVisibility(), mDisplay.getDisplayId(),
526         mAttachInfo.mContentIndex, mInputChannel);

```

VMS会建立与InputDispatcher的连接，流程如下：



ViewRootImpl通过Session中的addToDisplay()会最终调用WMS的addWindow().在WMS中，会创建一对InputChannel，本质上是一对本地socket。然后一个注册到InputDispatcher，一个作为输入参数传给App的ViewRootImpl，这样就建立了App与IMSAF 对连接。

```

2409     if (mInputChannel != null && !mInputChannel.isRunning()) {
2410         & WindowManager.LayoutParams<?> inputParam = new InputParam();
2411         String name = mInputManager.makeInputChannelName();
2412         InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
2413         win.setInputChannel(inputChannels[0]);
2414         inputChannels[1].transferTo(outInputChannel);
2415         mInputManager.registerInputChannel(win.mInputHandle, win.mInputWindowHandle);
2416     }
2417 }

在InputDispatcher::registerInputChannel():

3227     sp<Connection> connection = new Connection(inputChannel, inputWindowHandle, monitor);
3228
3229     int fd = inputChannel->getFd();
3330     mConnectionsByFd.add(fd, connection);
...
3336     mLooper->addFd(fd, 0, ALOOPER_EVENT_INPUT, handleReceiveCallback, this);

```

这里创建的Connection表示一个InputDispatcher到应用窗口的连接。更简单了用于存储的inputChannel, inputPublisher和表示事件接收窗口的inputWindowHandle, 还有两个私有成员，outInputEvent是重复的事件，winFocus是否是焦点但还没有从App接收到通知的。这是因为对于一些事件，Input Dispatcher在App没处理完前一个小时不会发送第二个。mLooper->addFd()将相应的fd放入InputDispatcher等待的集合中，回调函数为handleReceiveCallback()，这就是说InputDispatcher在收到App发来的消息时是调用它进行处理的。最后调用mLooper->wake()把InputDispatcherThread从 epoll_wait()中唤醒。

回到App端，如果前面没有问题，接下来会创建WindowInputEventReceiver，是App的事件接收器。

```
607     mInputEventReceiver = new WindowInputEventReceiver(mInputChannel,
608             Looper.myLooper());
```

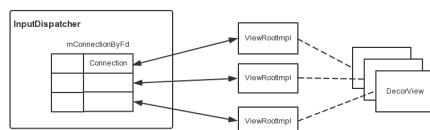
初始化完后，这个连接的线程将挂到主线程的等待fd集合去了(输入EventReceiver::nativeInit())。也就是说，当连接上有消息来，主线程就会调用相应的回调NativeInputEventReceiver::handleEvent()。

接下来初始化App事件处理的流水线，这里使用了Chain of responsibility模式，让事件经过各个InputStage，每一个Stage可以决定是否自己处理，也可以传递给下一家。家的意思是在nativeHandleEvent()可以看到它的方法。

```

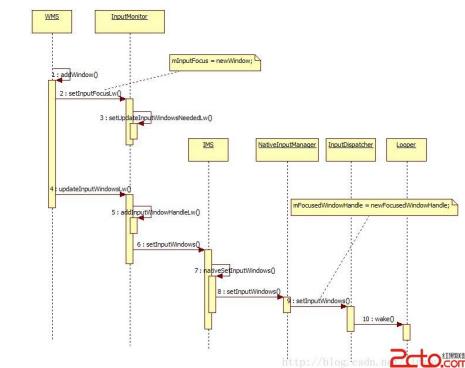
623     // Set the native receiver
624     SequenceCounter suffice = attrs.getTitle();
625     mSyntheticInputStage = new SyntheticInputStage();
626     InputStage viewPostimeStage = new ViewPostimeStage(mSyntheticInputStage);
627     InputStage nativePostimeStage = new NativePostimeInputStage(viewPostimeStage,
628     ...
629     InputStage earlyPostimeStage = new EarlyPostimeInputStage(nativePostimeStage);
630     InputStage imStage = new ImInputStage(earlyPostimeStage,
631     ...
632     suffice += counterSufffix;
633     InputStage viewPrimeInputStage = new ViewPrimeInputStage(imStage);
634     InputStage nativePrimeInputStage = new NativePrimeInputStage(viewPrimeStage,
635     "native-pre-im") + counterSufffix;
636     ...
637     mFirstInputStage = nativePrimeInputStage;
638     mFirstPostimeInputStage = earlyPostimeStage;
```

到这里，可以知道，InputDispatcher会维护WMS中所有窗口的连接，虽然一般只会往焦点窗口发事件。如下所示。



<http://blog.csdn.net>

连接建立后，接下来要考虑WMS如何将焦点窗口信息传给InputDispatcher。举例来说，当新的窗口加入到WMS中，一般焦点会放到新窗口上。来看下WMS的addWindow()函数。



首先，当焦点需要变化时，当焦点窗口变化时，WMS调用

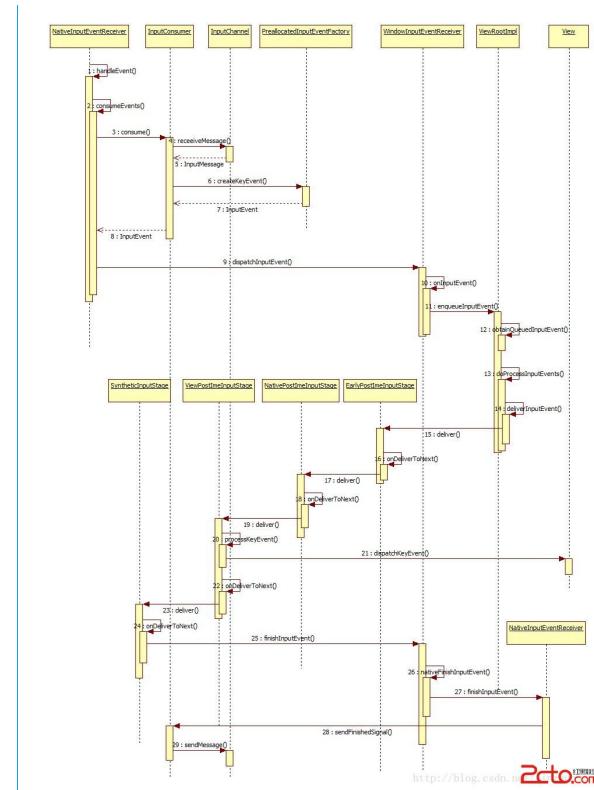
```
mInputMonitor.setInputFocus(w mCurrentFocus, updateInputWindows);  
将焦点窗口放到InputMonitor的mInputFocus中，然后调用
```

```
mInputMonitor.updateInputWindows(wTrue);
```

来创建InputWindowHandle列表，其中被设置焦点窗口的InputWindowHandle的hasFocus会被置位。之后会调用

```
mService.setInputManager(setInputWindowHandles);
```

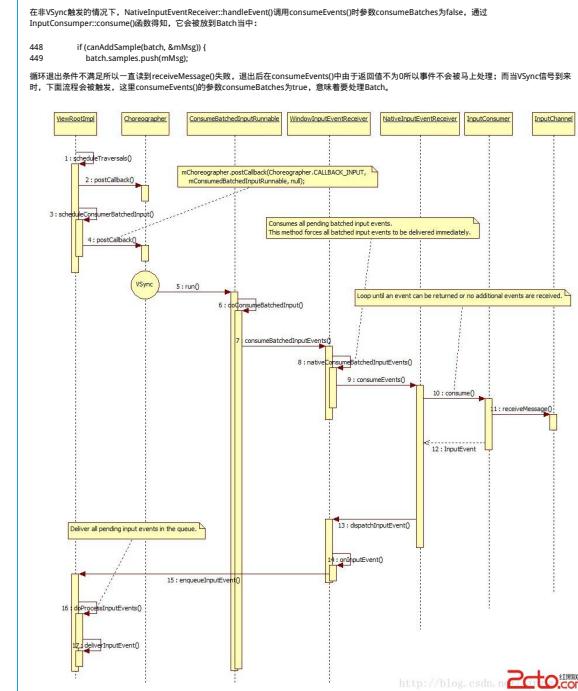
将这些信息传到Native的InputDispatcher，这样的InputDispatcher就能够知道要往哪个窗口传事件。在InputDispatcher的setInputWindows()中，会更新InputDispatcher中的焦点窗口映射。这样，InputDispatcher中就记录了所有焦点窗口信息。当IMSAF的InputDispatcher通过InputChannel发事件到焦点窗口时，NativeInputEventReceiver的handleEvent()会被调用。



基本流程比较直观，先接收事件，然后放入ViewRootImpl的处理队列，然后dispatch给View处理，经过上面提到的一系列InputStage，最后App处理完事件后还需要向IMS发送一个完成信号。

注意上面是用Key事件为例的，对于Motion事件就有差别了。因为触摸滚动中的事件不一定要每一个都处理，因为显示也是60HZ，你如果100HZ的人输入事件，全处理只会浪费计算资源。上面这条路是每帧inputDispatcher有事件发生过来时就会触发的，而对于Motion事件，系统会把它按VSync周期内的事件进行batch，当VSync来临时一起处理。从B开始，App对输入事件的处理是由VSync信号来驱动的，可以看到Recorder中的Sync部分首先处理的就是输入事件。

```
542 doCallbacks(Choreographer.Callback.INPUT, frameTimeNanos);
543 doCallbacks(Choreographer.Callback.ANIMATION, frameTimeNanos);
544 doCallbacks(Choreographer.Callback.TRAVERSAL, frameTimeNanos);
```



ViewRootImpl中调用PendingInputEvent的findPendingInputEvents方法，其中的元素为queuedInputEvent类型。这个类会由PendingInputEvent根据mRequiredInputEvent输入元素，然后向ViewRootImpl的mPendingInputEvents在doProcessInput事件中进行处理，并且处理的方式是前向调用deliverInputEvent，然后调用InputStage的deliver方法进行处理。最后调用finishInputEvent向mForwardingConsumer发送信息，它会调用的是NativeInputEventConsumer的finishInputEvent方法，该方法内部会使用InputConsumer的sendFinishSignal方法发送结束信号来发送信息。

事实上，当Input Resample机制打开时（属性是`io.reactivex.Resample`），对于Motion事件在InputConsumer里的处理会更复杂一点。Android在InputConsumer中加入了对Motion事件的重播。Sync/OffDisplay已完全一站式的完成。系统需要准备下一帧。也就是我们需要知道Sync信号来触发触摸坐标。那么问题来了，输入设备的信号不是按Sync来的，比如是10ms为周期（Sync周期为16.67ms）。那么怎么知道Sync的周期的输入坐标呢？只能靠估算。怎么估呢？估计是将采样点按插值组合的估值方法。详细请参见<http://www.masonchang.blog/2014/25/25/android-touch-sampling-algorithm>

前面提到，InputDispatcher除了给App端发送的，还有任务是处理系统按键。系统中有一些特殊的按键，是系统需要处理的，如音量、电源键等。它是通过`InterceptKeyBeforeDispatching`和`InterceptKeyBeforeUserEvent`两个函数来截获的。`InterceptKeyBeforeDispatching`主要用于处理`Home`、`MenuItem`、`Search`、`InterceptKeyBeforeQueueing`主要处理与输入相关的事件。这样处理的原因是在PMM中，而调用者是在`InputDispatcher`中。所以这个处理的出发点是把与平台无关的东西放在PMM中，而`InputDispatcher`中是平头的东东，所以要定期地去修改PMM。这样，做到了Mechanism and Policy的分离。那么`InputDispatcher`是如何应用到PMM中的呢？首先看`InputDispatcher`中的代码，有九个hook的地方。

```
3510 nsecs_t delay = mPolicy->interceptKeyBeforeDispatching(commandEntry->inputWindowHandle,
3511     &event_entry->policyFlags);
```

这里的mPolicy其实是NativeInputManager， NativeInputManager会通过JNI调用到Java世界中的IMS的相应函数。

1463 // Native callback.

```
1463 // Native callback.  
1464 private long interceptKeyBeforeDispatching(InputEvent event, int policyFlags) {  
1465     KeyEvent keyEvent = event.getKeyEvent();  
1466     if (keyEvent.getKeyCode() == KeyEvent.KEYCODE_DPAD_CENTER  
1467         && (policyFlags & InputEvent.POLICY_FOCUSABLE)  
1468             && !keyEvent.isFromUser()) {  
1469         // 紧急召喚天神  
1470         focus.setFocused(true);  
1471     }  
1472     return event.dispatch();  
1473 }
```

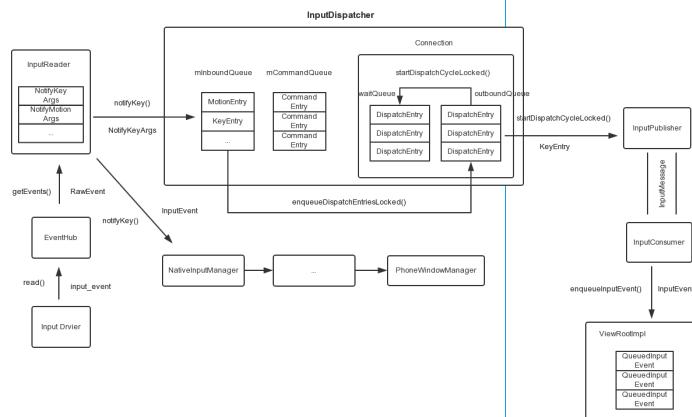
```
1465     KeyEvent event, int policyFlags) {  
1466         Return mWindowManager.callbacks.interceptKeyBeforeDispatch  
Focus, event, policyFlags);
```

正

```

1467 }
这里的mWindowManagerCallback其实是InputMonitor，然后就调用到PWM了。
380 public long interceptKeyBeforeDispatching(
381     InputWindowHandle focus, KeyEvent event, int policyFlags) {
382     WindowState windowState = focus != null ? focus.windowState : null;
383     return mService.mPolicy.interceptKeyBeforeDispatching(windowState, event, policyFlags);
384 }
好了，到这里可以小结一下了。输入事件从Kernel到App和PWM的流程大体如下图所示：

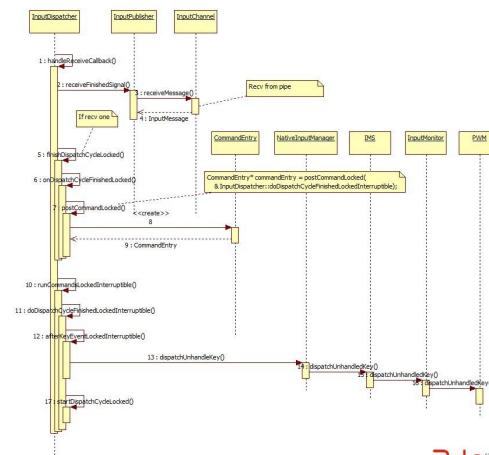
```



<http://blog.csdn.net/2cto.com>

可以看到，从Input_driver中读出的是input_event结构，之后在经过中间模块时会按需对其进行封装，变成或被封装进RawEvent, KeyEvent, DispatchEntry, InputMessage, InputEvent, QueuedInputEvent等结构，其中有些结构是针对另一些结构的封装，如DispatchEntry中封装了KeyEntry, QueuedInputEvent是对InputEvent的封装等。

回到主线，故事情节没讲完。上面说到App逐帧处理完输入事件，然后通过和IMS中InputDispatcher的通信管道InputChannel发了处理完成通知，InputDispatcher这边收到后如何处理呢？



<http://blog.csdn.net/2cto.com>

InputDispatcher会根据HandlerQueue.dequeue()来处理不同的事件。这里主要用CommandEntry来做一个处理事务执行。doDispatchCycleLocked方法中会遍历所有在mCommandEntriesContainer中的命令并进行执行。在doDispatchCycleFinishedLocked方法中会遍历所有在mDispatchEntriesContainer中的命令并进行执行。如果一个Key事件App没有处理，可以在onKeyEvent方法中进行处理，或者在InputDispatcher.handleKey()函数中进行处理。接着InputDispatcher会将该收到完成功信的事件从队列中移除。同时由于上一个事件已被App处理完，就可以调用startDispatchCycleLocked()来进行下一轮事件的处理了。

```

3558 // Dequeue the event and start the next cycle.
3559 // Note that because the lock might have been released, it is possible that the
3560 // contents of the wait queue to have been drained, so we need to double-check
3561 // a few things.
3562 if (dispatchCount == connection->findWaitQueueEntry(seg)) {
3563     connection->waitQueue.dequeue(dispatchEntry);

```

startDispatchCycleLocked函数会检查相应连接的输出缓冲区(connection->outboundQueue)是否有事件要发送的，有的话会通过InputChannel发送出去。



上一篇：android网络 HttpURLConnection抓取网络图片

下一篇：iOS-Swift开发中的单例设计模式

IOS常用加密算法	ios开发——公司测试内部环境搭建	网站升级中...为您提供更多免费学习资源...
IOS添加Cordova到已存在的XCode工程里	ios-prepareForSegue场景切换KVCF值详解	
IOS上动态绘制曲线	ios 中Category类别（扩展类）专题总结	
IOS 证书、描述文件、AppID、	记录遇到的ios下的bug5	
IOS-Swift开发中的单例设计模式	iOS开发-多线程NSOperation和NSOperationQueue	
android网络 HttpURLConnection抓取网络图片	Android 5.0(Lollipop)事件输入系统(Input System)	
Android Framework分析——Android默认Home	android通信HttpClient	
Android开源框架Universal-Image-Loader详解	Android 从硬件到应用：一步一步向上爬 3 - 硬件I	
Android 检测网络连接状态	Android:自定义DialogFragment的内容和按钮	
客户端Android和Webservice之间的图片文件传输	Android触摸屏中的手势识别	

