TECHNICAL NOTE

TCL ALCATEL mobile phones

SITE SHANGHAI	Software specifications
ORIGINATOR	**SOFTWARE TECHNICAL NOTE**

TP Lose Efficacy Summary

Domain: **Architecture**
Rubric: Technical document

CONTENTS:
This doc introduce the Translation & TP Lose Efficacy Summary tech detail

KEY WORDS : Translation, MGSM, Resource Separation

DISTRIBUTION LIST	
SWD-3/app	
SWD-3/perso	
SWD-3/framework	Zhou xinzhu
SWD3 Management	Gao fei, Yu Miao*
SQE	
SPM	
* = mandatory reader	

	AUTHOR	APPROVALS		QUALITY
		LEVEL 1	LEVEL 2	
NAME	Zhou Xinzhu	Xu Honeyue	Yu Miao	
FUNCTION	Application Engineer	Team Leader	Section Manager	
DATE	26/04/15			
SIGNATURE				

DOCUMENT HISTORY

Version	Date	Author	Type of Modification
0.1	04/27/15	Zhou xinzhu	Creation
0.2	11/18/14	Zhou xinzhu	Update P1,P4,P6,P9
0.3	11/20/14	Zhou xinzhu	Update p3, p5, p6, p9, p10

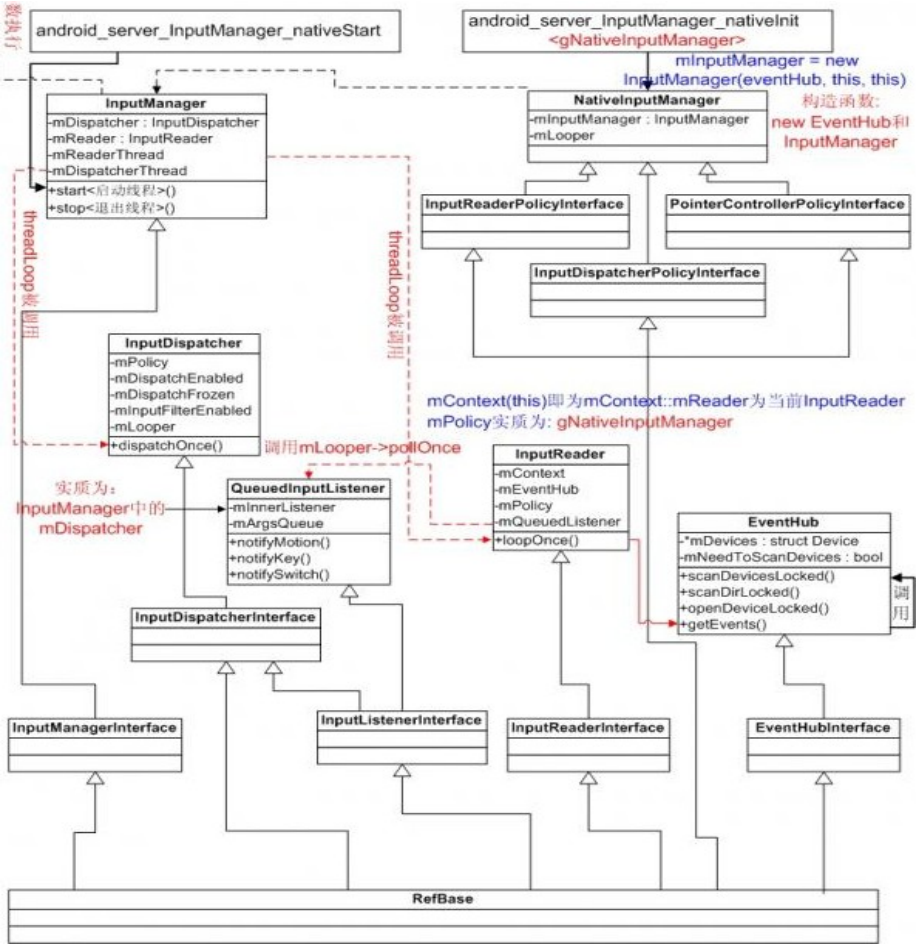
Table of Contents

1 Introduction

1.1 Introduction

本文档是针对 TP 失效问题作出的总结，主要包括 TP 相关事件输入的流程介绍以及 TP 问题 DEBUG 的技巧，主要目的是为了整理这类问题的解决方案，提高解决此类问题的效率！

2 TP 输入消息模块



用户输入系统涉及的主要模块:

1. EventHub：负责监听输入设备；

2. InputReader：负责从驱动获取用户输入事件信息,并交由 InputDispatcher 处理;

3. InputDispatcher：将 InputReader 传送过来的 TP 信息封装成 Events 并且负责分发;

4.nativeInputManager：创建 InputReader，InputDispatcher，EventHub，是 IMS 和底层交互的中介；

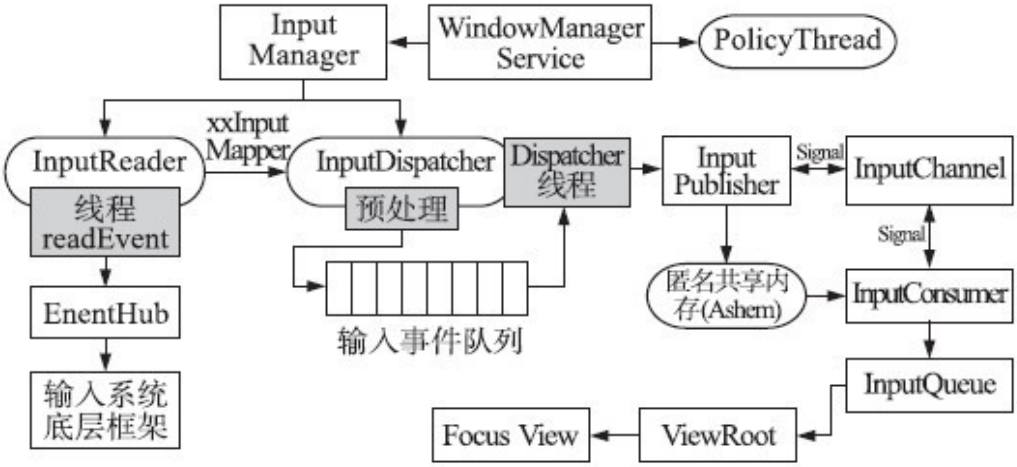
5. InputManagerService：负责通过 JNI 创建 nativeInputManager 文件，并且在创建完毕后负责和 nativeInputManager 文件交互;

6. WindowManagerService：负责创建销毁 Window 到底层的通信通道 InputChannel 以及控制 Window 在一些状态下是否接收 TP 事件;

3 analyze problems of input

3.1 input system flow

了解用户输入事件分发流程是解决这类问题的基础，我们需要保证这个输入系统分发通道是畅通的，在任何一个环节出现问题都有可能造成 TP 失效问题！



下面就来重点讲解这几个模块在通信过程中的作用:

1. EventHub 负责监听输入设备，并且将这些信息收集交由 InputReader 并且扫描和加载所有的输入设备，保存每个设备的配置信息；

2. InputReader：负责从驱动获取用户输入事件信息,并交由 InputDispatcher 处理;

3. InputDispatcher：将 InputReader 传送过来的 TP 信息封装成 Events 并且负责分发;

4. InputPublisher：往 InputChannel 写入事件数据；

5. InputConsumer：从 InputChannel 中读出事件数据；

6. InputQueue：存放事件消息队列；

7. ViewRootImpl：接收处理 input 事件并且分发事件给相应的 View；

3.2 check phenomenon

1. TP 失效是在任何界面都失效还是在某个界面失效？

如果是在任何界面失效我们就需要从驱动开始检查事件分发系统，是驱动报点中断还是整个 TP 被 disable，如果在某个界面失效，那么需要把重点放在 InputDispatcher 分发事件时是否因为窗口状态导致事件被丢弃或者被截获，InputChannel 是否创建成功或者发生异常，还包括检查这个界面上的 View 是否响应事件。

2. TP 是否概率性失效，失效后能否恢复，如果能够恢复那么恢复时间是否在一个合理的范围？

如果可以恢复，我们需要检查 TP 失效这段时间，是系统在哪个环节丢弃了事件还是事件处理被阻塞，主要需要检查 InputDispatcher 处理事件时是否因为窗口状态被丢弃或者被截获，ViewRootImpl 在 sync 事件是否有延迟阻塞！如果不能够恢复，需要检查 TP 状态并且回到 1 步骤检查。

3.3 most critical link

我们需要基于触摸屏事件的通信过程来重点排除几个环节：

1．驱动是否成功报点给 EventHub；

2．InputDispatcher 事件分发过程是否被拦截，被丢弃或者进入输入事件队列失败；

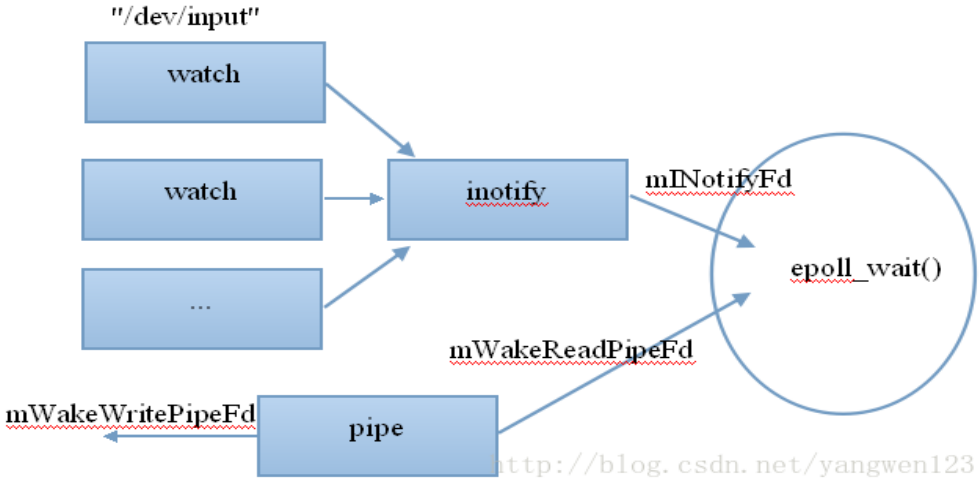
3．Window 注册到底层的通信通道 InputChannel 是否成功或者 InputChannel 通信是否正常；

4．ViewRootImpl 向下分发事件能否成功；

4 EventHub 监听驱动设备

4.1 Introduction

EventHub 可以看成是输入消息的集散地，它最重要的职责就是监听设备节点，收集输入事件，并传递给 InputReader，它会监听 input/devices 下所有的设备节点的变化，因此对上层来说，就不需要关注底层设备的多样性。EventHub 同时还负责扫描和加载所有的输入设备并且保存配置信息。



4.2 监听设备节点模式

4.2.1 inotify 机制

它是一个内核用于通知用户空间程序文件系统变化的机制，以便用户态能够及时地得知内核或底层硬件设备发生了什么，从而能够更好地管理设备，给用户提供更好的服务。

1.创建 inotify 实例，返回一个文件描述符：

```
int fd = inotify_init ();
```

2.文件系统的变化事件被称做 watches 的一个对象管理，每一个 watch 是一个二元组（目标，事件掩码），目标可以是文件或目录。Watch 对象通过 watch 描述符引用，watches 通过文件或目录的路径名来添加。watches 将返回在该目录下的所有文件上面发生的事件。下面函数用于添加一个 watch ：

```
int wd = inotify_add_watch (fd, path, mask);
```

4.2.2 epoll 机制

在 linux 新内核中，有一种事件触发机制，就是 epoll。epoll 最大的好处在于它不会随着监听 fd 数目的增长而降低效率，它的主要函数 epoll_create、epoll_ctl、epoll_wait、close。

创建一个 epoll 的句柄，该函数生成一个 epoll 专用的文件描述符。它其实是在内核申请一空间，用来存放你想关注的 socket fd 上是否发生以及发生了什么事件。在使用完 epoll 后，必须调用 close()关闭，否则可能导致 fd 被耗尽。

2.将被监听的描述符添加到 epoll 句柄或从 epoll 句柄中删除或者对监听事件进行修改。

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
```

参数说明：

epfd：由 epoll_create 生成的 epoll 专用的文件描述符；

op：要进行的操作,包括 EPOLL_CTL_ADD 注册，EPOLL_CTL_MOD 修 改，EPOLL_CTL_DEL 删除;

fd：需要监听的文件描述符；

event：关注 fd 上事件类型:

(a) EPOLLIN：对应的文件描述符上有可读数据。

(b) EPOLLOUT：对应的文件描述符上可以写数据；

(c) EPOLLPRI：对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

(d) EPOLLERR：对应的文件描述符发生错误；

(e) EPOLLHUP：对应的文件描述符被挂断；

(f) EPOLLET：将 EPOLL 设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个 socket 加入到 EPOLL 队列里。如果调用成功返回 0,不成功返回-1

3.等待事件触发，当超过 timeout 还没有事件触发时，就超时。

```
int epoll_wait(int epfd, struct epoll_event * events, intmaxevents,
               int timeout);
```

等待注册在 epfd 上的 socket fd 的事件的发生，如果发生则将发生的 sokct fd 和事件类型放入到 events 数组中，返回发生事件数。用于轮询 I/O 事件的发生；

参数：

epfd:由 epoll_create 生成的 epoll 专用的文件描述符；

epoll_event:用于回传代处理事件的数组；

maxevents:每次能处理的事件数；

timeout:等待 I/O 事件发生的超时值；-1 相当于阻塞，0 相当于非阻塞。

epoll_wait 运行的原理是将注册在 epfd 上的 socket fd 的事件类型给清空，所以如果下一个循环你还要关注这个 socket fd 的话，则需要用 epoll_ctl(epfd,EPOLL_CTL_MOD,listenfd,&ev)来重新设置 socket fd 的事件类型。

4.3 查看监听到的 TP驱动报点

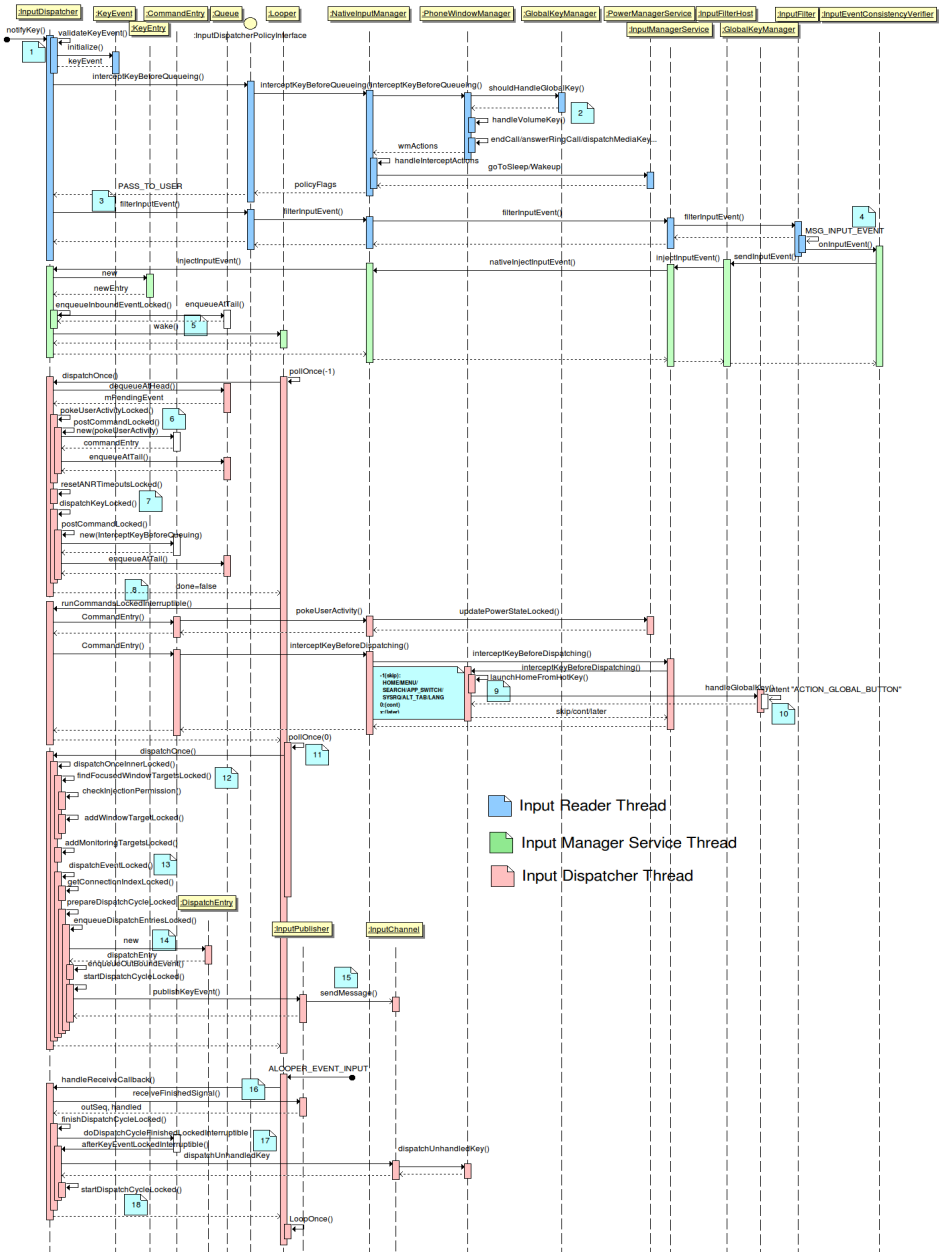
命令：adb shell getevent -tl TP 节点设备，实际上 getevent 命令也是通过 inotify 和 **epoll** 机制监听我们在命令行指定的驱动设备，具体文件可以查看：system/core/toolbox/getevent.c；

```
sdduser@aclgcl-ubnt:/data/project/l8909-20150213$ adb shell getevent -tl /dev/input/event7
[     689.481160] EV_KEY       BTN_TOUCH             DOWN
[     689.481160] EV_ABS       ABS_MT_TRACKING_ID    00000000
[     689.481160] EV_ABS       ABS_MT_TOUCH_MAJOR    00000001
[     689.481160] EV_ABS       ABS_MT_WIDTH_MAJOR    00000001
[     689.481160] EV_ABS       ABS_MT_POSITION_X     0000014f
[     689.481160] EV_ABS       ABS_MT_POSITION_Y     0000026b
[     689.481160] EV_SYN       SYN_MT_REPORT         00000000
[     689.481160] EV_SYN       SYN_REPORT            00000000
[     689.495910] EV_ABS       ABS_MT_TRACKING_ID    00000000
[     689.495910] EV_ABS       ABS_MT_TOUCH_MAJOR    00000001
[     689.495910] EV_ABS       ABS_MT_WIDTH_MAJOR    00000001
[     689.495910] EV_ABS       ABS_MT_POSITION_X     0000014f
[     689.495910] EV_ABS       ABS_MT_POSITION_Y     0000026b
[     689.495910] EV_SYN       SYN_MT_REPORT         00000000
[     689.495910] EV_SYN       SYN_REPORT            00000000
[     689.511238] EV_ABS       ABS_MT_TRACKING_ID    00000000
[     689.511238] EV_ABS       ABS_MT_TOUCH_MAJOR    00000001
[     689.511238] EV_ABS       ABS_MT_WIDTH_MAJOR    00000001
[     689.511238] EV_ABS       ABS_MT_POSITION_X     0000014f
[     689.511238] EV_ABS       ABS_MT_POSITION_Y     0000026b
[     689.511238] EV_SYN       SYN_MT_REPORT         00000000
[     689.511238] EV_SYN       SYN_REPORT            00000000
[     689.526078] EV_ABS       ABS_MT_TRACKING_ID    00000000
[     689.526078] EV_ABS       ABS_MT_TOUCH_MAJOR    00000001
[     689.526078] EV_ABS       ABS_MT_WIDTH_MAJOR    00000001
[     689.526078] EV_ABS       ABS_MT_POSITION_X     0000014f
[     689.526078] EV_ABS       ABS_MT_POSITION_Y     0000026b
```

5 InputDispatcher 分发事件

InputDisptacher 的主要任务是把前面收到的输入事件发送到 PWM 及 App 端的焦点窗口。

5.1 InputDispatcher work flow



这份流程图详细的说明了 InputDispatcher 的分发过程，我们重点看一下它的过程：

（1）1～5 步骤主要是筛选出满足条件的事件放入队列 enqueueInboundEventLocked。

（2）6～15 步骤主要是分发过程。

（3）16～18 的过程主要是发现事件已经成功写入 server 端的 inputchannel，再次唤醒分发过程。

前面提到 InputReaderThread 中收到事件后会调用 notifyMotion()来通知 InputDispatcher 将事件放在 mInboundQueue 中，在 InputDispatcher 的 dispatchOnce 函数中，从这个队列拿出进行分发，实际上 InputDispatcher 分发事件是一个异步过程，InputDispatcher 每次只取队列最前面的事件，写入 InputChannel 中，InputChannel 写入成功后，再次取队列最前面的事件，如此反复。

5.2 丢弃事件

并不是所有的 InputReader 发送来的事件我们都需要传递给应用，也会有部分的事件被丢弃，InputDispatcher 总会根据一些规则来丢弃掉一部分事件，我们来分析以下哪些情况下我们需要丢弃掉部分事件？基本上可以分为两个过程，第一需要成功进入队列，第二需要被成功分发给应用程序；

5.2.1 enqueueInboundEventLocked

InputReader 到 InputDispatcher 的入口是 notifyMotion 函数,只有在 notifyMotion 中没有被丢弃的事件才会进入 mInboundQueue,在 notifyMotion 函数的中，只有通过下面两个条件才会调用方法 enqueueInboundEventLocked 将事件放入队列：

1.validateMotionEvent 判断 TP 事件是否溢出，也就是多点触控如果支持 n 个点，那么在第 n 点以后的 TP 事件将不会进入队列！

2.filterInputEvent 判断 TP 事件是否被过滤器过滤，如果在过滤器名单中，将不会进入队列！

5.2.2 成功进入队列等待分发

在分发函数 dispatchOnceInnerLocked 中，下面两个条件是最重要的排查点：

1.它首先会判断 mDispatchFrozen 这个标志位的值，如果这个标志位为 true,就不会分发并且直接返回；这中情况主要是 WMS 在应用程序窗口切换动画，转屏等情况时，会先冻屏，也就是这个时候不会接收触摸屏事件，等待窗口动画完成，转屏操作完成时，才会解冻屏；可以查看相应的 log：

```
D InputDispatcher: dispatchOnce
D InputDispatcher: Dispatch frozen.  Waiting some more.
```

2.一些枚举类型的原因，InputDispatcher.h 中定义了一个包含有丢弃原因的枚举：

```
enum DropReason {
    DROP_REASON_NOT_DROPPED = 0,
    DROP_REASON_POLICY = 1,
    DROP_REASON_APP_SWITCH = 2,
    DROP_REASON_DISABLED = 3,
    DROP_REASON_BLOCKED = 4,
    DROP_REASON_STALE = 5,
};
```

a . DROP_REASON_NOT_DROPPED

不需要丢弃

b . DROP_REASON_POLICY

在 InputDispatcher 将事件放入队列前，会调用 NativeInputManager 的 interceptMotionBeforeQueueing 判断是否将事件传递给当前应用程序，如果发现 mInteractive 这个标志位置为 false,触摸屏事件将被 PhoneWindowManager 截获，这种情况主要针对在灭屏情况下 TP 事件的特殊处理。

这种情况打开 base/services/core/jni/com_android_server_input_InputManagerService.cpp 的 DEBUG_INPUT_DISPATCHER_POLICY 开关时，在 log 中会输出：

```
D InputManager-JNI handleInterceptActions: Not passing key to user.
```

c . DROP_REASON_APP_SWITCH

当有 App switch 按键如 HOME/ENDCALL/SWITCH 按键发生时，会设置一个 0.5S 的超时时间，当 0.5s 超时时，InputDispatcher 尚未 dispatch 到这个按键时，InputDispatcher 将会丢弃掉 mInboundQueue 中所有处在 app switch 按键前的所有事件，包括按键，触摸屏等事件。这么做的目的是保证 app switch 按键能够确保被处理。

d . DROP_REASON_DISABLED

这个标志表示当前的 InputDispatcher 被 disable 掉了，不能 dispatch 任何事件，比如当系统休眠时或者正在关机时会用到。

e . DROP_REASON_BLOCKED

在分发触摸屏事件之前，需要找到焦点窗口，当无法找到焦点窗口时，触摸屏事件将会被丢弃。

f . DROP_REASON_STALE

当分发事件的时间点距离事件发生的时间点超过 10S 中，将会被丢弃。

5.3 analyze log

（1）需要打开 InputDispatcher，com_android_server_input_InputManagerService.cpp 的 DEBUG 开关。

（2）在上面的这种情况，相应的 log 可以查看下面的关键字：

```
W/InputDispatcher( 850): Dropped event because
```

（3）如果需要查看具体原因，可以参考 dropInboundEventLocked 函数，像下面这种 log 输出就是 DROP_REASON_BLOCKED，原因是由于 application 启动了，window 还没有启动完毕：

```
I/InputDispatcher( 850): Dropped event because the current application is not responding and the user has started interacting with a different application.

D/InputDispatcher: Waiting for application to become ready for input: AppWindowToken{39054555 token=Token{329e080c ActivityRecord{23e5675e u0 com.google.android.gms/.auth.gsf.AccountIntroActivity t689}}}. Reason: Waiting because no window has focus but there is a focused application that may eventually add a window when it finishes starting up
```

6 InputChannel 通信

如果事件在 InputDispatcher 中没有被丢弃，那么就会被写入 InputChannel，InputChannel 是上层和底层的通信管道，一旦 InputChannel 关闭，那么将会出现数据丢失，因此排查 InputChannel 是非常重要的。

6.1 Looper 对文件描述符的监控与处理

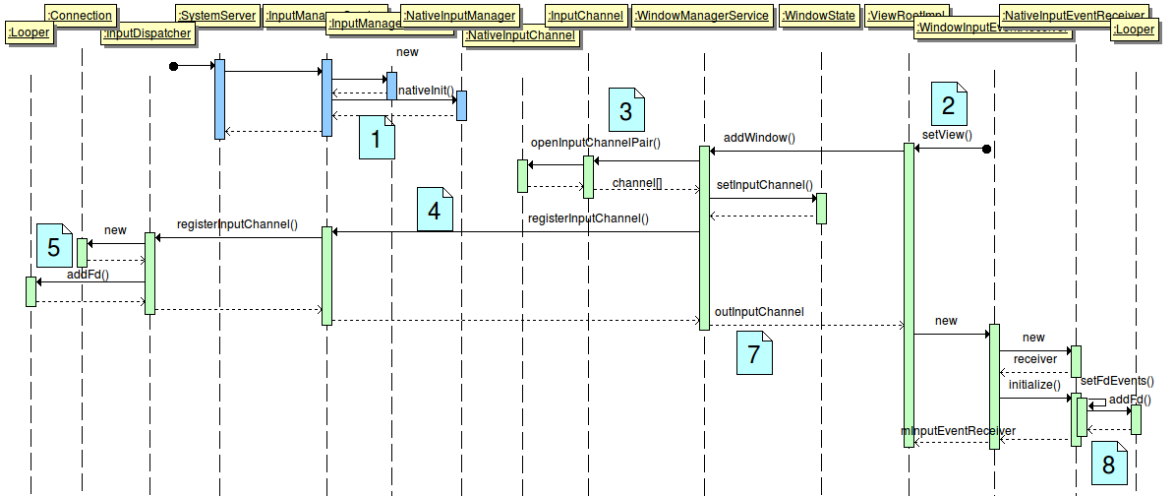
InputChannel 的工作原理就是 Looper 对文件描述符的监控与处理，Looper 内部实际上也是引用了 **epoll** 机制。

1. Looper 提供了 addFd 函数用于添加需要监控的文件描述符，调用者需要指定文件描述符，监控的事件类型，用于处理可 I/O 事件时的回调处理函数。 可在 LooperCallback 的子类中重载 handleEvent 来实现对可 I/O 事件的处理，也可以自定义回调函数。

```
Looper->addFd(fd, 0, ALOOPER_EVENT_INPUT, LooperCallback,this);
```

2. 借助于 Looper 的 pollOnce 和 addFd 函数，可以实现对文件描述符的监控。无数据到来时 pollOnce 的调用者将睡眠等待，有数据到来时其被自动唤醒，并执行指定的回调处理者。

6.2 inputchannel flow



了解 InputChannel 的工作原理是很重要的：

1. InputChannel 对应一个 socket fd，虽然创建了一对 InputChannel，实际上操作的是同一块共享内存；

2. 每一个 Window 对应一个 ViewRootImpl，ViewRootImpl 向 WMS 请求创建窗口时，WMS 通知 InputDispatcher 创建了一对 InputChannel，server 端通往 InputDispatcher，client 端保存在 ViewRootImpl 中；

3.IMS 创建 InputChannel 通过 JNI 最终调用了 InputDispatcher 的 RegisterInputChannel 方法，InputDispater 通过 Loop->addFd，让 Looper 监听 fd 上 ALOOPER_EVENT_INPUT 类型的事件，这样一旦我们往 InputDispatcher 的 InputChannal 的 fd 中写入数据,Looper 就 会马上从睡眠中醒来,进行处理，调用 handleReceiveCallback 回调函数；

```
int fd = inputChannel->getFd();
mLooper->addFd(fd, 0, ALOOPER_EVENT_INPUT, handleReceiveCallback,this);
```

4.InputDispatcher 会从队列中往 InputChannel 写入 Event 数据；

```
dispatchEventLocked ->
prepareDispatchCycleLocked ->
enqueueDispatchEntriesLocked ->
connection->inputPublisher.publishKeyEvent ->
mChannel->sendMessage ->
::send(mFd, msg, msgLength, MSG_DONTWAIT | MSG_NOSIGNAL);
```

（::send 调用的socket发出数据了）

5.handleReceiveCallback 函数主要作用是 Socket 通信，用于另一端接受完缓冲区数据发送 finish 信号后，开始下一轮发送；

```
int InputDispatcher::handleReceiveCallback(int fd, int events, void* data) {
    InputDispatcher* d = static_cast<InputDispatcher*>(data);
    { // acquire lock
     sp<Connection> connection = d->mConnectionsByFd.valueAt(connectionIndex);
         if (!(events & (ALOOPER_EVENT_ERROR | ALOOPER_EVENT_HANGUP))) {
              if (!(events & ALOOPER_EVENT_INPUT)) {
                   return 1;
              }
              nsecs_t currentTime = now();
              bool gotOne = false;
              status_t status;
              for (;;) {
                   uint32_t seq;
                   bool handled;
                   status = connection->inputPublisher.receiveFinishedSignal(&seq,
                            &handled);
                   if (status) {
                        break;
                   }
                    d->finishDispatchCycleLocked(currentTime, connection, seq, handled);
              }
          }
       }
    }
}
```

6.ViewRootImpl 中创建的 NativeInputEventReceiver 通过 Looper->addFd() 用于监听读端的 Socket FD 上 ALOOPER_EVENT_INPUT 类型的事件，接受 InputDispatcher 传过来的 input 事件并执行 handleEvent()回调函数；

7.NativeInputEventReceiver 在执行 hanleEvent 函数中会调用 consumeEvents 函数同步缓冲区中的数据，并提取转化对应的 Event 事件类型，交给 WindowInputEventReceiver 的 dispatchEvent 分发，并且往 InputChannel 中发送一个 finish 信号；

6.3 analyze log

（1）需要打开 InputDispatcher， InputTransport，InputPublisher，InputConsumer 的 DEBUG 开关。

（2）InputChannel 中的几种消息类型：

```
struct InputMessage {
   enum {
      TYPE_KEY = 1,
      TYPE_MOTION = 2,
      TYPE_FINISHED = 3,
   };
}
```

在 log 中我们常常能都看到类似的 log,type 为 2 表示 server 端往 channel 中写入的是 motion event,client 端读出数据后会往 channel 中写入 type 为 3 的事件，表明已经完成一次读操作；

```
D/InputTransport(  888): channel 'WindowManager (server)' ~ sent message of type 2
D/InputTransport(  888): channel 'WindowManager (client)' ~ sent message of type 3
D/InputTransport(  888): channel 'WindowManager (server)' publisher ~ receiveFinishedSignal
D/InputTransport(  888): channel 'WindowManager (server)' ~ received message of type 3
```

（3）log 分析技巧：

a. 提取某一条 InputChannel 通信的 log 信息，我们知道在创建 InputChannel 时，会创建一对 InputChannel,InputChannel 的名字是以窗口 title 为标志；

```
V/WindowManager(  888): Looking for focus: 8 = Window{2c35962c u0
fr.m6.m6replay/com.tapptic.m6.activity.VideoPlayerActivity}, flags=25231616, canReceive=true

D/InputTransport( 7653): channel '2c35962c fr.m6.m6replay/com.tapptic.m6.activity.VideoPlayerActivity
(client)' consumer ~ consume: consumeBatches=true, frameTime=1472002411471
```

b. 关注某一次通信过程，每一次往 InputChannel 中写数据和读数据时，都会有一个序列号;

```
D/InputDispatcher(  888): channel '19cf0509 StatusBar (server)' ~ startDispatchCycle
D/InputTransport(  888): channel '19cf0509 StatusBar (server)' publisher ~ publishMotionEvent: seq=9396,
deviceId=9, source=0x1002, action=0x2, flags=0x0, edgeFlags=0x0, metaState=0x0, buttonState=0x0,
xOffset=0.000000, yOffset=0.000000, xPrecision=1.000926, yPrecision=1.000521,
downTime=1428366070000, eventTime=1428409137000, pointerCount=1
```

c. 关注某一次通信过程读写是否都成功，会打出 handled=true 的 log，表示读端已经处理了事件；

```
D/InputTransport( 1218): channel '19cf0509 StatusBar (client)' consumer ~ sendFinishedSignal: seq=9396,
handled=true

D/InputDispatcher(  888): channel '19cf0509 StatusBar (server)' ~ finishDispatchCycle - seq=9396,
handled=true
```

（4）下面是一份正常的 log:

```
D/InputDispatcher(  888): channel '19cf0509 StatusBar (server)' ~ startDispatchCycle
D/InputTransport(  888): channel '19cf0509 StatusBar (server)' publisher ~ publishMotionEvent: seq=9396,
deviceId=9, source=0x1002, action=0x2, flags=0x0, edgeFlags=0x0, metaState=0x0, buttonState=0x0,
xOffset=0.000000, yOffset=0.000000, xPrecision=1.000926, yPrecision=1.000521,
downTime=1428366070000, eventTime=1428409137000, pointerCount=1
D/InputTransport( 1218): channel '19cf0509 StatusBar (client)' consumer ~ consumed batch event, seq=9396
D/InputTransport( 1218): channel '19cf0509 StatusBar (client)' consumer ~ sendFinishedSignal: seq=9396,
handled=true
D/InputDispatcher(  888): channel '19cf0509 StatusBar (server)' ~ finishDispatchCycle - seq=9396,
handled=true
```

（5）下面是一份异常的 log，表明序列号为 9548 的事件没有被 client 端处理;

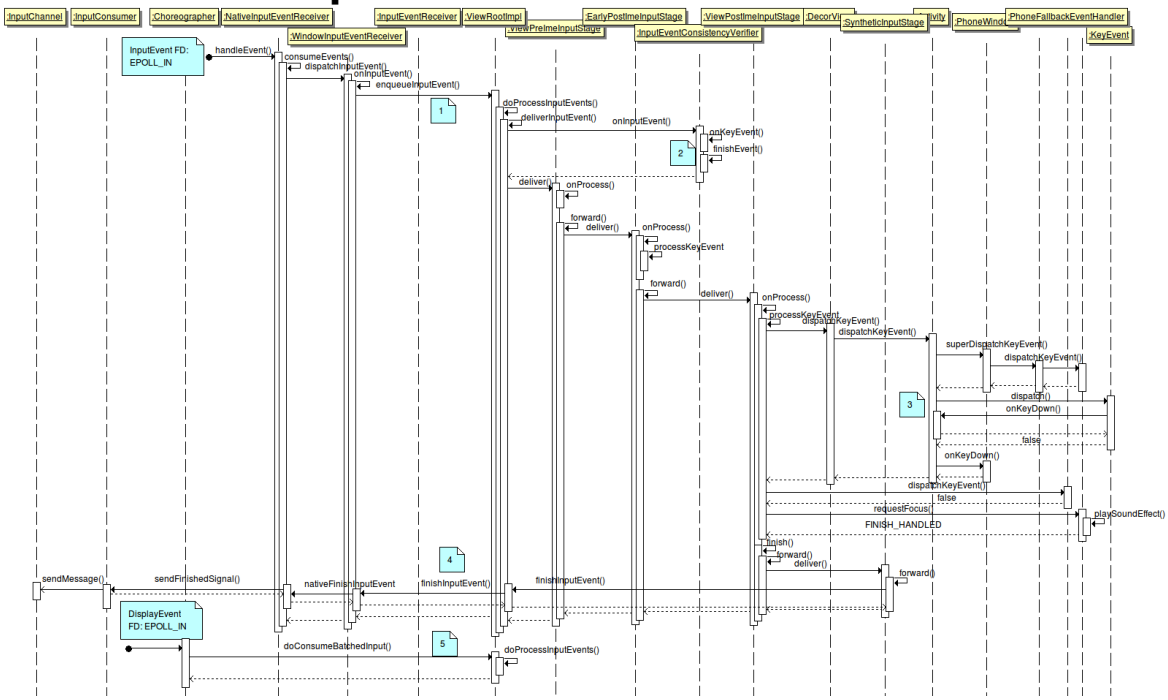
```
D/InputDispatcher(  888): channel '2c3375ca NavigationBar (server)' ~ startDispatchCycle
04-02 17:47:53.689 D/InputTransport(  888): channel '2c3375ca NavigationBar (server)' publisher ~
publishMotionEvent: seq=9548, deviceId=9, source=0x1002, action=0x4, flags=0x0, edgeFlags=0x0,
metaState=0x0, buttonState=0x0, xOffset=0.000000, yOffset=0.000000, xPrecision=1.000926,
yPrecision=1.000521, downTime=1483830174000, eventTime=1483830174000, pointerCount=1
D/InputTransport( 1218): channel '2c3375ca NavigationBar (client)' consumer ~ consumed motion event,
seq=9548
D/InputTransport( 1218): channel '2c3375ca NavigationBar (client)' consumer ~ sendFinishedSignal:
seq=9548, handled=false
D/InputDispatcher(  888): channel '2c3375ca NavigationBar (server)' ~ finishDispatchCycle - seq=9548,
handled=false
```

（6）下面的这份 log 就是由于 InputChannel 发生异常导致 TP 失效，是 InputDispatcher 在分发事件时，发现接收事件 server 端的 inputchannel 发生了错误，每个 inputchannel 都有一个文件描述符，从 events=0X9 这个值可以判断 ALOOPER_EVENT_INPUT | ALOOPER_EVENT_HANGUP，访问文件描述符出错，出现了 hangup 中断，表示远端的 Socket 和 pipe 已经关闭！

```
W/InputDispatcher(  771): channel '7744147
com..graphics.tools/com..graphics.tools.MainActivity (server)' ~ Consumer closed input channel or an error
occurred.  events=0x9
E/InputDispatcher(  771): channel '7744147
com..graphics.tools/com..graphics.tools.MainActivity (server)' ~ Channel is unrecoverably broken and will
be disposed!
W/InputDispatcher(  771): Attempted to unregister already unregistered input channel '7744147
com..graphics.tools/com..graphics.tools.MainActivity (server)'
```

7 ViewRootImpl 分发事件

7.1 viewRootImpl flow



1. NativeInputEventReceiver 监听 client 端的 Inputchannel,在发现有内容写入时，会调用 handleEvent 回调函数;

2. NativeInputEventReceiver 在执行 hanleEvent 函数中会调用 consumeEvents 函数同步缓冲区中的数据，并提取转化对应的 Event 事件类型，交给 WindowInputEventReceiver 的 dispatchEvent 分发;

3. NativeInputEventReceiver 会回调 WindowInputEventReceiver 的 dispatchEvent() 继续调用 ViewRootImpl 的以下方法:

（a) enqueueInputEvent

(b) doProcessInputEvents

(c) deliverInputEvent

(1)首先考虑输入法窗口, ime 分发完了后, 会有回调的 handleImeFinishedEvent;

(2) 没有输入法窗口, 继续分发给 View;

4. 调用 stage 的 onProcess 方法；

```
@Override
protected int onProcess(QueuedInputEvent q) {
    if (q.mEvent instanceof KeyEvent) {
        return processKeyEvent(q);
    } else {
        handleDispatchDoneAnimating();
        final int source = q.mEvent.getSource();
        if ((source & InputDevice.SOURCE_CLASS_POINTER) != 0) {
            return processPointerEvent(q);
        } else if ((source & InputDevice.SOURCE_CLASS_TRACKBALL) != 0) {
            return processTrackballEvent(q);
        } else {
            return processGenericMotionEvent(q);
        }
    }
}
```

```
private int processPointerEvent(QueuedInputEvent q) {
    final MotionEvent event = (MotionEvent)q.mEvent;
    boolean handled = mView.dispatchPointerEvent(event);
    return handled ? FINISH_HANDLED : FORWARD;
}
```

7.2 analyze log

下面这两份 log 都是由于 deliverInputEvent 过程中被丢弃！

```
W/ViewRootImpl( 4555): Dropping event due to root view being removed: MotionEvent
{ action=ACTION_MOVE, id[0]=0, x[0]=-231.13454, y[0]=-938.0859, toolType[0]=TOOL_TYPE_FINGER,
buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0,pointerCount=1,
historySize=0,eventTime=551629, downTime=551617, deviceId=13, source=0x1002 }
```

```
W/ViewRootImpl: Dropping event due to no window focus : MotionEvent { action=ACTION_MOVE, id[0]=0, x[0]=-831.13460, y[0]=-238.0815, toolType[0]=TOOL_TYPE_FINGER, buttonState=0, metaState=0, flags=0x0, edgeFlags=0x0,pointerCount=1,historySize=0,eventTime=758790, downTime=758765, deviceId=13, source=0x1002 }
```

8 联系方式：

Framework Team: xinzhu.zhou@jrdcom.com