# ToDO

<http://gityuan.com/2017/01/01/input-anr/>

看门狗原理

两步集成TV移动框架，从未如此简单：https://www.jianshu.com/p/66c3bbf65511

TV版焦点交互框架

https://www.jianshu.com/p/2de8b3841b29

Connected to the target VM, address: 'localhost:8676', transport: 'socket'

吸附调试原理

如何调试和查看控件树呢

vendor-id 和 product-id

输入源呢：SOURCE\_TOUCHSCREEN

[点见解: 焦点那点事(一)](https://www.jianshu.com/p/641f96c19203)

[如何自学Android？](https://zhuanlan.zhihu.com/p/20708611)

还是在Systemui里面实现吧？？？

<http://aijiawang-126-com.iteye.com/blog/2393863>

按键从Linux到Android

看看人家的组合键怎么实现的

首页 / Android/ 按键截屏功能源码流程讲解

按键截屏功能源码流程讲解

<http://www.codexiu.cn/android/blog/39070/>

Toast太慢了，看DisplayContent.canAddRoastWindForUId

inputDisptch 的trance机制是啥？

Dump呢

Test文件利用起来！

通过输入源来区分摇杆，播轮，5维按键等motion事件

# 目标

所以,如果apk中没有acitivity但是想监听Event事件怎么办呢?在PhoneWindowManager的interceptKeyBeforeQueueing或者interceptKeyBeforeDispatching方法中稍加处理就可以了。

但是

**mSession**.add(**mWindow**, 0, **mLp**, View.***VISIBLE***, **mInsets**, **mInsets**,**mInputChannel**);

InputChannel is not initialized.

static jlong nativeInit(JNIEnv\* env, jclass clazz, jobject receiverWeak,

jobject inputChannelObj, jobject messageQueueObj) {

sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,

inputChannelObj);

if (inputChannel == NULL) {

jniThrowRuntimeException(env, "InputChannel is not initialized.");

return 0;

}

## LOGcat思路

path open /dev/input/event2

InputManager: Initializing input manager, mUseDevInputEventForAudioJack=false

# 概述

## 相关类

frameworks/native/services/inputflinger/

- InputDispatcher.cpp

- InputReader.cpp

- InputManager.cpp

- EventHub.cpp

- InputListener.cpp

frameworks/native/libs/input/

- InputTransport.cpp

- Input.cpp

- InputDevice.cpp

- Keyboard.cpp

- KeyCharacterMap.cpp

- IInputFlinger.cpp

frameworks/base/services/core/

- java/com/android/server/input/InputManagerService.java

- jni/com\_android\_server\_input\_InputManagerService.cpp

### EventHub

它是系统中所有事件的中央处理站。它管理所有系统中可以识别的输入设备的输入事件，此外，当设备增加或删除时，EventHub将产生相应的输入事件给系统。EventHub通过getEvents函数，给系统提供一个输入事件流。它也支持查询输入设备当前的状态（如哪些键当前被按下）。而且EventHub还跟踪每个输入调入的能力，比如输入设备的类别，输入设备支持哪些按键。

### InputReader

　　InputReader从EventHub中读取原始事件数据(RawEvent)，并由各个InputMapper处理之后输入对应的input listener.InputReader拥有一个InputMapper集合。它做的大部分工作在InputReader线程中完成，但是InputReader可以接受任意线程的查询。为了可管理性，InputReader使用一个简单的Mutex来保护它的状态。InputReader拥有一个EventHub对象，但这个对象不是它创建的，而是在创建InputReader时作为参数传入的。

## 基本介绍

当用户触摸屏幕或者按键操作，首次触发的是硬件驱动，驱动收到事件后，将该相应事件写入到输入设备节点， 这便产生了最原生态的内核事件。接着，输入系统取出原生态的事件，经过层层封装后成为KeyEvent或者MotionEvent ；最后，交付给相应的目标窗口(Window)来消费该输入事件。可见，输入系统在整个过程起到承上启下的衔接作用。

Input模块的主要组成：

Native层的InputReader负责从EventHub取出事件并处理，再交给InputDispatcher；

Native层的InputDispatcher接收来自InputReader的输入事件，并记录WMS的窗口信息，用于派发事件到合适的窗口；

Java层的InputManagerService跟WMS交互，WMS记录所有窗口信息，并同步更新到IMS，为InputDispatcher正确派发事件到ViewRootImpl提供保障；

Input相关的动态库：

libinputflinger.so：frameworks/native/services/inputflinger/

libinputservice.so：frameworks/base/libs/input/

libinput.so： frameworks/native/libs/input/

## 整体框架类图

InputManagerService作为system\_server中的重要服务，继承于IInputManager.Stub， 作为Binder服务端，那么Client位于InputManager的内部通过IInputManager.Stub.asInterface() 获取Binder代理端，C/S两端通信的协议是由IInputManager.aidl来定义的。





图解:

* InputManagerService位于Java层的InputManagerService.java文件；
  + 其成员mPtr指向Native层的NativeInputManager对象；
* NativeInputManager位于Native层的com\_android\_server\_input\_InputManagerService.cpp文件；
  + 其成员mServiceObj指向Java层的IMS对象；
  + 其成员mLooper是指“android.display”线程的Looper;
* InputManager位于libinputflinger中的InputManager.cpp文件；
  + InputDispatcher和InputReader的成员变量mPolicy都是指NativeInputManager对象;
  + InputReader的成员mQueuedListener，数据类型为QueuedInputListener；通过其内部成员变量mInnerListener指向InputDispatcher对象； 这便是InputReader跟InputDispatcher交互的中间枢纽。

## Debug

Busybox lsusb 查看是否连接上了，

rm500:/ $ busybox lsusb

**Bus 002 Device 002: ID 2ca3:1501**

Bus 005 Device 002: ID 2ca3:001f

Bus 001 Device 001: ID 1d6b:0002

Bus 002 Device 001: ID 1d6b:0001

Bus 003 Device 001: ID 1d6b:0002

Bus 004 Device 001: ID 1d6b:0003

Bus 005 Device 001: ID 1d6b:0002

Bus 006 Device 001: ID 1d6b:0001

getevent

add device 1: /dev/input/event1

name: "D RC"

需要报event1的事件

点击，看是否过来了

# 源码分析-启动InputManagerService

SystemServer. Main

SystemServer .startOtherServices

New InputManagerService()

nativeInit

开始init

## 如何init





### nativeInit

返回的是一个NativeInputManager对象，该指针被强转成jlong，返回给mPtr。

[-> com\_android\_server\_input\_InputManagerService.cpp]

static jlong nativeInit(JNIEnv\* env, jclass /\* clazz \*/, jobject serviceObj, jobject contextObj, jobject messageQueueObj) {

**//获取native消息队列**

sp<MessageQueue> messageQueue = android\_os\_MessageQueue\_getMessageQueue(env, messageQueueObj);

...

//创建Native的InputManager【见小节2.3】

NativeInputManager\* im = new NativeInputManager(contextObj, serviceObj,

messageQueue->getLooper());

im->incStrong(0);

return reinterpret\_cast<jlong>(im); //返回Native对象的指针

}

### NativeInputManager

NativeInputManager::NativeInputManager(jobject contextObj,

jobject serviceObj, const sp<Looper>& looper) :

mLooper(looper), mInteractive(true) {

JNIEnv\* env = jniEnv();

mContextObj = env->NewGlobalRef(contextObj); //上层IMS的context

mServiceObj = env->NewGlobalRef(serviceObj); //上层IMS对象

...

sp<EventHub> eventHub = new EventHub(); // 创建EventHub对象【见小节2.4】

mInputManager = new InputManager(eventHub, this, this); // 创建InputManager对象【见小节2.5】

}

此处的mLooper是指“android.display”线程的Looper; libinputservice.so库中PointerController和SpriteController对象都继承于于MessageHandler， 这两个Handler采用的便是该mLooper.

### EventHub

EventHub::EventHub(void) :

mBuiltInKeyboardId(NO\_BUILT\_IN\_KEYBOARD), mNextDeviceId(1), mControllerNumbers(),

mOpeningDevices(0), mClosingDevices(0),

mNeedToSendFinishedDeviceScan(false),

mNeedToReopenDevices(false), mNeedToScanDevices(true),

mPendingEventCount(0), mPendingEventIndex(0), mPendingINotify(false) {

acquire\_wake\_lock(PARTIAL\_WAKE\_LOCK, WAKE\_LOCK\_ID);

//创建epoll

mEpollFd = epoll\_create(EPOLL\_SIZE\_HINT);

mINotifyFd = inotify\_init();

//此处DEVICE\_PATH为"/dev/input"，监听该设备路径

int result = inotify\_add\_watch(mINotifyFd, DEVICE\_PATH, IN\_DELETE | IN\_CREATE);

struct epoll\_event eventItem;

memset(&eventItem, 0, sizeof(eventItem));

eventItem.events = EPOLLIN;

eventItem.data.u32 = EPOLL\_ID\_INOTIFY;

//添加INotify到epoll实例

result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mINotifyFd, &eventItem);

int wakeFds[2];

result = pipe(wakeFds); //创建管道

mWakeReadPipeFd = wakeFds[0];

mWakeWritePipeFd = wakeFds[1];

//将pipe的读和写都设置为非阻塞方式

result = fcntl(mWakeReadPipeFd, F\_SETFL, O\_NONBLOCK);

result = fcntl(mWakeWritePipeFd, F\_SETFL, O\_NONBLOCK);

eventItem.data.u32 = EPOLL\_ID\_WAKE;

//添加管道的读端到epoll实例

result = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, mWakeReadPipeFd, &eventItem);

...

}

该方法主要功能：

初始化INotify（监听”/dev/input”），并添加到epoll实例

创建非阻塞模式的管道，并添加到epoll;

### InputManager

[-> InputManager.cpp]

InputManager::InputManager(

const sp<EventHubInterface>& eventHub,

const sp<InputReaderPolicyInterface>& readerPolicy,

const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {

//创建InputDispatcher对象【见小节2.6】

mDispatcher = new InputDispatcher(dispatcherPolicy);

//创建InputReader对象【见小节2.7】

mReader = new InputReader(eventHub, readerPolicy, mDispatcher);

initialize();//【见小节2.8】

}

InputDispatcher和InputReader的mPolicy成员变量都是指NativeInputManager对象。

### InputDispatcher

-> InputDispatcher.cpp]

InputDispatcher::InputDispatcher(const sp<InputDispatcherPolicyInterface>& policy) :

mPolicy(policy),

mPendingEvent(NULL), mLastDropReason(DROP\_REASON\_NOT\_DROPPED),

mAppSwitchSawKeyDown(false), mAppSwitchDueTime(LONG\_LONG\_MAX),

mNextUnblockedEvent(NULL),

mDispatchEnabled(false), mDispatchFrozen(false), mInputFilterEnabled(false),

mInputTargetWaitCause(INPUT\_TARGET\_WAIT\_CAUSE\_NONE) {

//创建Looper对象

mLooper = new Looper(false);

mKeyRepeatState.lastKeyEntry = NULL;

//获取分发超时参数

policy->getDispatcherConfiguration(&mConfig);

}

该方法主要工作：

创建属于自己线程的Looper对象；

超时参数来自于IMS，参数默认值keyRepeatTimeout = 500，keyRepeatDelay = 50。

### InputReader

[-> InputReader.cpp]

InputReader::InputReader(const sp<EventHubInterface>& eventHub,

const sp<InputReaderPolicyInterface>& policy,

const sp<InputListenerInterface>& listener) :

mContext(this), mEventHub(eventHub), mPolicy(policy),

mGlobalMetaState(0), mGeneration(1),

mDisableVirtualKeysTimeout(LLONG\_MIN), mNextTimeout(LLONG\_MAX),

mConfigurationChangesToRefresh(0) {

// 创建输入监听对象

mQueuedListener = new QueuedInputListener(listener);

{

AutoMutex \_l(mLock);

refreshConfigurationLocked(0);

updateGlobalMetaStateLocked();

}

}

此处mQueuedListener的成员变量mInnerListener便是InputDispatcher对象。 前面【小节2.5】InputManager创建完InputDispatcher和InputReader对象， 接下里便是调用initialize初始化。

### Initialize

[-> InputManager.cpp]

void InputManager::initialize() {

//创建线程“InputReader”

mReaderThread = new InputReaderThread(mReader);

//创建线程”InputDispatcher“

mDispatcherThread = new InputDispatcherThread(mDispatcher);

}

InputReaderThread::InputReaderThread(const sp<InputReaderInterface>& reader) :

Thread(/\*canCallJava\*/ true), mReader(reader) {

}

InputDispatcherThread::InputDispatcherThread(const sp<InputDispatcherInterface>& dispatcher) :

Thread(/\*canCallJava\*/ true), mDispatcher(dispatcher) {

}

初始化的主要工作就是创建两个能访问Java代码的native线程。

创建线程“InputReader”

创建线程”InputDispatcher“

到此[2.1-2.8]整个的InputManagerService对象初始化过程并完成，接下来便是调用其start方法。

## InputManagerService :Start

### start

[-> InputManagerService.java]

public void start() {

// 启动native对象[见小节2.10]

nativeStart(mPtr);

Watchdog.getInstance().addMonitor(this);

//注册触摸点速度和是否显示功能的观察者

registerPointerSpeedSettingObserver();

registerShowTouchesSettingObserver();

mContext.registerReceiver(new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

updatePointerSpeedFromSettings();

updateShowTouchesFromSettings();

}

}, new IntentFilter(Intent.ACTION\_USER\_SWITCHED), null, mHandler);

updatePointerSpeedFromSettings(); //更新触摸点的速度

updateShowTouchesFromSettings(); //是否在屏幕上显示触摸点

}

### nativeStart

[-> com\_android\_server\_input\_InputManagerService.cpp]

static void nativeStart(JNIEnv\* env, jclass /\* clazz \*/, jlong ptr) {

//此处ptr记录的便是NativeInputManager

NativeInputManager\* im = reinterpret\_cast<NativeInputManager\*>(ptr);

// [见小节2.11]

status\_t result = im->getInputManager()->start();

...

}

### InputManager.start

[InputManager.cpp]

status\_t InputManager::start() {

result = mDispatcherThread->run("InputDispatcher", PRIORITY\_URGENT\_DISPLAY);

result = mReaderThread->run("InputReader", PRIORITY\_URGENT\_DISPLAY);

...

return OK;

}

该方法的主要功能是启动两个线程:

启动线程“InputReader”

启动线程”InputDispatcher“

## 小结

### 分层视角：

Java层InputManagerService：采用android.display线程处理Message.

JNI的NativeInputManager：采用android.display线程处理Message,以及创建EventHub，InputManager。

Native的InputManager：创建InputReaderThread和InputDispatcherThread两个线程

### 主要功能：

IMS服务中的成员变量mPtr记录Native层的NativeInputManager对象；

IMS对象的初始化过程的重点在于native初始化，分别创建了以下对象：

NativeInputManager；

EventHub, InputManager；

InputReader，InputDispatcher；

InputReaderThread，InputDispatcherThread

IMS启动过程的主要功能是启动以下两个线程：

InputReader：从EventHub取出事件并处理，再交给InputDispatcher

InputDispatcher：接收来自InputReader的输入事件，并派发事件到合适的窗口。

**从整个启动过程，可知有system\_server进程中有3个线程跟Input输入系统息息相关，分别是android.display, InputReader,InputDispatcher。**

InputDispatcher线程：属于Looper线程，会创建属于自己的Looper，循环分发消息；

InputReader线程：通过getEvents()调用EventHub读取输入事件，循环读取消息；

android.display线程：属于Looper线程，用于处理Java层的IMS.InputManagerHandler和JNI层的NativeInputManager中指定的MessageHandler消息;

Input事件流程：Linux Kernel -> IMS(InputReader -> InputDispatcher) -> WMS -> ViewRootImpl， 后续再进一步介绍。

## 参考：

[Input系统—启动篇](http://gityuan.com/2016/12/10/input-manager/)

# 源码分析-InputReader

<http://gityuan.com/2016/12/11/input-reader/>

从InputReader线程的执行过程从threadLoop为起点开始分析。

## InputReader起点

### threadLoop

bool InputReaderThread::threadLoop() {

mReader->loopOnce(); //【见小节1.2】

return true;

}

threadLoop返回值true代表的是会不断地循环调用loopOnce()。另外，如果当返回值为false则会 退出循环。整个过程是不断循环的地调用InputReader的loopOnce()方法

### loopOnce

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputReader::loopOnce() {
2. ...
3. {
4. AutoMutex \_l(mLock);
5. uint32\_t changes = mConfigurationChangesToRefresh;
6. **if** (changes) {
7. timeoutMillis = 0;
8. ...
9. } **else** **if** (mNextTimeout != LLONG\_MAX) {
10. nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);
11. timeoutMillis = toMillisecondTimeoutDelay(now, mNextTimeout);
12. }
13. }
15. //从EventHub读取事件，其中EVENT\_BUFFER\_SIZE = 256【见小节2.1】
16. **size\_t** count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT\_BUFFER\_SIZE);
18. { // acquire lock
19. AutoMutex \_l(mLock);
20. mReaderIsAliveCondition.broadcast();
21. **if** (count) { //处理事件【见小节3.1】
22. processEventsLocked(mEventBuffer, count);
23. }
24. **if** (oldGeneration != mGeneration) {
25. inputDevicesChanged = **true**;
26. getInputDevicesLocked(inputDevices);
27. }
28. ...
29. } // release lock

32. **if** (inputDevicesChanged) { //输入设备发生改变
33. mPolicy->notifyInputDevicesChanged(inputDevices);
34. }
35. //发送事件到nputDispatcher【见小节4.1】
36. mQueuedListener->flush();
37. }

## EventHub

### getEvents

fasd

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **size\_t** EventHub::getEvents(**int** timeoutMillis, RawEvent\* buffer, **size\_t** bufferSize) {
2. AutoMutex \_l(mLock); //加锁
4. **struct** input\_event readBuffer[bufferSize];
5. RawEvent\* event = buffer; //原始事件
6. **size\_t** capacity = bufferSize; //容量大小为256
7. **bool** awoken = **false**;
8. **for** (;;) {
9. nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);
10. ...
12. **if** (mNeedToScanDevices) {
13. mNeedToScanDevices = **false**;
14. scanDevicesLocked(); //扫描设备【见小节2.2】
15. mNeedToSendFinishedDeviceScan = **true**;
16. }
18. **while** (mOpeningDevices != NULL) {
19. Device\* device = mOpeningDevices;
20. mOpeningDevices = device->next;
21. event->when = now;
22. event->deviceId = device->id == mBuiltInKeyboardId ? 0 : device->id;
23. event->type = DEVICE\_ADDED; //添加设备的事件
24. event += 1;
25. mNeedToSendFinishedDeviceScan = **true**;
26. **if** (--capacity == 0) {
27. **break**;
28. }
29. }
30. ...
32. **bool** deviceChanged = **false**;
33. **while** (mPendingEventIndex < mPendingEventCount) {
34. //从mPendingEventItems读取事件项
35. **const** **struct** epoll\_event& eventItem = mPendingEventItems[mPendingEventIndex++];
36. ...
37. //获取设备ID所对应的device
38. ssize\_t deviceIndex = mDevices.indexOfKey(eventItem.data.u32);
39. Device\* device = mDevices.valueAt(deviceIndex);
40. **if** (eventItem.events & EPOLLIN) {
41. //从设备不断读取事件，放入到readBuffer
42. int32\_t readSize = read(device->fd, readBuffer,
43. **sizeof**(**struct** input\_event) \* capacity);
45. **if** (readSize == 0 || (readSize < 0 && errno == ENODEV)) {
46. deviceChanged = **true**;
47. closeDeviceLocked(device);//设备已被移除则执行关闭操作
48. } **else** **if** (readSize < 0) {
49. ...
50. } **else** **if** ((readSize % **sizeof**(**struct** input\_event)) != 0) {
51. ...
52. } **else** {
53. int32\_t deviceId = device->id == mBuiltInKeyboardId ? 0 : device->id;
54. **size\_t** count = **size\_t**(readSize) / **sizeof**(**struct** input\_event);
56. **for** (**size\_t** i = 0; i < count; i++) {
57. //获取readBuffer的数据
58. **struct** input\_event& iev = readBuffer[i];
59. //将input\_event信息, 封装成RawEvent
60. event->when = nsecs\_t(iev.time.tv\_sec) \* 1000000000LL
61. + nsecs\_t(iev.time.tv\_usec) \* 1000LL;
62. event->deviceId = deviceId;
63. event->type = iev.type;
64. event->code = iev.code;
65. event->value = iev.value;
66. event += 1;
67. capacity -= 1;
68. }
69. **if** (capacity == 0) {
70. mPendingEventIndex -= 1;
71. **break**;
72. }
73. }
74. }
75. ...
76. }
77. ...
78. mLock.unlock(); //poll之前先释放锁
79. //等待input事件的到来
80. **int** pollResult = epoll\_wait(mEpollFd, mPendingEventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);
81. ...
82. mLock.lock(); //poll之后再次请求锁
84. **if** (pollResult < 0) { //出现错误
85. mPendingEventCount = 0;
86. **if** (errno != EINTR) {
87. usleep(100000); //系统发生错误则休眠1s
88. }
89. } **else** {
90. mPendingEventCount = **size\_t**(pollResult);
91. }
92. }
94. **return** event - buffer; //返回所读取的事件个数
95. }

EventHub采用INotify + epoll机制实现监听目录/dev/input下的设备节点，经过EventHub将input\_event结构体 + deviceId 转换成RawEvent结构体，如下：

#### RawEvent

[-> InputEventReader.h]

struct input\_event {

struct timeval time; //事件发生的时间点

\_\_u16 type;

\_\_u16 code;

\_\_s32 value;

};

struct RawEvent {

nsecs\_t when; //事件发生的时间店

int32\_t deviceId; //产生事件的设备Id

int32\_t type; // 事件类型

int32\_t code;

int32\_t value;

};

此处事件类型:

DEVICE\_ADDED(添加)

DEVICE\_REMOVED(删除)

FINISHED\_DEVICE\_SCAN(扫描完成)

type<FIRST\_SYNTHETIC\_EVENT(其他事件)

getEvents()已完成转换事件转换工作, 接下来,顺便看看设备扫描过程

2.2.1 scanDevicesLocked

void EventHub::scanDevicesLocked() {

//此处DEVICE\_PATH="/dev/input"【见小节2.3】

status\_t res = scanDirLocked(DEVICE\_PATH);

...

}

#### scanDirLocked

status\_t EventHub::scanDirLocked(const char \*dirname)

{

char devname[PATH\_MAX];

char \*filename;

DIR \*dir;

struct dirent \*de;

dir = opendir(dirname);

strcpy(devname, dirname);

filename = devname + strlen(devname);

\*filename++ = '/';

//读取/dev/input/目录下所有的设备节点

while((de = readdir(dir))) {

if(de->d\_name[0] == '.' &&

(de->d\_name[1] == '\0' ||

(de->d\_name[1] == '.' && de->d\_name[2] == '\0')))

continue;

strcpy(filename, de->d\_name);

//打开相应的设备节点【2.2.3】

openDeviceLocked(devname);

}

closedir(dir);

return 0;

}

#### openDeviceLocked

status\_t EventHub::openDeviceLocked(const char \*devicePath) {

char buffer[80];

//打开设备文件

int fd = open(devicePath, O\_RDWR | O\_CLOEXEC);

InputDeviceIdentifier identifier;

//获取设备名

if(ioctl(fd, EVIOCGNAME(sizeof(buffer) - 1), &buffer) < 1){

} else {

buffer[sizeof(buffer) - 1] = '\0';

identifier.name.setTo(buffer);

}

identifier.bus = inputId.bustype;

identifier.product = inputId.product;

identifier.vendor = inputId.vendor;

identifier.version = inputId.version;

//获取设备物理地址

if(ioctl(fd, EVIOCGPHYS(sizeof(buffer) - 1), &buffer) < 1) {

} else {

buffer[sizeof(buffer) - 1] = '\0';

identifier.location.setTo(buffer);

}

//获取设备唯一ID

if(ioctl(fd, EVIOCGUNIQ(sizeof(buffer) - 1), &buffer) < 1) {

} else {

buffer[sizeof(buffer) - 1] = '\0';

identifier.uniqueId.setTo(buffer);

}

//将identifier信息填充到fd

assignDescriptorLocked(identifier);

//设置fd为非阻塞方式

fcntl(fd, F\_SETFL, O\_NONBLOCK);

//获取设备ID，分配设备对象内存

int32\_t deviceId = mNextDeviceId++;

Device\* device = new Device(fd, deviceId, String8(devicePath), identifier);

...

//注册epoll

struct epoll\_event eventItem;

memset(&eventItem, 0, sizeof(eventItem));

eventItem.events = EPOLLIN;

if (mUsingEpollWakeup) {

eventItem.events |= EPOLLWAKEUP;

}

eventItem.data.u32 = deviceId;

if (epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, fd, &eventItem)) {

delete device; //添加失败则删除该设备

return -1;

}

...

//【见小节2.2.4】

addDeviceLocked(device);

}

#### addDeviceLocked

void EventHub::addDeviceLocked(Device\* device) {

mDevices.add(device->id, device); //添加到mDevices队列

device->next = mOpeningDevices;

mOpeningDevices = device;

}

介绍了EventHub从设备节点获取事件的流程，当收到事件后接下里便开始处理事件。

## InputReader

### processEventsLocked

sdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputReader::processEventsLocked(**const** RawEvent\* rawEvents, **size\_t** count) {
2. **for** (**const** RawEvent\* rawEvent = rawEvents; count;) {
3. int32\_t type = rawEvent->type;
4. **size\_t** batchSize = 1;
5. **if** (type < EventHubInterface::FIRST\_SYNTHETIC\_EVENT) {
6. int32\_t deviceId = rawEvent->deviceId;
7. **while** (batchSize < count) {
8. **if** (rawEvent[batchSize].type >= EventHubInterface::FIRST\_SYNTHETIC\_EVENT
9. || rawEvent[batchSize].deviceId != deviceId) {
10. **break**;
11. }
12. batchSize += 1; //同一设备的事件打包处理
13. }
14. //数据事件的处理【见小节3.3】
15. processEventsForDeviceLocked(deviceId, rawEvent, batchSize);
16. } **else** {
17. **switch** (rawEvent->type) {
18. **case** EventHubInterface::DEVICE\_ADDED:
19. //设备添加【见小节3.2】
20. addDeviceLocked(rawEvent->when, rawEvent->deviceId);
21. **break**;
22. **case** EventHubInterface::DEVICE\_REMOVED:
23. //设备移除
24. removeDeviceLocked(rawEvent->when, rawEvent->deviceId);
25. **break**;
26. **case** EventHubInterface::FINISHED\_DEVICE\_SCAN:
27. //设备扫描完成
28. handleConfigurationChangedLocked(rawEvent->when);
29. **break**;
30. **default**:
31. ALOG\_ASSERT(**false**);//不会发生
32. **break**;
33. }
34. }
35. count -= batchSize;
36. rawEvent += batchSize;
37. }
38. }

事件处理总共有下几类类型：

DEVICE\_ADDED(设备增加), [见小节3.2]

DEVICE\_REMOVED(设备移除)

FINISHED\_DEVICE\_SCAN(设备扫描完成)

数据事件[见小节3.4]

先来说说DEVICE\_ADDED设备增加的过程。

### 设备增加

#### addDeviceLocked

void InputReader::addDeviceLocked(nsecs\_t when, int32\_t deviceId) {

ssize\_t deviceIndex = mDevices.indexOfKey(deviceId);

if (deviceIndex >= 0) {

return; //已添加的相同设备则不再添加

}

InputDeviceIdentifier identifier = mEventHub->getDeviceIdentifier(deviceId);

uint32\_t classes = mEventHub->getDeviceClasses(deviceId);

int32\_t controllerNumber = mEventHub->getDeviceControllerNumber(deviceId);

//【见小节3.2.2】

InputDevice\* device = createDeviceLocked(deviceId, controllerNumber, identifier, classes);

device->configure(when, &mConfig, 0);

device->reset(when);

mDevices.add(deviceId, device); //添加设备到mDevices

...

}

#### createDeviceLocked

f

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. InputDevice\* InputReader::createDeviceLocked(int32\_t deviceId, int32\_t controllerNumber,
2. **const** InputDeviceIdentifier& identifier, uint32\_t classes) {
3. //创建InputDevice对象
4. InputDevice\* device = **new** InputDevice(&mContext, deviceId, bumpGenerationLocked(),
5. controllerNumber, identifier, classes);
6. ...
8. //获取键盘源类型
9. uint32\_t keyboardSource = 0;
10. int32\_t keyboardType = AINPUT\_KEYBOARD\_TYPE\_NON\_ALPHABETIC;
11. **if** (classes & INPUT\_DEVICE\_CLASS\_KEYBOARD) {
12. keyboardSource |= AINPUT\_SOURCE\_KEYBOARD;
13. }
14. **if** (classes & INPUT\_DEVICE\_CLASS\_ALPHAKEY) {
15. keyboardType = AINPUT\_KEYBOARD\_TYPE\_ALPHABETIC;
16. }
17. **if** (classes & INPUT\_DEVICE\_CLASS\_DPAD) {
18. keyboardSource |= AINPUT\_SOURCE\_DPAD;
19. }
20. **if** (classes & INPUT\_DEVICE\_CLASS\_GAMEPAD) {
21. keyboardSource |= AINPUT\_SOURCE\_GAMEPAD;
22. }
24. //添加键盘类设备InputMapper
25. **if** (keyboardSource != 0) {
26. device->addMapper(**new** KeyboardInputMapper(device, keyboardSource, keyboardType));
27. }
29. //添加鼠标类设备InputMapper
30. **if** (classes & INPUT\_DEVICE\_CLASS\_CURSOR) {
31. device->addMapper(**new** CursorInputMapper(device));
32. }
33. // Joystick-like devices.
34. if (classes & INPUT\_DEVICE\_CLASS\_JOYSTICK) {
35. device->addMapper(new JoystickInputMapper(device));
36. }
37. //添加触摸屏设备InputMapper
38. **if** (classes & INPUT\_DEVICE\_CLASS\_TOUCH\_MT) {
39. device->addMapper(**new** MultiTouchInputMapper(device));
40. } **else** **if** (classes & INPUT\_DEVICE\_CLASS\_TOUCH) {
41. device->addMapper(**new** SingleTouchInputMapper(device));
42. }
43. ...
44. **return** device;
45. }

该方法主要功能：

创建InputDevice对象，将InputReader的mContext赋给InputDevice对象所对应的变量

根据设备类型来创建并添加相对应的InputMapper，同时设置mContext.

input设备类型有很多种，以上代码只列举部分常见的设备以及相应的InputMapper：

键盘类设备：KeyboardInputMapper

触摸屏设备：MultiTouchInputMapper或SingleTouchInputMapper

鼠标类设备：CursorInputMapper

介绍完设备增加过程，继续回到[小节3.1]除了设备的增删，更常见事件便是数据事件，那么接下来介绍数据事件的 处理过程。

### 事件处理

#### processEventsForDeviceLocked

void InputReader::processEventsForDeviceLocked(int32\_t deviceId,

const RawEvent\* rawEvents, size\_t count) {

ssize\_t deviceIndex = mDevices.indexOfKey(deviceId);

...

InputDevice\* device = mDevices.valueAt(deviceIndex);

if (device->isIgnored()) {

return; //可忽略则直接返回

}

//【见小节3.3.2】

device->process(rawEvents, count);

}

#### InputDevice.process

void InputDevice::process(const RawEvent\* rawEvents, size\_t count) {

size\_t numMappers = mMappers.size();

for (const RawEvent\* rawEvent = rawEvents; count--; rawEvent++) {

if (mDropUntilNextSync) {

if (rawEvent->type == EV\_SYN && rawEvent->code == SYN\_REPORT) {

mDropUntilNextSync = false;

}

} else if (rawEvent->type == EV\_SYN && rawEvent->code == SYN\_DROPPED) {

mDropUntilNextSync = true;

reset(rawEvent->when);

} else {

for (size\_t i = 0; i < numMappers; i++) {

InputMapper\* mapper = mMappers[i];

//调用具体mapper来处理【见小节3.4】

mapper->process(rawEvent);

}

}

}

}

小节[3.2]createDeviceLocked创建设备并添加InputMapper，提到会有多种InputMapper。 这里以KeyboardInputMapper(按键事件)为例来展开说明

### 按键事件处理

#### KeyboardInputMapper.process

oid KeyboardInputMapper::process(const RawEvent\* rawEvent) {

switch (rawEvent->type) {

case EV\_KEY: {

int32\_t scanCode = rawEvent->code;

int32\_t usageCode = mCurrentHidUsage;

mCurrentHidUsage = 0;

if (isKeyboardOrGamepadKey(scanCode)) {

int32\_t keyCode;

//获取所对应的KeyCode【见小节3.4.2】

if (getEventHub()->mapKey(getDeviceId(), scanCode, usageCode, &keyCode, &flags)) {

keyCode = AKEYCODE\_UNKNOWN;

flags = 0;

}

//【见小节3.4.4】

processKey(rawEvent->when, rawEvent->value != 0, keyCode, scanCode, flags);

}

break;

}

case EV\_MSC: ...

case EV\_SYN: ...

}

}

#### EventHub::mapKey

[-> EventHub.cpp]

status\_t EventHub::mapKey(int32\_t deviceId,

int32\_t scanCode, int32\_t usageCode, int32\_t metaState,

int32\_t\* outKeycode, int32\_t\* outMetaState, uint32\_t\* outFlags) const {

AutoMutex \_l(mLock);

Device\* device = getDeviceLocked(deviceId); //获取设备对象

status\_t status = NAME\_NOT\_FOUND;

if (device) {

sp<KeyCharacterMap> kcm = device->getKeyCharacterMap();

if (kcm != NULL) {

//根据scanCode找到keyCode【见小节3.4.3】

if (!kcm->mapKey(scanCode, usageCode, outKeycode)) {

\*outFlags = 0;

status = NO\_ERROR;

}

}

}

...

return status;

}

将事件的扫描码(scanCode)转换成键盘码(Keycode)

#### KeyCharacterMap::mapKey

[-> KeyCharacterMap.cpp]

status\_t KeyCharacterMap::mapKey(int32\_t scanCode, int32\_t usageCode, int32\_t\* outKeyCode) const {

...

if (scanCode) {

ssize\_t index = mKeysByScanCode.indexOfKey(scanCode);

if (index >= 0) {

//根据scanCode找到keyCode

\*outKeyCode = mKeysByScanCode.valueAt(index);

return OK;

}

}

\*outKeyCode = AKEYCODE\_UNKNOWN;

return NAME\_NOT\_FOUND;

}

再回到[3.4.1],接下来进入如下过程:

#### InputMapper.processKey

Sdf

[-> InputReader.cpp]

void KeyboardInputMapper::processKey(nsecs\_t when, bool down, int32\_t keyCode,

int32\_t scanCode, uint32\_t policyFlags) {

if (down) {

if (mParameters.orientationAware && mParameters.hasAssociatedDisplay) {

keyCode = rotateKeyCode(keyCode, mOrientation);

}

ssize\_t keyDownIndex = findKeyDown(scanCode);

if (keyDownIndex >= 0) {

//mKeyDowns记录着所有按下的键

keyCode = mKeyDowns.itemAt(keyDownIndex).keyCode;

} else {

...

mKeyDowns.push(); //压入栈顶

KeyDown& keyDown = mKeyDowns.editTop();

keyDown.keyCode = keyCode;

keyDown.scanCode = scanCode;

}

mDownTime = when; //记录按下时间点

} else {

ssize\_t keyDownIndex = findKeyDown(scanCode);

if (keyDownIndex >= 0) {

//键抬起操作，则移除按下事件

keyCode = mKeyDowns.itemAt(keyDownIndex).keyCode;

mKeyDowns.removeAt(size\_t(keyDownIndex));

} else {

return; //键盘没有按下操作，则直接忽略抬起操作

}

}

nsecs\_t downTime = mDownTime;

...

//创建NotifyKeyArgs对象, when记录eventTime, downTime记录按下时间；

NotifyKeyArgs args(when, getDeviceId(), mSource, policyFlags,

down ? AKEY\_EVENT\_ACTION\_DOWN : AKEY\_EVENT\_ACTION\_UP,

AKEY\_EVENT\_FLAG\_FROM\_SYSTEM, keyCode, scanCode, newMetaState, downTime);

//通知key事件【见小节3.4.5】

getListener()->notifyKey(&args);

}

参数说明：

mKeyDowns记录着所有按下的键;

mDownTime记录按下时间点;

此处KeyboardInputMapper的mContext指向InputReader，getListener()获取的便是mQueuedListener。 接下来调用该对象的notifyKey.

#### QueuedInputListener.notifyKey

[-> InputListener.cpp]

void QueuedInputListener::notifyKey(const NotifyKeyArgs\* args) {

mArgsQueue.push(new NotifyKeyArgs(\*args));

}

mArgsQueue的数据类型为Vector<NotifyArgs\*>，将该key事件压人该栈顶。 到此,整个事件加工完成, 再然后就是将事件发送给InputDispatcher线程.

接下来,再回调小节[1.2] InputReader的loopOnce过程, 可知当执行完processEventsLocked()过程, 然后便开始执行mQueuedListener->flush()过程, 如下文.

## QueuedListener

### QueuedInputListener.flush

inputListener.cpp]

void QueuedInputListener::flush() {

size\_t count = mArgsQueue.size();

for (size\_t i = 0; i < count; i++) {

NotifyArgs\* args = mArgsQueue[i];

//【见小节4.2】

args->notify(mInnerListener);

delete args;

}

mArgsQueue.clear();

}

遍历整个mArgsQueue数组, 在input架构中NotifyArgs的实现子类主要有以下几类:

NotifyConfigurationChangedArgs

NotifyKeyArgs

NotifyMotionArgs

NotifySwitchArgs

NotifyDeviceResetArgs

紧接着上述的小节[3.4.5], 可知此处是NotifyKeyArgs对象. 从InputManager对象初始化的过程可知，mInnerListener便是InputDispatcher对象。

### NotifyKeyArgs.notify

-> InputListener.cpp]

void NotifyKeyArgs::notify(const sp<InputListenerInterface>& listener) const {

listener->notifyKey(this); // this是指NotifyKeyArgs【见小节4.3】

}

### InputDispatcher.notifyKey

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::notifyKey(**const** NotifyKeyArgs\* args) {
2. **if** (!validateKeyEvent(args->action)) {
3. **return**;
4. }
5. ...
6. int32\_t keyCode = args->keyCode;
8. **if** (keyCode == AKEYCODE\_HOME) {
9. **if** (args->action == AKEY\_EVENT\_ACTION\_DOWN) {
10. property\_set("sys.domekey.down", "1");
11. } **else** **if** (args->action == AKEY\_EVENT\_ACTION\_UP) {
12. property\_set("sys.domekey.down", "0");
13. }
14. }
16. **if** (metaState & AMETA\_META\_ON && args->action == AKEY\_EVENT\_ACTION\_DOWN) {
17. ...
18. } **else** **if** (args->action == AKEY\_EVENT\_ACTION\_UP) {
19. ...
20. }
22. KeyEvent event; //初始化KeyEvent对象
23. event.initialize(args->deviceId, args->source, args->action,
24. flags, keyCode, args->scanCode, metaState, 0,
25. args->downTime, args->eventTime);
26. //mPolicy是指NativeInputManager对象。【小节4.3.1】
27. mPolicy->interceptKeyBeforeQueueing(&event, /\*byref\*/ policyFlags);
29. **bool** needWake;
30. {
31. mLock.lock();
32. **if** (shouldSendKeyToInputFilterLocked(args)) {
33. mLock.unlock();
34. policyFlags |= POLICY\_FLAG\_FILTERED;
35. //当inputEventObj不为空, 则事件被filter所拦截【见小节4.3.2】
36. **if** (!mPolicy->filterInputEvent(&event, policyFlags)) {
37. **return**;
38. }
39. mLock.lock();
40. }
42. int32\_t repeatCount = 0;
43. //创建KeyEntry对象
44. KeyEntry\* newEntry = **new** KeyEntry(args->eventTime,
45. args->deviceId, args->source, policyFlags,
46. args->action, flags, keyCode, args->scanCode,
47. metaState, repeatCount, args->downTime);
48. //将KeyEntry放入队列【见小节4.3.3】
49. needWake = enqueueInboundEventLocked(newEntry);
50. mLock.unlock();
51. }
53. **if** (needWake) {
54. //唤醒InputDispatcher线程【见小节4.3.5】
55. mLooper->wake();
56. }
57. }

该方法的主要功能：

调用NativeInputManager.interceptKeyBeforeQueueing，加入队列前执行拦截动作，但并不改变流程，调用链：

IMS.interceptKeyBeforeQueueing

InputMonitor.interceptKeyBeforeQueueing (继承IMS.WindowManagerCallbacks)

PhoneWindowManager.interceptKeyBeforeQueueing (继承WindowManagerPolicy)

当mInputFilterEnabled=true(该值默认为false,可通过setInputFilterEnabled设置),则调用NativeInputManager.filterInputEvent过滤输入事件；

当返回值为false则过滤该事件，不再往下分发；

生成KeyEvent，并调用enqueueInboundEventLocked，将该事件加入到InputDispatcherd的成员变量mInboundQueue。

#### interceptKeyBeforeQueueing

void NativeInputManager::interceptKeyBeforeQueueing(const KeyEvent\* keyEvent,

uint32\_t& policyFlags) {

...

if ((policyFlags & POLICY\_FLAG\_TRUSTED)) {

nsecs\_t when = keyEvent->getEventTime(); //时间

JNIEnv\* env = jniEnv();

jobject keyEventObj = android\_view\_KeyEvent\_fromNative(env, keyEvent);

if (keyEventObj) {

// 调用Java层的IMS.interceptKeyBeforeQueueing

wmActions = env->CallIntMethod(mServiceObj,

gServiceClassInfo.interceptKeyBeforeQueueing,

keyEventObj, policyFlags);

...

} else {

...

}

handleInterceptActions(wmActions, when, /\*byref\*/ policyFlags);

} else {

...

}

}

该方法会调用Java层的InputManagerService的interceptKeyBeforeQueueing()方法。

#### filterInputEvent

bool NativeInputManager::filterInputEvent(const InputEvent\* inputEvent, uint32\_t policyFlags) {

jobject inputEventObj;

JNIEnv\* env = jniEnv();

switch (inputEvent->getType()) {

case AINPUT\_EVENT\_TYPE\_KEY:

inputEventObj = android\_view\_KeyEvent\_fromNative(env,

static\_cast<const KeyEvent\*>(inputEvent));

break;

case AINPUT\_EVENT\_TYPE\_MOTION:

inputEventObj = android\_view\_MotionEvent\_obtainAsCopy(env,

static\_cast<const MotionEvent\*>(inputEvent));

break;

default:

return true; // 走事件正常的分发流程

}

if (!inputEventObj) {

return true; // 当inputEventObj为空, 则走事件正常的分发流程

}

//当inputEventObj不为空,则调用Java层的IMS.filterInputEvent()

jboolean pass = env->CallBooleanMethod(mServiceObj, gServiceClassInfo.filterInputEvent,

inputEventObj, policyFlags);

if (checkAndClearExceptionFromCallback(env, "filterInputEvent")) {

pass = true; //出现Exception，则走事件正常的分发流程

}

env->DeleteLocalRef(inputEventObj);

return pass;

}

当inputEventObj不为空,则调用Java层的IMS.filterInputEvent(). 经过层层调用后, 最终会再调用InputDispatcher.injectInputEvent(),该基本等效于该方法的后半段:

enqueueInboundEventLocked

wakeup

#### enqueueInboundEventLocked

asdf

bool InputDispatcher::enqueueInboundEventLocked(EventEntry\* entry) {

bool needWake = mInboundQueue.isEmpty();

mInboundQueue.enqueueAtTail(entry); //将该事件放入mInboundQueue队列尾部

switch (entry->type) {

case EventEntry::TYPE\_KEY: {

KeyEntry\* keyEntry = static\_cast<KeyEntry\*>(entry);

if (isAppSwitchKeyEventLocked(keyEntry)) {

if (keyEntry->action == AKEY\_EVENT\_ACTION\_DOWN) {

mAppSwitchSawKeyDown = true; //按下事件

} else if (keyEntry->action == AKEY\_EVENT\_ACTION\_UP) {

if (mAppSwitchSawKeyDown) {

//其中APP\_SWITCH\_TIMEOUT=500ms

mAppSwitchDueTime = keyEntry->eventTime + APP\_SWITCH\_TIMEOUT;

mAppSwitchSawKeyDown = false;

needWake = true;

}

}

}

break;

}

case EventEntry::TYPE\_MOTION: {

//当前App无响应且用户希望切换到其他应用窗口，则drop该窗口事件，并处理其他窗口事件

MotionEntry\* motionEntry = static\_cast<MotionEntry\*>(entry);

if (motionEntry->action == AMOTION\_EVENT\_ACTION\_DOWN

&& (motionEntry->source & AINPUT\_SOURCE\_CLASS\_POINTER)

&& mInputTargetWaitCause == INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY

&& mInputTargetWaitApplicationHandle != NULL) {

int32\_t displayId = motionEntry->displayId;

int32\_t x = int32\_t(motionEntry->pointerCoords[0].

getAxisValue(AMOTION\_EVENT\_AXIS\_X));

int32\_t y = int32\_t(motionEntry->pointerCoords[0].

getAxisValue(AMOTION\_EVENT\_AXIS\_Y));

//查询可触摸的窗口【见小节4.3.4】

sp<InputWindowHandle> touchedWindowHandle = findTouchedWindowAtLocked(displayId, x, y);

if (touchedWindowHandle != NULL

&& touchedWindowHandle->inputApplicationHandle

!= mInputTargetWaitApplicationHandle) {

mNextUnblockedEvent = motionEntry;

needWake = true;

}

}

break;

}

}

return needWake;

}

AppSwitchKeyEvent是指keyCode等于以下值：

AKEYCODE\_HOME

AKEYCODE\_ENDCALL

AKEYCODE\_APP\_SWITCH

#### findTouchedWindowAtLocked

asdf

[-> InputDispatcher.cpp]

sp<InputWindowHandle> InputDispatcher::findTouchedWindowAtLocked(int32\_t displayId,

int32\_t x, int32\_t y) {

//从前台到后台来遍历查询可触摸的窗口

size\_t numWindows = mWindowHandles.size();

for (size\_t i = 0; i < numWindows; i++) {

sp<InputWindowHandle> windowHandle = mWindowHandles.itemAt(i);

const InputWindowInfo\* windowInfo = windowHandle->getInfo();

if (windowInfo->displayId == displayId) {

int32\_t flags = windowInfo->layoutParamsFlags;

if (windowInfo->visible) {

if (!(flags & InputWindowInfo::FLAG\_NOT\_TOUCHABLE)) {

bool isTouchModal = (flags & (InputWindowInfo::FLAG\_NOT\_FOCUSABLE

| InputWindowInfo::FLAG\_NOT\_TOUCH\_MODAL)) == 0;

if (isTouchModal || windowInfo->touchableRegionContainsPoint(x, y)) {

return windowHandle; //找到目标窗口

}

}

}

}

}

return NULL;

}

此处mWindowHandles的赋值过程是由Java层的InputMonitor.setInputWindows(),经过JNI调用后进入InputDispatcher::setInputWindows()方法完成. 进一步说, 就是WMS执行addWindow()过程或许UI改变等场景,都会触发该方法的修改.

#### Looper.wake

[-> system/core/libutils/Looper.cpp]

void Looper::wake() {

uint64\_t inc = 1;

ssize\_t nWrite = TEMP\_FAILURE\_RETRY(write(mWakeEventFd, &inc, sizeof(uint64\_t)));

if (nWrite != sizeof(uint64\_t)) {

if (errno != EAGAIN) {

ALOGW("Could not write wake signal, errno=%d", errno);

}

}

}

[小节4.3]的过程会调用enqueueInboundEventLocked()方法来决定是否需要将数字1写入句柄mWakeEventFd来唤醒InputDispatcher线程. 满足唤醒的条件:

执行enqueueInboundEventLocked方法前,mInboundQueue队列为空,执行完必然不再为空,则需要唤醒分发线程;

当事件类型为key事件,且发生一对按下和抬起操作,则需要唤醒;

当事件类型为motion事件,且当前可触摸的窗口属于另一个应用,则需要唤醒.

## 总结

### 核心工作

InputReader整个过程涉及多次事件封装转换，其主要工作核心是以下三大步骤:

#### input\_event -> EventEntry：getEvents

调用EventHub的getEvents()读取节点/dev/input的input\_event结构体转换成RawEvent结构体，RawEvent根据不同InputMapper来转换成相应的EventEntry,比如按键事件则对应KeyEntry,触摸事件则对应MotionEntry。转换结果：inut\_event -> EventEntry;

getEvents：通过EventHub(监听目录/dev/input)读取事件放入mEventBuffer,而mEventBuffer是一个大小为256的数组,[见小节2.1]

processEventsLocked: 对事件进行加工, 转换RawEvent -> NotifyKeyArgs(NotifyArgs) [见小节3.1]

QueuedListener->flush：将事件发送到InputDispatcher线程, 转换NotifyKeyArgs -> KeyEntry(EventEntry) [见小节4.1]

InputReader线程不断循环地执行InputReader.loopOnce(), 每次处理完生成的是EventEntry(比如KeyEntry, MotionEntry),

#### 入队拦截

将事件添加到mInboundQueue队列尾部，加入该队列前有以下两个过滤：

IMS.interceptKeyBeforeQueueing：事件分发前可增加业务逻辑；

IMS.filterInputEvent：可拦截事件，当返回值为false的事件都直接拦截，没有机会加入mInboundQueue队列，不会再往下分发；否则进入下一步;

enqueueInboundEventLocked：该事件放入mInboundQueue队列尾部;

mLooper->wake:并根据情况来唤醒InputDispatcher线程.

#### 记录down时间

KeyboardInputMapper.processKey()的过程, 记录下按下down事件的时间点.

接下来的工作就交给InputDispatcher线程。

### 流程图

InputReader的核心工作就是从EventHub获取数据后生成EventEntry事件，加入到InputDispatcher的mInboundQueue队列，再唤醒InputDispatcher线程。





说明:

IMS.filterInputEvent可以过滤无需上报的事件，当该方法返回值为false则代表是需要被过滤掉的事件，无机会交给InputDispatcher来分发。

节点/dev/input的event事件所对应的输入设备信息位于/proc/bus/input/devices，也可以通过getevent来获取事件. 不同的input事件所对应的物理input节点，比如常见的情形：

屏幕触摸和(MENU,HOME,BACK)3按键：对应同一个input设备节点；POWER和音量(下)键：对应同一个input设备节点；

音量(上)键：对应同一个input设备节点；、

# 源码分析-InputDispatcher

上篇文章输入系统之InputReader线程，介绍InputReader利用EventHub获取数据后生成EventEntry事件，加入到InputDispatcher的mInboundQueue队列，再唤醒InputDispatcher线程。本文将介绍InputDispatcher，同样从threadLoop为起点开始分析。

<http://gityuan.com/2016/12/17/input-dispatcher/>

## InputDispatcher起点

### threadLoop

先来回顾一下InputDispatcher对象的初始化过程:

InputDispatcher::InputDispatcher(const sp<InputDispatcherPolicyInterface>& policy) :

mPolicy(policy),

mPendingEvent(NULL), mLastDropReason(DROP\_REASON\_NOT\_DROPPED),

mAppSwitchSawKeyDown(false), mAppSwitchDueTime(LONG\_LONG\_MAX),

mNextUnblockedEvent(NULL),

mDispatchEnabled(false), mDispatchFrozen(false), mInputFilterEnabled(false),

mInputTargetWaitCause(INPUT\_TARGET\_WAIT\_CAUSE\_NONE) {

//创建Looper对象

mLooper = new Looper(false);

mKeyRepeatState.lastKeyEntry = NULL;

//获取分发超时参数

policy->getDispatcherConfiguration(&mConfig);

}

该方法主要工作：

创建属于自己线程的Looper对象；

超时参数来自于IMS，参数默认值keyRepeatTimeout = 500，keyRepeatDelay = 50。

[-> InputDispatcher.cpp]

bool InputDispatcherThread::threadLoop() {

mDispatcher->dispatchOnce(); //【见小节1.2】

return true;

}

整个过程不断循环地调用InputDispatcher的dispatchOnce()来分发事

### dispatchOnce

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::dispatchOnce() {
2. nsecs\_t nextWakeupTime = LONG\_LONG\_MAX;
3. {
4. AutoMutex \_l(mLock);
5. //唤醒等待线程，monitor()用于监控dispatcher是否发生死锁
6. mDispatcherIsAliveCondition.broadcast();
8. **if** (!haveCommandsLocked()) {
9. //当mCommandQueue不为空时处理【见小节2.1】
10. dispatchOnceInnerLocked(&nextWakeupTime);
11. }
13. //【见小节3.1】
14. **if** (runCommandsLockedInterruptible()) {
15. nextWakeupTime = LONG\_LONG\_MIN;
16. }
17. }
19. nsecs\_t currentTime = now();
20. **int** timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);
21. mLooper->pollOnce(timeoutMillis); //进入epoll\_wait
22. }

线程执行Looper->pollOnce，进入epoll\_wait等待状态，当发生以下任一情况则退出等待状态：

callback：通过回调方法来唤醒；

timeout：到达nextWakeupTime时间，超时唤醒；

wake: 主动调用Looper的wake()方法；

## InputDispatcher

### dispatchOnceInnerLocked

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. void InputDispatcher::dispatchOnceInnerLocked(nsecs\_t\* nextWakeupTime) {
2. nsecs\_t currentTime = now(); //当前时间
4. if (!mDispatchEnabled) { //默认值为false
5. resetKeyRepeatLocked(); //重置操作
6. }
7. if (mDispatchFrozen) { //默认值为false
8. return; //当分发被冻结，则不再处理超时和分发事件的工作，直接返回
9. }
11. //优化app切换延迟，当切换超时，则抢占分发，丢弃其他所有即将要处理的事件。
12. bool isAppSwitchDue = mAppSwitchDueTime **<**= currentTime;
13. ...
15. if (!mPendingEvent) {
16. if (mInboundQueue.isEmpty()) {
17. if (!mPendingEvent) {
18. return; //没有事件需要处理，则直接返回
19. }
20. } else {
21. //从mInboundQueue取出头部的事件
22. mPendingEvent = mInboundQueue.dequeueAtHead();
23. }
24. ...
25. resetANRTimeoutsLocked(); //重置ANR信息[见小节2.1.1]
26. }
28. bool done = false;
29. DropReason dropReason = DROP\_REASON\_NOT\_DROPPED;
30. if (!(mPendingEvent-**>**policyFlags & POLICY\_FLAG\_PASS\_TO\_USER)) {
31. dropReason = DROP\_REASON\_POLICY;
32. } else if (!mDispatchEnabled) {
33. dropReason = DROP\_REASON\_DISABLED;
34. }
35. ...
37. switch (mPendingEvent-**>**type) {
38. case EventEntry::TYPE\_KEY: {
39. KeyEntry\* typedEntry = static\_cast**<KeyEntry**\***>**(mPendingEvent);
40. if (isAppSwitchDue) {
41. if (isAppSwitchKeyEventLocked(typedEntry)) {
42. resetPendingAppSwitchLocked(true);
43. isAppSwitchDue = false;
44. } else if (dropReason == DROP\_REASON\_NOT\_DROPPED) {
45. dropReason = DROP\_REASON\_APP\_SWITCH;
46. }
47. }
48. if (dropReason == DROP\_REASON\_NOT\_DROPPED
49. && isStaleEventLocked(currentTime, typedEntry)) {
50. dropReason = DROP\_REASON\_STALE;
51. }
52. if (dropReason == DROP\_REASON\_NOT\_DROPPED && mNextUnblockedEvent) {
53. dropReason = DROP\_REASON\_BLOCKED;
54. }
55. // 分发按键事件[见小节2.2]
56. done = dispatchKeyLocked(currentTime, typedEntry, &dropReason, nextWakeupTime);
57. break;
58. }
59. **case EventEntry::TYPE\_MOTION:** {
60. MotionEntry\* typedEntry = static\_cast<MotionEntry\*>(mPendingEvent);
61. if (dropReason == DROP\_REASON\_NOT\_DROPPED && isAppSwitchDue) {
62. dropReason = DROP\_REASON\_APP\_SWITCH;
63. }
64. if (dropReason == DROP\_REASON\_NOT\_DROPPED
65. && isStaleEventLocked(currentTime, typedEntry)) {
66. dropReason = DROP\_REASON\_STALE;
67. }
68. if (dropReason == DROP\_REASON\_NOT\_DROPPED && mNextUnblockedEvent) {
69. dropReason = DROP\_REASON\_BLOCKED;
70. }
71. done = **dispatchMotionLocked**(currentTime, typedEntry,
72. &dropReason, nextWakeupTime);
73. break;
74. }
75. ...
76. }
77. ...
79. //分发操作完成，则进入该分支
80. if (done) {
81. if (dropReason != DROP\_REASON\_NOT\_DROPPED) {
82. //[见小节2.1.2]
83. dropInboundEventLocked(mPendingEvent, dropReason);
84. }
85. mLastDropReason = dropReason;
86. releasePendingEventLocked(); //释放pending事件见小节2.10]
87. \*nextWakeupTime = LONG\_LONG\_MIN; //强制立刻执行轮询
88. }
89. }

在enqueueInboundEventLocked()的过程中已设置mAppSwitchDueTime等于eventTime加上500ms:

mAppSwitchDueTime = keyEntry->eventTime + APP\_SWITCH\_TIMEOUT;

该方法主要功能:

mDispatchFrozen用于决定是否冻结事件分发工作不再往下执行;

当事件分发的时间点距离该事件加入mInboundQueue的时间超过500ms,则认为app切换过期,即isAppSwitchDue=true;

mInboundQueue不为空,则取出头部的事件,放入mPendingEvent变量;并重置ANR时间;

根据EventEntry的type类型分别处理，比如按键调用dispatchKeyLocked分发事件;再根据分发结果来决定是否进入done;

执行完成(done)的处理:

根据dropReason(默认NOT\_DROPPED不处理)来决定是否丢失事件; dropInboundEventLocked

释放当前正在处理的事件(即mPendingEvent)； releasePendingEventLocked

关于dispatchKeyLocked分发事件,

不会执行done过情况:

当前Event时间小于唤醒时间;

让policy有机会执行拦截操作;

调用findFocusedWindowTargetsLocked方法的返回结果是INPUT\_EVENT\_INJECTION\_PENDING, 即targets没有处于Ready状态;

会执行done的情况:

该事件需要丢弃, 即dropReason != DROP\_REASON\_NOT\_DROPPED;

findFocusedWindowTargetsLocked的返回结果不是INPUT\_EVENT\_INJECTION\_PENDING(没有正在处理的事件);

接下来以按键为例来展开说明, 则进入[小节2.2] dispatchKeyLocked.

#### resetANRTimeoutsLocked

sdf

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. void InputDispatcher::resetANRTimeoutsLocked() {
2. // 重置等待超时cause和handle
3. mInputTargetWaitCause = INPUT\_TARGET\_WAIT\_CAUSE\_NONE;
4. mInputTargetWaitApplicationHandle.clear();
5. }

#### dropInboundEventLocked

sdf

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. void InputDispatcher::dropInboundEventLocked(EventEntry\* entry, DropReason dropReason) {
2. const char\* reason;
3. switch (dropReason) {
4. case DROP\_REASON\_POLICY:
5. reason = "inbound event was dropped because the policy consumed it";
6. break;
7. case DROP\_REASON\_DISABLED:
8. if (mLastDropReason != DROP\_REASON\_DISABLED) {
9. ALOGI("Dropped event because input dispatch is disabled.");
10. }
11. reason = "inbound event was dropped because input dispatch is disabled";
12. break;
13. case DROP\_REASON\_APP\_SWITCH:
14. ALOGI("Dropped event because of pending overdue app switch.");
15. reason = "inbound event was dropped because of pending overdue app switch";
16. break;
17. case DROP\_REASON\_BLOCKED:
18. ALOGI("Dropped event because the current application is not responding and the user "
19. "has started interacting with a different application.");
20. reason = "inbound event was dropped because the current application is not responding "
21. "and the user has started interacting with a different application";
22. break;
23. case DROP\_REASON\_STALE:
24. ALOGI("Dropped event because it is stale.");
25. reason = "inbound event was dropped because it is stale";
26. break;
27. default:
28. return;
29. }
31. switch (entry-**>**type) {
32. case EventEntry::TYPE\_KEY: {
33. CancelationOptions options(CancelationOptions::CANCEL\_NON\_POINTER\_EVENTS, reason);
34. synthesizeCancelationEventsForAllConnectionsLocked(options);
35. break;
36. }
37. ...
38. }
39. }

### dispatchKeyLocked

sdf

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. bool InputDispatcher::dispatchKeyLocked(nsecs\_t currentTime, KeyEntry\* entry,
2. DropReason\* dropReason, nsecs\_t\* nextWakeupTime) {
3. ...
4. if (entry-**>**interceptKeyResult == KeyEntry::INTERCEPT\_KEY\_RESULT\_TRY\_AGAIN\_LATER) {
5. // case1: 当前时间小于唤醒时间，则进入等待状态。
6. if (currentTime **<** **entry->**interceptKeyWakeupTime) {
7. if (entry-**>**interceptKeyWakeupTime **<** \*nextWakeupTime) {
8. \*nextWakeupTime = entry-**>**interceptKeyWakeupTime;
9. }
10. return false; //直接返回
11. }
12. entry-**>**interceptKeyResult = KeyEntry::INTERCEPT\_KEY\_RESULT\_UNKNOWN;
13. entry-**>**interceptKeyWakeupTime = 0;
14. }
16. if (entry-**>**interceptKeyResult == KeyEntry::INTERCEPT\_KEY\_RESULT\_UNKNOWN) {
17. //case2: 让policy有机会执行拦截操作
18. if (entry-**>**policyFlags & POLICY\_FLAG\_PASS\_TO\_USER) {
19. CommandEntry\* commandEntry = postCommandLocked(
20. & InputDispatcher::doInterceptKeyBeforeDispatchingLockedInterruptible);
21. if (mFocusedWindowHandle != NULL) {
22. commandEntry-**>**inputWindowHandle = mFocusedWindowHandle;
23. }
24. commandEntry-**>**keyEntry = entry;
25. entry-**>**refCount += 1;
26. return false; //直接返回
27. } else {
28. entry-**>**interceptKeyResult = KeyEntry::INTERCEPT\_KEY\_RESULT\_CONTINUE;
29. }
30. } else if (entry-**>**interceptKeyResult == KeyEntry::INTERCEPT\_KEY\_RESULT\_SKIP) {
31. if (\*dropReason == DROP\_REASON\_NOT\_DROPPED) {
32. \*dropReason = DROP\_REASON\_POLICY;
33. }
34. }
36. //case3: 如果需要丢弃该事件，则执行清理操作
37. if (\*dropReason != DROP\_REASON\_NOT\_DROPPED) {
38. setInjectionResultLocked(entry, \*dropReason == DROP\_REASON\_POLICY
39. ? INPUT\_EVENT\_INJECTION\_SUCCEEDED : INPUT\_EVENT\_INJECTION\_FAILED);
40. return true; //直接返回
41. }
43. Vector**<InputTarget>** inputTargets;
44. //case4: 寻找焦点 【见小节2.3】
45. int32\_t injectionResult = findFocusedWindowTargetsLocked(currentTime,
46. entry, inputTargets, nextWakeupTime);
47. if (injectionResult == INPUT\_EVENT\_INJECTION\_PENDING) {
48. return false; //直接返回
49. }
51. setInjectionResultLocked(entry, injectionResult);
52. if (injectionResult != INPUT\_EVENT\_INJECTION\_SUCCEEDED) {
53. return true; //直接返回
54. }
55. addMonitoringTargetsLocked(inputTargets);
57. //只有injectionResult是成功，才有机会执行分发事件【见小节2.5】
58. dispatchEventLocked(currentTime, entry, inputTargets);
59. return true;
60. }

在以下场景下，有可能无法分发事件：

当前时间小于唤醒时间(nextWakeupTime)的情况；

policy需要提前拦截事件的情况；

需要drop事件的情况；

寻找聚焦窗口失败的情况；

如果成功跳过以上所有情况，则会进入执行事件分发的过程

### findFocusedWindowTargetsLocked

asd

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. int32\_t InputDispatcher::findFocusedWindowTargetsLocked(nsecs\_t currentTime,
2. const EventEntry\* entry, Vector**<InputTarget>**& inputTargets, nsecs\_t\* nextWakeupTime) {
3. int32\_t injectionResult;
4. String8 reason;
6. if (mFocusedWindowHandle == NULL) {
7. if (mFocusedApplicationHandle != NULL) {
8. //【见小节2.3.2】
9. injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
10. mFocusedApplicationHandle, NULL, nextWakeupTime,
11. "Waiting because no window has focus but there is a "
12. "focused application that may eventually add a window "
13. "when it finishes starting up.");
14. goto Unresponsive;
15. }
17. ALOGI("Dropping event because there is no focused window or focused application.");
18. injectionResult = INPUT\_EVENT\_INJECTION\_FAILED;
19. goto Failed;
20. }
22. //权限检查
23. if (! checkInjectionPermission(mFocusedWindowHandle, entry-**>**injectionState)) {
24. injectionResult = INPUT\_EVENT\_INJECTION\_PERMISSION\_DENIED;
25. goto Failed;
26. }
28. //检测窗口是否为更多的输入操作而准备就绪【见小节2.3.1】
29. reason = checkWindowReadyForMoreInputLocked(currentTime,
30. mFocusedWindowHandle, entry, "focused");
31. if (!reason.isEmpty()) {
32. //【见小节2.3.2】
33. injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
34. mFocusedApplicationHandle, mFocusedWindowHandle, nextWakeupTime, reason.string());
35. goto Unresponsive;
36. }
38. injectionResult = INPUT\_EVENT\_INJECTION\_SUCCEEDED;
39. //成功找到目标窗口，添加到目标窗口 [见小节2.3.3]
40. **addWindowTargetLocked**(mFocusedWindowHandle,
41. InputTarget::FLAG\_FOREGROUND | InputTarget::FLAG\_DISPATCH\_AS\_IS, BitSet32(0),
42. inputTargets);
44. Failed:
45. Unresponsive:
46. //TODO: 统计等待时长信息，目前没有实现，这个方法还是很值得去改造
47. nsecs\_t timeSpentWaitingForApplication = getTimeSpentWaitingForApplicationLocked(currentTime);
48. updateDispatchStatisticsLocked(currentTime, entry,
49. injectionResult, timeSpentWaitingForApplication);
50. return injectionResult;
51. }

此处mFocusedWindowHandle是何处赋值呢？是在InputDispatcher.setInputWindows()方法，具体见下一篇文章Input系统—UI线程.

寻找聚焦窗口失败的情况：

无窗口，无应用：Dropping event because there is no focused window or focused application.(这并不导致ANR的情况，因为没有机会调用handleTargetsNotReadyLocked)

无窗口, 有应用：Waiting because no window has focus but there is a focused application that may eventually add a window when it finishes starting up.

另外，还有更多多的失败场景见checkWindowReadyForMoreInputLocked的过程，如下：

#### checkWindowReadyForMoreInputLocked

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. String8 InputDispatcher::checkWindowReadyForMoreInputLocked(nsecs\_t currentTime,
2. const sp**<InputWindowHandle>**& windowHandle, const EventEntry\* eventEntry,
3. const char\* targetType) {
4. //当窗口暂停的情况，则保持等待
5. if (windowHandle-**>**getInfo()-**>**paused) {
6. return String8::format("Waiting because the %s window is paused.", targetType);
7. }
9. //当窗口连接未注册，则保持等待
10. ssize\_t connectionIndex = getConnectionIndexLocked(windowHandle-**>**getInputChannel());
11. if (connectionIndex **<** **0**) {
12. return String8::format("Waiting because the %s window's input channel is not "
13. "registered with the input dispatcher. The window may be in the process "
14. "of being removed.", targetType);
15. }
17. //当窗口连接已死亡，则保持等待
18. sp**<Connection>** connection = mConnectionsByFd.valueAt(connectionIndex);
19. if (connection-**>**status != Connection::STATUS\_NORMAL) {
20. return String8::format("Waiting because the %s window's input connection is %s."
21. "The window may be in the process of being removed.", targetType,
22. connection-**>**getStatusLabel());
23. }
25. // 当窗口连接已满，则保持等待
26. if (connection-**>**inputPublisherBlocked) {
27. return String8::format("Waiting because the %s window's input channel is full. "
28. "Outbound queue length: %d. Wait queue length: %d.",
29. targetType, connection-**>**outboundQueue.count(), connection-**>**waitQueue.count());
30. }

33. if (eventEntry-**>**type == EventEntry::TYPE\_KEY) {
34. // 按键事件，输出队列或事件等待队列不为空
35. if (!connection-**>**outboundQueue.isEmpty() || !connection-**>**waitQueue.isEmpty()) {
36. return String8::format("Waiting to send key event because the %s window has not "
37. "finished processing all of the input events that were previously "
38. "delivered to it. Outbound queue length: %d. Wait queue length: %d.",
39. targetType, connection-**>**outboundQueue.count(), connection-**>**waitQueue.count());
40. }
41. } else {
42. // 非按键事件，事件等待队列不为空且头事件分发超时500ms
43. if (!connection-**>**waitQueue.isEmpty()
44. && currentTime **>**= connection-**>**waitQueue.head-**>**deliveryTime
45. + STREAM\_AHEAD\_EVENT\_TIMEOUT) {
46. return String8::format("Waiting to send non-key event because the %s window has not "
47. "finished processing certain input events that were delivered to it over "
48. "%0.1fms ago. Wait queue length: %d. Wait queue head age: %0.1fms.",
49. targetType, STREAM\_AHEAD\_EVENT\_TIMEOUT \* 0.000001f,
50. connection-**>**waitQueue.count(),
51. (currentTime - connection-**>**waitQueue.head-**>**deliveryTime) \* 0.000001f);
52. }
53. }
54. return String8::empty();
55. }

#### handleTargetsNotReadyLocked

asdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. int32\_t InputDispatcher::handleTargetsNotReadyLocked(nsecs\_t currentTime,
2. **const** EventEntry\* entry,
3. **const** sp<InputApplicationHandle>& applicationHandle,
4. **const** sp<InputWindowHandle>& windowHandle,
5. nsecs\_t\* nextWakeupTime, **const** **char**\* reason) {
6. **if** (applicationHandle == NULL && windowHandle == NULL) {
7. **if** (mInputTargetWaitCause != INPUT\_TARGET\_WAIT\_CAUSE\_SYSTEM\_NOT\_READY) {
8. mInputTargetWaitCause = INPUT\_TARGET\_WAIT\_CAUSE\_SYSTEM\_NOT\_READY;
9. mInputTargetWaitStartTime = currentTime; //当前时间
10. mInputTargetWaitTimeoutTime = LONG\_LONG\_MAX;
11. mInputTargetWaitTimeoutExpired = **false**;
12. mInputTargetWaitApplicationHandle.clear();
13. }
14. } **else** {
15. **if** (mInputTargetWaitCause != INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY) {
16. nsecs\_t timeout;
17. **if** (windowHandle != NULL) {
18. timeout = windowHandle->getDispatchingTimeout(DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT);
19. } **else** **if** (applicationHandle != NULL) {
20. timeout = applicationHandle->getDispatchingTimeout(DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT);
21. } **else** {
22. timeout = DEFAULT\_INPUT\_DISPATCHING\_TIMEOUT; // 5s
23. }
24. //买砸蛋设置等待的原因，等待开始的时间，等待超时的时
25. mInputTargetWaitCause = INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY;
26. mInputTargetWaitStartTime = currentTime; //当前时间
27. mInputTargetWaitTimeoutTime = currentTime + timeout;
28. mInputTargetWaitTimeoutExpired = **false**;
29. mInputTargetWaitApplicationHandle.clear();
31. **if** (windowHandle != NULL) {
32. mInputTargetWaitApplicationHandle = windowHandle->inputApplicationHandle;
33. }
34. **if** (mInputTargetWaitApplicationHandle == NULL && applicationHandle != NULL) {
35. mInputTargetWaitApplicationHandle = applicationHandle;
36. }
37. }
38. }
40. **if** (mInputTargetWaitTimeoutExpired) {
41. **return** INPUT\_EVENT\_INJECTION\_TIMED\_OUT; //等待超时已过期,则直接返回
42. }
43. **//当前时间超过设定的超时时间**
44. //当超时5s则进入ANR流程
45. **if** (currentTime >= mInputTargetWaitTimeoutTime) {
46. onANRLocked(currentTime, applicationHandle, windowHandle,
47. entry->eventTime, mInputTargetWaitStartTime, reason);
49. \*nextWakeupTime = LONG\_LONG\_MIN; //强制立刻执行轮询来执行ANR策略
50. **return** INPUT\_EVENT\_INJECTION\_PENDING;
51. } **else** {
52. **if** (mInputTargetWaitTimeoutTime < \*nextWakeupTime) {
53. \*nextWakeupTime = mInputTargetWaitTimeoutTime; //当触发超时则强制执行轮询
54. }
55. **return** INPUT\_EVENT\_INJECTION\_PENDING;
56. }
57. }

此处mInputTargetWaitTimeoutTime是由当前时间戳+5s, 并设置mInputTargetWaitCause等于INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY. 也就是说ANR时间段是指input等待理由处于INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY(应用没有准备就绪)的时间长达5s的场景.

而前面resetANRTimeoutsLocked() 过程是唯一用于重置等待理由的地方.

那么, ANR时间区间是指当前这次的事件dispatch过程中执行findFocusedWindowTargetsLocked()方法到下一次执行resetANRTimeoutsLocked()的时间区间.

当applicationHandle和windowHandle同时为空, 且system准备就绪的情况下

设置等待理由 INPUT\_TARGET\_WAIT\_CAUSE\_SYSTEM\_NOT\_READY;

设置超时等待时长为无限大;

设置TimeoutExpired= false

清空等待队列;

当applicationHandle和windowHandle至少一个不为空, 且application准备就绪的情况下:

设置等待理由 INPUT\_TARGET\_WAIT\_CAUSE\_APPLICATION\_NOT\_READY;

设置超时等待时长为5s;

设置TimeoutExpired= false

清空等待队列;

继续回到[小节2.3]findFocusedWindowTargetsLocked，如果没有发生ANR，则addWindowTargetLocked()将该事件添加到inputTargets。

#### addWindowTargetLocked

asdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::addWindowTargetLocked(**const** sp<InputWindowHandle>& windowHandle,
2. int32\_t targetFlags, BitSet32 pointerIds, Vector<InputTarget>& inputTargets) {
3. inputTargets.push();
5. **const** InputWindowInfo\* windowInfo = windowHandle->getInfo();
6. InputTarget& target = inputTargets.editTop();
7. **target.inputChannel = windowInfo->inputChannel;**
8. target.flags = targetFlags;
9. target.xOffset = - windowInfo->frameLeft;
10. target.yOffset = - windowInfo->frameTop;
11. target.scaleFactor = windowInfo->scaleFactor;
12. target.pointerIds = pointerIds;
13. }

将当前聚焦窗口mFocusedWindowHandle的inputChannel传递到inputTargets。

### dispatchEventLocked

asdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::dispatchEventLocked(nsecs\_t currentTime,
2. EventEntry\* eventEntry, **const** Vector<InputTarget>& inputTargets) {
3. //【见小节2.4.1】向mCommandQueue队列添加doPokeUserActivityLockedInterruptible命令
4. pokeUserActivityLocked(eventEntry);
6. **for** (**size\_t** i = 0; i < inputTargets.size(); i++) {
7. **const** InputTarget& inputTarget = inputTargets.itemAt(i);
8. //[见小节2.4.3]
9. ssize\_t connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);
10. **if** (connectionIndex >= 0) {
11. sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);
12. //找到目标连接[见小节２.5]
13. prepareDispatchCycleLocked(currentTime, connection, eventEntry, &inputTarget);
14. }
15. }
16. }

该方法主要功能是将eventEntry发送到目标inputTargets．

其中pokeUserActivityLocked(eventEntry)方法最终会调用到Java层的PowerManagerService.java中的**userActivityFromNative**()方法． 这也是PMS中唯一的native call方法．

#### pokeUserActivityLocked

sdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::pokeUserActivityLocked(**const** EventEntry\* eventEntry) {
2. **if** (mFocusedWindowHandle != NULL) {
3. **const** InputWindowInfo\* info = mFocusedWindowHandle->getInfo();
4. **if** (info->inputFeatures & InputWindowInfo::INPUT\_FEATURE\_DISABLE\_USER\_ACTIVITY) {
5. **return**;
6. }
7. }
8. ...
9. //【见小节2.4.2】
10. CommandEntry\* commandEntry = postCommandLocked(
11. & InputDispatcher::doPokeUserActivityLockedInterruptible);
12. commandEntry->eventTime = eventEntry->eventTime;
13. commandEntry->userActivityEventType = eventType;
14. }

#### postCommandLocked

sadf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. InputDispatcher::CommandEntry\* InputDispatcher::postCommandLocked(Command command) {
2. CommandEntry\* commandEntry = **new** CommandEntry(command);
3. // 将命令加入mCommandQueue队尾
4. mCommandQueue.enqueueAtTail(commandEntry);
5. **return** commandEntry;
6. }

#### getConnectionIndexLocked

asd

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. ssize\_t InputDispatcher::getConnectionIndexLocked(**const** sp<InputChannel>& inputChannel) {
2. ssize\_t connectionIndex = mConnectionsByFd.indexOfKey(inputChannel->getFd());
3. **if** (connectionIndex >= 0) {
4. sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);
5. **if** (connection->inputChannel.get() == inputChannel.get()) {
6. **return** connectionIndex;
7. }
8. }
9. **return** -1;
10. }

根据inputChannel的fd从mConnectionsByFd队列中查询目标connection.

### prepareDispatchCycleLocked

sdf

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::prepareDispatchCycleLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection, EventEntry\* eventEntry, **const** InputTarget\* inputTarget) {
4. **if** (connection->status != Connection::STATUS\_NORMAL) {
5. **return**;　//当连接已破坏,则直接返回
6. }
7. ...
9. //[见小节2.6]
10. enqueueDispatchEntriesLocked(currentTime, connection, eventEntry, inputTarget);
11. }

当connection状态不正确，则直接返回。

### enqueueDispatchEntriesLocked

asd

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::enqueueDispatchEntriesLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection, EventEntry\* eventEntry, **const** InputTarget\* inputTarget) {
3. **bool** wasEmpty = connection->outboundQueue.isEmpty();
5. //[见小节2.7]
6. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
7. InputTarget::FLAG\_DISPATCH\_AS\_HOVER\_EXIT);
8. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
9. InputTarget::FLAG\_DISPATCH\_AS\_OUTSIDE);
10. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
11. InputTarget::FLAG\_DISPATCH\_AS\_HOVER\_ENTER);
12. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
13. InputTarget::FLAG\_DISPATCH\_AS\_IS);
14. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
15. InputTarget::FLAG\_DISPATCH\_AS\_SLIPPERY\_EXIT);
16. enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
17. InputTarget::FLAG\_DISPATCH\_AS\_SLIPPERY\_ENTER);
19. **if** (wasEmpty && !connection->outboundQueue.isEmpty()) {
20. //当原先的outbound队列为空, 且当前outbound不为空的情况执行.[见小节2.8]
21. startDispatchCycleLocked(currentTime, connection);
22. }
23. }

该方法主要功能：

根据dispatchMode来分别执行DispatchEntry事件加入队列的操作。

当起初connection.outboundQueue等于空, 经enqueueDispatchEntryLocked处理后, outboundQueue不等于空情况下, 则执行startDispatchCycleLocked()方法.

### enqueueDispatchEntryLocked

dfa

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::enqueueDispatchEntryLocked(
2. **const** sp<Connection>& connection, EventEntry\* eventEntry, **const** InputTarget\* inputTarget,
3. int32\_t dispatchMode) {
4. int32\_t inputTargetFlags = inputTarget->flags;
5. **if** (!(inputTargetFlags & dispatchMode)) {
6. **return**; //分发模式不匹配,则直接返回
7. }
8. inputTargetFlags = (inputTargetFlags & ~InputTarget::FLAG\_DISPATCH\_MASK) | dispatchMode;
10. //生成新的事件, 加入connection的outbound队列
11. DispatchEntry\* dispatchEntry = **new** DispatchEntry(eventEntry,
12. inputTargetFlags, inputTarget->xOffset, inputTarget->yOffset,
13. inputTarget->scaleFactor);
15. **switch** (eventEntry->type) {
16. **case** EventEntry::TYPE\_KEY: {
17. KeyEntry\* keyEntry = **static\_cast**<KeyEntry\*>(eventEntry);
18. dispatchEntry->resolvedAction = keyEntry->action;
19. dispatchEntry->resolvedFlags = keyEntry->flags;
21. **if** (!connection->inputState.trackKey(keyEntry,
22. dispatchEntry->resolvedAction, dispatchEntry->resolvedFlags)) {
23. **delete** dispatchEntry;
24. **return**; //忽略不连续的事件
25. }
26. **break**;
27. }
28. ...
29. }
30. ...
32. //添加到outboundQueue队尾
33. connection->outboundQueue.enqueueAtTail(dispatchEntry);
34. }

该方法主要功能:

根据dispatchMode来决定是否需要加入outboundQueue队列;

根据EventEntry,来生成DispatchEntry事件;

将dispatchEntry加入到connection的outbound队列.

执行到这里,其实等于由做了一次搬运的工作,**将InputDispatcher中mInboundQueue中的事件取出后, 找到目标window后,**封装dispatchEntry加入到connection的outbound队列.

### startDispatchCycleLocked

dsfg

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **void** InputDispatcher::startDispatchCycleLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection) {
4. //当Connection状态正常,且outboundQueue不为空
5. **while** (connection->status == Connection::STATUS\_NORMAL
6. && !connection->outboundQueue.isEmpty()) {
7. DispatchEntry\* dispatchEntry = connection->outboundQueue.head;
8. dispatchEntry->deliveryTime = currentTime; //设置deliveryTime时间
10. status\_t status;
11. EventEntry\* eventEntry = dispatchEntry->eventEntry;
12. **switch** (eventEntry->type) {
13. **case** EventEntry::TYPE\_KEY: {
14. KeyEntry\* keyEntry = **static\_cast**<KeyEntry\*>(eventEntry);
16. //发布Key事件 [见小节2.9]
17. status = connection->inputPublisher.publishKeyEvent(dispatchEntry->seq,
18. keyEntry->deviceId, keyEntry->source,
19. dispatchEntry->resolvedAction, dispatchEntry->resolvedFlags,
20. keyEntry->keyCode, keyEntry->scanCode,
21. keyEntry->metaState, keyEntry->repeatCount, keyEntry->downTime,
22. keyEntry->eventTime);
23. **break**;
24. }
25. ...
26. }
28. **if** (status) { //publishKeyEvent失败情况
29. **if** (status == WOULD\_BLOCK) {
30. **if** (connection->waitQueue.isEmpty()) {
31. //pipe已满,但waitQueue为空. 不正常的行为
32. abortBrokenDispatchCycleLocked(currentTime, connection, **true** /\*notify\*/);
33. } **else** {
34. // 处于阻塞状态
35. connection->inputPublisherBlocked = **true**;
36. }
37. } **else** {
38. //不不正常的行为
39. abortBrokenDispatchCycleLocked(currentTime, connection, **true** /\*notify\*/);
40. }
41. **return**;
42. }
44. //从outboundQueue中取出事件,重新放入waitQueue队列
45. connection->outboundQueue.dequeue(dispatchEntry);
46. connection->waitQueue.enqueueAtTail(dispatchEntry);
48. }
49. }

startDispatchCycleLocked的主要功能: 从outboundQueue中取出事件,重新放入waitQueue队列

startDispatchCycleLocked触发时机：当起初connection.outboundQueue等于空, 经enqueueDispatchEntryLocked处理后, outboundQueue不等于空。

startDispatchCycleLocked主要功能: 从outboundQueue中取出事件,重新放入waitQueue队列

publishKeyEvent执行结果status不等于OK的情况下：

WOULD\_BLOCK，且waitQueue等于空，则调用abortBrokenDispatchCycleLocked()，该方法最终会调用到Java层的IMS.notifyInputChannelBroken().

WOULD\_BLOCK，且waitQueue不等于空，则处于阻塞状态，即inputPublisherBlocked=true

其他情况，则调用abortBrokenDispatchCycleLocked

abortBrokenDispatchCycleLocked()方法最终会调用到Java层的IMS.notifyInputChannelBroken().

### inputPublisher.publishKeyEvent发布目标窗口事件

[-> InputTransport.cpp]

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. status\_t InputPublisher::publishKeyEvent(...) {
2. **if** (!seq) {
3. **return** BAD\_VALUE;
4. }
6. InputMessage msg;
7. msg.header.type = InputMessage::TYPE\_KEY;
8. msg.body.key.seq = seq;
9. msg.body.key.deviceId = deviceId;
10. msg.body.key.source = source;
11. msg.body.key.action = action;
12. msg.body.key.flags = flags;
13. msg.body.key.keyCode = keyCode;
14. msg.body.key.scanCode = scanCode;
15. msg.body.key.metaState = metaState;
16. msg.body.key.repeatCount = repeatCount;
17. msg.body.key.downTime = downTime;
18. msg.body.key.eventTime = eventTime;
19. //通过InputChannel来发送消息
20. **return** mChannel->sendMessage(&msg);
21. }

InputChannel通过socket向远端的socket发送消息。socket通道是如何建立的呢？ InputDispatcher又是如何与前台的window通信的呢？ 见下一篇文章Input系统—进程交互, 从文章的小节2.1开始继续往下说.

### releasePendingEventLocked

void InputDispatcher::releasePendingEventLocked() {

if (mPendingEvent) {

resetANRTimeoutsLocked(); //重置ANR超时时间

releaseInboundEventLocked(mPendingEvent); //释放mPendingEvent对象,并记录到mRecentQueue队列

mPendingEvent = NULL; //置空mPendingEvent变量.

}

}

## 处理Comand

### runCommandsLockedInterruptible

bool InputDispatcher::runCommandsLockedInterruptible() {

if (mCommandQueue.isEmpty()) {

return false;

}

do {

//从mCommandQueue队列的头部取出第一个元素

CommandEntry\* commandEntry = mCommandQueue.dequeueAtHead();

Command command = commandEntry->command;

//此处调用的命令隐式地包含'LockedInterruptible'

(this->\*command)(commandEntry);

commandEntry->connection.clear();

delete commandEntry;

} while (! mCommandQueue.isEmpty());

return true;

}

通过循环方式处理完mCommandQueue队列的所有命令，处理过程从mCommandQueue中取出CommandEntry.

typedef void (InputDispatcher::\*Command)(CommandEntry\* commandEntry);

struct CommandEntry : Link<CommandEntry> {

CommandEntry(Command command);

Command command;

sp<Connection> connection;

nsecs\_t eventTime;

KeyEntry\* keyEntry;

sp<InputApplicationHandle> inputApplicationHandle;

sp<InputWindowHandle> inputWindowHandle;

String8 reason;

int32\_t userActivityEventType;

uint32\_t seq;

bool handled;

};

前面小节【2.4.1】添加的doPokeUserActivityLockedInterruptible命令. 接下来进入该方法：

### doPokeUserActivityLockedInterruptible

[-> InputDispatcher]

void InputDispatcher::doPokeUserActivityLockedInterruptible(CommandEntry\* commandEntry) {

mLock.unlock();

//【见小节4.3】

mPolicy->pokeUserActivity(commandEntry->eventTime, commandEntry->userActivityEventType);

mLock.lock();

}

### pokeUserActivity

[-> com\_android\_server\_input\_InputManagerService.cpp]

void NativeInputManager::pokeUserActivity(nsecs\_t eventTime, int32\_t eventType) {

//[见小节4.4]

android\_server\_PowerManagerService\_userActivity(eventTime, eventType);

}

### android\_server\_PowerManagerService\_userActivity

dfg

[-> com\_android\_server\_power\_PowerManagerService.cpp]

void android\_server\_PowerManagerService\_userActivity(nsecs\_t eventTime, int32\_t eventType) {

// Tell the power HAL when user activity occurs.

if (gPowerModule && gPowerModule->powerHint) {

gPowerModule->powerHint(gPowerModule, POWER\_HINT\_INTERACTION, NULL);

}

if (gPowerManagerServiceObj) {

...

//[见小节4.5]

env->CallVoidMethod(gPowerManagerServiceObj,

gPowerManagerServiceClassInfo.userActivityFromNative,

nanoseconds\_to\_milliseconds(eventTime), eventType, 0);

}

}

### PowerMS.userActivityFromNative

[-> PowerManagerService.java]

private void userActivityFromNative(long eventTime, int event, int flags) {

userActivityInternal(eventTime, event, flags, Process.SYSTEM\_UID);

}

private void userActivityInternal(long eventTime, int event, int flags, int uid) {

synchronized (mLock) {

if (userActivityNoUpdateLocked(eventTime, event, flags, uid)) {

updatePowerStateLocked();

}

}

}

runCommandsLockedInterruptible是不断地从mCommandQueue队列取出命令，然后执行直到全部执行完成。 除了doPokeUserActivityLockedInterruptible，还有其他如下命令：

doNotifyANRLockedInterruptible

doInterceptKeyBeforeDispatchingLockedInterruptible

doDispatchCycleFinishedLockedInterruptible

doNotifyInputChannelBrokenLockedInterruptible

doNotifyConfigurationChangedInterruptible

## 总结

### 流程图



### 核心方法



图解:

dispatchOnceInnerLocked(): 从InputDispatcher的mInboundQueue队列，取出事件EventEntry。另外该方法开始执行的时间点(currentTime)便是后续事件dispatchEntry的分发时间(deliveryTime）

dispatchKeyLocked()：满足一定条件时会添加命令doInterceptKeyBeforeDispatchingLockedInterruptible；

enqueueDispatchEntryLocked()：生成事件DispatchEntry并加入connection的outbound队列

startDispatchCycleLocked()：从outboundQueue中取出事件DispatchEntry, 重新放入connection的waitQueue队列；

InputChannel.sendMessage通过socket方式将消息发送给远程进程；

runCommandsLockedInterruptible()：通过循环遍历地方式，依次处理mCommandQueue队列中的所有命令。而mCommandQueue队列中的命令是通过postCommandLocked()方式向该队列添加的。

## Motion事件分发

mPolicy->interceptMotionBeforeQueueing(args->eventTime, /\*byref\*/ policyFlags);

# UI线程

## 概述

前面文章都是介绍了两个线程InputReader和InputDispatcher的工作过程。在InputDispatcher的过程讲到 调用InputChanel通过socket与远程进程通信，本文便展开讲解这个socket是如何建立的。

对于InputReader和InputDispatcher都是运行在system\_server进程； 用户点击的界面往往可能是某一个app，而每个app一般地都运行在自己的进程，这里就涉及到跨进程通信，app进程是如何与system进程建立通信。

要解答这些问题，从Activity最基本的创建过程开始说起。我们都知道一般地Activity对应一个应用窗口, 每一个窗口对应一个ViewRootImpl。窗口是如何添加到Activity的，从Activity.onCreate()为起点讲解。

、

## UI线程

总所周知，Activity的生命周期的回调方法都是运行在主线程，也称之为UI线程，所有UI相关的操作都需要运行在该线程。本文虽然是UI线程，但并非只介绍所有运行在UI线程的流程，文中还涉及binder thread。

### onCreate

Activity启动是由system进程控制：

handleLaunchActivity()：会调用Activity.onCreate(), 该方法内再调用setContentView(),经过AMS与WMS的各种交互,层层调用后,进入step2

handleResumeActivity()：会调用Activity.makeVisible(),该方法继续调用便会执行到WindowManagerImpl.addView(), 该方法内部再调用WindowManagerGlobal.addView(),

### addView

[-> WindowManagerGlobal.java]

public void addView(View view, ViewGroup.LayoutParams params, Display display, Window parentWindow) {

...

//[见小节2.3]

ViewRootImpl root = new ViewRootImpl(view.getContext(), display);

//[见小节2.3.3]

root.setView(view, wparams, panelParentView);

...

}

### ViewRootImpl

Asdf

[-> ViewRootImpl.java]

public ViewRootImpl(Context context, Display display) {

mContext = context;

//获取IWindowSession的代理类【见小节2.3.1】

mWindowSession = WindowManagerGlobal.getWindowSession();

mDisplay = display;

mThread = Thread.currentThread(); //主线程

mWindow = new W(this);

mChoreographer = Choreographer.getInstance();

...

}

#### getWindowSession

fas

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **public** **static** IWindowSession getWindowSession() {
2. synchronized (WindowManagerGlobal.**class**) {
3. **if** (sWindowSession == null) {
4. **try** {
5. //获取IMS的代理类
6. InputMethodManager imm = InputMethodManager.getInstance();
7. //获取WMS的代理类
8. IWindowManager windowManager = getWindowManagerService();
9. //经过Binder调用，最终调用WMS[见小节2.3.2]
10. sWindowSession = windowManager.openSession(
11. **new** IWindowSessionCallback.Stub() {...},
12. imm.getClient(), imm.getInputContext());
13. } **catch** (RemoteException e) {
14. ...
15. }
16. }
17. **return** sWindowSession
18. }
19. }

#### WMS.openSession

public IWindowSession openSession(IWindowSessionCallback callback, IInputMethodClient client, IInputContext inputContext) {

//创建Session对象

Session session = new Session(this, callback, client, inputContext);

return session;

}

再次经过Binder将数据写回app进程，则获取的便是Session的代理对象。

#### ViewRootImpl.setView

Wfd

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **public** **void** setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
2. **synchronized** (**this**) {
3. ...
4. **if** ((mWindowAttributes.inputFeatures
5. & WindowManager.LayoutParams.INPUT\_FEATURE\_NO\_INPUT\_CHANNEL) == 0) {
6. mInputChannel = **new** InputChannel(); //创建InputChannel对象
7. }
8. //通过Binder调用，进入system进程的Session[见小节2.4]
9. res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
10. getHostVisibility(), mDisplay.getDisplayId(),
11. mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
12. mAttachInfo.mOutsets, mInputChannel);
13. ...
14. **if** (mInputChannel != **null**) {
15. **if** (mInputQueueCallback != **null**) {
16. mInputQueue = **new** InputQueue();
17. mInputQueueCallback.onInputQueueCreated(mInputQueue);
18. }
19. //创建WindowInputEventReceiver对象[见3.1]
20. mInputEventReceiver = **new** WindowInputEventReceiver(mInputChannel,
21. Looper.myLooper());
22. }
23. }
24. }

该方法主要功能:

创建Java层的InputChannel对象mInputChannel

向WMS注册InputChannel信息，通过InputChannel.openInputChannelPair创建的socket pair，将其中的客户端赋值给mInputChannel.

创建WindowInputEventReceiver对象

跨进程调用，进入binder thread执行如下方法：

### Session.addToDisplay

[-> Session.java]

final class Session extends IWindowSession.Stub implements IBinder.DeathRecipient {

public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs, int viewVisibility, int displayId, Rect outContentInsets, Rect outStableInsets, Rect outOutsets, InputChannel outInputChannel) {

//[见小节2.5]

return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,

outContentInsets, outStableInsets, outOutsets, outInputChannel);

}

}

### WMS.addWindow

WindowManagerService.java]

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **public** **int** addWindow(Session session, IWindow client, **int** seq, WindowManager.LayoutParams attrs, **int** viewVisibility, **int** displayId, Rect outContentInsets, Rect outStableInsets, Rect outOutsets, InputChannel outInputChannel) {
2. ...
3. //创建WindowState【见小节2.5.1】
4. WindowState win = **new** WindowState(**this**, session, client, token,
5. attachedWindow, appOp[0], seq, attrs, viewVisibility, displayContent);
7. **if** (outInputChannel != **null** && (attrs.inputFeatures
8. & WindowManager.LayoutParams.INPUT\_FEATURE\_NO\_INPUT\_CHANNEL) == 0) {
9. //根据WindowState的HashCode以及title来生成InputChannel名称
10. String name = win.makeInputChannelName();
12. //创建一对InputChannel[见小节2.6]
13. InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
14. //将socket服务端保存到WindowState的mInputChannel
15. win.setInputChannel(inputChannels[0]);
17. //socket客户端传递给outInputChannel [见小节2.7]
18. inputChannels[1].transferTo(**outInputChannel**);
19. //利用socket服务端作为参数[见小节2.8]
20. mInputManager.registerInputChannel(win.mInputChannel, win.mInputWindowHandle);
21. }
22. ...
23. **boolean** focusChanged = **false**;
24. **if** (win.canReceiveKeys()) {
25. //新添加window能接收按下操作，则更新聚焦窗口。
26. focusChanged = updateFocusedWindowLocked(UPDATE\_FOCUS\_WILL\_ASSIGN\_LAYERS,
27. **false** /\*updateInputWindows\*/);
28. }
29. ...
31. **if** (focusChanged) {
32. mInputMonitor.setInputFocusLw(mCurrentFocus, **false** /\*updateInputWindows\*/);
33. }
34. //设置当前聚焦窗口【见小节2.5.2】
35. mInputMonitor.updateInputWindowsLw(**false** /\*force\*/);
36. }

inputChannels数组：

inputChannels[0]所对应的InputChannel名称的后缀为(server);

inputChannels[1]所对应的InputChannel名称的后缀为(client)；

其中：

服务端inputChannels[0]保存到WindowState的mInputChannel；

客户端inputChannels[1]传递给outInputChannel，最终传递给ViewRootImpl的mInputChannel；

#### WindowState初始化

[-> WindowState.java]

WindowState(WindowManagerService service, Session s, IWindow c, WindowToken token,

WindowState attachedWindow, int appOp, int seq, WindowManager.LayoutParams a,

int viewVisibility, final DisplayContent displayContent) {

...

WindowState appWin = this;

while (appWin.mAttachedWindow != null) {

appWin = appWin.mAttachedWindow;

}

WindowToken appToken = appWin.mToken;

while (appToken.appWindowToken == null) {

WindowToken parent = mService.mTokenMap.get(appToken.token);

if (parent == null || appToken == parent) {

break;

}

appToken = parent;

}

mAppToken = appToken.appWindowToken;

//创建InputWindowHandle对象

mInputWindowHandle = new InputWindowHandle(

mAppToken != null ? mAppToken.mInputApplicationHandle : null, this,

displayContent.getDisplayId());

}

#### updateInputWindowsLw

[-> InputMonitor.java]

public void updateInputWindowsLw(boolean force) {

...

final InputWindowHandle dragWindowHandle = mService.mDragState.mDragWindowHandle;

if (dragWindowHandle != null) {

//将dragWindowHandle赋值给mInputWindowHandles

addInputWindowHandleLw(dragWindowHandle);

}

...

//将当前mInputWindowHandles传递到native【】

mService.mInputManager.setInputWindows(mInputWindowHandles);

...

}

setInputWindows的调用链：(最终设置mFocusedWindowHandle值)

> IMS.setInputWindows

-> NativeInputManager::setInputWindows

-> InputDispatcher::setInputWindows

dragWindowHandle的初始化过程：

View.startDrag

Session.prepareDrag

WMS.prepareDragSurface

mDragState = new DragState(...);

Session.performDrag

DragState.register

mDragWindowHandle = new InputWindowHandle(...);

### openInputChannelPair

[-> InputChannel.java]

public static InputChannel[] openInputChannelPair(String name) {

return nativeOpenInputChannelPair(name);

}

这个过程的主要功能

创建两个socket通道(非阻塞, buffer上限32KB)

创建两个InputChannel对象;

创建两个NativeInputChannel对象;

将nativeInputChannel保存到Java层的InputChannel的成员变量mPtr

#### nativeOpenInputChannelPair

[-> android\_view\_InputChannel.cpp]

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **static** jobjectArray android\_view\_InputChannel\_nativeOpenInputChannelPair(JNIEnv\* env,
2. jclass clazz, jstring nameObj) {
3. **const** **char**\* nameChars = env->GetStringUTFChars(nameObj, NULL);
4. String8 name(nameChars);
5. env->ReleaseStringUTFChars(nameObj, nameChars);
7. sp<InputChannel> serverChannel;
8. sp<InputChannel> clientChannel;
9. //创建一对socket[见小节2.6.2]
10. status\_t result = InputChannel::openInputChannelPair(name, serverChannel, clientChannel);
12. //创建Java数组
13. jobjectArray channelPair = env->NewObjectArray(2, gInputChannelClassInfo.clazz, NULL);
14. ...
16. //创建NativeInputChannel对象[见小节2.6.3]
17. jobject serverChannelObj = android\_view\_InputChannel\_createInputChannel(env,
18. **new** NativeInputChannel(serverChannel));
19. ...
21. //创建NativeInputChannel对象[见小节2.6.3]
22. jobject clientChannelObj = android\_view\_InputChannel\_createInputChannel(env,
23. **new** NativeInputChannel(clientChannel));
24. ...
26. //将client和server 两个插入到channelPair
27. env->SetObjectArrayElement(channelPair, 0, serverChannelObj);
28. env->SetObjectArrayElement(channelPair, 1, clientChannelObj);
29. **return** channelPair;
30. }

#### InputTransport. openInputChannelPair

Es

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. status\_t InputChannel::openInputChannelPair(**const** String8& name,
2. sp<InputChannel>& outServerChannel, sp<InputChannel>& outClientChannel) {
3. **int** sockets[2];
4. //真正创建socket对的地方【核心】
5. **if** (socketpair(AF\_UNIX, SOCK\_SEQPACKET, 0, sockets)) {
6. ...
7. **return** result;
8. }
10. **int** bufferSize = SOCKET\_BUFFER\_SIZE; //32k
11. setsockopt(sockets[0], SOL\_SOCKET, SO\_SNDBUF, &bufferSize, **sizeof**(bufferSize));
12. setsockopt(sockets[0], SOL\_SOCKET, SO\_RCVBUF, &bufferSize, **sizeof**(bufferSize));
13. setsockopt(sockets[1], SOL\_SOCKET, SO\_SNDBUF, &bufferSize, **sizeof**(bufferSize));
14. setsockopt(sockets[1], SOL\_SOCKET, SO\_RCVBUF, &bufferSize, **sizeof**(bufferSize));
16. String8 serverChannelName = name;
17. serverChannelName.append(" (server)");
18. //创建InputChannel对象
19. outServerChannel = **new** InputChannel(serverChannelName, sockets[0]);
21. String8 clientChannelName = name;
22. clientChannelName.append(" (client)");
23. //创建InputChannel对象
24. outClientChannel = **new** InputChannel(clientChannelName, sockets[1]);
25. **return** OK;
26. }

该方法主要功能:

创建socket pair; (非阻塞式的socket)

设置两个socket的接收和发送的buffer上限为32KB;

创建client和server的Native层InputChannel对象;

sockets[0]所对应的InputChannel名称的后缀为(server);

sockets[1]所对应的InputChannel名称的后缀为(client)

创建InputChannel对象位于文件InputTransport.cpp，如下：

InputChannel::InputChannel(const String8& name, int fd) :

mName(name), mFd(fd) {

//将socket设置成非阻塞方式

**int result = fcntl(mFd, F\_SETFL, O\_NONBLOCK);**

}

另外，创建NativeInputChannel对象位于文件android\_view\_InputChannel.cpp，如下：

NativeInputChannel::NativeInputChannel(const sp<InputChannel>& inputChannel) :

mInputChannel(inputChannel), mDisposeCallback(NULL) {

}

#### android\_view\_InputChannel\_createInputChannel

ef

[-> android\_view\_InputChannel.cpp]

static jobject android\_view\_InputChannel\_createInputChannel(JNIEnv\* env, NativeInputChannel\* nativeInputChannel) {

//创建Java的InputChannel

jobject inputChannelObj = env->NewObject(gInputChannelClassInfo.clazz,

gInputChannelClassInfo.ctor);

if (inputChannelObj) {

//将nativeInputChannel保存到Java层的InputChannel的成员变量mPtr

android\_view\_InputChannel\_setNativeInputChannel(env, inputChannelObj, nativeInputChannel);

}

return inputChannelObj;

}

static void android\_view\_InputChannel\_setNativeInputChannel(JNIEnv\* env, jobject inputChannelObj, NativeInputChannel\* nativeInputChannel) {

env->SetLongField(inputChannelObj, gInputChannelClassInfo.mPtr,

reinterpret\_cast<jlong>(nativeInputChannel));

}

此处:

* gInputChannelClassInfo.clazz是指Java层的InputChannel类
* gInputChannelClassInfo.ctor是指Java层的InputChannel构造方法;
* gInputChannelClassInfo.mPtr是指Java层的InputChannel的成员变量mPtr;

### transferTo

[-> InputChannel.java]

public void transferTo(InputChannel outParameter) {

nativeTransferTo(outParameter);

}

#### nativeTransferTo

[-> android\_view\_InputChannel.cp

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **static** **void** android\_view\_InputChannel\_nativeTransferTo(JNIEnv\* env, jobject obj,
2. jobject otherObj) {
4. **if** (android\_view\_InputChannel\_getNativeInputChannel(env, otherObj) != NULL) {
5. **return**; //当Java层的InputChannel.mPtr不为空,则返回
6. }
8. //将当前inputChannels[1]的mPtr赋值给nativeInputChannel
9. NativeInputChannel\* nativeInputChannel =
10. android\_view\_InputChannel\_getNativeInputChannel(env, obj);
11. // 将该nativeInputChannel保存到outInputChannel的参数
12. android\_view\_InputChannel\_setNativeInputChannel(env, otherObj, nativeInputChannel);
13. android\_view\_InputChannel\_setNativeInputChannel(env, obj, NULL);
14. }

Df

inputChannels[1].transferTo(outInputChannel)主要功能:

1. 当outInputChannel.mPtr不为空,则直接返回;否则进入step2;
2. 将inputChannels[1].mPtr的值赋给outInputChannel.mPtr;
3. 清空inputChannels[1].mPtr值;

也就是将socket客户端inputChannels[1]传递给outInputChannel；

### IMS.registerInputChannel

InputManagerService.java

public void registerInputChannel(InputChannel inputChannel, InputWindowHandle inputWindowHandle) {

nativeRegisterInputChannel(mPtr, inputChannel, inputWindowHandle, false);

}

* inputChannel是指inputChannels[0],即socket服务端；
* inputWindowHandle是指WindowState.mInputWindowHandle;

#### nativeRegisterInputChannel

dg

[-> com\_android\_server\_input\_InputManagerService.cpp]

static void nativeRegisterInputChannel(JNIEnv\* env, jclass /\* clazz \*/, jlong ptr, jobject inputChannelObj, jobject inputWindowHandleObj, jboolean monitor) {

NativeInputManager\* im = reinterpret\_cast<NativeInputManager\*>(ptr);

sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,

inputChannelObj);

sp<InputWindowHandle> inputWindowHandle =

android\_server\_InputWindowHandle\_getHandle(env, inputWindowHandleObj);

//[见小节2.8.2]

status\_t status = im->registerInputChannel(

env, inputChannel, inputWindowHandle, monitor);

...

if (! monitor) {

android\_view\_InputChannel\_setDisposeCallback(env, inputChannelObj,

handleInputChannelDisposed, im);

}

}

#### registerInputChannel

[-> com\_android\_server\_input\_InputManagerService.cpp]

status\_t NativeInputManager::registerInputChannel(JNIEnv\* /\* env \*/,

const sp<InputChannel>& inputChannel,

const sp<InputWindowHandle>& inputWindowHandle, bool monitor) {

//[见小节2.8.3]

return mInputManager->getDispatcher()->registerInputChannel(

inputChannel, inputWindowHandle, monitor);

}

mInputManager是指[NativeInputManager](http://gityuan.com/2016/12/10/input-manager/)初始化过程创建的InputManager对象(C+

#### registerInputChannel

[-> InputDispatcher.cpp]

status\_t InputDispatcher::registerInputChannel(const sp<InputChannel>& inputChannel,

const sp<InputWindowHandle>& inputWindowHandle, bool monitor) {

{

AutoMutex \_l(mLock);

...

//创建Connection[见小节2.8.4]

sp<Connection> connection = new Connection(inputChannel, inputWindowHandle, monitor);

//返回连接的fd

int fd = inputChannel->getFd();

//添加连接到集合中

mConnectionsByFd.add(fd, connection);

...

//将该fd添加到Looper监听[见小节2.8.5]

mLooper->addFd(fd, 0, ALOOPER\_EVENT\_INPUT, handleReceiveCallback, this);

}

mLooper->wake(); //connection改变, 则唤醒looper

return OK;

}

将新创建的connection保存到mConnectionsByFd成员变量，“InputDispatcher”线程的Looper添加对socket服务端的监听功能； 当该socket有消息时便会唤醒该线程工作。

mLooper是Native Looper，在/system/core/libutils/Looper.cpp文件中实现，主要的作用是将输入请求添加到请求队列中。

#### 初始化Connection

Sdf

[-> InputDispatcher.cpp]

InputDispatcher::Connection::Connection(const sp<InputChannel>& inputChannel,

const sp<InputWindowHandle>& inputWindowHandle, bool monitor) :

status(STATUS\_NORMAL), inputChannel(inputChannel), inputWindowHandle(inputWindowHandle),

monitor(monitor),

inputPublisher(inputChannel), inputPublisherBlocked(false) {

}

其中InputPublisher初始化位于文件InputTransport.cpp

InputPublisher:: InputPublisher(const sp<InputChannel>& channel) :

mChannel(channel) {

}

此处inputChannel是指前面openInputChannelPair创建的socket服务端，将其同时保存到Connection.inputChannel和InputPublisher.mChannel。

#### Looper.addFd

Fasf

[-> system/core/libutils/Looper.cpp]

int Looper::addFd(int fd, int ident, int events, Looper\_callbackFunc callback, void\* data) {

// 此处的callback为handleReceiveCallback

return addFd(fd, ident, events, callback ? new SimpleLooperCallback(callback) : NULL, data);

}

int Looper::addFd(int fd, int ident, int events, const sp<LooperCallback>& callback, void\* data) {

{

AutoMutex \_l(mLock);

Request request;

request.fd = fd;

request.ident = ident;

request.events = events;

request.seq = mNextRequestSeq++;

request.callback = callback; //是指SimpleLooperCallback

request.data = data;

if (mNextRequestSeq == -1) mNextRequestSeq = 0;

struct epoll\_event eventItem;

request.initEventItem(&eventItem);

ssize\_t requestIndex = mRequests.indexOfKey(fd);

if (requestIndex < 0) {

//通过epoll监听fd

int epollResult = epoll\_ctl(mEpollFd, EPOLL\_CTL\_ADD, fd, & eventItem);

...

mRequests.add(fd, request); //该fd的request加入到mRequests队列

} else {

int epollResult = epoll\_ctl(mEpollFd, EPOLL\_CTL\_MOD, fd, & eventItem);

...

mRequests.replaceValueAt(requestIndex, request);

}

}

return 1;

}

此处Loop便是“InputDispatcher”线程的Looper，将socket服务端的fd采用epoll机制注册监听

### 小节

虽然本文介绍的UI线程的工作，

* [小节2.1 ~ 2.3]： 运行在UI线程；
* [小节2.4 ~ 2.8]：通过IWindowSession的Binder IPC调用，运行在system\_server的binder thread;

ViewRootImpl的setView()过程:

* 创建socket pair，作为InputChannel:
  + socket服务端保存到system\_server中的WindowState的mInputChannel；
  + socket客户端通过binder传回到远程进程的UI主线程ViewRootImpl的mInputChannel；
* IMS.registerInputChannel()注册InputChannel，监听socket服务端：
  + Loop便是“InputDispatcher”线程的Looper;
  + 回调方法handleReceiveCallback。

## WindowInputEventReceiver

接下来，看看【小节2.3.3】创建WindowInputEventReceiver对象

### WindowInputEventReceiver初始化

D

[-> ViewRootImpl.java]

final class WindowInputEventReceiver extends InputEventReceiver {

//inputChannel是指socket客户端，Looper是指UI线程的Looper

public WindowInputEventReceiver(InputChannel inputChannel, Looper looper) {

super(inputChannel, looper); //【见小节3.2】

}

...

}

### InputEventReceiver

Sd

[-> InputEventReceiver.java]

public InputEventReceiver(InputChannel inputChannel, Looper looper) {

...

mInputChannel = inputChannel;

mMessageQueue = looper.getQueue(); //UI线程消息队列

//【加小节3.3】

mReceiverPtr = nativeInit(new WeakReference<InputEventReceiver>(this),

inputChannel, mMessageQueue);

}

### nativeInit

[-> android\_view\_InputEventReceiver.cpp]

static jlong nativeInit(JNIEnv\* env, jclass clazz, jobject receiverWeak, jobject inputChannelObj, jobject messageQueueObj) {

sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,

inputChannelObj);

//获取UI主线程的消息队列

sp<MessageQueue> messageQueue = android\_os\_MessageQueue\_getMessageQueue(env, messageQueueObj);

//创建NativeInputEventReceiver对象【见小节3.4】

sp<NativeInputEventReceiver> receiver = new NativeInputEventReceiver(env,

receiverWeak, inputChannel, messageQueue);

//【见小节3.5】

status\_t status = receiver->initialize();

...

receiver->incStrong(gInputEventReceiverClassInfo.clazz);

return reinterpret\_cast<jlong>(receiver.get());

}

### NativeInputEventReceiver

Sdf

[-> android\_view\_InputEventReceiver.cpp]

class NativeInputEventReceiver : public LooperCallback {

InputConsumer mInputConsumer;

sp<MessageQueue> mMessageQueue;

int mFdEvents;

bool mBatchedInputEventPending;

...

NativeInputEventReceiver::NativeInputEventReceiver(JNIEnv\* env,

jobject receiverWeak, const sp<InputChannel>& inputChannel,

const sp<MessageQueue>& messageQueue) :

mReceiverWeakGlobal(env->NewGlobalRef(receiverWeak)),

//【见3.4.1】

mInputConsumer(inputChannel), mMessageQueue(messageQueue),

mBatchedInputEventPending(false), mFdEvents(0) {

}

}

#### InputConsumer

 InputTransport.cpp]

InputConsumer::InputConsumer(const sp<InputChannel>& channel) :

mResampleTouch(isTouchResamplingEnabled()),

mChannel(channel), mMsgDeferred(false) {

}

此处inputChannel是指socket客户端

### initialize

 android\_view\_InputEventReceiver.cpp]

status\_t NativeInputEventReceiver::initialize() {

setFdEvents(ALOOPER\_EVENT\_INPUT); //【见小节3.6】

return OK;

}

### setFdEvents

sdf

[-> android\_view\_InputEventReceiver.cpp]

void NativeInputEventReceiver::setFdEvents(int events) {

if (mFdEvents != events) {

mFdEvents = events;

int fd = mInputConsumer.getChannel()->getFd();

if (events) {

//将socket客户端的fd添加到主线程的消息池【见小节3.6.1】

mMessageQueue->getLooper()->addFd(fd, 0, events, this, NULL);

} else {

mMessageQueue->getLooper()->removeFd(fd);

}

}

}

此处的Looper便是UI主线程的Looper，将socket客户端的fd添加到UI线程的Looper来监听，回调方法为NativeInputEventReceiver。

## 总结



首先，通过openInputChannelPair来创建socket pair，作为InputChannel:

* socket服务端保存到system\_server中的WindowState的mInputChannel；
* socket客户端通过binder传回到远程进程的UI主线程ViewRootImpl的mInputChannel；

紧接着，完成了两个线程的epoll监听工作：

* [小节2.8]IMS.registerInputChannel(): “InputDispatcher”线程监听socket服务端，收到消息后回调InputDispatcher.handleReceiveCallback()；
* [小节3.6]setFdEvents(): UI主线程监听socket客户端，收到消息后回调NativeInputEventReceiver.handleEvent().

有了这些“InputDispatcher”和“UI”主线程便可以进行跨进程通信与交互。

# 按键事件处理过程

以按键事件为例，说一说一次事件处理过程是如何完成。按键事件经过InputReader，再到InputDispatcher的startDispatchCycleLocked()过程会调用publishKeyEvent()，从该方法说起

## InputDispatcher线程

### InputPublisher.publishKeyEvent

Dfv

[-> InputTransport.cpp]

status\_t InputPublisher::publishKeyEvent(...) {

if (!seq) {

return BAD\_VALUE;

}

InputMessage msg;

msg.header.type = InputMessage::TYPE\_KEY;

msg.body.key.seq = seq;

msg.body.key.deviceId = deviceId;

msg.body.key.source = source;

msg.body.key.action = action;

msg.body.key.flags = flags;

msg.body.key.keyCode = keyCode;

msg.body.key.scanCode = scanCode;

msg.body.key.metaState = metaState;

msg.body.key.repeatCount = repeatCount;

msg.body.key.downTime = downTime;

msg.body.key.eventTime = eventTime;

//通过InputChannel来发送消息

return mChannel->sendMessage(&msg);

}

### InputChannel.sendMessage

status\_t InputChannel::sendMessage(const InputMessage\* msg) {

size\_t msgLength = msg->size();

ssize\_t nWrite;

do {// //向目标mFd写入消息，采用异步非阻塞方式

nWrite = ::send(mFd, msg, msgLength, MSG\_DONTWAIT | MSG\_NOSIGNAL);

} while (nWrite == -1 && errno == EINTR);

Input系统—UI线程讲解了会创建socket pair，用于两个进程的线程间相互通信。当mFd写入消息后，此时会唤醒处于epoll\_wait状态的应用进程的UI线程，见下文。

另外，当写入失败，则返回值为WOULD\_BLOCK或者DEAD\_OBJECT。

## UI主线程

当收到消息的处理过程，Android消息机制在获取下一条消息的时候,会调用lnativePollOnce(),最终进入到Looper::pollInner()过程，如下：

Looper::pollInner

当收到消息的处理过程，Android消息机制在获取下一条消息的时候,会调用lnativePollOnce(),最终进入到Looper::pollInner()过程，如下：

### Looper::pollInner

int Looper::pollInner(int timeoutMillis) {

...

int result = POLL\_WAKE;

mResponses.clear();

mResponseIndex = 0;

mPolling = true; //即将处于idle状态

struct epoll\_event eventItems[EPOLL\_MAX\_EVENTS]; //fd最大个数为16

//等待事件发生或者超时，在nativeWake()方法，向管道写端写入字符;

int eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

mPolling = false; //不再处于idle状态

mLock.lock(); //请求锁

...

//循环遍历，处理所有的事件

for (int i = 0; i < eventCount; i++) {

int fd = eventItems[i].data.fd;

uint32\_t epollEvents = eventItems[i].events;

if (fd == mWakeEventFd) {

if (epollEvents & EPOLLIN) {

awoken(); //已唤醒则读取并清空管道数据

}

} else {

ssize\_t requestIndex = mRequests.indexOfKey(fd);

if (requestIndex >= 0) {

int events = 0;

if (epollEvents & EPOLLIN) events |= EVENT\_INPUT;

if (epollEvents & EPOLLOUT) events |= EVENT\_OUTPUT;

if (epollEvents & EPOLLERR) events |= EVENT\_ERROR;

if (epollEvents & EPOLLHUP) events |= EVENT\_HANGUP;

//处理request，生成对应的reponse对象，push到mResponses数组

pushResponse(events, mRequests.valueAt(requestIndex));

}

}

}

Done: ;

//再处理Native的Message，调用相应回调方法

while (mMessageEnvelopes.size() != 0) {

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

//取出消息

const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);

if (messageEnvelope.uptime <= now) {

sp<MessageHandler> handler = messageEnvelope.handler;

Message message = messageEnvelope.message;

mMessageEnvelopes.removeAt(0); //移除该消息

mLock.unlock();

handler->handleMessage(message); // 处理消息事件

}

mLock.lock();

...

}

mLock.unlock(); //释放锁

//处理带有Callback()方法的Response事件，执行Reponse相应的回调方法

for (size\_t i = 0; i < mResponses.size(); i++) {

Response& response = mResponses.editItemAt(i);

if (response.request.ident == POLL\_CALLBACK) {

int fd = response.request.fd;

int events = response.events;

void\* data = response.request.data;

// 处理请求的回调方法【见小节3.2】

int callbackResult = response.request.callback->handleEvent(fd, events, data);

if (callbackResult == 0) {

removeFd(fd, response.request.seq); //移除fd

}

response.request.callback.clear(); //清除reponse引用的回调方法

result = POLL\_CALLBACK; // 发生回调

}

}

return result;

}

此处response.request.callback是指NativeInputEventReceiver，接下来便是执行NativeInputEventReceiver.handleEvent();

### handleEvent

fsd

[-> android\_view\_InputEventReceiver.cpp]

int NativeInputEventReceiver::handleEvent(int receiveFd, int events, void\* data) {

if (events & (ALOOPER\_EVENT\_ERROR | ALOOPER\_EVENT\_HANGUP)) {

return 0; //移除窗口或者IME对话框， 则移除该事件

}

if (events & ALOOPER\_EVENT\_INPUT) {

JNIEnv\* env = AndroidRuntime::getJNIEnv();

//【见小节3.3】

status\_t status = consumeEvents(env, false /\*consumeBatches\*/, -1, NULL);

mMessageQueue->raiseAndClearException(env, "handleReceiveCallback");

return status == OK || status == NO\_MEMORY ? 1 : 0;

}

if (events & ALOOPER\_EVENT\_OUTPUT) {

for (size\_t i = 0; i < mFinishQueue.size(); i++) {

const Finish& finish = mFinishQueue.itemAt(i);

//【见小节3.4】

status\_t status = mInputConsumer.sendFinishedSignal(finish.seq, finish.handled);

if (status) {

mFinishQueue.removeItemsAt(0, i);

if (status == WOULD\_BLOCK) {

return 1; //保留callback，稍后重试

}

if (status != DEAD\_OBJECT) {

JNIEnv\* env = AndroidRuntime::getJNIEnv();

String8 message;

message.appendFormat("Failed to finish input event. status=%d", status);

jniThrowRuntimeException(env, message.string());

mMessageQueue->raiseAndClearException(env, "finishInputEvent");

}

return 0; //移除callback

}

}

mFinishQueue.clear();

setFdEvents(ALOOPER\_EVENT\_INPUT);

return 1;

}

return 1;

}

UI线程收到Key事件后，开始处理该事件。

### NativeInputEventReceiver.consumeEvents

d[-> android\_view\_InputEventReceiver.cpp]

status\_t NativeInputEventReceiver::consumeEvents(JNIEnv\* env,

bool consumeBatches, nsecs\_t frameTime, bool\* outConsumedBatch) {

...

ScopedLocalRef<jobject> receiverObj(env, NULL);

bool skipCallbacks = false;

for (;;) {

uint32\_t seq;

InputEvent\* inputEvent;

//【见小节3.3.1】

status\_t status = mInputConsumer.consume(&mInputEventFactory,

consumeBatches, frameTime, &seq, &inputEvent);

if (status) {

if (status == WOULD\_BLOCK) {

...

return OK; //消费完成

}

return status; //消失失败

}

if (!skipCallbacks) {

if (!receiverObj.get()) {

receiverObj.reset(jniGetReferent(env, mReceiverWeakGlobal));

if (!receiverObj.get()) {

return DEAD\_OBJECT;

}

}

jobject inputEventObj;

switch (inputEvent->getType()) {

case AINPUT\_EVENT\_TYPE\_KEY:

//由Native的inputEvent来生成Java层的事件

inputEventObj = android\_view\_KeyEvent\_fromNative(env,

static\_cast<KeyEvent\*>(inputEvent));

break;

...

}

if (inputEventObj) {

//执行Java层的InputEventReceiver.dispachInputEvent【见小节3.3.3】

env->CallVoidMethod(receiverObj.get(),

gInputEventReceiverClassInfo.dispatchInputEvent, seq, inputEventObj);

if (env->ExceptionCheck()) {

skipCallbacks = true; //分发过程发生异常

}

env->DeleteLocalRef(inputEventObj);

} else {

skipCallbacks = true;

}

}

if (skipCallbacks) {

//发生异常，则直接向InputDispatcher线程发送完成信号。

mInputConsumer.sendFinishedSignal(seq, false);

}

}

}

#### InputConsumer.consume

InputConsumer.consume

[->InputTransport.cpp ::InputConsumer]

status\_t InputConsumer::consume(InputEventFactoryInterface\* factory,

bool consumeBatches, nsecs\_t frameTime, uint32\_t\* outSeq, InputEvent\*\* outEvent) {

\*outSeq = 0;

\*outEvent = NULL;

//循环遍历所有的Event

while (!\*outEvent) {

if (mMsgDeferred) {

mMsgDeferred = false; //上一次没有处理的消息

} else {

//收到新消息【见小节3.3.2】

status\_t result = mChannel->receiveMessage(&mMsg);

if (result) {

if (consumeBatches || result != WOULD\_BLOCK) {

result = consumeBatch(factory, frameTime, outSeq, outEvent);

if (\*outEvent) {

break;

}

}

return result;

}

}

switch (mMsg.header.type) {

case InputMessage::TYPE\_KEY: {

//从mKeyEventPool池中取出KeyEvent

KeyEvent\* keyEvent = factory->createKeyEvent();

if (!keyEvent) return NO\_MEMORY;

//将msg封装成KeyEvent

initializeKeyEvent(keyEvent, &mMsg);

\*outSeq = mMsg.body.key.seq;

\*outEvent = keyEvent;

break;

}

...

}

}

return OK;

}

#### InputChannel.receiveMessage

Dsf

[-> InputTransport.cpp]

status\_t InputChannel::receiveMessage(InputMessage\* msg) {

ssize\_t nRead;

do {

//读取InputDispatcher发送过来的消息

nRead = ::recv(mFd, msg, sizeof(InputMessage), MSG\_DONTWAIT);

} while (nRead == -1 && errno == EINTR);

if (nRead < 0) {

int error = errno;

if (error == EAGAIN || error == EWOULDBLOCK) {

return WOULD\_BLOCK;

}

if (error == EPIPE || error == ENOTCONN || error == ECONNREFUSED) {

return DEAD\_OBJECT;

}

return -error;

}

if (nRead == 0) {

return DEAD\_OBJECT;

}

if (!msg->isValid(nRead)) {

return BAD\_VALUE;

}

return OK;

}

#### InputEventReceiver. dispatchInputEvent

Dsg

InputEventReceiver.java]

private void dispatchInputEvent(int seq, InputEvent event) {

mSeqMap.put(event.getSequenceNumber(), seq);

onInputEvent(event); //[见小节3.3.4]

}

### ViewRootImpl事件传递

#### WindowInputEventReceiver.onInputEvent

sfdg

[-> ViewRootImpl.java ::WindowInputEventReceiver]

final class WindowInputEventReceiver extends InputEventReceiver {

public void onInputEvent(InputEvent event) {

enqueueInputEvent(event, this, 0, true); //【见小节3.3.5】

}

...

}

#### enqueueInputEvent

dsf

void enqueueInputEvent(InputEvent event, InputEventReceiver receiver, int flags, boolean processImmediately) {

adjustInputEventForCompatibility(event);

QueuedInputEvent q = obtainQueuedInputEvent(event, receiver, flags);

QueuedInputEvent last = mPendingInputEventTail;

if (last == null) {

mPendingInputEventHead = q;

mPendingInputEventTail = q;

} else {

last.mNext = q;

mPendingInputEventTail = q;

}

mPendingInputEventCount += 1;

if (processImmediately) {

doProcessInputEvents(); //【见小节3.3.6】

} else {

scheduleProcessInputEvents();

}

}

#### doProcessInputEvents

as

oid doProcessInputEvents() {

while (mPendingInputEventHead != null) {

QueuedInputEvent q = mPendingInputEventHead;

mPendingInputEventHead = q.mNext;

if (mPendingInputEventHead == null) {

mPendingInputEventTail = null;

}

q.mNext = null;

mPendingInputEventCount -= 1;

long eventTime = q.mEvent.getEventTimeNano();

long oldestEventTime = eventTime;

...

mChoreographer.mFrameInfo.updateInputEventTime(eventTime, oldestEventTime);

//[见小节3.3.7]

deliverInputEvent(q);

}

if (mProcessInputEventsScheduled) {

mProcessInputEventsScheduled = false;

mHandler.removeMessages(MSG\_PROCESS\_INPUT\_EVENTS);

}

}

#### 事件分发deliverInputEvent

private void deliverInputEvent(QueuedInputEvent q) {

if (mInputEventConsistencyVerifier != null) {

mInputEventConsistencyVerifier.onInputEvent(q.mEvent, 0);

}

InputStage stage;

if (q.shouldSendToSynthesizer()) {

stage = mSyntheticInputStage;

} else {

stage = q.shouldSkipIme() ? mFirstPostImeInputStage : mFirstInputStage;

}

if (stage != null) {

stage.deliver(q);

} else {

finishInputEvent(q); //[见小节3.4]

}

}

经过一系列的InputStage调用, 最终会分发到真正需要处理该事件的窗口. 当处理完后会调用finishInputEvent(), 见小节3.4

### finishInputEvent事件完成

#### finishInputEvent

iewRootImpl.java]

private void finishInputEvent(QueuedInputEvent q) {

if (q.mReceiver != null) {

boolean handled = (q.mFlags & QueuedInputEvent.FLAG\_FINISHED\_HANDLED) != 0;

//[见小节3.4.1]

q.mReceiver.finishInputEvent(q.mEvent, handled);

} else {

q.mEvent.recycleIfNeededAfterDispatch();

}

recycleQueuedInputEvent(q);

}

#### mReceiver.finishInputEvent

sd

public final void finishInputEvent(InputEvent event, boolean handled) {

if (mReceiverPtr == 0) {

...

} else {

int index = mSeqMap.indexOfKey(event.getSequenceNumber());

if (index < 0) {

...

} else {

int seq = mSeqMap.valueAt(index);

mSeqMap.removeAt(index);

//经过层层调用,见[小节3.5]

nativeFinishInputEvent(mReceiverPtr, seq, handled);

}

}

event.recycleIfNeededAfterDispatch();

#### nativeFinishInputEvent

static void nativeFinishInputEvent(JNIEnv\* env, jclass clazz, jlong receiverPtr,

jint seq, jboolean handled) {

sp<NativeInputEventReceiver> receiver =

reinterpret\_cast<NativeInputEventReceiver\*>(receiverPtr);

status\_t status = receiver->finishInputEvent(seq, handled);

if (status && status != DEAD\_OBJECT) {

String8 message;

message.appendFormat("Failed to finish input event. status=%d", status);

jniThrowRuntimeException(env, message.string());

}

}

#### NativeInputEventReceiver::finishInputEvent

status\_t NativeInputEventReceiver::finishInputEvent(uint32\_t seq, bool handled) {

if (kDebugDispatchCycle) {

ALOGD("channel '%s' ~ Finished input event.", getInputChannelName());

}

status\_t status = mInputConsumer.sendFinishedSignal(seq, handled);

if (status) {

if (status == WOULD\_BLOCK) {

if (kDebugDispatchCycle) {

ALOGD("channel '%s' ~ Could not send finished signal immediately. "

"Enqueued for later.", getInputChannelName());

}

Finish finish;

finish.seq = seq;

finish.handled = handled;

mFinishQueue.add(finish);

if (mFinishQueue.size() == 1) {

setFdEvents(ALOOPER\_EVENT\_INPUT | ALOOPER\_EVENT\_OUTPUT);

}

return OK;

}

ALOGW("Failed to send finished signal on channel '%s'. status=%d",

getInputChannelName(), status);

}

return status;

}

### InputTransport.cpp ::InputConsumer

#### sendFinishedSignal

status\_t InputConsumer::sendFinishedSignal(uint32\_t seq, bool handled) {

...

size\_t seqChainCount = mSeqChains.size();

if (seqChainCount) {

uint32\_t currentSeq = seq;

uint32\_t chainSeqs[seqChainCount];

size\_t chainIndex = 0;

for (size\_t i = seqChainCount; i-- > 0; ) {

const SeqChain& seqChain = mSeqChains.itemAt(i);

if (seqChain.seq == currentSeq) {

currentSeq = seqChain.chain;

chainSeqs[chainIndex++] = currentSeq;

mSeqChains.removeAt(i);

}

}

status\_t status = OK;

while (!status && chainIndex-- > 0) {

//[见小节3.5.1]

status = sendUnchainedFinishedSignal(chainSeqs[chainIndex], handled);

}

if (status) {

// An error occurred so at least one signal was not sent, reconstruct the chain.

do {

SeqChain seqChain;

seqChain.seq = chainIndex != 0 ? chainSeqs[chainIndex - 1] : seq;

seqChain.chain = chainSeqs[chainIndex];

mSeqChains.push(seqChain);

} while (chainIndex-- > 0);

return status;

}

}

return sendUnchainedFinishedSignal(seq, handled);

}

#### sendUnchainedFinishedSignal

InputTransport.cpp ::InputConsumer]

status\_t InputConsumer::sendUnchainedFinishedSignal(uint32\_t seq, bool handled) {

InputMessage msg;

msg.header.type = InputMessage::TYPE\_FINISHED;

msg.body.finished.seq = seq;

msg.body.finished.handled = handled;

return mChannel->sendMessage(&msg);

}

通过InputChannel->sendMessage，将TYPE\_FINISHED类型的消息，发送回InputDispatcher线程

## InputDispatcher线程

### Looper::pollInner

Df

int Looper::pollInner(int timeoutMillis) {

int eventCount = epoll\_wait(mEpollFd, eventItems, EPOLL\_MAX\_EVENTS, timeoutMillis);

...

Done:

...

for (size\_t i = 0; i < mResponses.size(); i++) {

Response& response = mResponses.editItemAt(i);

if (response.request.ident == POLL\_CALLBACK) {

int fd = response.request.fd;

int events = response.events;

void\* data = response.request.data;

// 处理请求的回调方法【见小节4.2】

int callbackResult = response.request.callback->handleEvent(fd, events, data);

...

}

}

return result;

}

此处response.request.callback是指SimpleLooperCallback。接下来调用SimpleLooperCallback.handleEvent(). 执行后的返回值callbackResult=0则移除该fd，否则稍后重新尝试。

### handleEvent

asdf

[-> Looper.cpp ::SimpleLooperCallback]

SimpleLooperCallback::SimpleLooperCallback(Looper\_callbackFunc callback) :

mCallback(callback) {

}

int SimpleLooperCallback::handleEvent(int fd, int events, void\* data) {

//handleReceiveCallback()【见小节4.3】

return mCallback(fd, events, data);

}

IMS.registerInputChannel()过程，会调用Looper.addFd()完成的赋值操作，mCallback等于handleReceiveCallback()方法

### handleReceiveCallback

dsf当开始处理请求队列中的输入事件请求时，会回调该输入事件注册的回调函数handleReceiveCallback

opy

[-> InputDispatcher]

int InputDispatcher::handleReceiveCallback(int fd, int events, void\* data) {

InputDispatcher\* d = static\_cast<InputDispatcher\*>(data);

{

AutoMutex \_l(d->mLock);

ssize\_t connectionIndex = d->mConnectionsByFd.indexOfKey(fd);

bool notify;

sp<Connection> connection = d->mConnectionsByFd.valueAt(connectionIndex);

**// /死循环，等待输入事件finish信号**

if (!(events & (ALOOPER\_EVENT\_ERROR | ALOOPER\_EVENT\_HANGUP))) {

...

nsecs\_t currentTime = now();

bool gotOne = false;

status\_t status;

for (;;) {

uint32\_t seq;

bool handled;

//【见小节4.4】

// //等待输入事件finish信号 ----------

status = connection->inputPublisher.receiveFinishedSignal(&seq, &handled);

if (status) {

break;

}

//【见小节4.5】

// //结束当前输入事件分发过程 ----------------

d->finishDispatchCycleLocked(currentTime, connection, seq, handled);

gotOne = true;

}

if (gotOne) {

d->runCommandsLockedInterruptible(); //执行命令【见小节4.6】

if (status == WOULD\_BLOCK) {

return 1;

}

}

notify = status != DEAD\_OBJECT || !connection->monitor;

} else {

...

//input channel被关闭或者发生错误

}

//取消注册channel

d->unregisterInputChannelLocked(connection->inputChannel, notify);

return 0;

}

}

　当开始处理输入事件后，会循环等待输入事件finish信号。如果接收到输入事件的finish信号后，则结束当前事件的分发过程。如果在5s内，还未收到输入事件的finish信号，则会调用InputDispatcher的handleTargetsNotReadyLocked()方法，发送ANR通知。 ---------------------

opy

### InputPublisher.receiveFinishedSignal

E

[-> InputTransport.cpp]

status\_t InputPublisher::receiveFinishedSignal(uint32\_t\* outSeq, bool\* outHandled) {

InputMessage msg;

//接收消息

status\_t result = mChannel->receiveMessage(&msg);

if (result) {

\*outSeq = 0;

\*outHandled = false;

return result;

}

if (msg.header.type != InputMessage::TYPE\_FINISHED) {

return UNKNOWN\_ERROR; //发生错误

}

\*outSeq = msg.body.finished.seq;

\*outHandled = msg.body.finished.handled;

return OK;

}

### finishDispatchCycleLocked

sad

InputDispatcher.cpp]

void InputDispatcher::finishDispatchCycleLocked(nsecs\_t currentTime,

const sp<Connection>& connection, uint32\_t seq, bool handled) {

connection->inputPublisherBlocked = false;

if (connection->status == Connection::STATUS\_BROKEN

|| connection->status == Connection::STATUS\_ZOMBIE) {

return;

}

//通知系统准备启动下一次分发流程【见小节4.5.1】

onDispatchCycleFinishedLocked(currentTime, connection, seq, handled);

}

### onDispatchCycleFinishedLocked

asd

[-> InputDispatcher.cpp]

void InputDispatcher::onDispatchCycleFinishedLocked(

nsecs\_t currentTime, const sp<Connection>& connection, uint32\_t seq, bool handled) {

**//向mCommandQueue添加命令**

CommandEntry\* commandEntry = postCommandLocked(

& InputDispatcher::doDispatchCycleFinishedLockedInterruptible);

commandEntry->connection = connection;

commandEntry->eventTime = currentTime;

commandEntry->seq = seq;

commandEntry->handled = handled;

}

### runCommandsLockedInterruptible

dfg

[-> InputDispatcher.cpp]

bool InputDispatcher::runCommandsLockedInterruptible() {

if (mCommandQueue.isEmpty()) {

return false;

}

do {

//从mCommandQueue队列的头部取出第一个元素【见小节4.6.1】

CommandEntry\* commandEntry = mCommandQueue.dequeueAtHead();

Command command = commandEntry->command;

//此处调用的命令隐式地包含'LockedInterruptible'

(this->\*command)(commandEntry);

commandEntry->connection.clear();

delete commandEntry;

} while (! mCommandQueue.isEmpty());

return true;

}

由【小节4.5】，可以队列中的元素至少有doDispatchCycleFinishedLockedInterruptible。

### doDispatchCycleFinishedLockedInterruptible

sd

[-> InputDispatcher.cpp]

void InputDispatcher::doDispatchCycleFinishedLockedInterruptible(

CommandEntry\* commandEntry) {

sp<Connection> connection = commandEntry->connection;

nsecs\_t finishTime = commandEntry->eventTime;

uint32\_t seq = commandEntry->seq;

bool handled = commandEntry->handled;

//获取分发事件

DispatchEntry\* dispatchEntry = connection->findWaitQueueEntry(seq);

if (dispatchEntry) {

nsecs\_t eventDuration = finishTime - dispatchEntry->deliveryTime;

//打印出所有分发时间超过2s的事件

if (eventDuration > SLOW\_EVENT\_PROCESSING\_WARNING\_TIMEOUT) {

String8 msg;

msg.appendFormat("Window '%s' spent %0.1fms processing the last input event: ",

connection->getWindowName(), eventDuration \* 0.000001f);

dispatchEntry->eventEntry->appendDescription(msg);

ALOGI("%s", msg.string());

}

bool restartEvent;

if (dispatchEntry->eventEntry->type == EventEntry::TYPE\_KEY) {

KeyEntry\* keyEntry = static\_cast<KeyEntry\*>(dispatchEntry->eventEntry);

restartEvent = afterKeyEventLockedInterruptible(connection,

dispatchEntry, keyEntry, handled);

} else if (dispatchEntry->eventEntry->type == EventEntry::TYPE\_MOTION) {

...

} else {

...

}

if (dispatchEntry == connection->findWaitQueueEntry(seq)) {

//将dispatchEntry事件从等待队列(waitQueue)中移除

connection->waitQueue.dequeue(dispatchEntry);

if (restartEvent && connection->status == Connection::STATUS\_NORMAL) {

connection->outboundQueue.enqueueAtHead(dispatchEntry);

} else {

releaseDispatchEntryLocked(dispatchEntry);

}

}

//启动下一个事件处理循环。

startDispatchCycleLocked(now(), connection);

}

}

该方法主要功能：

打印出所有分发时间超过2s的事件；

将dispatchEntry事件从等待队列(waitQueue)中移除；

启动下一个事件处理循环

## 小结

### 整体框架图



### 交互过程

用一张图来总结交互过程，主要是通过一对socket方式来通信。 当input时间分发到app端, 那么便进入来了InputEventReceiver.dispatchInputEvent()过程.



图解:

InputDispatcher线程调用InputPublisher的publishKeyEvent向UI主线程发送input事件;

UI主线程接收到该事件后,调用InputConsumer的consumeEvents来处理该事件, 一路执行到ViewRootImpl.deliverInputEvent()方法;

UI主线程经过一系列的InputStage来处理, 当事件分发完成,则会执行finishInputEvent()方法.再进一步调用InputConsumer::sendFinishedSignal 告知InputDispatcher线程该时事件已处理完成.

InputDispatcher线程收到该事件后, 执行InputDispatcher::handleReceiveCallback();最终会调用doDispatchCycleFinishedLockedInterruptible()方法 ,将dispatchEntry事件从等待队列(waitQueue)中移除.

# 焦点移动原理

焦点：焦点（Focus）可以理解为选中态，在Android TV上起很重要的作用。一个视图控件只有在获得焦点的状态下，才能响应按键的Click事件。





在使用Android TV的过程中，遥控器是一个主流的操作工具，通过点击遥控器的方向键来控制焦点的移动。当焦点移动到目标控件上之后，按下遥控器的确定键，才会触发一个Click事件，进而去做下一步的处理

有个大家族，已经经历多代，族人角色可以这样定义：

* 成员：View
* 有子女的成员：ViewGroup
* 辈分最高的长老：DecorView

家族中有一件宝贝，持有在一名成员手上。别的家族想参观，首先需要找长老。

长老不会一个个成员问，而是先找大儿子问，再找二儿子问，如此类推。儿子们也是这样问自己的儿子，过程也是如此类推。一层层地问，直到最后找到宝贝的持有人，再一层层向上通知。

宝贝就是焦点，寻找宝贝的过程就是焦点分发的过程。

## 基础的用法

在处理焦点的时候，有一些基础的用法需要知道。

### 使能焦点

首先，isFocusable()需要为true，一个控件才有资格可以获取到焦点。

Java:可以通过setFocusable(boolean)方法来设置。如果想要在触摸模式下获取焦点（在我们用手机开发的过程中），需要isFocusableInTouchMode()为true，可以通过setFocusableInTouchMode(boolean)来设置

在xml：android:focusable="true"。android:focusableInTouchMode="true"布局文件中指定同理。

### 控制焦点的移动

然后，就是控制焦点的移动了。在谷歌官方文档中提到：

焦点移动的时候（默认的情况下），会按照一种算法去找在指定移动方向上最近的邻居。在一些情况下，焦点的移动可能跟开发者的意图不符，这时开发者化也有两种方法：

1.在布局文件中使用下面这些XML属性来指定下一个焦点对象：

nextFocusDown，nextFocusLeft。nextFocusRight。nextFocusUp

2.在Java代码中，让一个指定的View获取焦点，可调用它的requestFocus()方法

### ****View的hasFocus()方法和isFocused()方法对比****

**Stackoverflow解释来了：**

**hasFocus() is different from isFocused(). hasFocus() == true means that the View or one of its descendants is focused. If you look closely, there's a chain of hasFocused Views till you reach the View that isFocused.**

### 遇到的问题

尽管有了官方文档中提到的基础用法，但是在进行Android TV开发的过程中，还是经常会遇到一些焦点方面的问题或者疑问，如

“明明指定了焦点id，焦点却跑丢了”

“onKeyDown里居然截获不到按键事件”

“我没有做任何焦点处理，焦点是怎么自己跑到那个View上的”

requestFocus和背后的焦点分发机制；

clearFocus真的无效吗？

如果让焦点按意志移动。

接下来，带着这些问题，我们就从源码的角度出发，简单分析一下焦点的移动原理

## KeyEvent分发

当按下遥控器的按键时，会产生一个按键事件，就是KeyEvent，包含“上”，“下”，“左”，“右”，“返回”，“确定”等指令。焦点的处理就在KeyEvent的分发当中完成。

首先，KeyEvent会流转到ViewRootImpl中开始进行处理，具体方法是内部类ViewPostImeInputStage中的processKeyEvent。

PFLAG\_HAS\_BOUNDS，表示当前View具有了明确的边界范围

### processKeyEvent

**final class** ViewPostImeInputStage **extends** InputStage

private int processKeyEvent(QueuedInputEvent q) {

final KeyEvent event = (KeyEvent)q.mEvent;

// Deliver the key to the view hierarchy.

// 1. 先去执行mView的dispatchKeyEvent

if (mView.dispatchKeyEvent(event)) {

return FINISH\_HANDLED;

}

// Handle automatic focus changes.

if (event.getAction() == KeyEvent.ACTION\_DOWN) {

int direction = 0;

if (direction != 0) {

View focused = mView.findFocus();

if (focused != null) {

// 2. 之后会通过focusSearch去找下一个焦点视图

View v = focused.focusSearch(direction);

if (v != null && v != focused) {

...

if (v.requestFocus(direction, mTempRect)) {

...

return FINISH\_HANDLED;

}

}

// Give the focused view a last chance to handle the dpad key.

if (mView.dispatchUnhandledMove(focused, direction)) {

return FINISH\_HANDLED;

}

} else {

// find the best view to give focus to in this non-touch-mode with no-focus

// 3. 如果当前本来就没有焦点视图，也会通过focusSearch找一个视图

View v = focusSearch(null, direction);

if (v != null && v.requestFocus(direction)) {

return FINISH\_HANDLED;

}

}

}

}

return FORWARD;

}

从几处关键的代码，可以看到这里的逻辑是：

先去执行mView的dispatchKeyEvent

之后会通过focusSearch去找下一个焦点视图

如果当前本来就没有焦点View，也会通过focusSearch找一个视图

ViewRootImpl就是ViewRoot，继承了ViewParent，但本身并不是一个View，可以看作是View树的管理者。而这里的成员变量mView就是DecorView，它指向的对象跟Window和Activity的mDecor指向的对象是同一个对象。所有的View组成了一个View树，每一个View都是树中的一个节点，如下图所示：



最上层的根是DecorView，中间是各ViewGroup，最下层是View。

本文的分析都是基于View树的。

在processKeyEvent中，首先走了mView的dispatchKeyEvent，也就是从DecorView开始进行KeyEvent的分发。

### dispatchKeyEvent

首先走DecorView的dispatchKeyEvent，之后会依次从Activity->ViewGroup->View的方向分发KeyEvent。

有兴趣的话可以通过trace看一下KeyEvent的流转方向：



对于KeyEvent的分发，之后会另开一篇细讲，包括KeyEvent的处理优先级，长按的识别等，这里只简单看一下ViewGroup和View的dispatchKeyEvent。

#### ViewGroup.dispatchKeyEvent

@Override

public boolean dispatchKeyEvent(KeyEvent event) {

...

if ((**mPrivateFlags & (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS))**

**== (PFLAG\_FOCUSED | PFLAG\_HAS\_BOUNDS))** {

// 1.1 以View的身份处理KeyEvent

if (super.dispatchKeyEvent(event)) {

return true;

}

} else if (mFocused != null && (mFocused.mPrivateFlags & PFLAG\_HAS\_BOUNDS)

== PFLAG\_HAS\_BOUNDS) {

// 1.2 以ViewGroup的身份把KeyEvent交给mFocused处理

if (mFocused.dispatchKeyEvent(event)) {

return true;

}

}

...

return false;

}

通过flag的判断，有两个处理路径，也可以看到在处理keyEvent时，ViewGroup扮演两个角色：

View的角色，也就是此时keyEvent需要在自己与其他View之间流转

ViewGroup的角色，此时keyEvent需要在自己的子View之间流转

当作View的时候，会调用自己View的dispatchKeyEvent。

当作ViewGroup的时候，会调用当前焦点View的dispatchKeyEvent。

其实，从概念上来看，都是调用当前有焦点View的dispatchKeyEvent，只不过有时是自己本身，有时是他的子View。

#### View.dispatchKeyEvent

public boolean dispatchKeyEvent(KeyEvent event) {

...

ListenerInfo li = mListenerInfo;

// 1.3 如果设置了mOnKeyListener，则优先走onKey方法

if (li != null && li.mOnKeyListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED

&& li.mOnKeyListener.onKey(this, event.getKeyCode(), event)) {

return true;

}

// 1.4 把View自己当作参数传入，调用KeyEvent的dispatch方法

if (event.dispatch(this, mAttachInfo != null

? mAttachInfo.mKeyDispatchState : null, this)) {

return true;

}

...

return false;

}

View这里，会优先处理OnKeyListener的onKey回调。

然后才可能会走KeyEvent的dispatch，最终走到View的OnKeyDown或者OnKeyUp。

将大体的流转顺序总结如下图



其中任何一步都可以通过return true的方式来消费掉这个KeyEvent，结束这个分发过程。

##### KeyEvent.dispatch

dispatch(receiver, **null**, **null**

**boolean** res = receiver.onKeyDown(**mKeyCode**, **this**);

ListView. onKeyDown

@Override  
**public boolean** onKeyDown(**int** keyCode, KeyEvent event) {  
 **return** commonKey(keyCode, 1, event);  
}

commonKey(**int** keyCode, **int** count, KeyEvent event)

**case** KeyEvent.***KEYCODE\_DPAD\_DOWN***:  
 **if** (event.hasNoModifiers()) {  
 handled = resurrectSelectionIfNeeded();  
 **if** (!handled) {  
 **while** (count-- > 0) {  
 **if** (arrowScroll(***FOCUS\_DOWN***)) {  
 handled = **true**;  
 } **else** {  
 **break**;  
 }  
 }  
 }  
 }

###### EXLV.arrowScroll(int direction)

**final boolean** handled = arrowScrollImpl(direction);

###### EXLV .arrowScrollImpl(int direction)

**if** (nextSelectedPosition != ***INVALID\_POSITION***) {  
 handleNewSelectionChange(selectedView, direction, nextSelectedPosition, focusResult != **null**);

// handleNewSelectionChange(View selectedView, **int** direction, **int** newSelectedPosition,  
 **boolean** newFocusAssigned)

###### EXLV .handleNewSelectionChange(View selectedView, int direction, int newSelectedPosition, boolean newFocusAssigned)

**if** (bottomView != **null**) {  
 bottomView.setSelected(!newFocusAssigned && !topSelected);  
 measureAndAdjustDown(bottomView, bottomViewIndex, numChildren);  
 }

**bottomView==MyGridView**

###### View.setSelected

**public void** setSelected(**boolean** selected) {  
 *//noinspection DoubleNegation* **if** (((**mPrivateFlags** & ***PFLAG\_SELECTED***) != 0) != selected) {  
 **mPrivateFlags** = (**mPrivateFlags** & ~***PFLAG\_SELECTED***) | (selected ? ***PFLAG\_SELECTED*** : 0);  
 **if** (!selected) resetPressedState();  
 invalidate(**true**);  
 refreshDrawableState();  
 dispatchSetSelected(selected);

###### GridView.dispatchSetSelected(GridView)

@Override  
**public void** dispatchSetSelected(**boolean** selected) {  
 **final** View[] children = **mChildren**;  
 **final int** count = **mChildrenCount**;  
 **for** (**int** i = 0; i < count; i++) {  
 children[i].setSelected(selected);  
 }  
}

问题：1.选项重来都获取不到焦点呢，是因为父控件没有了焦点的原因么？

### focusSearch

如果dispatchKeyEvent没有消费掉这个KeyEvent，会由系统来处理焦点的移动。

通过View的focusSearch方法找到下一个获取焦点的View，然后调用requestFocus

focusSearch其实是一层一层地网上调用**父View的focusSearch**方法，直到当前**view是根布局**（isRootNamespace()方法），通过注释可以知道focusSearch最终会调用**DecorView的focusSearch**方法。而DecorView的focusSearch方法找到的焦点view是通过**FocusFinder**来找到的。

那focusSearch是如何找到下一个焦点视图的呢？

// View.java

public View focusSearch(@FocusRealDirection int direction) {

if (mParent != null) {

return mParent.focusSearch(this, direction);

} else {

return null;

}

}

View并不会直接去找，而是交给它的parent去找。

// ViewGroup.java

public View focusSearch(View focused, int direction) {

if (isRootNamespace()) {

// root namespace means we should consider ourselves the top of the

// tree for focus searching; otherwise we could be focus searching

// into other tabs. see LocalActivityManager and TabHost for more info

return FocusFinder.getInstance().findNextFocus(this, focused, direction);

} else if (mParent != null) {

return mParent.focusSearch(focused, direction);

}

return null;

}

判断是否为顶层布局，若是则执行对应方法，若不是则继续向上寻找，说明会从内到外的一层层进行判断，直到最外层的布局为止。

有意思的是，Android提供了设置isRootNamespace的方法，但又hide了起来不让使用，看来这个逻辑还有待优化。

public void setIsRootNamespace(boolean isRoot) {

if (isRoot) {

mPrivateFlags |= PFLAG\_IS\_ROOT\_NAMESPACE;

} else {

mPrivateFlags &= ~PFLAG\_IS\_ROOT\_NAMESPACE;

}

}

最后的算法交给了FocusFinder

FocusFinder.getInstance().findNextFocus(this, focused, direction);

isRootNamespace()的ViewGroup把自己和当前焦点（View）以及方向传入。

public final View findNextFocus(ViewGroup root, View focused, int direction) {

return findNextFocus(root, focused, null, direction);

}

private View findNextFocus(ViewGroup root, View focused, Rect focusedRect, int direction) {

View next = null;

if (focused != null) {

// 2.1 优先从xml或者代码中指定focusid的View中找

next = findNextUserSpecifiedFocus(root, focused, direction);

}

if (next != null) {

return next;

}

ArrayList<View> focusables = mTempList;

try {

focusables.clear();

root.addFocusables(focusables, direction);

if (!focusables.isEmpty()) {

// 2.2 其次，根据算法去找，原理就是找在方向上最近的View

next = findNextFocus(root, focused, focusedRect, direction, focusables);

}

} finally {

focusables.clear();

}

return next;

}

这里root是上面isRootNamespace()为true的ViewGroup，focused是当前焦点视图

优先找开发者指定的下一个focus的视图 ，就是在xml或者代码中指定NextFocusDirection Id的视图

其次，根据算法去找，原理就是找在方向上最近的视图

#### findNextUserSpecifiedFocus

g

// FocusFinder.java

private View findNextUserSpecifiedFocus(ViewGroup root, View focused, int direction) {

// check for user specified next focus

View userSetNextFocus = focused.findUserSetNextFocus(root, direction);

if (userSetNextFocus != null && userSetNextFocus.isFocusable()

&& (!userSetNextFocus.isInTouchMode()

|| userSetNextFocus.isFocusableInTouchMode())) {

return userSetNextFocus;

}

return null;

}

##### View.findUserSetNextFocus方法

F

// View.java

View findUserSetNextFocus(View root, @FocusDirection int direction) {

switch (direction) {

case FOCUS\_LEFT:

if (mNextFocusLeftId == View.NO\_ID) return null;

return findViewInsideOutShouldExist(root, mNextFocusLeftId);

...

}

}

return null;

}

比如，按了“左”方向键，如果设置了mNextFocusLeftId，则会通过findViewInsideOutShouldExist去找这个View。

mNextFocusLeftId一般是在xml里面设置的，比如android:nextFocusLeft="@+id/btn\_2"，

mBtn1.setNextFocusLeftId(R.id.btn\_2);

##### findViewInsideOutShouldExist

private View findViewInsideOutShouldExist(View root, int id) {

if (mMatchIdPredicate == null) {

// 可以理解为一个判定器，如果id匹配则判定成功

mMatchIdPredicate = new MatchIdPredicate();

}

mMatchIdPredicate.mId = id;

View result = root.findViewByPredicateInsideOut(this, mMatchIdPredicate);

...

return result;

}

public final View findViewByPredicateInsideOut(View start, Predicate<View> predicate) {

View childToSkip = null;

for (;;) {

// 从当前起始节点开始寻找（ViewGroup是遍历自己的child），寻找id匹配的View，跳过childToSkip，具体可去看View和ViewGroup中该方法的具体实现

View view = start.findViewByPredicateTraversal(predicate, childToSkip);

if (view != null || start == this) {

return view;

}

ViewParent parent = start.getParent();

if (parent == null || !(parent instanceof View)) {

return null;

}

// 如果如果当前节点没有，则往上一级，从自己的parent中查找，并跳过自己

childToSkip = start;

start = (View) parent;

}

}

protected View findViewByPredicateTraversal(Predicate<View> predicate, View childToSkip) {

if (predicate.apply(this)) {

return this;

}

return null;

}

ViewGroup的findViewByPredicateTraversal

// ViewGroup

@Override

protected View findViewByPredicateTraversal(Predicate<View> predicate, View childToSkip) {

if (predicate.apply(this)) {

return this;

}

final View[] where = mChildren;

final int len = mChildrenCount;

for (int i = 0; i < len; i++) {

View v = where[i];

if (v != childToSkip && (v.mPrivateFlags & PFLAG\_IS\_ROOT\_NAMESPACE) == 0) {

v = v.findViewByPredicate(predicate);

if (v != null) {

return v;

}

}

}

return null;

}

可以看到，findViewInsideOutShouldExist这个方法从当前指定视图去寻找指定id的视图。首先从自己开始向下遍历，如果没找到则从自己的parent开始向下遍历，直到找到id匹配的视图为止。

这里要注意的是，也许存在多个相同id的视图（比如ListView，RecyclerView，ViewPager等场景），但是这个方法只会返回在View树中节点范围最近的一个视图，这就是为什么有时候看似指定了focusId，但实际上焦点却丢失的原因，因为焦点跑到了另一个“意想不到”的相同id的视图上。

#### findNextFocus

如果开发者没有指定nextFocusId，则用findNextFocus找指定方向上最近的视图

看一下这里的用法

focusables.clear();

// 2.2.1 找到所有isFocusable的View

**root.addFocusables(focusables, direction);**

if (!focusables.isEmpty()) {

// 2.2.2 从focusables中找到最近的一个

next = **findNextFocus(root, focused, focusedRect, direction, focusables);**

}

##### View.addFocusables

从root开始找所有isFocusable的视图

如果root是一个单纯View，则添加自己，但这种情况很少见，大部分的root都是ViewGroup

public void addFocusables(ArrayList<View> views, @FocusDirection int direction) {

addFocusables(views, direction, FOCUSABLES\_TOUCH\_MODE);

}

**public void** addFocusables(ArrayList<View> views, @FocusDirection **int** direction,  
 @FocusableMode **int** focusableMode) {  
 **if** (views == **null**) {  
 **return**;  
 }  
 **if** (!isFocusable()) {  
 **return**;  
 }  
 **if** ((focusableMode & ***FOCUSABLES\_TOUCH\_MODE***) == ***FOCUSABLES\_TOUCH\_MODE*** && !isFocusableInTouchMode()) {  
 **return**;  
 }  
 views.add(**this**);  
}

##### ViewGroup. addFocusables

/ ViewGroup.java

public void addFocusables(ArrayList<View> views, int direction, int focusableMode) {

final int focusableCount = views.size();

final int descendantFocusability = getDescendantFocusability();

if (descendantFocusability != FOCUS\_BLOCK\_DESCENDANTS) {

...

final int count = mChildrenCount;

final View[] children = mChildren;

for (int i = 0; i < count; i++) {

final View child = children[i];

if ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE) {

child.addFocusables(views, direction, focusableMode);

}

}

}

if ((descendantFocusability != FOCUS\_AFTER\_DESCENDANTS

// No focusable descendants

|| (focusableCount == views.size())) &&

(isFocusableInTouchMode() || !shouldBlockFocusForTouchscreen())) {

super.addFocusables(views, direction, focusableMode);

}

}

对于ViewGroup来说，遍历并添加自己的所有isFocusable的child

这里有个descendantFocusability变量，有三个取值

FOCUS\_BEFORE\_DESCENDANTS：在所有子视图之前获取焦点

FOCUS\_AFTER\_DESCENDANTS： 在所有子视图之后获取焦点

FOCUS\_BLOCK\_DESCENDANTS： 阻止所有子视图获取焦点，即使他们是focusable的

##### FocusFinder.findNextFocus

private View findNextFocus(ViewGroup root, View focused, Rect focusedRect,

int direction, ArrayList<View> focusables) {

if (focused != null) {

if (focusedRect == null) {

focusedRect = mFocusedRect;

}

// 2.2.2.1 取得考虑scroll之后的焦点Rect，该Rect是相对focused视图本身的

// fill in interesting rect from focused

focused.getFocusedRect(focusedRect);

// 2.2.2.2 将当前focused视图的坐标系，转换到root的坐标系中，统一坐标，以便进行下一步的计算

root.offsetDescendantRectToMyCoords(focused, focusedRect);

} else {

...

}

switch (direction) {

...

case View.FOCUS\_UP:

case View.FOCUS\_DOWN:

case View.FOCUS\_LEFT:

case View.FOCUS\_RIGHT:

2.2.2.3 找出指定方向上的下一个focus视图

return findNextFocusInAbsoluteDirection(focusables, root, focused,

focusedRect, direction);

default:

throw new IllegalArgumentException("Unknown direction: " + direction);

}

}

<https://cloud.tencent.com/developer/article/1006297>

打断电吧。。。

在统一坐标之后，对于所有focusable的视图，进行一次遍历比较，得到最“近”的视图作为下一个焦点视图。这里用到了一个方法isBetterCandidate，从两个候选Rect中找到在指定方向上离当前Rect最近的一个，具体算法这里不细讲了。

至此，就找到了下一个焦点视图，然后调用requestFocus方法，让其获得焦点。

## requestFocus

FOCUS\_BEFORE\_DESCENDANTS   0x20000  默认

ViewGroup本身先对焦点进行处理，如果没有处理则分发给child View进行处理

FOCUS\_AFTER\_DESCENDANTS     0x40000

先分发给Child View进行处理，如果所有的Child View都没有处理，则自己再处理  
FOCUS\_BLOCK\_DESCENDANTS     0x60000

ViewGroup本身进行处理，不管是否处理成功，都不会分发给ChildView进行处理

简单的描述下为什么FOCUS\_BEFORE\_DESCENDANTS FOCUS\_AFTER\_DESCENDANTS FOCUS\_BLOCK\_DESCENDANTS 三种的处理方式是怎么来的，为什么会被抢焦点

ViewGroup 分析 三种焦点拦截方式与 onRequestFocusInDescendants 函数（子view处理函数）

ViewGroup 中:

ViewGroup 本身先对焦点进行处理指的就是：**super.requestFocus(direction, previouslyFocusedRect);**

分发给child view 处理指的就是：onRequestFocusInDescendants() 函数里的处理过程，是这个函数决定具体分发给哪个childView处理，或者直接拦截不处理。

看源码 requestFocus() 函数 可以看出

**FOCUS\_BLOCK\_DESCENDANTS FOCUS\_BEFORE\_DESCENDANTS FOCUS\_AFTER\_DESCENDANTS 这三个标签的区别 和对**

### View.requestFocus

让View获取焦点，直接调用requestFocus，最终会调用到requestFocusNoSearch:

private boolean requestFocusNoSearch(int direction, Rect previouslyFocusedRect) {

// need to be focusable

if ((mViewFlags & FOCUSABLE\_MASK) != FOCUSABLE ||

(mViewFlags & VISIBILITY\_MASK) != VISIBLE) {

return false;

}

// need to be focusable in touch mode if in touch mode

if (isInTouchMode() &&

(FOCUSABLE\_IN\_TOUCH\_MODE != (mViewFlags & FOCUSABLE\_IN\_TOUCH\_MODE))) {

return false;

}

// need to not have any parents blocking us

if (hasAncestorThatBlocksDescendantFocus()) {

return false;

}

handleFocusGainInternal(direction, previouslyFocusedRect);

return true;

}

requestFocusNoSearch校验View的属性，获取焦点的前提条件是“可见的”和“可聚焦的”，并且“可聚焦的”需要同时符合：

android:focusable="true"

android:focusableInTouchMode="true"

接着调用了hasAncestorThatBlocksDescendantFocus

#### hasAncestorThatBlocksDescendantFocus

private boolean hasAncestorThatBlocksDescendantFocus() {

final boolean focusableInTouchMode = isFocusableInTouchMode();

ViewParent ancestor = mParent;

while (ancestor instanceof ViewGroup) {

final ViewGroup vgAncestor = (ViewGroup) ancestor;

if (vgAncestor.getDescendantFocusability() == ViewGroup.FOCUS\_BLOCK\_DESCENDANTS

|| (!focusableInTouchMode && vgAncestor.shouldBlockFocusForTouchscreen())) {

return true;

} else {

ancestor = vgAncestor.getParent();

}

}

return false;

}

hasAncestorThatBlocksDescendantFocus就很好理解，如果有祖先ViewGroup设置成blocksDescendants，那么它的子孙View都不能获取焦点。

#### handleFocusGainInternal

void handleFocusGainInternal(@FocusRealDirection int direction, Rect previouslyFocusedRect) {

if (DBG) {

System.out.println(this + " requestFocus()");

}

if ((mPrivateFlags & PFLAG\_FOCUSED) == 0) {

mPrivateFlags |= PFLAG\_FOCUSED;

View oldFocus = (mAttachInfo != null) ? getRootView().findFocus() : null;

if (mParent != null) {

mParent.requestChildFocus(this, this);

}

if (mAttachInfo != null) {

mAttachInfo.mTreeObserver.dispatchOnGlobalFocusChange(oldFocus, this);

}

onFocusChanged(true, direction, previouslyFocusedRect);

refreshDrawableState();

}

}

handleFocusGainInternal实现View获取焦点的具体逻辑，所以requestFocusNoSearch默认返回true。handleFocusGainInternal里面最重要的是调用了mParent.requestChildFocus，通知它的父view处理焦点。mParent的类型是ViewParent，每一个view都会保存它的父view，基本上实现类就是ViewGroup。

然后触发onFocusChanged这个listener，最后触发invalidate进行ui更新。

在继续探究requestChildFocus的代码前

#### ViewGroup对焦点的处理

看回handleFocusGainInternal里的requestChildFocus，view如果需要获取焦点，需要通知它的父view处理，所以我们来看ViewGroup的requestChildFocus：

@Override

public void requestChildFocus(View child, View focused) {

if (DBG) {

System.out.println(this + " requestChildFocus()");

}

if (getDescendantFocusability() == FOCUS\_BLOCK\_DESCENDANTS) {

return;

}

// Unfocus us, if necessary

super.unFocus(focused);

// We had a previous notion of who had focus. Clear it.

if (mFocused != child) {

if (mFocused != null) {

mFocused.unFocus(focused);

}

mFocused = child;

}

if (mParent != null) {

mParent.requestChildFocus(this, focused);

}

}

首先会调用unFocus清除自己的焦点，mFocused表示ViewGroup内部是否持有焦点，如果mFocused不是目标获取焦点的child，那么再清除当前mFocused的焦点，并将child赋给mFocused。

最后继续通过mParent递归调用requestChildFocus，直到顶层view，保证焦点唯一。

简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

作者：展翅而飞  
链接：https://www.jianshu.com/p/be7e84832632  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

### ViewGrop.requestFocus

iewGroup也可以获取焦点，和上面View的requestFocus方法不同

View是否能获取焦点

让View获取焦点，直接调用requestFocus，最终会调用到requestFocusNoSearch:

@Override

**public boolean** requestFocus(**int** direction, Rect previouslyFocusedRect) {

**if** (**DBG**) {

System.**out**.println(**this** + **" ViewGroup.requestFocus direction="**

+ direction);

}

**int** descendantFocusability = getDescendantFocusability();

**switch** (descendantFocusability) {

**case FOCUS\_BLOCK\_DESCENDANTS**:

//ViewGroup本身进行处理，不管是否处理成功，都不会分发给ChildView进行处理

**return super**.requestFocus(direction, previouslyFocusedRect);

**case FOCUS\_BEFORE\_DESCENDANTS**: {

//ViewGroup本身先对焦点进行处理，如果没有处理(返回false)则分发给child View进行处理

**final boolean** took = **super**.requestFocus(direction, previouslyFocusedRect);

**return** took ? took : onRequestFocusInDescendants(direction, previouslyFocusedRect);

}

**case FOCUS\_AFTER\_DESCENDANTS**: {

//先分发给Child View进行处理，如果所有的Child View都没有处理(false)，则自己再处理

//**final boolean** took = onRequestFocusInDescendants(direction, previouslyFocusedRect);

**return** took ? took : **super**.requestFocus(direction, previouslyFocusedRect);

}

**default**:

**throw new** IllegalStateException(**"descendant focusability must be "**

+ **"one of FOCUS\_BEFORE\_DESCENDANTS, FOCUS\_AFTER\_DESCENDANTS, FOCUS\_BLOCK\_DESCENDANTS "**

+ **"but is "** + descendantFocusability);

}

}

### onRequestFocusInDescendants() 的调用顺序

sdf

protected boolean onRequestFocusInDescendants(int direction,

Rect previouslyFocusedRect) {

int index;

int increment;

int end;

int count = mChildrenCount;

if ((direction & FOCUS\_FORWARD) != 0) {

index = 0;

increment = 1;

end = count;

} else {

index = count - 1;

increment = -1;

end = -1;

}

final View[] children = mChildren;

for (int i = index; i != end; i += increment) {

View child = children[i];

if ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE) {

if (child.requestFocus(direction, previouslyFocusedRect)) {

return true;

}

}

}

return false;

}

viewGroup的默认配置

initViewGroup() -> setDescendantFocusability(**FOCUS\_BEFORE\_DESCENDANTS**);//ViewGroup本身先对焦点进行处理，如果没有处理则分发给child View进行处理。

### 清除焦点

上面没有讲view失去焦点的处理，现在来看下ViewGroup的unFocus，还要探究一下clearFocus“无效”的背后原理。

@Override

void unFocus(View focused) {

if (DBG) {

System.out.println(this + " unFocus()");

}

if (mFocused == null) {

super.unFocus(focused);

} else {

mFocused.unFocus(focused);

mFocused = null;

}

}

ViewGroup的unFocus，最终调用了View的unFocus。

void unFocus(View focused) {

if (DBG) {

System.out.println(this + " unFocus()");

}

clearFocusInternal(focused, false, false);

}

void clearFocusInternal(View focused, boolean propagate, boolean refocus) {

if ((mPrivateFlags & PFLAG\_FOCUSED) != 0) {

mPrivateFlags &= ~PFLAG\_FOCUSED;

if (propagate && mParent != null) {

mParent.clearChildFocus(this);

}

onFocusChanged(false, 0, null);

refreshDrawableState();

if (propagate && (!refocus || !rootViewRequestFocus())) {

notifyGlobalFocusCleared(this);

}

}

}

clearFocusInternal是真正操作焦点失去的地方，通过mParent调用ViewGroup的clearChildFocus。

@Override

public void clearChildFocus(View child) {

if (DBG) {

System.out.println(this + " clearChildFocus()");

}

mFocused = null;

if (mParent != null) {

mParent.clearChildFocus(this);

}

}

clearChildFocus将当前mFocused置空，通过递归向上处理直到顶层view，保证整颗view树失去焦点。

注意，unFocus我们并不能调用，View提供clearFocus，内部同样调用clearFocusInternal，它们不同的地方是refocus传入不同。

boolean rootViewRequestFocus() {

final View root = getRootView();

return root != null && root.requestFocus();

}

refocus的不同，决定是否会触发rootViewRequestFocus，因此clearFocus“无效”的问题很好理解。如果一个页面只有一个EditText，使用clearFocus清除焦点，马上地，焦点又被设置上啦，所以会有清除无效的错觉。因此，让父view自动获取焦点是很好的解决方法。

作者：展翅而飞  
链接：https://www.jianshu.com/p/be7e84832632  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

作者：展翅而飞  
链接：https://www.jianshu.com/p/be7e84832632  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

### listView 的分析

ListView 的源码中没有看到对viewGroup 设置 setDescendantFocusability 采用默认的值 则自己没有处理 逐个分发给子view处理，那么

## 安卓TV中Gridview或listview获取不到焦点

Sdf

如：listview或gridview这些常用的类型主控件，但是实际用遥控器操作时，往往子item得不到焦点的请求，这时检查会发现我们已经设置了焦点或者焦点框。可是问题出在哪里呢，刚开始入手这个的时候，我是直接网上找资料，发现一个共性问题，就是总是每个控件都会设置

android:focusable="true"和

android:focusableInTouchMode="true"

认为这样，这个控件就可以得到焦点了，但是对于istview或gridview来说，要子item得到焦点，则必须要主控件不能得到焦点，如：

<https://blog.csdn.net/aijiameng/article/details/78818784>

不对啊

复写dispatchKeyEvent？

https://blog.csdn.net/coder\_hcy/article/details/79964937

## 小结

Df

经过对源码的分析，系统本身寻找下一个焦点视图的过程是：

首先寻找用户指定了id的视图，从当前焦点视图的节点开始遍历，直到找到匹配该id的视图。也许存在多个相同id的视图，但是只会找到视图节点树中最近的一个。

如果没有指定id，则遍历找出所有isFocusable的视图，统一坐标系，然后计算出指定方向上离当前焦点视图最近的一个视图。

结合KeyEvent事件的流转，处理焦点的时机，按照优先级（顺序）依次是：

dispatchKeyEvent

mOnKeyListener.onKey回调

onKeyDown/onKeyUp

focusSearch

指定nextFocusId

系统自动从所有isFocusable的视图中找下一个焦点视图

以上任一处都可以指定焦点，一旦使用了就不再往下走。

很多视图控件就重写了其中一些方法。

比如ScrollView，它会在dispatchKeyEvent的时候，自己去处理，用来进行内部的焦点移动或者整体滑动。

// ScrollView.java

@Override

public boolean dispatchKeyEvent(KeyEvent event) {

// Let the focused view and/or our descendants get the key first

return super.dispatchKeyEvent(event) || executeKeyEvent(event);

}

public boolean executeKeyEvent(KeyEvent event) {

mTempRect.setEmpty();

if (!canScroll()) {

if (isFocused() && event.getKeyCode() != KeyEvent.KEYCODE\_BACK) {

View currentFocused = findFocus();

if (currentFocused == this) currentFocused = null;

View nextFocused = FocusFinder.getInstance().findNextFocus(this,

currentFocused, View.FOCUS\_DOWN);

// 如果不能滑动，则直接让下一个Focus视图获取焦点

return nextFocused != null

&& nextFocused != this

&& nextFocused.requestFocus(View.FOCUS\_DOWN);

}

return false;

}

boolean handled = false;

// 如果可以滑动，则进行ScrollView本身的滑动

if (event.getAction() == KeyEvent.ACTION\_DOWN) {

switch (event.getKeyCode()) {

case KeyEvent.KEYCODE\_DPAD\_UP:

if (!event.isAltPressed()) {

handled = arrowScroll(View.FOCUS\_UP);

} else {

handled = fullScroll(View.FOCUS\_UP);

}

break;

case KeyEvent.KEYCODE\_DPAD\_DOWN:

if (!event.isAltPressed()) {

handled = arrowScroll(View.FOCUS\_DOWN);

} else {

handled = fullScroll(View.FOCUS\_DOWN);

}

break;

case KeyEvent.KEYCODE\_SPACE:

pageScroll(event.isShiftPressed() ? View.FOCUS\_UP : View.FOCUS\_DOWN);

break;

}

}

return handled;

}

由于在dispatchKeyEvent里优先处理的，因此对于滑动方向的KeyEvent，onKeyDown就监听不到了。这也就是为什么onKeyDown里居然截获不到按键事件的原因。

## Ref

[从源码出发浅析 Android TV 的焦点移动原理 （上篇）](https://cloud.tencent.com/developer/article/1006289)

[从源码出发浅析 Android TV 的焦点移动原理（下篇](https://cloud.tencent.com/developer/article/1006297)

[Android焦点分发和移动的原理](https://www.jianshu.com/p/be7e84832632)

[listview用键盘获取焦点，选中上次失去焦点时的item，而不是就近的item](https://blog.csdn.net/cgq1030699125/article/details/54343693)

## 实力分析

### 首个view会自动获取焦点

D

如果Activity里有EditText，那么打开Activity后，EditText会自动获取焦点。

为什么呢，很多时候我们不想要这个效果，参照网上的方法将father layout设置成获取焦点就解决问题。知其然知其所以然，翻了一下代码，答案隐藏在ViewRootImpl.performTraversals方法中，就是那个view绘制的核心方法，中间有一段：

private void performTraversals() {

//...

if (mFirst) {

// handle first focus request

if (DEBUG\_INPUT\_RESIZE) Log.v(mTag, "First: mView.hasFocus()="

+ mView.hasFocus());

if (mView != null) {

if (!mView.hasFocus()) {

mView.requestFocus(View.FOCUS\_FORWARD);

if (DEBUG\_INPUT\_RESIZE) Log.v(mTag, "First: requested focused view="

+ mView.findFocus());

} else {

if (DEBUG\_INPUT\_RESIZE) Log.v(mTag, "First: existing focused view="

+ mView.findFocus());

}

}

}

//...

}

当是第一个view时，会调用requestFocus获取焦点。

### android tv列表焦点记忆实现

sdf

在Android tv中的开发中,经常要跟焦点打交道,一个常见的需求是要有焦点记忆功能,焦点移动到列表中的某一项中,焦点移出去,在回来时焦点还要定位到原来的项目上,对于这种需求,常见的实现方式是列表用用listview或者recyclerview实现,维护一个变量去存储上次的焦点位置,并在焦点变动或者按键事件中去维护这个变量和使用这个变量来定位.  
具体实现  
比如说用recyclerview实现列表时,在每个itemview的按键事件中,根据按键的方向和当前view的位置,判断是否是向外移走焦点的,如果是,那么就去设置父view也就是recyclerview的焦点模式,并且要保存当前view的位置信息

recyclerview.setDescendantFocusability(ViewGroup.FOCUS\_BLOCK\_DESCENDANTS);

在用户操作移回焦点时,由于前一步设置了焦点模式为block,因此会父view会首先获取焦点,增加焦点变动的事件处理:

recyclerview.setOnFocusChangeListener(new OnFocusChangeListener() {

@Override

public void onFocusChange(View v, boolean hasFocus) {

if(hasFocus){

Recyclerview recyclerview = (Recyclerview)v;

recyclerView.setDescendantFocusability(ViewGroup.FOCUS\_AFTER\_DESCENDANTS);

if(mCurrentFocusPosition>0){

LayoutManager layoutManager = recyclerView.getLayoutManager();

View viewByPosition = layoutManager.findViewByPosition(mCurrentFocusPosition);

if(viewByPosition!=null){

viewByPosition.requestFocus();

}

}

}

}

});

如代码所示,在焦点变动处理中,设置view的焦点模式,可以让子view获取焦点;拿到上一步保存的位置信息来手动设置焦点.  
后面介绍另一种更简便的方法,有更好的封装性,不需要外层逻辑做维护处理.  
文章已经发布，请查看：[RecyclerView的焦点记忆封装](https://www.jianshu.com/p/75465a7fdd0f)

LIstVIew嵌套GrideView/LineaeLayout 末端无法获取焦点

List需要有焦点，Gv不要，末端需要，怎么处理呢

# Input ANR原理分析

当input事件处理得慢就会触发ANR，那ANR内部原理是什么，哪些场景会产生ANR呢。 “工欲善其事必先利其器”，为了理解input ANR原理，前面几篇文章疏通了整个input框架的处理流程，都是为了这篇文章而做铺垫。在正式开始分析ANR触发原理以及触发场景之前，先来回顾一下input流程。

## InputReader

InputReader的主要工作分两部分：



调用EventHub的getEvents()读取节点/dev/input的input\_event结构体转换成RawEvent结构体，RawEvent根据不同InputMapper来转换成相应的EventEntry,比如按键事件则对应KeyEntry,触摸事件则对应MotionEntry。

转换结果：inut\_event -> EventEntry;

将事件添加到mInboundQueue队列尾部，加入该队列前有以下两个过滤：

IMS.interceptKeyBeforeQueueing：事件分发前可增加业务逻辑；

IMS.filterInputEvent：可拦截事件，当返回值为false的事件都直接拦截，没有机会加入mInboundQueue队列，不会再往下分发；否则进入下一步;

enqueueInboundEventLocked：该事件放入mInboundQueue队列尾部;

mLooper->wake:并根据情况来唤醒InputDispatcher线程.

KeyboardInputMapper.processKey()的过程, 记录下按下down事件的时间点.

## InputDispatcher

D



当input事件处理得慢就会触发ANR，那ANR内部原理是什么，哪些场景会产生ANR呢。 “工欲善其事必先利其器”，为了理解input ANR原理，前面几篇文章疏通了整个input框架的处理流程，都是为了这篇文章而做铺垫。

dispatchOnceInnerLocked(): 从InputDispatcher的mInboundQueue队列，取出事件EventEntry。另外该方法开始执行的时间点(currentTime)便是后续事件dispatchEntry的分发时间(deliveryTime）

dispatchKeyLocked()：满足一定条件时会添加命令doInterceptKeyBeforeDispatchingLockedInterruptible；

enqueueDispatchEntryLocked()：生成事件DispatchEntry并加入connection的outbound队列

startDispatchCycleLocked()：从outboundQueue中取出事件DispatchEntry, 重新放入connection的waitQueue队列；

runCommandsLockedInterruptible()：通过循环遍历地方式，依次处理mCommandQueue队列中的所有命令。而mCommandQueue队列中的命令是通过postCommandLocked()方式向该队列添加的。ANR回调命令便是在这个时机执行。

handleTargetsNotReadyLocked(): 该过程会判断是否等待超过5s来决定是否调用onANRLocked().

流程15中sendMessage是将input事件分发到app端,当app处理完该事件后会发送finishInputEvent()事件. 接下来又回到pollOnce()方法.

## UI Thread

Ds



input\_ui

“InputDispatcher”线程监听socket服务端，收到消息后回调InputDispatcher.handleReceiveCallback()；

UI主线程监听socket客户端，收到消息后回调NativeInputEventReceiver.handleEvent().

对于ANR的触发主要是在InputDispatcher过程，下面再从ANR的角度来说一说ANR触发过程。

## 触发原理

ANR时间区别便是指当前这次的事件dispatch过程中执行findFocusedWindowTargetsLocked()方法到下一次执行resetANRTimeoutsLocked()的时间区间. 以下5个时机会reset. 都位于InputDispatcher.cpp文件:

* resetAndDropEverythingLocked
* releasePendingEventLocked
* setFocusedApplication
* dispatchOnceInnerLocked
* setInputDispatchMode

简单来说, 主要是以下4个场景,会有机会执行resetANRTimeoutsLocked:

* 解冻屏幕, 系统开/关机的时刻点 (thawInputDispatchingLw, setEventDispatchingLw)
* wms聚焦app的改变 (WMS.setFocusedApp, WMS.removeAppToken)
* 设置input filter的过程 (IMS.setInputFilter)
* 再次分发事件的过程(dispatchOnceInnerLocked)

当InputDispatcher线程 findFocusedWindowTargetsLocked()/findTouchedWindowTargetsLocked-> handleTargetsNotReadyLocked(满足超时5s)-》onANRLocked().

## 上报流程

### onANRLocked

void InputDispatcher::onANRLocked(

nsecs\_t currentTime, const sp<InputApplicationHandle>& applicationHandle,

const sp<InputWindowHandle>& windowHandle,

nsecs\_t eventTime, nsecs\_t waitStartTime, const char\* reason) {

float dispatchLatency = (currentTime - eventTime) \* 0.000001f;

float waitDuration = (currentTime - waitStartTime) \* 0.000001f;

ALOGI("Application is not responding: %s. "

"It has been %0.1fms since event, %0.1fms since wait started. Reason: %s",

getApplicationWindowLabelLocked(applicationHandle, windowHandle).string(),

dispatchLatency, waitDuration, reason);

//捕获ANR的现场信息

time\_t t = time(NULL);

struct tm tm;

localtime\_r(&t, &tm);

char timestr[64];

strftime(timestr, sizeof(timestr), "%F %T", &tm);

mLastANRState.clear();

mLastANRState.append(INDENT "ANR:\n");

mLastANRState.appendFormat(INDENT2 "Time: %s\n", timestr);

mLastANRState.appendFormat(INDENT2 "Window: %s\n",

getApplicationWindowLabelLocked(applicationHandle, windowHandle).string());

mLastANRState.appendFormat(INDENT2 "DispatchLatency: %0.1fms\n", dispatchLatency);

mLastANRState.appendFormat(INDENT2 "WaitDuration: %0.1fms\n", waitDuration);

mLastANRState.appendFormat(INDENT2 "Reason: %s\n", reason);

dumpDispatchStateLocked(mLastANRState);

//将ANR命令加入mCommandQueue

CommandEntry\* commandEntry = postCommandLocked(

& InputDispatcher::doNotifyANRLockedInterruptible);

commandEntry->inputApplicationHandle = applicationHandle;

commandEntry->inputWindowHandle = windowHandle;

commandEntry->reason = reason;

}

发生ANR调用onANRLocked()的过程会将doNotifyANRLockedInterruptible加入mCommandQueue。 在下一轮InputDispatcher.dispatchOnce的过程中会先执行runCommandsLockedInterruptible()方法，取出 mCommandQueue队列的所有命令逐一执行。那么ANR所对应的命令doNotifyANRLockedInterruptible，接下来看该方法

### Native.doNotifyANRLockedInterruptible

Cs

[-> InputDispatcher.cpp]

void InputDispatcher::doNotifyANRLockedInterruptible(

CommandEntry\* commandEntry) {

mLock.unlock();

//[见小节3.3]

nsecs\_t newTimeout = mPolicy->notifyANR(

commandEntry->inputApplicationHandle, commandEntry->inputWindowHandle,

commandEntry->reason);

mLock.lock();

//newTimeout =5s [见小节3.8]

resumeAfterTargetsNotReadyTimeoutLocked(newTimeout,

commandEntry->inputWindowHandle != NULL

? commandEntry->inputWindowHandle->getInputChannel() : NULL);

}

mPolicy是指NativeInputManager

### notifyANR路由

NativeInputManager.

[-> com\_android\_server\_input\_InputManagerService.cpp]

nsecs\_t NativeInputManager::notifyANR(const sp<InputApplicationHandle>& inputApplicationHandle,

const sp<InputWindowHandle>& inputWindowHandle, const String8& reason) {

JNIEnv\* env = jniEnv();

jobject inputApplicationHandleObj =

getInputApplicationHandleObjLocalRef(env, inputApplicationHandle);

jobject inputWindowHandleObj =

getInputWindowHandleObjLocalRef(env, inputWindowHandle);

jstring reasonObj = env->NewStringUTF(reason.string());

//调用Java方法[见小节3.4]

jlong newTimeout = env->CallLongMethod(mServiceObj,

gServiceClassInfo.notifyANR, inputApplicationHandleObj, inputWindowHandleObj,

reasonObj);

if (checkAndClearExceptionFromCallback(env, "notifyANR")) {

newTimeout = 0; //抛出异常,则清理并重置timeout

}

...

return newTimeout;

}

Jni动态注册

可知gServiceClassInfo.notifyANR是指IMS.notifyANR

先看看register\_android\_server\_InputManager过程：

int register\_android\_server\_InputManager(JNIEnv\* env) {

int res = jniRegisterNativeMethods(env, "com/android/server/input/InputManagerService",

gInputManagerMethods, NELEM(gInputManagerMethods));

jclass clazz;

FIND\_CLASS(clazz, "com/android/server/input/InputManagerService");

...

GET\_METHOD\_ID(gServiceClassInfo.notifyANR, clazz,

"notifyANR",

"(Lcom/android/server/input/InputApplicationHandle;Lcom/android/server/input/InputWindowHandle;Ljava/lang/String;)J");

...

}

可知gServiceClassInfo.notifyANR是指IMS.notifyANR

IMS.notifyANR

mWindowManagerCallbacks.notifyANR== InputMonitor

### InputMonitor.notifyANR

[-> InputMonitor.java]

public long notifyANR(InputApplicationHandle inputApplicationHandle, InputWindowHandle inputWindowHandle, String reason) {

AppWindowToken appWindowToken = null;

WindowState windowState = null;

boolean aboveSystem = false;

synchronized (mService.mWindowMap) {

if (inputWindowHandle != null) {

windowState = (WindowState) inputWindowHandle.windowState;

if (windowState != null) {

appWindowToken = windowState.mAppToken;

}

}

if (appWindowToken == null && inputApplicationHandle != null) {

appWindowToken = (AppWindowToken)inputApplicationHandle.appWindowToken;

}

//输出input事件分发超时log

if (windowState != null) {

Slog.i(WindowManagerService.TAG, "Input event dispatching timed out "

+ "sending to " + windowState.mAttrs.getTitle()

+ ". Reason: " + reason);

int systemAlertLayer = mService.mPolicy.windowTypeToLayerLw(

WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT);

aboveSystem = windowState.mBaseLayer > systemAlertLayer;

} else if (appWindowToken != null) {

Slog.i(WindowManagerService.TAG, "Input event dispatching timed out "

+ "sending to application " + appWindowToken.stringName

+ ". Reason: " + reason);

} else {

Slog.i(WindowManagerService.TAG, "Input event dispatching timed out "

+ ". Reason: " + reason);

}

mService.saveANRStateLocked(appWindowToken, windowState, reason);

}

if (appWindowToken != null && appWindowToken.appToken != null) {

//【见小节3.6.1】

boolean abort = appWindowToken.appToken.keyDispatchingTimedOut(reason);

if (! abort) {

return appWindowToken.inputDispatchingTimeoutNanos; //5s

}

} else if (windowState != null) {

//【见小节3.6.2】

long timeout = ActivityManagerNative.getDefault().inputDispatchingTimedOut(

windowState.mSession.mPid, aboveSystem, reason);

if (timeout >= 0) {

return timeout \* 1000000L; //5s

}

}

return 0;

}

发生input相关的ANR时在system log输出ANR信息,并且tag为WindowManager. 主要有3类log:

* Input event dispatching timed out sending to [windowState.mAttrs.getTitle()]
* Input event dispatching timed out sending to application [appWindowToken.stringName)]
* Input event dispatching timed out sending.

### DispatchingTimedOut

#### Token.keyDispatchingTimedOut

[-> ActivityRecord.java :: Token]

final class ActivityRecord {

static class Token extends IApplicationToken.Stub {

public boolean keyDispatchingTimedOut(String reason) {

ActivityRecord r;

ActivityRecord anrActivity;

ProcessRecord anrApp;

synchronized (mService) {

r = tokenToActivityRecordLocked(this);

if (r == null) {

return false;

}

anrActivity = r.getWaitingHistoryRecordLocked();

anrApp = r != null ? r.app : null;

}

//[见小节3.7]

return mService.inputDispatchingTimedOut(anrApp, anrActivity, r, false, reason);

}

...

}

}

#### AMS.inputDispatchingTimedOut

inputDispatching的超时为KEY\_DISPATCHING\_TIMEOUT

1. public long inputDispatchingTimedOut(int pid, final boolean aboveSystem, String reason) {

2. ...

3. ProcessRecord proc;

4. long timeout;

5. synchronized (this) {

6. synchronized (mPidsSelfLocked) {

7. proc = mPidsSelfLocked.get(pid); //根据pid查看进程record

8. }

9. timeout = getInputDispatchingTimeoutLocked(proc);

10. }

11. //【见小节3.7】

12. if (!inputDispatchingTimedOut(proc, null, null, aboveSystem, reason)) {

13. return -1;

14. }

15.

16. return timeout;

}

f

### AMS.inputDispatchingTimedOut

Fdg

public boolean inputDispatchingTimedOut(final ProcessRecord proc, final ActivityRecord activity, final ActivityRecord parent, final boolean aboveSystem, String reason) {

...

final String annotation;

if (reason == null) {

annotation = "Input dispatching timed out";

} else {

annotation = "Input dispatching timed out (" + reason + ")";

}

if (proc != null) {

...

//通过handler机制，交由“ActivityManager”线程执行ANR处理过程。

mHandler.post(new Runnable() {

public void run() {

appNotResponding(proc, activity, parent, aboveSystem, annotation);

}

});

}

return true;

}

[appNotResponding](http://gityuan.com/2016/12/02/app-not-response/)会输出现场的重要进程的trace等信息。 再回到【小节3.2】处理完ANR后再调用resumeAfterTargetsNotReadyTimeoutLocked。

### resumeAfterTargetsNotReadyTimeoutLocked

df

void InputDispatcher::resumeAfterTargetsNotReadyTimeoutLocked(nsecs\_t newTimeout,

const sp<InputChannel>& inputChannel) {

if (newTimeout > 0) {

//超时时间增加5s

mInputTargetWaitTimeoutTime = now() + newTimeout;

} else {

// Give up.

mInputTargetWaitTimeoutExpired = true;

// Input state will not be realistic. Mark it out of sync.

if (inputChannel.get()) {

ssize\_t connectionIndex = getConnectionIndexLocked(inputChannel);

if (connectionIndex >= 0) {

sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);

sp<InputWindowHandle> windowHandle = connection->inputWindowHandle;

if (windowHandle != NULL) {

const InputWindowInfo\* info = windowHandle->getInfo();

if (info) {

ssize\_t stateIndex = mTouchStatesByDisplay.indexOfKey(info->displayId);

if (stateIndex >= 0) {

mTouchStatesByDisplay.editValueAt(stateIndex).removeWindow(

windowHandle);

}

}

}

if (connection->status == Connection::STATUS\_NORMAL) {

CancelationOptions options(CancelationOptions::CANCEL\_ALL\_EVENTS,

"application not responding");

synthesizeCancelationEventsForConnectionLocked(connection, options);

}

}

}

}

}

## input死锁监测机制

<http://gityuan.com/2017/01/01/input-anr/>

看门狗原理

## 总结

### ANR分类

**if** (windowState != **null**) {  
 Slog.i(TAG\_WM, **"Input event dispatching timed out "** + **"sending to "** + windowState.mAttrs.getTitle()  
 + **". Reason: "** + reason);  
 *// Figure out whether this window is layered above system windows.  
 // We need to do this here to help the activity manager know how to  
 // layer its ANR dialog.* **int** systemAlertLayer = mService.mPolicy.windowTypeToLayerLw(  
 WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT);  
 aboveSystem = windowState.mBaseLayer > systemAlertLayer;  
} **else if** (appWindowToken != **null**) {  
 Slog.i(TAG\_WM, **"Input event dispatching timed out "** + **"sending to application "** + appWindowToken.stringName  
 + **". Reason: "** + reason);  
} **else** {  
 Slog.i(TAG\_WM, **"Input event dispatching timed out "** + **". Reason: "** + reason);  
}

由小节[3.5] InputMonitor.notifyANR完成, 当发生ANR时system log中会出现以下信息, 并且TAG=WindowManager:

Input event dispatching timed out xxx. Reason: + reason, 其中xxx取值:

#### Title

* 窗口类型: sending to windowState.mAttrs.getTitle()
* 应用类型: sending to application appWindowToken.stringName
* 其他类型: 则为空.

#### reason

由小节[2.3.1]checkWindowReadyForMoreInputLocked完成，

ANR reason主要有以下几类：

1. **无窗口, 有应用**：Waiting because no window has focus but there is a focused application that may eventually add a window when it finishes starting up.
2. **窗口暂停**: Waiting because the [targetType] window is paused.
3. **窗口未连接**: Waiting because the [targetType] window’s input channel is not registered with the input dispatcher. The window may be in the process of being removed.
4. **窗口连接已死亡**：Waiting because the [targetType] window’s input connection is [Connection.Status]. The window may be in the process of being removed.
5. **窗口连接已满**：Waiting because the [targetType] window’s input channel is full. Outbound queue length: [outboundQueue长度]. Wait queue length: [waitQueue长度].
6. **按键事件，输出队列或事件等待队列不为空**：Waiting to send key event because the [targetType] window has not finished processing all of the input events that were previously delivered to it. Outbound queue length: [outboundQueue长度]. Wait queue length: [waitQueue长度].
7. **非按键事件，事件等待队列不为空且头事件分发超时500ms**：Waiting to send non-key event because the [targetType] window has not finished processing certain input events that were delivered to it over 500ms ago. Wait queue length: [waitQueue长度]. Wait queue head age: [等待时长].

其中

* targetType: 取值为”focused”或者”touched”
* Connection.Status: 取值为”NORMAL”，”BROKEN”，”ZOMBIE”

另外, findFocusedWindowTargetsLocked, findTouchedWindowTargetsLocked这两个方法中可以通过实现 updateDispatchStatisticsLocked()来分析anr问题

### drop事件分类

由小节[2.1.2] dropInboundEventLocked完成，输出事件丢弃的原因：

1. DROP\_REASON\_POLICY: “inbound event was dropped because the policy consumed it”;
2. DROP\_REASON\_DISABLED: “inbound event was dropped because input dispatch is disabled”;
3. DROP\_REASON\_APP\_SWITCH: “inbound event was dropped because of pending overdue app switch”;
4. DROP\_REASON\_BLOCKED: “inbound event was dropped because the current application is not responding and the user has started interacting with a different application””;
5. DROP\_REASON\_STALE: “inbound event was dropped because it is stale”;

其他:

* doDispatchCycleFinishedLockedInterruptible的过程, 会记录分发时间超过2s的事件,
* findFocusedWindowTargetsLocked的过程, 可以统计等待时长信息.

### 参考

[Input系统—ANR原理分析](http://gityuan.com/2017/01/01/input-anr/)

# 情景分析

## 事件拦截原理

### 基本原理

interceptMotionBeforeQueueingNonInteractive

在熄屏幕上，motionEvent可以监听到

WindowManagerService中新建了PhoneWindowManager和InputMonitor,并且将设置为InputManagerService对象的回调对象。其实这2个类都和Event的处理有关,在Event分发之前进行处理,相当于截取,具体的细节就一层一层抽丝剥茧了。

PhoneWindowManager是没有对应的C/C++层代码了。

看下InputManagerService的setWindowManagerCallbacks方法,

https://blog.csdn.net/u012439416/article/details/54605998

### interceptMotionBeforeQueueingNonInteractive

## input事件模拟injectKeyEvent

在adb的input命令中或者AccessibilityManagerService的input部分，最终调起了injectKeyEvent，顾名思义，这个的目的是注入事件，用于自动化测试用的

### 二、InputManager相关代码

我们再来看看InputManager的injectInputEvent函数，最后还是调用了InputManagerService的injectInputEvent函数。

1. public boolean injectInputEvent(InputEvent event, int mode) {
2. if (event == null) {
3. throw new IllegalArgumentException("event must not be null");
4. }
5. if (mode != INJECT\_INPUT\_EVENT\_MODE\_ASYNC
6. && mode != INJECT\_INPUT\_EVENT\_MODE\_WAIT\_FOR\_FINISH
7. && mode != INJECT\_INPUT\_EVENT\_MODE\_WAIT\_FOR\_RESULT) {
8. throw new IllegalArgumentException("mode is invalid");
9. }
11. try {
12. return mIm.injectInputEvent(event, mode);*//调用了InputManagerService的injectInputEvent函数*
13. } catch (RemoteException ex) {
14. return false;
15. }
16. }

我们再来看看InputManagerService的injectInputEvent函数

1. @Override *// Binder call*
2. public boolean injectInputEvent(InputEvent event, int mode) {
3. return injectInputEventInternal(event, Display.DEFAULT\_DISPLAY, mode);
4. }

我们再来看injectInputEventInternal函数

1. private boolean injectInputEventInternal(InputEvent event, int displayId, int mode) {
2. if (event == null) {
3. throw new IllegalArgumentException("event must not be null");
4. }
5. if (mode != InputManager.INJECT\_INPUT\_EVENT\_MODE\_ASYNC
6. && mode != InputManager.INJECT\_INPUT\_EVENT\_MODE\_WAIT\_FOR\_FINISH
7. && mode != InputManager.INJECT\_INPUT\_EVENT\_MODE\_WAIT\_FOR\_RESULT) {
8. throw new IllegalArgumentException("mode is invalid");
9. }
11. final int pid = Binder.getCallingPid();
12. final int uid = Binder.getCallingUid();
13. final long ident = Binder.clearCallingIdentity();
14. final int result;
15. try {
16. result = nativeInjectInputEvent(mPtr, event, displayId, pid, uid, mode,*//主要看这个jni函数*
17. INJECTION\_TIMEOUT\_MILLIS, WindowManagerPolicy.FLAG\_DISABLE\_KEY\_REPEAT);
18. } finally {
19. Binder.restoreCallingIdentity(ident);
20. }
21. switch (result) {
22. case INPUT\_EVENT\_INJECTION\_PERMISSION\_DENIED:
23. Slog.w(TAG, "Input event injection from pid " + pid + " permission denied.");
24. throw new SecurityException(
25. "Injecting to another application requires INJECT\_EVENTS permission");
26. case INPUT\_EVENT\_INJECTION\_SUCCEEDED:
27. return true;
28. case INPUT\_EVENT\_INJECTION\_TIMED\_OUT:
29. Slog.w(TAG, "Input event injection from pid " + pid + " timed out.");
30. return false;
31. case INPUT\_EVENT\_INJECTION\_FAILED:
32. default:
33. Slog.w(TAG, "Input event injection from pid " + pid + " failed.");
34. return false;
35. }
36. }

### 三、native层代码

上面这个函数主要调用了nativeInjectInputEvent这个native函数。

1. static jint nativeInjectInputEvent(JNIEnv\* env, jclass */\* clazz \*/*,
2. jlong ptr, jobject inputEventObj, jint displayId, jint injectorPid, jint injectorUid,
3. jint syncMode, jint timeoutMillis, jint policyFlags) {
4. NativeInputManager\* im = reinterpret\_cast<NativeInputManager\*>(ptr);
6. if (env->IsInstanceOf(inputEventObj, gKeyEventClassInfo.clazz)) {
7. KeyEvent keyEvent;
8. status\_t status = android\_view\_KeyEvent\_toNative(env, inputEventObj, & keyEvent);
9. if (status) {
10. jniThrowRuntimeException(env, "Could not read contents of KeyEvent object.");
11. return INPUT\_EVENT\_INJECTION\_FAILED;
12. }
14. return (jint) im->getInputManager()->getDispatcher()->injectInputEvent(
15. & keyEvent, displayId, injectorPid, injectorUid, syncMode, timeoutMillis,
16. uint32\_t(policyFlags));
17. } else if (env->IsInstanceOf(inputEventObj, gMotionEventClassInfo.clazz)) {
18. const MotionEvent\* motionEvent = android\_view\_MotionEvent\_getNativePtr(env, inputEventObj);
19. if (!motionEvent) {
20. jniThrowRuntimeException(env, "Could not read contents of MotionEvent object.");
21. return INPUT\_EVENT\_INJECTION\_FAILED;
22. }
24. return (jint) im->getInputManager()->getDispatcher()->injectInputEvent(
25. motionEvent, displayId, injectorPid, injectorUid, syncMode, timeoutMillis,
26. uint32\_t(policyFlags));
27. } else {
28. jniThrowRuntimeException(env, "Invalid input event type.");
29. return INPUT\_EVENT\_INJECTION\_FAILED;
30. }
31. }

这个函数先根据event的种类，有KeyEvent，MoveEvent来调用相关函数。我们再来看看InputDispatcher::injectInputEvent函数

1. int32\_t InputDispatcher::injectInputEvent(const InputEvent\* event, int32\_t displayId,
2. int32\_t injectorPid, int32\_t injectorUid, int32\_t syncMode, int32\_t timeoutMillis,
3. uint32\_t policyFlags) {
5. nsecs\_t endTime = now() + milliseconds\_to\_nanoseconds(timeoutMillis);
7. policyFlags |= POLICY\_FLAG\_INJECTED;
8. if (hasInjectionPermission(injectorPid, injectorUid)) {
9. policyFlags |= POLICY\_FLAG\_TRUSTED;
10. }
12. EventEntry\* firstInjectedEntry;
13. EventEntry\* lastInjectedEntry;
14. switch (event->getType()) {
15. case AINPUT\_EVENT\_TYPE\_KEY: {
16. const KeyEvent\* keyEvent = static\_cast<const KeyEvent\*>(event);
17. int32\_t action = keyEvent->getAction();
18. if (! validateKeyEvent(action)) {
19. return INPUT\_EVENT\_INJECTION\_FAILED;
20. }
22. int32\_t flags = keyEvent->getFlags();
23. if (flags & AKEY\_EVENT\_FLAG\_VIRTUAL\_HARD\_KEY) {
24. policyFlags |= POLICY\_FLAG\_VIRTUAL;
25. }
27. if (!(policyFlags & POLICY\_FLAG\_FILTERED)) {
28. mPolicy->interceptKeyBeforeQueueing(keyEvent, */\*byref\*/* policyFlags);
29. }
31. mLock.lock();
32. firstInjectedEntry = new KeyEntry(keyEvent->getEventTime(),
33. keyEvent->getDeviceId(), keyEvent->getSource(),
34. policyFlags, action, flags,
35. keyEvent->getKeyCode(), keyEvent->getScanCode(), keyEvent->getMetaState(),
36. keyEvent->getRepeatCount(), keyEvent->getDownTime());
37. lastInjectedEntry = firstInjectedEntry;
38. break;
39. ......
40. }
42. InjectionState\* injectionState = new InjectionState(injectorPid, injectorUid);
43. if (syncMode == INPUT\_EVENT\_INJECTION\_SYNC\_NONE) {
44. injectionState->injectionIsAsync = true;
45. }
47. injectionState->refCount += 1;
48. lastInjectedEntry->injectionState = injectionState;
50. bool needWake = false;
51. for (EventEntry\* entry = firstInjectedEntry; entry != NULL; ) {
52. EventEntry\* nextEntry = entry->next;
53. needWake |= enqueueInboundEventLocked(entry);
54. entry = nextEntry;
55. }
56. ......

这个函数和notifyKey函数很像，notifyKey函数是正常走按键流程在dispatchReader中调用的函数。这里也会想notifyKey一样，先调用PhoneWindowManager的interceptKeyBeforeQueueing函数，然后根据不同类型的Event，然后创建EventEntry，最后调用了enqueueInboundEventLocked函数，这个函数之前在按键流程中分析过了。最后也会调用mLooper->wake函数，把InputDispatcherThread线程唤醒，然后执行dispatchOnce函数：

## 拦截ConfigurationChanged

InputDispatcherThread能源源不断的获知系统设备中实时发生的时, 它还可以向InputReader注册监听多种事件如notifyConfigurationChanged()

### InputDispatcher.notifyConfigurationChanged

DsfnotifyConfigurationChanged中，构造一个消息实体入队了，当队列有消息入队时thredLoop就会执行，

1. void InputDispatcher::notifyConfigurationChanged(const NotifyConfigurationChangedArgs\* args) {
2. #if DEBUG\_INBOUND\_EVENT\_DETAILS
3. ALOGD("notifyConfigurationChanged - eventTime=%lld", args->eventTime);
4. #endif
6. bool needWake;
7. { *// acquire lock*
8. AutoMutex \_l(mLock);
10. ConfigurationChangedEntry\* newEntry = new ConfigurationChangedEntry(args->eventTime);
11. needWake = enqueueInboundEventLocked(newEntry);
12. } *// release lock*
14. if (needWake) {
15. mLooper->wake();
16. }

### dispatchOnceInnerLocked

进而调用到dispatchOnceInnerLocked()

1. void InputDispatcher::dispatchOnceInnerLocked(nsecs\_t\* nextWakeupTime)
2. {
3. ......
4. switch (mPendingEvent->type) {
5. case EventEntry::TYPE\_CONFIGURATION\_CHANGED: {
6. ConfigurationChangedEntry\* typedEntry =
7. static\_cast<ConfigurationChangedEntry\*>(mPendingEvent);
8. done = dispatchConfigurationChangedLocked(currentTime, typedEntry);
9. dropReason = DROP\_REASON\_NOT\_DROPPED; *// configuration changes are never dropped*
10. break;
11. }
12. ......
13. }

如果获取到的消息是TYPE\_CONFIGURATION\_CHANGED则调用dispatchConfigurationChangedLocked

### dispatchConfigurationChangedLocked

f

1. bool InputDispatcher::dispatchConfigurationChangedLocked(
2. nsecs\_t currentTime, ConfigurationChangedEntry\* entry) {
3. #if DEBUG\_OUTBOUND\_EVENT\_DETAILS
4. ALOGD("dispatchConfigurationChanged - eventTime=%lld", entry->eventTime);
5. #endif
7. *// Reset key repeating in case a keyboard device was added or removed or something.*
8. resetKeyRepeatLocked();
10. *// Enqueue a command to run outside the lock to tell the policy that the configuration changed.*
11. CommandEntry\* commandEntry = postCommandLocked(
12. & InputDispatcher::<span style="color:#ff0000;">doNotifyConfigurationChangedInterruptible</span>);
13. commandEntry->eventTime = entry->eventTime;
14. return true;
15. }
16. void InputDispatcher::doNotifyConfigurationChangedInterruptible(
17. CommandEntry\* commandEntry) {
18. mLock.unlock();
20. mPolicy-><span style="color:#ff0000;">notifyConfigurationChanged</span>(commandEntry->eventTime);
22. mLock.lock();

mPolicy->notifyConfigurationChanged(commandEntry->eventTime)会回调到InputManagerService.java的notifyConfigurationChanged

### notifyConfigurationChanged

一次走c和java链路，所以又进一步调用到了InputMonitor的notifyConfigurationChanged方法

public void notifyConfigurationChanged() {

mService.sendNewConfiguration();

synchronized (mInputDevicesReadyMonitor) {

if (!mInputDevicesReady) {

mInputDevicesReady = true;

mInputDevicesReadyMonitor.notifyAll();

}

}

}

### WindowManagerService，sendNewConfiguration

mService为WindowManagerService，sendNewConfiguration实现

void sendNewConfiguration() {

try {

mActivityManager.updateConfiguration(null);

} catch (RemoteException e) {

}

}

mActivityManager为ActivityManagerService，并调用到它的updateConfiguration

public void updateConfiguration(Configuration values) {

enforceCallingPermission(android.Manifest.permission.CHANGE\_CONFIGURATION,

"updateConfiguration()");

synchronized(this) {

if (values == null && mWindowManager != null) {

// sentinel: fetch the current configuration from the window manager

values = mWindowManager.computeNewConfiguration();

}

if (mWindowManager != null) {

mProcessList.applyDisplaySize(mWindowManager);

}

final long origId = Binder.clearCallingIdentity();

if (values != null) {

Settings.System.clearConfiguration(values);

}

updateConfigurationLocked(values, null, false, false);

Binder.restoreCallingIdentity(origId);

}

至此路由结束，上层完成业务定制

https://blog.csdn.net/csm\_qz/article/details/50757113

# KeyBoardLayout（KeyMapper）

Joystick修改

1： 概述

2： KeyboardLayout的设置流程

3： RawEvent上报根据相应的KeyMapper设置进行修改

这是键盘！！

<https://www.jianshu.com/p/544984e19e5c>

# Input之JoyStick

KW：SyntheticJoystickHandler android Joystick ViewRootImpl 拦截MotionEvent update

Keycode方向键 D-pad KeyEvent

D-pad KeyEvent 全局 拦截



<https://blog.csdn.net/tankai19880619/article/details/8556282>

## 思路

在phoneWindowManger拦截？

**ViewRootImpl再往下面走走**

logcat -b main -b system -b radio -b events -v time

com.android.settings I/DualScreen: onKeyUp->keyCode:19

Db通信

灭屏下不要触发事件了

全局事件，在systemui监听？有用么？

/\*\* Key code constant: Directional Pad Up key.

\* May also be synthesized from trackball motions. \*/

public static final int KEYCODE\_DPAD\_UP = 19;

/\*\* Key code constant: Directional Pad Down key.

\* May also be synthesized from trackball motions. \*/

public static final int KEYCODE\_DPAD\_DOWN = 20;

/\*\* Key code constant: Directional Pad Left key.

\* May also be synthesized from trackball motions. \*/

public static final int KEYCODE\_DPAD\_LEFT = 21;

/\*\* Key code constant: Directional Pad Right key.

\* May also be synthesized from trackball motions. \*/

public static final int KEYCODE\_DPAD\_RIGHT = 22;

/\*\* Key code constant: Directional Pad Center key.

\* May also be synthesized from trackball motions. \*/

public static final int KEYCODE\_DPAD\_CENTER = 23;

### 相关类

**InputDevice**.java (core\java\android\view):

**MotionEvent**.java (core\java\android\view)

**ViewRootImpl**.java (core\java\android\view):

View.java (core\java\android\view):

InputEventConsistencyVerifier.java (core\java\android\view):

PointerLocationView.java (core\java\com\android\internal\widget):

System-current.txt (api):

Test-current.txt (api):

Current.txt (api): field public static final int SOURCE\_CLASS\_JOYSTICK = 16; // 0x10

KeyEvent.java (core\java\android\view): \* joystick is pressed. \*/

MLand.java (packages\systemui\src\com\android\systemui\egg):

虚拟按键定义

/\*\* Key code constant: Left Thumb Button key.

\* On a game controller, the left thumb button indicates that the left (or only)

\* joystick is pressed. \*/

public static final int KEYCODE\_BUTTON\_THUMBL = 106;

/\*\* Key code constant: Right Thumb Button key.

\* On a game controller, the right thumb button indicates that the right

\* joystick is pressed. \*/

public static final int KEYCODE\_BUTTON\_THUMBR = 107;

### 交流

#### Xnady

<https://github.com/KoVszone/GamePad>

[https://blog.csdn.net/s278777851/article/details/6956226](javascript:%20void%200)



https://github.com/KoVszone/GamePad

http://www.technorange.com/2017/01/how-to-map-gamepad-or-joystick-on-android-to-play-games-using-usb-bt-joycenter/

https://blog.csdn.net/AWNUXCVBN/article/details/12285761

https://blog.csdn.net/luoshengyang/article/details/6882903

https://developer.android.com/reference/android/view/MotionEvent

#### Rian之前的游戏同事

#### 同事交流，明建

## 方案

### app进程局部拦截：

[**Android Tv 中的按键事件 KeyEvent 分发处理流程**](https://www.cnblogs.com/dasusu/p/7403698.html)

ViewRootImpl拦截

<https://www.jianshu.com/p/9528114bcded?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

### AccessibilityService拦截失败

<https://blog.csdn.net/w815878564/article/details/53331086>

　第四种办法是使用系统的无障碍服务，也就是AccessibilityService。我最终采用的方式就是这个，因为这个不仅可以监听按键，还能拦截。关于AccessibilityService的实现，网上讲的比较多了。我这里只提几个需要注意的点：   
　　1、按键拦截需要覆写onKeyEvent方法   
　　2、AndroidManifest.xml文件中配置如下：

http://www.voidcn.com/article/p-qskfndfg-bqe.html

### PhoneWm/Win

处在当前界面的才会分发事件，winds的选择窗口拦截

长按：keyCode=23

#### 未解决问题

目前FN键在PhoneWm有实现，能获取

1. PhoneWm.interceptKeyBeforeDispatching:为何5d方向键无法响应，但是能够收到KEYCODE\_DPAD\_CENTER 的KeyEvent,
2. ViewRootImp:在ViewRootImp之上能接收到5d方向键的MotionEvent事件，然后在SyntheticJoystickHandler的update中计算转化为类似KEYCODE\_DPAD\_RIGHT的new KeyEvent，并通过enqueueInputEvent进入队列，开始KeyEvent的事件处理，最终路由到app
3. PhoneWindowManager :interceptKeyBeforeDispatching只能拦截KeyEvent， 那么MotionEvent怎么拦截到呢

4. 关于 InputChannel，在WinMS的HideNavInputConsumer有实现，在

5D按下才有事件：

Performing fallback: KeyEvent { action=ACTION\_UP, keyCode=KEYCODE\_DPAD\_CENTER, scanCode=304, metaState=0, flags=0x408, repeatCount=0, eventTime=3087852, downTime=3086556, deviceId=3, source=0x501 }

@Override

public long interceptKeyBeforeDispatching(WindowState win, KeyEvent event, int policyFlags) {

final boolean keyguardOn = keyguardOn();

final int keyCode = event.getKeyCode();

final int repeatCount = event.getRepeatCount();

final int metaState = event.getMetaState();

final int flags = event.getFlags();

final boolean down = event.getAction() == KeyEvent.ACTION\_DOWN;

final boolean canceled = event.isCanceled();

if (DEBUG\_INPUT) {

Log.d(TAG, "interceptKeyTi keyCode=" + keyCode + " down=" + down + " repeatCount="

+ repeatCount + " keyguardOn=" + keyguardOn + " mHomePressed=" + mHomePressed

+ " canceled=" + canceled);

}

### DualScreen

Framework找不到呢

LOGcat

### Native framework方案

#### interceptMotionBeforeQueueing

interceptMotionBeforeDispatch

参数比较好传递，这个还是在systeserver里面，可以整！

触屏不要拦截了，只是拦截摇杆的按键

##### 增加一个函数过程

|  |  |  |
| --- | --- | --- |
|  |  | out |
|  |  |  |
| Framework.jar | WindowManagerPolicy :  (base\core\java\android\view) | Framework.jar |
| Services.jar | PhoneWindowManager  (base\services\core\java\com\android\server\policy)  InputMonitor.java  (base\services\core\java\com\android\server\wm  InputManagerService.java (base\services\core\java\com\android\server\input) | Services.jar |
| Jni | com\_android\_server\_input\_InputManagerService  (base\services\core\jni) | Lib\_android\_Services.so |
| native | InputDispatcher.cpp (native\services\inputflinger)  InputDispatcher.h (native\services\inputflinger) | Inputflinger.so |

事件转化

后期事件优化

事件封装修改大

基本原理

mPolicy->interceptMotionBeforeQueueing(args->eventTime, /\*byref\*/ policyFlags);

void NativeInputManager::interceptMotionBeforeQueueing(nsecs\_t when, uint32\_t& policyFlags) {

// Policy:

// - Ignore untrusted events and pass them along.

// - No special filtering for injected events required at this time.

// - Filter normal events based on screen state.

// - For normal events brighten (but do not wake) the screen if currently dim.

bool interactive = mInteractive.load();

if (interactive) {

policyFlags |= POLICY\_FLAG\_INTERACTIVE;

}

if ((policyFlags & POLICY\_FLAG\_TRUSTED) && !(policyFlags & POLICY\_FLAG\_INJECTED)) {

if (policyFlags & POLICY\_FLAG\_INTERACTIVE) {

policyFlags |= POLICY\_FLAG\_PASS\_TO\_USER;

} else {

JNIEnv\* env = jniEnv();

jint wmActions = env->CallIntMethod(mServiceObj,

gServiceClassInfo.interceptMotionBeforeQueueingNonInteractive,

when, policyFlags);

if (checkAndClearExceptionFromCallback(env,

"interceptMotionBeforeQueueingNonInteractive")) {

wmActions = 0;

}

handleInterceptActions(wmActions, when, /\*byref\*/ policyFlags);

}

} else {

if (interactive) {

policyFlags |= POLICY\_FLAG\_PASS\_TO\_USER;

}

}

为什么log没有差异呢？

#### startDispatchCycleLocked

模块

.java (core\java\android\view): public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags);

#### 如何兼容其他按键

滚轮的按键信息为60，但是有4个，需要过滤处理，让rc容错，或者系统

## 配置文件

system/usr/keylayout/Generic.kl

**[plain]** [view plain](https://blog.csdn.net/tankai19880619/article/details/8556282) [copy](https://blog.csdn.net/tankai19880619/article/details/8556282)

1. ......
2. # Joystick and game controller axes.
3. # Axes that are not mapped will be assigned generic axis numbers by the input subsystem.
4. axis 0x00 X
5. axis 0x01 Y
6. axis 0x02 Z
7. axis 0x03 RX
8. axis 0x04 RY
9. axis 0x05 RZ
10. axis 0x06 THROTTLE
11. axis 0x07 RUDDER
12. axis 0x08 WHEEL
13. axis 0x09 GAS
14. axis 0x0a BRAKE
15. axis 0x10 HAT\_X
16. axis 0x11 HAT\_Y

## 二、原理部分

ics/frameworks/base/core/java/android/view/ViewRootImpl.java

添加对axis 0x03 RX和axis 0x04 RY左边做上下左右按键的支持：

### 调用关系如下：

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/8556282) [copy](https://blog.csdn.net/tankai19880619/article/details/8556282)

1. **public** **void** handleMotion(MotionEvent event, InputQueue.FinishedCallback finishedCallback) {
2. ......
3. }
4. **private** **void** dispatchGenericMotion(MotionEvent event, **boolean** sendDone) {
5. ......
6. }
8. **public** **void** handleMessage(Message msg) {
9. ......
10. }
11. **private** **void** deliverGenericMotionEvent(MotionEvent event, **boolean** sendDone) {
12. ......
13. }

### 2.核心代码片段

**[java]** [view plain](https://blog.csdn.net/tankai19880619/article/details/8556282) [copy](https://blog.csdn.net/tankai19880619/article/details/8556282)

1. **private** **void** updateJoystickDirection(MotionEvent event, **boolean** synthesizeNewKeys) {
2. ......
3. **int** xDirection = joystickAxisValueToDirection(event.getAxisValue(MotionEvent.AXIS\_HAT\_X));
4. **if** (xDirection == 0) {
5. xDirection = joystickAxisValueToDirection(event.getX());
6. }
7. //add by tankai
8. **if** (xDirection == 0) {
9. xDirection = joystickAxisValueToDirection(event.getAxisValue(MotionEvent.AXIS\_RX));
10. }
11. //end
12. **int** yDirection = joystickAxisValueToDirection(event.getAxisValue(MotionEvent.AXIS\_HAT\_Y));
13. **if** (yDirection == 0) {
14. yDirection = joystickAxisValueToDirection(event.getY());
15. }
16. //add by tankai
17. **if** (yDirection == 0) {
18. yDirection = joystickAxisValueToDirection(event.getAxisValue(MotionEvent.AXIS\_RY));
19. }
20. //end
21. **if** (xDirection != mLastJoystickXDirection) {
22. **if** (mLastJoystickXKeyCode != 0) {
23. deliverKeyEvent(**new** KeyEvent(time, time,
24. KeyEvent.ACTION\_UP, mLastJoystickXKeyCode, 0, metaState,
25. deviceId, 0, KeyEvent.FLAG\_FALLBACK, source), **false**);
26. mLastJoystickXKeyCode = 0;
27. }
28. mLastJoystickXDirection = xDirection;
29. **if** (xDirection != 0 && synthesizeNewKeys) {
30. mLastJoystickXKeyCode = xDirection > 0
31. ? KeyEvent.KEYCODE\_DPAD\_RIGHT : KeyEvent.KEYCODE\_DPAD\_LEFT;
32. deliverKeyEvent(**new** KeyEvent(time, time,
33. KeyEvent.ACTION\_DOWN, mLastJoystickXKeyCode, 0, metaState,
34. deviceId, 0, KeyEvent.FLAG\_FALLBACK, source), **false**);
35. }
36. }
38. **if** (yDirection != mLastJoystickYDirection) {
39. **if** (mLastJoystickYKeyCode != 0) {
40. deliverKeyEvent(**new** KeyEvent(time, time,
41. KeyEvent.ACTION\_UP, mLastJoystickYKeyCode, 0, metaState,
42. deviceId, 0, KeyEvent.FLAG\_FALLBACK, source), **false**);
43. mLastJoystickYKeyCode = 0;
44. }
45. mLastJoystickYDirection = yDirection;
46. **if** (yDirection != 0 && synthesizeNewKeys) {
47. mLastJoystickYKeyCode = yDirection > 0
48. ? KeyEvent.KEYCODE\_DPAD\_DOWN : KeyEvent.KEYCODE\_DPAD\_UP;
49. deliverKeyEvent(**new** KeyEvent(time, time,
50. KeyEvent.ACTION\_DOWN, mLastJoystickYKeyCode, 0, metaState,
51. deviceId, 0, KeyEvent.FLAG\_FALLBACK, source), **false**);
52. }
53. }
54. }

# WindowManagerPolicy

frameworks\native\include\input\input.h

frameworks/base/core/java/android/view/WindowManagerPolicy.java.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| POLICY\_FLAG\_WAKE | 该事件应该唤醒设备 |  |
| POLICY\_FLAG\_VIRTUAL | 虚拟按键：电容处理  // Indicates that the key is virtual, such as a capacitive button, and should  // generate haptic feedback. Virtual keys may be suppressed for some time  // after a recent touch to prevent accidental activation of virtual keys adjacent  // to the touch screen during an edge swipe. |  |
| POLICY\_FLAG\_FUNCTION | Indicates that the key is the special function modifier. |  |
| POLICY\_FLAG\_GESTURE = 0x00000008, | special gesture that has been detected by  // the touch firmware or driver. Causes touch events from the same device to be canceled. |  |
| POLICY\_FLAG\_RAW\_MASK = 0x0000ffff, |  |  |
| POLICY\_FLAG\_INJECTED | input event was injected. |  |
| POLICY\_FLAG\_TRUSTED  0x02000000, | input event is from a trusted source such as a directly attached  // input device or an application with system-wide event injection permission. | 默认链接的host的都会有这个值，可信输入 |
| POLICY\_FLAG\_FILTERED | event has passed through an input filter. |  |
| POLICY\_FLAG\_DISABLE\_KEY\_REPEAT |  |  |
| POLICY\_FLAG\_INTERACTIVE  0x20000000 | /\* These flags are set by the input reader policy as it intercepts each event. \*/  // Indicates that the device was in an interactive state when the  // event was intercepted. | 可交互的，亮屏就有 |
| POLICY\_FLAG\_PASS\_TO\_USER | // Indicates that the event should be dispatched to applications.  // The input event should still be sent to the InputDispatcher so that it can see all  // input events received include those that it will not deliver. |  |

:

/\*\*

\* Pass this event to the user / app. To be returned from

\* {@link #interceptKeyBeforeQueueing}.

\*/

public final static int ACTION\_PASS\_TO\_USER = 0x00000001;

ACTION\_PASS\_TO\_USER

# 参考

[探讨Android实现后台(Service)按键监听的功能](https://blog.csdn.net/w815878564/article/details/53331086)

[android Input子系统分析(内核层->android系统层)](http://www.cnblogs.com/MMLoveMeMM/articles/4119812.html)

[Android 4.0 input touch解析（一）](http://www.cnblogs.com/tnxk/archive/2012/10/26/2741326.html)