# 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。

# 源码分析-启动

SystemServer

New InputManagerService()

nativeInit

开始init

## 如何init

### nativeInit

[-> 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。

## 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

d

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

sdf

**[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

Asdf

[-> 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

Dfg

nputListener.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

Sdfg

-> InputListener.cpp]

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

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

}

### InputDispatcher.notifyKey

Sdf

**[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

dfg

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

sadf

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:

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, 再将事件input\_event转换为RawEvent; [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), 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/

# 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

Df 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

## 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. }

# TSAK

## 点见解: 焦点那点事(一)

https://www.jianshu.com/p/641f96c19203