# 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。

# 源码分析-启动

## 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。

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/>

# 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

<https://blog.csdn.net/tankai19880619/article/details/8556282>

## 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. }