[Android应用程序键盘（Keyboard）消息处理机制分析](http://blog.csdn.net/luoshengyang/article/details/6882903)

在Android系统中，键盘按键事件是由WindowManagerService服务来管理的，然后再以消息的形式来分发给应用程序处理，不过和普通消息不一样，它是由硬件中断触发的；在上一篇文章《Android应用程序消息处理机制（Looper、Handler）分析》中，我们分析了Android应用程序的消息处理机制，本文将结合这种消息处理机制来详细分析Android应用程序是如何获得键盘按键消息的。

        在系统启动的时候，SystemServer会启动窗口管理服务WindowManagerService，WindowManagerService在启动的时候就会通过系统输入管理器InputManager来总负责监控键盘消息。这些键盘消息一般都是分发给当前激活的Activity窗口来处理的，因此，当前激活的Activity窗口在创建的时候，会到WindowManagerService中去注册一个接收键盘消息的通道，表明它要处理键盘消息，而当InputManager监控到有键盘消息时，就会分给给它处理。当当前激活的Activity窗口不再处于激活状态时，它也会到WindowManagerService中去反注册之前的键盘消息接收通道，这样，InputManager就不会再把键盘消息分发给它来处理。

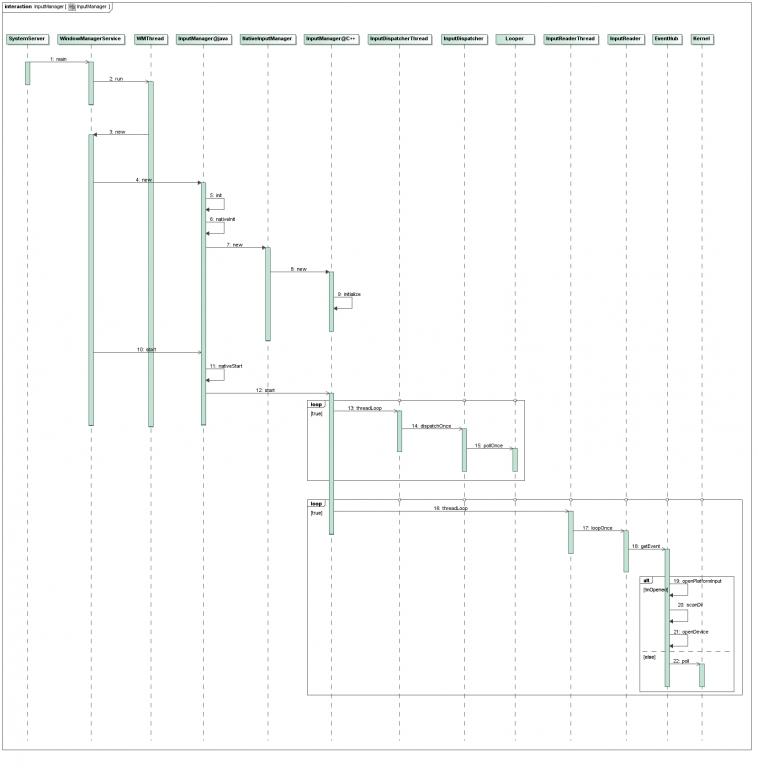
        由于本文的内容比较多，在接下面的章节中，我们将分为五个部分来详细描述Android应用程序获得键盘按键消息的过程，每一个部分都是具体描述键盘消息处理过程中的一个过程。结合上面的键盘消息处理框架，这四个过程分别是InputManager的启动过程、应用程序注册键盘消息接收通道的过程、InputManager分发键盘消息给应用程序的过程以及应用程序注销键盘消息接收通道的过程。为了更好地理解Android应用程序获得键盘按键消息的整个过程，建议读者首先阅读[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)一文，理解了Android应用程序的消息处理机制后，就能很好的把握本文的内容。

        1. InputManager的启动过程分析

        前面说过，Android系统的键盘事件是由InputManager来监控的，而InputManager是由窗口管理服务WindowManagerService来启动的。

        从前面一篇文章[Android系统进程Zygote启动过程的源代码分析](http://blog.csdn.net/luoshengyang/article/details/6768304)中，我们知道在Android系统中，Zygote进程负责启动系统服务进程SystemServer，而系统服务进程SystemServer负责启动系统中的各种关键服务，例如我们在前面两篇文章[Android应用程序安装过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6766010)和[Android系统默认Home应用程序（Launcher）的启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6767736)中提到的Package管理服务PackageManagerService和Activity管理服务ActivityManagerService。这里我们所讨论的窗口管理服务WindowManagerService也是由SystemServer来启动的，具体的启动过程这里就不再详述了，具体可以参考PackageManagerService和ActivityManagerService的启动过程。

       了解了WindowManagerService的启动过程之后，我们就可以继续分析InputManager的启动过程了。我们先来看一下InputManager启动过程的序列图，然后根据这个序列图来一步步分析它的启动过程：



[点击查看大图](http://hi.csdn.net/attachment/201110/23/0_1319371717FBD2.gif)

        Step 1. WindowManagerService.main

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **static** WindowManagerService main(Context context,
6. PowerManagerService pm, **boolean** haveInputMethods) {
7. WMThread thr = **new** WMThread(context, pm, haveInputMethods);
8. thr.start();
10. **synchronized** (thr) {
11. **while** (thr.mService == **null**) {
12. **try** {
13. thr.wait();
14. } **catch** (InterruptedException e) {
15. }
16. }
17. **return** thr.mService;
18. }
19. }
21. ......
22. }

        它通过一个线程WMThread实例来执行全局唯一的WindowManagerService实例的启动操作。这里调用WMThread实例thr的start成员函数时，会进入到WMThread实例thr的run函数中去。

        Step 2. WMThread.run

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **static** **class** WMThread **extends** Thread {
6. ......
8. **public** **void** run() {
9. ......
11. WindowManagerService s = **new** WindowManagerService(mContext, mPM,
12. mHaveInputMethods);
13. ......
14. }
15. }

18. ......
19. }

       这里执行的主要操作就是创建一个WindowManagerService实例，这样会调用到WindowManagerService构造函数中去。

       Step 3. WindowManagerService<init>

       WindowManagerService类的构造函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **final** InputManager mInputManager;
7. ......
9. **private** WindowManagerService(Context context, PowerManagerService pm,
10. **boolean** haveInputMethods) {
11. ......
13. mInputManager = **new** InputManager(context, **this**);
15. ......
17. mInputManager.start();
19. ......
20. }

23. ......
24. }

         这里我们只关心InputManager的创建过程，而忽略其它无关部分。首先是创建一个InputManager实例，然后再调用它的start成员函数来监控键盘事件。在创建InputManager实例的过程中，会执行一些初始化工作，因此，我们先进入到InputManager类的构造函数去看看，然后再回过头来分析它的start成员函数。

         Step 4. InputManager<init>@java

         Java层的InputManager类的构造函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. **public** InputManager(Context context, WindowManagerService windowManagerService) {
5. **this**.mContext = context;
6. **this**.mWindowManagerService = windowManagerService;
8. **this**.mCallbacks = **new** Callbacks();
10. init();
11. }
13. ......
14. }

        这里只是简单地初始化InputManager类的一些成员变量，然后调用init函数进一步执行初始化操作。

        Step 5. InputManager.init

        这个函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. **private** **void** init() {
5. Slog.i(TAG, "Initializing input manager");
6. nativeInit(mCallbacks);
7. }
9. ......
10. }

       函数init通过调用本地方法nativeInit来执行C++层的相关初始化操作。

       Step 6. InputManager.nativeInit

       这个函数定义在frameworks/base/services/jni$ vi com\_android\_server\_InputManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_server\_InputManager\_nativeInit(JNIEnv\* env, jclass clazz,
2. jobject callbacks) {
3. **if** (gNativeInputManager == NULL) {
4. gNativeInputManager = **new** NativeInputManager(callbacks);
5. } **else** {
6. LOGE("Input manager already initialized.");
7. jniThrowRuntimeException(env, "Input manager already initialized.");
8. }
9. }

        这个函数的作用是创建一个NativeInputManager实例，并保存在gNativeInputManager变量中。由于是第一次调用到这里，因此，gNativeInputManager为NULL，于是就会new一个NativeInputManager对象出来，这样就会执行NativeInputManager类的构造函数来执其它的初始化操作。

       Step 7. NativeInputManager<init>

       NativeInputManager类的构造函数定义在frameworks/base/services/jni$ vi com\_android\_server\_InputManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. NativeInputManager::NativeInputManager(jobject callbacksObj) :
2. mFilterTouchEvents(-1), mFilterJumpyTouchEvents(-1), mVirtualKeyQuietTime(-1),
3. mMaxEventsPerSecond(-1),
4. mDisplayWidth(-1), mDisplayHeight(-1), mDisplayOrientation(ROTATION\_0) {
5. JNIEnv\* env = jniEnv();
7. mCallbacksObj = env->NewGlobalRef(callbacksObj);
9. sp<EventHub> eventHub = **new** EventHub();
10. mInputManager = **new** InputManager(eventHub, **this**, **this**);
11. }

        这里只要是创建了一个EventHub实例，并且把这个EventHub作为参数来创建InputManager对象。注意，这里的InputManager类是定义在C++层的，和前面在Java层的InputManager不一样，不过它们是对应关系。EventHub类是真正执行监控键盘事件操作的地方，后面我们会进一步分析到，现在我们主要关心InputManager实例的创建过程，它会InputManager类的构造函数里面执行一些初始化操作。

        Step 8. InputManager<init>@C++

        C++层的InputManager类的构造函数定义在frameworks/base/libs/ui/InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. InputManager::InputManager(
2. **const** sp<EventHubInterface>& eventHub,
3. **const** sp<InputReaderPolicyInterface>& readerPolicy,
4. **const** sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
5. mDispatcher = **new** InputDispatcher(dispatcherPolicy);
6. mReader = **new** InputReader(eventHub, readerPolicy, mDispatcher);
7. initialize();
8. }

        这里主要是创建了一个InputDispatcher对象和一个InputReader对象，并且分别保存在成员变量mDispatcher和mReader中。InputDispatcher类是负责把键盘消息分发给当前激活的Activity窗口的，而InputReader类则是通过EventHub类来实现读取键盘事件的，后面我们会进一步分析。创建了这两个对象后，还要调用initialize函数来执行其它的初始化操作。

        Step 9. InputManager.initialize

        这个函数定义在frameworks/base/libs/ui/InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputManager::initialize() {
2. mReaderThread = **new** InputReaderThread(mReader);
3. mDispatcherThread = **new** InputDispatcherThread(mDispatcher);
4. }

        这个函数创建了一个InputReaderThread线程实例和一个InputDispatcherThread线程实例，并且分别保存在成员变量mReaderThread和mDispatcherThread中。这里的InputReader实列mReader就是通过这里的InputReaderThread线程实列mReaderThread来读取键盘事件的，而InputDispatcher实例mDispatcher则是通过这里的InputDispatcherThread线程实例mDisptacherThread来分发键盘消息的。

        至此，InputManager的初始化工作就完成了，在回到Step 3中继续分析InputManager的进一步启动过程之前，我们先来作一个小结，看看这个初始化过程都做什么事情：

        A. 在Java层中的WindowManagerService中创建了一个InputManager对象，由它来负责管理Android应用程序框架层的键盘消息处理；

        B. 在C++层也相应地创建一个InputManager本地对象来负责监控键盘事件；

        C. 在C++层中的InputManager对象中，分别创建了一个InputReader对象和一个InputDispatcher对象，前者负责读取系统中的键盘消息，后者负责把键盘消息分发出去；

        D. InputReader对象和一个InputDispatcher对象分别是通过InputReaderThread线程实例和InputDispatcherThread线程实例来实键盘消息的读取和分发的。

        有了这些对象之后，万事就俱备了，回到Step 3中，调用InputManager类的start函数来执行真正的启动操作。

        Step 10. InputManager.start

        这个函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. **public** **void** start() {
5. Slog.i(TAG, "Starting input manager");
6. nativeStart();
7. }
9. ......
10. }

        这个函数通过调用本地方法nativeStart来执行进一步的启动操作。

        Step 11. InputManager.nativeStart

        这个函数定义在frameworks/base/services/jni$ vi com\_android\_server\_InputManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_server\_InputManager\_nativeStart(JNIEnv\* env, jclass clazz) {
2. **if** (checkInputManagerUnitialized(env)) {
3. **return**;
4. }
6. status\_t result = gNativeInputManager->getInputManager()->start();
7. **if** (result) {
8. jniThrowRuntimeException(env, "Input manager could not be started.");
9. }
10. }

        这里的gNativeInputManager对象是在前面的Step 6中创建的，通过它的getInputManager函数可以返回C++层的InputManager对象，接着调用这个InputManager对象的start函数。

        Step 12. InputManager.start

        这个函数定义在frameworks/base/libs/ui/InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputManager::start() {
2. status\_t result = mDispatcherThread->run("InputDispatcher", PRIORITY\_URGENT\_DISPLAY);
3. **if** (result) {
4. LOGE("Could not start InputDispatcher thread due to error %d.", result);
5. **return** result;
6. }
8. result = mReaderThread->run("InputReader", PRIORITY\_URGENT\_DISPLAY);
9. **if** (result) {
10. LOGE("Could not start InputReader thread due to error %d.", result);
12. mDispatcherThread->requestExit();
13. **return** result;
14. }
16. **return** OK;
17. }

        这个函数主要就是分别启动一个InputDispatcherThread线程和一个InputReaderThread线程来读取和分发键盘消息的了。这里的InputDispatcherThread线程对象mDispatcherThread和InputReaderThread线程对象是在前面的Step 9中创建的，调用了它们的run函数后，就会进入到它们的threadLoop函数中去，只要threadLoop函数返回true，函数threadLoop就会一直被循环调用，于是这两个线程就起到了不断地读取和分发键盘消息的作用。

        我们先来分析InputDispatcherThread线程分发消息的过程，然后再回过头来分析InputReaderThread线程读取消息的过程。

        Step 13. InputDispatcherThread.threadLoop

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** InputDispatcherThread::threadLoop() {
2. mDispatcher->dispatchOnce();
3. **return** **true**;
4. }

        这里的成员变量mDispatcher即为在前面Step 8中创建的InputDispatcher对象，调用它的dispatchOnce成员函数执行一次键盘消息分发的操作。

        Step 14. InputDispatcher.dispatchOnce

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::dispatchOnce() {
2. nsecs\_t keyRepeatTimeout = mPolicy->getKeyRepeatTimeout();
3. nsecs\_t keyRepeatDelay = mPolicy->getKeyRepeatDelay();
5. nsecs\_t nextWakeupTime = LONG\_LONG\_MAX;
6. { // acquire lock
7. AutoMutex \_l(mLock);
8. dispatchOnceInnerLocked(keyRepeatTimeout, keyRepeatDelay, & nextWakeupTime);
10. **if** (runCommandsLockedInterruptible()) {
11. nextWakeupTime = LONG\_LONG\_MIN;  // force next poll to wake up immediately
12. }
13. } // release lock
15. // Wait for callback or timeout or wake.  (make sure we round up, not down)
16. nsecs\_t currentTime = now();
17. int32\_t timeoutMillis;
18. **if** (nextWakeupTime > currentTime) {
19. uint64\_t timeout = uint64\_t(nextWakeupTime - currentTime);
20. timeout = (timeout + 999999LL) / 1000000LL;
21. timeoutMillis = timeout > INT\_MAX ? -1 : int32\_t(timeout);
22. } **else** {
23. timeoutMillis = 0;
24. }
26. mLooper->pollOnce(timeoutMillis);
27. }

        这个函数很简单，把键盘消息交给dispatchOnceInnerLocked函数来处理，这个过程我们在后面再详细分析，然后调用mLooper->pollOnce函数等待下一次键盘事件的发生。这里的成员变量mLooper的类型为Looper，它定义在C++层中，具体可以参考前面[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)一文。

        Step 15. Looper.pollOnce

        这个函数定义在frameworks/base/libs/utils/Looper.cpp文件中，具体可以参考前面[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)一文，这里就不再详述了。总的来说，就是在Looper类中，会创建一个管道，当调用Looper类的pollOnce函数时，如果管道中没有内容可读，那么当前线程就会进入到空闲等待状态；当有键盘事件发生时，InputReader就会往这个管道中写入新的内容，这样就会唤醒前面正在等待键盘事件发生的线程。

        InputDispatcher类分发消息的过程就暂时分析到这里，后面会有更进一步的分析，现在，我们回到Step 12中，接着分析InputReader类读取键盘事件的过程。在调用了InputReaderThread线程类的run就函数后，同样会进入到InputReaderThread线程类的threadLoop函数中去。

        Step 16. InputReaderThread.threadLoop

        这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** InputReaderThread::threadLoop() {
2. mReader->loopOnce();
3. **return** **true**;
4. }

       这里的成员变量mReader即为在前面Step 8中创建的InputReader对象，调用它的loopOnce成员函数执行一次键盘事件的读取操作。

       Step 17. InputReader.loopOnce

       这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputReader::loopOnce() {
2. RawEvent rawEvent;
3. mEventHub->getEvent(& rawEvent);
5. #if DEBUG\_RAW\_EVENTS
6. LOGD("Input event: device=0x%x type=0x%x scancode=%d keycode=%d value=%d",
7. rawEvent.deviceId, rawEvent.type, rawEvent.scanCode, rawEvent.keyCode,
8. rawEvent.value);
9. #endif
11. process(& rawEvent);
12. }

        这里通过成员函数mEventHub来负责键盘消息的读取工作，如果当前有键盘事件发生或者有键盘事件等待处理，通过mEventHub的getEvent函数就可以得到这个事件，然后交给process函数进行处理，这个函数主要就是唤醒前面的InputDispatcherThread线程，通知它有新的键盘事件发生了，它需要进行一次键盘消息的分发操作了，这个函数我们后面再进一步详细分析；如果没有键盘事件发生或者没有键盘事件等待处理，那么调用mEventHub的getEvent函数时就会进入等待状态。

        Step 18. EventHub.getEvent

        这个函数定义在frameworks/base/libs/ui/EventHub.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** EventHub::getEvent(RawEvent\* outEvent)
2. {
3. outEvent->deviceId = 0;
4. outEvent->type = 0;
5. outEvent->scanCode = 0;
6. outEvent->keyCode = 0;
7. outEvent->flags = 0;
8. outEvent->value = 0;
9. outEvent->when = 0;
11. // Note that we only allow one caller to getEvent(), so don't need
12. // to do locking here...  only when adding/removing devices.
14. **if** (!mOpened) {
15. mError = openPlatformInput() ? NO\_ERROR : UNKNOWN\_ERROR;
16. mOpened = **true**;
17. mNeedToSendFinishedDeviceScan = **true**;
18. }
20. **for** (;;) {
21. // Report any devices that had last been added/removed.
22. **if** (mClosingDevices != NULL) {
23. device\_t\* device = mClosingDevices;
24. LOGV("Reporting device closed: id=0x%x, name=%s\n",
25. device->id, device->path.string());
26. mClosingDevices = device->next;
27. **if** (device->id == mFirstKeyboardId) {
28. outEvent->deviceId = 0;
29. } **else** {
30. outEvent->deviceId = device->id;
31. }
32. outEvent->type = DEVICE\_REMOVED;
33. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
34. **delete** device;
35. mNeedToSendFinishedDeviceScan = **true**;
36. **return** **true**;
37. }
39. **if** (mOpeningDevices != NULL) {
40. device\_t\* device = mOpeningDevices;
41. LOGV("Reporting device opened: id=0x%x, name=%s\n",
42. device->id, device->path.string());
43. mOpeningDevices = device->next;
44. **if** (device->id == mFirstKeyboardId) {
45. outEvent->deviceId = 0;
46. } **else** {
47. outEvent->deviceId = device->id;
48. }
49. outEvent->type = DEVICE\_ADDED;
50. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
51. mNeedToSendFinishedDeviceScan = **true**;
52. **return** **true**;
53. }
55. **if** (mNeedToSendFinishedDeviceScan) {
56. mNeedToSendFinishedDeviceScan = **false**;
57. outEvent->type = FINISHED\_DEVICE\_SCAN;
58. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
59. **return** **true**;
60. }
62. // Grab the next input event.
63. **for** (;;) {
64. // Consume buffered input events, if any.
65. **if** (mInputBufferIndex < mInputBufferCount) {
66. **const** **struct** input\_event& iev = mInputBufferData[mInputBufferIndex++];
67. **const** device\_t\* device = mDevices[mInputDeviceIndex];
69. LOGV("%s got: t0=%d, t1=%d, type=%d, code=%d, v=%d", device->path.string(),
70. (**int**) iev.time.tv\_sec, (**int**) iev.time.tv\_usec, iev.type, iev.code, iev.value);
71. **if** (device->id == mFirstKeyboardId) {
72. outEvent->deviceId = 0;
73. } **else** {
74. outEvent->deviceId = device->id;
75. }
76. outEvent->type = iev.type;
77. outEvent->scanCode = iev.code;
78. **if** (iev.type == EV\_KEY) {
79. status\_t err = device->layoutMap->map(iev.code,
80. & outEvent->keyCode, & outEvent->flags);
81. LOGV("iev.code=%d keyCode=%d flags=0x%08x err=%d\n",
82. iev.code, outEvent->keyCode, outEvent->flags, err);
83. **if** (err != 0) {
84. outEvent->keyCode = AKEYCODE\_UNKNOWN;
85. outEvent->flags = 0;
86. }
87. } **else** {
88. outEvent->keyCode = iev.code;
89. }
90. outEvent->value = iev.value;
92. // Use an event timestamp in the same timebase as
93. // java.lang.System.nanoTime() and android.os.SystemClock.uptimeMillis()
94. // as expected by the rest of the system.
95. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
96. **return** **true**;
97. }
99. // Finish reading all events from devices identified in previous poll().
100. // This code assumes that mInputDeviceIndex is initially 0 and that the
101. // revents member of pollfd is initialized to 0 when the device is first added.
102. // Since mFDs[0] is used for inotify, we process regular events starting at index 1.
103. mInputDeviceIndex += 1;
104. **if** (mInputDeviceIndex >= mFDCount) {
105. **break**;
106. }
108. **const** **struct** pollfd& pfd = mFDs[mInputDeviceIndex];
109. **if** (pfd.revents & POLLIN) {
110. int32\_t readSize = read(pfd.fd, mInputBufferData,
111. **sizeof**(**struct** input\_event) \* INPUT\_BUFFER\_SIZE);
112. **if** (readSize < 0) {
113. **if** (errno != EAGAIN && errno != EINTR) {
114. LOGW("could not get event (errno=%d)", errno);
115. }
116. } **else** **if** ((readSize % **sizeof**(**struct** input\_event)) != 0) {
117. LOGE("could not get event (wrong size: %d)", readSize);
118. } **else** {
119. mInputBufferCount = readSize / **sizeof**(**struct** input\_event);
120. mInputBufferIndex = 0;
121. }
122. }
123. }
125. ......
127. mInputDeviceIndex = 0;
129. // Poll for events.  Mind the wake lock dance!
130. // We hold a wake lock at all times except during poll().  This works due to some
131. // subtle choreography.  When a device driver has pending (unread) events, it acquires
132. // a kernel wake lock.  However, once the last pending event has been read, the device
133. // driver will release the kernel wake lock.  To prevent the system from going to sleep
134. // when this happens, the EventHub holds onto its own user wake lock while the client
135. // is processing events.  Thus the system can only sleep if there are no events
136. // pending or currently being processed.
137. release\_wake\_lock(WAKE\_LOCK\_ID);
139. **int** pollResult = poll(mFDs, mFDCount, -1);
141. acquire\_wake\_lock(PARTIAL\_WAKE\_LOCK, WAKE\_LOCK\_ID);
143. **if** (pollResult <= 0) {
144. **if** (errno != EINTR) {
145. LOGW("poll failed (errno=%d)\n", errno);
146. usleep(100000);
147. }
148. }
150. }
151. }

        这个函数比较长，我们一步一步来分析。

        首先，如果是第一次进入到这个函数中时，成员变量mOpened的值为false，于是就会调用openPlatformInput函数来打开系统输入设备，在本文中，我们主要讨论的输入设备就是键盘了。打开了这些输入设备文件后，就可以对这些输入设备进行是监控了。如果不是第一次进入到这个函数，那么就会分析当前有没有输入事件发生，如果有，就返回这个事件，否则就会进入等待状态，等待下一次输入事件的发生。在我们这个场景中，就是等待下一次键盘事件的发生了。

       我们先分析openPlatformInput函数的实现，然后回过头来分析这个getEvent函数的具体的实现。

       Step 19. EventHub.openPlatformInput

       这个函数定义在frameworks/base/libs/ui/EventHub.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** EventHub::openPlatformInput(**void**)
2. {
3. ......
5. res = scanDir(device\_path);
6. **if**(res < 0) {
7. LOGE("scan dir failed for %s\n", device\_path);
8. }
10. **return** **true**;
11. }

        这个函数主要是扫描device\_path目录下的设备文件，然后打开它们，这里的变量device\_path定义在frameworks/base/libs/ui/EventHub.cpp文件开始的地方：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **const** **char** \*device\_path = "/dev/input";

        在设备目录/dev/input中，一般有三个设备文件存在，分别是event0、mice和mouse0设备文件，其中，键盘事件就包含在event0设备文件中了。

        Step 20. EventHub.scanDir

        这个函数定义在frameworks/base/libs/ui/EventHub.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **int** EventHub::scanDir(**const** **char** \*dirname)
2. {
3. **char** devname[PATH\_MAX];
4. **char** \*filename;
5. DIR \*dir;
6. **struct** dirent \*de;
7. dir = opendir(dirname);
8. **if**(dir == NULL)
9. **return** -1;
10. strcpy(devname, dirname);
11. filename = devname + strlen(devname);
12. \*filename++ = '/';
13. **while**((de = readdir(dir))) {
14. **if**(de->d\_name[0] == '.' &&
15. (de->d\_name[1] == '\0' ||
16. (de->d\_name[1] == '.' && de->d\_name[2] == '\0')))
17. **continue**;
18. strcpy(filename, de->d\_name);
19. openDevice(devname);
20. }
21. closedir(dir);
22. **return** 0;
23. }

       根据上面一步的分析，这个函数主要就是调用openDevice函数来分别打开/dev/input/event0、/dev/input/mice和/dev/input/mouse0三个设备文件了。

       Step 21. EventHub.openDevice  
       这个函数定义在frameworks/base/libs/ui/EventHub.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **int** EventHub::openDevice(**const** **char** \*deviceName) {
2. **int** version;
3. **int** fd;
4. **struct** pollfd \*new\_mFDs;
5. device\_t \*\*new\_devices;
6. **char** \*\*new\_device\_names;
7. **char** name[80];
8. **char** location[80];
9. **char** idstr[80];
10. **struct** input\_id id;
12. LOGV("Opening device: %s", deviceName);
14. AutoMutex \_l(mLock);
16. fd = open(deviceName, O\_RDWR);
17. **if**(fd < 0) {
18. LOGE("could not open %s, %s\n", deviceName, strerror(errno));
19. **return** -1;
20. }
22. ......
24. **int** devid = 0;
25. **while** (devid < mNumDevicesById) {
26. **if** (mDevicesById[devid].device == NULL) {
27. **break**;
28. }
29. devid++;
30. }
32. ......
34. mDevicesById[devid].seq = (mDevicesById[devid].seq+(1<<SEQ\_SHIFT))&SEQ\_MASK;
35. **if** (mDevicesById[devid].seq == 0) {
36. mDevicesById[devid].seq = 1<<SEQ\_SHIFT;
37. }
39. new\_mFDs = (pollfd\*)realloc(mFDs, **sizeof**(mFDs[0]) \* (mFDCount + 1));
40. new\_devices = (device\_t\*\*)realloc(mDevices, **sizeof**(mDevices[0]) \* (mFDCount + 1));
41. **if** (new\_mFDs == NULL || new\_devices == NULL) {
42. LOGE("out of memory");
43. **return** -1;
44. }
45. mFDs = new\_mFDs;
46. mDevices = new\_devices;
48. ......
50. device\_t\* device = **new** device\_t(devid|mDevicesById[devid].seq, deviceName, name);
51. **if** (device == NULL) {
52. LOGE("out of memory");
53. **return** -1;
54. }
56. device->fd = fd;
57. mFDs[mFDCount].fd = fd;
58. mFDs[mFDCount].events = POLLIN;
59. mFDs[mFDCount].revents = 0;
61. // Figure out the kinds of events the device reports.
63. uint8\_t key\_bitmask[sizeof\_bit\_array(KEY\_MAX + 1)];
64. memset(key\_bitmask, 0, **sizeof**(key\_bitmask));
66. LOGV("Getting keys...");
67. **if** (ioctl(fd, EVIOCGBIT(EV\_KEY, **sizeof**(key\_bitmask)), key\_bitmask) >= 0) {
68. // See if this is a keyboard.  Ignore everything in the button range except for
69. // gamepads which are also considered keyboards.
70. **if** (containsNonZeroByte(key\_bitmask, 0, sizeof\_bit\_array(BTN\_MISC))
71. || containsNonZeroByte(key\_bitmask, sizeof\_bit\_array(BTN\_GAMEPAD),
72. sizeof\_bit\_array(BTN\_DIGI))
73. || containsNonZeroByte(key\_bitmask, sizeof\_bit\_array(KEY\_OK),
74. sizeof\_bit\_array(KEY\_MAX + 1))) {
75. device->classes |= INPUT\_DEVICE\_CLASS\_KEYBOARD;
77. device->keyBitmask = **new** uint8\_t[**sizeof**(key\_bitmask)];
78. **if** (device->keyBitmask != NULL) {
79. memcpy(device->keyBitmask, key\_bitmask, **sizeof**(key\_bitmask));
80. } **else** {
81. **delete** device;
82. LOGE("out of memory allocating key bitmask");
83. **return** -1;
84. }
85. }
86. }
88. ......
90. **if** ((device->classes & INPUT\_DEVICE\_CLASS\_KEYBOARD) != 0) {
91. **char** tmpfn[**sizeof**(name)];
92. **char** keylayoutFilename[300];
94. // a more descriptive name
95. device->name = name;
97. // replace all the spaces with underscores
98. strcpy(tmpfn, name);
99. **for** (**char** \*p = strchr(tmpfn, ' '); p && \*p; p = strchr(tmpfn, ' '))
100. \*p = '\_';
102. // find the .kl file we need for this device
103. **const** **char**\* root = getenv("ANDROID\_ROOT");
104. snprintf(keylayoutFilename, **sizeof**(keylayoutFilename),
105. "%s/usr/keylayout/%s.kl", root, tmpfn);
106. **bool** defaultKeymap = **false**;
107. **if** (access(keylayoutFilename, R\_OK)) {
108. snprintf(keylayoutFilename, **sizeof**(keylayoutFilename),
109. "%s/usr/keylayout/%s", root, "qwerty.kl");
110. defaultKeymap = **true**;
111. }
112. status\_t status = device->layoutMap->load(keylayoutFilename);
113. **if** (status) {
114. LOGE("Error %d loading key layout.", status);
115. }
117. // tell the world about the devname (the descriptive name)
118. **if** (!mHaveFirstKeyboard && !defaultKeymap && strstr(name, "-keypad")) {
119. // the built-in keyboard has a well-known device ID of 0,
120. // this device better not go away.
121. mHaveFirstKeyboard = **true**;
122. mFirstKeyboardId = device->id;
123. property\_set("hw.keyboards.0.devname", name);
124. } **else** {
125. // ensure mFirstKeyboardId is set to -something-.
126. **if** (mFirstKeyboardId == 0) {
127. mFirstKeyboardId = device->id;
128. }
129. }
130. **char** propName[100];
131. sprintf(propName, "hw.keyboards.%u.devname", device->id);
132. property\_set(propName, name);
134. // 'Q' key support = cheap test of whether this is an alpha-capable kbd
135. **if** (hasKeycodeLocked(device, AKEYCODE\_Q)) {
136. device->classes |= INPUT\_DEVICE\_CLASS\_ALPHAKEY;
137. }
139. // See if this device has a DPAD.
140. **if** (hasKeycodeLocked(device, AKEYCODE\_DPAD\_UP) &&
141. hasKeycodeLocked(device, AKEYCODE\_DPAD\_DOWN) &&
142. hasKeycodeLocked(device, AKEYCODE\_DPAD\_LEFT) &&
143. hasKeycodeLocked(device, AKEYCODE\_DPAD\_RIGHT) &&
144. hasKeycodeLocked(device, AKEYCODE\_DPAD\_CENTER)) {
145. device->classes |= INPUT\_DEVICE\_CLASS\_DPAD;
146. }
148. // See if this device has a gamepad.
149. **for** (**size\_t** i = 0; i < **sizeof**(GAMEPAD\_KEYCODES)/**sizeof**(GAMEPAD\_KEYCODES[0]); i++) {
150. **if** (hasKeycodeLocked(device, GAMEPAD\_KEYCODES[i])) {
151. device->classes |= INPUT\_DEVICE\_CLASS\_GAMEPAD;
152. **break**;
153. }
154. }
156. LOGI("New keyboard: device->id=0x%x devname='%s' propName='%s' keylayout='%s'\n",
157. device->id, name, propName, keylayoutFilename);
158. }
160. ......
162. mDevicesById[devid].device = device;
163. device->next = mOpeningDevices;
164. mOpeningDevices = device;
165. mDevices[mFDCount] = device;
167. mFDCount++;
168. **return** 0;
169. }

         函数首先根据文件名来打开这个设备文件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. fd = open(deviceName, O\_RDWR);

        系统中所有输入设备文件信息都保存在成员变量mDevicesById中，因此，先在mDevicesById找到一个空位置来保存当前打开的设备文件信息：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mDevicesById[devid].seq = (mDevicesById[devid].seq+(1<<SEQ\_SHIFT))&SEQ\_MASK;
2. **if** (mDevicesById[devid].seq == 0) {
3. mDevicesById[devid].seq = 1<<SEQ\_SHIFT;
4. }

        找到了空闲位置后，就为这个输入设备文件创建相应的device\_t信息：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mDevicesById[devid].seq = (mDevicesById[devid].seq+(1<<SEQ\_SHIFT))&SEQ\_MASK;
2. **if** (mDevicesById[devid].seq == 0) {
3. mDevicesById[devid].seq = 1<<SEQ\_SHIFT;
4. }
6. new\_mFDs = (pollfd\*)realloc(mFDs, **sizeof**(mFDs[0]) \* (mFDCount + 1));
7. new\_devices = (device\_t\*\*)realloc(mDevices, **sizeof**(mDevices[0]) \* (mFDCount + 1));
8. **if** (new\_mFDs == NULL || new\_devices == NULL) {
9. LOGE("out of memory");
10. **return** -1;
11. }
12. mFDs = new\_mFDs;
13. mDevices = new\_devices;
15. ......
17. device\_t\* device = **new** device\_t(devid|mDevicesById[devid].seq, deviceName, name);
18. **if** (device == NULL) {
19. LOGE("out of memory");
20. **return** -1;
21. }
23. device->fd = fd;

        同时，这个设备文件还会保存在数组mFDs中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mFDs[mFDCount].fd = fd;
2. mFDs[mFDCount].events = POLLIN;
3. mFDs[mFDCount].revents = 0;

       接下来查看这个设备是不是键盘：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Figure out the kinds of events the device reports.
3. uint8\_t key\_bitmask[sizeof\_bit\_array(KEY\_MAX + 1)];
4. memset(key\_bitmask, 0, **sizeof**(key\_bitmask));
6. LOGV("Getting keys...");
7. **if** (ioctl(fd, EVIOCGBIT(EV\_KEY, **sizeof**(key\_bitmask)), key\_bitmask) >= 0) {
8. // See if this is a keyboard.  Ignore everything in the button range except for
9. // gamepads which are also considered keyboards.
10. **if** (containsNonZeroByte(key\_bitmask, 0, sizeof\_bit\_array(BTN\_MISC))
11. || containsNonZeroByte(key\_bitmask, sizeof\_bit\_array(BTN\_GAMEPAD),
12. sizeof\_bit\_array(BTN\_DIGI))
13. || containsNonZeroByte(key\_bitmask, sizeof\_bit\_array(KEY\_OK),
14. sizeof\_bit\_array(KEY\_MAX + 1))) {
15. device->classes |= INPUT\_DEVICE\_CLASS\_KEYBOARD;
17. device->keyBitmask = **new** uint8\_t[**sizeof**(key\_bitmask)];
18. **if** (device->keyBitmask != NULL) {
19. memcpy(device->keyBitmask, key\_bitmask, **sizeof**(key\_bitmask));
20. } **else** {
21. **delete** device;
22. LOGE("out of memory allocating key bitmask");
23. **return** -1;
24. }
25. }
26. }

        如果是的话，还要继续进一步初始化前面为这个设备文件所创建的device\_t结构体，主要就是把结构体device的classes成员变量的INPUT\_DEVICE\_CLASS\_KEYBOARD位置为1了，以表明这是一个键盘。  
        如果是键盘设备，初始化工作还未完成，还要继续设置键盘的布局等信息：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** ((device->classes & INPUT\_DEVICE\_CLASS\_KEYBOARD) != 0) {
2. **char** tmpfn[**sizeof**(name)];
3. **char** keylayoutFilename[300];
5. // a more descriptive name
6. device->name = name;
8. // replace all the spaces with underscores
9. strcpy(tmpfn, name);
10. **for** (**char** \*p = strchr(tmpfn, ' '); p && \*p; p = strchr(tmpfn, ' '))
11. \*p = '\_';
13. // find the .kl file we need for this device
14. **const** **char**\* root = getenv("ANDROID\_ROOT");
15. snprintf(keylayoutFilename, **sizeof**(keylayoutFilename),
16. "%s/usr/keylayout/%s.kl", root, tmpfn);
17. **bool** defaultKeymap = **false**;
18. **if** (access(keylayoutFilename, R\_OK)) {
19. snprintf(keylayoutFilename, **sizeof**(keylayoutFilename),
20. "%s/usr/keylayout/%s", root, "qwerty.kl");
21. defaultKeymap = **true**;
22. }
23. status\_t status = device->layoutMap->load(keylayoutFilename);
24. **if** (status) {
25. LOGE("Error %d loading key layout.", status);
26. }
28. // tell the world about the devname (the descriptive name)
29. **if** (!mHaveFirstKeyboard && !defaultKeymap && strstr(name, "-keypad")) {
30. // the built-in keyboard has a well-known device ID of 0,
31. // this device better not go away.
32. mHaveFirstKeyboard = **true**;
33. mFirstKeyboardId = device->id;
34. property\_set("hw.keyboards.0.devname", name);
35. } **else** {
36. // ensure mFirstKeyboardId is set to -something-.
37. **if** (mFirstKeyboardId == 0) {
38. mFirstKeyboardId = device->id;
39. }
40. }
41. **char** propName[100];
42. sprintf(propName, "hw.keyboards.%u.devname", device->id);
43. property\_set(propName, name);
45. // 'Q' key support = cheap test of whether this is an alpha-capable kbd
46. **if** (hasKeycodeLocked(device, AKEYCODE\_Q)) {
47. device->classes |= INPUT\_DEVICE\_CLASS\_ALPHAKEY;
48. }
50. // See if this device has a DPAD.
51. **if** (hasKeycodeLocked(device, AKEYCODE\_DPAD\_UP) &&
52. hasKeycodeLocked(device, AKEYCODE\_DPAD\_DOWN) &&
53. hasKeycodeLocked(device, AKEYCODE\_DPAD\_LEFT) &&
54. hasKeycodeLocked(device, AKEYCODE\_DPAD\_RIGHT) &&
55. hasKeycodeLocked(device, AKEYCODE\_DPAD\_CENTER)) {
56. device->classes |= INPUT\_DEVICE\_CLASS\_DPAD;
57. }
59. // See if this device has a gamepad.
60. **for** (**size\_t** i = 0; i < **sizeof**(GAMEPAD\_KEYCODES)/**sizeof**(GAMEPAD\_KEYCODES[0]); i++) {
61. **if** (hasKeycodeLocked(device, GAMEPAD\_KEYCODES[i])) {
62. device->classes |= INPUT\_DEVICE\_CLASS\_GAMEPAD;
63. **break**;
64. }
65. }
67. LOGI("New keyboard: device->id=0x%x devname='%s' propName='%s' keylayout='%s'\n",
68. device->id, name, propName, keylayoutFilename);
69. }

        到这里，系统中的输入设备文件就打开了。

        回到Step 18中，我们继续分析EventHub.getEvent函数的实现。

        在中间的for循环里面，首先会检查当前是否有输入设备被关闭，如果有，就返回一个设备移除的事件给调用方：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Report any devices that had last been added/removed.
2. **if** (mClosingDevices != NULL) {
3. device\_t\* device = mClosingDevices;
4. LOGV("Reporting device closed: id=0x%x, name=%s\n",
5. device->id, device->path.string());
6. mClosingDevices = device->next;
7. **if** (device->id == mFirstKeyboardId) {
8. outEvent->deviceId = 0;
9. } **else** {
10. outEvent->deviceId = device->id;
11. }
12. outEvent->type = DEVICE\_REMOVED;
13. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
14. **delete** device;
15. mNeedToSendFinishedDeviceScan = **true**;
16. **return** **true**;
17. }

        接着，检查当前是否有新的输入设备加入进来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** (mOpeningDevices != NULL) {
2. device\_t\* device = mOpeningDevices;
3. LOGV("Reporting device opened: id=0x%x, name=%s\n",
4. device->id, device->path.string());
5. mOpeningDevices = device->next;
6. **if** (device->id == mFirstKeyboardId) {
7. outEvent->deviceId = 0;
8. } **else** {
9. outEvent->deviceId = device->id;
10. }
11. outEvent->type = DEVICE\_ADDED;
12. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
13. mNeedToSendFinishedDeviceScan = **true**;
14. **return** **true**;
15. }

        接着，再检查是否需要结束监控输入事件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** (mNeedToSendFinishedDeviceScan) {
2. mNeedToSendFinishedDeviceScan = **false**;
3. outEvent->type = FINISHED\_DEVICE\_SCAN;
4. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
5. **return** **true**;
6. }

        最后，就是要检查当前是否有还未处理的输入设备事件发生了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Grab the next input event.
2. **for** (;;) {
3. // Consume buffered input events, if any.
4. **if** (mInputBufferIndex < mInputBufferCount) {
5. **const** **struct** input\_event& iev = mInputBufferData[mInputBufferIndex++];
6. **const** device\_t\* device = mDevices[mInputDeviceIndex];
8. LOGV("%s got: t0=%d, t1=%d, type=%d, code=%d, v=%d", device->path.string(),
9. (**int**) iev.time.tv\_sec, (**int**) iev.time.tv\_usec, iev.type, iev.code, iev.value);
10. **if** (device->id == mFirstKeyboardId) {
11. outEvent->deviceId = 0;
12. } **else** {
13. outEvent->deviceId = device->id;
14. }
15. outEvent->type = iev.type;
16. outEvent->scanCode = iev.code;
17. **if** (iev.type == EV\_KEY) {
18. status\_t err = device->layoutMap->map(iev.code,
19. & outEvent->keyCode, & outEvent->flags);
20. LOGV("iev.code=%d keyCode=%d flags=0x%08x err=%d\n",
21. iev.code, outEvent->keyCode, outEvent->flags, err);
22. **if** (err != 0) {
23. outEvent->keyCode = AKEYCODE\_UNKNOWN;
24. outEvent->flags = 0;
25. }
26. } **else** {
27. outEvent->keyCode = iev.code;
28. }
29. outEvent->value = iev.value;
31. // Use an event timestamp in the same timebase as
32. // java.lang.System.nanoTime() and android.os.SystemClock.uptimeMillis()
33. // as expected by the rest of the system.
34. outEvent->when = systemTime(SYSTEM\_TIME\_MONOTONIC);
35. **return** **true**;
36. }
38. // Finish reading all events from devices identified in previous poll().
39. // This code assumes that mInputDeviceIndex is initially 0 and that the
40. // revents member of pollfd is initialized to 0 when the device is first added.
41. // Since mFDs[0] is used for inotify, we process regular events starting at index 1.
42. mInputDeviceIndex += 1;
43. **if** (mInputDeviceIndex >= mFDCount) {
44. **break**;
45. }
47. **const** **struct** pollfd& pfd = mFDs[mInputDeviceIndex];
48. **if** (pfd.revents & POLLIN) {
49. int32\_t readSize = read(pfd.fd, mInputBufferData,
50. **sizeof**(**struct** input\_event) \* INPUT\_BUFFER\_SIZE);
51. **if** (readSize < 0) {
52. **if** (errno != EAGAIN && errno != EINTR) {
53. LOGW("could not get event (errno=%d)", errno);
54. }
55. } **else** **if** ((readSize % **sizeof**(**struct** input\_event)) != 0) {
56. LOGE("could not get event (wrong size: %d)", readSize);
57. } **else** {
58. mInputBufferCount = readSize / **sizeof**(**struct** input\_event);
59. mInputBufferIndex = 0;
60. }
61. }
62. }

        未处理的输入事件保存在成员变量mInputBufferData中，如果有的话，就可以直接返回了，否则的话，就要通过系统调用poll来等待输入设备上发生新的事件了，在我们这个场景中，就是等待键盘有键被按下或者松开了。：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **int** pollResult = poll(mFDs, mFDCount, -1);

        这里的mFDs包含了我们所要监控的输入设备的打开文件描述符，这是在前面的openPlatformInput函数中初始化的。

        Step 22. poll

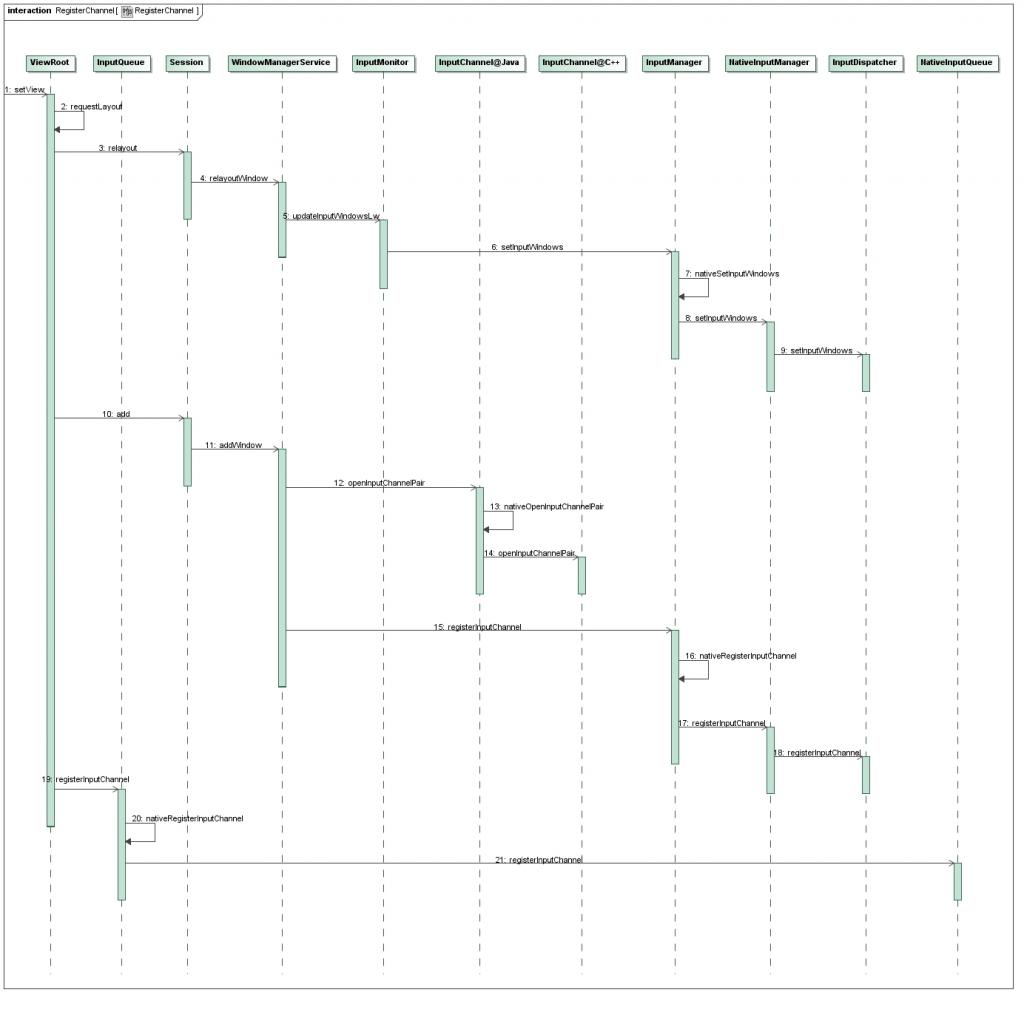
        这是一个Linux系统的文件操作系统调用，它用来查询指定的文件列表是否有有可读写的，如果有，就马上返回，否则的话，就阻塞线程，并等待驱动程序唤醒，重新调用poll函数，或超时返回。在我们的这个场景中，就是要查询是否有键盘事件发生，如果有的话，就返回，否则的话，当前线程就睡眠等待键盘事件的发生了。

        这样，InputManager的启动过程就分析完了，下面我们再分析应用程序注册键盘消息接收通道的过程。

        2. 应用程序注册键盘消息接收通道的过程分析

        InputManager启动以后，就开始负责监控键盘输入事件了。当InputManager监控到键盘输入事件时，它应该把这个键盘事件分发给谁呢？当然是要把这个键盘消息分发给当前激活的Activity窗口了，不过，当前激活的Activity窗口还需要主动注册一个键盘消息接收通道到InputManager中去，InputManager才能把这个键盘消息分发给它处理。那么，当前被激活的Activity窗口又是什么时候去注册这个键盘消息接收通道的呢？在前面一篇文章Android应用程序启动过程源代码分析中，我们分析Android应用程序的启动过程时，在Step 33中分析到ActivityThread类的handleLaunchActivity函数中，我们曾经说过，当函数handleLaunchActivity调用performLaunchActivity函数来加载这个完毕应用程序的默认Activity后，再次回到handleLaunchActivity函数时，会调用handleResumeActivity函数来使这个Activity进入Resumed状态。在调用handleResumeActivity函数的过程中，ActivityThread会通过android.view.WindowManagerImpl类为该Activity创建一个ViewRoot实例，并且会通过调用ViewRoot类的setView成员函数把与该Activity关联的View设置到这个ViewRoot中去，而Activity正是通过ViewRoot类的setView成员函数来注册键盘消息接收通道的。

        有了这些背影知识后，接下来，我们就可以从ViewRoot.setView函数开始分析应用程序注册键盘消息接收通道的过程了。首先看一下这个注册过程的序列图，然后再详细分析每一个步骤：



[点击查看大图](http://hi.csdn.net/attachment/201110/23/0_1319372058cnwW.gif)

        Step 1. ViewRoot.setView

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **public** **void** setView(View view, WindowManager.LayoutParams attrs,
6. View panelParentView) {
7. ......
9. **synchronized** (**this**) {
10. **if** (mView == **null**) {
11. ......
13. // Schedule the first layout -before- adding to the window
14. // manager, to make sure we do the relayout before receiving
15. // any other events from the system.
16. requestLayout();
17. mInputChannel = **new** InputChannel();
18. **try** {
19. res = sWindowSession.add(mWindow, mWindowAttributes,
20. getHostVisibility(), mAttachInfo.mContentInsets,
21. mInputChannel);
22. } **catch** (RemoteException e) {
23. ......
24. } **finally** {
25. ......
26. }
28. ......
30. **if** (view **instanceof** RootViewSurfaceTaker) {
31. mInputQueueCallback =
32. ((RootViewSurfaceTaker)view).willYouTakeTheInputQueue();
33. }
34. **if** (mInputQueueCallback != **null**) {
35. mInputQueue = **new** InputQueue(mInputChannel);
36. mInputQueueCallback.onInputQueueCreated(mInputQueue);
37. } **else** {
38. InputQueue.registerInputChannel(mInputChannel, mInputHandler,
39. Looper.myQueue());
40. }
42. ......
43. }
44. }
45. }
47. }

        这个函数中与注册键盘消息接收通道（InputChannel）相关的逻辑主要有三处，一是调用requestLayout函数来通知InputManager，这个Activity窗口是当前被激活的窗口，二是调用sWindowSession（WindowManagerService内部类Session的远程接口）的add成员函数来把键盘消息接收通道的一端注册在InputManager中，三是调用InputQueue的registerInputChannel成员函数来把键盘消息接收通道的另一端注册在本应用程序的消息循环（Looper）中。这样，当InputManager监控到有键盘消息时，就会先找到当前被激活的窗口，然后找到其在InputManager中对应的键盘消息接收通道，通过这个通道在InputManager中的一端来通知在应用程序消息循环中的另一端，就把键盘消息分发给当前激活的Activity窗口了。

        在接下来的内容中，我们首先描述requestLayout函数是如何告诉InputManager当前的Activity窗口便是激活窗口的，接着再回过头来分析应用程序是如何把键盘消息接收通道的一端注册到InputManager中去的，最后分析应用程序是如何键盘消息接收通道的另一端注册到本应用程序的消息循环中去了。

        Step 2. ViewRoot.requestLayout

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **public** **void** requestLayout() {
6. ......
7. mLayoutRequested = **true**;
8. scheduleTraversals();
9. }
11. ......
12. }

        这个函数调用了scheduleTraversals函数来进一步执行操作，由于篇幅关系，我们就不详细描述scheduleTraversals函数了，简单来说，在scheduleTraversals函数中，会通过sendEmptyMessage(DO\_TRAVERSAL)发送一个消息到应用程序的消息队列中，这个消息最终由ViewRoot的handleMessage函数处理，而ViewRoot的handleMessage函数把这个消息交给ViewRoot类的performTraversals来处理，在performTraversals函数中，又会调用ViewRoot类的relayoutWindow函数来进一步执行操作，最后在relayoutWindow函数中，就会通过WindowManagerService内部类Session的远程接口sWindowSession的relayout函数来进入到WindowManagerService中。

        Step 3. WindowManagerService.Session.relayout

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java 文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **class** Session **extends** IWindowSession.Stub
6. **implements** IBinder.DeathRecipient {
7. ......
9. **public** **int** relayout(IWindow window, WindowManager.LayoutParams attrs,
10. **int** requestedWidth, **int** requestedHeight, **int** viewFlags,
11. **boolean** insetsPending, Rect outFrame, Rect outContentInsets,
12. Rect outVisibleInsets, Configuration outConfig, Surface outSurface) {
13. //Log.d(TAG, ">>>>>> ENTERED relayout from " + Binder.getCallingPid());
14. **int** res = relayoutWindow(**this**, window, attrs,
15. requestedWidth, requestedHeight, viewFlags, insetsPending,
16. outFrame, outContentInsets, outVisibleInsets, outConfig, outSurface);
17. //Log.d(TAG, "<<<<<< EXITING relayout to " + Binder.getCallingPid());
18. **return** res;
19. }
21. ......
22. }
24. ......
25. }

        这个函数只是简单地调用WindowManagerService的成员函数relayoutWIndow来进一步处理。

        Step 4. WindowManagerService.relayoutWIndow

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java 文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **int** relayoutWindow(Session session, IWindow client,
6. WindowManager.LayoutParams attrs, **int** requestedWidth,
7. **int** requestedHeight, **int** viewVisibility, **boolean** insetsPending,
8. Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,
9. Configuration outConfig, Surface outSurface) {
10. ......
12. **synchronized**(mWindowMap) {
13. ......
15. mInputMonitor.updateInputWindowsLw();
16. }
18. ......
19. }
21. ......
22. }

         这个函数又会继续调用mInputMonitor的updateInputWindowsLw成员函数来更新当前的输入窗口，mInputMonitor是WindowManagerService的成员变量，它的类型为InputMonitor。

         Step 5. InputMonitor.updateInputWindowsLw

         这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java 文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **final** **class** InputMonitor {
6. ......
8. /\* Updates the cached window information provided to the input dispatcher. \*/
9. **public** **void** updateInputWindowsLw() {
10. // Populate the input window list with information about all of the windows that
11. // could potentially receive input.
12. // As an optimization, we could try to prune the list of windows but this turns
13. // out to be difficult because only the native code knows for sure which window
14. // currently has touch focus.
15. **final** ArrayList<WindowState> windows = mWindows;
16. **final** **int** N = windows.size();
17. **for** (**int** i = N - 1; i >= 0; i--) {
18. **final** WindowState child = windows.get(i);
19. **if** (child.mInputChannel == **null** || child.mRemoved) {
20. // Skip this window because it cannot possibly receive input.
21. **continue**;
22. }
24. ......
26. // Add a window to our list of input windows.
27. **final** InputWindow inputWindow = mTempInputWindows.add();
29. ......
30. }
32. // Send windows to native code.
33. mInputManager.setInputWindows(mTempInputWindows.toNullTerminatedArray());
35. ......
36. }
38. ......
39. }
41. ......
42. }

        这个函数将当前系统中带有InputChannel的Activity窗口都设置为InputManager的输入窗口，但是后面我们会看到，只有当前激活的窗口才会响应键盘消息。

        Step 6. InputManager.setInputWindows

        这个函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. **public** **void** setInputWindows(InputWindow[] windows) {
5. nativeSetInputWindows(windows);
6. }
8. ......
9. }

        这个函数调用了本地方法nativeSetInputWindows来进一步执行操作。  
        Step 7. InputManager.nativeSetInputWindows

        这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_server\_InputManager\_nativeSetInputWindows(JNIEnv\* env, jclass clazz,
2. jobjectArray windowObjArray) {
3. **if** (checkInputManagerUnitialized(env)) {
4. **return**;
5. }
7. gNativeInputManager->setInputWindows(env, windowObjArray);
8. }

        这里的gNativeInputManager我们前面分析InputManager的启动过程时已经见过了，这是一个本地InputManager对象，通过它进一步设置当前系统的输入窗口。

        Step 8. NativeInputManager.setInputWindows

        这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** NativeInputManager::setInputWindows(JNIEnv\* env, jobjectArray windowObjArray) {
2. Vector<InputWindow> windows;
4. jsize length = env->GetArrayLength(windowObjArray);
5. **for** (jsize i = 0; i < length; i++) {
6. jobject inputTargetObj = env->GetObjectArrayElement(windowObjArray, i);
7. **if** (! inputTargetObj) {
8. **break**; // found null element indicating end of used portion of the array
9. }
11. windows.push();
12. InputWindow& window = windows.editTop();
13. **bool** valid = populateWindow(env, inputTargetObj, window);
14. **if** (! valid) {
15. windows.pop();
16. }
18. env->DeleteLocalRef(inputTargetObj);
19. }
21. mInputManager->getDispatcher()->setInputWindows(windows);
22. }

        这个函数首先将Java层的Window转换成C++层的InputWindow，然后放在windows向量中，最后将这些输入窗口设置到InputDispatcher中去。

        Step 9. InputDispatcher.setInputWindows

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::setInputWindows(**const** Vector<InputWindow>& inputWindows) {
2. ......
4. { // acquire lock
5. AutoMutex \_l(mLock);
7. // Clear old window pointers.
8. sp<InputChannel> oldFocusedWindowChannel;
9. **if** (mFocusedWindow) {
10. oldFocusedWindowChannel = mFocusedWindow->inputChannel;
11. mFocusedWindow = NULL;
12. }
14. mWindows.clear();
16. // Loop over new windows and rebuild the necessary window pointers for
17. // tracking focus and touch.
18. mWindows.appendVector(inputWindows);
20. **size\_t** numWindows = mWindows.size();
21. **for** (**size\_t** i = 0; i < numWindows; i++) {
22. **const** InputWindow\* window = & mWindows.itemAt(i);
23. **if** (window->hasFocus) {
24. mFocusedWindow = window;
25. **break**;
26. }
27. }
29. ......
31. } // release lock
33. ......
34. }

        这里InputDispatcher的成员变量mFocusedWindow就代表当前激活的窗口的。这个函数首先清空mFocusedWindow，然后再通过一个for循环检查当前的输入窗口中的哪一个窗口是获得焦点的，获得焦点的输入窗口即为当前激活的窗口。

        这样，InputManager就把当前激活的Activity窗口保存在InputDispatcher中了，后面就可以把键盘消息分发给它来处理。

        回到Step 1中的ViewRoot.setView函数中，接下来就调用下面语句来注册键盘消息接收通道的一端到InputManager中去：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mInputChannel = **new** InputChannel();
2. **try** {
3. res = sWindowSession.add(mWindow, mWindowAttributes,
4. getHostVisibility(), mAttachInfo.mContentInsets,
5. mInputChannel);
6. } **catch** (RemoteException e) {
7. ......
8. } **finally** {
9. ......
10. }

        前面说过，这里的sWindowSession是WindowManagerService内部类Session的一个远程接口，通过它可以进入到WindowManagerService中去。

        Step 10. WindowManagerService.Session.add

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java 文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **class** Session **extends** IWindowSession.Stub
6. **implements** IBinder.DeathRecipient {
7. ......
9. **public** **int** add(IWindow window, WindowManager.LayoutParams attrs,
10. **int** viewVisibility, Rect outContentInsets, InputChannel outInputChannel) {
11. **return** addWindow(**this**, window, attrs, viewVisibility, outContentInsets,
12. outInputChannel);
13. }
15. ......
16. }
18. ......
19. }

        这里调用WindowManagerService类的addWindow函数来进一步执行操作。

        Step 11. WindowManagerService.addWindow

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java 文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **int** addWindow(Session session, IWindow client,
6. WindowManager.LayoutParams attrs, **int** viewVisibility,
7. Rect outContentInsets, InputChannel outInputChannel) {
8. ......
10. WindowState win = **null**;
12. **synchronized**(mWindowMap) {
13. ......
15. win = **new** WindowState(session, client, token,
16. attachedWindow, attrs, viewVisibility);
18. ......
20. **if** (outInputChannel != **null**) {
21. String name = win.makeInputChannelName();
22. InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
23. win.mInputChannel = inputChannels[0];
24. inputChannels[1].transferToBinderOutParameter(outInputChannel);
25. mInputManager.registerInputChannel(win.mInputChannel);
26. }
28. ......
29. }
31. ......
32. }
34. ......
35. }

        这里的outInputChannel即为前面在Step 1中创建的InputChannel，它不为NULL，因此，这里会通过InputChannel.openInputChannelPair函数来创建一对输入通道，其中一个位于WindowManagerService中，另外一个通过outInputChannel参数返回到应用程序中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. inputChannels[1].transferToBinderOutParameter(outInputChannel);

        创建输入通道之前，WindowManagerService会为当前Activity窗口创建一个WindowState对象win，用来记录这个Activity窗口的状态信息。当创建这对输入管道成功以后，也会把其中的一个管道保存在这个WindowState对象win的成员变量mInputChannel中，后面要注销这个管道的时候，就是从这个WindownState对象中取回这个管道的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. win.mInputChannel = inputChannels[0];

        接下来我们就看一下InputChannel.openInputChannelPair函数的实现。

        Step 12. InputChannel.openInputChannelPair

        这个函数定义在frameworks/base/core/java/android/view/InputChannel.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** InputChannel **implements** Parcelable {
2. ......
4. /\*\*
5. \* Creates a new input channel pair.  One channel should be provided to the input
6. \* dispatcher and the other to the application's input queue.
7. \* @param name The descriptive (non-unique) name of the channel pair.
8. \* @return A pair of input channels.  They are symmetric and indistinguishable.
9. \*/
10. **public** **static** InputChannel[] openInputChannelPair(String name) {
11. ......
13. **return** nativeOpenInputChannelPair(name);
14. }
16. ......
17. }

         这个函数调用本地方法nativeOpenInputChannelPair来进一步执行操作。

         Step 13. InputChannel.nativeOpenInputChannelPair  
         这个函数定义在frameworks/base/core/jni/android\_view\_InputChannel.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

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. status\_t result = InputChannel::openInputChannelPair(name, serverChannel, clientChannel);
11. **if** (result) {
12. LOGE("Could not open input channel pair.  status=%d", result);
13. jniThrowRuntimeException(env, "Could not open input channel pair.");
14. **return** NULL;
15. }
17. // TODO more robust error checking
18. jobject serverChannelObj = android\_view\_InputChannel\_createInputChannel(env,
19. **new** NativeInputChannel(serverChannel));
20. jobject clientChannelObj = android\_view\_InputChannel\_createInputChannel(env,
21. **new** NativeInputChannel(clientChannel));
23. jobjectArray channelPair = env->NewObjectArray(2, gInputChannelClassInfo.clazz, NULL);
24. env->SetObjectArrayElement(channelPair, 0, serverChannelObj);
25. env->SetObjectArrayElement(channelPair, 1, clientChannelObj);
26. **return** channelPair;
27. }

        这个函数根据传进来的参数name在C++层分别创建两个InputChannel，一个作为Server端使用，一个作为Client端使用，这里的Server端即是指InputManager，而Client端即是指应用程序。这两个本地的InputChannel是通过InputChannel::openInputChannelPair函数创建的，创建完成后，再相应地在Java层创建相应的两个InputChannel，然后返回。

        Step 14. InputChannel.openInputChannelPair  
        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputChannel::openInputChannelPair(**const** String8& name,
2. sp<InputChannel>& outServerChannel, sp<InputChannel>& outClientChannel) {
3. status\_t result;
5. **int** serverAshmemFd = ashmem\_create\_region(name.string(), DEFAULT\_MESSAGE\_BUFFER\_SIZE);
6. **if** (serverAshmemFd < 0) {
7. ......
8. } **else** {
9. result = ashmem\_set\_prot\_region(serverAshmemFd, PROT\_READ | PROT\_WRITE);
10. **if** (result < 0) {
11. ......
12. } **else** {
13. // Dup the file descriptor because the server and client input channel objects that
14. // are returned may have different lifetimes but they share the same shared memory region.
15. **int** clientAshmemFd;
16. clientAshmemFd = dup(serverAshmemFd);
17. **if** (clientAshmemFd < 0) {
18. ......
19. } **else** {
20. **int** forward[2];
21. **if** (pipe(forward)) {
22. ......
23. } **else** {
24. **int** reverse[2];
25. **if** (pipe(reverse)) {
26. ......
27. } **else** {
28. String8 serverChannelName = name;
29. serverChannelName.append(" (server)");
30. outServerChannel = **new** InputChannel(serverChannelName,
31. serverAshmemFd, reverse[0], forward[1]);
33. String8 clientChannelName = name;
34. clientChannelName.append(" (client)");
35. outClientChannel = **new** InputChannel(clientChannelName,
36. clientAshmemFd, forward[0], reverse[1]);
37. **return** OK;
38. }
39. ......
40. }
41. ......
42. }
43. }
44. }
45. ......
46. }

        在阅读这个函数之前，我们首先了解一下C++层的InputChannel的构造函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. InputChannel::InputChannel(**const** String8& name, int32\_t ashmemFd, int32\_t receivePipeFd,
2. int32\_t sendPipeFd) :
3. mName(name), mAshmemFd(ashmemFd), mReceivePipeFd(receivePipeFd), mSendPipeFd(sendPipeFd) {
4. ......
5. }

        为了创建一个InputChannel，我们需要准备四个参数，一个是输入通道的名称name，一个是[匿名共享内存](http://blog.csdn.net/luoshengyang/article/details/6651971)文件描述符，一个是管道的读端文件描述符，一个是管道的写端文件描述符。在上面的openInputChannelPair函数，输入通道的名称已经作为参数传递进来，因此，还需要创建匿名共享内存文件，还有管道。这里需要创建两个管道，一个称为前向管道（forward pipe），一个称为反向管道（reverse pipe），它们交叉使用在Server端和Client端的InputChannel中，这样就使入Server和Client可以互相通信了。

        具体来说，Server端和Client端的InputChannel分别是这样构成的：

**Server Input Channel:  *ashmem - reverse(read) - forward(write)***

**Client Input Channel:   *ashmem - forward(read) - reverse(write)***  
        前面我们在[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)一文中学习Android应用程序的消息处理机制时知道，管道可以用作进程间通信，其中一个进程在管道的读端等待新的内空可读，另一个进程在管道的写端写入新的内容以唤醒在管道读端等待的进程，这样就实现了进程间通信。在我们这个情景中，Client端可以在前向管道（forward pipe）的读端睡眠等待新的内容可读，而Server端可以通过向前向管道（forward pipe）的写端写入新的内容来唤醒Client端，同样，把前向管道（forward pipe）换成反向管道（reverse pipe），也能实现Client端唤醒Server端。在后面我们分析InputDispatcher分发键盘消息时，会看到它们的用法。

        有了这些背景知识后，相信上面的openInputChannelPair的代码就容易理解了，这里就不再详述了。

        创建好了这两个输入通道后，回到Step 11中的WindowManagerService.addWindow函数中，一方面它把刚才创建的Client端的输入通道通过outInputChannel参数返回到应用程序中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. inputChannels[1].transferToBinderOutParameter(outInputChannel);

       另一方面，它还要把刚才创建的Server端的输入通道注册到InputManager中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mInputManager.registerInputChannel(win.mInputChannel);

       Step 15. InputManager.registerInputChannel

       这个函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. /\*\*
5. \* Registers an input channel so that it can be used as an input event target.
6. \* @param inputChannel The input channel to register.
7. \*/
8. **public** **void** registerInputChannel(InputChannel inputChannel) {
9. **if** (inputChannel == **null**) {
10. **throw** **new** IllegalArgumentException("inputChannel must not be null.");
11. }
13. nativeRegisterInputChannel(inputChannel, **false**);
14. }
16. ......
17. }

         它通过调用本地方法nativeRegisterInputChannel来执行进一步的操作。

         Step 16. InputManager.nativeRegisterInputChannel

         这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_server\_InputManager\_nativeRegisterInputChannel(JNIEnv\* env, jclass clazz,
2. jobject inputChannelObj, jboolean monitor) {
3. ......
5. sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,
6. inputChannelObj);
7. ......
9. status\_t status = gNativeInputManager->registerInputChannel(
10. env, inputChannel, inputChannelObj, monitor);
12. ......
13. }

        这里首先通过Java层的InputChannel对象获得C++层的InputChannel对象，它们之间的对应关系是在前面的Step 13中设置好的，接着调用NativeInputManager的registerInputChannel执行进一步的操作。

        Step 17. NativeInputManager.registerInputChannel

        这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp 文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t NativeInputManager::registerInputChannel(JNIEnv\* env,
2. **const** sp<InputChannel>& inputChannel, jobject inputChannelObj, **bool** monitor) {
3. ......
5. status = mInputManager->getDispatcher()->registerInputChannel(inputChannel, monitor);
7. ......
8. }

        这个函数主要是调用了InputDispatcher的registerInputChannel来真正执行注册输入通道的操作。

        Step 18. InputDispatcher.registerInputChannel  
        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputDispatcher::registerInputChannel(**const** sp<InputChannel>& inputChannel, **bool** monitor) {
2. ......
4. { // acquire lock
5. AutoMutex \_l(mLock);
7. **if** (getConnectionIndexLocked(inputChannel) >= 0) {
8. LOGW("Attempted to register already registered input channel '%s'",
9. inputChannel->getName().string());
10. **return** BAD\_VALUE;
11. }
13. sp<Connection> connection = **new** Connection(inputChannel);
14. status\_t status = connection->initialize();
15. **if** (status) {
16. LOGE("Failed to initialize input publisher for input channel '%s', status=%d",
17. inputChannel->getName().string(), status);
18. **return** status;
19. }
21. int32\_t receiveFd = inputChannel->getReceivePipeFd();
22. mConnectionsByReceiveFd.add(receiveFd, connection);
24. **if** (monitor) {
25. mMonitoringChannels.push(inputChannel);
26. }
28. mLooper->addFd(receiveFd, 0, ALOOPER\_EVENT\_INPUT, handleReceiveCallback, **this**);
30. runCommandsLockedInterruptible();
31. } // release lock
32. **return** OK;
33. }

        这个函数首先会通过getConnectionIndexLocked检查从参数传进来的InputChannel是否已经注册过了，如果已经注册过了，就返回一个BAD\_VALUE值了，否则的话，就会创建一个Connection对象来封装即将要注册的inputChannel，我们可以不关心这个Connection对象的实现，接着还通过调用inputChannel->getReceivePipeFd获得一个管  
道的读端描述符。回忆一下Step 14中的InputChannel.openInputChannelPair函数，我们创建了一个Server端的InputChannel，就是对应这里的inputChannel了，这个inputChannel的Receive Pipe Fd就是我们前面说的反向管道的读端描述符了。有了这个Receive Pipe Fd后，就以它作为Key值来把前面创建的Connection对象保存在InputDispatcher中，这样就基本完成键盘消息接收通道的注册了。但是，注册的工作还未完成，最后，还要把这个Receive Pipe Fd添加到InputDispatcher的成员变量mLooper中去，这里的成员变量mLooper的类型为Looper，我们在前面介绍InputManager的启动过程的Step 15中已经见过了，这里就不再详述了，不过这里仍然值得介绍一下它的addFd函数。

        在前面一篇文章[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)中，我们在介绍到Android应用程序的消息循环一节时，曾经说过，在Looper类内部，会创建一个管道，然后Looper会睡眠在这个管道的读端，等待另外一个线程来往这个管道的写端写入新的内容，从而唤醒等待在这个管道读端的线程，除此之外，Looper还可以同时睡眠等待在其它的文件描述符上，因为它是通过Linux系统的epoll机制来批量等待指定的文件有新的内容可读的。这些其它的文件描述符就是通过Looper类的addFd成函数添加进去的了，在添加的时候，还可以指定回调函数，即当这个文件描述符所指向的文件有新的内容可读时，Looper就会调用这个hanldeReceiveCallback函数，有兴趣的读者可以自己研究一下Looper类的addFd函数的实现，它位于frameworks/base/libs/utils/Looper.cpp文件中。

        分析到这里，Server端的InputChannel就注册完成了。回忆一下前面介绍InputManager启动过程的Step 14，这时InputDispatcherThread同时睡眠在InputDispatcher的成员变量mLooper内部的管道的读端以及这里的Server端InputChannel里面的反向管道的读端上，mLooper内部的管道的读端等待键盘事件的发生而被唤醒，而Server端InputChannel里面的反向管道的读端等待Client端InputChannel里面的反向管道的写端被写入新的内容而被唤醒。

        Server端的InputChannel注册完成后，回到Step 11中的WindowManagerService.addWindow函数，接下来就是把Client端的InputChannel转换成addWindow的参数outInputChannel中，然后返回到Step 1中的ViewRoot.setView函数中，继续执行Client端的InputChannel的注册过程，即为应用程序这一侧注册键盘消息接收通道：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** (view **instanceof** RootViewSurfaceTaker) {
2. mInputQueueCallback =
3. ((RootViewSurfaceTaker)view).willYouTakeTheInputQueue();
4. }
5. **if** (mInputQueueCallback != **null**) {
6. mInputQueue = **new** InputQueue(mInputChannel);
7. mInputQueueCallback.onInputQueueCreated(mInputQueue);
8. } **else** {
9. InputQueue.registerInputChannel(mInputChannel, mInputHandler,
10. Looper.myQueue());
11. }

        这里的变量view一般不为RootViewSurfaceTaker的实例，因此，最后会执行下面语句：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. InputQueue.registerInputChannel(mInputChannel, mInputHandler,
2. Looper.myQueue());

        它调用InputQueue的registerInputChannel函数为应用程序注册键盘消息接收通道，这里的mInputChannel即为我们在前面Step 14中创建的Client端的InputChannel；Looper.myQueue函数返回的便是应用程序主线程的消息队列，具体可以参考前面一篇文章[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)；参数mInputHandler是一个回调对象，当有键盘事件发生时，这个mInputHandler的handleKey函数就会被调用，在后面的分析中，我们将会看到。

        Step 19. InputQueue.registerInputChannel

        这个函数定义在frameworks/base/core/java/android/view/InputQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** InputQueue {
2. ......
4. **public** **static** **void** registerInputChannel(InputChannel inputChannel, InputHandler inputHandler,
5. MessageQueue messageQueue) {
6. ......
8. **synchronized** (sLock) {
9. ......
11. nativeRegisterInputChannel(inputChannel, inputHandler, messageQueue);
12. }
13. }
15. ......
16. }

         这个函数调用本地方法nativeRegisterInputChannel函数来执行进一步的操作。

         Step 20. InputQueue.nativeRegisterInputChannel

         这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_view\_InputQueue\_nativeRegisterInputChannel(JNIEnv\* env, jclass clazz,
2. jobject inputChannelObj, jobject inputHandlerObj, jobject messageQueueObj) {
3. status\_t status = gNativeInputQueue.registerInputChannel(
4. env, inputChannelObj, inputHandlerObj, messageQueueObj);
6. ......
7. }

        这里继续调用NativeInputQueue的registerInputChannel函数来执行真正的键盘消息接收通道的工作。

        Step 21. NativeInputQueue.registerInputChannel

        这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t NativeInputQueue::registerInputChannel(JNIEnv\* env, jobject inputChannelObj,
2. jobject inputHandlerObj, jobject messageQueueObj) {
3. sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,
4. inputChannelObj);
5. ......
7. sp<Looper> looper = android\_os\_MessageQueue\_getLooper(env, messageQueueObj);
9. { // acquire lock
10. AutoMutex \_l(mLock);
12. **if** (getConnectionIndex(inputChannel) >= 0) {
13. LOGW("Attempted to register already registered input channel '%s'",
14. inputChannel->getName().string());
15. **return** BAD\_VALUE;
16. }
18. uint16\_t connectionId = mNextConnectionId++;
19. sp<Connection> connection = **new** Connection(connectionId, inputChannel, looper);
20. status\_t result = connection->inputConsumer.initialize();
21. **if** (result) {
22. LOGW("Failed to initialize input consumer for input channel '%s', status=%d",
23. inputChannel->getName().string(), result);
24. **return** result;
25. }
27. connection->inputHandlerObjGlobal = env->NewGlobalRef(inputHandlerObj);
29. int32\_t receiveFd = inputChannel->getReceivePipeFd();
30. mConnectionsByReceiveFd.add(receiveFd, connection);
32. looper->addFd(receiveFd, 0, ALOOPER\_EVENT\_INPUT, handleReceiveCallback, **this**);
33. } // release lock
35. ......
36. **return** OK;
37. }

        这里注册应用程序的InputChannel的逻辑和前面介绍的Step 18中在InputDispatcher中注册Server端的InputChannel是一样的，所不同的是，这里用的looper是应用程序主线程中的消息循环对象Looper，而添加到这个looper对象中的Receive Pipe Fd是前面在Step 14中创建的前向管道的读端文件描述符，而使用的回调函数是NativeInputQueue的成员函数handleReceiveCallback。

        介绍到这里，应用程序注册键盘消息接收通道的过程就分析完成了。这个过程比较复杂，这里小结一下：

        A. 即将会被激活的Activity窗口，会通知InputManager，它是当前激活的窗口，因此，一旦发生键盘事件的时候，InputManager就把这个键盘事件抛给这个Activity处理；

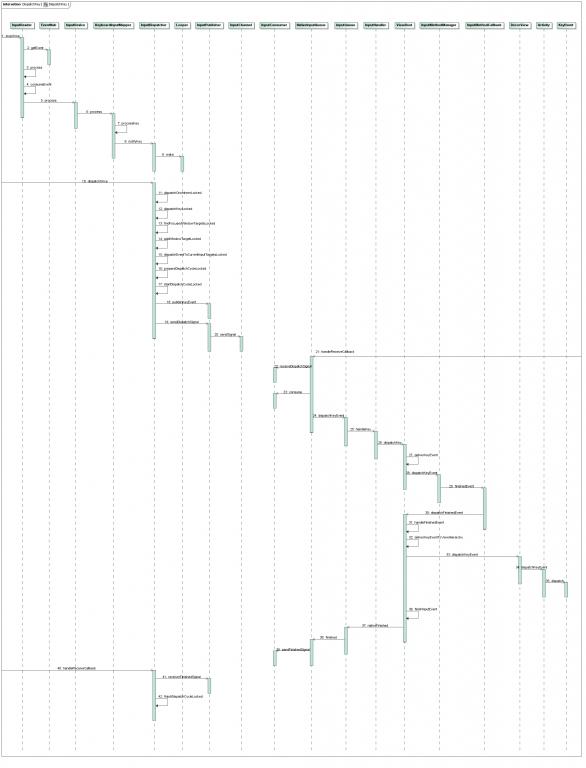
        B. 应用程序会为这个Activity窗口和InputManager之间创建一个键盘消息接收通道，这个通道的一端由一个Server端的InputChannel构成，另一端由Client端的InputChannel构成，Server端的InputChannel注册在由InputManager所管理的InputDispatcher中，而Client端的InputChannel注册在由应用程序主线程的消息循环对象Looper中；

        C. 注册在InputDispatcher中的InputChannel由一个反向管道的读端和一个前向管道的写端组成，而注册在应用程序主线程的消息循环对象Looper中的InputChannel由这个前向管道的读端和反向管道的写端组成，这种交叉结构使得当有键盘事件发生时，InputDispatcher可以把这个事件通知给应用程序。

        应用程序注册好键盘消息接收通道后，接下来就开始分析InputManager分发键盘消息给应用程序的过程了。

        3. InputManager分发键盘消息给应用程序的过程分析

        在分析InputManager分发键盘消息给应用程序的过程之前，我们先假设现在没有键盘事件发生，因此，InputManager中的InputReader正在睡眠等待键盘事件的发生，而InputManager中的InputDispatcher正在等待InputReader从睡眠中醒过来并且唤醒它，而应用程序也正在消息循环中等待InputDispatcher从睡眠中醒过来并且唤醒它。这时候，用户按下键盘中的一个键，于是，一系列唤醒的事件就依次发生了，一直到应用程序中正在显示的Activity得到通知，有键盘事件发生了。我们先来看这个过程的序列图，然后再详细分析每一个步骤：



[点击查看大图](http://hi.csdn.net/attachment/201110/23/0_1319372816yZZx.gif)

        Step 1. InputReader.pollOnce

        Step 2. EventHub.getEvent

        这两个函数分别定义在frameworks/base/libs/ui/InputReader.cpp和frameworks/base/libs/ui/EventHub.cpp文件中，前面我们在分析InputManager的启动过程的Step 17和Step 18时，已经看到过这两个函数了。InputReaderThread线程会不民地循环调用InputReader.pollOnce函数来读入键盘事件，而实际的键盘事件读入操作是由EventHub.getEvent函数来进行的。如果当前没有键盘事件发生，InputReaderThread线程就会睡眠在EventHub.getEvent函数上，而当键盘事件发生后，就会把这个事件封装成一个RawEvent对象，然后返回到pollOnce函数中，执行process函数进一步处理：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputReader::loopOnce() {
2. RawEvent rawEvent;
3. mEventHub->getEvent(& rawEvent);
5. ......
7. process(& rawEvent);
8. }

        Step 3. InputReader.process

        这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputReader::process(**const** RawEvent\* rawEvent) {
2. **switch** (rawEvent->type) {
3. **case** EventHubInterface::DEVICE\_ADDED:
4. addDevice(rawEvent->deviceId);
5. **break**;
7. **case** EventHubInterface::DEVICE\_REMOVED:
8. removeDevice(rawEvent->deviceId);
9. **break**;
11. **case** EventHubInterface::FINISHED\_DEVICE\_SCAN:
12. handleConfigurationChanged(rawEvent->when);
13. **break**;
15. **default**:
16. consumeEvent(rawEvent);
17. **break**;
18. }
19. }

        当键盘事件发生时，rawEvent->type的值为EV\_KEY，这是一个宏定义，具体可以参考bionic/libc/kernel/common/linux/input.h文件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. #define EV\_KEY 0x01

        因此，接下来会调用consumeEvent函数进一步处理。

        Step 4. InputReader.consumeEvent

        这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputReader::consumeEvent(**const** RawEvent\* rawEvent) {
2. int32\_t deviceId = rawEvent->deviceId;
4. { // acquire device registry reader lock
5. RWLock::AutoRLock \_rl(mDeviceRegistryLock);
7. ssize\_t deviceIndex = mDevices.indexOfKey(deviceId);
8. **if** (deviceIndex < 0) {
9. LOGW("Discarding event for unknown deviceId %d.", deviceId);
10. **return**;
11. }
13. InputDevice\* device = mDevices.valueAt(deviceIndex);
14. **if** (device->isIgnored()) {
15. //LOGD("Discarding event for ignored deviceId %d.", deviceId);
16. **return**;
17. }
19. device->process(rawEvent);
20. } // release device registry reader lock
21. }

         首先从rawEvent中取得触发键盘事件设备对象device，然后调用它的process函数进行处理。

         Step 5. InputDevice.process

         这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDevice::process(**const** RawEvent\* rawEvent) {
2. **size\_t** numMappers = mMappers.size();
3. **for** (**size\_t** i = 0; i < numMappers; i++) {
4. InputMapper\* mapper = mMappers[i];
5. mapper->process(rawEvent);
6. }
7. }

         这里的mMapper成员变量保存了一系列输入设备事件处理象，例如负责处理键盘事件的KeyboardKeyMapper对象、负责处理轨迹球事件的TrackballInputMapper对象以及负责处理触摸屏事件的TouchInputMapper对象， 它们是在InputReader类的成员函数createDevice中创建的。这里查询每一个InputMapper对象是否要对当前发生的事件进行处理。由于发生的是键盘事件，真正会对该事件进行处理的只有KeyboardKeyMapper对象。

         Step 6. KeyboardInputMapper.process

         这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** KeyboardInputMapper::process(**const** RawEvent\* rawEvent) {
2. **switch** (rawEvent->type) {
3. **case** EV\_KEY: {
4. int32\_t scanCode = rawEvent->scanCode;
5. **if** (isKeyboardOrGamepadKey(scanCode)) {
6. processKey(rawEvent->when, rawEvent->value != 0, rawEvent->keyCode, scanCode,
7. rawEvent->flags);
8. }
9. **break**;
10. }
11. }
12. }

        这个函数首先会检查一下键盘扫描码是否正确，如果正确的话，就会调用processKey函数进一步处理。

        Step 7. KeyboardInputMapper.processKey

        这个函数定义在frameworks/base/libs/ui/InputReader.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** KeyboardInputMapper::processKey(nsecs\_t when, **bool** down, int32\_t keyCode,
2. int32\_t scanCode, uint32\_t policyFlags) {
3. int32\_t newMetaState;
4. nsecs\_t downTime;
5. **bool** metaStateChanged = **false**;
7. { // acquire lock
8. AutoMutex \_l(mLock);
10. **if** (down) {
11. // Rotate key codes according to orientation if needed.
12. // Note: getDisplayInfo is non-reentrant so we can continue holding the lock.
13. **if** (mAssociatedDisplayId >= 0) {
14. int32\_t orientation;
15. **if** (! getPolicy()->getDisplayInfo(mAssociatedDisplayId, NULL, NULL, & orientation)) {
16. **return**;
17. }
19. keyCode = rotateKeyCode(keyCode, orientation);
20. }
22. // Add key down.
23. ssize\_t keyDownIndex = findKeyDownLocked(scanCode);
24. **if** (keyDownIndex >= 0) {
25. // key repeat, be sure to use same keycode as before in case of rotation
26. keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
27. } **else** {
28. // key down
29. **if** ((policyFlags & POLICY\_FLAG\_VIRTUAL)
30. && mContext->shouldDropVirtualKey(when, getDevice(), keyCode, scanCode)) {
31. **return**;
32. }
33. mLocked.keyDowns.push();
34. KeyDown& keyDown = mLocked.keyDowns.editTop();
35. keyDown.keyCode = keyCode;
36. keyDown.scanCode = scanCode;
37. }
39. mLocked.downTime = when;
40. } **else** {
41. // Remove key down.
42. ssize\_t keyDownIndex = findKeyDownLocked(scanCode);
43. **if** (keyDownIndex >= 0) {
44. // key up, be sure to use same keycode as before in case of rotation
45. keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
46. mLocked.keyDowns.removeAt(**size\_t**(keyDownIndex));
47. } **else** {
48. // key was not actually down
49. LOGI("Dropping key up from device %s because the key was not down.  "
50. "keyCode=%d, scanCode=%d",
51. getDeviceName().string(), keyCode, scanCode);
52. **return**;
53. }
54. }
56. int32\_t oldMetaState = mLocked.metaState;
57. newMetaState = updateMetaState(keyCode, down, oldMetaState);
58. **if** (oldMetaState != newMetaState) {
59. mLocked.metaState = newMetaState;
60. metaStateChanged = **true**;
61. }
63. downTime = mLocked.downTime;
64. } // release lock

67. **if** (metaStateChanged) {
68. getContext()->updateGlobalMetaState();
69. }
71. getDispatcher()->notifyKey(when, getDeviceId(), AINPUT\_SOURCE\_KEYBOARD, policyFlags,
72. down ? AKEY\_EVENT\_ACTION\_DOWN : AKEY\_EVENT\_ACTION\_UP,
73. AKEY\_EVENT\_FLAG\_FROM\_SYSTEM, keyCode, scanCode, newMetaState, downTime);
74. }

        这个函数首先对对按键作一些处理，例如，当某一个DPAD键被按下时，根据当时屏幕方向的不同，它所表示的意义也不同，因此，这里需要根据当时屏幕的方向来调整键盘码：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Rotate key codes according to orientation if needed.
2. // Note: getDisplayInfo is non-reentrant so we can continue holding the lock.
3. **if** (mAssociatedDisplayId >= 0) {
4. int32\_t orientation;
5. **if** (! getPolicy()->getDisplayInfo(mAssociatedDisplayId, NULL, NULL, & orientation)) {
6. **return**;
7. }
9. keyCode = rotateKeyCode(keyCode, orientation);
10. }

        如果这个键是一直按着不放的，不管屏幕的方向如何，必须保证后面的键盘码和前面的一样：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Add key down.
2. ssize\_t keyDownIndex = findKeyDownLocked(scanCode);
3. **if** (keyDownIndex >= 0) {
4. // key repeat, be sure to use same keycode as before in case of rotation
5. keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
6. } **else** {
7. // key down
8. **if** ((policyFlags & POLICY\_FLAG\_VIRTUAL)
9. && mContext->shouldDropVirtualKey(when, getDevice(), keyCode, scanCode)) {
10. **return**;
11. }
12. mLocked.keyDowns.push();
13. KeyDown& keyDown = mLocked.keyDowns.editTop();
14. keyDown.keyCode = keyCode;
15. keyDown.scanCode = scanCode;
16. }

       如果是第一次按下某个键，还必须把它保存在mLocked.keyDowns里面，就是为了处理上面讲的当这个键盘一直按着不放的时候屏幕方向发生改变的情况。  
       如果是松开键盘上的某个键，就把它从mLocked.keyDowns里面删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Remove key down.
2. ssize\_t keyDownIndex = findKeyDownLocked(scanCode);
3. **if** (keyDownIndex >= 0) {
4. // key up, be sure to use same keycode as before in case of rotation
5. keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
6. mLocked.keyDowns.removeAt(**size\_t**(keyDownIndex));
7. } **else** {
8. // key was not actually down
9. LOGI("Dropping key up from device %s because the key was not down.  "
10. "keyCode=%d, scanCode=%d",
11. getDeviceName().string(), keyCode, scanCode);
12. **return**;
13. }

        当然，对键盘事件的这些处理不是本文的重点，本文的重点是分析从键盘事件到当前激活的Activity窗口接收到这个键盘消息的过程。

        最后，KeyboardInputMappger函数通知InputDispatcher，有键盘事件发生了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. getDispatcher()->notifyKey(when, getDeviceId(), AINPUT\_SOURCE\_KEYBOARD, policyFlags,
2. down ? AKEY\_EVENT\_ACTION\_DOWN : AKEY\_EVENT\_ACTION\_UP,
3. AKEY\_EVENT\_FLAG\_FROM\_SYSTEM, keyCode, scanCode, newMetaState, downTime);

        Step 8. InputDispatcher.notifyKey

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::notifyKey(nsecs\_t eventTime, int32\_t deviceId, int32\_t source,
2. uint32\_t policyFlags, int32\_t action, int32\_t flags,
3. int32\_t keyCode, int32\_t scanCode, int32\_t metaState, nsecs\_t downTime) {
4. ......
6. **if** (! validateKeyEvent(action)) {
7. **return**;
8. }
10. /\* According to http://source.android.com/porting/keymaps\_keyboard\_input.html
11. \* Key definitions: Key definitions follow the syntax key SCANCODE KEYCODE [FLAGS...],
12. \* where SCANCODE is a number, KEYCODE is defined in your specific keylayout file
13. \* (android.keylayout.xxx), and potential FLAGS are defined as follows:
14. \*     SHIFT: While pressed, the shift key modifier is set
15. \*     ALT: While pressed, the alt key modifier is set
16. \*     CAPS: While pressed, the caps lock key modifier is set
17. \*     Since KeyEvent.java doesn't check if Cap lock is ON and we don't have a
18. \*     modifer state for cap lock, we will not support it.
19. \*/
20. **if** (policyFlags & POLICY\_FLAG\_ALT) {
21. metaState |= AMETA\_ALT\_ON | AMETA\_ALT\_LEFT\_ON;
22. }
23. **if** (policyFlags & POLICY\_FLAG\_ALT\_GR) {
24. metaState |= AMETA\_ALT\_ON | AMETA\_ALT\_RIGHT\_ON;
25. }
26. **if** (policyFlags & POLICY\_FLAG\_SHIFT) {
27. metaState |= AMETA\_SHIFT\_ON | AMETA\_SHIFT\_LEFT\_ON;
28. }
30. policyFlags |= POLICY\_FLAG\_TRUSTED;
31. mPolicy->interceptKeyBeforeQueueing(eventTime, deviceId, action, /\*byref\*/ flags,
32. keyCode, scanCode, /\*byref\*/ policyFlags);
34. **bool** needWake;
35. { // acquire lock
36. AutoMutex \_l(mLock);
38. int32\_t repeatCount = 0;
39. KeyEntry\* newEntry = mAllocator.obtainKeyEntry(eventTime,
40. deviceId, source, policyFlags, action, flags, keyCode, scanCode,
41. metaState, repeatCount, downTime);
43. needWake = enqueueInboundEventLocked(newEntry);
44. } // release lock
46. **if** (needWake) {
47. mLooper->wake();
48. }
49. }

        函数首先是调用validateKeyEvent函数来验证action参数是否正确：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **bool** isValidKeyAction(int32\_t action) {
2. **switch** (action) {
3. **case** AKEY\_EVENT\_ACTION\_DOWN:
4. **case** AKEY\_EVENT\_ACTION\_UP:
5. **return** **true**;
6. **default**:
7. **return** **false**;
8. }
9. }
11. **static** **bool** validateKeyEvent(int32\_t action) {
12. **if** (! isValidKeyAction(action)) {
13. LOGE("Key event has invalid action code 0x%x", action);
14. **return** **false**;
15. }
16. **return** **true**;
17. }

        正确的action参数的值只能为AKEY\_EVENT\_ACTION\_DOWN（按下）或者AKEY\_EVENT\_ACTION\_UP（松开）。

        参数action检查通过后，还通过policyFlags参数来检查一下同时是否有ALT和SHIFT键被按下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** (policyFlags & POLICY\_FLAG\_ALT) {
2. metaState |= AMETA\_ALT\_ON | AMETA\_ALT\_LEFT\_ON;
3. }
4. **if** (policyFlags & POLICY\_FLAG\_ALT\_GR) {
5. metaState |= AMETA\_ALT\_ON | AMETA\_ALT\_RIGHT\_ON;
6. }
7. **if** (policyFlags & POLICY\_FLAG\_SHIFT) {
8. metaState |= AMETA\_SHIFT\_ON | AMETA\_SHIFT\_LEFT\_ON;
9. }

        最后，调用enqueueInboundEventLocked函数把这个按键事件封装成一个KeyEntry结构加入到InputDispatcher类的mInboundQueue队列中去：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** InputDispatcher::enqueueInboundEventLocked(EventEntry\* entry) {
2. **bool** needWake = mInboundQueue.isEmpty();
3. mInboundQueue.enqueueAtTail(entry);
5. **switch** (entry->type) {
6. **case** EventEntry::TYPE\_KEY: {
7. KeyEntry\* keyEntry = **static\_cast**<KeyEntry\*>(entry);
8. **if** (isAppSwitchKeyEventLocked(keyEntry)) {
9. **if** (keyEntry->action == AKEY\_EVENT\_ACTION\_DOWN) {
10. mAppSwitchSawKeyDown = **true**;
11. } **else** **if** (keyEntry->action == AKEY\_EVENT\_ACTION\_UP) {
12. **if** (mAppSwitchSawKeyDown) {
13. <span style="white-space:pre">      </span>    ......
14. mAppSwitchDueTime = keyEntry->eventTime + APP\_SWITCH\_TIMEOUT;
15. mAppSwitchSawKeyDown = **false**;
16. needWake = **true**;
17. }
18. }
19. }
20. **break**;
21. }
22. }
24. **return** needWake;
25. }

        从这个函数我们可以看出，在两种情况下，它的返回值为true，一是当加入该键盘事件到mInboundQueue之前，mInboundQueue为空，这表示InputDispatccherThread线程正在睡眠等待InputReaderThread线程的唤醒，因此，它返回true表示要唤醒InputDispatccherThread线程；二是加入该键盘事件到mInboundQueue之前，mInboundQueue不为空，但是此时用户按下的是Home键，按下Home键表示要切换App，我们知道，在切换App时，新的App会把它的键盘消息接收通道注册到InputDispatcher中去，并且会等待InputReader的唤醒，因此，在这种情况下，也需要返回true，表示要唤醒InputDispatccherThread线程。如果不是这两种情况，那么就说明InputDispatccherThread线程现在正在处理前面的键盘事件，不需要唤醒它。

        回到前面的notifyKey函数中，根据enqueueInboundEventLocked函数的返回值来决定是否要唤醒InputDispatccherThread线程：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **if** (needWake) {
2. mLooper->wake();
3. }

        这里，假设needWake为true，于是，就会调用mLooper对象的wake函数来唤醒InputDispatccherThread线程了。

        Step 9. Looper.wake

        这个函数定义在frameworks/base/libs/utils/Looper.cpp文件中，在前面一篇文章[Android应用程序消息处理机制（Looper、Handler）分析](http://blog.csdn.net/luoshengyang/article/details/6817933)中，我们已经分析过这个函数了，这里不再详述，简单来说，它的作用就是用来唤醒睡眠在Looper对象内部的管道读端的线程，在我们的这个场景中，睡眠在Looper对象内部的管道读端的线程就是InputDispatccherThread线程了。

        从上面InputManager启动过程的Step 15中，我们知道，此时InputDispatccherThread线程正在InputDispatcher类的dispatchOnceb函数中通过调用mLooper->loopOnce函数进入睡眠状态。当它被唤醒以后，它就会从InputDispatcher类的dispatchOnceb函数返回到InputDispatcherThread类的threadLoop函数，而InputDispatcherThread类的threadLoop函数是循环执行的，于是，它又会再次进入到InputDispatcher类的dispatchOnce函数来处理当前发生的键盘事件。

        Step 10. InputDispatcher.dispatchOnce

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::dispatchOnce() {
2. nsecs\_t keyRepeatTimeout = mPolicy->getKeyRepeatTimeout();
3. nsecs\_t keyRepeatDelay = mPolicy->getKeyRepeatDelay();
5. nsecs\_t nextWakeupTime = LONG\_LONG\_MAX;
6. { // acquire lock
7. AutoMutex \_l(mLock);
8. dispatchOnceInnerLocked(keyRepeatTimeout, keyRepeatDelay, & nextWakeupTime);
10. ......
11. } // release lock
13. ......
14. }

        它调用dispatchOnceInnerLocked函数来进一步处理这个键盘事件。

        Step 11. InputDispatcher.dispatchOnceInnerLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::dispatchOnceInnerLocked(nsecs\_t keyRepeatTimeout,
2. nsecs\_t keyRepeatDelay, nsecs\_t\* nextWakeupTime) {
3. ......
5. // Ready to start a new event.
6. // If we don't already have a pending event, go grab one.
7. **if** (! mPendingEvent) {
8. **if** (mInboundQueue.isEmpty()) {
9. ......
10. } **else** {
11. // Inbound queue has at least one entry.
12. EventEntry\* entry = mInboundQueue.headSentinel.next;
14. ......
16. mInboundQueue.dequeue(entry);
17. mPendingEvent = entry;
18. }
20. ......
21. }
23. ......
25. **switch** (mPendingEvent->type) {
26. ......
28. **case** EventEntry::TYPE\_KEY: {
29. KeyEntry\* typedEntry = **static\_cast**<KeyEntry\*>(mPendingEvent);
30. ......
31. done = dispatchKeyLocked(currentTime, typedEntry, keyRepeatTimeout,
32. &dropReason, nextWakeupTime);
33. **break**;
34. }
36. ......
37. }
39. ......
40. }

        我们忽略了这个函数的次要逻辑，主要关注键盘事件的主要处理流程。首先，如果前面发生的键盘事件都已经处理完毕，那么这里的mPendingEvent就为NULL，又因为前面我们把刚刚发生的键盘事件加入了mInboundQueue队列，因此，这里mInboundQueue不为NULL，于是，这里就把mInboundQueue队列中的键盘事件取出来，放在mPendingEvent变量中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. mInboundQueue.dequeue(entry);
2. mPendingEvent = entry;

        由于这里发生的是键盘事件，即mPendingEvent->type的值为EventEntry::TYPE\_KEY，于是，在接下来的switch语句中就会执行dispatchKeyLocked函数来分发键盘消息。

        Step 12. InputDispatcher.dispatchKeyLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **bool** InputDispatcher::dispatchKeyLocked(
2. nsecs\_t currentTime, KeyEntry\* entry, nsecs\_t keyRepeatTimeout,
3. DropReason\* dropReason, nsecs\_t\* nextWakeupTime) {
4. ......
6. // Identify targets.
7. **if** (! mCurrentInputTargetsValid) {
8. int32\_t injectionResult = findFocusedWindowTargetsLocked(currentTime,
9. entry, nextWakeupTime);
11. ......
12. }
14. // Dispatch the key.
15. dispatchEventToCurrentInputTargetsLocked(currentTime, entry, **false**);
16. **return** **true**;
17. }

         InputDispatcher类中的mCurrentInputTargetsValid成员变量表示InputDispatcher是否已经标志出谁是当前激活的Activity窗口，如果没有，就需要通过findFocusedWindowTargetsLocked函数来把它找出来。当把当前激活的Activity窗口找出来以后，接下来就调用dispatchEventToCurrentInputTargetsLocked函数把键盘事件分发给它了。

        我们先来看一InputDispatcher是如何找到当前激活的Activity窗口的，然后再分析它把键盘事件分发给当前激活Activity窗口的过程。

        Step 13. InputDispatcher.findFocusedWindowTargetsLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. int32\_t InputDispatcher::findFocusedWindowTargetsLocked(nsecs\_t currentTime,
2. **const** EventEntry\* entry, nsecs\_t\* nextWakeupTime) {
3. mCurrentInputTargets.clear();
5. int32\_t injectionResult;
7. // If there is no currently focused window and no focused application
8. // then drop the event.
9. **if** (! mFocusedWindow) {
10. **if** (mFocusedApplication) {
11. ......
12. injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
13. mFocusedApplication, NULL, nextWakeupTime);
14. **goto** Unresponsive;
15. }
17. ......
18. injectionResult = INPUT\_EVENT\_INJECTION\_FAILED;
19. **goto** Failed;
20. }
22. // Check permissions.
23. **if** (! checkInjectionPermission(mFocusedWindow, entry->injectionState)) {
24. injectionResult = INPUT\_EVENT\_INJECTION\_PERMISSION\_DENIED;
25. **goto** Failed;
26. }
28. // If the currently focused window is paused then keep waiting.
29. **if** (mFocusedWindow->paused) {
30. ......
31. injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
32. mFocusedApplication, mFocusedWindow, nextWakeupTime);
33. **goto** Unresponsive;
34. }
36. // If the currently focused window is still working on previous events then keep waiting.
37. **if** (! isWindowFinishedWithPreviousInputLocked(mFocusedWindow)) {
38. ......
39. injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
40. mFocusedApplication, mFocusedWindow, nextWakeupTime);
41. **goto** Unresponsive;
42. }
44. // Success!  Output targets.
45. injectionResult = INPUT\_EVENT\_INJECTION\_SUCCEEDED;
46. addWindowTargetLocked(mFocusedWindow, InputTarget::FLAG\_FOREGROUND, BitSet32(0));
48. ......
50. **return** injectionResult;
51. }

        回忆前面我们分析应用程序注册键盘消息接收通道的过程时，在Step 9中，当前处于激活状态的应用程序会通过调用InputDispatcher类setInputWindows函数把把当前获得焦点的Activity窗口设置到mFocusedWindow中去，因此，这里的mFocusedWindow不为NULL，于是，就通过了第一个if语句的检查。

        第二个if语句检查权限问题，原来，这个键盘事件除了是由硬件触发的外，也可以由其它进程注入进来的，如果这个键盘事件是由其它进程注入进来的，那么entry->injectState就不为NULL，它里面包含了事件注册者的进程ID和用户ID，于是，这里就会调用checkInjectionPermission来检查这个事件注入者的进程ID和用户ID，看看它是否具有这个权限。这里我们不考虑这种情况，因此，这里的entry->injectState为NULL，于是，这个if语句的检查也通过了。

        第三个if语句检查当前激活的Activity窗口是否是处于paused状态，如果是的话，也不用进一步处理了。一般情况下，当前激活的Activity窗口都是处于resumed状态的，于是，这个if语句的检查也通过了。

        第四个if语句检查当前激活的Activity窗口是否还正在处理前一个键盘事件，如果是的话，那就要等待它处理完前一个键盘事件后再来处理新的键盘事件了。这里我们也假设当前激活的Activity窗口不是正在处理前面的键盘事件，因此，这个if语句的检查也通过了。

        最后，就调用addWindowTargetLocked函数把当前激活的Activity窗口添加到InputDispatcher类的mCurrentInputTargets成员变量中去。

        Step 14. InputDispatcher.addWindowTargetLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::addWindowTargetLocked(**const** InputWindow\* window, int32\_t targetFlags,
2. BitSet32 pointerIds) {
3. mCurrentInputTargets.push();
5. InputTarget& target = mCurrentInputTargets.editTop();
6. target.inputChannel = window->inputChannel;
7. target.flags = targetFlags;
8. target.xOffset = - window->frameLeft;
9. target.yOffset = - window->frameTop;
10. target.pointerIds = pointerIds;
11. }

        这个函数简单，就是把传进来的参数window添加到mCurrentInputTargets中去就完事了，后面InputDispatcher就会从mCurrentInputTargets中取出恰当的Activity窗口，然后把键盘事件分发给它。

        回到Step 12中的dispatchKeyLocked函数，它接下来就调用dispatchEventToCurrentInputTargetsLocked来进一步处理了。

        Step 15. InputDispatcher.dispatchEventToCurrentInputTargetsLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::dispatchEventToCurrentInputTargetsLocked(nsecs\_t currentTime,
2. EventEntry\* eventEntry, **bool** resumeWithAppendedMotionSample) {
3. ......
5. **for** (**size\_t** i = 0; i < mCurrentInputTargets.size(); i++) {
6. **const** InputTarget& inputTarget = mCurrentInputTargets.itemAt(i);
8. ssize\_t connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);
9. **if** (connectionIndex >= 0) {
10. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
11. prepareDispatchCycleLocked(currentTime, connection, eventEntry, & inputTarget,
12. resumeWithAppendedMotionSample);
13. } **else** {
14. ......
15. }
16. }

        这个函数的实现也比较简单，前面我们已经把当前需要接受键盘事件的Activity窗口添加到mCurrentInputTargets中去了，因此，这里就分别把它们取出来，然后调用prepareDispatchCycleLocked函数把键盘事件分发给它们处理。

        前面我们在分析应用程序注册键盘消息接收通道的过程时，在Step 18中（InputDispatcher.registerInputChannel），把Server端的InputChannel封装成了一个Connection，然后以这个InputChannel中的Receive Pipe Fd作为键值把这个Connection对象保存在mConnectionsByReceiveFd中。这里，既然我们已经通过mCurrentInputTargets得到了表示当前需要接收键盘事件的Activity窗口的InputTarget对象，而且这个InputTarget对象的inputChannel就表示当初在InputDispatcher中注册的Server端InputChannel，因此，这里就可以把这个Connection对象取出来，最后调用prepareDispatchCycleLocked函数来进一步处理。

        Step 16. InputDispatcher.prepareDispatchCycleLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::prepareDispatchCycleLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection, EventEntry\* eventEntry, **const** InputTarget\* inputTarget,
3. **bool** resumeWithAppendedMotionSample) {
4. ......

7. // Resume the dispatch cycle with a freshly appended motion sample.
8. // First we check that the last dispatch entry in the outbound queue is for the same
9. // motion event to which we appended the motion sample.  If we find such a dispatch
10. // entry, and if it is currently in progress then we try to stream the new sample.
11. **bool** wasEmpty = connection->outboundQueue.isEmpty();
13. **if** (! wasEmpty && resumeWithAppendedMotionSample) {
14. ......
15. **return**;
16. }
18. // This is a new event.
19. // Enqueue a new dispatch entry onto the outbound queue for this connection.
20. DispatchEntry\* dispatchEntry = mAllocator.obtainDispatchEntry(eventEntry, // increments ref
21. inputTarget->flags, inputTarget->xOffset, inputTarget->yOffset);
23. ......
25. // Enqueue the dispatch entry.
26. connection->outboundQueue.enqueueAtTail(dispatchEntry);
28. // If the outbound queue was previously empty, start the dispatch cycle going.
29. **if** (wasEmpty) {
30. ......
32. startDispatchCycleLocked(currentTime, connection);
33. }
34. }

         在开始处理键盘事件之前，这个函数会检查一下传进来的参数connection中的outboundQueue事件队列是否为空，如果不为空，就要看看当前要处理的事件和outboundQueue队列中的最后一个事件是不是同一个motion事件，如果是的话，并且从上面传进来的resumeWithAppendedMotionSample参数为true，这时候就要以流水线的方式来处理这些motion事件了。在我们这个情景中，要处理的是键盘事件，因此在上面Step 12中传进来的resumeWithAppendedMotionSample参数为false，因此，我们略过这种情况。

         接下来，就会把当前的键盘事件封装成一个DispatchEntry对象，然后添加到connection对象的outboundQueue队列中去，表示当前键盘事件是一个待处理的键盘事件。

         当connection中的outboundQueue事件队列不为空，即wasEmpty为false时，说明当前这个Activity窗口正在处键盘事件了，因此，就不需要调用startDispatchCycleLocked来启动Activity窗口来处理这个事件了，因为一旦这个Activity窗口正在处键盘事件，它就会一直处理下去，直到它里的connection对象的outboundQueue为空为止。当connection中的outboundQueue事件队列为空时，就需要调用startDispatchCycleLocked来通知这个Activity窗口来执行键盘事件处理的流程了。

         Step 17. InputDispatcher.startDispatchCycleLocked

        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::startDispatchCycleLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection) {
3. ......
5. DispatchEntry\* dispatchEntry = connection->outboundQueue.headSentinel.next;
7. // Mark the dispatch entry as in progress.
8. dispatchEntry->inProgress = **true**;
10. // Update the connection's input state.
11. EventEntry\* eventEntry = dispatchEntry->eventEntry;
12. ......
14. // Publish the event.
15. status\_t status;
16. **switch** (eventEntry->type) {
17. **case** EventEntry::TYPE\_KEY: {
18. KeyEntry\* keyEntry = **static\_cast**<KeyEntry\*>(eventEntry);
20. // Apply target flags.
21. int32\_t action = keyEntry->action;
22. int32\_t flags = keyEntry->flags;
24. // Publish the key event.
25. status = connection->inputPublisher.publishKeyEvent(keyEntry->deviceId, keyEntry->source,
26. action, flags, keyEntry->keyCode, keyEntry->scanCode,
27. keyEntry->metaState, keyEntry->repeatCount, keyEntry->downTime,
28. keyEntry->eventTime);
30. ......
31. **break**;
32. }
33. ......
34. }
36. // Send the dispatch signal.
37. status = connection->inputPublisher.sendDispatchSignal();
39. ......
40. }

         这个函数主要围绕传进来的Connection对象做两件事情，一是从它的outboundQueue队列中取出当前需要处理的键盘事件，然后把这个事件记录在它的内部对象inputPublisher中，二是通过它的内部对象inputPublisher通知它所关联的Activity窗口，现在有键盘事件需要处理了。第一件事情是通过调用它的InputPublisher对象的publishKeyEvent函数来完成的，而第二件事情是通过调用它的InputPublisher对象的sendDispatchSignal来完成的。我们先来看InputPublisher的成员函数publishKeyEvent的实现，然后再回来分析它的另外一个成员函数sendDispatchSignal的实现。

        Step 18. InputPublisher.publishKeyEvent

        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputPublisher::publishKeyEvent(
2. int32\_t deviceId,
3. int32\_t source,
4. int32\_t action,
5. int32\_t flags,
6. int32\_t keyCode,
7. int32\_t scanCode,
8. int32\_t metaState,
9. int32\_t repeatCount,
10. nsecs\_t downTime,
11. nsecs\_t eventTime) {
12. ......
14. status\_t result = publishInputEvent(AINPUT\_EVENT\_TYPE\_KEY, deviceId, source);
15. **if** (result < 0) {
16. **return** result;
17. }
19. mSharedMessage->key.action = action;
20. mSharedMessage->key.flags = flags;
21. mSharedMessage->key.keyCode = keyCode;
22. mSharedMessage->key.scanCode = scanCode;
23. mSharedMessage->key.metaState = metaState;
24. mSharedMessage->key.repeatCount = repeatCount;
25. mSharedMessage->key.downTime = downTime;
26. mSharedMessage->key.eventTime = eventTime;
27. **return** OK;
28. }

        这个函数主要就是把键盘事件记录在InputPublisher类的成员变量mSharedMessage中了，这个mSharedMessage成员变量指向的是一个匿名共享内存。

        这个匿名共享内存是什么时候创建的呢？前面我们在分析应用程序注册键盘消息接收通道的过程时，在Step 18中（InputDispatcher.registerInputChannel），在把Server端的InputChannel封装成一个 Connection对象时，会调用它的initialize成员函数来执行一些初始化工作，就是在这个时候创建这个匿名共享内存的了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. sp<Connection> connection = **new** Connection(inputChannel);
2. status\_t status = connection->initialize();

       我们来看一下这个initialize函数的实现，它定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputPublisher::initialize() {
2. ......
4. **int** ashmemFd = mChannel->getAshmemFd();
5. **int** result = ashmem\_get\_size\_region(ashmemFd);
6. ......
8. mAshmemSize = (**size\_t**) result;
10. mSharedMessage = **static\_cast**<InputMessage\*>(mmap(NULL, mAshmemSize,
11. PROT\_READ | PROT\_WRITE, MAP\_SHARED, ashmemFd, 0));
12. ......
14. mPinned = **true**;
15. mSharedMessage->consumed = **false**;
17. **return** reset();
18. }

        InputPublisher的成员变量mChannel就是指注册在InputDispatcher中的Server端InputChannel了。我们知道，这个InputChannel除了拥有一个反向管道的读端文件描述符和一个前向管道的写端文件描述符之后，还有一个匿名共享文件描述符，这个匿名共享文件描述符就是用来创建匿名共享内存mSharedMessage的了。

        这个匿名共享内存mSharedMessage的作用是什么呢？原来，在InputChannel中，前向管道和反向管道的作用只是用来在Server端和Client端之间相互通知有事件发生了，但是具体是什么样的事件，还需要去读取这个匿名共享内存的内容才知道。前面我们在分析应用程序注册键盘消息接收通道的过程时，在Step 14中（InputChannel.openInputChannelPair）创建Server端和Client端的InputChannel对时，创建一个匿名共享内存，这个匿名共享内存有两个文件描述符同时指向它，其中一个放在Server端的InputChannel中，另外一个放在Client端的InputChannel中。这样，当InputDispatcher通过Server端的InputChannel的前向管道来通知Client端有键盘事件发生时，Client端只要通过它的InputChannel中的匿名共享内存文件描述符去读取匿名共享内存中的内容，就可以知道发生了什么事情了。有关匿名共享内存的相关知识，请参考[Android系统匿名共享内存Ashmem（Anonymous Shared Memory）简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/6651971)一文。

        回到Step 17中，接下来就是调用InputPublisher的成员函数sendDispatchSignal来通知Activity窗口处理键盘事件了。

        Step 19. InputPublishe.sendDispatchSignal  
        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputPublisher::sendDispatchSignal() {
2. ......
4. mWasDispatched = **true**;
5. **return** mChannel->sendSignal(INPUT\_SIGNAL\_DISPATCH);
6. }

        这个函数很简单，它通过调用内部成员变量mChannel的sendSignal函数来通知相应的Activity窗口来处理键盘事件。

        Step 20. InputChannel.sendSignal

        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputChannel::sendSignal(**char** signal) {
2. ssize\_t nWrite;
3. **do** {
4. nWrite = ::write(mSendPipeFd, & signal, 1);
5. } **while** (nWrite == -1 && errno == EINTR);
7. **if** (nWrite == 1) {
8. ......
9. **return** OK;
10. }
12. **return** -errno;
13. }

        这里所谓的发送信号通知，其实是通过向其内部一个管道的写端写入一个字符来实现的。前面我们分析应用程序注册键盘消息接收通道的过程时，在Step 21中（NativeInputQueue.registerInputChannel），它把一个InputChannel注册到应用程序主线程中的Looper对象中，然后应用程序的主线程就通过这个Looper对象睡眠等待在这个InputChannel中的前向管道中有新的内容可读了，这里的mSendPipeFd就是对应这个前向管道的写端。现在既然向这个前向管道的写端写入新的内容了，于是，应用程序的主线程就被唤醒了。

        在前面分析应用程序注册键盘消息接收通道过程的Step 21中，我们也说过，当应用程序的主线程因为这个InputChannel中的前向管道的写端唤醒时，NativeInputQueue的成员函数handleReceiveCallback就会被回调，因此，接下来，应用程序的主线程就会被唤醒，然后执行NativeInputQueue的成员函数handleReceiveCallback。

        Step 21. NativeInputQueue.handleReceiveCallback

        这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **int** NativeInputQueue::handleReceiveCallback(**int** receiveFd, **int** events, **void**\* data) {
2. NativeInputQueue\* q = **static\_cast**<NativeInputQueue\*>(data);
3. JNIEnv\* env = AndroidRuntime::getJNIEnv();
5. sp<Connection> connection;
6. InputEvent\* inputEvent;
7. jobject inputHandlerObjLocal;
8. jlong finishedToken;
9. { // acquire lock
10. AutoMutex \_l(q->mLock);
12. ssize\_t connectionIndex = q->mConnectionsByReceiveFd.indexOfKey(receiveFd);
13. ......
15. connection = q->mConnectionsByReceiveFd.valueAt(connectionIndex);
16. ......

19. status\_t status = connection->inputConsumer.receiveDispatchSignal();
20. **if** (status) {
21. ......
22. **return** 0; // remove the callback
23. }
25. ......
27. status = connection->inputConsumer.consume(& connection->inputEventFactory, & inputEvent);
28. ......
30. finishedToken = generateFinishedToken(receiveFd, connection->id, connection->messageSeqNum);
32. inputHandlerObjLocal = env->NewLocalRef(connection->inputHandlerObjGlobal);
33. } // release lock
35. ......
37. int32\_t inputEventType = inputEvent->getType();
39. jobject inputEventObj;
40. jmethodID dispatchMethodId;
41. **switch** (inputEventType) {
42. **case** AINPUT\_EVENT\_TYPE\_KEY:
43. ......
44. inputEventObj = android\_view\_KeyEvent\_fromNative(env,
45. **static\_cast**<KeyEvent\*>(inputEvent));
46. dispatchMethodId = gInputQueueClassInfo.dispatchKeyEvent;
47. **break**;
48. }
49. ......
50. }
52. ......
54. env->CallStaticVoidMethod(gInputQueueClassInfo.clazz,
55. dispatchMethodId, inputHandlerObjLocal, inputEventObj,
56. jlong(finishedToken));
58. ......
60. **return** 1;
61. }

        这个函数首先是通过参数data获得当初注册InputChannel的NativeInputQueue对象，具体可以参考前面介绍的应用程序注册键盘消息接收通道过程的Step 21。接下来再通过参数receiveFd获得保存在这个NativeInputQueue对象中的mConnectionsByReceiveFd成员变量中的Connection对象。有了这个Connection对象后，就可以获得它内部的InputConsumer对象，这个InputConsumer对象是和上面的Step 18中介绍的InputPublisher对象相应的。

        在InputChannel内部中，分别有一个InputPublisher对象和一个InputConsumer对象，对于Server端的InputChannel来说，它使用的是InputPublisher对象，通过它进行键盘消息的分发，而对于Client端的InputChannel来说，它使用的是InputConsumer对象，通过它进行键盘消息的读取。

        获得了这个InputConsumer对象后首先是调用它的receiveDispatchSignal来确认是否是接收到了键盘消息的通知，如果是的话，再调用它的consume函数来把键盘事件读取出来，最后，调用Java层的回调对象InputQueue的DispatchKeyEvent来处理这个键盘事件。下面，我们就依次来分析这些过程。

        Step 22. InputConsumer.receiveDispatchSignal

        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputConsumer::receiveDispatchSignal() {
2. ......
4. **char** signal;
5. status\_t result = mChannel->receiveSignal(& signal);
6. **if** (result) {
7. **return** result;
8. }
9. **if** (signal != INPUT\_SIGNAL\_DISPATCH) {
10. ......
11. **return** UNKNOWN\_ERROR;
12. }
13. **return** OK;
14. }

         这个函数很简单，它通过它内部对象mChannel来从前向管道的读端读入一个字符，看看是否是前面的Step 20中写入的INPUT\_SIGNAL\_DISPATCH字符。

         InputChannel类的receiveSignal函数也是定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputChannel::receiveSignal(**char**\* outSignal) {
2. ssize\_t nRead;
3. **do** {
4. nRead = ::read(mReceivePipeFd, outSignal, 1);
5. } **while** (nRead == -1 && errno == EINTR);
7. **if** (nRead == 1) {
8. ......
9. **return** OK;
10. }
12. ......
13. **return** -errno;
14. }

        Step 23. InputConsumer.consume

        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputConsumer::consume(InputEventFactoryInterface\* factory, InputEvent\*\* outEvent) {
2. ......
4. \*outEvent = NULL;
6. **int** ashmemFd = mChannel->getAshmemFd();
7. **int** result = ashmem\_pin\_region(ashmemFd, 0, 0);
8. ......
10. **if** (mSharedMessage->consumed) {
11. ......
12. **return** INVALID\_OPERATION;
13. }
15. // Acquire but \*never release\* the semaphore.  Contention on the semaphore is used to signal
16. // to the publisher that the message has been consumed (or is in the process of being
17. // consumed).  Eventually the publisher will reinitialize the semaphore for the next message.
18. result = sem\_wait(& mSharedMessage->semaphore);
19. ......
21. mSharedMessage->consumed = **true**;
23. **switch** (mSharedMessage->type) {
24. **case** AINPUT\_EVENT\_TYPE\_KEY: {
25. KeyEvent\* keyEvent = factory->createKeyEvent();
26. **if** (! keyEvent) **return** NO\_MEMORY;
28. populateKeyEvent(keyEvent);
30. \*outEvent = keyEvent;
31. **break**;
32. }
33. ......
34. }
36. **return** OK;
37. }

        这个函数很简单，只要对照前面的Step 18（InputPublisher.publishKeyEvent）来逻辑来看就可以了，后者是往匿名共享内存中写入键盘事件，前者是从这个匿名共享内存中把这个键盘事件的内容读取出来。

        回到Step 21中的handleReceiveCallback函数中，从InputConsumer中获得了键盘事件的内容（保存在本地变量inputEvent中）后，就开始要通知Java层的应用程序了。在前面分析应用程序注册键盘消息接收通道的过程时，在Step 21中（NativeInputQueue.registerInputChannel），会把传进来的对象inputHandlerObj保存在Connection对象中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. connection->inputHandlerObjGlobal = env->NewGlobalRef(inputHandlerObj);

        这个inputHandlerObj对象的类型为Java层的InputHandler对象，因此，这里首先把它取回来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. inputHandlerObjLocal = env->NewLocalRef(connection->inputHandlerObjGlobal);

        取回来之后，我们要把作为参数来调用InputQueue类的dispatchKeyEvent静态成员函数来通知应用程序，有键盘事件发生了，因此，先找到InputQueue类的静态成员函数dispatchKeyEvent的ID：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. dispatchMethodId = gInputQueueClassInfo.dispatchKeyEvent;

        在回调用这个InputQueue类的dispatchKeyEvent静态成员函数之前，还要把前面获得的inputEvent对象转换成Java层的KeyEvent对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. inputEventObj = android\_view\_KeyEvent\_fromNative(env,
2. **static\_cast**<KeyEvent\*>(inputEvent));

        万事具备了，就可以通知Java层的InputQueue来处理这个键盘事件了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. env->CallStaticVoidMethod(gInputQueueClassInfo.clazz,
2. dispatchMethodId, inputHandlerObjLocal, inputEventObj,
3. jlong(finishedToken));

Step 24. InputQueue.dispatchKeyEvent

        这个函数定义在frameworks/base/core/java/android/view/InputQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** InputQueue {
2. ......
4. **private** **static** **void** dispatchKeyEvent(InputHandler inputHandler,
5. KeyEvent event, **long** finishedToken) {
6. Runnable finishedCallback = FinishedCallback.obtain(finishedToken);
7. inputHandler.handleKey(event, finishedCallback);
8. }
10. ......
11. }

        这个函数首先会创建一个FinishedCallback类型的对象finishedCallback，FinishedCallback是InputQueue的一个内部类，它继承于Runnable类。这个finishedCallback对象是提供给当前Activity窗口的，当它处理完毕键盘事件后，需要通过消息分发的方式来回调这个finishedCallback对象，以及InputQueue类处理一个手尾的工作，后面我们会分析到。

        这里的inputHandler对象是在前面分析应用程序注册键盘消息接收通道的过程时，在Step 1（ViewRoot.setView）中传进来的：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. InputQueue.registerInputChannel(mInputChannel, mInputHandler,
2. Looper.myQueue());

         它是ViewRoot类的一个成员变量mInputHandler。因此，这里将调用ViewRoot类的内部对象mInputHandler的成员函数handleKey来处理键盘事件。

         Step 25. InputHandler.handleKey

         这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **final** InputHandler mInputHandler = **new** InputHandler() {
6. **public** **void** handleKey(KeyEvent event, Runnable finishedCallback) {
7. startInputEvent(finishedCallback);
8. dispatchKey(event, **true**);
9. }
10. ......
11. };
13. ......
14. }

         这个函数首先调用其外部类ViewRoot的startInputEvent成员函数来把回调对象finishedCallback保存下来：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **void** startInputEvent(Runnable finishedCallback) {
6. ......
8. mFinishedCallback = finishedCallback;
9. }
11. ......
12. }

        然后再调用其外部类ViewRoot的dispatchKey成员函数来进一步处这个键盘事件。

        Step 26. ViewRoot.dispatchKey

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **void** dispatchKey(KeyEvent event, **boolean** sendDone) {
6. ......
8. Message msg = obtainMessage(DISPATCH\_KEY);
9. msg.obj = event;
10. msg.arg1 = sendDone ? 1 : 0;
12. ......
14. sendMessageAtTime(msg, event.getEventTime());
15. }
17. ......
18. }

        ViewRoot不是直接处理这个键盘事件，而是把作为一个消息（DISPATCH\_KEY）它放到消息队列中去处理，这个消息最后由ViewRoot类的deliverKeyEvent成员函数来处理。  
        Step 27. ViewRoot.deliverKeyEvent

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **void** deliverKeyEvent(KeyEvent event, **boolean** sendDone) {
6. // If mView is null, we just consume the key event because it doesn't
7. // make sense to do anything else with it.
8. **boolean** handled = mView != **null**
9. ? mView.dispatchKeyEventPreIme(event) : **true**;
10. ......
12. // If it is possible for this window to interact with the input
13. // method window, then we want to first dispatch our key events
14. // to the input method.
15. **if** (mLastWasImTarget) {
16. InputMethodManager imm = InputMethodManager.peekInstance();
17. **if** (imm != **null** && mView != **null**) {
18. ......
20. imm.dispatchKeyEvent(mView.getContext(), seq, event,
21. mInputMethodCallback);
22. **return**;
23. }
24. }
25. ......
26. }
28. ......
29. }

        ViewRoot在把这个键盘事件分发给当前激活的Activity窗口处理之前，首先会调用InputMethodManager的dispatchKeyEvent成员函数来处理这个键盘事件。InputMethodManager处理完这个键盘事件后，再回调用这里的mInputMethodCallback对象的finishedEvent成员函数来把键盘事件分发给当前激活的Activity窗口处理。当然，在把这个键盘事件分发给InputMethodManager处理之前，ViewRoot也会先把这个键盘事件分发给当前激活的Activity窗口的dispatchKeyEventPreIme成员函数处理。

        Step 28. InputMethodManager.dispatchKeyEvent

        这个函数定义在frameworks/base/core/java/android/view/inputmethod/InputMethodManager.java文件中。这是一个输入法相关的类，我们这里就不关注了，只要知道当输入法处理完成之后，它就会调用ViewRoot类的mInputMehtodCallback对象的finishedEvent成员函数。

        Step 29.  InputMethodCallack.finishedEvent

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **static** **class** InputMethodCallback **extends** IInputMethodCallback.Stub {
6. **private** WeakReference<ViewRoot> mViewRoot;
8. **public** InputMethodCallback(ViewRoot viewRoot) {
9. mViewRoot = **new** WeakReference<ViewRoot>(viewRoot);
10. }
12. **public** **void** finishedEvent(**int** seq, **boolean** handled) {
13. **final** ViewRoot viewRoot = mViewRoot.get();
14. **if** (viewRoot != **null**) {
15. viewRoot.dispatchFinishedEvent(seq, handled);
16. }
17. }
19. ......
20. }
22. ......
23. }

         这个函数最终调用ViewRoot的dispatchFinishedEvent来进一步处理。

         Step 30. ViewRoot.dispatchFinishedEvent

         这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **public** **void** dispatchFinishedEvent(**int** seq, **boolean** handled) {
6. Message msg = obtainMessage(FINISHED\_EVENT);
7. msg.arg1 = seq;
8. msg.arg2 = handled ? 1 : 0;
9. sendMessage(msg);
10. }
12. ......
13. }

        和前面的Step 26一样，ViewRoot不是直接处理这个键盘事件，而是把它作为一个消息（FINISHED\_EVENT）放在消息队列中去，最后，这个消息由ViewRoot的handleFinishedEvent函数来处理。

        Step 31. ViewRoot.handleFinishedEvent

         这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **void** handleFinishedEvent(**int** seq, **boolean** handled) {
6. **final** KeyEvent event = (KeyEvent)retrievePendingEvent(seq);
7. ......
9. **if** (event != **null**) {
10. **final** **boolean** sendDone = seq >= 0;
11. **if** (!handled) {
12. deliverKeyEventToViewHierarchy(event, sendDone);
13. **return**;
14. } **else** **if** (sendDone) {
15. ......
16. } **else** {
17. ......
18. }
19. }
20. }
22. ......
23. }

        如果InputMethodManager没有处理这个键盘事件，那么ViewRoot就会调用deliverKeyEventToViewHierarchy函数来把这个键盘事件分发给当前激活的Activity窗口来处理。

        Step 32. ViewRoot.deliverKeyEventToViewHierarchy

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **void** deliverKeyEventToViewHierarchy(KeyEvent event, **boolean** sendDone) {
6. **try** {
7. **if** (mView != **null** && mAdded) {
8. ......
10. **boolean** keyHandled = mView.dispatchKeyEvent(event);
11. }
13. ......
14. } **finally** {
15. **if** (sendDone) {
16. finishInputEvent();
17. }
18. }
19. }
21. ......
22. }

        这个函数首先会调用ViewRoot类的成员变量mView的dispatchKeyEvent来处理这个键盘事件，然后最调用ViewRoot类的finishInputEvent来处理手尾工作。

        ViewRoot类的成员变量mView的类型为DecorView，它是由ActivityThread类第一次Resume当前的Activity窗口时创建的，具体可以参考ActivityThread类的handleResumeActivity成员函数，这里就不关注了。

        Step 33. DecorView.dispatchKeyEvent

        这个函数定义在frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java文件中，它是PhoneWindow类的一个内部类：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** PhoneWindow **extends** Window **implements** MenuBuilder.Callback {
2. ......
4. **private** **final** **class** DecorView **extends** FrameLayout **implements** RootViewSurfaceTaker {
5. ......
7. @Override
8. **public** **boolean** dispatchKeyEvent(KeyEvent event) {
9. ......
11. **final** Callback cb = getCallback();
12. **final** **boolean** handled = cb != **null** && mFeatureId < 0 ? cb.dispatchKeyEvent(event)
13. : **super**.dispatchKeyEvent(event);
15. ......
16. }
18. ......
19. }
21. ......
22. }

        这里通过getCallback函数返回的是当前应用程序的激活的Activity窗口的Window.Callback接口，一般它不为NULL，因此，这个函数会调用Activity类的dispatchKeyEvent来处理这个键盘事件。  
        Step 34. Activity.dispatchKeyEvent

        这个函数定义在frameworks/base/core/java/android/app/Activity.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** Activity **extends** ContextThemeWrapper
2. **implements** LayoutInflater.Factory,
3. Window.Callback, KeyEvent.Callback,
4. OnCreateContextMenuListener, ComponentCallbacks {
5. ......
7. **public** **boolean** dispatchKeyEvent(KeyEvent event) {
8. ......
10. View decor = mDecor;
11. **if** (decor == **null**) decor = win.getDecorView();
12. **return** event.dispatch(**this**, decor != **null**
13. ? decor.getKeyDispatcherState() : **null**, **this**);
14. }
16. ......
17. }

         这里，Activity不是直接处理这个键盘事件，而是通过KeyEvent的dispatch转发一下。注意，KeyEvent的成中函数dispatch的第一个参数的类型是KeyEvent.Callback，而Activity实现了这个接口，因此，这里可以传this引用过去。

         Step 35. KeyEvent.dispatch

         这个函数定义在frameworks/base/core/java/android/view/KeyEvent.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** KeyEvent **extends** InputEvent **implements** Parcelable {
2. ......
4. **public** **final** **boolean** dispatch(Callback receiver, DispatcherState state,
5. Object target) {
6. **switch** (mAction) {
7. **case** ACTION\_DOWN: {
8. ......
9. **boolean** res = receiver.onKeyDown(mKeyCode, **this**);
10. ......
11. **return** res;
12. }
13. **case** ACTION\_UP:
14. ......
15. **return** receiver.onKeyUp(mKeyCode, **this**);
16. **case** ACTION\_MULTIPLE:
17. **final** **int** count = mRepeatCount;
18. **final** **int** code = mKeyCode;
19. **if** (receiver.onKeyMultiple(code, count, **this**)) {
20. **return** **true**;
21. }
22. ......
23. **return** **false**;
24. }
25. **return** **false**;
26. }
28. ......
29. }

         这里就根据一个键是按下（ACTION\_DOWN）、还是松开（ACTION\_UP）或者是一个相同的键被多次按下和松开（ACTION\_MULTIPLE）等不同事件类型来分别调用Activity的onKeyDown、onKeyUp和onKeyMultiple函数了。  
         Activity窗口处理完这个键盘事件后，层层返回，最后回到Step 32中，调用finishInputEvent事件来处理一些手尾工，下面我们将会看到这些手尾工是什么。

         Step 36. ViewRoot.finishInputEvent

         这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **private** **void** finishInputEvent() {
6. ......
8. **if** (mFinishedCallback != **null**) {
9. mFinishedCallback.run();
10. mFinishedCallback = **null**;
11. } **else** {
12. ......
13. }
14. }
16. ......
17. }

         ViewRoot类里面的成员变量mFinishedCallback是在前面Step 25中由InputQueue设置的，它是一个Runnable对象，实际类型是定义在InputQueue的内部类FinishedCallback，因此，这里调用它的run方法时，接下来就会调用InputQueue的内部类FinishedCallback的run成员函数：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** InputQueue {
2. ......
4. **private** **static** **class** FinishedCallback **implements** Runnable {
5. ......
7. **public** **void** run() {
8. **synchronized** (sLock) {
9. ......
11. nativeFinished(mFinishedToken);
13. ......
14. }
15. }
17. ......
18. }
20. ......
21. }

        这里它调用外部类InputQueue的本地方法nativeFinished来进一步处理。

        Step 37.  InputQueue.nativeFinished

        这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_view\_InputQueue\_nativeFinished(JNIEnv\* env, jclass clazz,
2. jlong finishedToken) {
3. status\_t status = gNativeInputQueue.finished(
4. env, finishedToken, **false** /\*ignoreSpuriousFinish\*/);
6. ......
7. }

        这个函数只是简单只调用NativeInputQueue的finished方法来进一处处理。

        Step 38. NativeInputQueue.finished

        这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t NativeInputQueue::finished(JNIEnv\* env, jlong finishedToken, **bool** ignoreSpuriousFinish) {
2. int32\_t receiveFd;
3. uint16\_t connectionId;
4. uint16\_t messageSeqNum;
5. parseFinishedToken(finishedToken, &receiveFd, &connectionId, &messageSeqNum);
7. { // acquire lock
8. AutoMutex \_l(mLock);
10. ssize\_t connectionIndex = mConnectionsByReceiveFd.indexOfKey(receiveFd);
11. ......
13. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
14. ......
16. connection->messageInProgress = **false**;
18. status\_t status = connection->inputConsumer.sendFinishedSignal();
20. ......
21. } // release lock
23. **return** OK;
24. }

        这个函数最重要的参数便是finishedToken了，通过它可以获得之前通知Java层的InputQueue类来处理键盘事件的Connection对象，它的值是在上面的Step 21（NativeInputQueue.handleReceiveCallback）中生成的：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. finishedToken = generateFinishedToken(receiveFd, connection->id, connection->messageSeqNum);

        函数generateFinishedToken的定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. jlong NativeInputQueue::generateFinishedToken(int32\_t receiveFd, uint16\_t connectionId,
2. uint16\_t messageSeqNum) {
3. **return** (jlong(receiveFd) << 32) | (jlong(connectionId) << 16) | jlong(messageSeqNum);
4. }

        它的实现很简单，只是把receiveFd（前向管道的读端文件描述符）、connectionId（Client端的InputChannel对应的Connection对象在NativeInputQueue中的索引）和messageSeqNum（键盘消息的序列号）三个数值通过移位的方式编码在一个jlong值里面，即编码在上面的finishedToken参数里面。

        因此，在上面的finished函数里面，首先就是要对参数值finishedToken进行解码，把receiveFd、connectionId和messageSeqNum三个值分别取回来：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. parseFinishedToken(finishedToken, &receiveFd, &connectionId, &messageSeqNum);

       parseFinishedToken的定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** NativeInputQueue::parseFinishedToken(jlong finishedToken,
2. int32\_t\* outReceiveFd, uint16\_t\* outConnectionId, uint16\_t\* outMessageIndex) {
3. \*outReceiveFd = int32\_t(finishedToken >> 32);
4. \*outConnectionId = uint16\_t(finishedToken >> 16);
5. \*outMessageIndex = uint16\_t(finishedToken);
6. }

       有了这个receiveFd和connectionId之后，就可以把相应的Connection对象取回来了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. ssize\_t connectionIndex = mConnectionsByReceiveFd.indexOfKey(receiveFd);
2. ......
4. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);

       接下来就是调用这个connection对象中的inputConsumer对象来发送信号通知Server端的InputChannel，应用程序这一侧处理完刚才发生的键盘事件了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t status = connection->inputConsumer.sendFinishedSignal();

       Step 39. InputConsumer.sendFinishedSignal

       这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputConsumer::sendFinishedSignal() {
2. ......
4. **return** mChannel->sendSignal(INPUT\_SIGNAL\_FINISHED);
5. }

        这个函数的实现很简单，只是调用其内部对象mChannel的sendSignal函数来执行发送信号的通知。前面我们已经说过，这里的mChannel的类型为InputChannel，它是注册在应用程序一侧的Client端InputChannel，它的成员函数sendSignal的定义我们在上面的Step 20中已经分析过了，这里不再详述，不过，这里和上面Step 20不一样的地方是，它里的通知方向是从反向管道的写端（在应用程序这一侧）到反向管道的读端（在InputDispatcher这一侧）。

        前面我们在分析应用程序注册键盘消息接收通道的过程时，在Step 18（InputDispatcher.registerInputChannel）中，说到InputDispatcher把一个反向管道的读端文件描述符添加到WindowManagerService所运行的线程中的Looper对象中去，然后就会在这个反向管道的读端上睡眠等待有这个管道有新的内容可读。现在，InputConsumer往这个反向管道写入新的内容了，于是，InputDispatcher就被唤醒过来了，唤醒过来后，它所调用的函数是InputDispatcher.handleReceiveCallback函数，这与前面的Step 21的逻辑是一样的。

       Step 40. InputDispatcher.handleReceiveCallack

       这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **int** InputDispatcher::handleReceiveCallback(**int** receiveFd, **int** events, **void**\* data) {
2. InputDispatcher\* d = **static\_cast**<InputDispatcher\*>(data);
4. { // acquire lock
5. AutoMutex \_l(d->mLock);
7. ssize\_t connectionIndex = d->mConnectionsByReceiveFd.indexOfKey(receiveFd);
8. ......
10. nsecs\_t currentTime = now();
12. sp<Connection> connection = d->mConnectionsByReceiveFd.valueAt(connectionIndex);
13. ......
15. status\_t status = connection->inputPublisher.receiveFinishedSignal();
16. **if** (status) {
17. ......
18. **return** 0; // remove the callback
19. }
21. d->finishDispatchCycleLocked(currentTime, connection);
22. ......
24. **return** 1;
25. } // release lock
26. }

        这个函数首先是通过传进来的receiveFd参数（反向管道的读端文件描述符）的值取得相应的Connection对象：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. ssize\_t connectionIndex = d->mConnectionsByReceiveFd.indexOfKey(receiveFd);
2. ......
4. sp<Connection> connection = d->mConnectionsByReceiveFd.valueAt(connectionIndex);

        然后通过调用这个connection对象的内部对象inputPublisher的receiveFinishedSignal函数来确认是否真的收到键盘事件处理完成的信号，确认之后，就会调用InputDispatcher对象d的finishDispatchCycleLocked函数来执行一些善后工作。下面我们就依次分析这两个过程。

        Step 41. InputPublisher.receiverFinishedSignal

        这个函数定义在frameworks/base/libs/ui/InputTransport.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputPublisher::receiveFinishedSignal() {
2. ....
4. **char** signal;
5. status\_t result = mChannel->receiveSignal(& signal);
6. **if** (result) {
7. **return** result;
8. }
9. **if** (signal != INPUT\_SIGNAL\_FINISHED) {
10. .......
11. **return** UNKNOWN\_ERROR;
12. }
13. **return** OK;
14. }

        这里的逻辑和前面的Step 22中NativeInputQueue确认是否真的收到键盘事件分发的信号的逻辑是一致的，都是通过InputChannel的receiveSignal函数来确认是否在管道中收到了某一个约定的字符值，不过，这里约定的字符值为INPUT\_SIGNAL\_FINISHED。

        回到前面的Step 40中，确认了是真的收到了键盘事件处理完成的信号后，就调用InputDispatcher的finishDispatchCycleLocked函数来执行一些善后工作了。

        Step 42. InputDispatcher.finishDispatchCycleLocked  
        这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **void** InputDispatcher::finishDispatchCycleLocked(nsecs\_t currentTime,
2. **const** sp<Connection>& connection) {
3. ......
5. // Notify other system components.
6. onDispatchCycleFinishedLocked(currentTime, connection);
8. // Reset the publisher since the event has been consumed.
9. // We do this now so that the publisher can release some of its internal resources
10. // while waiting for the next dispatch cycle to begin.
11. status\_t status = connection->inputPublisher.reset();
12. ......
14. startNextDispatchCycleLocked(currentTime, connection);
15. }

        这个函数主要就是做了三件事情：

       一是通知其它系统，InputDispatcher完成了一次键盘事件的处理：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Notify other system components.
2. onDispatchCycleFinishedLocked(currentTime, connection);

        二是调用相应的connection对象的内部对象inputPublisher来的reset函数来回收一些资源，它里面其实就是释放前面在Step 18（InputPublisher.publishKeyEvent）使用的匿名共享内存了：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. // Reset the publisher since the event has been consumed.
2. // We do this now so that the publisher can release some of its internal resources
3. // while waiting for the next dispatch cycle to begin.
4. status\_t status = connection->inputPublisher.reset();

        三是调用InputDispatcher的startNextDispatchCycleLocked函数来处理下一个键盘事件：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. startNextDispatchCycleLocked(currentTime, connection);

        因为正在处理当前这个键盘事件的时候，很有可能又同时发生了其它的键盘事件，因此，这里InputDispatcher还不能停下来，需要继续调用startNextDispatchCycleLocked继续处理键盘事件，不过下一个键盘事件的处理过程和我们现在分析的过程就是一样的了。

        至此，InputManager分发键盘消息给应用程序的过程就分析完成了，这是一个比较复杂的过程，不过，只要我们抓住主要的线索，就不难理解了，现在我们就小结一下这个过程的四个主要线索：

        A. 键盘事件发生，InputManager中的InputReader被唤醒，此前InputReader睡眠在/dev/input/event0这个设备文件上；

        B. InputReader被唤醒后，它接着唤醒InputManager中的InputDispatcher，此前InputDispatcher睡眠在InputManager所运行的线程中的Looper对象里面的管道的读端上；

        C. InputDispatcher被唤醒后，它接着唤醒应用程序的主线程来处理这个键盘事件，此前应用程序的主线程睡眠在Client端InputChannel中的前向管道的读端上；

        D. 应用程序处理处理键盘事件之后，它接着唤醒InputDispatcher来执行善后工作，此前InputDispatcher睡眠在Server端InputChannel的反向管道的读端上，注意这里与第二个线索处的区别。

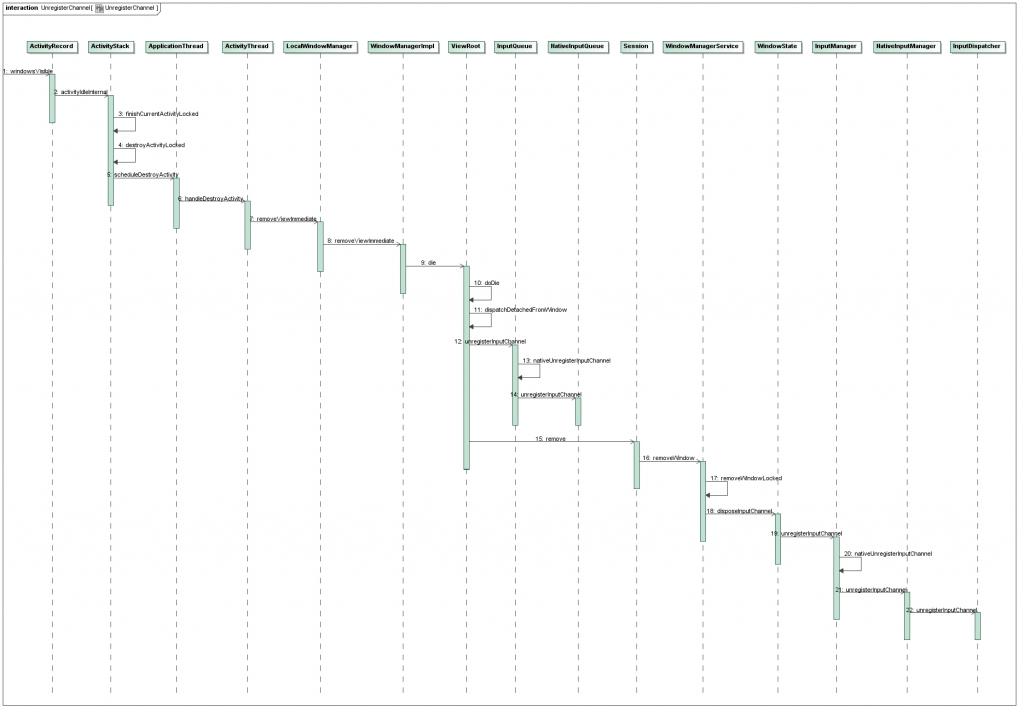
        4. 应用程序注销键盘消息接收通道的过程分析  
        当Activity窗口创建时，它会向InputManager注册键盘消息接收通道，而当Activity窗口销毁时，它就会向InputManager注销前面注册的键盘消息接收通道了，本节内容就来看看应用程序注销键盘消息接收通道的过程。

        当我们按下键盘上的Back键时，当前激活的Activity窗口就会被失去焦点，但是这时候它还没有被销毁，它的状态被设置为Stopped；当新的Activity窗口即将要显示时，它会通知WindowManagerService，这时候WindowManagerService就会处理当前处理Stopped状态的Activity窗口了，要执行的操作就是销毁它们了，在销毁的时候，就会注销它们之前所注册的键盘消息接收通道。

        新的Activity窗口通知WindowManagerService它即将要显示的过程比较复杂，但是它与我们本节要介绍的内容不是很相关，因此，这里就略过大部过程了，我们从ActvitiyRecord的windowsVisible函数开始分析。注意，这里的ActivityRecord是新的Activity窗口在ActivityManangerService的代表，而那些处于Stopped状态的Activity窗口

会放在ActivityStack类的一个等待可见的mWaitingVisibleActivities列表里面，事实于，对于那些Stopped状态的Activity窗口来说，它们是等待销毁，而不是等待可见。

        像前面一样，我们先来看一张应用程序注销键盘消息接收通道的过程的序列图，然后根据这个序列图来详细分析互一个步骤：



[点击查看大图](http://hi.csdn.net/attachment/201110/23/0_1319373166rRrp.gif)

        Step 1. ActivityRecord.windowsVisible

        这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityRecord.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **class** ActivityRecord **extends** IApplicationToken.Stub {
2. ......
3. **boolean** nowVisible;     // is this activity's window visible?
4. **boolean** idle;           // has the activity gone idle?
5. ......

8. **public** **void** windowsVisible() {
9. **synchronized**(service) {
10. ......
12. **if** (!nowVisible) {
13. nowVisible = **true**;
14. **if** (!idle) {
15. .......
16. } **else** {
17. // If this activity was already idle, then we now need to
18. // make sure we perform the full stop of any activities
19. // that are waiting to do so.  This is because we won't
20. // do that while they are still waiting for this one to
21. // become visible.
22. **final** **int** N = stack.mWaitingVisibleActivities.size();
23. **if** (N > 0) {
24. **for** (**int** i=0; i<N; i++) {
25. ActivityRecord r = (ActivityRecord)
26. stack.mWaitingVisibleActivities.get(i);
27. r.waitingVisible = **false**;
28. ......
29. }
30. stack.mWaitingVisibleActivities.clear();
32. Message msg = Message.obtain();
33. msg.what = ActivityStack.IDLE\_NOW\_MSG;
34. stack.mHandler.sendMessage(msg);
36. }
37. }
38. ......
39. }
40. }
41. }
43. ......
44. }

        应用程序中的每一个Activity在ActivityManagerService都有一个代表ActivityRecord，它们以堆栈的形式组织在ActivityManaerService中的ActivityStack中。一个即将要显示，但是还没有显示的Activity，它在ActivityManagerService中的ActivityRecord的成员变量nowVisible为false，而成员变量idle为ture，表示这个即将要显示的Activity窗口处于空闲状态。因此，在上面的这个函数中，会执行下面的语句：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **final** **int** N = stack.mWaitingVisibleActivities.size();
2. **if** (N > 0) {
3. **for** (**int** i=0; i<N; i++) {
4. ActivityRecord r = (ActivityRecord)
5. stack.mWaitingVisibleActivities.get(i);
6. r.waitingVisible = **false**;
7. ......
8. }
9. stack.mWaitingVisibleActivities.clear();
11. Message msg = Message.obtain();
12. msg.what = ActivityStack.IDLE\_NOW\_MSG;
13. stack.mHandler.sendMessage(msg);
15. }

        前面我们说过，当用户按下键盘上的Back键时，当前激活的Activity记录就被放在ActivityStack对象stack的成员变量mWaitingVisibleActivities中了，这时候就要对它进行处理了。首先是将它们的Activity记录的waitingVisible设置为false，然后就把它们从ActivityStack对象stack的成员变量mWaitingVisibleActivities清空，最后向ActivityStack对象stack发送一个ActivityStack.IDLE\_NOW\_MSG消息。这个消息最终是由ActivityStack类的activityIdleInternal函数来处理的。

        Step 2. ActivityStack.activityIdleInternal

        这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** ActivityStack {
2. ......
4. **final** **void** activityIdleInternal(IBinder token, **boolean** fromTimeout,
5. Configuration config) {
6. ......
8. ArrayList<ActivityRecord> stops = **null**;
9. ......
11. **int** NS = 0;
12. ......
14. **synchronized** (mService) {
15. ......
17. // Atomically retrieve all of the other things to do.
18. stops = processStoppingActivitiesLocked(**true**);
19. NS = stops != **null** ? stops.size() : 0;
20. ......
21. }
23. **int** i;
25. ......
27. // Stop any activities that are scheduled to do so but have been
28. // waiting for the next one to start.
29. **for** (i=0; i<NS; i++) {
30. ActivityRecord r = (ActivityRecord)stops.get(i);
31. **synchronized** (mService) {
32. **if** (r.finishing) {
33. finishCurrentActivityLocked(r, FINISH\_IMMEDIATELY);
34. } **else** {
35. ......
36. }
37. }
38. }
40. ......
41. }
43. ......
44. }

        这个函数首先会调用processStoppingActivitiesLocked函数把所有处于Stopped状态的Activity取回来，然后逐个分析它们，如果它们的ActivityRecord中的finishing成员变量为true，就说明这个Activity需要销毁了，于是，就调用finishCurrentActivityLocked函数来销毁它们。

        Step 3. ActivityStack.finishCurrentActivityLocked

        这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** ActivityStack {
2. ......
4. **private** **final** ActivityRecord finishCurrentActivityLocked(ActivityRecord r,
5. **int** mode) {
6. ......
8. **return** finishCurrentActivityLocked(r, index, mode);
9. }
11. **private** **final** ActivityRecord finishCurrentActivityLocked(ActivityRecord r,
12. **int** index, **int** mode) {
13. ......
15. // make sure the record is cleaned out of other places.
16. mStoppingActivities.remove(r);
17. mWaitingVisibleActivities.remove(r);
18. ......
20. **final** ActivityState prevState = r.state;
21. r.state = ActivityState.FINISHING;
23. **if** (mode == FINISH\_IMMEDIATELY
24. || prevState == ActivityState.STOPPED
25. || prevState == ActivityState.INITIALIZING) {
26. // If this activity is already stopped, we can just finish
27. // it right now.
28. **return** destroyActivityLocked(r, **true**) ? **null** : r;
29. } **else** {
30. ......
31. }
33. **return** r;
34. }
36. ......
37. }

        从上面的Step 2中传进来的参数mode为FINISH\_IMMEDIATELY，并且这个即将要被销毁的Activity的状态为Stopped，因此，接下来就会调用destroyActivityLocked函数来销毁它。

        Step 4. ActivityStack.destroyActivityLocked

        这个函数定义在frameworks/base/services/java/com/android/server/am/ActivityStack.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** ActivityStack {
2. ......
4. **final** **boolean** destroyActivityLocked(ActivityRecord r,
5. **boolean** removeFromApp) {
6. ......
8. **boolean** removedFromHistory = **false**;
10. ......
12. **final** **boolean** hadApp = r.app != **null**;
14. **if** (hadApp) {
15. ......
17. **try** {
18. ......
19. r.app.thread.scheduleDestroyActivity(r, r.finishing,
20. r.configChangeFlags);
21. } **catch** (Exception e) {
22. ......
23. }
25. ......
26. } **else** {
27. ......
28. }
30. ......
32. **return** removedFromHistory;
33. }
35. ......
36. }

         在前面一篇文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)中，我们说到，每一个应用程序进程在ActivityManagerService中，都ProcessRecord记录与之对应，而每一个Activity，都是运行在一个进程上下文中，因此，在ActivityManagerService中，每一个ActivityRecord的app成员变量都应该指向一个ProcessRecord记录，于是，这里得到的hadApp为true。在ProcessRecord类中，有一个成员变量thread，它的类型为IApplicationThread。在文章[Android应用程序启动过程源代码分析](http://blog.csdn.net/luoshengyang/article/details/6689748)中，我们也曾经说过，每一个应用程序在启动的时候，它都会在内部创建一个ActivityThread对象，而在这个ActivityThread对象中，有一个成员变量mAppThread，它的类型为ApplicationThread，这是一个Binder对象，专门用来负责在应用程序和ActivityManagerService之间执行进程间通信工作的。应用程序在启动的时候，就会将这个Binder对象传递给ActivityManagerService，而ActivityManagerService就会把它保存在相应的ProcessRecord记录的thread成员变量中。因此，ProcessRecord记录的thread成员变量其实就是ApplicationThread对象的远程接口，于是，执行下面这个语句的时候：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. r.app.thread.scheduleDestroyActivity(r, r.finishing,
2. r.configChangeFlags);

        就会进入到ApplicationThread类中的scheduleDestroyActivity函数来。

        Step 5. ApplicationThread.scheduleDestroyActivity

        这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ActivityThread {
2. ......
4. **private** **final** **class** ApplicationThread **extends** ApplicationThreadNative {
5. ......
7. **public** **final** **void** scheduleDestroyActivity(IBinder token, **boolean** finishing,
8. **int** configChanges) {
10. queueOrSendMessage(H.DESTROY\_ACTIVITY, token, finishing ? 1 : 0,
11. configChanges);
12. }
14. ......
15. }
17. ......
18. }

        这个函数调用外部类ActivityThread的queueOrSendMessage函数来往应用程序的消息队列中发送一个H.DESTROY\_ACTIVITY消息，这个消息最终由ActivityThread类的handleDestroyActivity函数来处理。

        Step 6. ActivityThread.handleDestroyActivity

        这个函数定义在frameworks/base/core/java/android/app/ActivityThread.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ActivityThread {
2. ......
4. **private** **final** **void** handleDestroyActivity(IBinder token, **boolean** finishing,
5. **int** configChanges, **boolean** getNonConfigInstance) {
6. ......
8. ActivityClientRecord r = performDestroyActivity(token, finishing,
9. configChanges, getNonConfigInstance);
10. **if** (r != **null**) {
11. WindowManager wm = r.activity.getWindowManager();
12. View v = r.activity.mDecor;
13. **if** (v != **null**) {
14. ......
16. **if** (r.activity.mWindowAdded) {
17. wm.removeViewImmediate(v);
18. }
20. ......
21. }
22. ......
23. }
25. ......
26. }
28. ......
29. }

        这里首先调用performDestroyActivity来执行一些销毁Activity的操作，期间就会调用Activity的onDestroy函数让Activity本身有机会执行一些销毁前的工作了。这里通过r.activity.getWindowManager函数返回的是一个LocalWindowManager对象，而通过r.activity.mDecor得到的是一个DecorView对象，这些都是在Activity启动的时候设置好的。函数最后调用LocalWindowManager对象wm的removeViewImmediate函员来从LocalWindowManager移除这个DecorView对象。

        Step 7. LocalWindowManager.removeViewImmediate

        这个函数定义在frameworks/base/core/java/android/view/Window.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **abstract** **class** Window {
2. ......
4. **private** **class** LocalWindowManager **implements** WindowManager {
5. ......
7. **public** **final** **void** removeViewImmediate(View view) {
8. mWindowManager.removeViewImmediate(view);
9. }
11. ......
13. **private** **final** WindowManager mWindowManager;
14. }
16. ......
17. }

        LocalWindowManager类的成员变量mWindowManager是一个WndowManagerImpl对象，这个函数只是简单地调用WndowManagerImpl类的removeViewImmediate来进一步处理。

       Step 8. WndowManagerImpl.removeViewImmediate

       这个函数定义在frameworks/base/core/java/android/view/WindowManagerImpl.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerImpl **implements** WindowManager {
2. ......
4. **public** **void** removeViewImmediate(View view) {
5. **synchronized** (**this**) {
6. **int** index = findViewLocked(view, **true**);
7. ViewRoot root = mRoots[index];
8. ......
10. root.die(**true**);
12. ......
13. }
14. }
16. ......
17. }

         这个函数首先是找到这个view所属的ViewRoot对象root，然后调用这个root对象的die函数来销毁它。

         Step 9. ViewRoot.die

         这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **public** **void** die(**boolean** immediate) {
6. **if** (immediate) {
7. doDie();
8. } **else** {
9. ......
10. }
11. }
13. ......
14. }

        上面Step 8传进来的immediate参数为true，因此，这里直接调用doDie函数来进一步处理。

        Step 10. ViewRoot.doDie

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **void** doDie() {
6. ......
8. **synchronized** (**this**) {
9. ......
11. **if** (mAdded) {
12. mAdded = **false**;
13. dispatchDetachedFromWindow();
14. }
15. }
16. }
18. ......
19. }

        当我们把Activity窗口中的View添加到一个ViewRoot对象时，就会把它的成员变量mAdded设置为true，这样就表示这个ViewRoot中有View存在，于是，这里就会调用dispatchDetachedFromWindow函数来进一步处理。

        Step 11. ViewRoot.ispatchDetachedFromWindow

        这个函数定义在frameworks/base/core/java/android/view/ViewRoot.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** ViewRoot **extends** Handler **implements** ViewParent,
2. View.AttachInfo.Callbacks {
3. ......
5. **void** dispatchDetachedFromWindow() {
6. ......
8. **if** (mInputChannel != **null**) {
9. **if** (mInputQueueCallback != **null**) {
10. ......
11. } **else** {
12. InputQueue.unregisterInputChannel(mInputChannel);
13. }
14. }
16. **try** {
17. sWindowSession.remove(mWindow);
18. } **catch** (RemoteException e) {
19. }
21. ......
22. }
24. ......
25. }

        前面在介绍应用程序注册键盘消息接收通道的过程时，在Step 18，我们说到，ViewRoot类中的mInputQueueCallback为null，表示由这个ViewRoot自己来管理键盘输入事件，因此，这里首先会调用InputQueue的unregisterInputChannel函数来注销注册在应用程序这一侧的Client端InputChannel，然后再调用sWindowSession的remove函数来注销注册在InputManager这一侧的Server端InputChannel，这个逻辑是和前面介绍应用程序注册键盘消息接收通道的逻辑相对应的，前面分别注册了这两个InputChannel，现在Activity要销毁了，当然就要把它们注销了。

        我们先来看注销注册在应用程序这一侧的Client端InputChannel，然后再回过头来分析注销注册在InputManager这一侧的Server端InputChannel。

        Step 12. InputQueue.unregisterInputChannel

        这个函数定义在frameworks/base/core/java/android/view/InputQueue.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **final** **class** InputQueue {
2. ......
4. **public** **static** **void** unregisterInputChannel(InputChannel inputChannel) {
5. ......
7. **synchronized** (sLock) {
8. ......
10. nativeUnregisterInputChannel(inputChannel);
11. }
12. }
14. ......
15. }

         这个函数只是简单地调用本地方法nativeUnregisterInputChannel来执行具体的操作。

         Step 13. InputQueue.nativeUnregisterInputChannel  
         这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_view\_InputQueue\_nativeUnregisterInputChannel(JNIEnv\* env, jclass clazz,
2. jobject inputChannelObj) {
3. status\_t status = gNativeInputQueue.unregisterInputChannel(env, inputChannelObj);
5. ......
6. }

        这里调用NativeInputQueue的成员函数unregisterInputChannel来进一步处理。

        Step 14. NativeInputQueue.unregisterInputChannel  
        这个函数定义在frameworks/base/core/jni/android\_view\_InputQueue.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t NativeInputQueue::unregisterInputChannel(JNIEnv\* env, jobject inputChannelObj) {
2. sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,
3. inputChannelObj);
4. ......
6. { // acquire lock
7. AutoMutex \_l(mLock);
9. ssize\_t connectionIndex = getConnectionIndex(inputChannel);
10. ......
12. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
13. mConnectionsByReceiveFd.removeItemsAt(connectionIndex);
15. connection->status = Connection::STATUS\_ZOMBIE;
17. connection->looper->removeFd(inputChannel->getReceivePipeFd());
19. env->DeleteGlobalRef(connection->inputHandlerObjGlobal);
20. connection->inputHandlerObjGlobal = NULL;
21. ......
22. } // release lock
24. ......
25. **return** OK;
26. }

        真正的注销工作就是这里实现的了，读者可以对照前面介绍应用程序注册键盘消息接收通道过程中的Step 21（NativeInputQueue.registerInputChannel）来分析，它首先是将在之前创建的Connection对象从NativeInputQueue中的mConnectionByReceiveFd向量中删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. ssize\_t connectionIndex = getConnectionIndex(inputChannel);
2. ......
4. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
5. mConnectionsByReceiveFd.removeItemsAt(connectionIndex);

        然后再把这个Client端InputChannel的前向管道的读端文件描述符从应用程序主线程中的Looper对象中删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. connection->looper->removeFd(inputChannel->getReceivePipeFd());

        这样，这个Activity窗口以后就不会接收到键盘事件了。

        最后将Connection对象中的回调对象inputHandlerOjbGlobal对象删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. env->DeleteGlobalRef(connection->inputHandlerObjGlobal);
2. connection->inputHandlerObjGlobal = NULL;

        回忆一下前面我们在分析InputManager分发键盘消息给应用程序处理时，曾经说到，每当有键盘事件发生时，InputManager首先就会调用NativeInputQueue类的handleReceiveCallback函数。在这个handleReceiveCallback函数里面，NativeInputQueue会找到相应的Connection对象，然后把它里面的内部对象inputHandlerOjbGlobal作为参数来调用Java层的InputQueue类的dispatchKeyEvent函数来通知应用程序，有键盘事件发生了。在InputQueue类的dispatchKeyEvent函数里面，就是通过这个inputHandlerOjbGlobal对象来直正通知到当前激活的Activity窗口来处理这个键盘事件的。

        注册在应用程序这一侧的Client端InputChannel被注销以后，回到前面的Step 11中，我们继续分析注销注册在InputManager这一侧的Server端InputChannel。

        Step 15. WindowManagerService.Session.remove

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **class** Session **extends** IWindowSession.Stub
6. **implements** IBinder.DeathRecipient {
7. ......
9. **public** **void** remove(IWindow window) {
10. removeWindow(**this**, window);
11. }
13. ......
14. }
16. ......
17. }

        这个函数只是简单地调用其外部类WindowManagerService的removeWindow函数来进一步执行操作。

        Step 16. WindowManagerService.removeWindow  
        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **void** removeWindow(Session session, IWindow client) {
6. **synchronized**(mWindowMap) {
7. WindowState win = windowForClientLocked(session, client, **false**);
8. **if** (win == **null**) {
9. **return**;
10. }
11. removeWindowLocked(session, win);
12. }
13. }
15. ......
16. }

        回忆一下前面我们在分析应用程序注册键盘消息管道的过程时，在Step 11（WindowManagerService.addWindow）中，WindowManagerService为这个即将要激活的Activity窗口创建了一个WindowState对象win，创建的时候，使用了从ViewRoot中传过来的两个参数，分别是一个Session对象session和一个IWindow对象client。

       在这个函数中，ViewRoot传过来的两个参数session和client和上面说的两个参数是一致的，因此，这个函数首先通过参数session和client得到一个WindowState对象win，然后调用removeWindowLocked来把它从WindowManagerService删除。

        Step 17. WindowManagerService.removeWindowLocked  
        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **public** **void** removeWindowLocked(Session session, WindowState win) {
6. ......
8. win.disposeInputChannel();
10. ......
11. }
13. ......
14. }

        我们忽略了这个函数的其它逻辑，只关注注销之前注册的Server端InputChannel的逻辑，这里，注销的操作就是调用win的disposeInputChannel进行的了。

        Step 18. WindowState.disposeInputChannel

        这个函数定义在frameworks/base/services/java/com/android/server/WindowManagerService.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** WindowManagerService **extends** IWindowManager.Stub
2. **implements** Watchdog.Monitor {
3. ......
5. **private** **final** **class** WindowState **implements** WindowManagerPolicy.WindowState {
6. ......
8. **void** disposeInputChannel() {
9. **if** (mInputChannel != **null**) {
10. mInputManager.unregisterInputChannel(mInputChannel);
12. mInputChannel.dispose();
13. mInputChannel = **null**;
14. }
15. }
17. ......
18. }
20. ......
21. }

        上面说到，在前面分析应用程序注册键盘消息管道的过程时，在Step 11（WindowManagerService.addWindow）中，为当前这个Activity窗口创建了一个WindowState对象，接着创建了一个输入管道后，把Server端的InputChannel保存了在这个WindowState对象的成员变量mInputChannel中，因此，这里，就可以把它取回来，然后调用mInputManager对象的unregisterInputChannel函数来把它注销掉了。

        Step 19. InputManager.unregisterInputChannel

        这个函数定义在frameworks/base/services/java/com/android/server/InputManager.java文件中：

**[java]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **public** **class** InputManager {
2. ......
4. **public** **void** unregisterInputChannel(InputChannel inputChannel) {
5. ......
7. nativeUnregisterInputChannel(inputChannel);
8. }
10. ......
11. }

         这个函数很简单，它调用本地方法nativeUnregisterInputChannel来进一步处理。

         Step 20. InputManager.nativeUnregisterInputChannel

         这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. **static** **void** android\_server\_InputManager\_nativeUnregisterInputChannel(JNIEnv\* env, jclass clazz,
2. jobject inputChannelObj) {
3. ......

6. sp<InputChannel> inputChannel = android\_view\_InputChannel\_getInputChannel(env,
7. inputChannelObj);
8. ......

11. status\_t status = gNativeInputManager->unregisterInputChannel(env, inputChannel);
13. ......
14. }

        这个函数首先调用android\_view\_InputChannel\_getInputChannel函数根据Java层的InputChannel对象找到C++层的InputChannel对象，然后调用NativeInputManager的unregisterInputChannel函数来执行注销的操作。

        Step 21. NativeInputManager.unregisterInputChannel  
        这个函数定义在frameworks/base/services/jni/com\_android\_server\_InputManager.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t NativeInputManager::unregisterInputChannel(JNIEnv\* env,
2. **const** sp<InputChannel>& inputChannel) {
3. ......
5. **return** mInputManager->getDispatcher()->unregisterInputChannel(inputChannel);
6. }

       这个函数与前面分析应用程序注册键盘消息通道的Step 17（NativeInputManager.registerInputChannel）相对应，主要是调用InputDispatcher对象的unregisterInputChannel函数来执行真正注销的操作。

       Step 22. InputDispatcher.unregisterInputChannel  
       这个函数定义在frameworks/base/libs/ui/InputDispatcher.cpp文件中：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. status\_t InputDispatcher::unregisterInputChannel(**const** sp<InputChannel>& inputChannel) {
2. ......
4. { // acquire lock
5. AutoMutex \_l(mLock);
7. ssize\_t connectionIndex = getConnectionIndexLocked(inputChannel);
8. ......
10. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
11. mConnectionsByReceiveFd.removeItemsAt(connectionIndex);
13. ......
15. mLooper->removeFd(inputChannel->getReceivePipeFd());
17. .....
19. } // release lock
21. ......
23. **return** OK;
24. }

        这一步与前面的Step 14注销应用程序一侧的Client端InputChannel是差不多的，只不过这里是从InputDispatcher中把Server端的InputChannel注销掉。首先是根据传进来的参数inputChannel找到它在InputDispatcher中对应的Connection对象在mConnectionsByReceiveFd中的索引，然后把它从mConnectionsByReceiveFd中删除：

**[cpp]** [view plaincopy](http://blog.csdn.net/luoshengyang/article/details/6882903)

1. ssize\_t connectionIndex = getConnectionIndexLocked(inputChannel);
2. ......
4. sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
5. mConnectionsByReceiveFd.removeItemsAt(connectionIndex);

        最后，还需要把这个InputChannel中的反向管道读端文件描述符从InputDispatcher的内部对象mLooper中删除，因为这个文件描述符是在前面注册Server端的InputChannel时加入到mLooper对象去的，具体可以参考上面分析应用程序注册键盘消息接收通道的过程中的Step 18（InputDispatcher.registerInputChannel）。

        这样， 应用程序注销键盘消息接收通道的过程就分析完成了，整个应用程序键盘消息处理机制也分析完成了，这是一个比较复杂的过程，要完全理解它还需要花费一些努力和时间，不过，理解了这个过程之后，对Android应用程序框架层的理解就更进一步了。