Android应用程序输入事件处理机制

罗升阳

http://weibo.com/shengyangluo

http://blog.csdn.net/luoshengyang

About Me

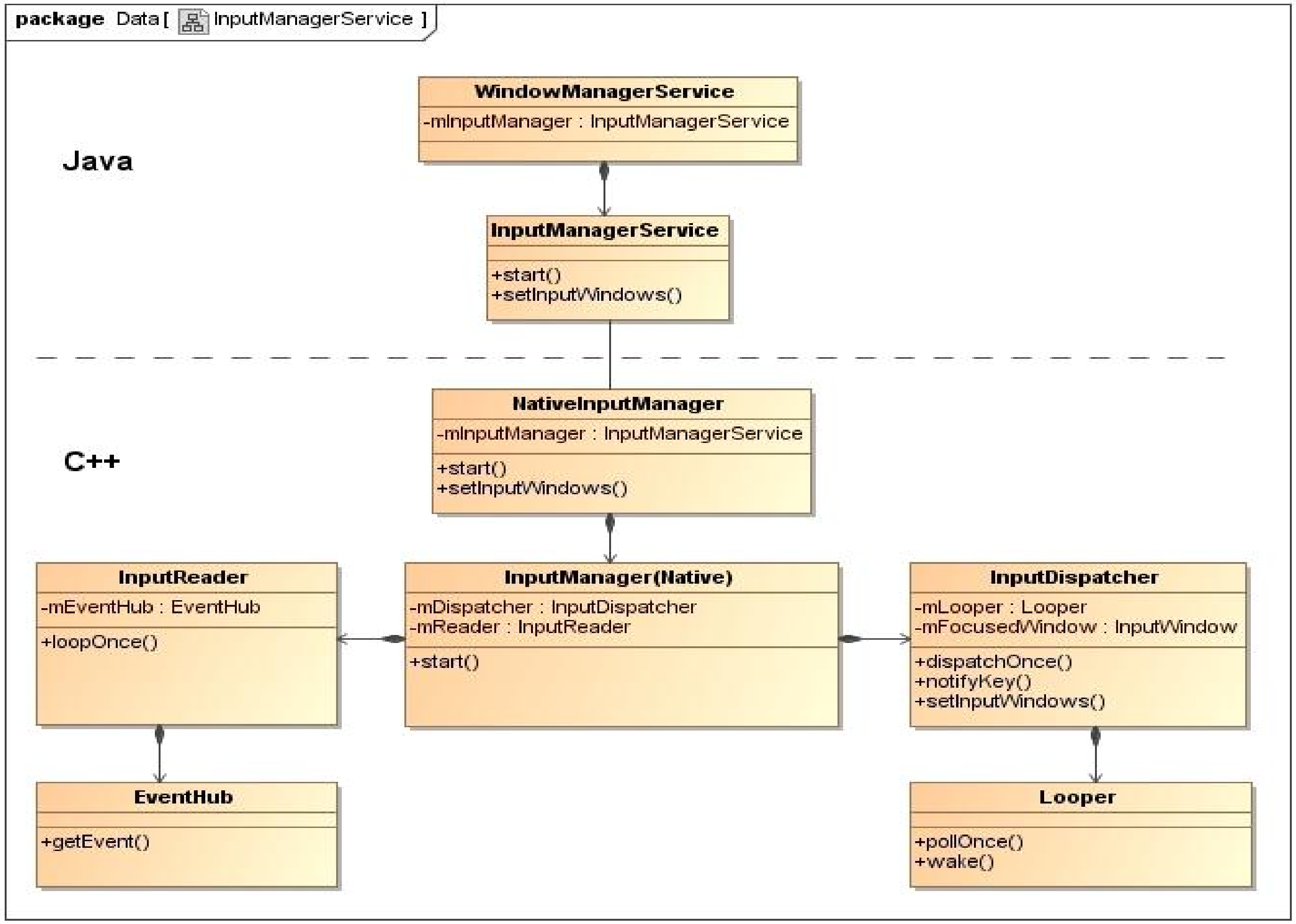
- 《老罗的Android之旅》博客作者
- 《Android系统源代码情景分析》书籍作者
- 博客：http://blog.csdn.net/Luoshengyang
- 微博：http://weibo.com/shengyangluo

Agenda

- Android输入系统概述
- 输入管理器的启动过程
- 输入通道的注册过程
- 输入事件的分发过程
- 软键盘输入事件的分发过程

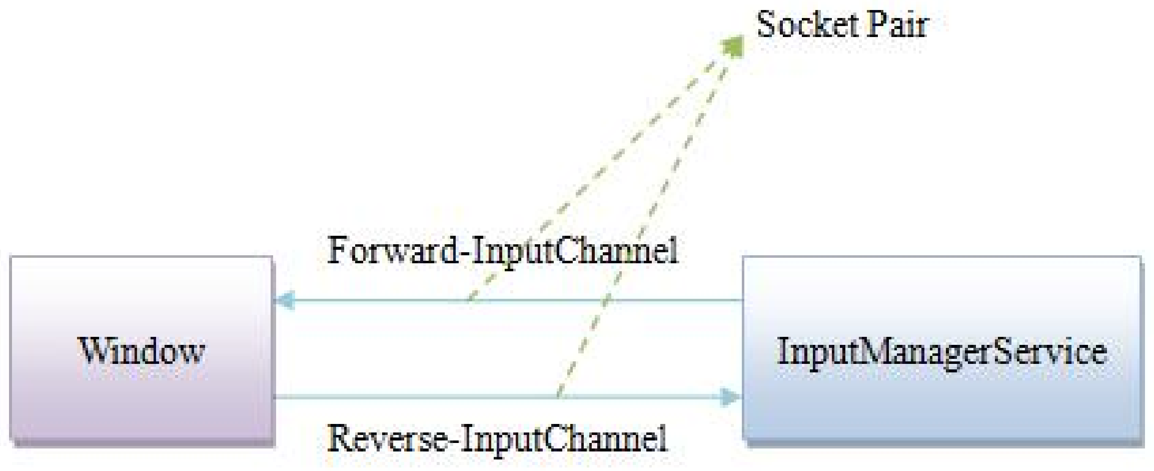
Android输入系统概述

- 输入管理器框架



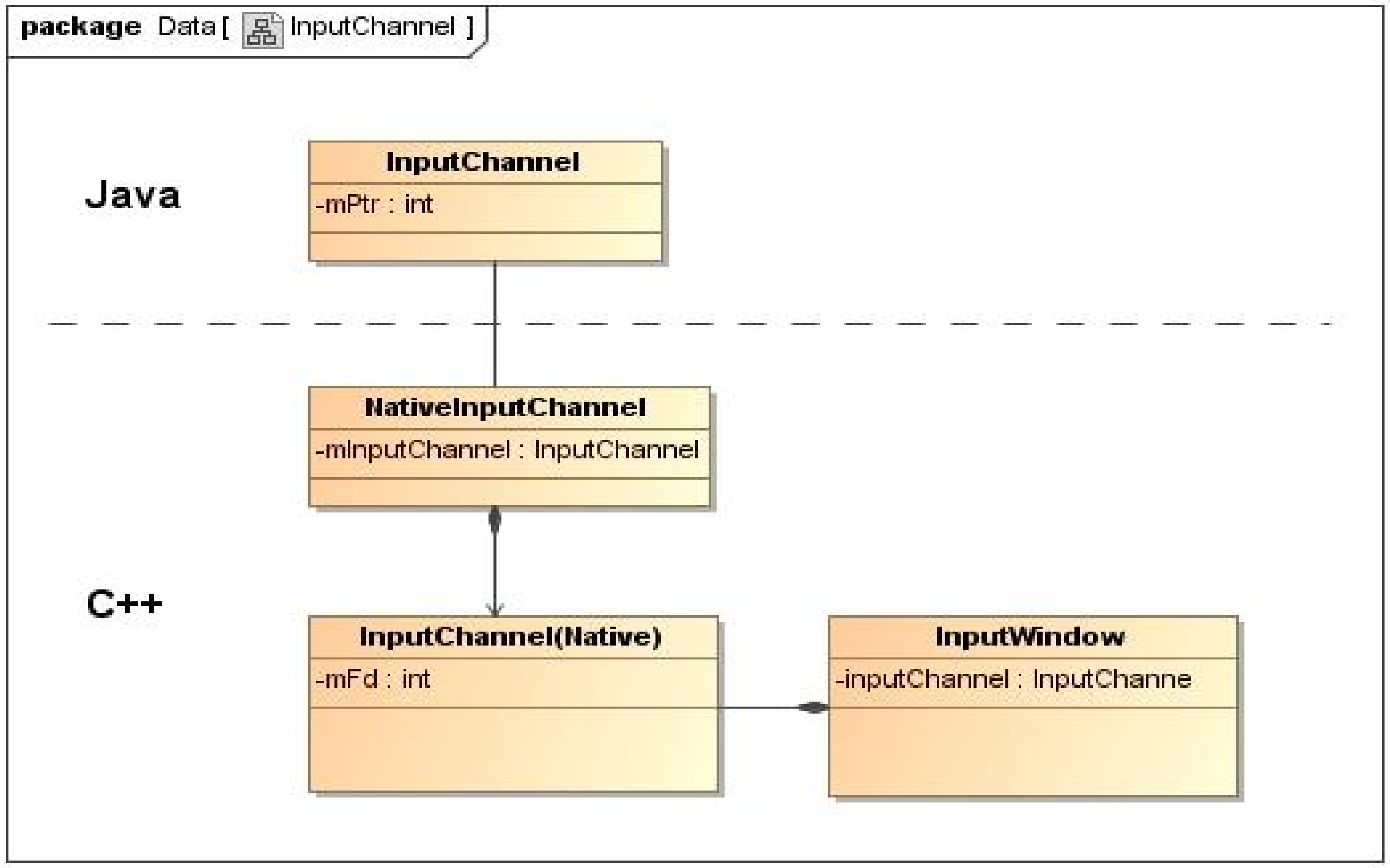
Android输入系统概述(续)

- 输入管理器与应用程序通过输入通道交互



Android输入系统概述(续)

- 输入通道



输入管理器的启动过程

- 由System Server创建和启动

```
class ServerThread extends Thread {
    ......
    @Override
    public void run() {
        ......

        WindowManagerService wm = null;
        .......
        InputManagerService inputManager = null;
        ......

        try {
            ......

            inputManager = new InputManagerService(context, wmHandler);

            wm = WindowManagerService.main(context, power, display, inputManager,
                    uiHandler, wmHandler,
                    factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL,
                    !firstBoot, onlyCore);

            ServiceManager.addService(Context.INPUT_SERVICE, inputManager);
            ......

            inputManager.setWindowManagerCallbacks(wm.getInputMonitor());
            inputManager.start();
            ......

        } catch (RuntimeException e) {
        }
        ......
    }
    ......
}
```

输入管理器的启动过程(续)

- 创建InputManagerService

```
public class InputManagerService extends IInputManager.Stub
        implements Watchdog.Monitor, DisplayManagerService.InputManagerFuncs {
    ......

    // Pointer to native input manager service object.
    private final int mPtr;
    ......

    public InputManagerService(Context context, Handler handler) {
        ......

        mPtr = nativeInit(this, mContext, mHandler.getLooper().getQueue());
    }

    ......
}
```

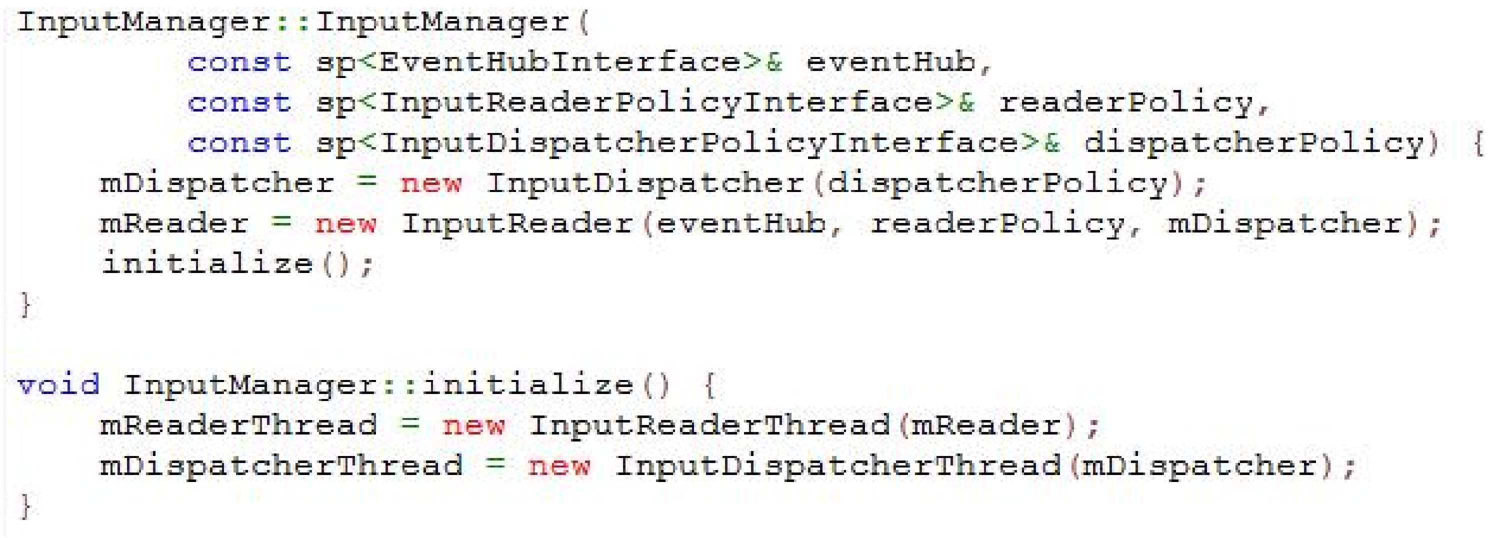
输入管理器的启动过程(续)

- 创建NativeInputManager

```
static jint nativeInit(JNIEnv* env, jclass clazz,
        jobject serviceObj, jobject contextObj, jobject messageQueueObj) {
    sp<MessageQueue> messageQueue = android_os_MessageQueue_getMessageQueue(env, messageQueueObj);
    NativeInputManager* im = new NativeInputManager(contextObj, serviceObj,
            messageQueue->getLooper());
    im->incStrong(serviceObj);
    return reinterpret_cast<jint>(im);
}

NativeInputManager::NativeInputManager(jobject contextObj,
        jobject serviceObj, const sp<Looper>& looper) :
        mLooper(looper) {
     ......

    sp<EventHub> eventHub = new EventHub();
    mInputManager = new InputManager(eventHub, this, this);
}
```

输入管理器的启动过程(续)

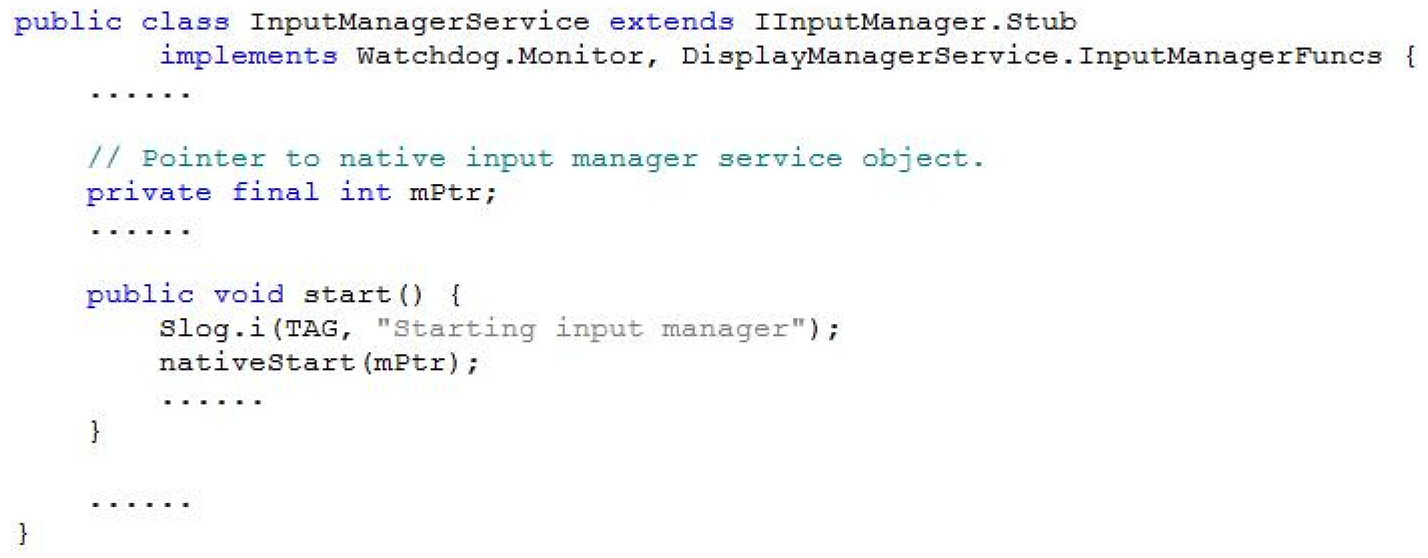
- 创建InputManager、InputReader和InputDispatcher，以及InputReaderThread、InputDispatcherThread



```
InputManager::InputManager(
        const sp<EventHubInterface>& eventHub,
        const sp<InputReaderPolicyInterface>& readerPolicy,
        const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
    mDispatcher = new InputDispatcher(dispatcherPolicy);
    mReader = new InputReader(eventHub, readerPolicy, mDispatcher);
    initialize();
}

void InputManager::initialize() {
    mReaderThread = new InputReaderThread(mReader);
    mDispatcherThread = new InputDispatcherThread(mDispatcher);
}
```

输入管理器的启动过程(续)

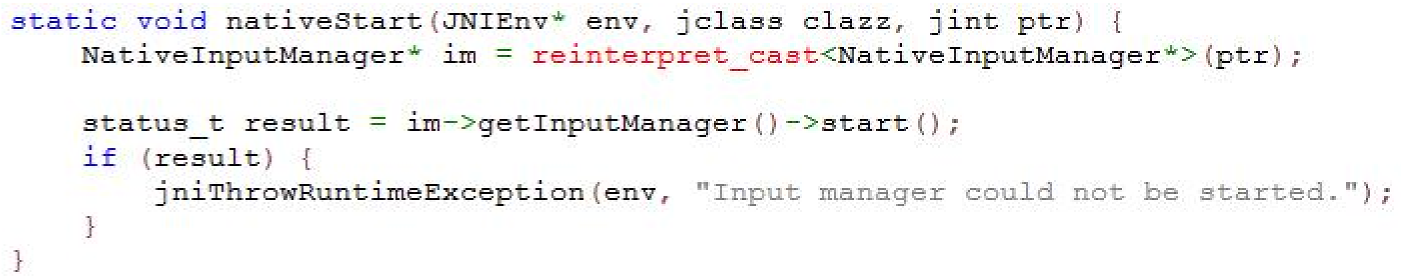
- 启动InputManagerService



```
public class InputManagerService extends IInputManager.Stub
        implements Watchdog.Monitor, DisplayManagerService.InputManagerFuncs {
    ......

    // Pointer to native input manager service object.
    private final int mPtr;
    ......

    public void start() {
        Slog.i(TAG, "Starting input manager");
        nativeStart(mPtr);
        ......
    }

    ......
}
```

输入管理器的启动过程(续)

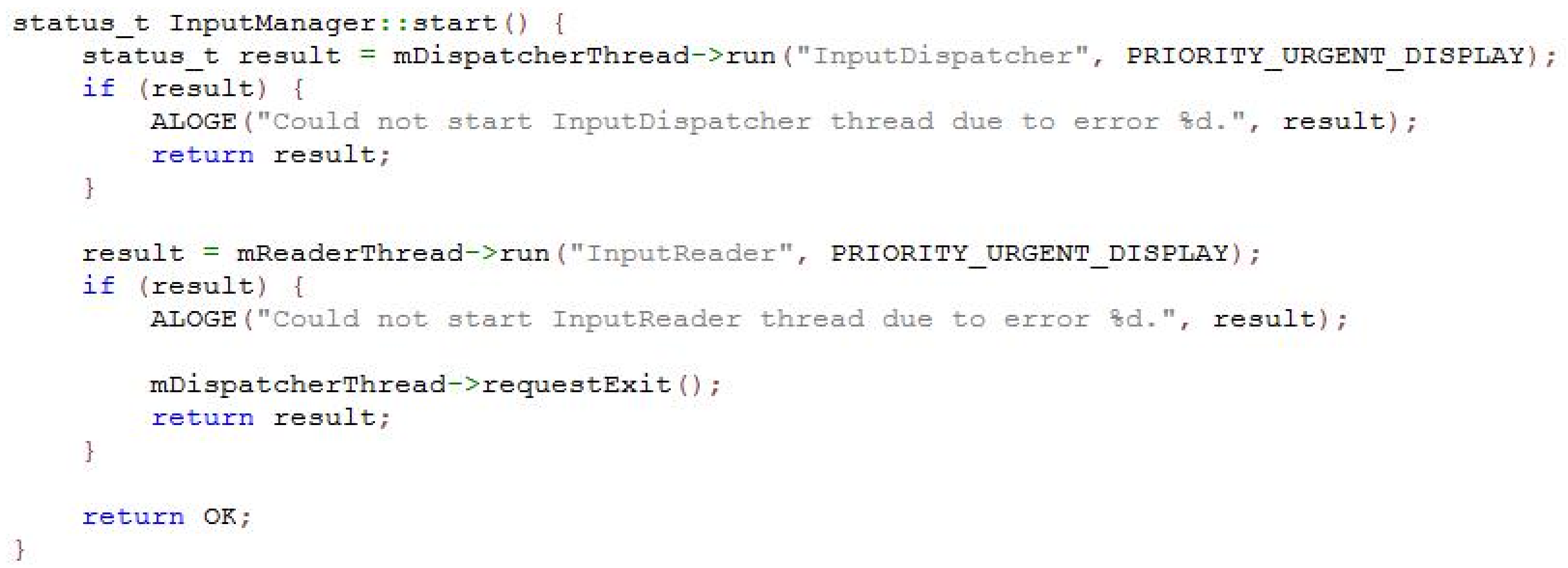
- 启动NativeInputManager



```
static void nativeStart(JNIEnv* env, jclass clazz, jint ptr) {
    NativeInputManager* im = reinterpret_cast<NativeInputManager*>(ptr);

    status_t result = im->getInputManager()->start();
    if (result) {
        jniThrowRuntimeException(env, "Input manager could not be started.");
    }
}
```

输入管理器的启动过程(续)

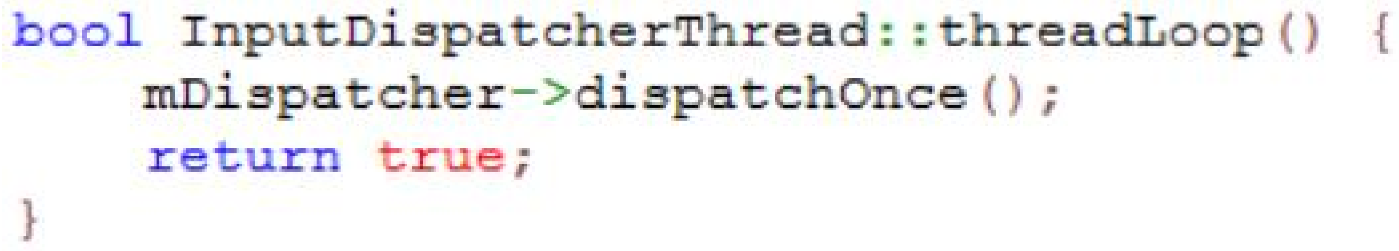
- 启动InputManager



```
status_t InputManager::start() {
    status_t result = mDispatcherThread->run("InputDispatcher", PRIORITY_URGENT_DISPLAY);
    if (result) {
        ALOGE("Could not start InputDispatcher thread due to error %d.", result);
        return result;
    }

    result = mReaderThread->run("InputReader", PRIORITY_URGENT_DISPLAY);
    if (result) {
        ALOGE("Could not start InputReader thread due to error %d.", result);

        mDispatcherThread->requestExit();
        return result;
    }

    return OK;
}
```

输入管理器的启动过程(续)

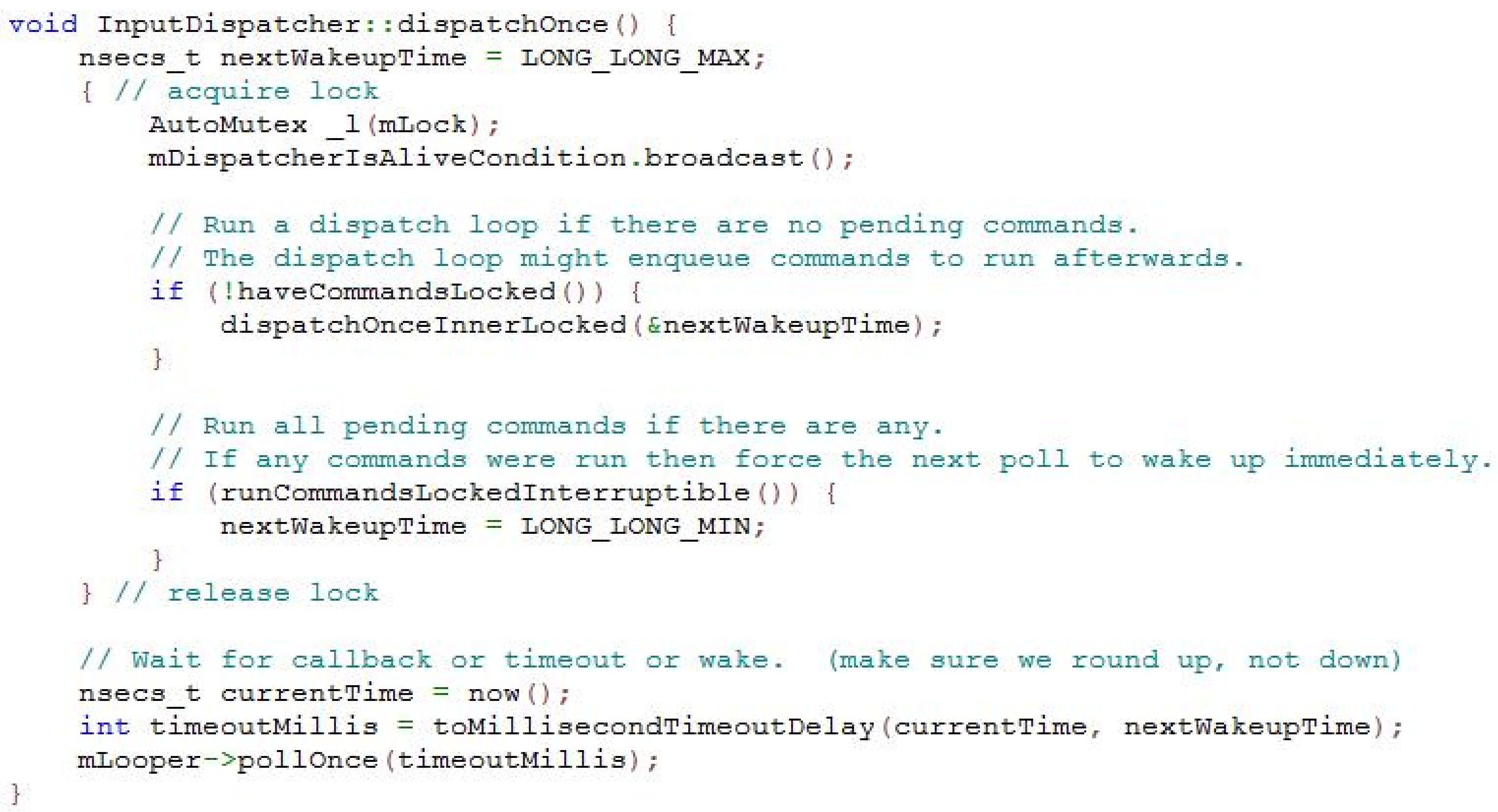
- 启动InputDispatcher
  - InputDispatcherThread.threadLoop



```
bool InputDispatcherThread::threadLoop() {
    mDispatcher->dispatchOnce();
    return true;
}
```

输入管理器的启动过程(续)

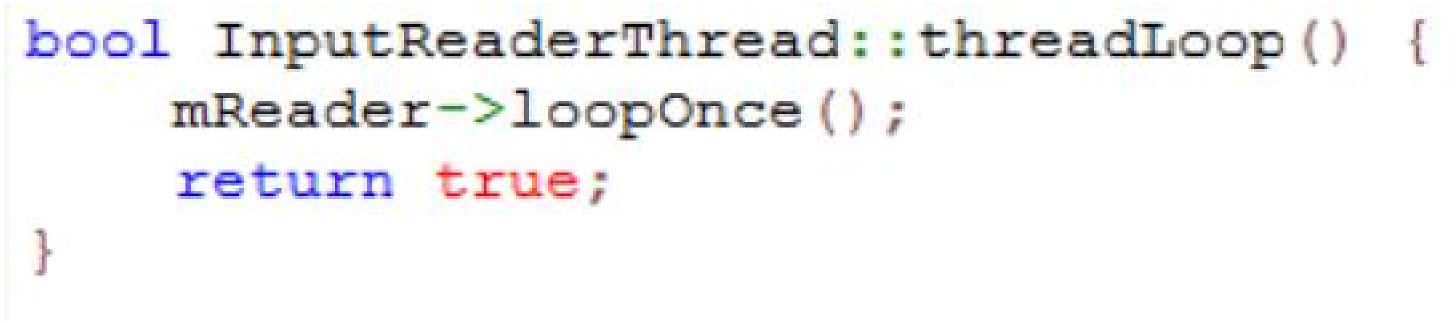
- 启动InputDispatcher
  - InputDispatcher.dispatchOnce



```
void InputDispatcher::dispatchOnce() {
    nsecs_t nextWakeupTime = LONG_LONG_MAX;
    { // acquire lock
        AutoMutex _l(mLock);
        mDispatcherIsAliveCondition.broadcast();

        // Run a dispatch loop if there are no pending commands.
        // The dispatch loop might enqueue commands to run afterwards.
        if (!haveCommandsLocked()) {
            dispatchOnceInnerLocked(&nextWakeupTime);
        }

        // Run all pending commands if there are any.
        // If any commands were run then force the next poll to wake up immediately.
        if (runCommandsLockedInterruptible()) {
            nextWakeupTime = LONG_LONG_MIN;
        }
    } // release lock

    // Wait for callback or timeout or wake.  (make sure we round up, not down)
    nsecs_t currentTime = now();
    int timeoutMillis = toMillisecondTimeoutDelay(currentTime, nextWakeupTime);
    mLooper->pollOnce(timeoutMillis);
}
```

输入管理器的启动过程(续)

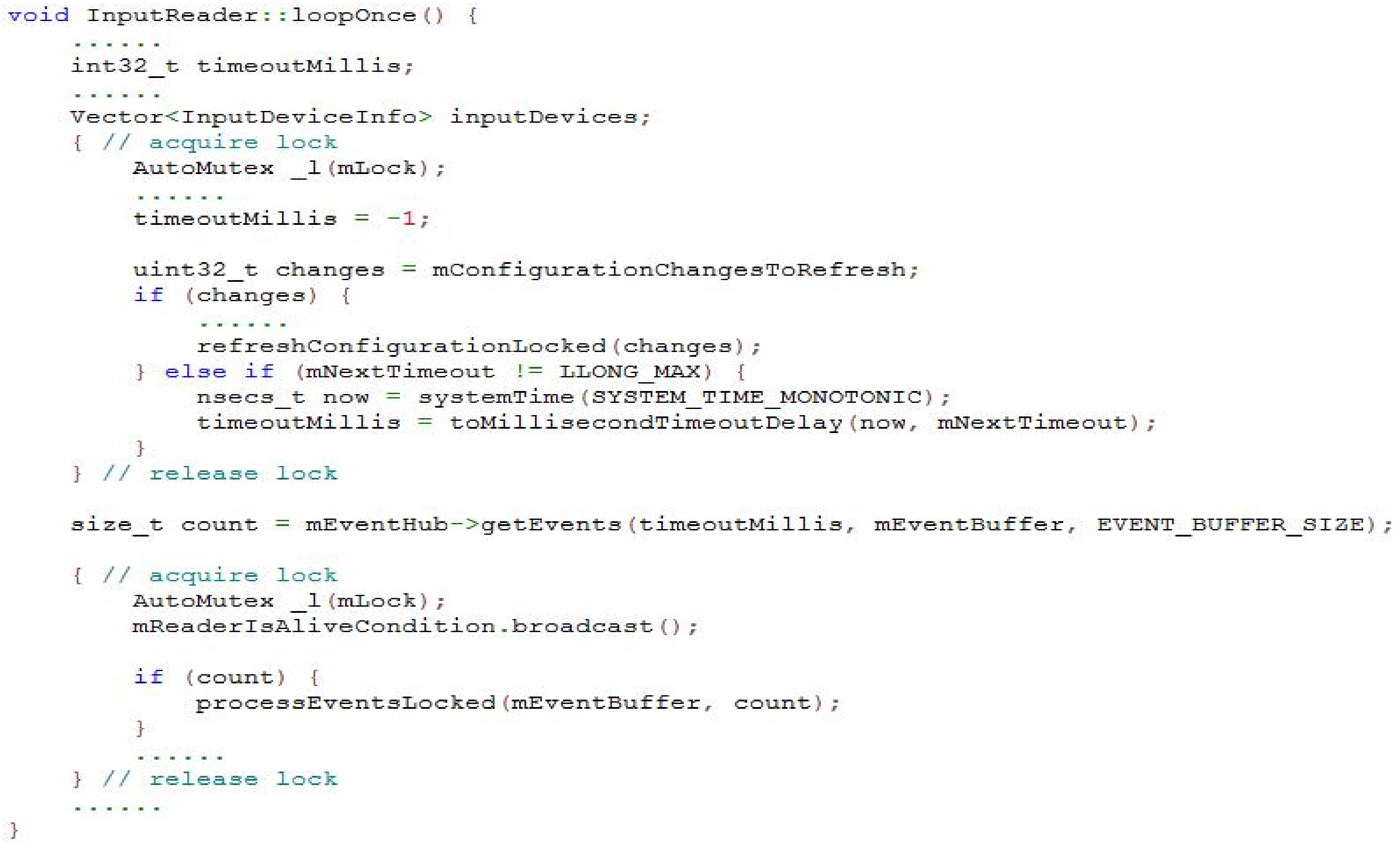
- 启动InputReader
  - InputReaderThread.threadLoop



```
bool InputReaderThread::threadLoop() {
    mReader->loopOnce();
    return true;
}
```

输入管理器的启动过程(续)

- 启动InputReader
  - InputReader.loopOnce



```
void InputReader::loopOnce() {
    ......
    int32_t timeoutMillis;
    ......
    Vector<InputDeviceInfo> inputDevices;
    { // acquire lock
        AutoMutex _l(mLock);
        ......
        timeoutMillis = -1;

        uint32_t changes = mConfigurationChangesToRefresh;
        if (changes) {
            ......
            refreshConfigurationLocked(changes);
        } else if (mNextTimeout != LLONG_MAX) {
            nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
            timeoutMillis = toMillisecondTimeoutDelay(now, mNextTimeout);
        }
    } // release lock

    size_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT_BUFFER_SIZE);

    { // acquire lock
        AutoMutex _l(mLock);
        mReaderIsAliveCondition.broadcast();

        if (count) {
            processEventsLocked(mEventBuffer, count);
        }
        ......
    } // release lock
    ......
}
```

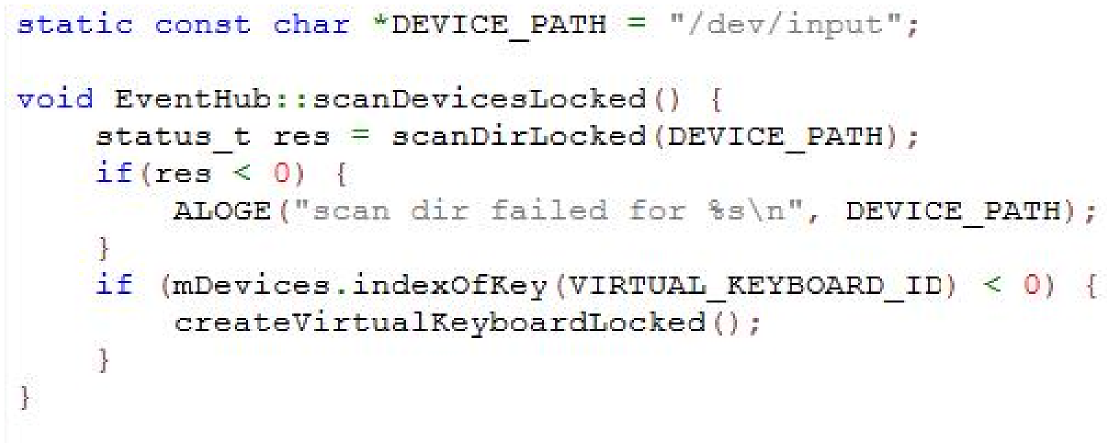
输入管理器的启动过程(续)

- 启动InputReader
  - EventHub.getEvents

```
size_t EventHub::getEvents(int timeoutMillis, RawEvent* buffer, size_t bufferSize) {
    ......
    RawEvent* event = buffer;
    ......
    for (;;) {
        ......
        if (mNeedToScanDevices) {
            mNeedToScanDevices = false;
            scanDevicesLocked();
            ......
        }
        ......

        // Grab the next input event.
        bool deviceChanged = false;
        while (mPendingEventIndex < mPendingEventCount) {
            const struct epoll_event& eventItem = mPendingEventItems[mPendingEventIndex++];
            ......
            /*Read input event and save into buffer*/
        }
        ......

        // Return now if we have collected any events or if we were explicitly awoken.
        if (event != buffer || awoken) {
            break;
        }

        ......
        int pollResult = epoll_wait(mEpollFd, mPendingEventItems, EPOLL_MAX_EVENTS, timeoutMillis);
        ......
    }
    // All done, return the number of events we read.
    return event - buffer;
}
```

输入管理器的启动过程(续)

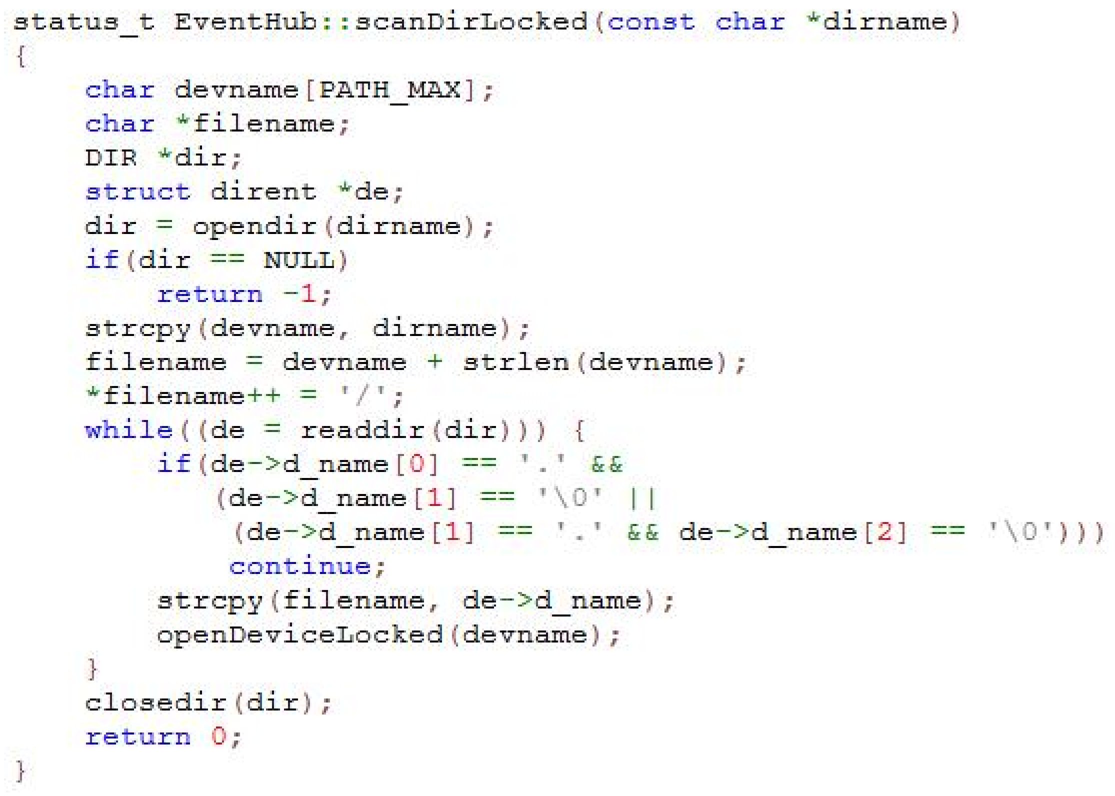
- 启动InputReader
  - EventHub.scanDevicesLocked



```
static const char *DEVICE_PATH = "/dev/input";

void EventHub::scanDevicesLocked() {
    status_t res = scanDirLocked(DEVICE_PATH);
    if(res < 0) {
        ALOGE("scan dir failed for %s\n", DEVICE_PATH);
    }
    if (mDevices.indexOfKey(VIRTUAL_KEYBOARD_ID) < 0) {
        createVirtualKeyboardLocked();
    }
}
```

输入管理器的启动过程(续)

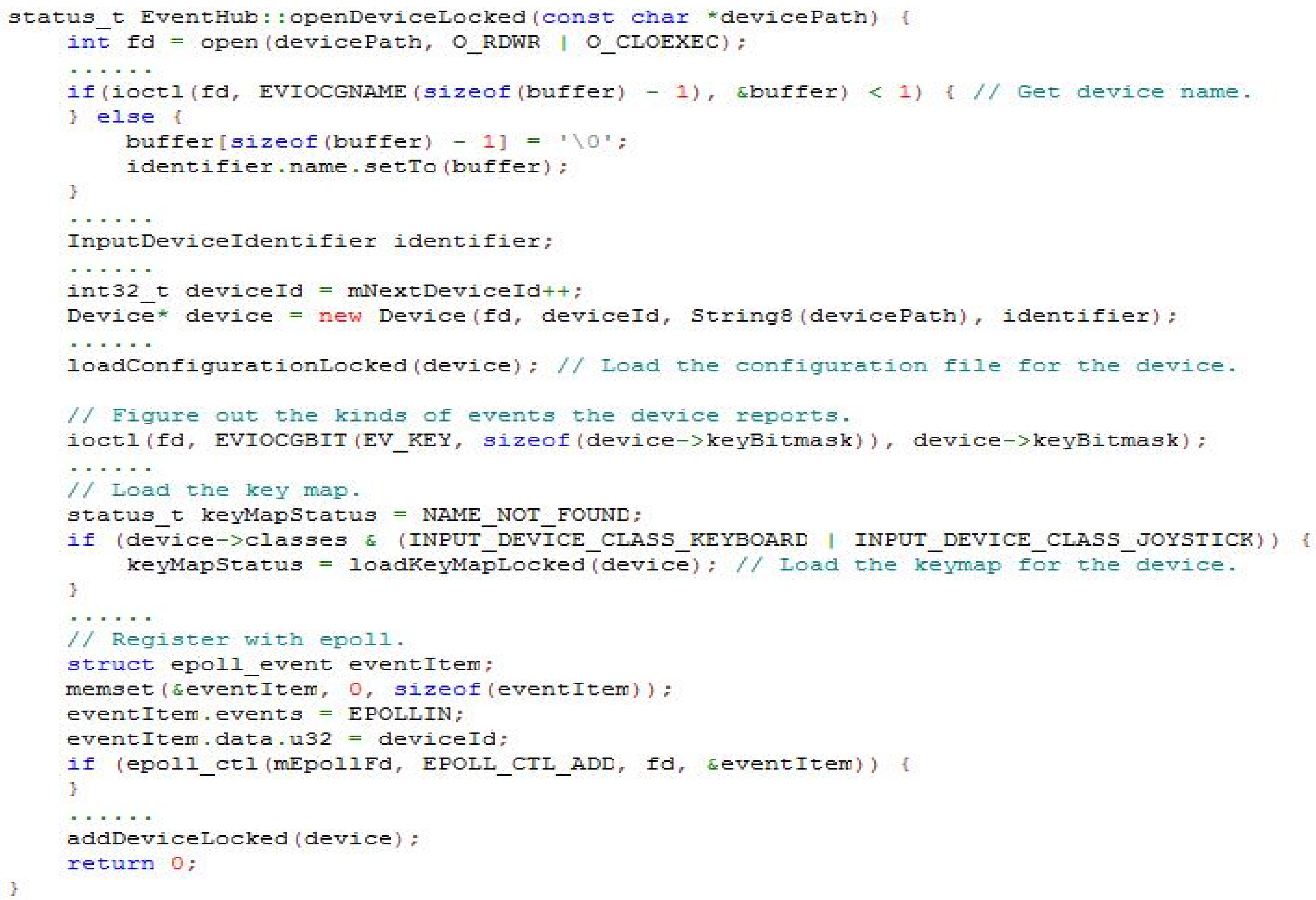
- 启动InputReader
  - EventHub.scanDirLocked



```
status_t EventHub::scanDirLocked(const char *dirname)
{
    char devname[PATH_MAX];
    char *filename;
    DIR *dir;
    struct dirent *de;
    dir = opendir(dirname);
    if(dir == NULL)
        return -1;
    strcpy(devname, dirname);
    filename = devname + strlen(devname);
    *filename++ = '/';
    while((de = readdir(dir))) {
        if(de->d_name[0] == '.' &&
           (de->d_name[1] == '\0' ||
            (de->d_name[1] == '.' && de->d_name[2] == '\0')))
            continue;
        strcpy(filename, de->d_name);
        openDeviceLocked(devname);
    }
    closedir(dir);
    return 0;
}
```

输入管理器的启动过程(续)

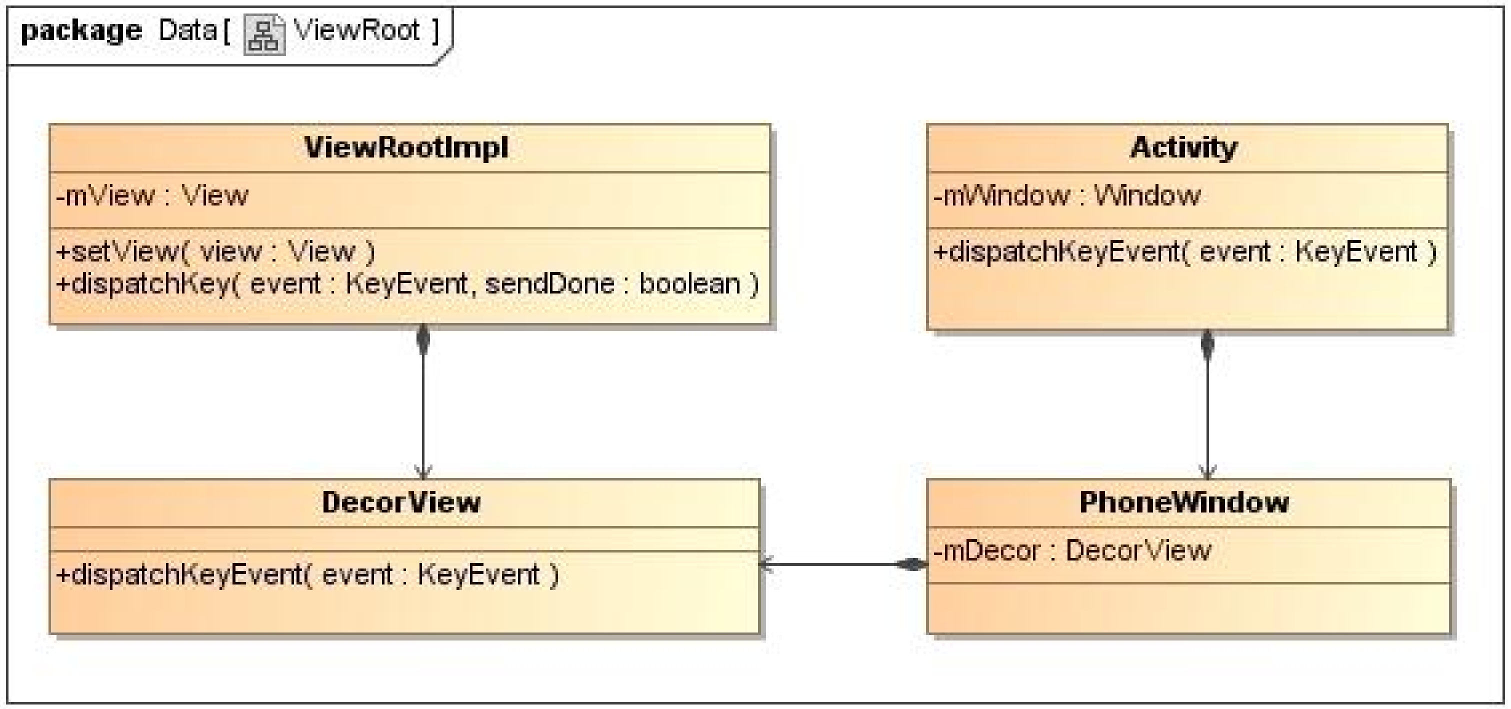
- 启动InputReader
  - EventHub.openDeviceLocked



```
status_t EventHub::openDeviceLocked(const char *devicePath) {
    int fd = open(devicePath, O_RDWR | O_CLOEXEC);
    ......
    if(ioctl(fd, EVIOCGNAME(sizeof(buffer) - 1), &buffer) < 1) { // Get device name.
    } else {
        buffer[sizeof(buffer) - 1] = '\0';
        identifier.name.setTo(buffer);
    }
    ......
    InputDeviceIdentifier identifier;
    ......
    int32_t deviceId = mNextDeviceId++;
    Device* device = new Device(fd, deviceId, String8(devicePath), identifier);
    ......
    loadConfigurationLocked(device); // Load the configuration file for the device.

    // Figure out the kinds of events the device reports.
    ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(device->keyBitmask)), device->keyBitmask);
    ......
    // Load the key map.
    status_t keyMapStatus = NAME_NOT_FOUND;
    if (device->classes & (INPUT_DEVICE_CLASS_KEYBOARD | INPUT_DEVICE_CLASS_JOYSTICK)) {
        keyMapStatus = loadKeyMapLocked(device); // Load the keymap for the device.
    }
    ......
    // Register with epoll.
    struct epoll_event eventItem;
    memset(&eventItem, 0, sizeof(eventItem));
    eventItem.events = EPOLLIN;
    eventItem.data.u32 = deviceId;
    if (epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, &eventItem)) {
    }
    ......
    addDeviceLocked(device);
    return 0;
}
```

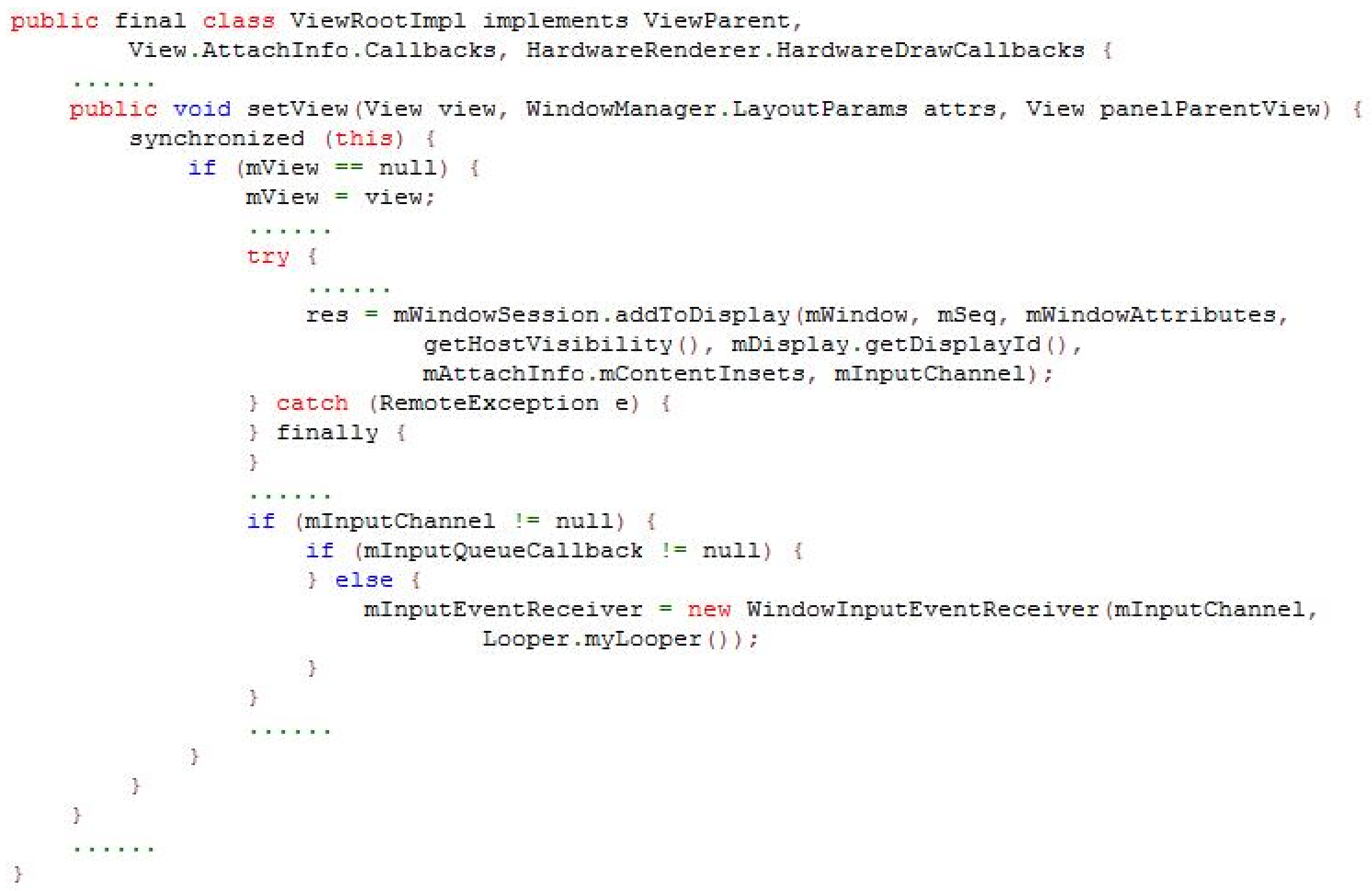
输入通道的注册过程

- Activity窗口的组成


package Data [ViewRoot]
ViewRootImpl
-mView : View
+setView(view : View)
+dispatchKey(event : KeyEvent, sendDone : boolean)
Activity
-mWindow : Window
+dispatchKeyEvent(event : KeyEvent)
DecorView
+dispatchKeyEvent(event : KeyEvent)
PhoneWindow
-mDecor : DecorView

输入通道的注册过程

- 创建InputChannel – ViewRootImpl.setView



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......
    public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
        synchronized (this) {
            if (mView == null) {
                mView = view;
                ......
                try {
                    ......
                    res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                            getHostVisibility(), mDisplay.getDisplayId(),
                            mAttachInfo.mContentInsets, mInputChannel);
                } catch (RemoteException e) {
                } finally {
                }
                ......
                if (mInputChannel != null) {
                    if (mInputQueueCallback != null) {
                    } else {
                        mInputEventReceiver = new WindowInputEventReceiver(mInputChannel,
                                Looper.myLooper());
                    }
                }
                ......
            }
        }
    }
    ......
}
```

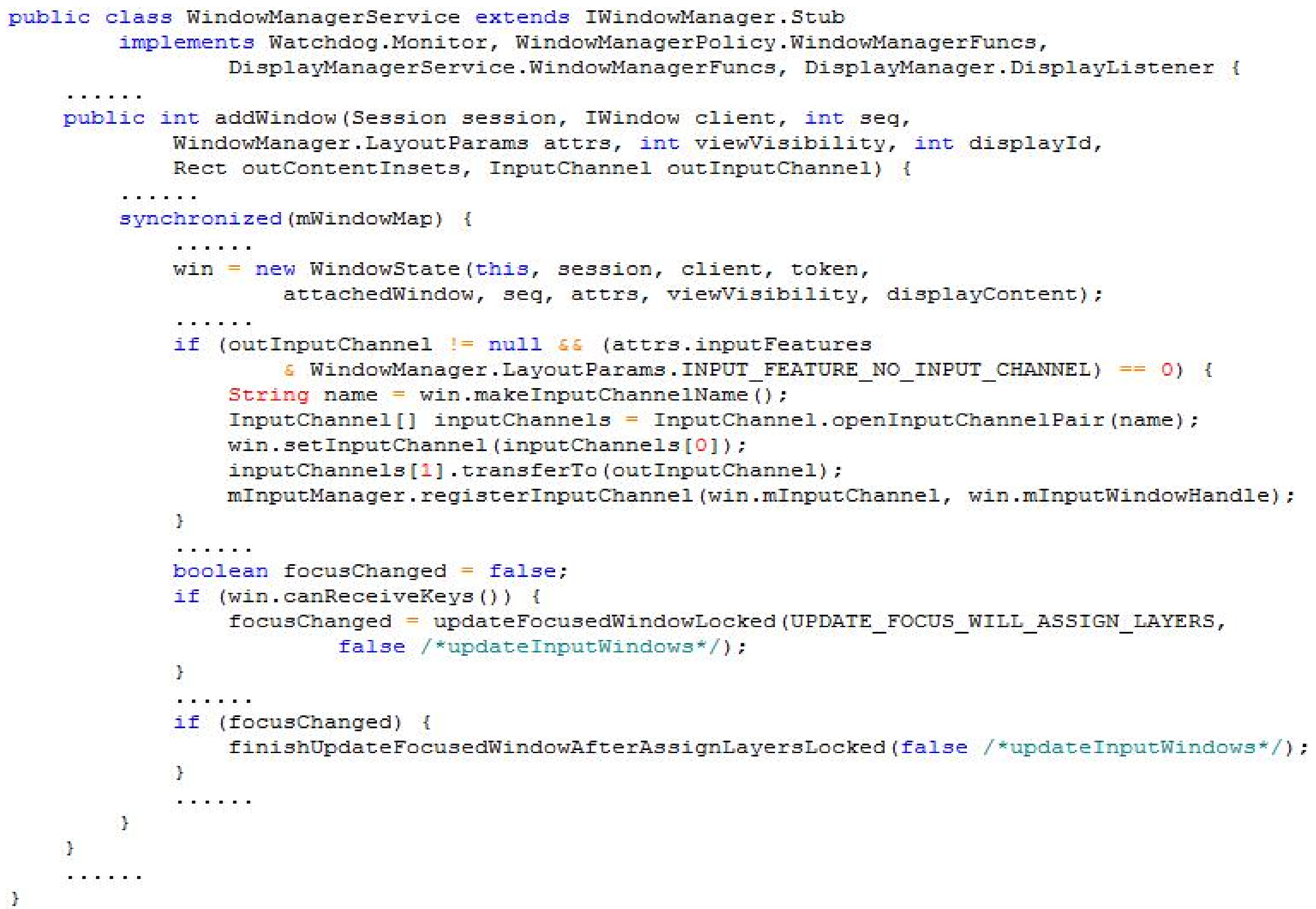
输入通道的注册过程(续)

- 创建InputChannel – Session.addToDisplay

```
final class Session extends IWindowSession.Stub
        implements IBinder.DeathRecipient {
    ......
    @Override
    public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
            int viewVisibility, int displayId, Rect outContentInsets,
            InputChannel outInputChannel) {
        return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
                outContentInsets, outInputChannel);
    }
    ......
}
```

输入通道的注册过程(续)

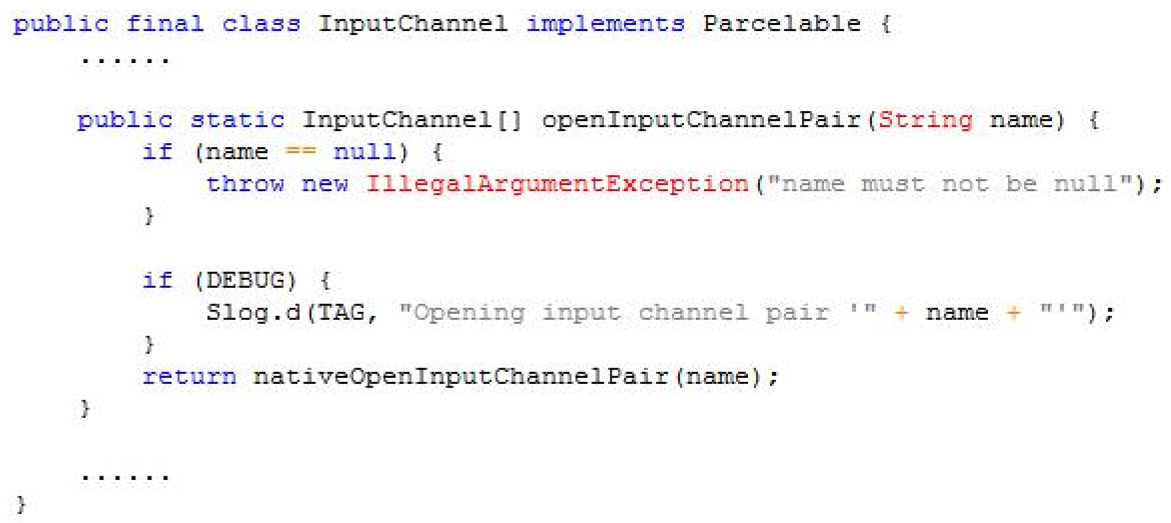
- 创建InputChannel – WMS.addWindow



```
public class WindowManagerService extends IWindowManager.Stub
        implements Watchdog.Monitor, WindowManagerPolicy.WindowManagerFuncs,
                DisplayManagerService.WindowManagerFuncs, DisplayManager.DisplayListener {
    ......
    public int addWindow(Session session, IWindow client, int seq,
            WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
            Rect outContentInsets, InputChannel outInputChannel) {
        ......
        synchronized(mWindowMap) {
            ......
            win = new WindowState(this, session, client, token,
                    attachedWindow, seq, attrs, viewVisibility, displayContent);
            ......
            if (outInputChannel != null && (attrs.inputFeatures
                    & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
                String name = win.makeInputChannelName();
                InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
                win.setInputChannel(inputChannels[0]);
                inputChannels[1].transferTo(outInputChannel);
                mInputManager.registerInputChannel(win.mInputChannel, win.mInputWindowHandle);
            }
            ......
            boolean focusChanged = false;
            if (win.canReceiveKeys()) {
                focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,
                        false /*updateInputWindows*/);
            }
            ......
            if (focusChanged) {
                finishUpdateFocusedWindowAfterAssignLayersLocked(false /*updateInputWindows*/);
            }
            ......
        }
    }
    ......
}
```

输入通道的注册过程(续)

- 创建InputChannel – InputChannel.openInputChannelPair



```java
public final class InputChannel implements Parcelable {
    ......

    public static InputChannel[] openInputChannelPair(String name) {
        if (name == null) {
            throw new IllegalArgumentException("name must not be null");
        }

        if (DEBUG) {
            Slog.d(TAG, "Opening input channel pair '" + name + "'");
        }
        return nativeOpenInputChannelPair(name);
    }

    ......
}
```

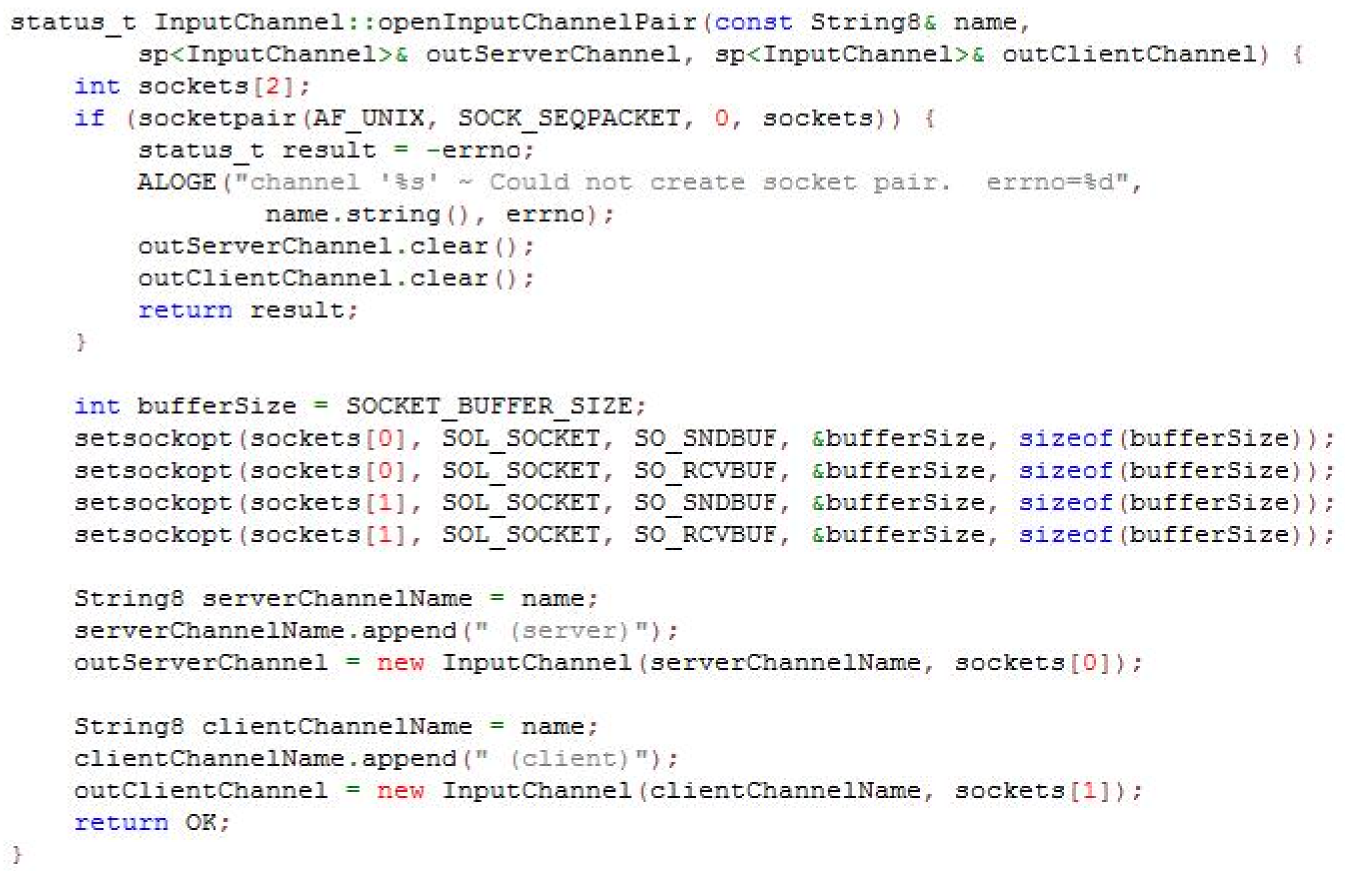
输入通道的注册过程(续)

- 创建InputChannel – nativeOpenInputChannelPair

```
static jobjectArray android_view_InputChannel_nativeOpenInputChannelPair(JNIEnv* env,
        jclass clazz, jstring nameObj) {
    ......
    sp<InputChannel> serverChannel;
    sp<InputChannel> clientChannel;
    status_t result = InputChannel::openInputChannelPair(name, serverChannel, clientChannel);
    ......
    jobjectArray channelPair = env->NewObjectArray(2, gInputChannelClassInfo.clazz, NULL);
    ......
    jobject serverChannelObj = android_view_InputChannel_createInputChannel(env,
            new NativeInputChannel(serverChannel));
    ......
    jobject clientChannelObj = android_view_InputChannel_createInputChannel(env,
            new NativeInputChannel(clientChannel));
    ......
    env->SetObjectArrayElement(channelPair, 0, serverChannelObj);
    env->SetObjectArrayElement(channelPair, 1, clientChannelObj);
    return channelPair;
}
```

输入通道的注册过程(续)

- 创建InputChannel – InputChannel::openInputChannelPair



```
status_t InputChannel::openInputChannelPair(const String8& name,
        sp<InputChannel>& outServerChannel, sp<InputChannel>& outClientChannel) {
    int sockets[2];
    if (socketpair(AF_UNIX, SOCK_SEQPACKET, 0, sockets)) {
        status_t result = -errno;
        ALOGE("channel '%s' ~ Could not create socket pair.  errno=%d",
                name.string(), errno);
        outServerChannel.clear();
        outClientChannel.clear();
        return result;
    }

    int bufferSize = SOCKET_BUFFER_SIZE;
    setsockopt(sockets[0], SOL_SOCKET, SO_SNDBUF, &bufferSize, sizeof(bufferSize));
    setsockopt(sockets[0], SOL_SOCKET, SO_RCVBUF, &bufferSize, sizeof(bufferSize));
    setsockopt(sockets[1], SOL_SOCKET, SO_SNDBUF, &bufferSize, sizeof(bufferSize));
    setsockopt(sockets[1], SOL_SOCKET, SO_RCVBUF, &bufferSize, sizeof(bufferSize));

    String8 serverChannelName = name;
    serverChannelName.append(" (server)");
    outServerChannel = new InputChannel(serverChannelName, sockets[0]);

    String8 clientChannelName = name;
    clientChannelName.append(" (client)");
    outClientChannel = new InputChannel(clientChannelName, sockets[1]);
    return OK;
}
```

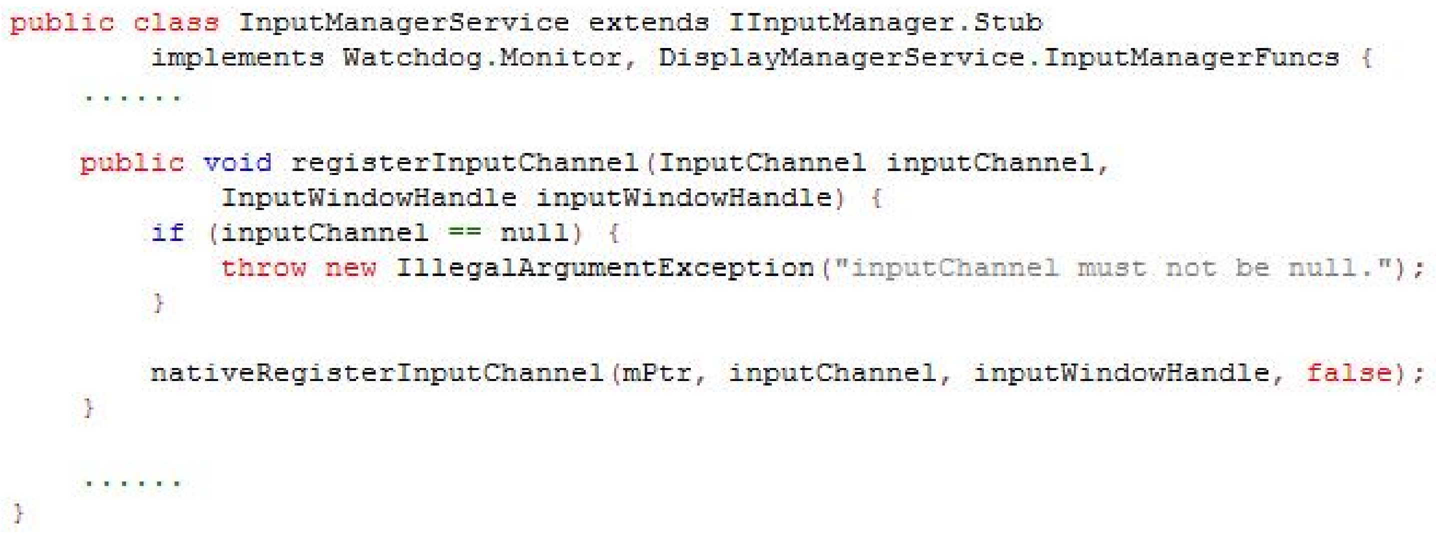
输入通道的注册过程(续)

- 注册Server端InputChannel

```
public class WindowManagerService extends IWindowManager.Stub
        implements Watchdog.Monitor, WindowManagerPolicy.WindowManagerFuncs,
                DisplayManagerService.WindowManagerFuncs, DisplayManager.DisplayListener {
    ......
    public int addWindow(Session session, IWindow client, int seq,
            WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
            Rect outContentInsets, InputChannel outInputChannel) {
        ......
        synchronized(mWindowMap) {
            ......
            win = new WindowState(this, session, client, token,
                    attachedWindow, seq, attrs, viewVisibility, displayContent);
            ......
            if (outInputChannel != null && (attrs.inputFeatures
                    & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
                String name = win.makeInputChannelName();
                InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
                win.setInputChannel(inputChannels[0]);
                inputChannels[1].transferTo(outInputChannel);
                mInputManager.registerInputChannel(win.mInputChannel, win.mInputWindowHandle);
            }
            ......
            boolean focusChanged = false;
            if (win.canReceiveKeys()) {
                focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,
                        false /*updateInputWindows*/);
            }
            ......
            if (focusChanged) {
                finishUpdateFocusedWindowAfterAssignLayersLocked(false /*updateInputWindows*/);
            }
            ......
        }
    }
    ......
}
```

输入通道的注册过程(续)

- 注册Server端InputChannel—IMS.registerInputChannel



```
public class InputManagerService extends IInputManager.Stub
        implements Watchdog.Monitor, DisplayManagerService.InputManagerFuncs {
    ......

    public void registerInputChannel(InputChannel inputChannel,
            InputWindowHandle inputWindowHandle) {
        if (inputChannel == null) {
            throw new IllegalArgumentException("inputChannel must not be null.");
        }

        nativeRegisterInputChannel(mPtr, inputChannel, inputWindowHandle, false);
    }

    ......
}
```

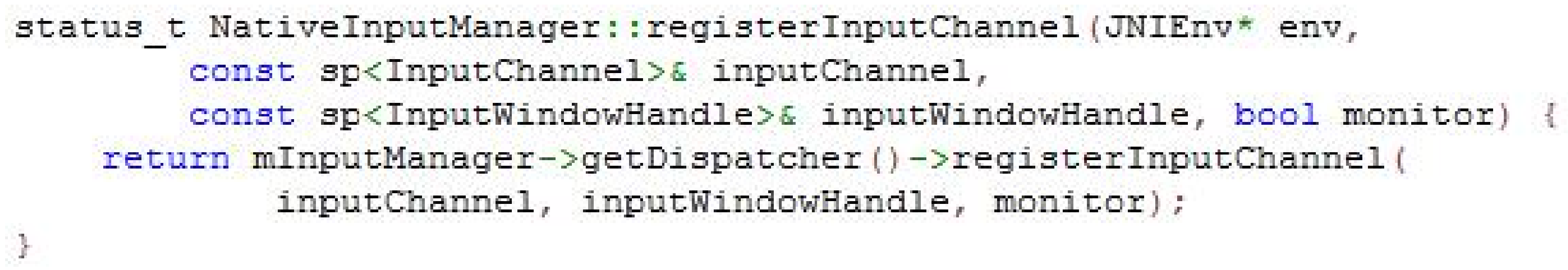
输入通道的注册过程(续)

- 注册Server端InputChannel—nativeRegisterInputChannel

```
static void nativeRegisterInputChannel(JNIEnv* env, jclass clazz,
        jint ptr, jobject inputChannelObj, jobject inputWindowHandleObj, jboolean monitor) {
    NativeInputManager* im = reinterpret_cast<NativeInputManager*>(ptr);

    sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
            inputChannelObj);
    ......

    sp<InputWindowHandle> inputWindowHandle =
            android_server_InputWindowHandle_getHandle(env, inputWindowHandleObj);

    status_t status = im->registerInputChannel(
            env, inputChannel, inputWindowHandle, monitor);
    ......
}
```

输入通道的注册过程(续)

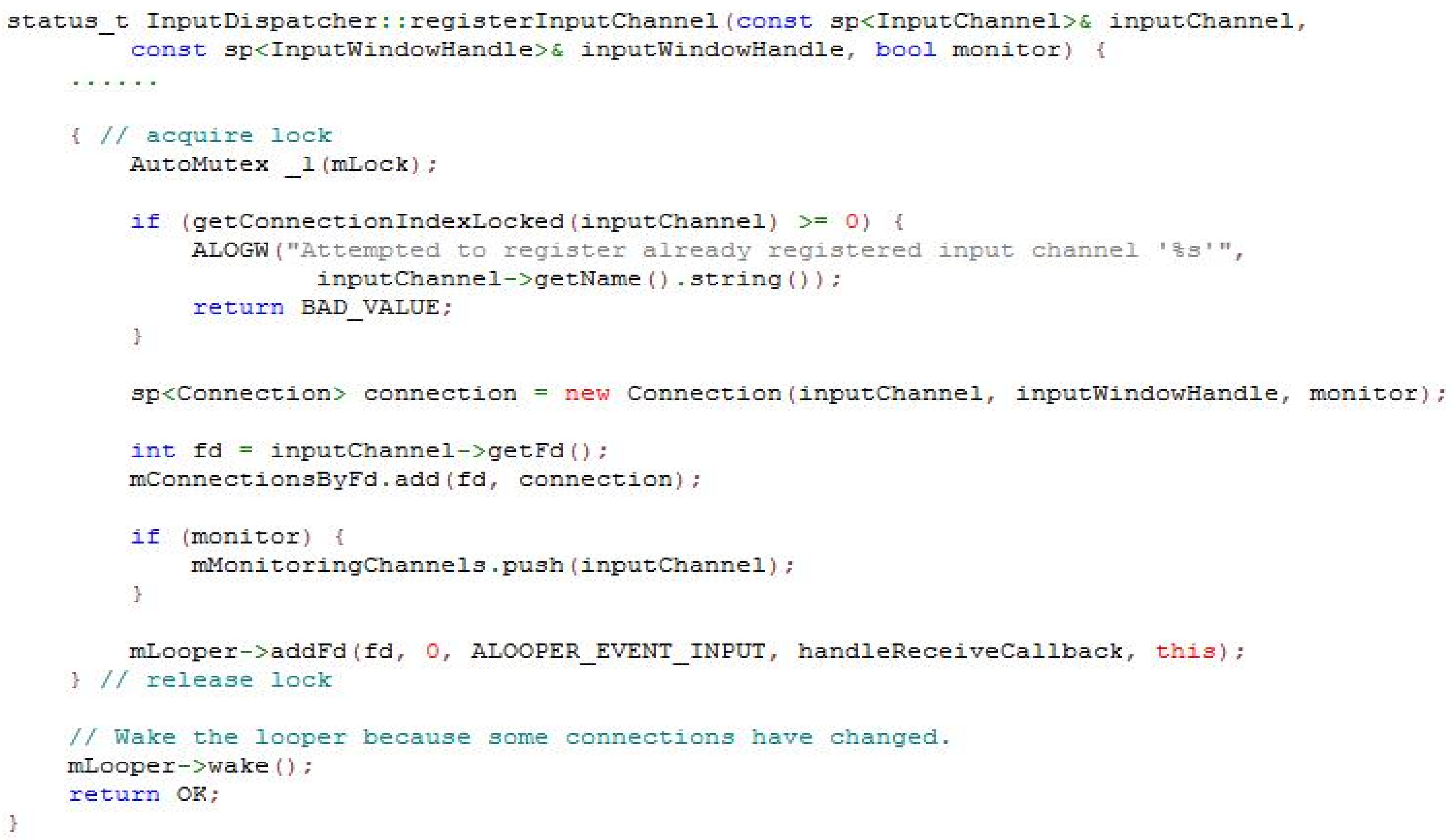
- 注册Server端InputChannel—NativeInputManager.registerInputChannel



```
status_t NativeInputManager::registerInputChannel(JNIEnv* env,
        const sp<InputChannel>& inputChannel,
        const sp<InputWindowHandle>& inputWindowHandle, bool monitor) {
    return mInputManager->getDispatcher()->registerInputChannel(
            inputChannel, inputWindowHandle, monitor);
}
```

输入通道的注册过程(续)

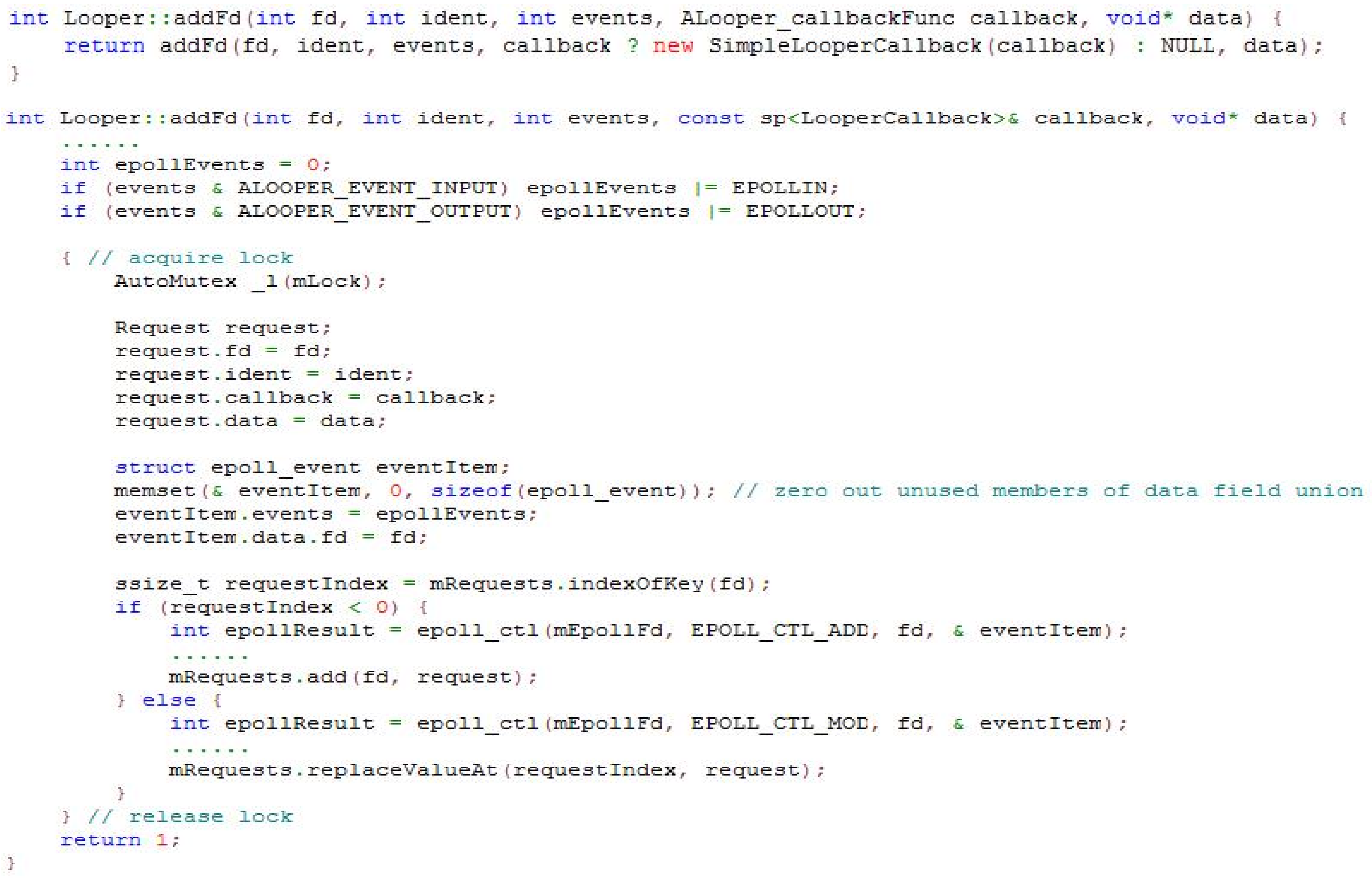
- 注册Server端InputChannel—InputDispatcher.registerInputChannel



```
status_t InputDispatcher::registerInputChannel(const sp<InputChannel>& inputChannel,
        const sp<InputWindowHandle>& inputWindowHandle, bool monitor) {
    ......

    { // acquire lock
        AutoMutex _l(mLock);

        if (getConnectionIndexLocked(inputChannel) >= 0) {
            ALOGW("Attempted to register already registered input channel '%s'",
                    inputChannel->getName().string());
            return BAD_VALUE;
        }

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

        int fd = inputChannel->getFd();
        mConnectionsByFd.add(fd, connection);

        if (monitor) {
            mMonitoringChannels.push(inputChannel);
        }

        mLooper->addFd(fd, 0, ALOOPER_EVENT_INPUT, handleReceiveCallback, this);
    } // release lock

    // Wake the looper because some connections have changed.
    mLooper->wake();
    return OK;
}
```

输入通道的注册过程(续)

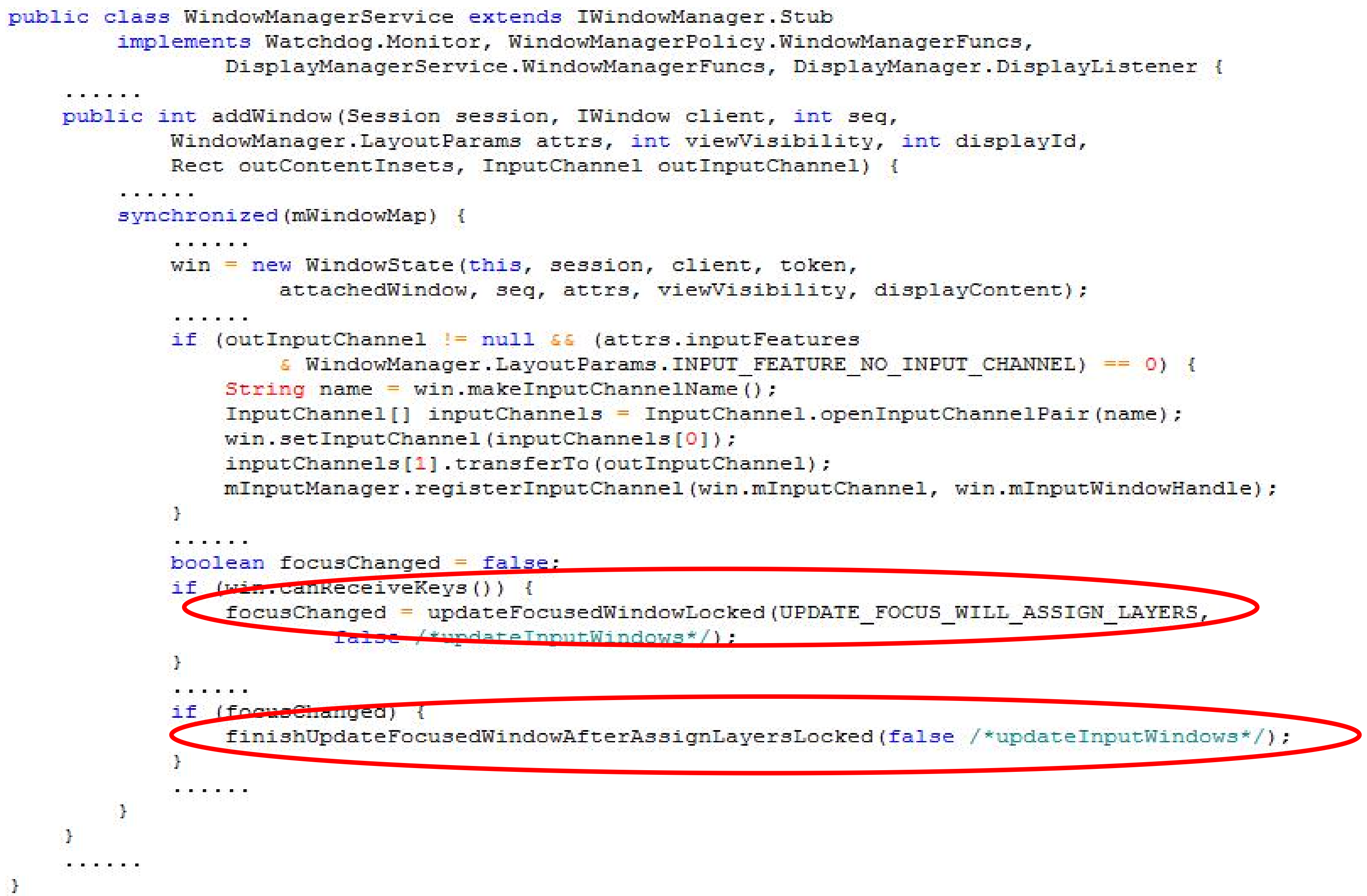
- 注册Server端InputChannel—Looper.addFd



```
int Looper::addFd(int fd, int ident, int events, ALooper_callbackFunc callback, void* data) {
    return addFd(fd, ident, events, callback ? new SimpleLooperCallback(callback) : NULL, data);
}

int Looper::addFd(int fd, int ident, int events, const sp<LooperCallback>& callback, void* data) {
    ......
    int epollEvents = 0;
    if (events & ALOOPER_EVENT_INPUT) epollEvents |= EPOLLIN;
    if (events & ALOOPER_EVENT_OUTPUT) epollEvents |= EPOLLOUT;

    { // acquire lock
        AutoMutex _l(mLock);

        Request request;
        request.fd = fd;
        request.ident = ident;
        request.callback = callback;
        request.data = data;

        struct epoll_event eventItem;
        memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data field union
        eventItem.events = epollEvents;
        eventItem.data.fd = fd;

        ssize_t requestIndex = mRequests.indexOfKey(fd);
        if (requestIndex < 0) {
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, & eventItem);
            ......
            mRequests.add(fd, request);
        } else {
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_MOD, fd, & eventItem);
            ......
            mRequests.replaceValueAt(requestIndex, request);
        }
    } // release lock
    return 1;
}
```

输入通道的注册过程(续)

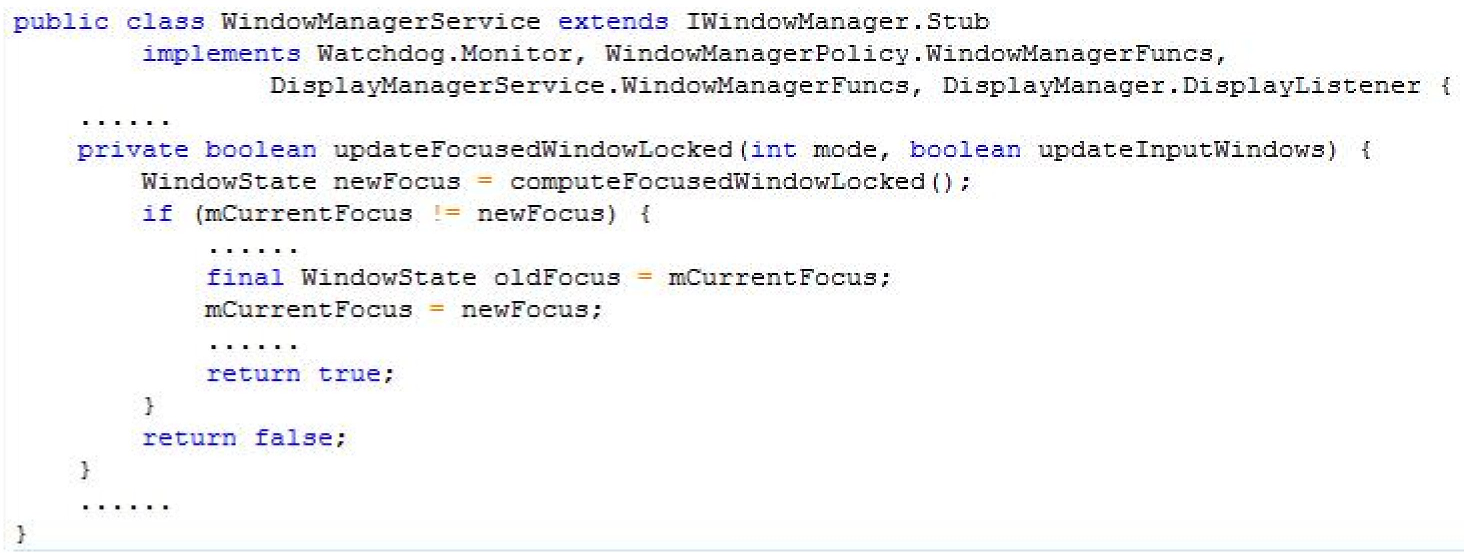
- 更新当前激活窗口



```
public class WindowManagerService extends IWindowManager.Stub
        implements Watchdog.Monitor, WindowManagerPolicy.WindowManagerFuncs,
                DisplayManagerService.WindowManagerFuncs, DisplayManager.DisplayListener {
    ......
    public int addWindow(Session session, IWindow client, int seq,
            WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
            Rect outContentInsets, InputChannel outInputChannel) {
        ......
        synchronized(mWindowMap) {
            ......
            win = new WindowState(this, session, client, token,
                    attachedWindow, seq, attrs, viewVisibility, displayContent);
            ......
            if (outInputChannel != null && (attrs.inputFeatures
                    & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) {
                String name = win.makeInputChannelName();
                InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
                win.setInputChannel(inputChannels[0]);
                inputChannels[1].transferTo(outInputChannel);
                mInputManager.registerInputChannel(win.mInputChannel, win.mInputWindowHandle);
            }
            ......
            boolean focusChanged = false;
            if (win.canReceiveKeys()) {
                focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS,
                        false /*updateInputWindows*/);
            }
            ......
            if (focusChanged) {
                finishUpdateFocusedWindowAfterAssignLayersLocked(false /*updateInputWindows*/);
            }
            ......
        }
    }
    ......
}
```

输入通道的注册过程(续)

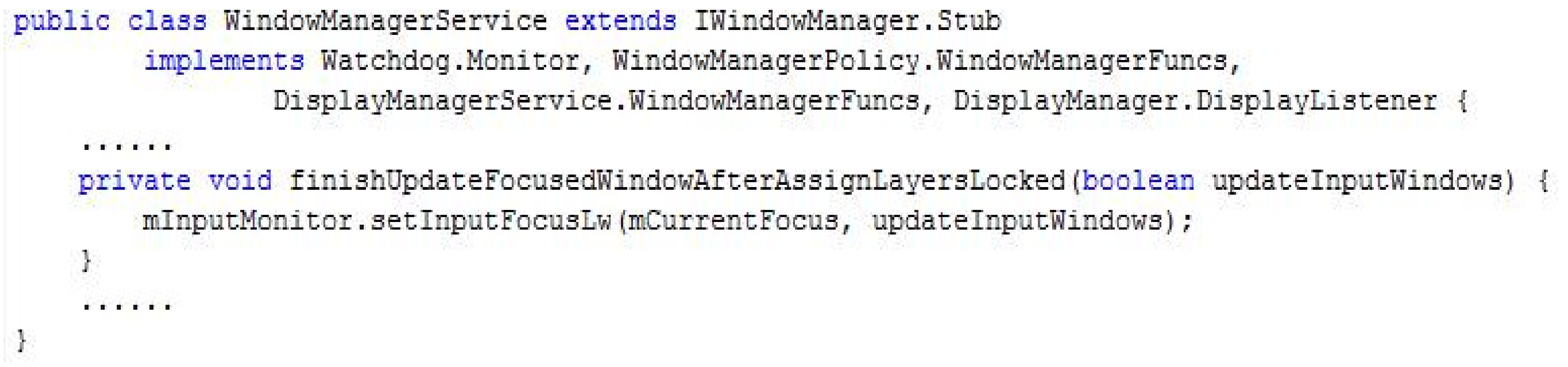
- 更新当前激活窗口— WMS.updateFoucsedWindowLocked



```
public class WindowManagerService extends IWindowManager.Stub
        implements Watchdog.Monitor, WindowManagerPolicy.WindowManagerFuncs,
                DisplayManagerService.WindowManagerFuncs, DisplayManager.DisplayListener {
    ......
    private boolean updateFocusedWindowLocked(int mode, boolean updateInputWindows) {
        WindowState newFocus = computeFocusedWindowLocked();
        if (mCurrentFocus != newFocus) {
            ......
            final WindowState oldFocus = mCurrentFocus;
            mCurrentFocus = newFocus;
            ......
            return true;
        }
        return false;
    }
    ......
}
```

输入通道的注册过程(续)

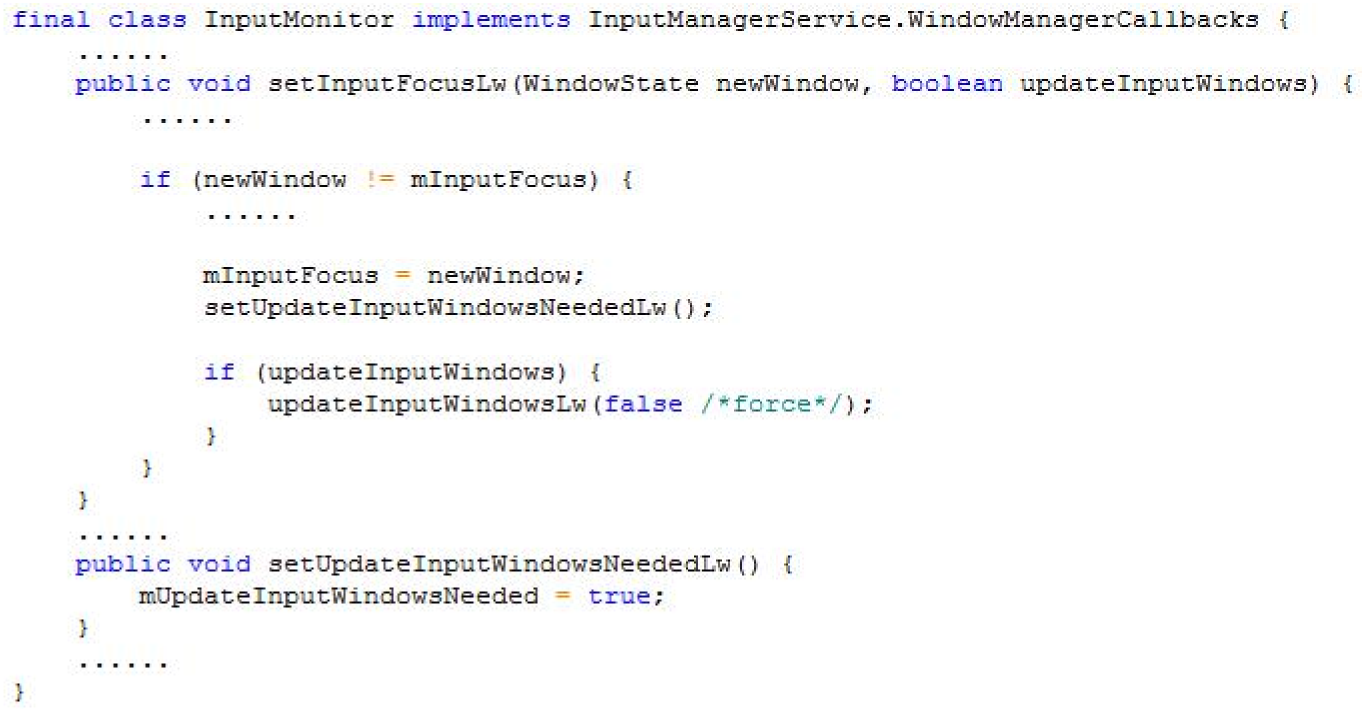
- 更新当前激活窗口—WMS. finishUpdateFocusedWindowAfterAssignLayersLocked



```
public class WindowManagerService extends IWindowManager.Stub
        implements Watchdog.Monitor, WindowManagerPolicy.WindowManagerFuncs,
                DisplayManagerService.WindowManagerFuncs, DisplayManager.DisplayListener {
    ......
    private void finishUpdateFocusedWindowAfterAssignLayersLocked(boolean updateInputWindows) {
        mInputMonitor.setInputFocusLw(mCurrentFocus, updateInputWindows);
    }
    ......
}
```

输入通道的注册过程(续)

- 更新当前激活窗口—InputMonitor.setInputFocusLw



```
final class InputMonitor implements InputManagerService.WindowManagerCallbacks {
    ......
    public void setInputFocusLw(WindowState newWindow, boolean updateInputWindows) {
        ......

        if (newWindow != mInputFocus) {
            ......

            mInputFocus = newWindow;
            setUpdateInputWindowsNeededLw();

            if (updateInputWindows) {
                updateInputWindowsLw(false /*force*/);
            }
        }
    }
    ......
    public void setUpdateInputWindowsNeededLw() {
        mUpdateInputWindowsNeeded = true;
    }
    ......
}
```

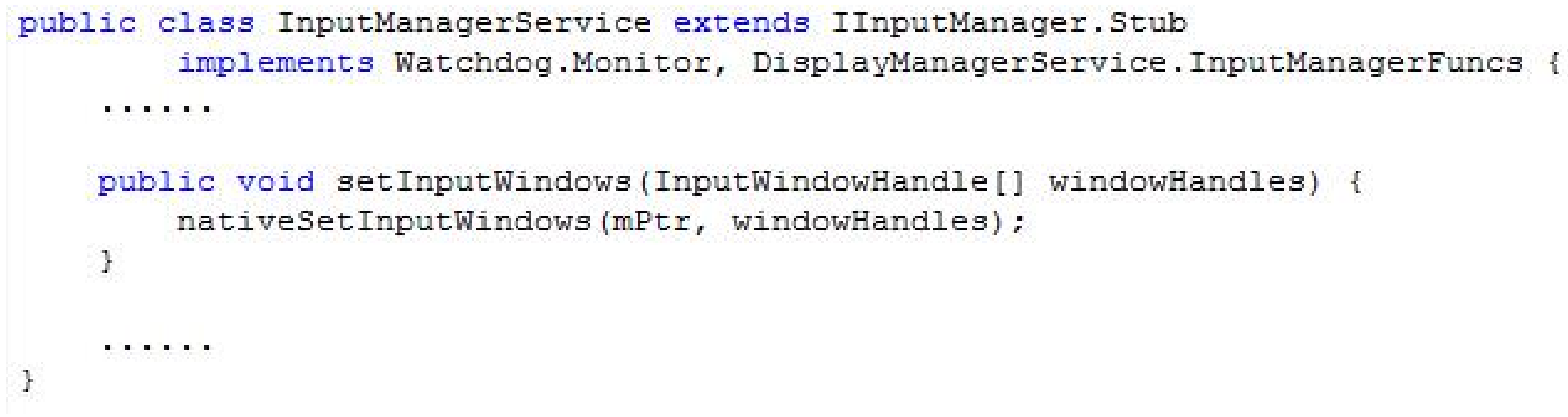
输入通道的注册过程(续)

- 更新当前激活窗口—InputMonitor. updateInputWindowsLw

```
final class InputMonitor implements InputManagerService.WindowManagerCallbacks {
    ......
    public void updateInputWindowsLw(boolean force) {
        if (!force && !mUpdateInputWindowsNeeded) {
            return;
        }
        mUpdateInputWindowsNeeded = false;
        ......
        // Add all windows on the default display.
        final AllWindowsIterator iterator = mService.new AllWindowsIterator(
                WindowManagerService.REVERSE_ITERATOR);
        while (iterator.hasNext()) {
            final WindowState child = iterator.next();
            final InputChannel inputChannel = child.mInputChannel;
            final InputWindowHandle inputWindowHandle = child.mInputWindowHandle;
            if (inputChannel == null || inputWindowHandle == null || child.mRemoved) {
                // Skip this window because it cannot possibly receive input.
                continue;
            }
            ......
            final boolean hasFocus = (child == mInputFocus);
            ......
            if (child.mWinAnimator != universeBackground) {
                addInputWindowHandleLw(inputWindowHandle, child, flags, type,
                        isVisible, hasFocus, hasWallpaper);
            }
        }

        // Send windows to native code.
        mService.mInputManager.setInputWindows(mInputWindowHandles);
        ......
    }
    ......
}
```

输入通道的注册过程(续)

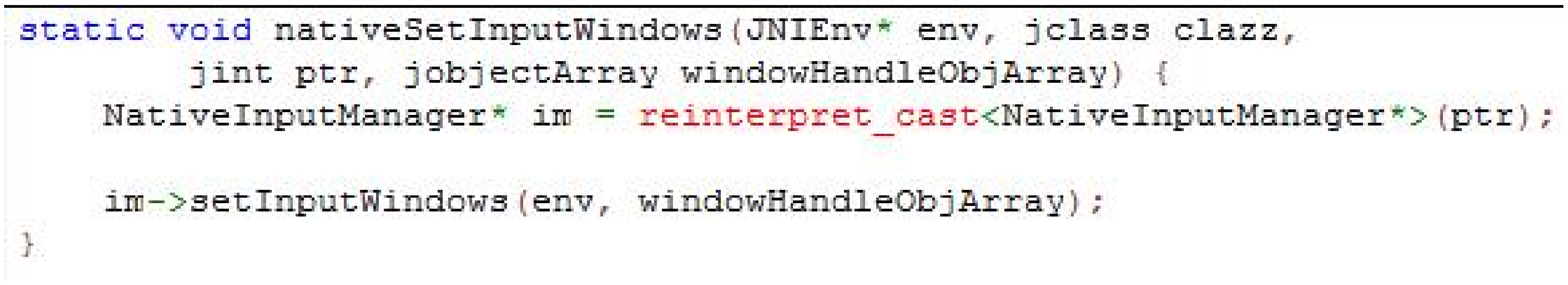
- 更新当前激活窗口—InputManagerService.setInputWindows



```
public class InputManagerService extends IInputManager.Stub
        implements Watchdog.Monitor, DisplayManagerService.InputManagerFuncs {
    ......

    public void setInputWindows(InputWindowHandle[] windowHandles) {
        nativeSetInputWindows(mPtr, windowHandles);
    }

    ......
}
```

输入通道的注册过程(续)

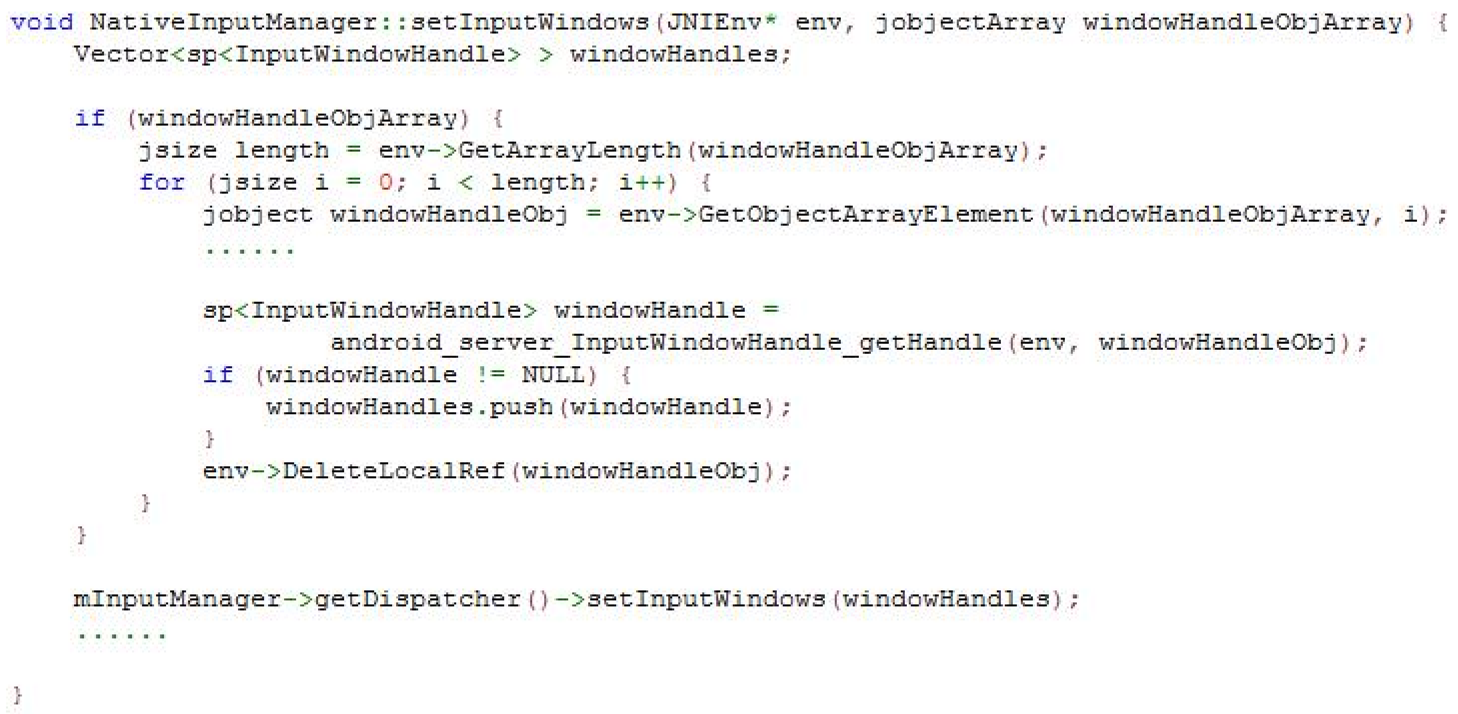
- 更新当前激活窗口--nativeSetInputWindows



```
static void nativeSetInputWindows(JNIEnv* env, jclass clazz,
        jint ptr, jobjectArray windowHandleObjArray) {
    NativeInputManager* im = reinterpret_cast<NativeInputManager*>(ptr);

    im->setInputWindows(env, windowHandleObjArray);
}
```

输入通道的注册过程(续)

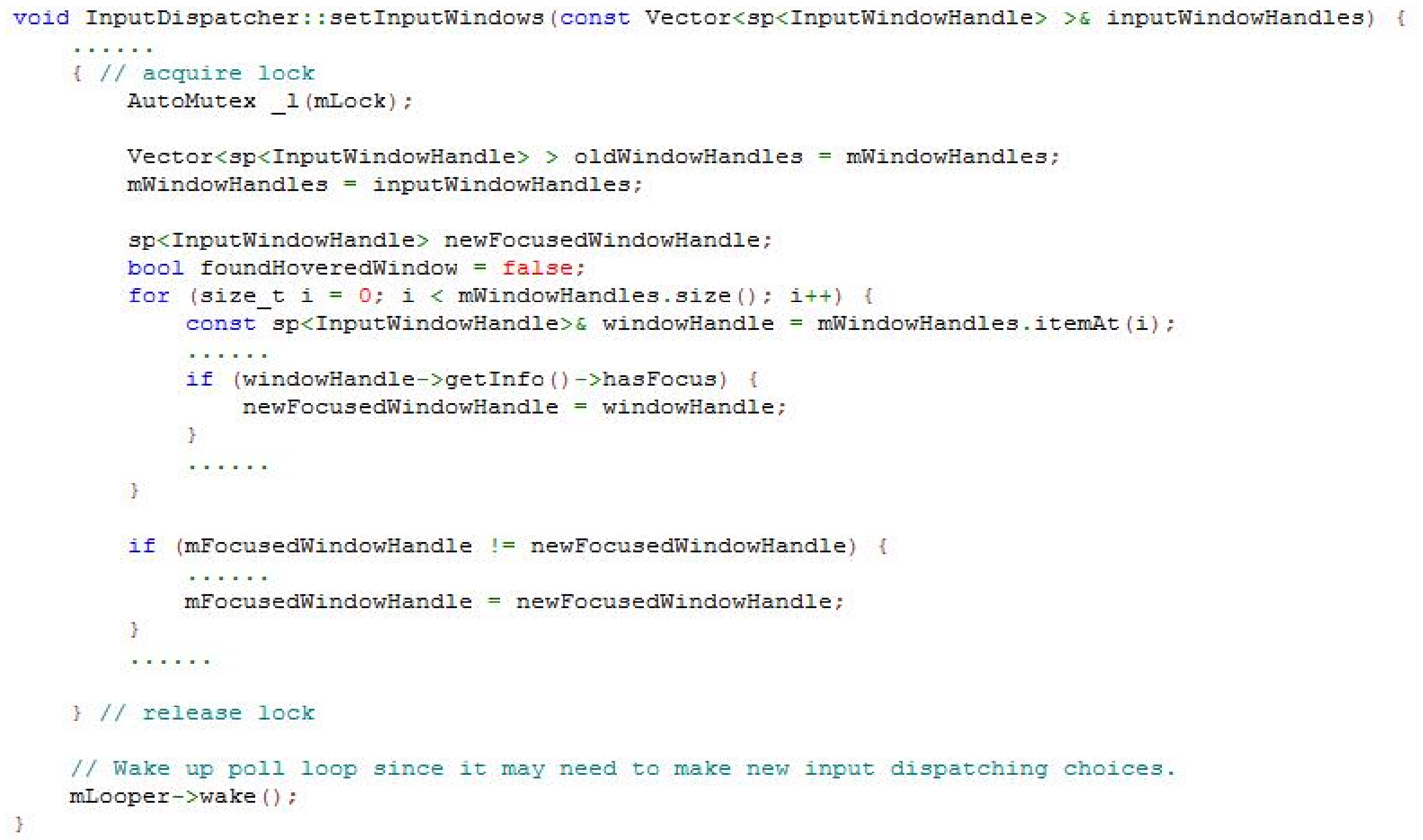
- 更新当前激活窗口—NativeInputManager.setInputWindows



```
void NativeInputManager::setInputWindows(JNIEnv* env, jobjectArray windowHandleObjArray) {
    Vector<sp<InputWindowHandle> > windowHandles;

    if (windowHandleObjArray) {
        jsize length = env->GetArrayLength(windowHandleObjArray);
        for (jsize i = 0; i < length; i++) {
            jobject windowHandleObj = env->GetObjectArrayElement(windowHandleObjArray, i);
            ......

            sp<InputWindowHandle> windowHandle =
                    android_server_InputWindowHandle_getHandle(env, windowHandleObj);
            if (windowHandle != NULL) {
                windowHandles.push(windowHandle);
            }
            env->DeleteLocalRef(windowHandleObj);
        }
    }

    mInputManager->getDispatcher()->setInputWindows(windowHandles);
    ......

}
```

输入通道的注册过程(续)

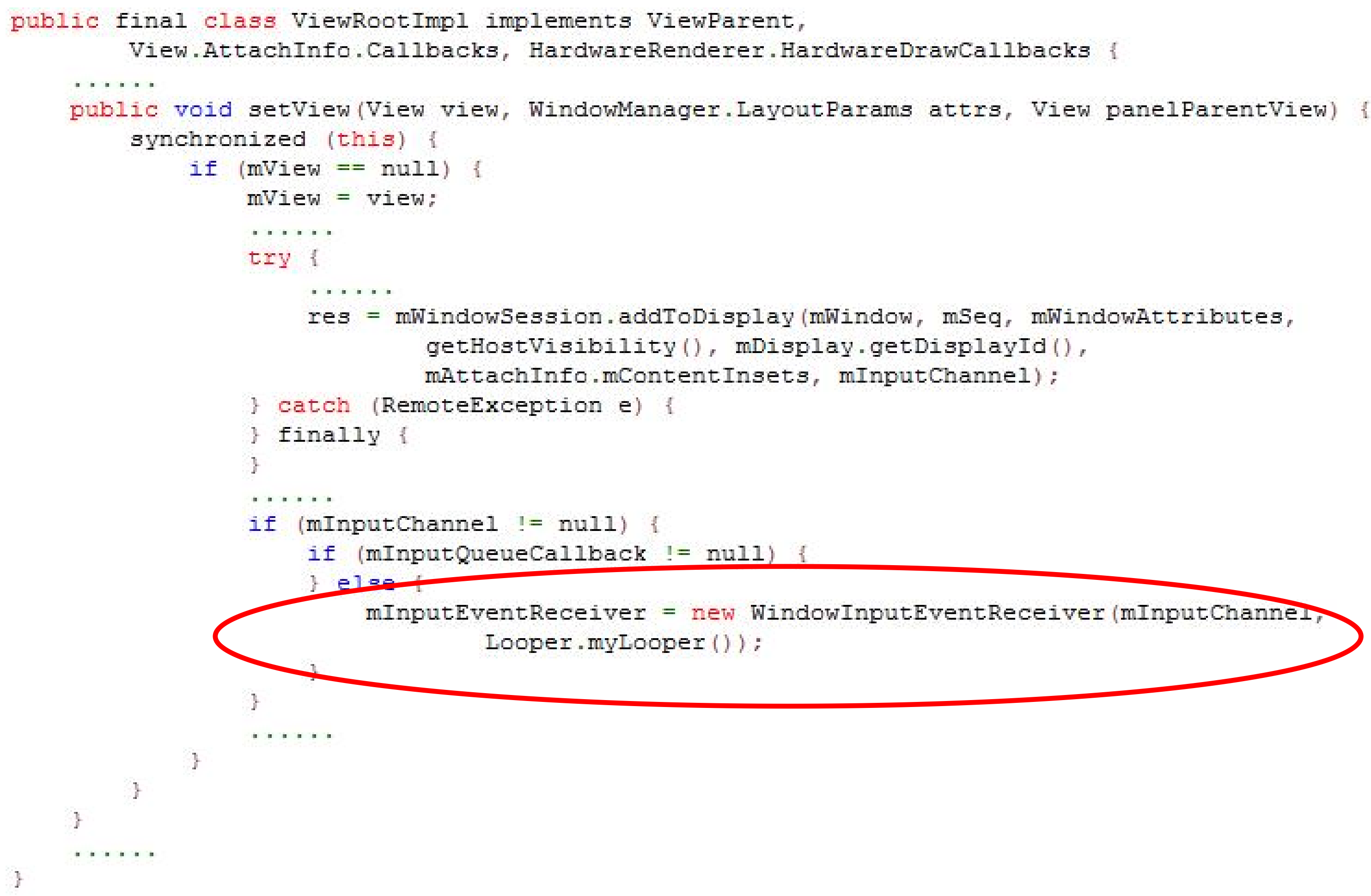
- 更新当前激活窗口 – InputDispatcher.setInputWindows



```
void InputDispatcher::setInputWindows(const Vector<sp<InputWindowHandle> >& inputWindowHandles) {
    ......
    { // acquire lock
        AutoMutex _l(mLock);

        Vector<sp<InputWindowHandle> > oldWindowHandles = mWindowHandles;
        mWindowHandles = inputWindowHandles;

        sp<InputWindowHandle> newFocusedWindowHandle;
        bool foundHoveredWindow = false;
        for (size_t i = 0; i < mWindowHandles.size(); i++) {
            const sp<InputWindowHandle>& windowHandle = mWindowHandles.itemAt(i);
            ......
            if (windowHandle->getInfo()->hasFocus) {
                newFocusedWindowHandle = windowHandle;
            }
            ......
        }

        if (mFocusedWindowHandle != newFocusedWindowHandle) {
            ......
            mFocusedWindowHandle = newFocusedWindowHandle;
        }
        ......

    } // release lock

    // Wake up poll loop since it may need to make new input dispatching choices.
    mLooper->wake();
}
```

输入通道的注册过程(续)

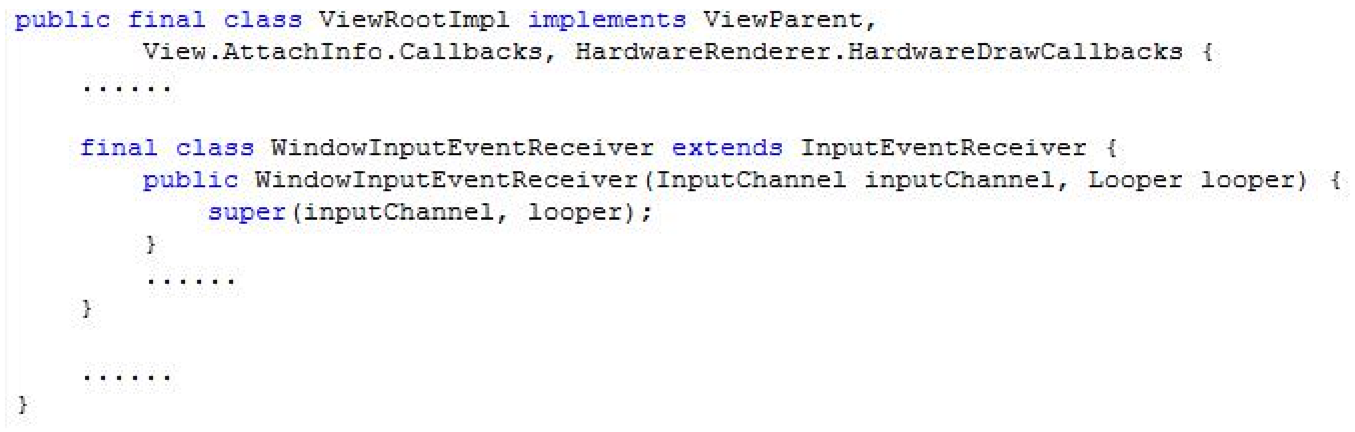
- 注册Client端InputChannel



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......
    public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
        synchronized (this) {
            if (mView == null) {
                mView = view;
                ......
                try {
                    ......
                    res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                            getHostVisibility(), mDisplay.getDisplayId(),
                            mAttachInfo.mContentInsets, mInputChannel);
                } catch (RemoteException e) {
                } finally {
                }
                ......
                if (mInputChannel != null) {
                    if (mInputQueueCallback != null) {
                    } else {
                        mInputEventReceiver = new WindowInputEventReceiver(mInputChannel,
                                Looper.myLooper());
                    }
                }
                ......
            }
        }
    }
    ......
}
```

输入通道的注册过程(续)

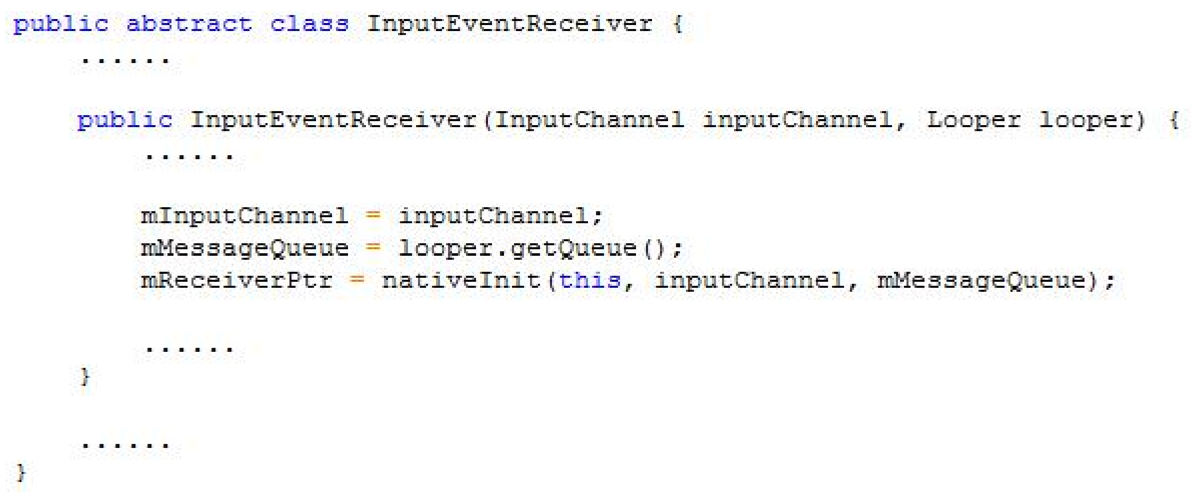
- 注册Client端InputChannel—new WindowInputEventReceiver



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    final class WindowInputEventReceiver extends InputEventReceiver {
        public WindowInputEventReceiver(InputChannel inputChannel, Looper looper) {
            super(inputChannel, looper);
        }
        ......
    }

    ......
}
```

输入通道的注册过程(续)

- 注册Client端InputChannel—new InputEventReceiver



```
public abstract class InputEventReceiver {
    ......

    public InputEventReceiver(InputChannel inputChannel, Looper looper) {
        ......

        mInputChannel = inputChannel;
        mMessageQueue = looper.getQueue();
        mReceiverPtr = nativeInit(this, inputChannel, mMessageQueue);

        ......
    }

    ......
}
```

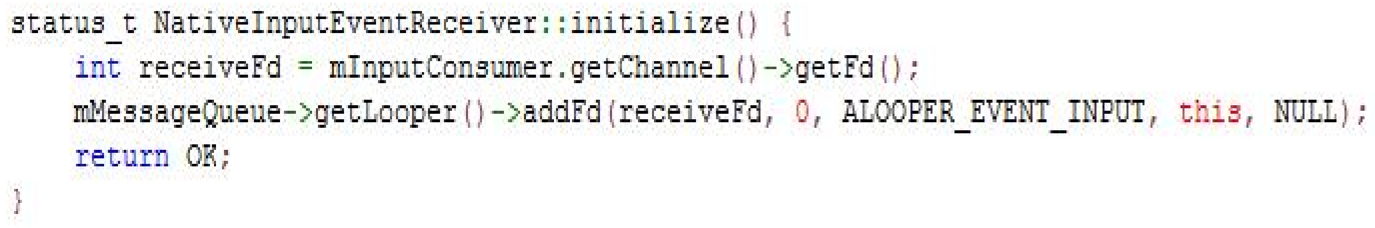
输入通道的注册过程(续)

- 注册Client端InputChannel -- nativeInit

```
static jint nativeInit(JNIEnv* env, jclass clazz, jobject receiverObj,
        jobject inputChannelObj, jobject messageQueueObj) {
    sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
            inputChannelObj);
    ......

    sp<MessageQueue> messageQueue = android_os_MessageQueue_getMessageQueue(env, messageQueueObj);
    ......

    sp<NativeInputEventReceiver> receiver = new NativeInputEventReceiver(env,
            receiverObj, inputChannel, messageQueue);
    status_t status = receiver->initialize();
    ......

    return reinterpret_cast<jint>(receiver.get());
}
```

输入通道的注册过程(续)

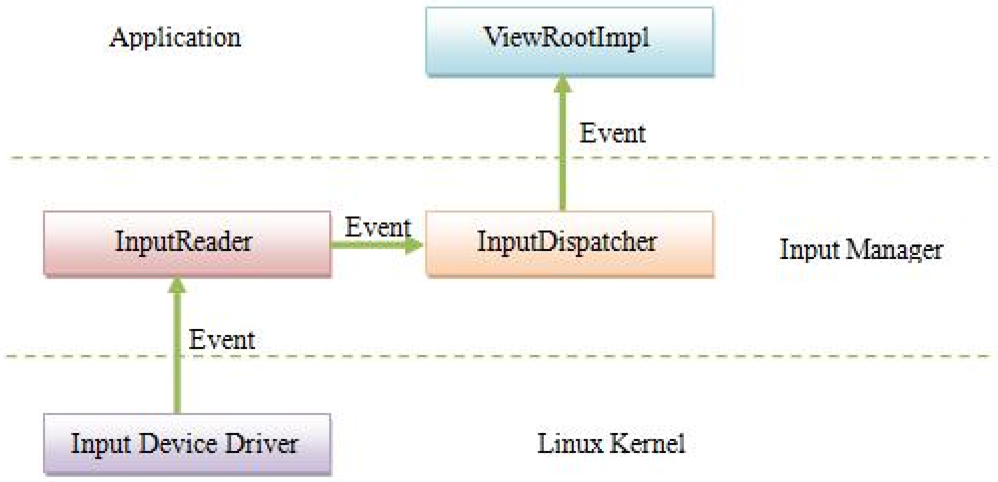
- 注册Client端InputChannel – NativeInputEventReceiver.initialize



```
status_t NativeInputEventReceiver::initialize() {
    int receiveFd = mInputConsumer.getChannel()->getFd();
    mMessageQueue->getLooper()->addFd(receiveFd, 0, ALOOPER_EVENT_INPUT, this, NULL);
    return OK;
}
```

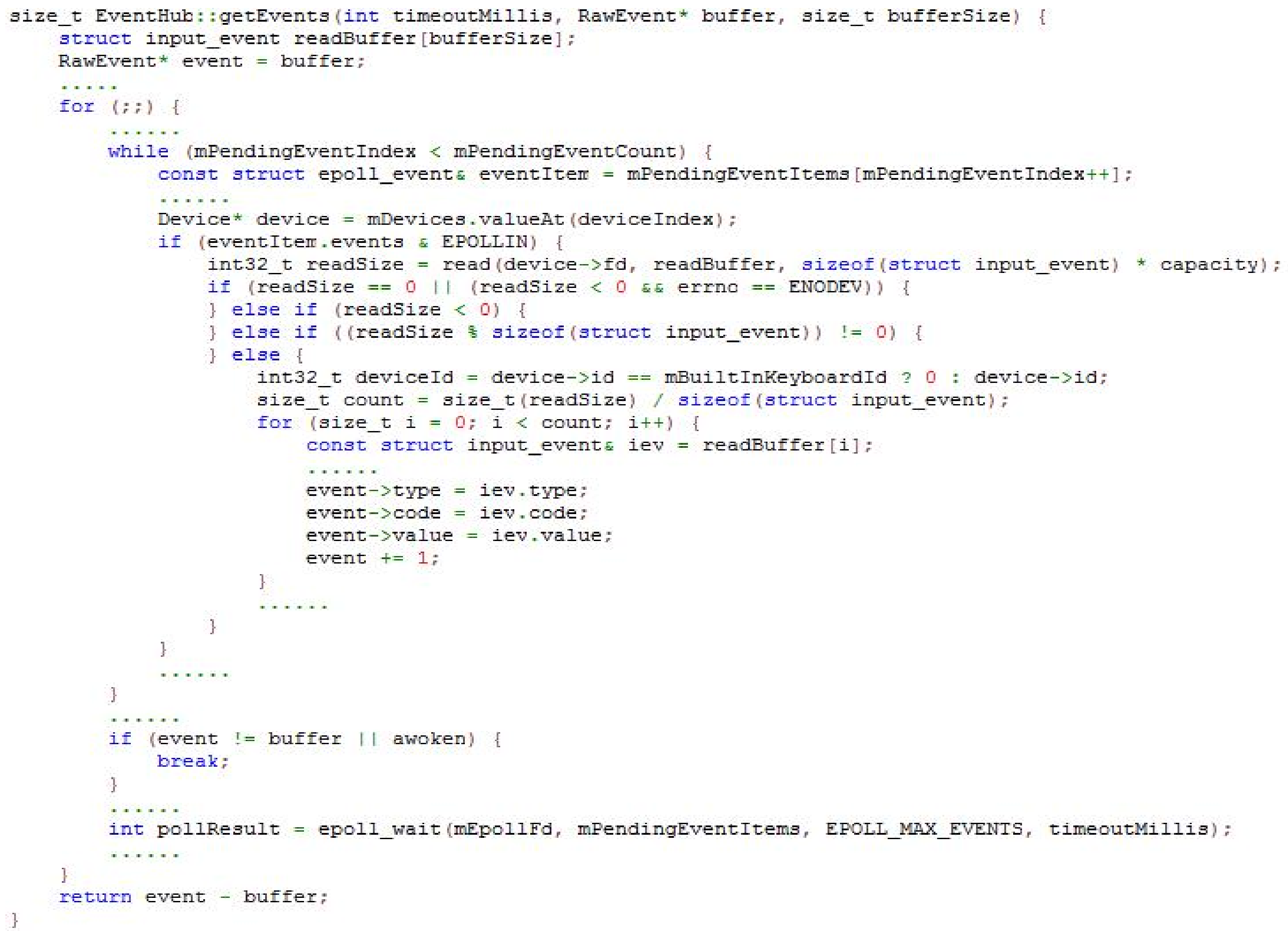
输入事件的分发过程

- 输入事件处理框架



输入事件的分发过程

- InputReader获得输入事件--EventHub.getEvents



```
size_t EventHub::getEvents(int timeoutMillis, RawEvent* buffer, size_t bufferSize) {
    struct input_event readBuffer[bufferSize];
    RawEvent* event = buffer;
    ......
    for (;;) {
        ......
        while (mPendingEventIndex < mPendingEventCount) {
            const struct epoll_event& eventItem = mPendingEventItems[mPendingEventIndex++];
            ......
            Device* device = mDevices.valueAt(deviceIndex);
            if (eventItem.events & EPOLLIN) {
                int32_t readSize = read(device->fd, readBuffer, sizeof(struct input_event) * capacity);
                if (readSize == 0 || (readSize < 0 && errno == ENODEV)) {
                } else if (readSize < 0) {
                } else if ((readSize % sizeof(struct input_event)) != 0) {
                } else {
                    int32_t deviceId = device->id == mBuiltInKeyboardId ? 0 : device->id;
                    size_t count = size_t(readSize) / sizeof(struct input_event);
                    for (size_t i = 0; i < count; i++) {
                        const struct input_event& iev = readBuffer[i];
                        ......
                        event->type = iev.type;
                        event->code = iev.code;
                        event->value = iev.value;
                        event += 1;
                    }
                    ......
                }
            }
            ......
        }
        ......
        if (event != buffer || awoken) {
            break;
        }
        ......
        int pollResult = epoll_wait(mEpollFd, mPendingEventItems, EPOLL_MAX_EVENTS, timeoutMillis);
        ......
    }
    return event - buffer;
}
```

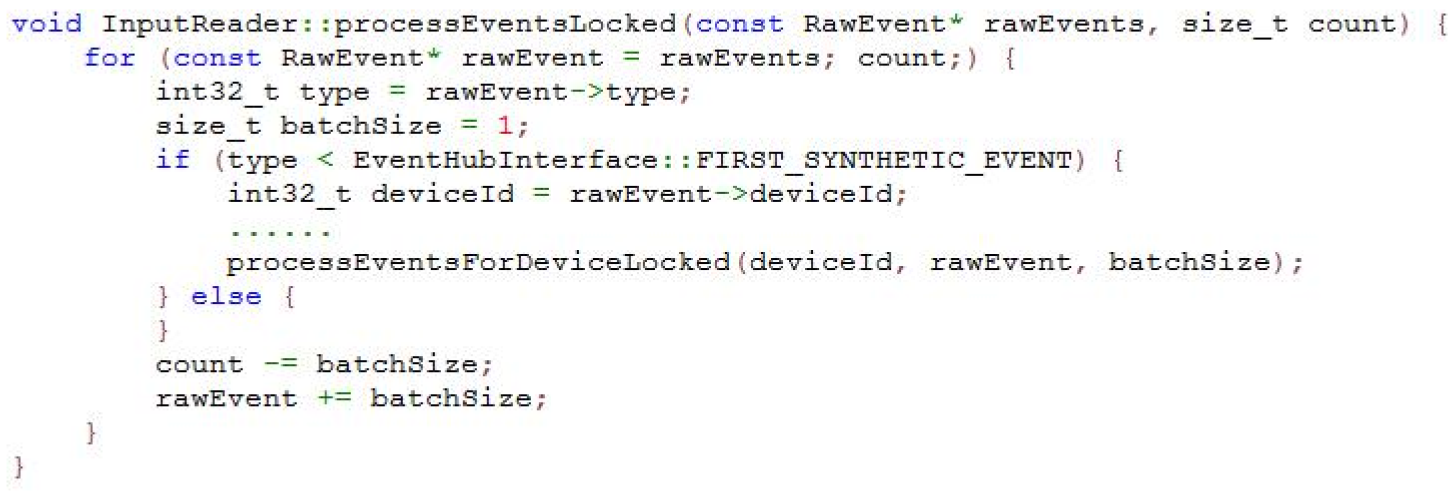
输入事件的分发过程(续)

- InputReader获得输入事件 – InputReader.loopOnce

```
void InputReader::loopOnce() {
    ......
    int32_t timeoutMillis;
    ......
    Vector<InputDeviceInfo> inputDevices;
    { // acquire lock
        AutoMutex _l(mLock);
        ......
        timeoutMillis = -1;

        uint32_t changes = mConfigurationChangesToRefresh;
        if (changes) {
            ......
            refreshConfigurationLocked(changes);
        } else if (mNextTimeout != LLONG_MAX) {
            nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
            timeoutMillis = toMillisecondTimeoutDelay(now, mNextTimeout);
        }
    } // release lock

    size_t count = mEventHub->getEvents(timeoutMillis, mEventBuffer, EVENT_BUFFER_SIZE);

    { // acquire lock
        AutoMutex _l(mLock);
        mReaderIsAliveCondition.broadcast();

        if (count) {
            processEventsLocked(mEventBuffer, count);
        }
        ......
    } // release lock
    ......
}
```

输入事件的分发过程(续)

- InputReader获得输入事件 – InputReader.processEventsLocked



```
void InputReader::processEventsLocked(const RawEvent* rawEvents, size_t count) {
    for (const RawEvent* rawEvent = rawEvents; count;) {
        int32_t type = rawEvent->type;
        size_t batchSize = 1;
        if (type < EventHubInterface::FIRST_SYNTHETIC_EVENT) {
            int32_t deviceId = rawEvent->deviceId;
            ......
            processEventsForDeviceLocked(deviceId, rawEvent, batchSize);
        } else {
        }
        count -= batchSize;
        rawEvent += batchSize;
    }
}
```

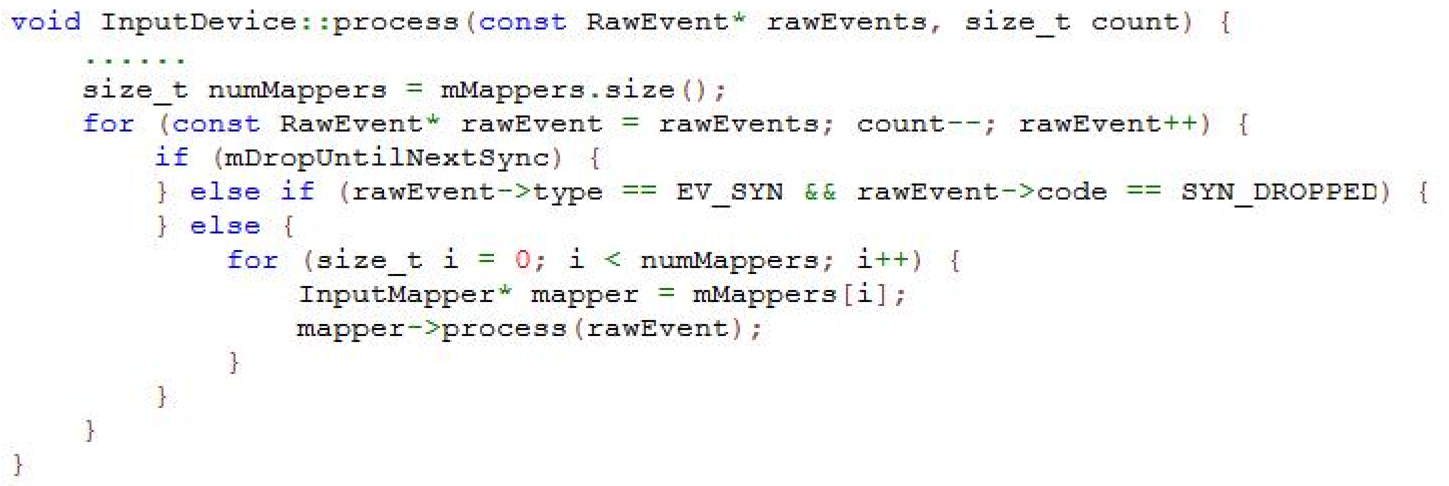
输入事件的分发过程(续)

- InputReader获得输入事件 – InputReader.processEventsForDeviceLocked

```
void InputReader::processEventsForDeviceLocked(int32_t deviceId,
        const RawEvent* rawEvents, size_t count) {
    ssize_t deviceIndex = mDevices.indexOfKey(deviceId);
    if (deviceIndex < 0) {
        ALOGW("Discarding event for unknown deviceId %d.", deviceId);
        return;
    }

    InputDevice* device = mDevices.valueAt(deviceIndex);
    if (device->isIgnored()) {
        //ALOGD("Discarding event for ignored deviceId %d.", deviceId);
        return;
    }

    device->process(rawEvents, count);
}
```

输入事件的分发过程(续)

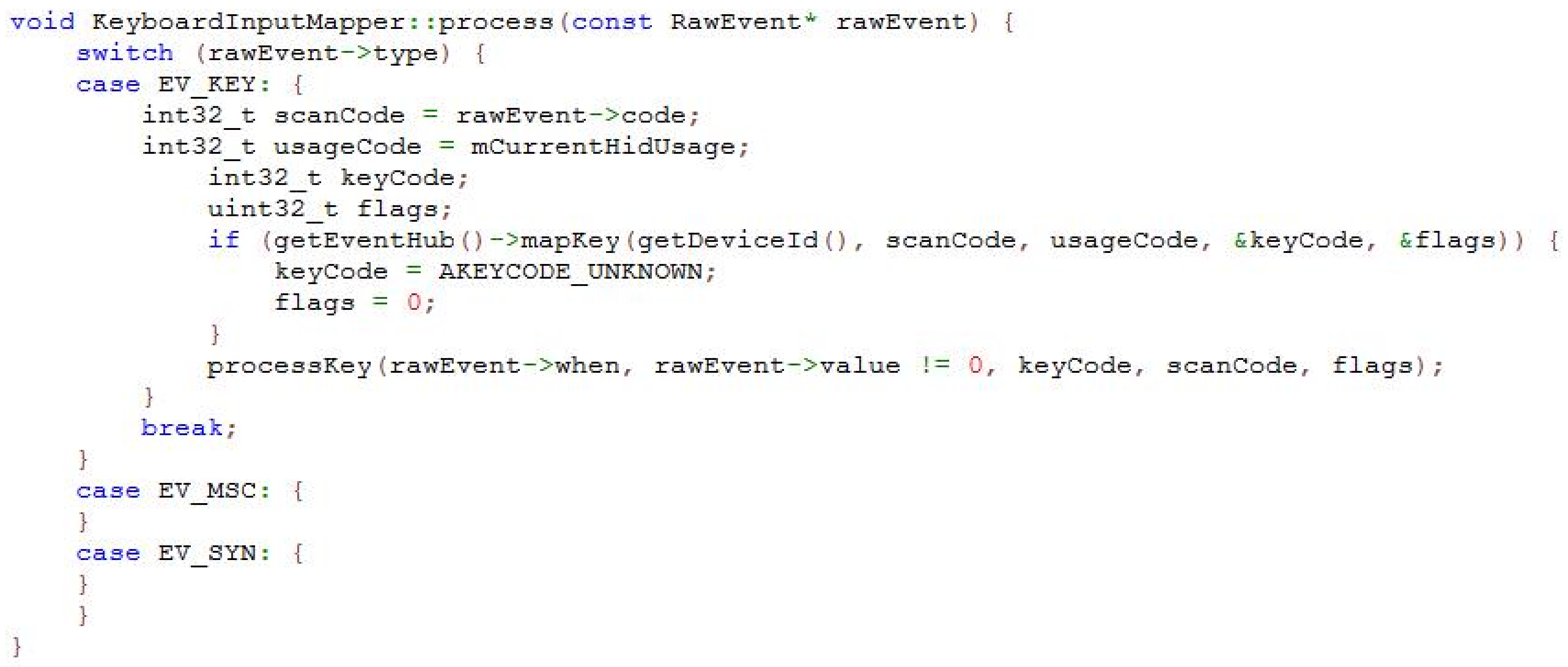
- InputReader获得输入事件—InputDevice.process



```
void InputDevice::process(const RawEvent* rawEvents, size_t count) {
    ......
    size_t numMappers = mMappers.size();
    for (const RawEvent* rawEvent = rawEvents; count--; rawEvent++) {
        if (mDropUntilNextSync) {
        } else if (rawEvent->type == EV_SYN && rawEvent->code == SYN_DROPPED) {
        } else {
            for (size_t i = 0; i < numMappers; i++) {
                InputMapper* mapper = mMappers[i];
                mapper->process(rawEvent);
            }
        }
    }
}
```

输入事件的分发过程(续)

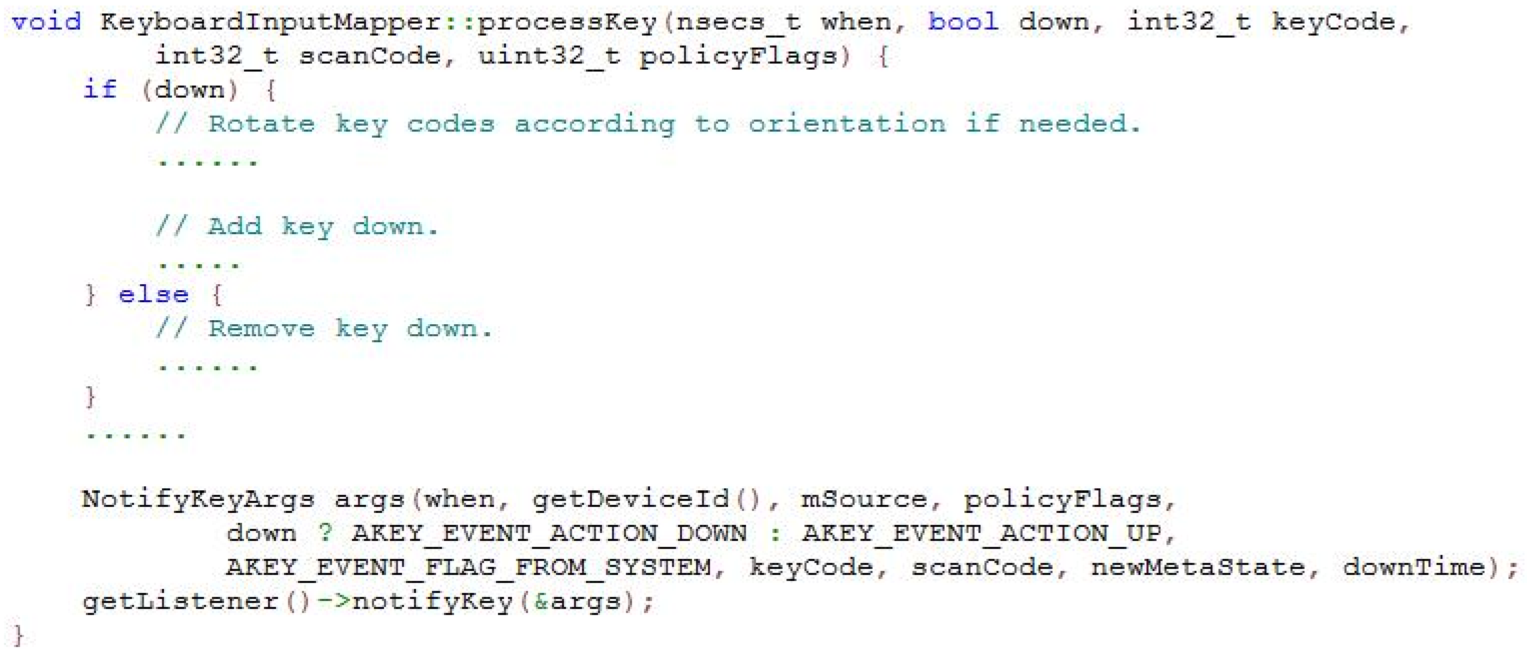
- InputReader获得输入事件—KeyboardInputMapper.process



```
void KeyboardInputMapper::process(const RawEvent* rawEvent) {
    switch (rawEvent->type) {
    case EV_KEY: {
        int32_t scanCode = rawEvent->code;
        int32_t usageCode = mCurrentHidUsage;
            int32_t keyCode;
            uint32_t flags;
            if (getEventHub()->mapKey(getDeviceId(), scanCode, usageCode, &keyCode, &flags)) {
                keyCode = AKEYCODE_UNKNOWN;
                flags = 0;
            }
            processKey(rawEvent->when, rawEvent->value != 0, keyCode, scanCode, flags);
        }
        break;
    }
    case EV_MSC: {
    }
    case EV_SYN: {
    }
    }
}
```

输入事件的分发过程(续)

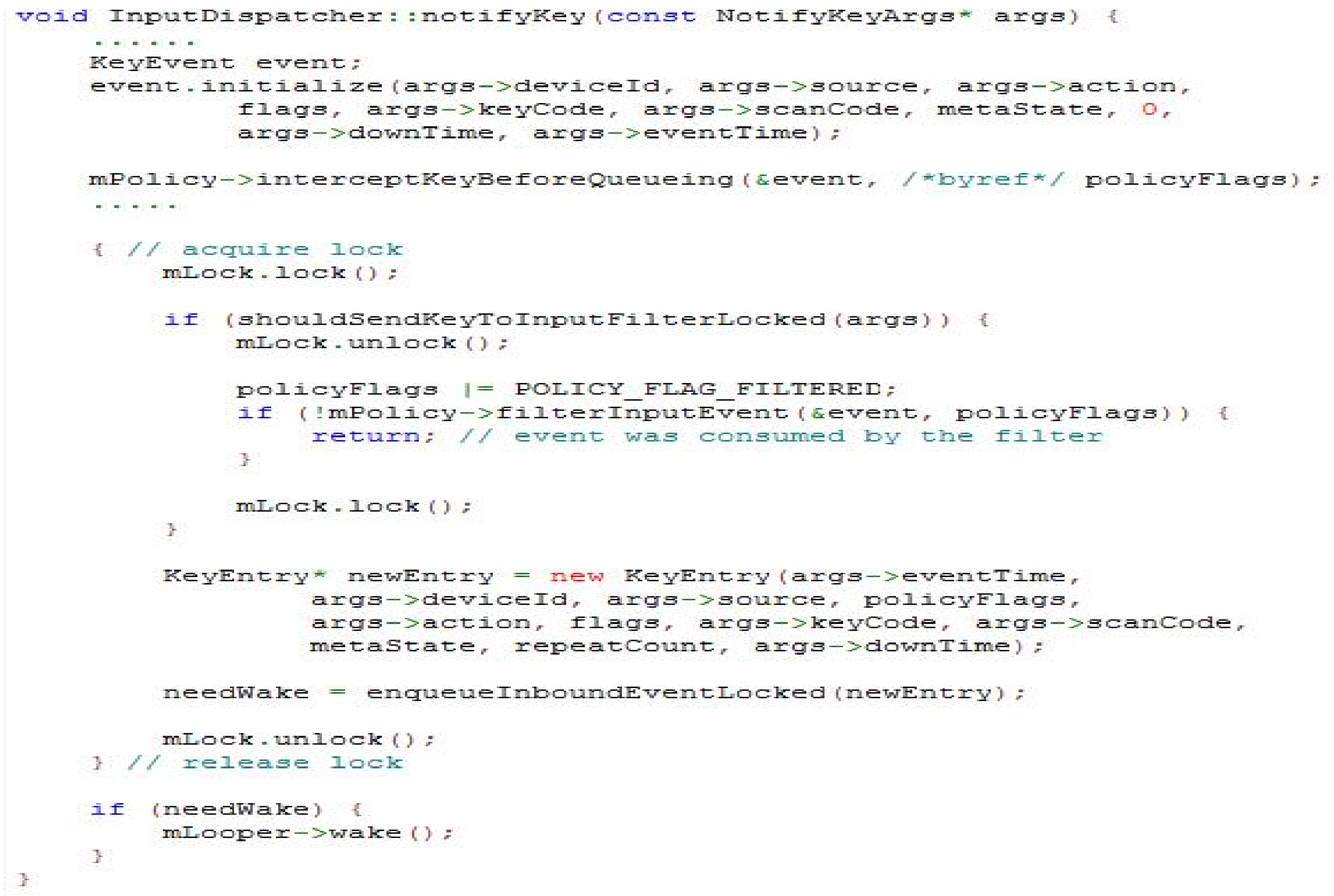
- InputReader获得输入事件—KeyboardInputMapper.processKey



```
void KeyboardInputMapper::processKey(nsecs_t when, bool down, int32_t keyCode,
        int32_t scanCode, uint32_t policyFlags) {
    if (down) {
        // Rotate key codes according to orientation if needed.
        ......

        // Add key down.
        .....
    } else {
        // Remove key down.
        ......
    }
    ......

    NotifyKeyArgs args(when, getDeviceId(), mSource, policyFlags,
            down ? AKEY_EVENT_ACTION_DOWN : AKEY_EVENT_ACTION_UP,
            AKEY_EVENT_FLAG_FROM_SYSTEM, keyCode, scanCode, newMetaState, downTime);
    getListener()->notifyKey(&args);
}
```

输入事件的分发过程(续)

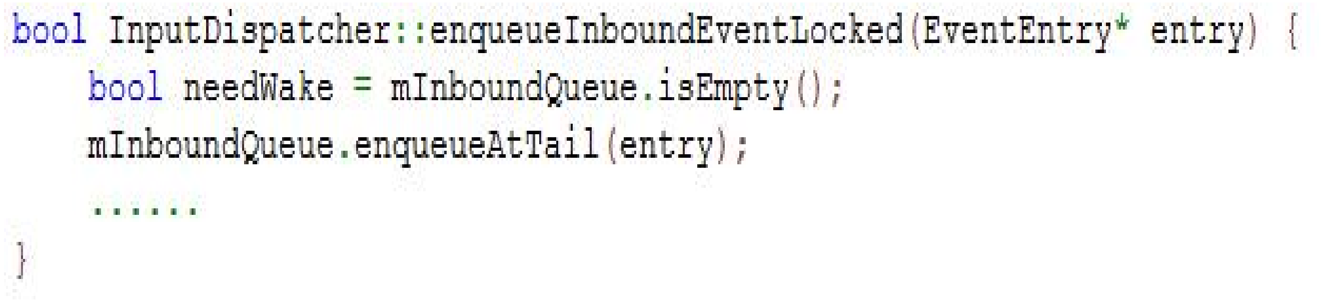
- InputReader获得输入事件—InputDispatcher.notifyKey



```
void InputDispatcher::notifyKey(const NotifyKeyArgs* args) {
    .......
    KeyEvent event;
    event.initialize(args->deviceId, args->source, args->action,
            flags, args->keyCode, args->scanCode, metaState, 0,
            args->downTime, args->eventTime);

    mPolicy->interceptKeyBeforeQueueing(&event, /*byref*/ policyFlags);
    ......

    { // acquire lock
        mLock.lock();

        if (shouldSendKeyToInputFilterLocked(args)) {
            mLock.unlock();

            policyFlags |= POLICY_FLAG_FILTERED;
            if (!mPolicy->filterInputEvent(&event, policyFlags)) {
                return; // event was consumed by the filter
            }

            mLock.lock();
        }

        KeyEntry* newEntry = new KeyEntry(args->eventTime,
                args->deviceId, args->source, policyFlags,
                args->action, flags, args->keyCode, args->scanCode,
                metaState, repeatCount, args->downTime);

        needWake = enqueueInboundEventLocked(newEntry);

        mLock.unlock();
    } // release lock

    if (needWake) {
        mLooper->wake();
    }
}
```

输入事件的分发过程(续)

- InputReader获得输入事件—InputDispatcher.
enqueueInboundKeyLocked



```
bool InputDispatcher::enqueueInboundEventLocked(EventEntry* entry) {
    bool needWake = mInboundQueue.isEmpty();
    mInboundQueue.enqueueAtTail(entry);
    ......
}
```

输入事件的分发过程(续)

- InputDispatcher分发键盘事件 – InputDispatcher. dispatchOnceInnerLocked

```
void InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {
    ......

    // Ready to start a new event.
    // If we don't already have a pending event, go grab one.
    if (! mPendingEvent) {
        if (mInboundQueue.isEmpty()) {
        } else {
            // Inbound queue has at least one entry.
            mPendingEvent = mInboundQueue.dequeueAtHead();
            ......
        }
        ......
    }

    switch (mPendingEvent->type) {
    ......
    case EventEntry::TYPE_KEY: {
        KeyEntry* typedEntry = static_cast<KeyEntry*>(mPendingEvent);
        ......
        done = dispatchKeyLocked(currentTime, typedEntry, &dropReason, nextWakeupTime);
        break;
    case EventEntry::TYPE_MOTION: {
        MotionEntry* typedEntry = static_cast<MotionEntry*>(mPendingEvent);
        ......
        done = dispatchMotionLocked(currentTime, typedEntry,
                &dropReason, nextWakeupTime);
        break;
    }

    ......
}
```

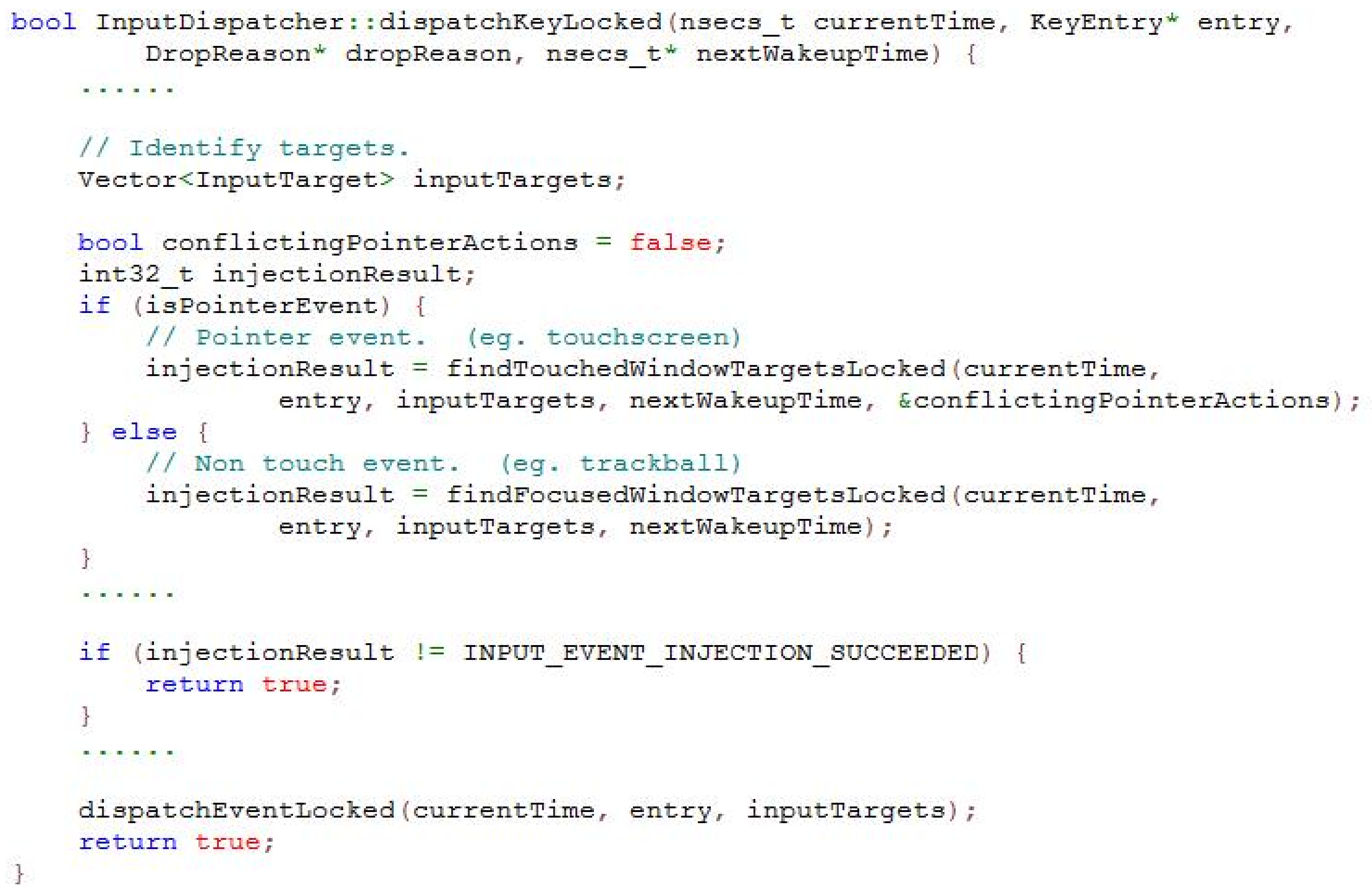
输入事件的分发过程(续)

- InputDispatcher分发键盘事件 – InputDispatcher. dispatchKeyLocked

```
bool InputDispatcher::dispatchKeyLocked(nsecs_t currentTime, KeyEntry* entry,
        DropReason* dropReason, nsecs_t* nextWakeupTime) {
    ......

    // Give the policy a chance to intercept the key.
    if (entry->interceptKeyResult == KeyEntry::INTERCEPT_KEY_RESULT_UNKNOWN) {
        if (entry->policyFlags & POLICY_FLAG_PASS_TO_USER) {
            CommandEntry* commandEntry = postCommandLocked(
                    & InputDispatcher::doInterceptKeyBeforeDispatchingLockedInterruptible);
            ......
            return false; // wait for the command to run
        }
    }
    ......
    // Identify targets.
    Vector<InputTarget> inputTargets;
    int32_t injectionResult = findFocusedWindowTargetsLocked(currentTime,
            entry, inputTargets, nextWakeupTime);
    ......

    setInjectionResultLocked(entry, injectionResult);
    if (injectionResult != INPUT_EVENT_INJECTION_SUCCEEDED) {
        return true;
    }

    ......

    // Dispatch the key.
    dispatchEventLocked(currentTime, entry, inputTargets);
    return true;
}
```

输入事件的分发过程(续)

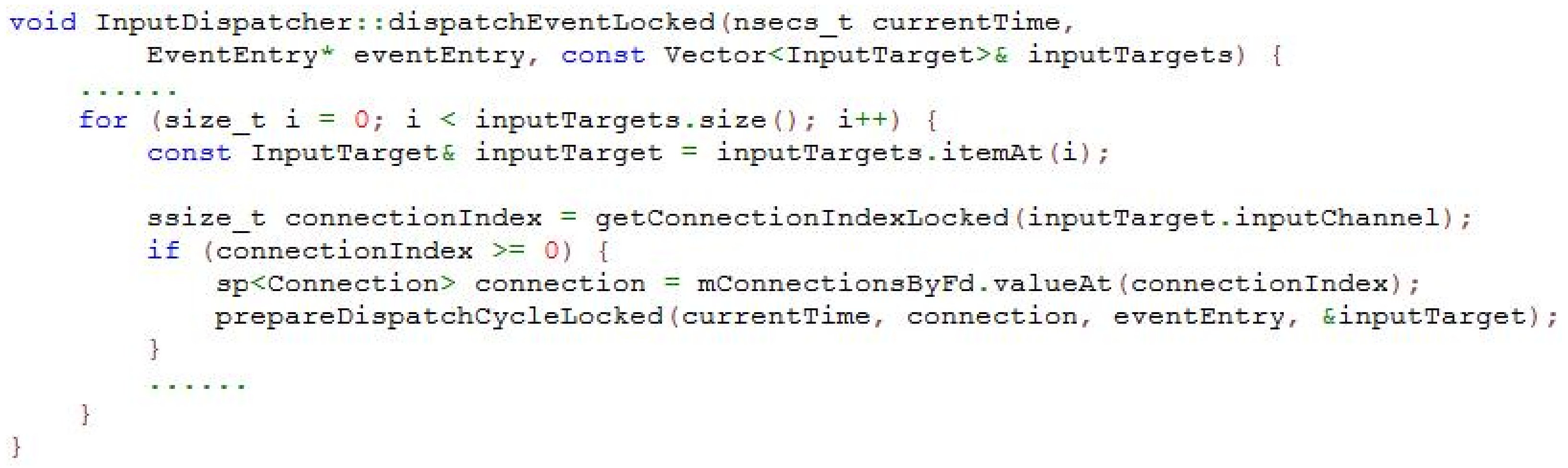
- InputDispatcher分发键盘事件 – InputDispatcher.dispatchMotionLocked



```
bool InputDispatcher::dispatchKeyLocked(nsecs_t currentTime, KeyEntry* entry,
        DropReason* dropReason, nsecs_t* nextWakeupTime) {
    ......

    // Identify targets.
    Vector<InputTarget> inputTargets;

    bool conflictingPointerActions = false;
    int32_t injectionResult;
    if (isPointerEvent) {
        // Pointer event.  (eg. touchscreen)
        injectionResult = findTouchedWindowTargetsLocked(currentTime,
                entry, inputTargets, nextWakeupTime, &conflictingPointerActions);
    } else {
        // Non touch event.  (eg. trackball)
        injectionResult = findFocusedWindowTargetsLocked(currentTime,
                entry, inputTargets, nextWakeupTime);
    }
    ......

    if (injectionResult != INPUT_EVENT_INJECTION_SUCCEEDED) {
        return true;
    }
    ......

    dispatchEventLocked(currentTime, entry, inputTargets);
    return true;
}
```

输入事件的分发过程(续)

- InputDispatcher分发键盘事件--InputDispatcher.dispatchEventLocked



```
void InputDispatcher::dispatchEventLocked(nsecs_t currentTime,
        EventEntry* eventEntry, const Vector<InputTarget>& inputTargets) {
    ......
    for (size_t i = 0; i < inputTargets.size(); i++) {
        const InputTarget& inputTarget = inputTargets.itemAt(i);

        ssize_t connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);
        if (connectionIndex >= 0) {
            sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex);
            prepareDispatchCycleLocked(currentTime, connection, eventEntry, &inputTarget);
        }
        ......
    }
}
```

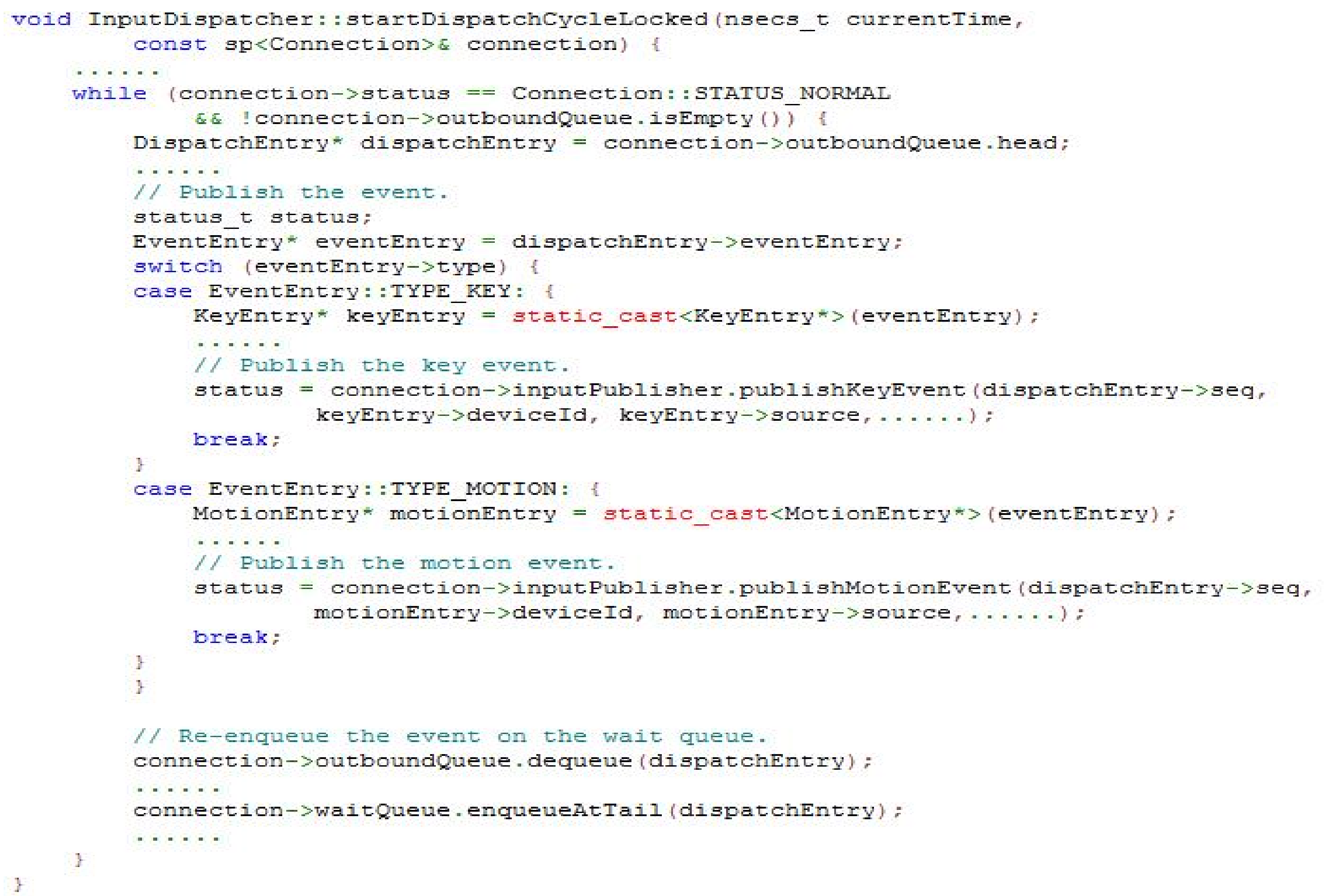
输入事件的分发过程(续)

- InputDispatcher分发键盘事件--InputDispatcher.prepareDispatchCycleLocked

```
void InputDispatcher::prepareDispatchCycleLocked(nsecs_t currentTime,
        const sp<Connection>& connection, EventEntry* eventEntry, const InputTarget* inputTarget
    ......
    enqueueDispatchEntriesLocked(currentTime, connection, eventEntry, inputTarget);
}

void InputDispatcher::enqueueDispatchEntriesLocked(nsecs_t currentTime,
        const sp<Connection>& connection, EventEntry* eventEntry, const InputTarget* inputTarget
    bool wasEmpty = connection->outboundQueue.isEmpty();

    // Enqueue dispatch entries for the requested modes.
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_HOVER_EXIT);
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_OUTSIDE);
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_HOVER_ENTER);
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_IS);
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_SLIPPERY_EXIT);
    enqueueDispatchEntryLocked(connection, eventEntry, inputTarget,
            InputTarget::FLAG_DISPATCH_AS_SLIPPERY_ENTER);

    // If the outbound queue was previously empty, start the dispatch cycle going.
    if (wasEmpty && !connection->outboundQueue.isEmpty()) {
        startDispatchCycleLocked(currentTime, connection);
    }
}
```

输入事件的分发过程(续)

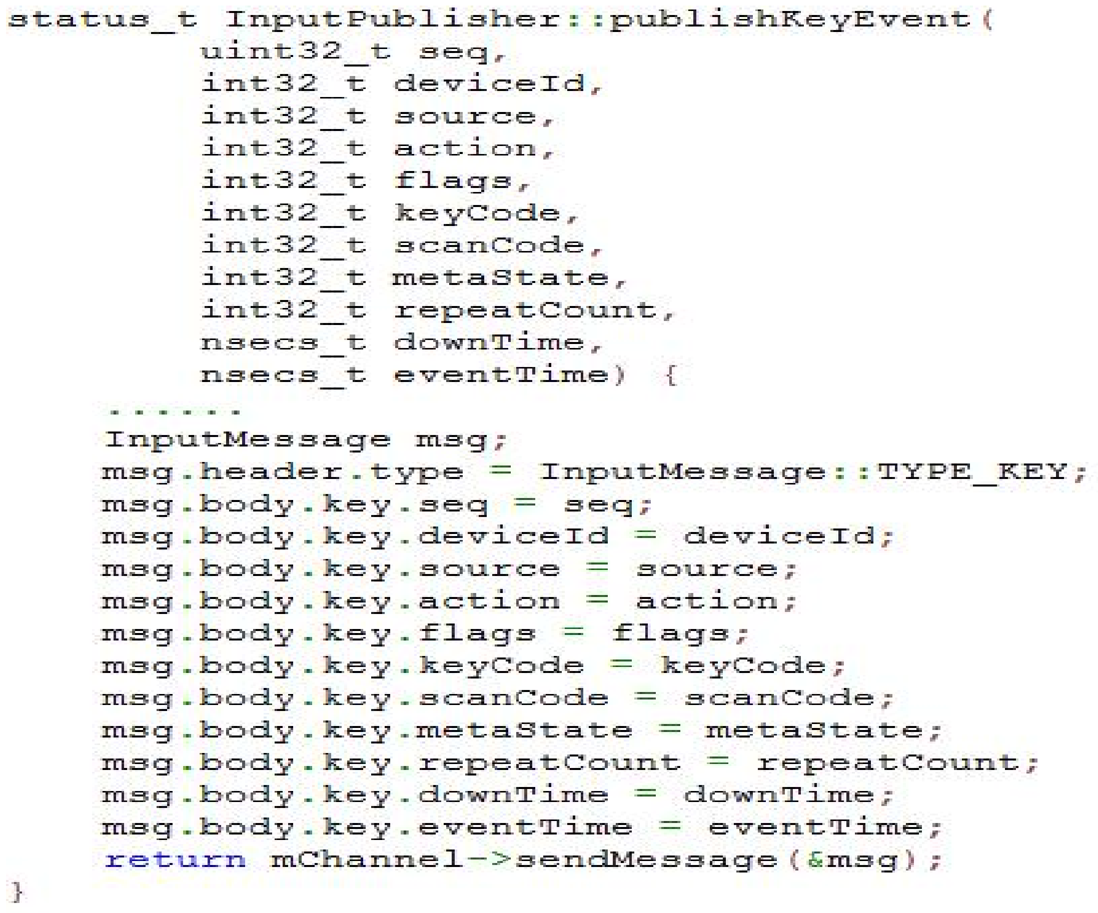
- InputDispatcher分发键盘事件--InputDispatcher.startDispatchCycleLocked



```
void InputDispatcher::startDispatchCycleLocked(nsecs_t currentTime,
        const sp<Connection>& connection) {
    ......
    while (connection->status == Connection::STATUS_NORMAL
            && !connection->outboundQueue.isEmpty()) {
        DispatchEntry* dispatchEntry = connection->outboundQueue.head;
        ......
        // Publish the event.
        status_t status;
        EventEntry* eventEntry = dispatchEntry->eventEntry;
        switch (eventEntry->type) {
        case EventEntry::TYPE_KEY: {
            KeyEntry* keyEntry = static_cast<KeyEntry*>(eventEntry);
            ......
            // Publish the key event.
            status = connection->inputPublisher.publishKeyEvent(dispatchEntry->seq,
                    keyEntry->deviceId, keyEntry->source,......);
            break;
        }
        case EventEntry::TYPE_MOTION: {
            MotionEntry* motionEntry = static_cast<MotionEntry*>(eventEntry);
            ......
            // Publish the motion event.
            status = connection->inputPublisher.publishMotionEvent(dispatchEntry->seq,
                    motionEntry->deviceId, motionEntry->source,......);
            break;
        }
        }

        // Re-enqueue the event on the wait queue.
        connection->outboundQueue.dequeue(dispatchEntry);
        ......
        connection->waitQueue.enqueueAtTail(dispatchEntry);
        ......
    }
}
```

输入事件的分发过程(续)

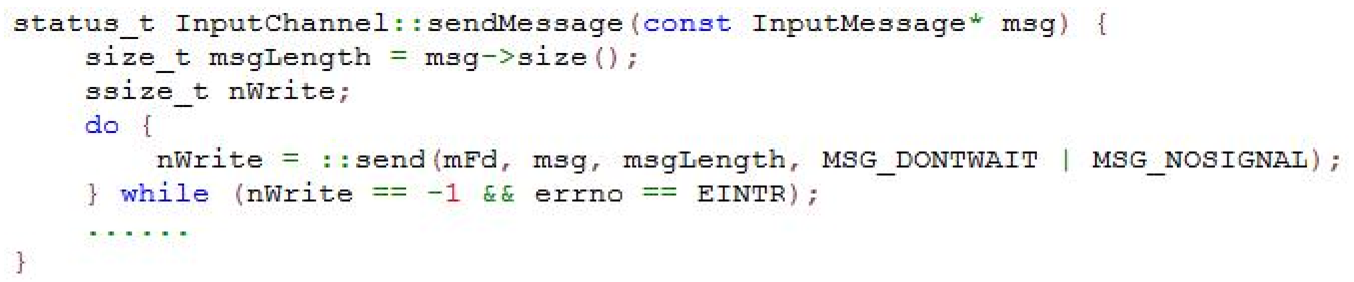
- InputDispatcher分发键盘事件—InputPublisher.publishKeyEvent



```
status_t InputPublisher::publishKeyEvent(
        uint32_t seq,
        int32_t deviceId,
        int32_t source,
        int32_t action,
        int32_t flags,
        int32_t keyCode,
        int32_t scanCode,
        int32_t metaState,
        int32_t repeatCount,
        nsecs_t downTime,
        nsecs_t eventTime) {
    .......
    InputMessage msg;
    msg.header.type = InputMessage::TYPE_KEY;
    msg.body.key.seq = seq;
    msg.body.key.deviceId = deviceId;
    msg.body.key.source = source;
    msg.body.key.action = action;
    msg.body.key.flags = flags;
    msg.body.key.keyCode = keyCode;
    msg.body.key.scanCode = scanCode;
    msg.body.key.metaState = metaState;
    msg.body.key.repeatCount = repeatCount;
    msg.body.key.downTime = downTime;
    msg.body.key.eventTime = eventTime;
    return mChannel->sendMessage(&msg);
}
```

输入事件的分发过程(续)

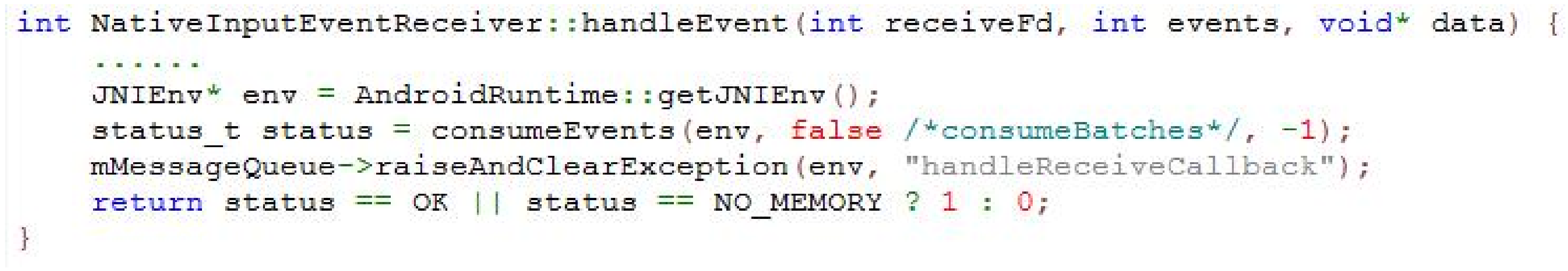
- InputDispatcher分发键盘事件—InputChannel::sendMessage



```
status_t InputChannel::sendMessage(const InputMessage* msg) {
    size_t msgLength = msg->size();
    ssize_t nWrite;
    do {
        nWrite = ::send(mFd, msg, msgLength, MSG_DONTWAIT | MSG_NOSIGNAL);
    } while (nWrite == -1 && errno == EINTR);
    ......
}
```

输入事件的分发过程(续)

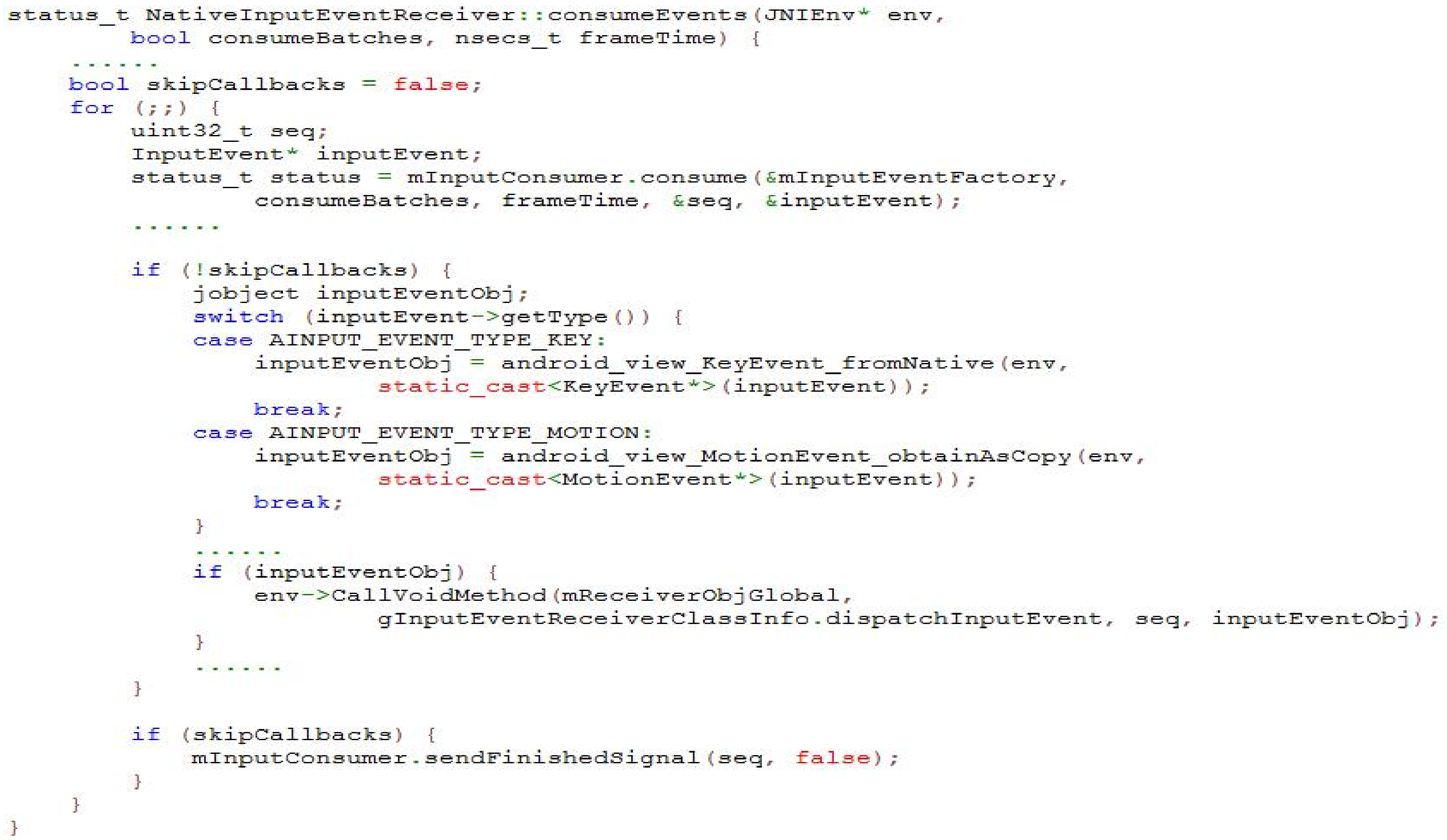
- App获得键盘事件—NativeInputEventReceiver.handleEvent



```
int NativeInputEventReceiver::handleEvent(int receiveFd, int events, void* data) {
    ......
    JNIEnv* env = AndroidRuntime::getJNIEnv();
    status_t status = consumeEvents(env, false /*consumeBatches*/, -1);
    mMessageQueue->raiseAndClearException(env, "handleReceiveCallback");
    return status == OK || status == NO_MEMORY ? 1 : 0;
}
```

输入事件的分发过程(续)

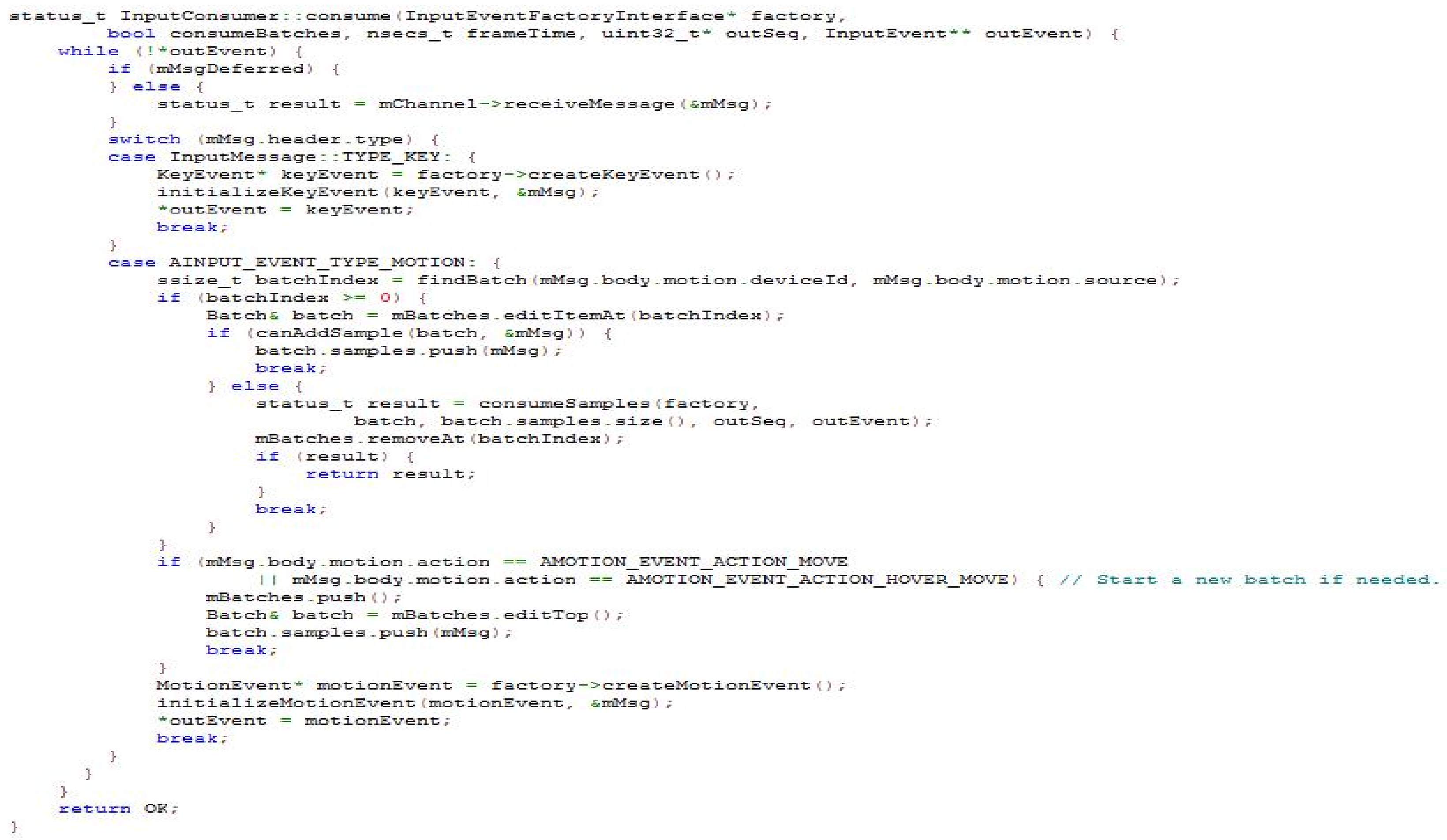
- App获得键盘事件－NativeInputEventReceiver.consumeEvents



```
status_t NativeInputEventReceiver::consumeEvents(JNIEnv* env,
        bool consumeBatches, nsecs_t frameTime) {
    ......
    bool skipCallbacks = false;
    for (;;) {
        uint32_t seq;
        InputEvent* inputEvent;
        status_t status = mInputConsumer.consume(&mInputEventFactory,
                consumeBatches, frameTime, &seq, &inputEvent);
        ......

        if (!skipCallbacks) {
            jobject inputEventObj;
            switch (inputEvent->getType()) {
            case AINPUT_EVENT_TYPE_KEY:
                inputEventObj = android_view_KeyEvent_fromNative(env,
                        static_cast<KeyEvent*>(inputEvent));
                break;
            case AINPUT_EVENT_TYPE_MOTION:
                inputEventObj = android_view_MotionEvent_obtainAsCopy(env,
                        static_cast<MotionEvent*>(inputEvent));
                break;
            }
            ......
            if (inputEventObj) {
                env->CallVoidMethod(mReceiverObjGlobal,
                        gInputEventReceiverClassInfo.dispatchInputEvent, seq, inputEventObj);
            }
            ......
        }

        if (skipCallbacks) {
            mInputConsumer.sendFinishedSignal(seq, false);
        }
    }
}
```

输入事件的分发过程(续)

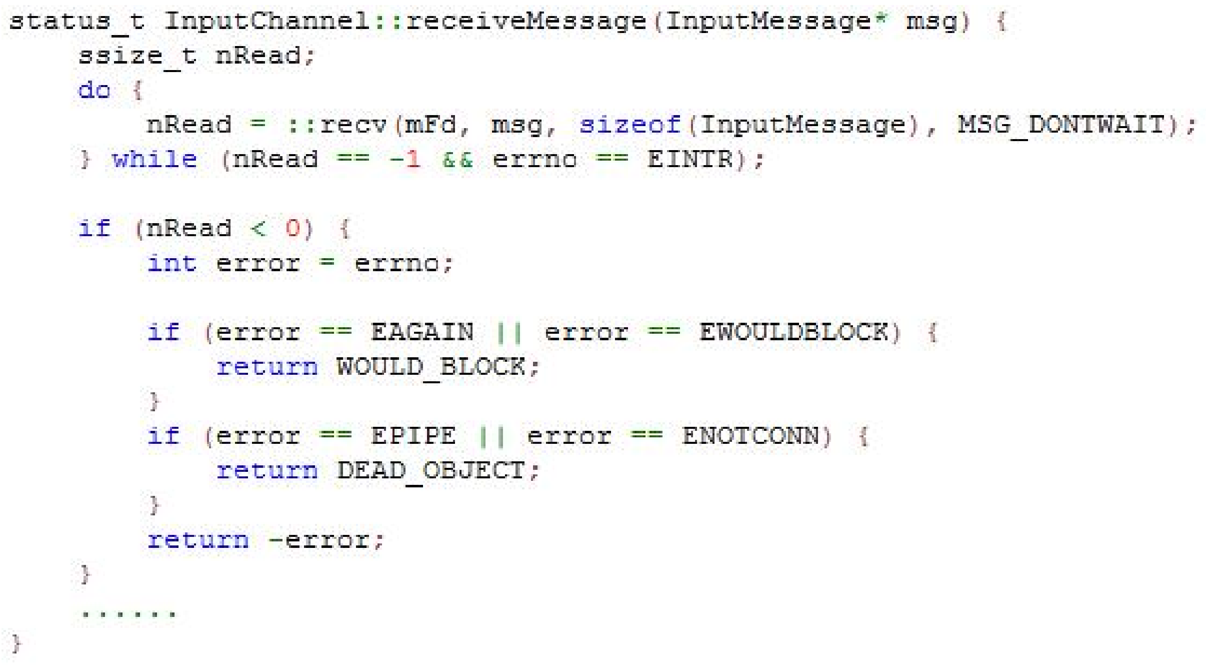
- App获得键盘事件—InputComsumer.consume



```
status_t InputConsumer::consume(InputEventFactoryInterface* factory,
        bool consumeBatches, nsecs_t frameTime, uint32_t* outSeq, InputEvent** outEvent) {
    while (!*outEvent) {
        if (mMsgDeferred) {
        } else {
            status_t result = mChannel->receiveMessage(&mMsg);
        }
        switch (mMsg.header.type) {
        case InputMessage::TYPE_KEY: {
            KeyEvent* keyEvent = factory->createKeyEvent();
            initializeKeyEvent(keyEvent, &mMsg);
            *outEvent = keyEvent;
            break;
        }
        case AINPUT_EVENT_TYPE_MOTION: {
            ssize_t batchIndex = findBatch(mMsg.body.motion.deviceId, mMsg.body.motion.source);
            if (batchIndex >= 0) {
                Batch& batch = mBatches.editItemAt(batchIndex);
                if (canAddSample(batch, &mMsg)) {
                    batch.samples.push(mMsg);
                    break;
                } else {
                    status_t result = consumeSamples(factory,
                            batch, batch.samples.size(), outSeq, outEvent);
                    mBatches.removeAt(batchIndex);
                    if (result) {
                        return result;
                    }
                    break;
                }
            }
            if (mMsg.body.motion.action == AMOTION_EVENT_ACTION_MOVE
                    || mMsg.body.motion.action == AMOTION_EVENT_ACTION_HOVER_MOVE) { // Start a new batch if needed.
                mBatches.push();
                Batch& batch = mBatches.editTop();
                batch.samples.push(mMsg);
                break;
            }
            MotionEvent* motionEvent = factory->createMotionEvent();
            initializeMotionEvent(motionEvent, &mMsg);
            *outEvent = motionEvent;
            break;
        }
        }
    }
    return OK;
}
```

输入事件的分发过程(续)

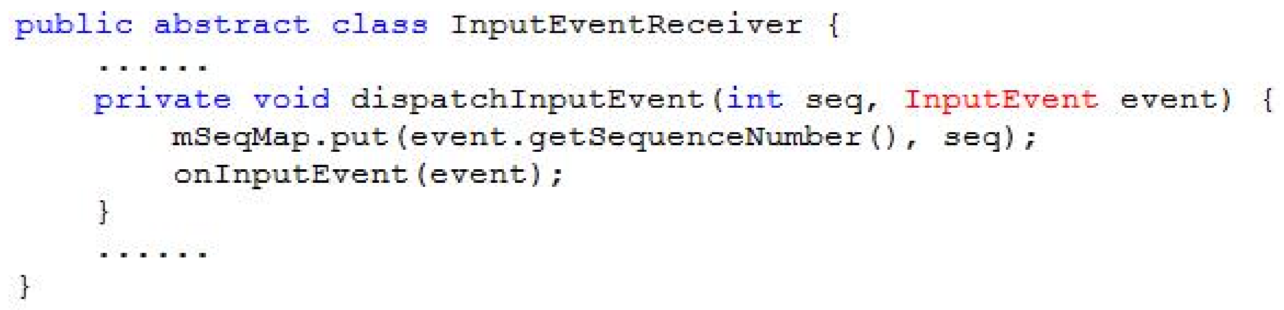
- App获得键盘事件— InputChannel.receiveMessage



```
status_t InputChannel::receiveMessage(InputMessage* msg) {
    ssize_t nRead;
    do {
        nRead = ::recv(mFd, msg, sizeof(InputMessage), MSG_DONTWAIT);
    } while (nRead == -1 && errno == EINTR);

    if (nRead < 0) {
        int error = errno;

        if (error == EAGAIN || error == EWOULDBLOCK) {
            return WOULD_BLOCK;
        }
        if (error == EPIPE || error == ENOTCONN) {
            return DEAD_OBJECT;
        }
        return -error;
    }
    ......
}
```

输入事件的分发过程(续)

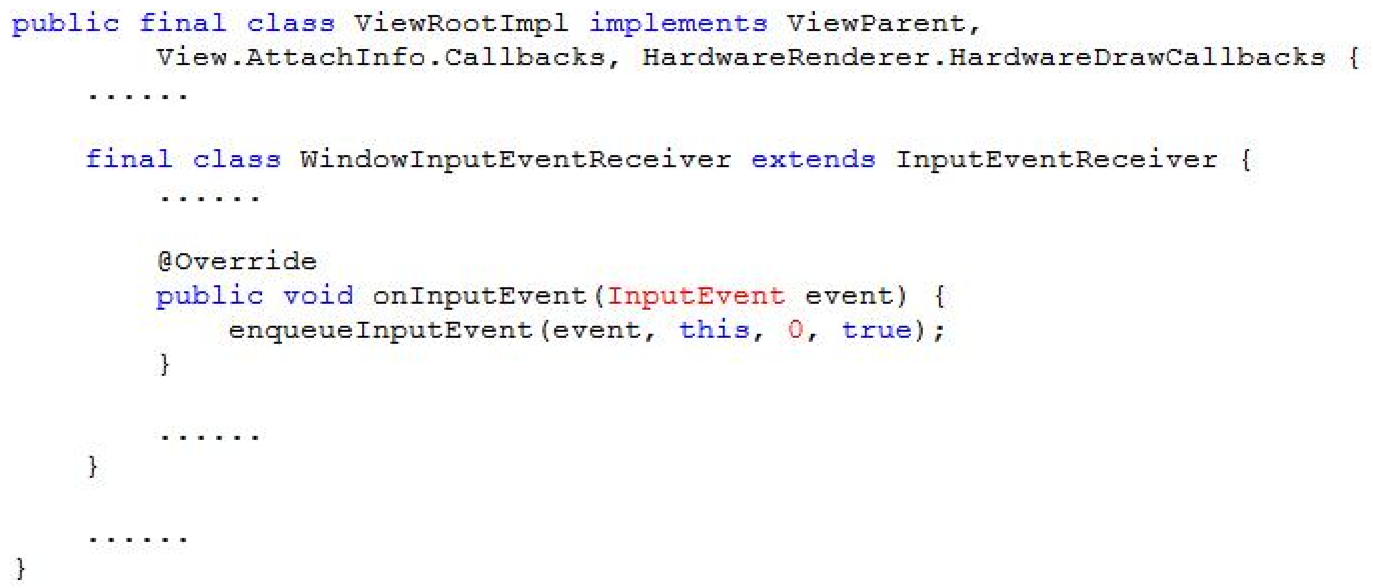
- App获得键盘事件—InputEventReceiver.dispatchInputEvent



```
public abstract class InputEventReceiver {
    ......
    private void dispatchInputEvent(int seq, InputEvent event) {
        mSeqMap.put(event.getSequenceNumber(), seq);
        onInputEvent(event);
    }
    ......
}
```

输入事件的分发过程(续)

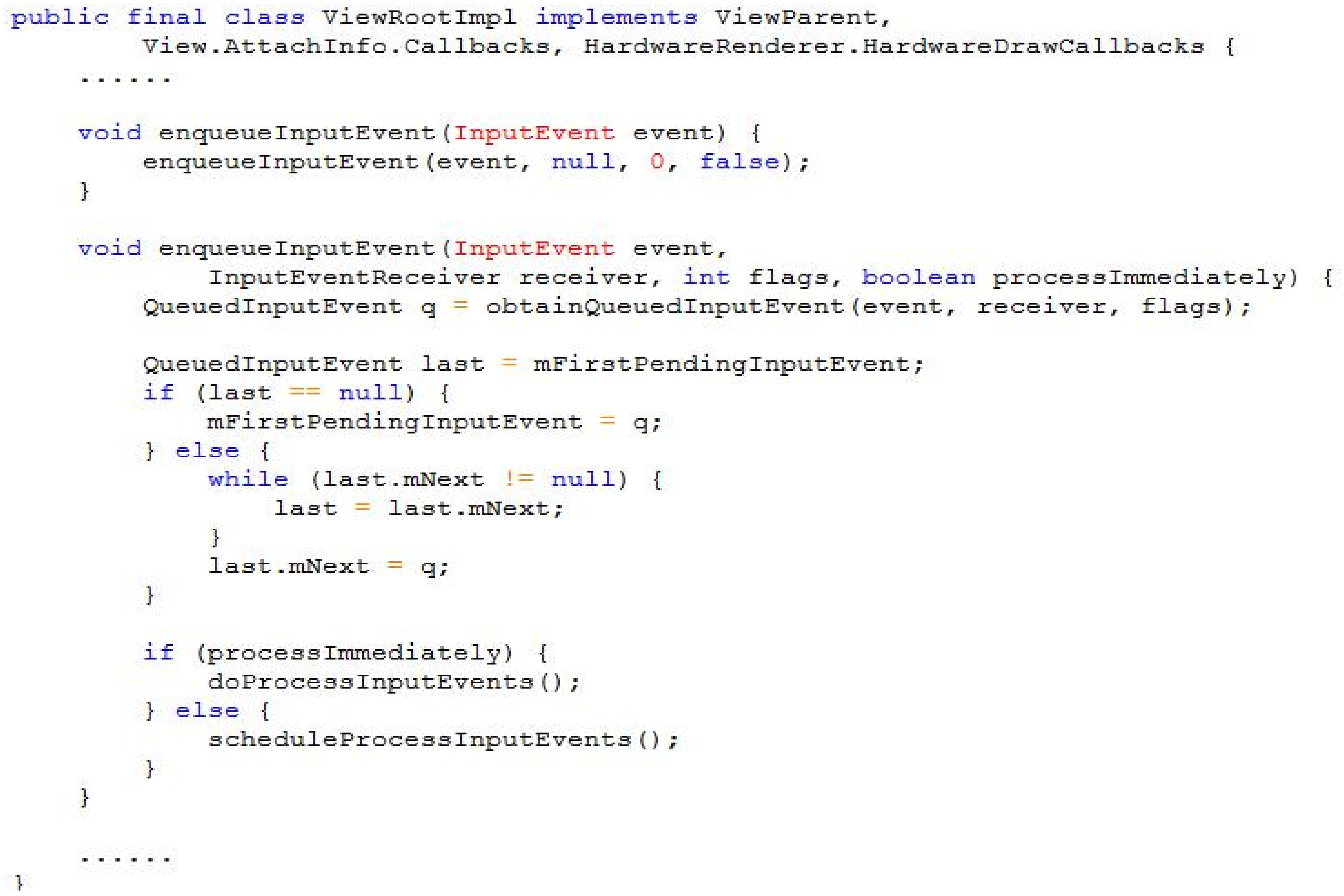
- App获得键盘事件—WindowInputEventReceiver.onInputEvent



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    final class WindowInputEventReceiver extends InputEventReceiver {
        ......

        @Override
        public void onInputEvent(InputEvent event) {
            enqueueInputEvent(event, this, 0, true);
        }

        ......
    }

    ......
}
```

输入事件的分发过程(续)

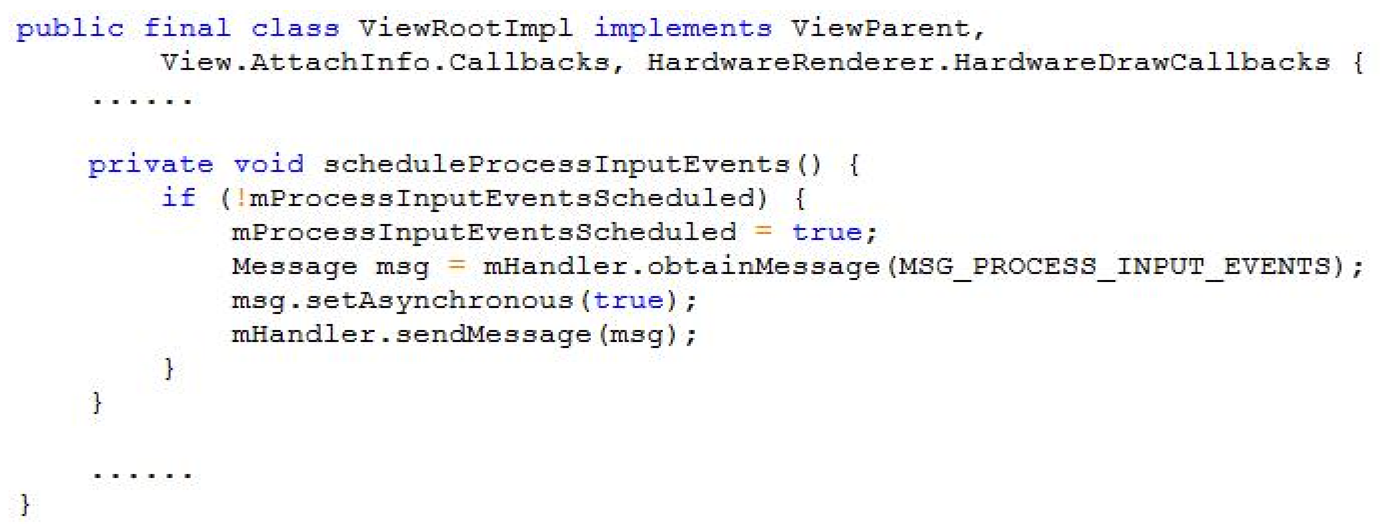
- App获得键盘事件—ViewRootImpl.enqueueInputEvent



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    void enqueueInputEvent(InputEvent event) {
        enqueueInputEvent(event, null, 0, false);
    }

    void enqueueInputEvent(InputEvent event,
            InputEventReceiver receiver, int flags, boolean processImmediately) {
        QueuedInputEvent q = obtainQueuedInputEvent(event, receiver, flags);

        QueuedInputEvent last = mFirstPendingInputEvent;
        if (last == null) {
            mFirstPendingInputEvent = q;
        } else {
            while (last.mNext != null) {
                last = last.mNext;
            }
            last.mNext = q;
        }

        if (processImmediately) {
            doProcessInputEvents();
        } else {
            scheduleProcessInputEvents();
        }
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl. scheduleProcessInputEvents



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    private void scheduleProcessInputEvents() {
        if (!mProcessInputEventsScheduled) {
            mProcessInputEventsScheduled = true;
            Message msg = mHandler.obtainMessage(MSG_PROCESS_INPUT_EVENTS);
            msg.setAsynchronous(true);
            mHandler.sendMessage(msg);
        }
    }

    ......
}
```

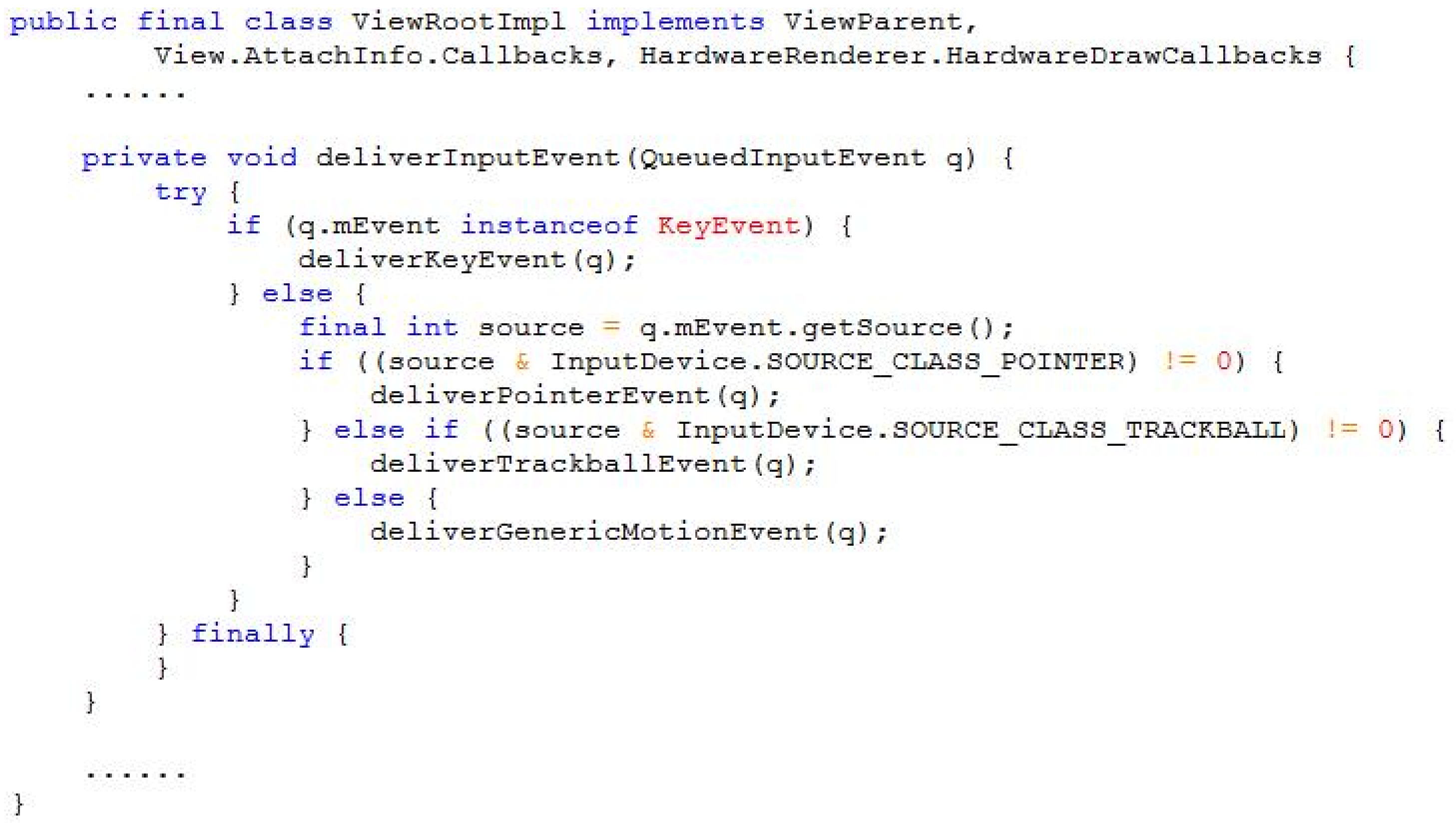
输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl.doProcessInputEvents

```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    void doProcessInputEvents() {
        while (mCurrentInputEvent == null && mFirstPendingInputEvent != null) {
            QueuedInputEvent q = mFirstPendingInputEvent;
            mFirstPendingInputEvent = q.mNext;
            q.mNext = null;
            mCurrentInputEvent = q;
            deliverInputEvent(q);
        }
        ......
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl.
  deliverInputEvent



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    private void deliverInputEvent(QueuedInputEvent q) {
        try {
            if (q.mEvent instanceof KeyEvent) {
                deliverKeyEvent(q);
            } else {
                final int source = q.mEvent.getSource();
                if ((source & InputDevice.SOURCE_CLASS_POINTER) != 0) {
                    deliverPointerEvent(q);
                } else if ((source & InputDevice.SOURCE_CLASS_TRACKBALL) != 0) {
                    deliverTrackballEvent(q);
                } else {
                    deliverGenericMotionEvent(q);
                }
            }
        } finally {
        }
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl. deliverKeyEvent

```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    private void deliverKeyEvent(QueuedInputEvent q) {
        final KeyEvent event = (KeyEvent)q.mEvent;
        ......

        if (mView != null && mAdded && (q.mFlags & QueuedInputEvent.FLAG_DELIVER_POST_IME) == 0) {
            ......

            // Perform predispatching before the IME.
            if (mView.dispatchKeyEventPreIme(event)) {
                finishInputEvent(q, true);
                return;
            }

            if (mLastWasImTarget) {
                InputMethodManager imm = InputMethodManager.peekInstance();
                if (imm != null) {
                    final int seq = event.getSequenceNumber();
                    ......
                    imm.dispatchKeyEvent(mView.getContext(), seq, event, mInputMethodCallback);
                    return;
                }
            }
        }

        // Not dispatching to IME, continue with post IME actions.
        deliverKeyEventPostIme(q);
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—InputMethodCallback.finishedEvent

```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    static final class InputMethodCallback implements InputMethodManager.FinishedEventCallback {
        private WeakReference<ViewRootImpl> mViewAncestor;

        public InputMethodCallback(ViewRootImpl viewAncestor) {
            mViewAncestor = new WeakReference<ViewRootImpl>(viewAncestor);
        }

        @Override
        public void finishedEvent(int seq, boolean handled) {
            final ViewRootImpl viewAncestor = mViewAncestor.get();
            if (viewAncestor != null) {
                viewAncestor.dispatchImeFinishedEvent(seq, handled);
            }
        }
    }

    ......
}
```

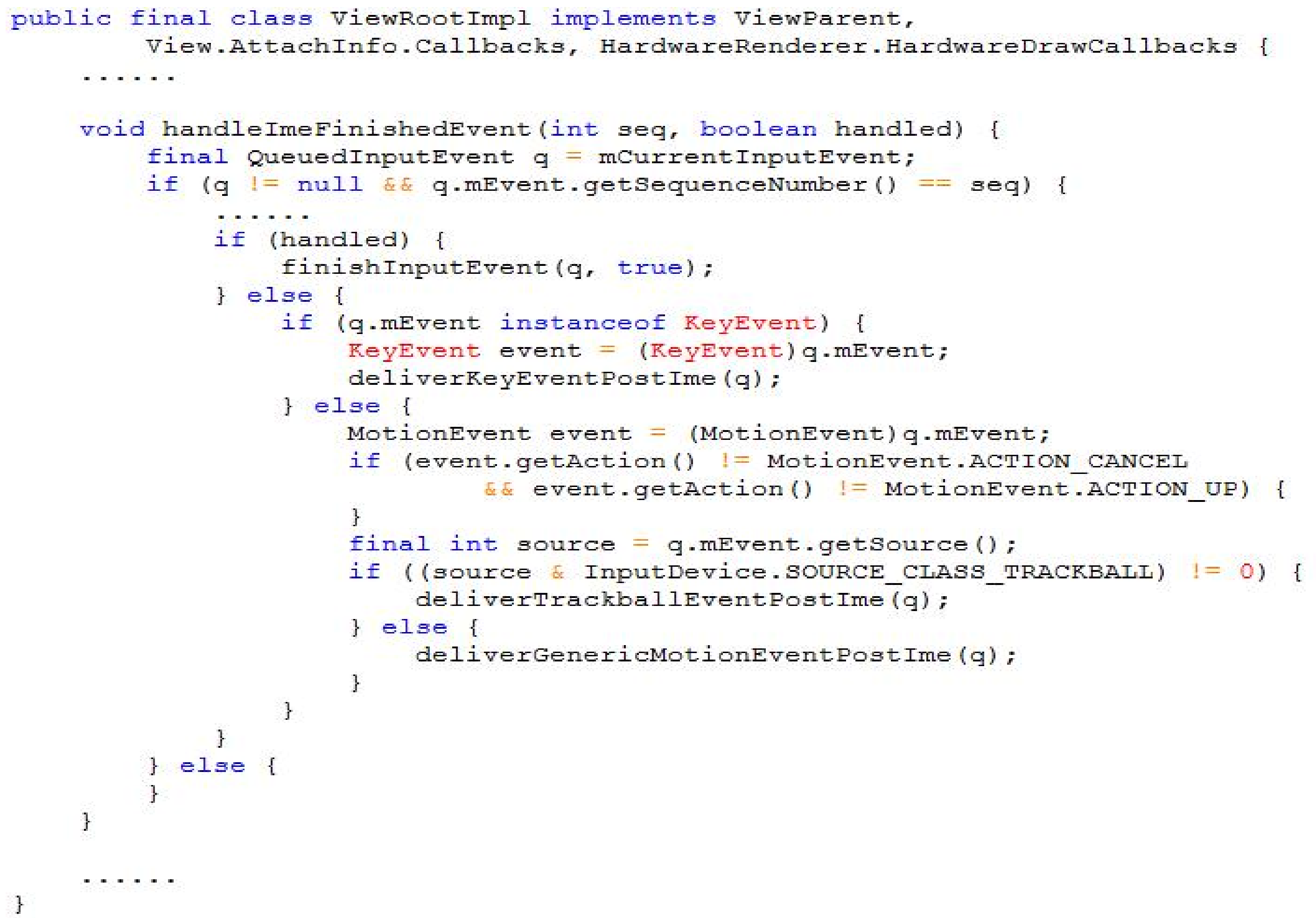
输入事件的分发过程(续)

- App获得键盘事件－ViewRootImpl.dispatchImeFinishedEvent

```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    void dispatchImeFinishedEvent(int seq, boolean handled) {
        Message msg = mHandler.obtainMessage(MSG_IME_FINISHED_EVENT);
        msg.arg1 = seq;
        msg.arg2 = handled ? 1 : 0;
        msg.setAsynchronous(true);
        mHandler.sendMessage(msg);
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl. handleImeFinishedEvent



```
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    void handleImeFinishedEvent(int seq, boolean handled) {
        final QueuedInputEvent q = mCurrentInputEvent;
        if (q != null && q.mEvent.getSequenceNumber() == seq) {
            ......
            if (handled) {
                finishInputEvent(q, true);
            } else {
                if (q.mEvent instanceof KeyEvent) {
                    KeyEvent event = (KeyEvent)q.mEvent;
                    deliverKeyEventPostIme(q);
                } else {
                    MotionEvent event = (MotionEvent)q.mEvent;
                    if (event.getAction() != MotionEvent.ACTION_CANCEL
                            && event.getAction() != MotionEvent.ACTION_UP) {
                    }
                    final int source = q.mEvent.getSource();
                    if ((source & InputDevice.SOURCE_CLASS_TRACKBALL) != 0) {
                        deliverTrackballEventPostIme(q);
                    } else {
                        deliverGenericMotionEventPostIme(q);
                    }
                }
            }
        } else {
        }
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—ViewRootImpl.deliverKeyEventPostIme

```java
public final class ViewRootImpl implements ViewParent,
        View.AttachInfo.Callbacks, HardwareRenderer.HardwareDrawCallbacks {
    ......

    private void deliverKeyEventPostIme(QueuedInputEvent q) {
        final KeyEvent event = (KeyEvent)q.mEvent;
        ......

        // Deliver the key to the view hierarchy.
        if (mView.dispatchKeyEvent(event)) {
            finishInputEvent(q, true);
            return;
        }
        ......

        // Key was unhandled.
        finishInputEvent(q, false);
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件—DecorView.dispatchKeyEvent

```
public class PhoneWindow extends Window implements MenuBuilder.Callback {
    ......

    private final class DecorView extends FrameLayout implements RootViewSurfaceTaker {
        ......

        @Override
        public boolean dispatchKeyEvent(KeyEvent event) {
            final int keyCode = event.getKeyCode();
            final int action = event.getAction();
            final boolean isDown = action == KeyEvent.ACTION_DOWN;
            ......

            if (!isDestroyed()) {
                final Callback cb = getCallback();
                final boolean handled = cb != null && mFeatureId < 0 ? cb.dispatchKeyEvent(event)
                        : super.dispatchKeyEvent(event);
                if (handled) {
                    return true;
                }
            }

            return isDown ? PhoneWindow.this.onKeyDown(mFeatureId, event.getKeyCode(), event)
                    : PhoneWindow.this.onKeyUp(mFeatureId, event.getKeyCode(), event);
        }

        ......
    }

    ......
}
```

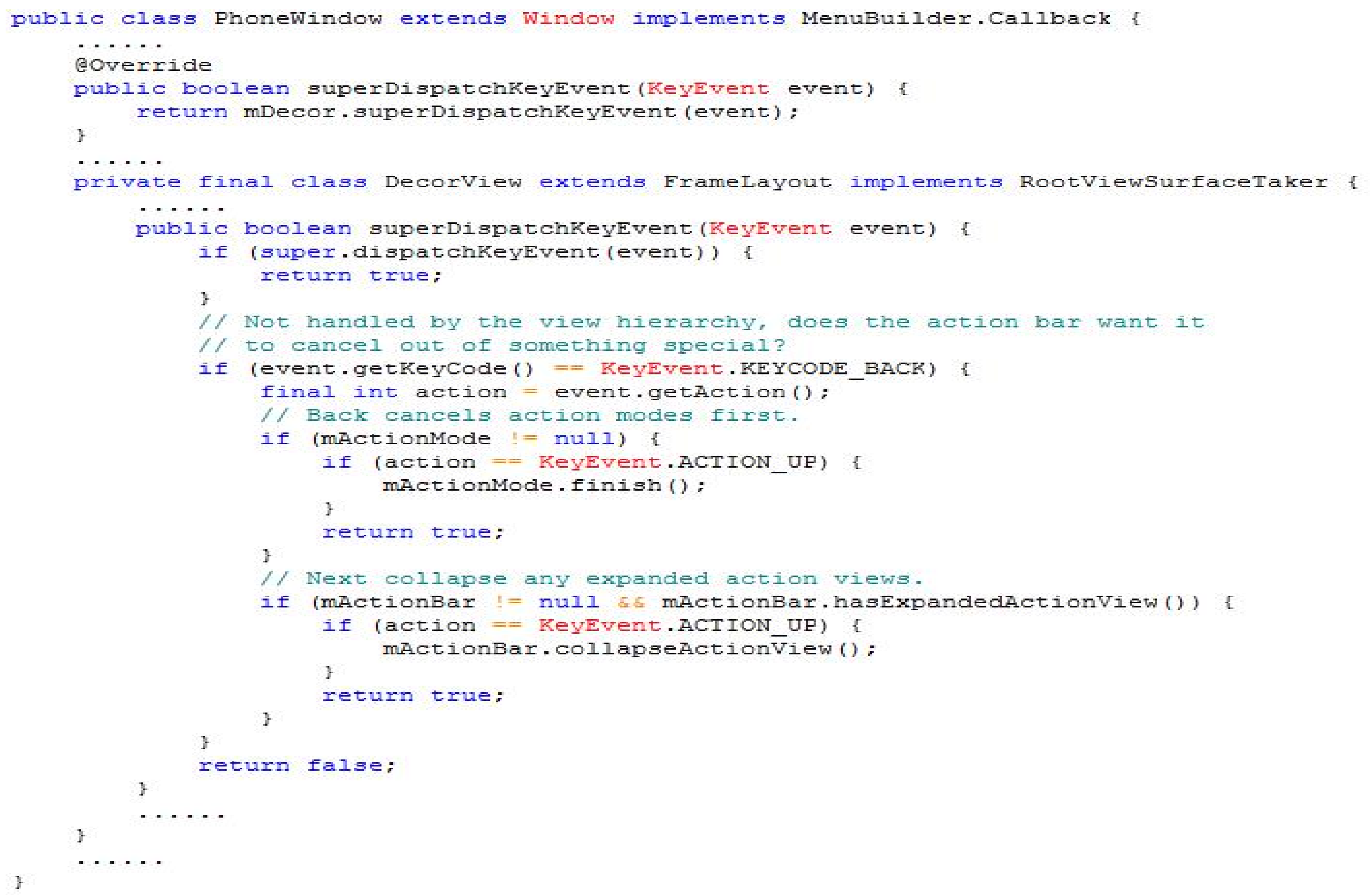
输入事件的分发过程(续)

- App获得键盘事件—Activity.dispatchKeyEvent

```
public class Activity extends ContextThemeWrapper
        implements LayoutInflater.Factory2,
        Window.Callback, KeyEvent.Callback,
        OnCreateContextMenuListener, ComponentCallbacks2 {
    ......

    public boolean dispatchKeyEvent(KeyEvent event) {
        onUserInteraction();
        Window win = getWindow();
        if (win.superDispatchKeyEvent(event)) {
            return true;
        }
        View decor = mDecor;
        if (decor == null) decor = win.getDecorView();
        return event.dispatch(this, decor != null
                ? decor.getKeyDispatcherState() : null, this);
    }

    ......
}
```

输入事件的分发过程(续)

- App获得键盘事件–PhoneWindow.superDispatchKeyEvent



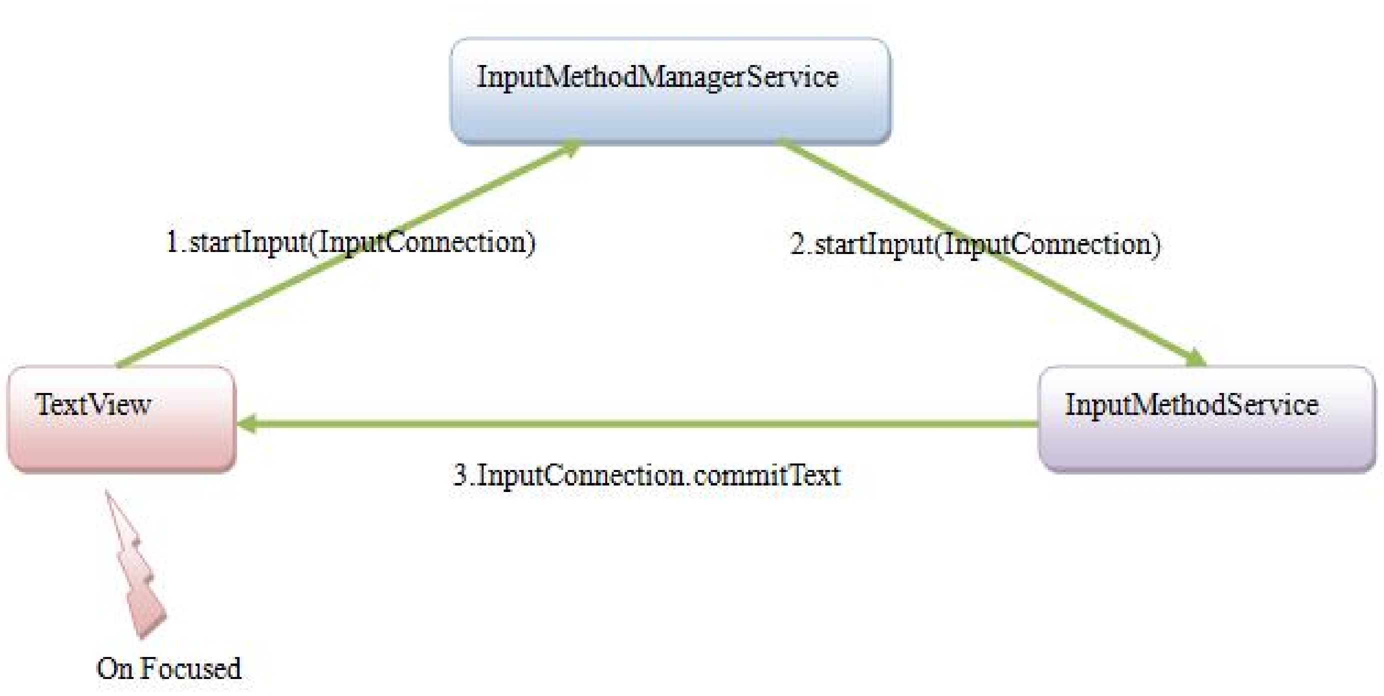
```
public class PhoneWindow extends Window implements MenuBuilder.Callback {
    ......
    @Override
    public boolean superDispatchKeyEvent(KeyEvent event) {
        return mDecor.superDispatchKeyEvent(event);
    }
    ......
    private final class DecorView extends FrameLayout implements RootViewSurfaceTaker {
        ......
        public boolean superDispatchKeyEvent(KeyEvent event) {
            if (super.dispatchKeyEvent(event)) {
                return true;
            }
            // Not handled by the view hierarchy, does the action bar want it
            // to cancel out of something special?
            if (event.getKeyCode() == KeyEvent.KEYCODE_BACK) {
                final int action = event.getAction();
                // Back cancels action modes first.
                if (mActionMode != null) {
                    if (action == KeyEvent.ACTION_UP) {
                        mActionMode.finish();
                    }
                    return true;
                }
                // Next collapse any expanded action views.
                if (mActionBar != null && mActionBar.hasExpandedActionView()) {
                    if (action == KeyEvent.ACTION_UP) {
                        mActionBar.collapseActionView();
                    }
                    return true;
                }
            }
            return false;
        }
        ......
    }
    ......
}
```

输入事件的分发过程(续)

- 在App中，依次获得键盘事件的顺序
  - View(Pre Input Method)
  - Input Method
  - View(Post Input Method)
  - Activity
  - Phone Window(处理MENU、BACK等按键)
- HOME按键被PhoneWindowManager拦截，直接切换至Home App

软键盘输入事件的分发过程

- TextView、输入法和输入法管理器的关系


InputMethodManagerService
1.startInput(InputConnection)
2.startInput(InputConnection)
TextView
InputMethodService
3.InputConnection.commitText
On Focused

软键盘输入事件的分发过程(续)

- 输入法通过InputConnection.commitText分发过来的字符被封装成一个类型为FLAG_DELIVER_POST_IME的KeyEvent
- 在ViewRootImpl中，类型为FLAG_DELIVER_POST_IME的KeyEvent不用经过输入法处理，而直接通过deliverKeyEventPostIme分发给View Hierarchy处理
- deliverKeyEventPostIme的处理过程与实体 键经过输入法处理后的过程是一样的
  - View
  - Activity
  - Phone Window

Q&A

Thank You