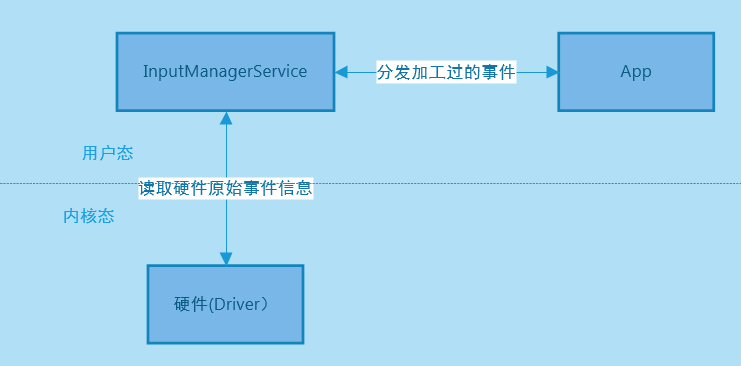
# Android事件系统

## **概述**

我们都知道事件的产生往往是始于硬件设备（比如键盘，触摸屏等输入设备），然后经过系统的封装与调度，最终才能够到达App来处理。接下来就分析一下在Android系统中事件分发的设计与实现。

我们都知道Android系统中广泛的采用C\S的模式对资源进行管理和提供服务。事件输入系统在此也并不例外。这里的服务方就是InputManagerService，客户端往往就是各个App。所以整个输入系统的框架就可以简化为下图。



从上图可以看出各个模块的职责情况。

硬件模块:产生原始的事件

InputMangerService：1、负责从硬件设备驱动中读取到输入的事件。2、将输入的事件进行加工（原始事件与硬件相关，为了给App层的事件保持统一接口，所以需要做一下转换操作）。3、将加工过的事件分发给App

App：负责处理输入事件的逻辑部分。

## InputMangerService

通过上面的描述，我们大致了解了InputManagerService的职责。接下来通过系统源代码来分析InputManagerService的整体设计架构与运行流程。

### 启动与初始化

InputManagerService作为系统用来管理输入事件的服务，它的启动在com.android.server.SystemServer中。

private void startOtherServices() {

Slog.i(TAG, "Input Manager");

inputManager = new InputManagerService(context);

Slog.i(TAG, "Window Manager");

wm = WindowManagerService.main(context, inputManager,

mFactoryTestMode != FactoryTest.FACTORY\_TEST\_LOW\_LEVEL,

!mFirstBoot, mOnlyCore);

ServiceManager.addService(Context.WINDOW\_SERVICE, wm);

ServiceManager.addService(Context.INPUT\_SERVICE, inputManager);

mActivityManagerService.setWindowManager(wm);

inputManager.setWindowManagerCallbacks(wm.getInputMonitor());

inputManager.start();

}

首先创建了InputManagerService对象，并将对象加入到ServiceManager里进行管理。同时将对象作为参数用于构建WindowManagerService对象，由此说明WindowManagerService服务需要使用到InputManagerService的相关功能，这个从稍后的代码分析中可以看到。最后调用了InputManagerService.start()方法，开启了InputManagerService服务。现在进入到InputManagerService的构造函数中。代码位于com.android.server.input.InputManagerService

public InputManagerService(Context context) {

this.mContext = context;

this.mHandler = new InputManagerHandler(DisplayThread.get().getLooper());

mUseDevInputEventForAudioJack =

context.getResources().getBoolean(R.bool.config\_useDevInputEventForAudioJack);

Slog.i(TAG, "Initializing input manager, mUseDevInputEventForAudioJack="

+ mUseDevInputEventForAudioJack);

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

LocalServices.addService(InputManagerInternal.class, new LocalService());

}

这里主要调用了nativeInit()方法，进行初始化。在一些对效率要求很高的地方，Android系统都会采用C/C++进行处理。这里追踪到c++代码查看。代码位于

static jlong nativeInit(JNIEnv\* env, jclass /\* clazz \*/,

jobject serviceObj, jobject contextObj, jobject messageQueueObj) {

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

if (messageQueue == NULL) {

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

return 0;

}

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

messageQueue->getLooper());

im->incStrong(0);

return reinterpret\_cast<jlong>(im);

}

这里根据传递进来的消息队列的Looper（主要是用来对消息进行异步处理）对象构建了NativeInputManager对象。并把该对象的指针传递到了java层，所以java层的InputManagerService对象的mPtr实际保存的就是c++层的指向NativeInputManger对象的指针。接下来看一下NativeInputManager的构造方法。

NativeInputManager::NativeInputManager(jobject contextObj,

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

mLooper(looper), mInteractive(true) {

JNIEnv\* env = jniEnv();

mContextObj = env->NewGlobalRef(contextObj);

mServiceObj = env->NewGlobalRef(serviceObj);

{

AutoMutex \_l(mLock);

mLocked.systemUiVisibility = ASYSTEM\_UI\_VISIBILITY\_STATUS\_BAR\_VISIBLE;

mLocked.pointerSpeed = 0;

mLocked.pointerGesturesEnabled = true;

mLocked.showTouches = false;

}

mInteractive = true;

sp<EventHub> eventHub = new EventHub();

mInputManager = new InputManager(eventHub, this, this);

}

从构造方法中，我们可以看到主要是构建了EventHub对象，并且使用EventHub对象构建了InputManger对象。这里就可以看出NativeInputManager对象只是对InputManger对象的简单包装，方便提供给java层使用。Android中大量的使用了这种包装的方式，来连接java层和c++层的对象关系。

