Android 耳机系统综述

本文为本人随笔所写，欢迎转载。由于个人的见识和能力有限，不能面面俱到，也可能存在谬误，敬请各位指出，本人的邮箱是hyouyan@126.com，博客是http://hyouyan.cublog.cn

在 Android 中其实并没有耳机系统这个称呼，只是我为了方便解释而加的。在整个 android 系统中，跟耳机相关的部分有：

- Linux 驱动：主要完成耳机的插入的检测，hook 键的检测，其中 hook 键包括长按和短按。
- 在 frameworks 中的耳机的观察的文件（HeadsetObserver.java），这个文件主要是检测耳机是否插入和名字，并把相关的内容通过 Intent 广播出去。
- 跟音频相关，改变音频输出的路径（这边涉及到播放音乐和电话部分）。
- 跟事件的处理相关，这部分主要体现 hook 的功能，主要是接听电话，挂断电话等。事件的处理又分为 linux 的事件处理和 android 上的事件处理。

我将分块叙述，由于各种原因，我在这不便把源代码公布，如果你需要的我的帮助，可以发邮件给我，也可以在我 blog 上留言，谢谢!

Linux驱动

首先要定义一个 switch_dev（struct switch_dev sdev;）并把它初始化，如（sdev.name = ……）；然后注册一个 switch device：

```
        ret = switch_dev_register(&switch_data->sdev);
        if (ret < 0)
        {
                goto err_switch_dev_register;
        }
```

switch_dev_register 这个函数在 switch_class.c 中实现

```
int switch_dev_register(struct switch_dev *sdev)
{
        int ret;
        if (!switch_class) {
                ret = create_switch_class();
                if (ret < 0)
                        return ret;
        }

        sdev->index = atomic_inc_return(&device_count);
        sdev->dev = device_create(switch_class, NULL,
                MKDEV(0, sdev->index), NULL, sdev->name);
        if (IS_ERR(sdev->dev))
                return PTR_ERR(sdev->dev);
        ret = device_create_file(sdev->dev, &dev_attr_state);
        if (ret < 0)
                goto err_create_file_1;
        ret = device_create_file(sdev->dev, &dev_attr_name);
        if (ret < 0)
                goto err_create_file_2;

        dev_set_drvdata(sdev->dev, sdev);
        sdev->state = 0;
        return 0;

err_create_file_2:
        device_remove_file(sdev->dev, &dev_attr_state);
err_create_file_1:
        device_destroy(switch_class, MKDEV(0, sdev->index));
        printk(KERN_ERR "switch: Failed to register driver %s\n", sdev->name);

        return ret;
```

}

这个函数中主要是以下几个函数

- create_switch_class()
- device_create(switch_class, NULL, MKDEV(0, sdev->index), NULL, sdev->name);
- device_create_file(sdev->dev, &dev_attr_state);
- device_create_file(sdev->dev, &dev_attr_name);

经过以上函数后将会生成路径和被用户空间访问的节点
"/sys/class/switch/h2w/name";
"/sys/class/switch/h2w/state";
这两个供用户空间访问

在这个函数中要注意到
static DEVICE_ATTR(state, S_IRUGO | S_IWUSR, state_show, NULL);
static DEVICE_ATTR(name, S_IRUGO | S_IWUSR, name_show, NULL);
这两项中用于设置节点 state 和 name 的属性
DEVICE_ATTR 有四个参数，分别为名称、权限位、读函数、写函数
有此可以知道 state 和 name，虽然有读写权限，但都只有读函数，没有写函数。
其中 state 对 headsetobserver.java 区分有无 mic 和耳机是否插入起作用

```
static ssize_t state_show(struct device *dev, struct device_attribute *attr,
        char *buf)
{
    struct switch_dev *sdev = (struct switch_dev *)
        dev_get_drvdata(dev);

    if (sdev->print_state) {   //如果用户有定义 print_state 函数，将调用用户定义的
        int ret = sdev->print_state(sdev, buf);
        if (ret >= 0)
            return ret;
    }
    return sprintf(buf, "%d\n", sdev->state);//把 sdev->state 以%d 的格式装如 buf 中
}
```

在这个函数得注意：如果你想你的 frameworks 能区别出有没有 mic，并且你用的是 switch_gpio.c 这个文件的话，你需要把 switch_gpio.c 中的 sdev->print_state 的定义去掉。我就在这卡了半天的时间。State 原先出来一直是 1，后来才发现原来是自己定义了 sdev->print_state 并只返回 0 和 1，没有其他值。

现重新回到 driver，接下来时 input 子系统的内容
input_allocate_device();分配内存给新的输入设备
接下去初始化 input_dev 这个结构体，给输入设备命名 dev->name，
设置 input 支持的键值 input_set_capability，如：

```
    input_set_capability(ipdev, EV_KEY, KEY_MEDIA);
    input_set_capability(ipdev, EV_SW, SW_HEADPHONE_INSERT);
    input_set_capability(ipdev, EV_KEY, KEY_END);
```

注册 input 设备 input_register_device(ipdev);

在驱动中还涉及到工作队列等问题，就请各位自己去看一下吧。

接下来是对于中断的处理，这个中断方式我是从 HTC 的驱动中学的，有点巧妙，想到了叶就不算巧妙了，呵呵。

先申请为高电平中断，我的板子是插入耳机检测脚我高电平，在进入中断后再申请为低电平中断，这个相对于上升和下降有个好处——当设置为上升或下降沿触发中断时，开机之前插入耳机，当开机后，将识别不到耳机。而当设置为电平触发可以解决这个问题。

我的观点是在耳机在插槽内时，检测引脚直接被拉倒插入耳机稳定后的电平，而不会产生上升和下降沿。中断申请的代码如下：

```
request_irq(gpio_to_irq(18),gpio_irq_handler,IRQF_TRIGGER_HIGH,pdev->name,switch_data);
```

中断处理的代码如下：

```
set_irq_type(gpio_to_irq(18), gpio_get_value(18) ? IRQF_TRIGGER_LOW : IRQF_TRIGGER_HIGH);
```

由上可以看到 C 语言的问号表达式的好处了吧，呵呵。C 语言博大精深！还有很多精髓的问题，以后用了，慢慢体会，如果你觉得你的 C 非常好了，呵呵，找一个 C 语言的笔试题来做做，哈哈，你真会发现又学到一堆的东西。呵呵。继续我们的驱动。

接下来是有无 mic 的判断和设置 state 的值了，有 HeadsetObserver.java 这个文件中可以得出 state 的值：

- 有 mic：state 等于 1
- 没有 mic：state 等于 2

扯点题外，我原先以为在"/sys/class/switch/h2w/state";下的 state 只有 0 和 1，我再问了我的一些同事，他们也跟我说是 bool 类型。但我看到 headsetobserver.java 中又有 1 和 2，后面觉得有点可疑。再看源代码之前，真的不想看源代码，看了源代码后，发现源代码真好。哈哈。通过一步步跟，后面发现时可以大于 1 的，呵呵。

这个将要用到 switch_get_state(&data->sdev)这个函数，它也是在 switch_class.c 中实现的。

```
void switch_set_state(struct switch_dev *sdev, int state)
{
    char name_buf[120];
    char state_buf[120];
    char *prop_buf;
    char *envp[3];
    int env_offset = 0;
    int length;

    if (sdev->state != state) {
        sdev->state = state;    //实现你要设置的值
        prop_buf = (char *)get_zeroed_page(GFP_KERNEL);
        if (prop_buf) {
            length = name_show(sdev->dev, NULL, prop_buf); //给 HeadsetObserver.java 读取
名字
            if (length > 0) {
                if (prop_buf[length - 1] == '\n')
                    prop_buf[length - 1] = 0;
                snprintf(name_buf, sizeof(name_buf),
                    "SWITCH_NAME=%s", prop_buf);
                envp[env_offset++] = name_buf;
            }
            length = state_show(sdev->dev, NULL, prop_buf); //给 HeadsetObserver.java 读取
```
读取状态，这个函数我们在前面分析过了，这个函数比较重要，关系到区分有无 mic。

```
            if (length > 0) {
                if (prop_buf[length - 1] == '\n')
                    prop_buf[length - 1] = 0;
                snprintf(state_buf, sizeof(state_buf),
                    "SWITCH_STATE=%s", prop_buf);
                envp[env_offset++] = state_buf;
            }
            envp[env_offset] = NULL;
            kobject_uevent_env(&sdev->dev->kobj, KOBJ_CHANGE, envp);
            free_page((unsigned long)prop_buf);
        } else {
            printk(KERN_ERR "out of memory in switch_set_state\n");
            kobject_uevent(&sdev->dev->kobj, KOBJ_CHANGE);
        }
    }
}
EXPORT_SYMBOL_GPL(switch_set_state); //供外部所使用。
```
由于 hook 键和检测 mic 的有关联，故如果有 mic 则要申请 hook 的中断。
具体 mic 的检测可以参考我的 blog 中转载别人的的一篇文章，链接地址如下

http://blog.chinaunix.net/u3/106866/showart_2273977.html

接下来是 HOOK 键功能的处理了，在 google 论坛里有些说实现 hook 键接听和挂断电话的问题。Hook 键只有一个，要实现两个功能就得要用时间来区分了，

- 短按：代表接听。
- 长按：代表拒接。

这样两种功能就实现了，呵呵。对于长短的检测最好用纳秒，用秒的准确性比较低。存在误判性比较高，可以利用把时间转换成纳秒来计算，我用如下实现检测时间的长短：

```
        do_gettimeofday(&time);
        timens=timeval_to_ns(&time);
        while(gpio_get_value(123)==0){};
        do_gettimeofday(&time);
        (timeval_to_ns(&time)-timens)由这个式子可以得到比较准确的时间。
```
在利用这个时间，你确定一个判断长短的依据，就可以了如：
```
if(    (timeval_to_ns(&time)-timens)<1000000000l)
{//短按
    if(    (timeval_to_ns(&time)-timens) > 50000000)//为了取出噪音，而设置一定的最低值
    {
```

```
                        input_report_key(switch_data->ipdev,KEY_MEDIA,1);
                        input_sync(switch_data->ipdev);
                        msleep(100);
                        input_report_key(switch_data->ipdev,KEY_MEDIA,0);
                        input_sync(switch_data->ipdev);
        }
}
else
{//长按
                        input_report_key(switch_data->ipdev,KEY_END,1);
                        input_sync(switch_data->ipdev);
                        msleep(100);
                        input_report_key(switch_data->ipdev,KEY_END,0);
                        input_sync(switch_data->ipdev);
}
```

在这传上去的是 KEY_MEDIA 和 KEY_END，然而这两个键值又如何对应上层的接听和挂断呢？其中 KEY_END 在 frameworks 层已经映射成挂机键了，然而 KEY_MEDIA 却要你自己映射成 HEADSETHOOK 键，在你 android 的根目录下在
sdk\emulator\keymaps 下 qwerty.kl 中加入

```
key 226     HEADSETHOOK          WAKE
```

在这说明一下有些地方说是
./development/emulator/keymaps/qwerty.kl
我的是 android2.1 的版本，我在我的版本下没发现 qwerty.kl。我想这可能是版本的差异吧。

到此，linux 驱动层算是大体结束了。

Frameworks层耳机相关

启动服务在 systemserver.java 中

```
public class SystemServer
{
        public static void main(String[] args)
        {
                ……
                init1(args);
                ……
        }
        public static final void init2() {
                Log.i(TAG, "Entered the Android system server!");
                Thread thr = new ServerThread();
                thr.setName("android.server.ServerThread");
                thr.start();
        }

}
```

init1 将会调用到 android_server_SystemServer_init1.cpp

```
extern "C" int system_init();
static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
        system_init();
}
```

由上可以得到，将调用到 System_init.cpp

```
extern "C" status_t system_init()
{
……
runtime->callStatic("com/android/server/SystemServer", "init2");
……
}
```

由上可以看出，此时将调回到 systemserver.java，并且调用

```
public static final void init2() {
Log.i(TAG, "Entered the Android system server!");
                Thread thr = new ServerThread();
                thr.setName("android.server.ServerThread");
                thr.start();
}
```

新建线程

```
class ServerThread extends Thread
{
        ……
        public void run()
        {
                ……
                try {
                        Log.i(TAG, "Headset Observer");
                        // Listen for wired headset changes
                        headset = new HeadsetObserver(context); // new a thread to observer headset status
                      } catch (Throwable e) {
                                  Log.e(TAG, "Failure starting HeadsetObserver", e);
                 }
        }
}
```

开始服务：HeadsetObserver.java

```
class HeadsetObserver extends UeventObserver
{
        ……
        public HeadsetObserver(Context context)
        {
                ……
                startObserving(HEADSET_UEVENT_MATCH);

                init();    // set initial status
        }
}
```

运行以上程序后会一直监测 HEADSET_UEVENT_MATCH 路径的事件，

HEADSET_UEVENT_MATCH = "DEVPATH=/devices/virtual/switch/h2w";

如果有事件的变化，则会调用

```
public void onUEvent(UEventObserver.UEvent event)
{
        if (LOG) Log.v(TAG, "Headset UEVENT: " + event.toString());

        try{
        update(event.get("SWITCH_NAME"), Integer.parseInt(event.get("SWITCH_STATE")));
        } catch (NumberFormatException e) {
                        Log.e(TAG, "Could not parse switch state from event " + event);
        }
}

private synchronized final void update(String newName, int newState)
```

```
{
    ……
    mHandler.sendMessageDelayed(mHandler.obtainMessage(0,mHeadsetState,
                                                       mPrevHeadsetState,
                                                       mHeadsetName),//send message
}
```

一下一段没有考证：但我猜应该是由于这个原因会调用到 sendIntents

```
private final Handler mHandler = new Handler()
{
  @Override
  public void handleMessage(Message msg)
  {
    sendIntents(msg.arg1, msg.arg2, (String)msg.obj);
    mWakeLock.release();
  }
};
```

如果有新的事件，将会调用

rivate synchronized final void sendIntents

再调用到

private final void sendIntent

此处填充 Intent。

```
private final void sendIntent(int headset, int headsetState, int prevHeadsetState, String
headsetName)
{
    ……
    Intent intent = new Intent(Intent.ACTION_HEADSET_PLUG);
    intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
    ……
    if ((headset & HEADSETS_WITH_MIC) != 0)
    {
        microphone = 1;// 是否有 mic
    }
    if ((headsetState & headset) != 0)
    {
       state = 1;
    }
    intent.putExtra("state", state);
    intent.putExtra("name", headsetName);
    intent.putExtra("microphone", microphone);
    ……
    ActivityManagerNative.broadcastStickyIntent(intent, null);   //broadcast intent
}
```

跟音频相关

此时在 AudioService.java 中将接收 Broadcast

```
private class AudioServiceBroadcastReceiver extends BroadcastReceiver
{
        public void onReceive(Context context, Intent intent)
        {
                ……
                else if (action.equals(Intent.ACTION_HEADSET_PLUG))
                {
                        ……
                        //if you first insert headset, will implement fellow code
                        AudioSystem.setDeviceConnectionState(AudioSystem.DEVICE_OUT_WIRED_H
                EADSET,AudioSystem.DEVICE_STATE_AVAILABLE,"");
                }
        }
}
```

调用 setDeviceConnectionState 由在 android_media_AudioSystem.cpp 中可以得到

```
static JNINativeMethod gMethods[] = {

          "setDeviceConnectionState","(IILjava/lang/String;)I",(void*)android_media_AudioSystem_
setDeviceConnectionState},

};
```

所以将调用到 android_media_AudioSystem_setDeviceConnectionState

```
android_media_AudioSystem_setDeviceConnectionState(JNIEnv *env, jobject thiz, jint device,
jint state, jstring device_address)
{
        ……
        Int status =check_AudioSystem_Command
(AudioSystem::setDeviceConnectionState(static_cast <AudioSystem::audio_devices>(device),
static_cast <AudioSystem::device_connection_state>(state),c_address));
        ……
}
```

由上段程序可以看出，将会调用到 AudioSystem.cpp 中的 setDeviceConnectionState

```
status_t AudioSystem::setDeviceConnectionState(audio_devices device,
                                               device_connection_state state,
                                               const char *device_address)
{
        const sp<IAudioPolicyService>& aps = AudioSystem::get_audio_policy_service();
        if (aps == 0) return PERMISSION_DENIED;

        return aps->setDeviceConnectionState(device, state, device_address);
}
```
get_audio_policy_service();这个函数具体做什么我现在还没弄清楚。

一下这边我没找到具体的联系，我通过打印得知会调用到 AudioPolicyManager.cpp 的 setDeviceConnectionState 函数，以下的函数很重要，关系到设置输出路径等

```
status_t AudioPolicyManager::setDeviceConnectionState(AudioSystem::audio_devices device,
                                                      AudioSystem::device_connection_state state,
                                                              const char *device_address)
{
        ……
        // handle output devices
        if (AudioSystem::isOutputDevice(device))
        {
               switch (state)
               {
                      case AudioSystem::DEVICE_STATE_AVAILABLE:
                      ……
                  if (AudioSystem::isBluetoothScoDevice(device))
                  {
                      ……
                  }
               else if (device == AudioSystem::DEVICE_OUT_WIRED_HEADSET ||
                            device == AudioSystem::DEVICE_OUT_WIRED_HEADPHONE)
                  {
                        if (getDeviceForStrategy(STRATEGY_PHONE) == device &&
                            (mPhoneState == AudioSystem::MODE_IN_CALL ||

  mOutputs.valueFor(mHardwareOutput)->isUsedByStrategy(STRATEGY_PHONE)))
                        {
                                newDevice = device;
                        }
                        else if ((getDeviceForStrategy(STRATEGY_SONIFICATION) & device) &&

mOutputs.valueFor(mHardwareOutput)->isUsedByStrategy(STRATEGY_SONIFICATION))
                        {
                          newDevice = getDeviceForStrategy(STRATEGY_SONIFICATION);
                          }
                        else if ((getDeviceForStrategy(STRATEGY_MEDIA) == device) &&

mOutputs.valueFor(mHardwareOutput)->isUsedByStrategy(STRATEGY_MEDIA))
                        {
                            newDevice = device;
```

```
                }
                else if (getDeviceForStrategy(STRATEGY_DTMF) == device &&

mOutputs.valueFor(mHardwareOutput)->isUsedByStrategy(STRATEGY_DTMF))
                {
                    newDevice = device;
                }
            }
        }

}
```

以上两个个主要函数是：
getDeviceForStrategy，
mOutputs.valueFor(mHardwareOutput)->isUsedByStrategy(STRATEGY_PHONE))
存在以下疑问：
1 ：getDeviceForStrategy 的作用是什么？

```
bool AudioPolicyManager::AudioOutputDescriptor::isUsedByStrategy(routing_strategy strategy)
{
    for (int i = 0; i < (int)AudioSystem::NUM_STREAM_TYPES; i++)
    {
        if (AudioPolicyManager::getStrategy((AudioSystem::stream_type)i) == strategy &&
            isUsedByStream((AudioSystem::stream_type)i))
        {
            return true;
        }
    }
    return false;
}
```
这个函数很重要主要是为以后设置为耳机，蓝牙这类的输出.
以上函数会调用到
bool isUsedByStream(AudioSystem::stream_type stream) { return mRefCount[stream] > 0 ? true : false; }

这个函数也很重要.这个函数用到 mRefCount 这个数组，
这个函数在 void AudioPolicyManager::AudioOutputDescriptor::changeRefCount 中改变
然而 changeRefCount 将会在 startOutput 调用。
具体什么时候改变 mRefCount 这个数组，现不是非常的清楚。

上面的走完后将设置输出
setOutputDevice(mHardwareOutput, newDevice);
其中 newDevice 决定什么样的输出。

跟事件的处理相关

文件流程流程

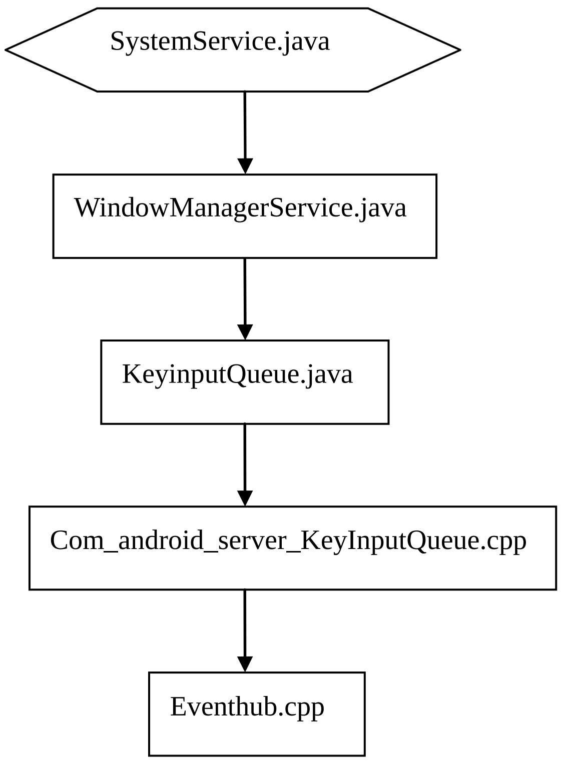
SystemService.java
WindowManagerService.java
KeyinputQueue.java
Com_android_server_KeyInputQueue.cpp
Eventhub.cpp

Fig 1

从**SystemService.java**中启动服务：

```
public class SystemServer
{
        ……
        native public static void init1(String[] args);

        public static void main(String[] args)
        {
                ……
                init1(args);
        }
        public static final void init2()
        {
            Log.i(TAG, "Entered the Android system server!");
            Thread thr = new ServerThread();
            thr.setName("android.server.ServerThread");
            thr.start();
        }
}
```

运行 init1(args); 在 com_android_server_SystemServer.cpp 中有

```
static JNINativeMethod gMethods[] = {
      /* name, signature, funcPtr */
      { "init1", "([Ljava/lang/String;)V", (void*) android_server_SystemServer_init1 },
};
```

而又有如下：

```
extern "C" int system_init();

static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
     system_init();
}
```

system_init()将调用在 System_init.cpp 中

```
extern "C" status_t system_init()
{
        ……
        runtime->callStatic("com/android/server/SystemServer",    "init2");// 这 句 后 将 跳 会
SystemService.java 中的 init2。
        ……
}
```

也即是如下代码

```
public static final void init2()
{
        Log.i(TAG, "Entered the Android system server!");
        Thread thr = new ServerThread(); //建立一个 service 的线程
        thr.setName("android.server.ServerThread");
        thr.start();
}
```

服务线程：

```
class ServerThread extends Thread
```

```
{
        ……
        public void run()
        {
                ……
                Log.i(TAG, "Window Manager");
                wm = WindowManagerService.main(context, power,
                                factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL);
                ServiceManager.addService(Context.WINDOW_SERVICE, wm);
                ((ActivityManagerService)ServiceManager.getService("activity"))
                                        .setWindowManager(wm);
                ……
        }
        ……
}
```

将跳转到 WindowManagerService.java 的 main 中

```
public static WindowManagerService main(Context context,
        PowerManagerService pm, boolean haveInputMethods)
{
        WMThread thr = new WMThread(context, pm, haveInputMethods);//建立线程
        thr.start();
        synchronized (thr)
        {
                while (thr.mService == null)
                {
                      try {
                                thr.wait();
                            } catch (InterruptedException e) {}
                  }
        }
                return thr.mService;
}
static class WMThread extends Thread
{
        ……
        public void run()
        {
                ……
                WindowManagerService s = new WindowManagerService(mContext, mPM,
                                        mHaveInputMethods); //新建一个 WindowManagerService 的线程
                ……
        }
}
```

即将跳到 WindowManagerService 的构造函数

```
private WindowManagerService(Context context, PowerManagerService pm,
                    boolean haveInputMethods)
{
      ……
      mQueue = new KeyQ();
      ……
}
```

而

```
private class KeyQ extends KeyInputQueue
```

在 KeyInputQueue 的构造函数中

```
KeyInputQueue(Context context, HapticFeedbackCallback    hapticFeedbackCallback)
{
      ……
      mThread.start();
}
```

而

```
Thread mThread = new Thread("InputDeviceReader")
{
      ……
      readEvent(ev);
      ……
}
```

private static native boolean readEvent(RawInputEvent outEvent);由这句可知 readEvent 在 JNI 层

在 com_android_server_KeyInputQueue.cpp 中有

```
static JNINativeMethod gInputMethods[] = {
      /* name, signature, funcPtr */
      { "readEvent",            "(Landroid/view/RawInputEvent;)Z",
                    (void*) android_server_KeyInputQueue_readEvent },
      ……
}
```

由以上可知在 KeyInputQueue.java 中调的 readEvent 将调用到 com_android_server_KeyInputQueue.cpp 的 android_server_KeyInputQueue_readEvent。

```
static jboolean
android_server_KeyInputQueue_readEvent(JNIEnv* env, jobject clazz, jobject event)
{
      ……
      if (hub == NULL)
      {
            hub = new EventHub;
            gHub = hub;
      }
```

```
    ……
    bool res = hub->getEvent(&deviceId, &type, &scancode, &keycode,
                    &flags, &value, &when);
    ……
}
```
hub->getEvent 将调用 EventHub.cpp 的 getEvent 函数
```
bool EventHub::getEvent(int32_t* outDeviceId, int32_t* outType,
        int32_t* outScancode, int32_t* outKeycode, uint32_t *outFlags,
        int32_t* outValue, nsecs_t* outWhen)
{
    ……
    if (!mOpened)
    {
        mError = openPlatformInput() ? NO_ERROR : UNKNOWN_ERROR;
        mOpened = true;
    }

}
```

openPlatformInput()将扫描/dev/input 下的所有 event 并打开它
```
/*
 * Open the platform-specific input device.
 */
bool EventHub::openPlatformInput(void)
{
    ……
    res = scan_dir(device_path);//其中 static const char *device_path = "/dev/input";
    ……
}
```

而
```
int EventHub::scan_dir(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++ = '/';
    //扫描/dev/input 下的所有 event 并打开它
    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);
        open_device(devname);//打开 event 设备

    }
    closedir(dir);
    return 0;
}

int EventHub::open_device(const char *deviceName)
{
    ……
    fd = open(deviceName, O_RDWR);
    ……
    if ((device->classes&CLASS_KEYBOARD) != 0)
    {
        char tmpfn[sizeof(name)];
        char keylayoutFilename[300];

        // a more descriptive name
        device->name = name;

        // replace all the spaces with underscores
        strcpy(tmpfn, name);
        for (char *p = strchr(tmpfn, ' '); p && *p; p = strchr(tmpfn, ' '))
                *p = '_';

        // find the .kl file we need for this device
        const char* root = getenv("ANDROID_ROOT");
        snprintf(keylayoutFilename, sizeof(keylayoutFilename),
                     "%s/usr/keylayout/%s.kl", root, tmpfn);
        bool defaultKeymap = false;
        if (access(keylayoutFilename, R_OK))
        {
            snprintf(keylayoutFilename, sizeof(keylayoutFilename),
                         "%s/usr/keylayout/%s", root, "qwerty.kl");
            defaultKeymap = true;
            }
            device->layoutMap->load(keylayoutFilename);
}
```

如果上面的操作都成功则把所有设备都打开了，根据注册的 input 设备的名字查找对应的.kl 文件，如果有该设备就用该.kl 把扫描码映射键码。文件现回到 EventHub::getEvent。

```
        release_wake_lock(WAKE_LOCK_ID);

        pollres = poll(mFDs, mFDCount, -1);

        acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_ID);
```

在这边 poll，如果没有新事件将在这等待，如果有则开始下面的读事件

```
        res = read(mFDs[i].fd, &iev, sizeof(iev));
```

到此整个从上面开始的读过程结束。

现在回到

```
Thread mThread = new Thread("InputDeviceReader")
{
        ……
        readEvent(ev);
        ……
        else
        {
              send = preprocessEvent(di, ev);
        }
}
```

由这个 abstract boolean preprocessEvent(InputDevice device, RawInputEvent event);可以看出上面调用的 preprocessEvent 将调到 windowmanagerservice.java 中的

boolean preprocessEvent(InputDevice device, RawInputEvent event)

```
boolean preprocessEvent(InputDevice device, RawInputEvent event)
{

        if (mPolicy.preprocessInputEventTq(event))
        {
              return true;
        }

        switch (event.type)
        {
              case RawInputEvent.EV_KEY:
              {
              ……
              if ((actions & WindowManagerPolicy.ACTION_PASS_TO_USER) != 0) //这段代码不
是很清楚，做什么用的
              {
                  if (event.value != 0 && mPolicy.isAppSwitchKeyTqTiLwLi(event.keycode))
                  {
                      filterQueue(this);
                      mKeyWaiter.appSwitchComing();
                  }
                  return true;
              }
              else
              {
                  return false;
              }
```

往事件队列里放入事件

在 WindowManagerService.java 的构造函数中又有

mInputThread = new InputDispatcherThread();

InputDispatcherThread 线程实际上从 KeyQ 的事件队列中读取按键事件

mInputThread.start();

又有如下

```
    private final class InputDispatcherThread extends Thread {
        // Time to wait when there is nothing to do: 9999 seconds.
        static final int LONG_WAIT=9999*1000;
        public InputDispatcherThread() {
            super("InputDispatcher");
        }
       @Override
        public void run() {
            while (true) {
                try {
                    process();
                } catch (Exception e) {
                    Log.e(TAG, "Exception in input dispatcher", e);
                }
            }
        }
private void process()
{
……
while (true)
{

    // Retrieve next event, waiting only as long as the next
```

```
        // repeat timeout.    If the configuration has changed, then
        // don't wait at all -- we'll report the change as soon as
        // we have processed all events.
        QueuedEvent ev = mQueue.getEvent(
                                (int)((!configChanged && curTime < nextKeyTime)
                                        ? (nextKeyTime-curTime) : 0));
……
switch (ev.classType)
{
        case RawInputEvent.CLASS_KEYBOARD:
        if (ke.isDown())
        {
                lastKey = ke;
                downTime = curTime;
                keyRepeatCount = 0;
                lastKeyTime = curTime;
                nextKeyTime = lastKeyTime+ ViewConfiguration.getLongPressTimeout();
                if (DEBUG_INPUT) Log.v(TAG, "Received key down: first repeat @ "
                                                                + nextKeyTime);
      }
      else
      {
                lastKey = null;
                downTime = 0;
                // Arbitrary long timeout.
                lastKeyTime = curTime;
                nextKeyTime = curTime + LONG_WAIT;
                if (DEBUG_INPUT) Log.v(TAG, "Received key up: ignore repeat @ "
                                                                + nextKeyTime);
      }
      dispatchKey((KeyEvent)ev.event, 0, 0);    //发布事件
      mQueue.recycleEvent(ev);
      break;
      ……
}


        /**
          * @return Returns true if event was dispatched, false if it was dropped for any reason
          */
        private int dispatchKey(KeyEvent event, int pid, int uid) {
                if (DEBUG_INPUT) Log.v(TAG, "Dispatch key: " + event);

                Object focusObj = mKeyWaiter.waitForNextEventTarget(event, null,
                                null, false, false, pid, uid);
                if (focusObj == null) {
                        Log.w(TAG, "No focus window, dropping: " + event);
                        return INJECT_FAILED;
                }
                if (focusObj == mKeyWaiter.CONSUMED_EVENT_TOKEN) {
                        return INJECT_SUCCEEDED;
                }

                // Okay we have finished waiting for the last event to be processed.
                // First off, if this is a repeat event, check to see if there is
                // a corresponding up event in the queue.    If there is, we will
                // just drop the repeat, because it makes no sense to repeat after
                // the user has released a key.    (This is especially important for
                // long presses.)
                if (event.getRepeatCount() > 0 && mQueue.hasKeyUpEvent(event)) {
                        return INJECT_SUCCEEDED;
                }

                WindowState focus = (WindowState)focusObj;

                if (DEBUG_INPUT) Log.v(
                        TAG, "Dispatching to " + focus + ": " + event);

                if (uid != 0 && uid != focus.mSession.mUid) {
                        if (mContext.checkPermission(
                                android.Manifest.permission.INJECT_EVENTS, pid, uid)
                                != PackageManager.PERMISSION_GRANTED) {
                                Log.w(TAG, "Permission denied: injecting key event from pid "
                                                + pid + " uid " + uid + " to window " + focus
                                                + " owned by uid " + focus.mSession.mUid);
                                return INJECT_NO_PERMISSION;
                        }
                }

                synchronized(mWindowMap) {
                        mKeyWaiter.bindTargetWindowLocked(focus);
                }

                // NOSHIP extra state logging
                mKeyWaiter.recordDispatchState(event, focus);
                // END NOSHIP

                try {
```

```
            if (DEBUG_INPUT || DEBUG_FOCUS) {
                Log.v(TAG, "Delivering key " + event.getKeyCode()
                        + " to " + focus);
            }
            focus.mClient.dispatchKey(event);
            return INJECT_SUCCEEDED;
        } catch (android.os.RemoteException e) {
            Log.i(TAG, "WINDOW DIED during key dispatch: " + focus);
            try {
                removeWindow(focus.mSession, focus.mClient);
            } catch (java.util.NoSuchElementException ex) {
                // This will happen if the window has already been
                // removed.
            }
        }

        return INJECT_FAILED;
    }
```

到此先结束，后续有时间，将会把有些问题理清。谢谢你的阅读。共享知识！！