# Todo

[**ADB的公私钥认证机制**](http://www.miui.com/thread-1703592-1-1.html)

<http://www.miui.com/thread-1703592-1-1.html>

发送消息弹窗，我们直接弹窗

Device/google/accessory/Arduino/AndroidAccessory

<https://www.jianshu.com/p/3dc561555f12>

锁屏状态，插入USB为何只是显示充电，不显示dlg选择呢

<https://www.jianshu.com/p/c2a8987e1c0d>

# [WiFi debug调试环境搭建](https://confluence.djicorp.com/pages/viewpage.action?pageId=73949128)

使用场景：手机或者平板设备（下统称平板）的type c接口已经被飞机或者手持云台等设备占用，无法使用usb线连接电脑，进而无法使用有线连接进行调试。

使用前提：平板和电脑处于同一个局域网，最简单的方案就是平板或者电脑开热点，另外一个设备去连接，但是无线连接可能会存在不稳定的情况，如果手上的平板可以通过转接口连接网线（如dji Crystalsky 平板就除了提供一个type c口外，还提供了一个usb口，那么我们就可以使用网口转usb的设备，让平板进行有线网络连接），则使用有线连接会更稳定。

步骤1:先使用usb线连接平板与电脑，开启调试端口

使用usb线连接平板与电脑后，使用**adb devices** 可以查看所有连接上的android设备.

确认设备连接上以后，可以使用 **adb tcpip 端口号** 进行调试端口的打开，如adb tcpip 5555 ，如果开启成功 则会返回 restarting in TCP mode port: 5555

步骤2:查看平板ip

使用 **adb shell netcfg** 或者 **adb shell ifconfig** 进行平板的ip查看

可能列出的ip比较多，找到是跟电脑在同一局域网的那个即可。

注：不同系统可能使用的命令不一样，比如 安卓5.1 的dji Crystalsky  的是adb shell netcfg 而 安卓8 的米6是adb shell ifconfig

adb shell 后接的命令是在平板上执行的命令

步骤3: 拔掉usb线，进行无线连接

使用 **adb connect ip:端口号** 即可连接

如果连接成功 则会输出 connected to ip:端口号

如果连接失败，则会输出相应信息。如端口没打开，会 输出unable to connect to ip:端口号: Connection refused

连接上以后在android studio里面调试就可以看到对应的设备了，同时type c口也空出来可以用来连接飞机了

 ps：

如果要断开连接，可以使用adb disconnect ip:端口号

如果平板重启后，则端口需要重新打开（ip可能不变），如果没重启，则断开后可以直接连接，无需重新打开端口。

如果平板插拔了typec口，则可能会让连接断开，重新执行 adb connect ip:端口号 重新连接即可

如果电脑连接了多个设备 则在adb 后面接 -s 设备号 即可对特定设备执行 设备号可以使用 adb devices 查看

# 0背景

lnx.ker.drivers.usb.docx

# Kernel层

<https://android.googlesource.com/kernel/tegra/+/050d171d5e47a539c616ffea9e2da82ef16bb5bc/drivers/usb/gadget/f_accessory.c>

/kernel/drivers/usb/gadget/f\_accessory.c中收到ACCESSORY\_START这个ioctl后（其实是由usb中断传递上来的）就会发送ACCESSORY=START的uevent。

## acc\_ctrlrequest

static int acc\_ctrlrequest(struct usb\_composite\_dev \*cdev,

const struct usb\_ctrlrequest \*ctrl)

{

struct acc\_dev \*dev = \_acc\_dev;

int value = -EOPNOTSUPP;

struct acc\_hid\_dev \*hid;

int offset;

u8 b\_requestType = ctrl->bRequestType;

u8 b\_request = ctrl->bRequest;

u16 w\_index = le16\_to\_cpu(ctrl->wIndex);

u16 w\_value = le16\_to\_cpu(ctrl->wValue);

u16 w\_length = le16\_to\_cpu(ctrl->wLength);

unsigned long flags;

if (b\_requestType == (USB\_DIR\_OUT | USB\_TYPE\_VENDOR)) {

if (b\_request == ACCESSORY\_START) {

dev->start\_requested = 1;

schedule\_delayed\_work(

&dev->start\_work, msecs\_to\_jiffies(10));

value = 0;

}

## acc\_start\_work

|  |
| --- |
| static void acc\_start\_work(struct work\_struct \*data)  { |
| char \*envp[2] = { "ACCESSORY=START", NULL }; |
| kobject\_uevent\_env(&acc\_device.this\_device->kobj, KOBJ\_CHANGE, envp); |
| } |

UsbDeviceManager.java中，接收到uevent后调用startAccessoryMode();--->setCurrentFunctions(xxx)-->设置sys.usb.config这个属性后，就触发init.xxx.usb.rc去通知kernel切换到accessory模式。

这个3399设备的节点为dev/usb\_accessory

# FW层

api定义：

android/hardware/usb/UsbManager.java

**public static final** String ***ACTION\_USB\_ACCESSORY\_ATTACHED*** =**"android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED"**;

相关源码：

UsbSettingsManager.java

**mHandler** = **new** UsbHandler(FgThread.*get*().getLooper());

**public static final** String ***USB\_FUNCTION\_NONE*** = **"none"**;

**public static final** String ***USB\_FUNCTION\_ADB*** = **"adb"**;

**public static final** String ***USB\_FUNCTION\_RNDIS*** = **"rndis"**;

**public static final** String ***USB\_FUNCTION\_MTP*** = **"mtp"**;

**public static final** String ***USB\_FUNCTION\_PTP*** = **"ptp"**;

**public static final** String ***USB\_FUNCTION\_AUDIO\_SOURCE*** = **"audio\_source"**;

**public static final** String ***USB\_FUNCTION\_MIDI*** = **"midi"**;

**public static final** String ***USB\_FUNCTION\_ACCESSORY*** = **"accessory"**;

进入配件模式超时时间为10秒！！

**private static final int *ACCESSORY\_REQUEST\_TIMEOUT*** = 10 \* 1000;

## 注册usb事件

### UsbHandler

**private static final** String ***USB\_STATE\_MATCH*** =  
 **"DEVPATH=/devices/virtual/android\_usb/android0"**;  
**private static final** String ***ACCESSORY\_START\_MATCH*** =  
 **"DEVPATH=/devices/virtual/misc/usb\_accessory"**;

*// Watch for USB configuration changes***mUEventObserver**.startObserving(***USB\_STATE\_MATCH***);  
**mUEventObserver**.startObserving(***ACCESSORY\_START\_MATCH***);

### mUEventObserver

*/\*  
 \* Listens for uevent messages from the kernel to monitor the USB state  
 \*/***private final** UEventObserver **mUEventObserver** = **new** UEventObserver() {  
 @Override  
 **public void** onUEvent(UEventObserver.UEvent event) {  
 **if** (***DEBUG***) Slog.*v*(***TAG***, **"USB UEVENT: "** + event.toString());  
  
 String state = event.get(**"USB\_STATE"**);  
 String accessory = event.get(**"ACCESSORY"**);  
 **if** (state != **null**) {  
 **mHandler**.updateState(state);  
 } **else if** (**"START"**.equals(accessory)) {  
 **if** (***DEBUG***) Slog.*d*(***TAG***, **"got accessory start"**);  
 startAccessoryMode();  
 }  
 }  
};

*2019-04-07 12:15:43.594 3655-3964/? V/UsbDeviceManager: onUEvent(Device Cable) : {DEVNAME=usb\_accessory, SUBSYSTEM=misc, SEQNUM=55367, MAJOR=10, ACTION=change, DEVPATH=/devices/virtual/misc/usb\_accessory, MINOR=20, ACCESSORY=START}*

#### startAccessoryMode

UsbDeviceManager.java中，接收到uevent后调用startAccessoryMode();--->setCurrentFunctions(xxx)-->设置sys.usb.config这个属性后，就触发init.xxx.usb.rc去通知kernel切换到accessory模式。

这个3399设备的节点为dev/usb\_accessory

**private void** startAccessoryMode() {  
 **if** (!**mHasUsbAccessory**) **return**;  
  
 **mAccessoryStrings** = nativeGetAccessoryStrings();  
 **boolean** enableAudio = (nativeGetAudioMode() == ***AUDIO\_MODE\_SOURCE***);  
 *// don't start accessory mode if our mandatory strings have not been set* **boolean** enableAccessory = (**mAccessoryStrings** != **null** &&  
 **mAccessoryStrings**[UsbAccessory.MANUFACTURER\_STRING] != **null** &&  
 **mAccessoryStrings**[UsbAccessory.MODEL\_STRING] != **null**);  
 String functions = **null**;  
  
 **if** (enableAccessory && enableAudio) {  
 functions = UsbManager.USB\_FUNCTION\_ACCESSORY + **","** + UsbManager.USB\_FUNCTION\_AUDIO\_SOURCE;  
 } **else if** (enableAccessory) {  
  **functions = UsbManager.USB\_FUNCTION\_ACCESSORY;**  
 } **else if** (enableAudio) {  
 functions = UsbManager.USB\_FUNCTION\_AUDIO\_SOURCE;  
 }  
  
 **if** (functions != **null**) {  
 **mAccessoryModeRequestTime** = SystemClock.*elapsedRealtime*();  
 setCurrentFunctions(functions);  
 }  
}

C:\k\android-8.0.0\_r1\frameworks\base\services\core\jni\com\_android\_server\_UsbDeviceManager.cpp

nativeGetAccessoryStrings从对端获取到的数据

通过这个设备节点读取#define DRIVER\_NAME **"/dev/usb\_accessory"**

返回值String[] **mAccessoryStrings**

**static** jobjectArray android\_server\_UsbDeviceManager\_getAccessoryStrings(JNIEnv \*env,  
 jobject */\* thiz \*/*)  
{  
 **int** fd = open(DRIVER\_NAME, O\_RDWR);  
 **if** (fd < 0) {  
 ALOGE(**"could not open %s"**, DRIVER\_NAME);  
 **return** NULL;  
 }  
 jclass stringClass = env->FindClass(**"java/lang/String"**);  
 jobjectArray strArray = env->NewObjectArray(6, stringClass, NULL);  
 **if** (!strArray) **goto** out;  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_MANUFACTURER, strArray, 0);  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_MODEL, strArray, 1);  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_DESCRIPTION, strArray, 2);  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_VERSION, strArray, 3);  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_URI, strArray, 4);  
 set\_accessory\_string(env, fd, ACCESSORY\_GET\_STRING\_SERIAL, strArray, 5);  
  
out:  
 close(fd);  
 **return** strArray;  
}

**static void** set\_accessory\_string(JNIEnv \*env, **int** fd, **int** cmd, jobjectArray strArray, **int** index)  
{  
 **char** buffer[256];  
  
 buffer[0] = 0;  
 ioctl(fd, cmd, buffer);  
 **if** (buffer[0]) {  
 jstring obj = env->NewStringUTF(buffer);  
 env->SetObjectArrayElement(strArray, index, obj);  
 env->DeleteLocalRef(obj);  
 }  
}

C:\k\android-8.0.0\_r1\bionic\libc\kernel\android\uapi\linux\usb\f\_accessory.h

#define ACCESSORY\_GET\_STRING\_MANUFACTURER \_IOW(**'M'**, 1, **char**[256])

#### setCurrentFunctions

functions= ***USB\_FUNCTION\_ACCESSORY*** = **"accessory"**

**public void** setCurrentFunctions(String functions) {  
 **if** (***DEBUG***) Slog.*d*(***TAG***, **"setCurrentFunctions("** + functions + **")"**);  
 **mHandler**.sendMessage(***MSG\_SET\_CURRENT\_FUNCTIONS***, functions);  
}

**case** MSG\_SET\_CURRENT\_FUNCTIONS:  
 String functions = (String)msg.obj;  
 setEnabledFunctions(functions, **false**);  
 **break**;

#### setEnabledFunctions

setEnabledFunctions(**"accessory"**, **false**);

**private void** setEnabledFunctions(String functions, **boolean** forceRestart) {  
 **if** (***DEBUG***) Slog.*d*(***TAG***, **"setEnabledFunctions functions="** + functions + **", "** + **"forceRestart="** + forceRestart);  
  
 *// Try to set the enabled functions.* **final** String oldFunctions = **mCurrentFunctions**;  
 **final boolean** oldFunctionsApplied = **mCurrentFunctionsApplied**;  
 **if** (trySetEnabledFunctions(functions, forceRestart)) {  
 **return**;  
 }  
  
 *// Didn't work. Try to revert changes.  
 // We always reapply the policy in case certain constraints changed such as  
 // user restrictions independently of any other new functions we were  
 // trying to activate.* **if** (oldFunctionsApplied && !oldFunctions.equals(functions)) {  
 Slog.*e*(***TAG***, **"Failsafe 1: Restoring previous USB functions."**);  
 **if** (trySetEnabledFunctions(oldFunctions, **false**)) {  
 **return**;  
 }  
 }

#### trySetEnabledFunctions

每次更新都会带上adb的原因咯，不然掉线了哈

trySetEnabledFunctions(**"accessory"**, **false**);

**private boolean** trySetEnabledFunctions(String functions, **boolean** forceRestart) {  
 **if** (functions == **null**) {  
 functions = getDefaultFunctions();  
 }  
 functions = applyAdbFunction(functions);  
 functions = applyOemOverrideFunction(functions);  
  
 **if** (!**mCurrentFunctions**.equals(functions) || !**mCurrentFunctionsApplied** || forceRestart) {  
 Slog.*i*(***TAG***, **"Setting USB config to "** + functions);  
 **mCurrentFunctions** = functions;  
 **mCurrentFunctionsApplied** = **false**;  
  
 *// Kick the USB stack to close existing connections.* setUsbConfig(UsbManager.USB\_FUNCTION\_NONE);  
  
 *// Set the new USB configuration.* **if** (!setUsbConfig(functions)) {  
 Slog.*e*(***TAG***, **"Failed to switch USB config to "** + functions);  
 **return false**;  
 }  
  
 **mCurrentFunctionsApplied** = **true**;  
 }  
 **return true**;  
}

**private** String applyAdbFunction(String functions) {  
 **if** (**mAdbEnabled**) {  
 functions = UsbManager.addFunction(functions, UsbManager.USB\_FUNCTION\_ADB);  
 } **else** {  
 functions = UsbManager.removeFunction(functions, UsbManager.USB\_FUNCTION\_ADB);  
 }  
 **return** functions;  
}

Setting USB config [mtp,adb] to [accessory,adb]

#### setUsbConfig

setUsbConfig(accessory,adb)

**private boolean** setUsbConfig(String config) {  
 **if** (***DEBUG***) Slog.*d*(***TAG***, **"setUsbConfig("** + config + **")"**);  
 *// set the new configuration  
 // we always set it due to b/23631400, where adbd was getting killed  
 // and not restarted due to property timeouts on some devices* SystemProperties.*set*(***USB\_CONFIG\_PROPERTY***, config);  
 **return** waitForState(config);  
}  
**private static final** String ***USB\_CONFIG\_PROPERTY*** = **"sys.usb.config"**;

## 内部路由事件

mHandler.updateState(state);

### updateState(String state)

**public void** updateState(String state) {  
 **int** connected, configured;  
  
 **if** (**"DISCONNECTED"**.equals(state)) {  
 connected = 0;  
 configured = 0;  
 } **else if** (**"CONNECTED"**.equals(state)) {  
 connected = 1;  
 configured = 0;  
 } **else if** (**"CONFIGURED"**.equals(state)) {  
 connected = 1;  
 configured = 1;  
 } **else** {  
 Slog.*e*(***TAG***, **"unknown state "** + state);  
 **return**;  
 }  
 removeMessages(***MSG\_UPDATE\_STATE***);  
 Message msg = Message.*obtain*(**this**, ***MSG\_UPDATE\_STATE***);  
 msg.**arg1** = connected;  
 msg.**arg2** = configured;  
 *// debounce disconnects to avoid problems bringing up USB tethering* sendMessageDelayed(msg, (connected == 0) ? ***UPDATE\_DELAY*** : 0);  
}

### handleMessage

**case** MSG\_UPDATE\_STATE:  
 mConnected = (msg.arg1 == 1);  
 mConfigured = (msg.arg2 == 1);  
 **if** (!mConnected) {  
 *// When a disconnect occurs, relock access to sensitive user data* mUsbDataUnlocked = **false**;  
 }  
 updateUsbNotification();  
 updateAdbNotification();  
 **if** (UsbManager.containsFunction(mCurrentFunctions,  
 UsbManager.USB\_FUNCTION\_ACCESSORY)) {  
 updateCurrentAccessory();  
 } **else if** (!mConnected) {  
 *// restore defaults when USB is disconnected* setEnabledFunctions(**null**, **false**);  
 }  
 **if** (mBootCompleted) {  
 updateUsbStateBroadcastIfNeeded();  
 updateUsbFunctions();  
 }

启动完成之后才会发广播，因此可能会遗漏事件

### updateCurrentAccessory

**mCurrentAccessory** = **new** UsbAccessory(**mAccessoryStrings**);

com/android/server/usb/UsbSettingsManager.java

## 查找接收者UsbSettingsManager .accessoryAttached

**public void** accessoryAttached(UsbAccessory accessory) {  
 Intent **intent** = **new** Intent(UsbManager.***ACTION\_USB\_ACCESSORY\_ATTACHED***);  
 intent.putExtra(UsbManager.***EXTRA\_ACCESSORY***, accessory);  
 intent.addFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***);  
  
 ArrayList<ResolveInfo> matches;  
 String defaultPackage;  
 **synchronized** (**mLock**) {  
 matches = getAccessoryMatchesLocked(accessory, intent);  
 *// Launch our default activity directly, if we have one.  
 // Otherwise we will start the UsbResolverActivity to allow the user to choose.* defaultPackage = **mAccessoryPreferenceMap**.get(**new** AccessoryFilter(accessory));  
 }  
  
 resolveActivity(**intent**, matches, defaultPackage, **null**, accessory);  
}

### getAccessoryMatchesLocked

**private final** ArrayList<ResolveInfo> getAccessoryMatchesLocked(  
 UsbAccessory accessory, Intent intent) {  
 ArrayList<ResolveInfo> matches = **new** ArrayList<ResolveInfo>();  
 List<ResolveInfo> resolveInfos = **mPackageManager**.queryIntentActivities(intent,  
 PackageManager.***GET\_META\_DATA***);  
 **int** count = resolveInfos.size();  
 **for** (**int** i = 0; i < count; i++) {  
 ResolveInfo resolveInfo = resolveInfos.get(i);  
 **if** (packageMatchesLocked(resolveInfo, intent.getAction(), **null**, accessory)) {  
 matches.add(resolveInfo);  
 }  
 }  
 **return** matches;  
}

### packageMatchesLocked

**if** (accessory != **null** && **"usb-accessory"**.equals(tagName)) {  
 AccessoryFilter filter = AccessoryFilter.*read*(parser);  
 **if** (filter.matches(accessory)) {  
 **return true**;  
 }  
}

### AccessoryFilter

**public static** AccessoryFilter read(XmlPullParser parser)  
 **throws** XmlPullParserException, IOException {  
 String manufacturer = **null**;  
 String model = **null**;  
 String version = **null**;  
  
 **int** count = parser.getAttributeCount();  
 **for** (**int** i = 0; i < count; i++) {  
 String name = parser.getAttributeName(i);  
 String value = parser.getAttributeValue(i);  
  
 **if** (**"manufacturer"**.equals(name)) {  
 manufacturer = value;  
 } **else if** (**"model"**.equals(name)) {  
 model = value;  
 } **else if** (**"version"**.equals(name)) {  
 version = value;  
 }  
 }  
 **return new** AccessoryFilter(manufacturer, model, version);  
}

**public boolean** matches(AccessoryFilter f) {  
 **if** (**mManufacturer** != **null** && !f.**mManufacturer**.equals(**mManufacturer**)) **return false**;  
 **if** (**mModel** != **null** && !f.**mModel**.equals(**mModel**)) **return false**;  
 **if** (**mVersion** != **null** && !f.**mVersion**.equals(**mVersion**)) **return false**;  
 **return true**;  
}

## 接收者转发

### FW层路由UsbSettingsManager. resolveActivity

*// don't show the resolver activity if there are no choices available***if** (count == 0) {  
 **if** (accessory != **null**) {  
 String uri = accessory.getUri();  
 **if** (uri != **null** && uri.length() > 0) {  
 *// display URI to user  
 // start UsbResolverActivity so user can choose an activity* Intent dialogIntent = **new** Intent();  
 dialogIntent.setClassName(**"com.android.systemui"**,  
 **"com.android.systemui.usb.UsbAccessoryUriActivity"**);  
 dialogIntent.addFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***);  
 dialogIntent.putExtra(UsbManager.***EXTRA\_ACCESSORY***, accessory);  
 dialogIntent.putExtra(**"uri"**, uri);  
 **try** {  
 **mUserContext**.startActivityAsUser(dialogIntent, **mUser**);  
 } **catch** (ActivityNotFoundException e) {  
 Slog.*e*(***TAG***, **"unable to start UsbAccessoryUriActivity"**);  
 }  
 }  
 }  
  
 *// do nothing* **return**;  
}

**if** (defaultRI != **null**) {  
 *// grant permission for default activity* **if** (device != **null**) {  
 grantDevicePermission(device, defaultRI.**activityInfo**.**applicationInfo**.**uid**);  
 } **else if** (accessory != **null**) {  
 grantAccessoryPermission(accessory, defaultRI.**activityInfo**.**applicationInfo**.**uid**);  
 }  
  
 *// start default activity directly* **try** {  
 intent.setComponent(  
 **new** ComponentName(defaultRI.**activityInfo**.**packageName**,  
 defaultRI.**activityInfo**.**name**));  
 **mUserContext**.startActivityAsUser(intent, **mUser**);  
 } **catch** (ActivityNotFoundException e) {  
 Slog.*e*(***TAG***, **"startActivity failed"**, e);  
 }  
} **else** {  
 Intent resolverIntent = **new** Intent();  
 resolverIntent.addFlags(Intent.***FLAG\_ACTIVITY\_NEW\_TASK***);  
  
  **if (count == 1) {**  
 *// start UsbConfirmActivity if there is only one choice* resolverIntent.setClassName(**"com.android.systemui"**,  
 **"com.android.systemui.usb.UsbConfirmActivity"**);  
 resolverIntent.putExtra(**"rinfo"**, matches.get(0));  
  
 **if** (device != **null**) {  
 resolverIntent.putExtra(UsbManager.***EXTRA\_DEVICE***, device);  
 } **else** {  
 resolverIntent.putExtra(UsbManager.***EXTRA\_ACCESSORY***, accessory);  
 }  
 } **else** {  
 *// start UsbResolverActivity so user can choose an activity* resolverIntent.setClassName(**"com.android.systemui"**,  
 **"com.android.systemui.usb.UsbResolverActivity"**);  
  **resolverIntent.putParcelableArrayListExtra("rlist", matches);**  
 resolverIntent.putExtra(Intent.***EXTRA\_INTENT***, intent);  
 }  
 **try** {  
 **mUserContext**.startActivityAsUser(resolverIntent, **mUser**);  
 } **catch** (ActivityNotFoundException e) {  
 Slog.*e*(***TAG***, **"unable to start activity "** + resolverIntent);  
 }  
}

### 系统APP层

Systemui.UsbResolverActivity

### 第三方app层

## 架构图

时序图

# 实战问题

多次插拔USB线，系统异常，并且造成系统奔溃

## TODO

**public class** UsbConfirmActivity **extends** AlertActivity

# 三.相关API

## 1.两个相关的类

| **Class** | **Description** |
| --- | --- |
| [UsbManager](https://developer.android.com/reference/android/hardware/usb/UsbManager.html) | 允许枚举已连接的USB设备并与其通信 |
| [UsbAccessory](https://developer.android.com/reference/android/hardware/usb/UsbAccessory.html) | 是代表USB配件的类，该类提供了方法访问配件的信息 |

通过

UsbManager manager = UsbManager.getInstance(this);或

UsbManager manager = (UsbManager) getSystemService(Context.USB\_SERVICE);

UsbAccessory accessory = UsbManager.getAccessory(intent);或

UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA\_ACCESSORY);

来获取这两个类的实例

## 2.Android manifest

* 声明<uses-feature android:name="android.hardware.usb.accessory" />
* 为主活动中的 android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED 意图指定<intent-filter>和<meta-data>元素对。<meta-data>元素指向外部XML资源文件，该文件声明有关要检测的附件的标识信息。
* 在XML资源文件中，声明要过滤的附件的<usb-accessory>元素。每个<usb-accessory>都可以具有以下属性：manufacturer，model，version，将资源文件保存在res / xml /目录中。资源文件名（不带.xml扩展名）必须与在<meta-data>元素中指定的名称相同。

示例：

**AndroidManifest.xml**

|  |
| --- |
| <manifest ...>      <uses-feature android:name="android.hardware.usb.accessory" />        <uses-sdk android:minSdkVersion="<version>" />      ...      <application>        <uses-library android:name="com.android.future.usb.accessory" />          <activity ...>              ...              <intent-filter>                  <action android:name="android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED" />              </intent-filter>                <meta-data android:name="android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED"                  android:resource="@xml/accessory\_filter" />          </activity>      </application>  </manifest> |

**res/xml/accessory\_filter.xml：**

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <resources>      <usb-accessory model="DemoKit" manufacturer="Google" version="1.0"/>  </resources> |

# 四.通信过程

## 1.监听usb设备attach

先注册监听usb设备attach的广播，然后通过

|  |
| --- |
| list = usbManager.getAccessoryList();  accessory = list[0]; |

拿到辅助设备UsbAccessory的对象

## 2.授权

注册监听授权的广播，可以和监听usb设备attach的receiver合并

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | private static final String ACTION\_USB\_PERMISSION =      "com.android.example.USB\_PERMISSION";  private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {        public void onReceive(Context context, Intent intent) {          String action = intent.getAction();          if (ACTION\_USB\_PERMISSION.equals(action)) {              synchronized (this) {                  UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA\_ACCESSORY);                    if (intent.getBooleanExtra(UsbManager.EXTRA\_PERMISSION\_GRANTED, false)) {                      if(accessory != null){                          //call method to set up accessory communication                      }                  }                  else {                      Log.d(TAG, "permission denied for accessory " + accessory);                  }              }          }      }  }; |

在Activity onCreat()注册此receiver

|  |
| --- |
| UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB\_SERVICE);  private static final String ACTION\_USB\_PERMISSION =      "com.android.example.USB\_PERMISSION";  ...  mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION\_USB\_PERMISSION), 0);  IntentFilter filter = new IntentFilter(ACTION\_USB\_PERMISSION);  registerReceiver(mUsbReceiver, filter); |

在检测到usb设备attach后请求权限

|  |
| --- |
| UsbAccessory accessory;  ...  mUsbManager.requestPermission(accessory, mPermissionIntent); |

## 3.通信

授权成功后可以通过文件描述符进行通信

|  |
| --- |
| UsbAccessory mAccessory;  ParcelFileDescriptor mFileDescriptor;  FileInputStream mInputStream;  FileOutputStream mOutputStream;    ...    private void openAccessory() {      Log.d(TAG, "openAccessory: " + accessory);      mFileDescriptor = mUsbManager.openAccessory(mAccessory);      if (mFileDescriptor != null) {          FileDescriptor fd = mFileDescriptor.getFileDescriptor();          mInputStream = new FileInputStream(fd);          mOutputStream = new FileOutputStream(fd);          Thread thread = new Thread(null, this, "AccessoryThread");          thread.start();      }  } |

## 4.设备退出后关闭fd，清理

|  |
| --- |
| BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {      public void onReceive(Context context, Intent intent) {          String action = intent.getAction();            if (UsbManager.ACTION\_USB\_ACCESSORY\_DETACHED.equals(action)) {              UsbAccessory accessory = (UsbAccessory)intent.getParcelableExtra(UsbManager.EXTRA\_ACCESSORY);              if (accessory != null) {                  // call your method that cleans up and closes communication with the accessory                  mFileDescriptor.close()                  ...              }          }      }  }; |

# Host-bulk

连接usb存储设备（u盘，硬盘等）时，设备节点在/dev/bus/usb下，由UsbHostManager.java来管理

Android设备最早都是以PC的USB外设形式而存在,主要功能就是[Android fastboot](https://link.jianshu.com/?t=http://en.wikipedia.org/wiki/Android_software_development" \l "Fastboot" \t "_blank) 或[Android Debug Bridge (adb)](https://link.jianshu.com/?t=http://developer.android.com/tools/help/adb.html" \t "_blank),都是基于bulk传输模式的.

## 启动原理

### UsbService.java中的systemReady

1. fw/services/\*\*\*/usb/UsbService.java中的systemReady()调用mHostManager.systemReady()。

UsbService监听启动完成的科学方法

**public class** UsbService **extends** IUsbManager.Stub {  
  
 **public static class** Lifecycle **extends** SystemService {  
 **private** UsbService **mUsbService**;  
  
 **public** Lifecycle(Context context) {  
 **super**(context);  
 }  
  
 @Override  
 **public void** onStart() {  
 **mUsbService** = **new** UsbService(getContext());  
 publishBinderService(Context.***USB\_SERVICE***, **mUsbService**);  
 }  
  
 @Override  
 **public void** onBootPhase(**int** phase) {  
 **if** (phase == SystemService.***PHASE\_ACTIVITY\_MANAGER\_READY***) {  
 **mUsbService**.systemReady();  
 } **else if** (phase == SystemService.***PHASE\_BOOT\_COMPLETED***) {  
 **mUsbService**.bootCompleted();  
 }  
 }  
 }

**public void** systemReady() {  
 **if** (**this**.mDeviceManager != **null**) {  
 **this**.mDeviceManager.systemReady();  
 }  
  
 **if** (**this**.mHostManager != **null**) {  
 **this**.mHostManager.systemReady();  
 }  
  
}

### UsbHostManager. monitorUsbHostBus

UsbHostManager.java的systemReady中启动一个线程来运行monitorUsbHostBus();

**public void** systemReady() {  
 **synchronized** (**mLock**) {  
 *// Create a thread to call into native code to wait for USB host events.  
 // This thread will call us back on usbDeviceAdded and usbDeviceRemoved.* Runnable runnable = **new** Runnable() {  
 **public void** run() {  
 monitorUsbHostBus();  
 }  
 };  
 **new** Thread(**null**, runnable, **"UsbService host thread"**).start();  
 }  
}

**private native void** monitorUsbHostBus();

### jni.monitorUsbHostBus

frameworks/base/services/core/jni/com\_android\_server\_UsbHostManager.cpp

android\_server\_UsbHostManager\_monitorUsbHostBus函数，在这个函数调用了usb\_host\_init函数，创建了一个INotify的fd，以及创建了一个usb\_host\_context对象。usb\_host\_run函数就是循环读取INotify的fd的事件，我们把usb\_device\_added, usb\_device\_removed两个回调函数也传入了usb\_host\_run函数了。

---------------------

|  |
| --- |
| static void android\_server\_UsbHostManager\_monitorUsbHostBus(JNIEnv \*env, jobject thiz)  { |
| struct usb\_host\_context\* context = **usb\_host\_ini**t(); |
| if (!context) { |
| ALOGE("usb\_host\_init failed"); |
| return; |
| } |
| // this will never return so it is safe to pass thiz directly |
| **usb\_host\_run**(context, usb\_device\_added, usb\_device\_removed, NULL, (void \*)thiz); |
| } |

其中分别调用到了system/core/libusbhost/usbhost.c中的usb\_host\_init(...)和usb\_host\_run(...)

#### libusbhost .usb\_host\_init

在usb\_host\_init()中，最主要的是初始化context->fd = inotify\_init();，这个会在后面用来监听**/dev/bus/usb**目录的创建和删除

usb\_host\_init是在system\core\libusbhost\usbhost.c文件中，这个函数中新建一个usb\_host\_context对象，还有新建了一个INotify，并且usb\_host\_context的fd就是INotify的fd。

#### libusbhost .usb\_host\_run

在usb\_host\_run中，主要是添加监控的目录ret = inotify\_add\_watch(context->fd, path, IN\_CREATE | IN\_DELETE);如果发现目录有create或者是delete操作，通知回调函数usb\_device\_added和usb\_device\_removed

我们再来看看usb\_host\_run函数。先调用了usb\_host\_load函数，这个函数主要把add和remove的回调，放到context相应的成员变量中，然后增加了dev目录放入INotify的观察。下面循环调用usb\_host\_read\_event函数去读取INotify fd的事件。

void usb\_host\_run(struct usb\_host\_context \*context,

usb\_device\_added\_cb added\_cb,

usb\_device\_removed\_cb removed\_cb,

usb\_discovery\_done\_cb discovery\_done\_cb,

void \*client\_data)

{

int done;

done = usb\_host\_load(context, added\_cb, removed\_cb, discovery\_done\_cb, client\_data);

while (!done) {

done = usb\_host\_read\_event(context);

}

}

usb\_host\_load函数先把add和remove的两个回调设置到usb\_host\_context 中，然后将dev目录放入INotify观察。然后调用watch\_existing\_subdirs将/dev/bus/usb下的目录都添加到INotify中观察，最后再调用find\_existing\_devices函数，把找到的设备调用added\_cb（增加的回调函数）

int usb\_host\_load(struct usb\_host\_context \*context,

usb\_device\_added\_cb added\_cb,

usb\_device\_removed\_cb removed\_cb,

usb\_discovery\_done\_cb discovery\_done\_cb,

void \*client\_data)

{

int done = 0;

int i;

context->cb\_added = added\_cb;//回调赋值

context->cb\_removed = removed\_cb;

context->data = client\_data;

D("Created device discovery thread\n");

/\* watch for files added and deleted within USB\_FS\_DIR \*/

context->wddbus = -1;

for (i = 0; i < MAX\_USBFS\_WD\_COUNT; i++)

context->wds[i] = -1;

/\* watch the root for new subdirectories \*/

context->wdd = inotify\_add\_watch(context->fd, DEV\_DIR, IN\_CREATE | IN\_DELETE);//将dev目录放入INotify观察

if (context->wdd < 0) {

fprintf(stderr, "inotify\_add\_watch failed\n");

if (discovery\_done\_cb)

discovery\_done\_cb(client\_data);

return done;

}

watch\_existing\_subdirs(context, context->wds, MAX\_USBFS\_WD\_COUNT);

/\* check for existing devices first, after we have inotify set up \*/

done = find\_existing\_devices(added\_cb, client\_data);

if (discovery\_done\_cb)

done |= discovery\_done\_cb(client\_data);

return done;

}

watch\_existing\_subdirs，就是将/dev/bus/usb下的设备目录（001开始到MAX\_USBFS\_WD\_COUNT值都inotify\_add\_watch）都添加到INotify中去。

static void watch\_existing\_subdirs(struct usb\_host\_context \*context,

int \*wds, int wd\_count)

{

char path[100];

int i, ret;

wds[0] = inotify\_add\_watch(context->fd, USB\_FS\_DIR, IN\_CREATE | IN\_DELETE);

if (wds[0] < 0)

return;

/\* watch existing subdirectories of USB\_FS\_DIR \*/

for (i = 1; i < wd\_count; i++) {

snprintf(path, sizeof(path), USB\_FS\_DIR "/%03d", i);

ret = inotify\_add\_watch(context->fd, path, IN\_CREATE | IN\_DELETE);

if (ret >= 0)

wds[i] = ret;

}

}

然后我们再来看find\_existing\_devices函数，就是遍历dev/bus/usb的目录然后再调用find\_existing\_devices\_bus函数

static int find\_existing\_devices(usb\_device\_added\_cb added\_cb,

void \*client\_data)

{

char busname[32];

DIR \*busdir;

struct dirent \*de;

int done = 0;

busdir = opendir(USB\_FS\_DIR);

if(busdir == 0) return 0;

while ((de = readdir(busdir)) != 0 && !done) {

if(badname(de->d\_name)) continue;

snprintf(busname, sizeof(busname), USB\_FS\_DIR "/%s", de->d\_name);

done = find\_existing\_devices\_bus(busname, added\_cb,

client\_data);

} //end of busdir while

closedir(busdir);

return done;

}

find\_existing\_devices\_bus函数，就是将dev/bus/usb下的目录比如001，然后001目录里面的文件，作为一个设备（组成一个devname），再调用added\_cb（增加设备的回调函数）

static int find\_existing\_devices\_bus(char \*busname,

usb\_device\_added\_cb added\_cb,

void \*client\_data)

{

char devname[32];

DIR \*devdir;

struct dirent \*de;

int done = 0;

devdir = opendir(busname);

if(devdir == 0) return 0;

while ((de = readdir(devdir)) && !done) {

if(badname(de->d\_name)) continue;

snprintf(devname, sizeof(devname), "%s/%s", busname, de->d\_name);

done = added\_cb(devname, client\_data);

} // end of devdir while

closedir(devdir);

return done;

}

因此总结下usb\_host\_load函数就是把增加、去除设备的回调赋值到usb\_host\_context 的相关变量中，然后增加相关目录的观察，最后查找已经存在的设备调用added\_cb（增加设备的回调函数）。

然后我们再来看看usb\_host\_read\_event函数，这个函数就是去INotify中的fd读取相关的事件。具体分析我们就看注释，有一点我们要注意了，除了增加bus目录失败返回的done是1，其他的返回的done都是0.也就是我们的usb\_host\_run函数会在while循环中一直循环。

int usb\_host\_read\_event(struct usb\_host\_context \*context)

{

struct inotify\_event\* event;

char event\_buf[512];

char path[100];

int i, ret, done = 0;

int offset = 0;

int wd;

ret = read(context->fd, event\_buf, sizeof(event\_buf));

if (ret >= (int)sizeof(struct inotify\_event)) {

while (offset < ret && !done) {

event = (struct inotify\_event\*)&event\_buf[offset];

done = 0;

wd = event->wd;

if (wd == context->wdd) {

if ((event->mask & IN\_CREATE) && !strcmp(event->name, "bus")) {//增加bus目录，并且将bus目录也增加到watch中

context->wddbus = inotify\_add\_watch(context->fd, DEV\_BUS\_DIR, IN\_CREATE | IN\_DELETE);

if (context->wddbus < 0) {

done = 1;//增加bus目录失败

} else {

watch\_existing\_subdirs(context, context->wds, MAX\_USBFS\_WD\_COUNT);

done = find\_existing\_devices(context->cb\_added, context->data);

}

}

} else if (wd == context->wddbus) {

if ((event->mask & IN\_CREATE) && !strcmp(event->name, "usb")) {//如果是dev/bus目录的事件

watch\_existing\_subdirs(context, context->wds, MAX\_USBFS\_WD\_COUNT);//将dev/bus/usb目录再增加到watch

done = find\_existing\_devices(context->cb\_added, context->data);//然后再看看usb下面时候有设备，有就调用添加函数

} else if ((event->mask & IN\_DELETE) && !strcmp(event->name, "usb")) {

for (i = 0; i < MAX\_USBFS\_WD\_COUNT; i++) {

if (context->wds[i] >= 0) {

inotify\_rm\_watch(context->fd, context->wds[i]);//是删除事件，就把这个watch删除了

context->wds[i] = -1;

}

}

}

} else if (wd == context->wds[0]) {//是第一个wds，也就是dev/bus/usb目录的事件

i = atoi(event->name);

snprintf(path, sizeof(path), USB\_FS\_DIR "/%s", event->name);

D("%s subdirectory %s: index: %d\n", (event->mask & IN\_CREATE) ?

"new" : "gone", path, i);

if (i > 0 && i < MAX\_USBFS\_WD\_COUNT) {

int local\_ret = 0;

if (event->mask & IN\_CREATE) {

local\_ret = inotify\_add\_watch(context->fd, path,//我们需要把usb下新增的目录也增加到watch中

IN\_CREATE | IN\_DELETE);

if (local\_ret >= 0)

context->wds[i] = local\_ret;

done = find\_existing\_devices\_bus(path, context->cb\_added,//然后再看看是否有已经存在的设备

context->data);

} else if (event->mask & IN\_DELETE) {

inotify\_rm\_watch(context->fd, context->wds[i]);

context->wds[i] = -1;

}

}

} else {

for (i = 1; (i < MAX\_USBFS\_WD\_COUNT) && !done; i++) {//最后剩下的肯定是dev/bus/usb下目录的事件

if (wd == context->wds[i]) {

snprintf(path, sizeof(path), USB\_FS\_DIR "/%03d/%s", i, event->name);

if (event->mask == IN\_CREATE) {//这里就直接调用增加设备或者删除设备的回调函数。

D("new device %s\n", path);

done = context->cb\_added(path, context->data);

} else if (event->mask == IN\_DELETE) {

D("gone device %s\n", path);

done = context->cb\_removed(path, context->data);

}

}

}

}

offset += sizeof(struct inotify\_event) + event->len;//读取的字节数增加

}

}

return done;

}

usb\_device\_added函数会调用UsbHostManager的beginUsbDeviceAdded和endUsbDeviceAdded函数，在UsbHostManager中会新建一个UsbDevice，然后放入mDevices中。

static int usb\_device\_added(const char \*devname, void\* client\_data) {

struct usb\_descriptor\_header\* desc;

struct usb\_descriptor\_iter iter;

struct usb\_device \*device = usb\_device\_open(devname);

if (!device) {

ALOGE("usb\_device\_open failed\n");

return 0;

}

JNIEnv\* env = AndroidRuntime::getJNIEnv();

jobject thiz = (jobject)client\_data;

const usb\_device\_descriptor\* deviceDesc = usb\_device\_get\_device\_descriptor(device);

char \*manufacturer = usb\_device\_get\_manufacturer\_name(device);

char \*product = usb\_device\_get\_product\_name(device);

int version = usb\_device\_get\_version(device);

char \*serial = usb\_device\_get\_serial(device);

jstring deviceName = env->NewStringUTF(devname);

jstring manufacturerName = AndroidRuntime::NewStringLatin1(env, manufacturer);

jstring productName = AndroidRuntime::NewStringLatin1(env, product);

jstring serialNumber = AndroidRuntime::NewStringLatin1(env, serial);

jboolean result = env->CallBooleanMethod(thiz, method\_beginUsbDeviceAdded,

deviceName, usb\_device\_get\_vendor\_id(device), usb\_device\_get\_product\_id(device),

deviceDesc->bDeviceClass, deviceDesc->bDeviceSubClass, deviceDesc->bDeviceProtocol,

manufacturerName, productName, version, serialNumber);

env->DeleteLocalRef(serialNumber);

env->DeleteLocalRef(productName);

env->DeleteLocalRef(manufacturerName);

env->DeleteLocalRef(deviceName);

free(manufacturer);

free(product);

free(serial);

if (!result) goto fail;

usb\_descriptor\_iter\_init(device, &iter);

while ((desc = usb\_descriptor\_iter\_next(&iter)) != NULL) {

if (desc->bDescriptorType == USB\_DT\_CONFIG) {

struct usb\_config\_descriptor \*config = (struct usb\_config\_descriptor \*)desc;

char \*name = usb\_device\_get\_string(device, config->iConfiguration);

jstring configName = AndroidRuntime::NewStringLatin1(env, name);

env->CallVoidMethod(thiz, method\_addUsbConfiguration,

config->bConfigurationValue, configName, config->bmAttributes,

config->bMaxPower);

env->DeleteLocalRef(configName);

free(name);

} else if (desc->bDescriptorType == USB\_DT\_INTERFACE) {

struct usb\_interface\_descriptor \*interface = (struct usb\_interface\_descriptor \*)desc;

char \*name = usb\_device\_get\_string(device, interface->iInterface);

jstring interfaceName = AndroidRuntime::NewStringLatin1(env, name);

env->CallVoidMethod(thiz, method\_addUsbInterface,

interface->bInterfaceNumber, interfaceName, interface->bAlternateSetting,

interface->bInterfaceClass, interface->bInterfaceSubClass,

interface->bInterfaceProtocol);

env->DeleteLocalRef(interfaceName);

free(name);

} else if (desc->bDescriptorType == USB\_DT\_ENDPOINT) {

struct usb\_endpoint\_descriptor \*endpoint = (struct usb\_endpoint\_descriptor \*)desc;

env->CallVoidMethod(thiz, method\_addUsbEndpoint,

endpoint->bEndpointAddress, endpoint->bmAttributes,

\_\_le16\_to\_cpu(endpoint->wMaxPacketSize), endpoint->bInterval);

}

}

env->CallVoidMethod(thiz, method\_endUsbDeviceAdded);

fail:

usb\_device\_close(device);

checkAndClearExceptionFromCallback(env, \_\_FUNCTION\_\_);

return 0;

}

去除函数，也会调用UsbHostManager的usbDeviceRemoved函数，注意这个函数返回就是0

static int usb\_device\_removed(const char \*devname, void\* client\_data) {

JNIEnv\* env = AndroidRuntime::getJNIEnv();

jobject thiz = (jobject)client\_data;

jstring deviceName = env->NewStringUTF(devname);

env->CallVoidMethod(thiz, method\_usbDeviceRemoved, deviceName);

env->DeleteLocalRef(deviceName);

checkAndClearExceptionFromCallback(env, \_\_FUNCTION\_\_);

return 0;

}

UsbHostManager的usbDeviceRemoved函数，会把mDevices相关的设备去除。

private void usbDeviceRemoved(String deviceName) {

synchronized (mLock) {

UsbDevice device = mDevices.remove(deviceName);

if (device != null) {

mUsbAlsaManager.usbDeviceRemoved(device);

getCurrentSettings().deviceDetached(device);

}

}

---------------------

作者：kc专栏

来源：CSDN

原文：https://blog.csdn.net/kc58236582/article/details/54691334

版权声明：本文为博主原创文章，转载请附上博文链接！

## jni设计

java主要通过monitorUsbHostBus来初始化cpp

cpp回调了java大部分的函数，因此你会发现UsbHostManager.java都没有地方调用，原因是通过jni方式调用的

### java调用cpp

**static const** JNINativeMethod method\_table[] = {  
 { **"monitorUsbHostBus"**, **"()V"**, (**void**\*)android\_server\_UsbHostManager\_monitorUsbHostBus },  
 { **"nativeOpenDevice"**, **"(Ljava/lang/String;)Landroid/os/ParcelFileDescriptor;"**,  
 (**void**\*)android\_server\_UsbHostManager\_openDevice },  
};

### cpp调用java

**int** register\_android\_server\_UsbHostManager(JNIEnv \*env)  
{  
 jclass clazz = env->FindClass(**"com/android/server/usb/UsbHostManager"**);  
 **if** (clazz == NULL) {  
 ALOGE(**"Can't find com/android/server/usb/UsbHostManager"**);  
 **return** -1;  
 }  
 method\_beginUsbDeviceAdded = env->GetMethodID(clazz, **"beginUsbDeviceAdded"**,  
 **"(Ljava/lang/String;IIIIILjava/lang/String;Ljava/lang/String;ILjava/lang/String;)Z"**);  
 **if** (method\_beginUsbDeviceAdded == NULL) {  
 ALOGE(**"Can't find beginUsbDeviceAdded"**);  
 **return** -1;  
 }  
 method\_addUsbConfiguration = env->GetMethodID(clazz, **"addUsbConfiguration"**,  
 **"(ILjava/lang/String;II)V"**);  
 **if** (method\_addUsbConfiguration == NULL) {  
 ALOGE(**"Can't find addUsbConfiguration"**);  
 **return** -1;  
 }  
 method\_addUsbInterface = env->GetMethodID(clazz, **"addUsbInterface"**,  
 **"(ILjava/lang/String;IIII)V"**);  
 **if** (method\_addUsbInterface == NULL) {  
 ALOGE(**"Can't find addUsbInterface"**);  
 **return** -1;  
 }  
 method\_addUsbEndpoint = env->GetMethodID(clazz, **"addUsbEndpoint"**, **"(IIII)V"**);  
 **if** (method\_addUsbEndpoint == NULL) {  
 ALOGE(**"Can't find addUsbEndpoint"**);  
 **return** -1;  
 }  
 method\_endUsbDeviceAdded = env->GetMethodID(clazz, **"endUsbDeviceAdded"**, **"()V"**);  
 **if** (method\_endUsbDeviceAdded == NULL) {  
 ALOGE(**"Can't find endUsbDeviceAdded"**);  
 **return** -1;  
 }  
 method\_usbDeviceRemoved = env->GetMethodID(clazz, **"usbDeviceRemoved"**,  
 **"(Ljava/lang/String;)V"**);  
 **if** (method\_usbDeviceRemoved == NULL) {  
 ALOGE(**"Can't find usbDeviceRemoved"**);  
 **return** -1;  
 }  
  
 clazz = env->FindClass(**"android/os/ParcelFileDescriptor"**);  
 LOG\_FATAL\_IF(clazz == NULL, **"Unable to find class android.os.ParcelFileDescriptor"**);  
 gParcelFileDescriptorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);  
 gParcelFileDescriptorOffsets.mConstructor = env->GetMethodID(clazz, **"<init>"**,  
 **"(Ljava/io/FileDescriptor;)V"**);  
 LOG\_FATAL\_IF(gParcelFileDescriptorOffsets.mConstructor == NULL,  
 **"Unable to find constructor for android.os.ParcelFileDescriptor"**);  
  
 **return** jniRegisterNativeMethods(env, **"com/android/server/usb/UsbHostManager"**,  
 method\_table, NELEM(method\_table));  
}

## 插入bulk设备

当新连接一个设备之后，UsbHostManager.java中会依次从jni回调beginUsbDeviceAdded—>addUsbConfiguration—>addUsbInterface—>addUsbEndpoint—>endUsbDeviceAdded 这几个方法，分别创建UsbDevice，UsbConfigration,UsbInterface,UsbEndpoint Parcebale 对象

首先可以通过dump查一下字符串

[mName=/dev/bus/usb/003/003,mVendorId=11427,mProductId=31,**mClass**=239,**mSubclass**=2,**mProtocol**=1,mManufacturerName=KJI,mProductName=KJI,mVersion=2.0,mSerialNumber=XX,mConfigurations=[  
UsbConfiguration[mId=1,mName=**null**,mAttributes=192,mMaxPower=0,mInterfaces=[  
 **UsbInterface[mId=3,mAlternateSetting=0,mName=libusb-win32,mClass=255,mSubclass=67,mProtocol=1,mEndpoints=[  
 UsbEndpoint[mAddress=129,mAttributes=2,mMaxPacketSize=512,mInterval=0]  
 UsbEndpoint[mAddress=1,mAttributes=2,mMaxPacketSize=512,mInterval=0]]** UsbInterface[mId=4,mAlternateSetting=0,mName=CDC Communications Control,**mClass**=2,**mSubclass**=2,**mProtocol**=1,**mEndpoints**=[  
 UsbEndpoint[mAddress=131,mAttributes=3,mMaxPacketSize=16,mInterval=32]]  
 UsbInterface[mId=5,mAlternateSetting=0,mName=serial **Data**,**mClass**=10,**mSubclass**=0,**mProtocol**=0,**mEndpoints**=[  
 UsbEndpoint[mAddress=130,mAttributes=2,mMaxPacketSize=512,mInterval=0]  
 UsbEndpoint[mAddress=2,mAttributes=2,mMaxPacketSize=512,mInterval=0]]]

所以从逻辑上来说，一个device->Configuration->Interface->Endpoint都是一对多的关系！

添加设备的过程，核心的功能就是这个解析字符串

### 协议

由上面的日志可以看出，

UsbEndpoint epOut = **null**;  
UsbEndpoint epIn = **null**;  
*// look for our bulk endpoints***for** (**int** i = 0; i < intf.getEndpointCount(); i++) {  
 UsbEndpoint ep = intf.getEndpoint(i);  
 **if** (ep.getType() == UsbConstants.***USB\_ENDPOINT\_XFER\_BULK***) {  
 **if** (ep.getDirection() == UsbConstants.***USB\_DIR\_OUT***) {//<128  
 epOut = ep;  
 } **else** {  
 epIn = ep;  
 }  
 }  
}

端点协议：mAddress&***USB\_ENDPOINT\_DIR\_MASK*** = 0x80;为0就是输出，因此地址上来看，小于128的是输出，一般而言，第二端点就是写入，输出端点

1. //USBEndpoint为读写数据所需的节点
2. UsbEndpoint inEndpoint = usbInterface.getEndpoint(0);  //读数据节点
3. UsbEndpoint outEndpoint = usbInterface.getEndpoint(1); //写数据节点

### jni.usb\_device\_added

frameworks/base/services/core/jni/com\_android\_server\_UsbHostManager.cpp

**static int** usb\_device\_added(**const char** \*devname, **void**\* client\_data) {  
 **struct** usb\_descriptor\_header\* desc;  
 **struct** usb\_descriptor\_iter iter;  
  
 **struct** usb\_device \*device = usb\_device\_open(devname);//1  
 **if** (!device) {  
 ALOGE(**"usb\_device\_open failed\n"**);  
 **return** 0;  
 }  
  
 JNIEnv\* env = AndroidRuntime::getJNIEnv();  
 jobject thiz = (jobject)client\_data;  
 **const** usb\_device\_descriptor\* deviceDesc = usb\_device\_get\_device\_descriptor(device);  
  
 **char** \*manufacturer = usb\_device\_get\_manufacturer\_name(device,  
 USB\_CONTROL\_TRANSFER\_TIMEOUT\_MS);  
 **char** \*product = usb\_device\_get\_product\_name(device,  
 USB\_CONTROL\_TRANSFER\_TIMEOUT\_MS);  
 **int** version = usb\_device\_get\_version(device);  
 **char** \*serial = usb\_device\_get\_serial(device,  
 USB\_CONTROL\_TRANSFER\_TIMEOUT\_MS);  
  
 jstring deviceName = env->NewStringUTF(devname);  
 jstring manufacturerName = AndroidRuntime::NewStringLatin1(env, manufacturer);  
 jstring productName = AndroidRuntime::NewStringLatin1(env, product);  
 jstring serialNumber = AndroidRuntime::NewStringLatin1(env, serial);  
  
 jboolean result = env->CallBooleanMethod(thiz, **method\_beginUsbDeviceAdded**,//2  
 deviceName, usb\_device\_get\_vendor\_id(device), usb\_device\_get\_product\_id(device),  
 deviceDesc->bDeviceClass, deviceDesc->bDeviceSubClass, deviceDesc->bDeviceProtocol,  
 manufacturerName, productName, version, serialNumber);  
  
 env->DeleteLocalRef(serialNumber);  
 env->DeleteLocalRef(productName);  
 env->DeleteLocalRef(manufacturerName);  
 env->DeleteLocalRef(deviceName);  
 free(manufacturer);  
 free(product);  
 free(serial);  
  
 **if** (!result) **goto** fail;  
  
 usb\_descriptor\_iter\_init(device, &iter);  
  
 **while** ((desc = usb\_descriptor\_iter\_next(&iter)) != NULL) {  
 **if** (desc->bDescriptorType == USB\_DT\_CONFIG) {  
 **struct** usb\_config\_descriptor \*config = (**struct** usb\_config\_descriptor \*)desc;  
 **char** \*name = usb\_device\_get\_string(device, config->iConfiguration,  
 USB\_CONTROL\_TRANSFER\_TIMEOUT\_MS);  
 jstring configName = AndroidRuntime::NewStringLatin1(env, name);  
  
 env->CallVoidMethod(thiz, method\_addUsbConfiguration,//3.  
 config->bConfigurationValue, configName, config->bmAttributes,  
 config->bMaxPower);  
  
 env->DeleteLocalRef(configName);  
 free(name);  
 } **else if** (desc->bDescriptorType == USB\_DT\_INTERFACE) {  
 **struct** usb\_interface\_descriptor \***interface** = (**struct** usb\_interface\_descriptor \*)desc;  
 **char** \*name = usb\_device\_get\_string(device, **interface**->iInterface,  
 USB\_CONTROL\_TRANSFER\_TIMEOUT\_MS);  
 jstring interfaceName = AndroidRuntime::NewStringLatin1(env, name);  
  
 env->CallVoidMethod(thiz, method\_addUsbInterface,//4  
 **interface**->bInterfaceNumber, interfaceName, **interface**->bAlternateSetting,  
 **interface**->bInterfaceClass, **interface**->bInterfaceSubClass,  
 **interface**->bInterfaceProtocol);  
  
 env->DeleteLocalRef(interfaceName);  
 free(name);  
 } **else if** (desc->bDescriptorType == USB\_DT\_ENDPOINT) {  
 **struct** usb\_endpoint\_descriptor \*endpoint = (**struct** usb\_endpoint\_descriptor \*)desc;  
  
 env->CallVoidMethod(thiz, method\_addUsbEndpoint,//5  
 endpoint->bEndpointAddress, endpoint->bmAttributes,  
 \_\_le16\_to\_cpu(endpoint->wMaxPacketSize), endpoint->bInterval);  
 }  
 }  
  
 env->CallVoidMethod(thiz, method\_endUsbDeviceAdded);//6  
  
fail:  
 usb\_device\_close(device);  
 checkAndClearExceptionFromCallback(env, \_\_FUNCTION\_\_);  
  
 **return** 0;  
}

### UsbHostManager.beginUsbDeviceAdded

设置了一些过滤条件，

1. 同一个设备不会被add两次，虽然内核支持，但是fw层会过滤掉
2. **mNewDevice is not null in endUsbDeviceAdded**

**private boolean** beginUsbDeviceAdded(String deviceName, **int** vendorID, **int** productID,  
 **int** deviceClass, **int** deviceSubclass, **int** deviceProtocol,  
 String manufacturerName, String productName, **int** version, String serialNumber) {

**if** (isBlackListed(deviceName) ||  
 isBlackListed(deviceClass, deviceSubclass, deviceProtocol)) {  
 **return false**;  
}  
  
**synchronized** (**mLock**) {  
 **if** (**mDevices**.get(deviceName) != **null**) {  
 Slog.*w*(***TAG***, **"device already on mDevices list: "** + deviceName);  
 **return false**;  
 }  
  
 **if** (**mNewDevice** != **null**) {  
 Slog.*e*(***TAG***, **"mNewDevice is not null in endUsbDeviceAdded"**);  
 **return false**;  
 }  
  
 *// Create version string in "%.%" format* String versionString = Integer.*toString*(version >> 8) + **"."** + (version & 0xFF);  
  
 **mNewDevice** = **new** UsbDevice(deviceName, vendorID, productID,  
 deviceClass, deviceSubclass, deviceProtocol,  
 manufacturerName, productName, versionString, serialNumber);  
  
 **mNewConfigurations** = **new** ArrayList<UsbConfiguration>();  
 **mNewInterfaces** = **new** ArrayList<UsbInterface>();  
 **mNewEndpoints** = **new** ArrayList<UsbEndpoint>();  
}  
  
**return true**;

}

### UsbHostManager .addUsbConfiguration

**private void** addUsbConfiguration(**int** id, String name, **int** attributes, **int** maxPower) {  
 **if** (**mNewConfiguration** != **null**) {  
 **mNewConfiguration**.setInterfaces(  
 **mNewInterfaces**.toArray(**new** UsbInterface[**mNewInterfaces**.size()]));  
 **mNewInterfaces**.clear();  
 }  
  
 **mNewConfiguration** = **new** UsbConfiguration(id, name, attributes, maxPower);  
 **mNewConfigurations**.add(**mNewConfiguration**);  
}

### UsbHostManager .addUsbInterface

**private void** addUsbInterface(**int** id, String name, **int** altSetting,  
 **int** Class, **int** subClass, **int** protocol) {  
 **if** (**mNewInterface** != **null**) {  
 **mNewInterface**.setEndpoints(  
 **mNewEndpoints**.toArray(**new** UsbEndpoint[**mNewEndpoints**.size()]));  
 **mNewEndpoints**.clear();  
 }  
  
 **mNewInterface** = **new** UsbInterface(id, altSetting, name, Class, subClass, protocol);  
 **mNewInterfaces**.add(**mNewInterface**);  
}

### addUsbEndpoint

*/\* Called from JNI in monitorUsbHostBus() to report new USB endpoint for the device  
 currently being added. Returns true if successful, false in case of error.  
 \*/***private void** addUsbEndpoint(**int** address, **int** attributes, **int** maxPacketSize, **int** interval) {  
 **mNewEndpoints**.add(**new** UsbEndpoint(address, attributes, maxPacketSize, interval));  
}

### endUsbDeviceAdded

*/\* Called from JNI in monitorUsbHostBus() to finish adding a new device \*/***private void** endUsbDeviceAdded() {  
 **if** (**mNewInterface** != **null**) {  
 **mNewInterface**.setEndpoints(  
 **mNewEndpoints**.toArray(**new** UsbEndpoint[**mNewEndpoints**.size()]));  
 }  
 **if** (**mNewConfiguration** != **null**) {  
 **mNewConfiguration**.setInterfaces(  
 **mNewInterfaces**.toArray(**new** UsbInterface[**mNewInterfaces**.size()]));  
 }  
  
  
 **synchronized** (**mLock**) {  
 **if** (**mNewDevice** != **null**) {  
 **mNewDevice**.setConfigurations(  
 **mNewConfigurations**.toArray(  
 **new** UsbConfiguration[**mNewConfigurations**.size()]));  
 **mDevices**.put(**mNewDevice**.getDeviceName(), **mNewDevice**);  
   
 *// It is fine to call this only for the current user as all broadcasts are sent to  
 // all profiles of the user and the dialogs should only show once.* ComponentName usbDeviceConnectionHandler = getUsbDeviceConnectionHandler();  
 **if** (usbDeviceConnectionHandler == **null**) {  
  **getCurrentUserSettings().deviceAttac**hed(**mNewDevice**);  
 } **else** {  
 getCurrentUserSettings().deviceAttachedForFixedHandler(**mNewDevice**,  
 usbDeviceConnectionHandler);  
 }  
 **mUsbAlsaManager**.usbDeviceAdded(**mNewDevice**);  
 } **else** {  
 Slog.*e*(***TAG***, **"mNewDevice is null in endUsbDeviceAdded"**);  
 }  
 **mNewDevice** = **null**;  
 **mNewConfigurations** = **null**;  
 **mNewInterfaces** = **null**;  
 **mNewEndpoints** = **null**;  
 **mNewConfiguration** = **null**;  
 **mNewInterface** = **null**;  
 }  
}

主要是获取usb设备的属性,然后调用UsbHostManager.java中的usbDeviceAdded（），并将这些usb属性传递上去

### UsbSettingsManager.deviceAttached

deviceAttached（）函数，主要是检查系统中是否有安装能处理UsbManager.ACTION\_USB\_DEVICE\_ATTACHED这个广播的activity，并转到该activity.

|  |
| --- |
| public void deviceAttached(UsbDevice device) {  Intent intent = new Intent(UsbManager.ACTION\_USB\_DEVICE\_ATTACHED); |
| intent.putExtra(UsbManager.EXTRA\_DEVICE, device); |
| intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK); |
|  |
| ArrayList<ResolveInfo> matches; |
| String defaultPackage; |
| synchronized (mLock) { |
| matches = getDeviceMatchesLocked(device, intent); |
| // Launch our default activity directly, if we have one. |
| // Otherwise we will start the UsbResolverActivity to allow the user to choose. |
| defaultPackage = mDevicePreferenceMap.get(new DeviceFilter(device)); |
| } |
|  |
| // Send broadcast to running activity with registered intent |
| mUserContext.sendBroadcast(intent); |
|  |
| // Start activity with registered intent |
| resolveActivity(intent, matches, defaultPackage, device, null); |
| } |

## openDevice

而当用户进程通过UsbManger获取到设备名后，可以通过openDevice来获取连接，从而收发数据

内部实现：先通过设备名字打开得到一个文件描述符，然后利用new一个连接来关联这个描述符，最终把连接返回给用户

*/\*\*  
 \* Opens the device so it can be used to send and receive  
 \* data using {****@link*** *android.hardware.usb.UsbRequest}.  
 \*  
 \** ***@param device*** *the device to open  
 \** ***@return*** *a {****@link*** *UsbDeviceConnection}, or {****@code*** *null} if open failed  
 \*/***public** UsbDeviceConnection openDevice(UsbDevice device) {  
 **try** {  
 String deviceName = device.getDeviceName();  
 ParcelFileDescriptor pfd = **mService**.openDevice(deviceName);  
 **if** (pfd != **null**) {  
 UsbDeviceConnection connection = **new** UsbDeviceConnection(device);  
 **boolean** result = connection.open(deviceName, pfd, **mContext**);  
 pfd.close();  
 **if** (result) {  
 **return** connection;  
 }  
 }  
 } **catch** (Exception e) {  
 Log.*e*(***TAG***, **"exception in UsbManager.openDevice"**, e);  
 }  
 **return null**;  
}

### openDevice(String deviceName)

ParcelFileDescriptor 作为参数调用这个UsbDeviceConnection的open函数

*/\* Opens the specified USB device \*/***public** ParcelFileDescriptor openDevice(String deviceName) {  
 **synchronized** (**mLock**) {  
 **if** (isBlackListed(deviceName)) {  
 **throw new** SecurityException(**"USB device is on a restricted bus"**);  
 }  
 UsbDevice device = **mDevices**.get(deviceName);  
 **if** (device == **null**) {  
 *// if it is not in mDevices, it either does not exist or is blacklisted* **throw new** IllegalArgumentException(  
 **"device "** + deviceName + **" does not exist or is restricted"**);  
 }  
 getCurrentSettings().checkPermission(device);  
 **return** nativeOpenDevice(deviceName);  
 }  
}

最后也是调用了JNI函数nativeOpenDevice

### Jni. openDevice

android\_server\_UsbHostManager\_openDevice就是对应的JNI函数，这里主要调用了usb\_device\_open函数，并且返回一个usb\_device，最后我们通过usb\_device来获取其fd，并且把它封装在Java层ParcelFileDescriptor类中

**static** jobject android\_server\_UsbHostManager\_openDevice(JNIEnv \*env, jobject */\* thiz \*/*,  
 jstring deviceName)  
{  
 **const char** \*deviceNameStr = env->GetStringUTFChars(deviceName, NULL);  
 **struct** usb\_device\* device = usb\_device\_open(deviceNameStr);  
 env->ReleaseStringUTFChars(deviceName, deviceNameStr);  
  
 **if** (!device)  
 **return** NULL;  
  
 **int** fd = **usb\_device\_get\_fd(device);**  
 **if** (fd < 0) {  
 usb\_device\_close(device);  
 **return** NULL;  
 }  
 **int** newFD = dup(fd);  
 usb\_device\_close(device);  
  
 jobject fileDescriptor = jniCreateFileDescriptor(env, newFD);  
 **if** (fileDescriptor == NULL) {  
 **return** NULL;  
 }  
 **return** env->NewObject(gParcelFileDescriptorOffsets.mClass,  
 gParcelFileDescriptorOffsets.mConstructor, fileDescriptor);  
}

#### Libusb.usb\_device\_get\_fd

**int** usb\_device\_get\_fd(**struct** usb\_device \*device)  
{  
 **if** (!usb\_device\_reopen\_writeable(device))  
 **return** -1;  
 **return** device->fd;  
}

#### usb\_device\_reopen\_writeable

**static int** usb\_device\_reopen\_writeable(**struct** usb\_device \*device)  
{  
 **if** (device->writeable)  
 **return** 1;  
  
 **int** fd = open(device->dev\_name, O\_RDWR);  
 **if** (fd >= 0) {  
 close(device->fd);  
 device->fd = fd;  
 device->writeable = 1;  
 **return** 1;  
 }  
 D(**"usb\_device\_reopen\_writeable failed errno %d\n"**, errno);  
 **return** 0;  
}

### connection.open

**boolean** open(String name, ParcelFileDescriptor pfd, @NonNull Context context) {  
 **mContext** = context.getApplicationContext();  
  
 **return** native\_open(name, pfd.getFileDescriptor());  
}

#### native\_open

frameworks/base/core/jni/android\_hardware\_UsbDeviceConnection.cpp

**static** jboolean  
android\_hardware\_UsbDeviceConnection\_open(JNIEnv \*env, jobject thiz, jstring deviceName,  
 jobject fileDescriptor)  
{  
 **int** fd = jniGetFDFromFileDescriptor(env, fileDescriptor);  
 *// duplicate the file descriptor, since ParcelFileDescriptor will eventually close its copy* fd = dup(fd);  
 **if** (fd < 0)  
 **return** JNI\_FALSE;  
  
 **const char** \*deviceNameStr = env->GetStringUTFChars(deviceName, NULL);  
 **struct** usb\_device\* device = **usb\_device\_new**(deviceNameStr, fd);  
 **if** (device) {  
 env->SetLongField(thiz, field\_context, (jlong)device);  
 } **else** {  
 ALOGE(**"usb\_device\_open failed for %s"**, deviceNameStr);  
 close(fd);  
 }  
  
 env->ReleaseStringUTFChars(deviceName, deviceNameStr);  
 **return** (device != NULL) ? JNI\_TRUE : JNI\_FALSE;  
}

#### libusb.usb\_device\_new

**struct** usb\_device \*usb\_device\_new(**const char** \*dev\_name, **int** fd)  
{  
 **struct** usb\_device \*device = calloc(1, **sizeof**(**struct** usb\_device));  
 **int** length;  
  
 D(**"usb\_device\_new %s fd: %d\n"**, dev\_name, fd);  
  
 **if** (lseek(fd, 0, SEEK\_SET) != 0)  
 **goto** failed;  
 length = read(fd, device->desc, **sizeof**(device->desc));  
 D(**"usb\_device\_new read returned %d errno %d\n"**, length, errno);  
 **if** (length < 0)  
 **goto** failed;  
  
 strncpy(device->dev\_name, dev\_name, **sizeof**(device->dev\_name) - 1);  
 device->fd = fd;  
 device->desc\_length = length;  
 *// assume we are writeable, since usb\_device\_get\_fd will only return writeable fds* device->writeable = 1;  
 **return** device;  
  
failed:  
 close(fd);  
 free(device);  
 **return** NULL;  
}

---------------------

作者：kc专栏

来源：CSDN

原文：https://blog.csdn.net/kc58236582/article/details/54691334

版权声明：本文为博主原创文章，转载请附上博文链接！

## 其他方法

我们来看USBManager中关于获取Usb设备，以及使用它们的函数

### getDeviceList

通过Bundle对象来传输的

**public void** getDeviceList(Bundle devices) {  
 **synchronized** (**mLock**) {  
 **for** (String name : **mDevices**.keySet()) {  
 devices.putParcelable(name, **mDevices**.get(name));  
 }  
 }  
}

getDeviceList可以得到一个设备名，以及对应的UsbDevice。

## API

类 介绍

UsbManager 获取连接的USB设备并与之通信

UsbDevice 代表一个连接的USB 设备，包含一系列方法获取自身信息，包括interfaces，endpoints

UsbInterface 代表USB 设备上定义的一系列功能接口，一个usb设备可以有一个或多个接口

UsbEndpoint 代表一个interface通信频道，一个interface可以有一个或多个endpoints，一般含有输入输出两个端点来支持双工通信

UsbDeviceConnection 代表设备连接的一个链路，将数据传输到端点上，这个类允许你同步或异步的来回发送数据

UsbRequest 代表一个异步请求，通过UsbDeviceConnection来跟设备通信

UsbConstants 定义了linux内核文件linux/usb/ch9.h中的常量

## 关API

在绝大部分情况下，当你需要跟USB通信时，你需要用到这些类（UsbRequest类仅在异步通信时用到）。一般的，获取UsbManager找到目标UsbDevice，接着找到恰当的UsbInterface和这个interface的UsbEndpoint，得到了正确的UsbEndpoint后，打开设备获得UsbDeviceConnection来跟USB 设备通信。

作者：yqyzxd  
链接：https://www.jianshu.com/p/e2e57cddac6a  
来源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

## AdbTest

demo：\development\samples\USB\AdbTest

### 配置

<**activity android:name="AdbTestActivity" android:label="ADB Test"**>  
 <**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"** />  
 <**category android:name="android.intent.category.DEFAULT"** />  
 <**category android:name="android.intent.category.LAUNCHER"** />  
 </**intent-filter**>  
  
 <**intent-filter**>  
 <**action android:name="android.hardware.usb.action.USB\_DEVICE\_ATTACHED"** />  
 </**intent-filter**>  
  
 <**meta-data android:name="android.hardware.usb.action.USB\_DEVICE\_ATTACHED"  
 android:resource="@xml/device\_filter"** />  
</**activity**>

<**resources**>  
 <**usb-device class="255" subclass="66" protocol="1"** />  
</**resources**>

### mUsbReceiver

*// listen for new devices*IntentFilter filter = **new** IntentFilter();  
filter.addAction(UsbManager.***ACTION\_USB\_DEVICE\_ATTACHED***);  
filter.addAction(UsbManager.***ACTION\_USB\_DEVICE\_DETACHED***);  
registerReceiver(**mUsbReceiver**, filter);

BroadcastReceiver **mUsbReceiver** = **new** BroadcastReceiver() {  
 **public void** onReceive(Context context, Intent intent) {  
 String action = intent.getAction();  
  
 **if** (UsbManager.***ACTION\_USB\_DEVICE\_ATTACHED***.equals(action)) {  
 UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.***EXTRA\_DEVICE***);  
 UsbInterface intf = *findAdbInterface*(device);  
 **if** (intf != **null**) {  
 log(**"Found adb interface "** + intf);  
 setAdbInterface(device, intf);  
 }  
 } **else if** (UsbManager.***ACTION\_USB\_DEVICE\_DETACHED***.equals(action)) {  
 UsbDevice device = intent.getParcelableExtra(UsbManager.***EXTRA\_DEVICE***);  
 String deviceName = device.getDeviceName();  
 **if** (**mDevice** != **null** && **mDevice**.equals(deviceName)) {  
 log(**"adb interface removed"**);  
 setAdbInterface(**null**, **null**);  
 }  
 }  
 }  
};

### 交互

#### findAdbInterface

*// searches for an adb interface on the given USB device***static private** UsbInterface findAdbInterface(UsbDevice device) {  
 Log.*d*(***TAG***, **"findAdbInterface "** + device);  
 **int** count = device.getInterfaceCount();  
 **for** (**int** i = 0; i < count; i++) {  
 UsbInterface intf = device.getInterface(i);  
 **if** (i**ntf.getInterfaceClass() == 255 && intf.getInterfaceSubclass() == 66 &&  
 intf.getInterfaceProtocol() == 1**) {  
 **return** intf;  
 }  
 }  
 **return null**;  
}

哈哈，原来adb能用，是因为约定了过滤配置器啊255

#### setAdbInterface

**private boolean** setAdbInterface(UsbDevice device, UsbInterface intf) {  
 **if** (**mDeviceConnection** != **null**) {  
 **if** (**mInterface** != **null**) {  
 **mDeviceConnection**.releaseInterface(**mInterface**);  
 **mInterface** = **null**;  
 }  
 **mDeviceConnection**.close();  
 **mDevice** = **null**;  
 **mDeviceConnection** = **null**;  
 }  
  
 **if** (device != **null** && intf != **null**) {  
 UsbDeviceConnection connection = **mManager**.openDevice(device);  
 **if** (connection != **null**) {  
 log(**"open succeeded"**);  
 **if** (connection.claimInterface(intf, **false**)) {  
 log(**"claim interface succeeded"**);  
 **mDevice** = device;  
 **mDeviceConnection** = connection;  
 **mInterface** = intf;  
 **mAdbDevice** = **new** AdbDevice(**this**, **mDeviceConnection**, intf);  
 log(**"call start"**);  
 **mAdbDevice**.start();  
 **return true**;  
 } **else** {  
 log(**"claim interface failed"**);  
 connection.close();  
 }  
 } **else** {  
 log(**"open failed"**);  
 }  
 }  
  
 **if** (**mDeviceConnection** == **null** && **mAdbDevice** != **null**) {  
 **mAdbDevice**.stop();  
 **mAdbDevice** = **null**;  
 }  
 **return false**;  
}

### AdbDevice

#### 配置输入输出点

**public** AdbDevice(AdbTestActivity activity, UsbDeviceConnection connection,  
 UsbInterface intf) {  
 **mDeviceConnection** = connection;  
 **mSerial** = connection.getSerial();  
  
 UsbEndpoint epOut = **null**;  
 UsbEndpoint epIn = **null**;  
 *// look for our bulk endpoints* **for** (**int** i = 0; i < intf.getEndpointCount(); i++) {  
 UsbEndpoint ep = intf.getEndpoint(i);  
 **if** (ep.getType() == UsbConstants.***USB\_ENDPOINT\_XFER\_BULK***) {  
 **if** (ep.getDirection() == UsbConstants.***USB\_DIR\_OUT***) {  
 epOut = ep;  
 } **else** {  
 epIn = ep;  
 }  
 }  
 }  
   
 **mEndpointOut** = epOut;  
 **mEndpointIn** = epIn;  
}

start()

**public void** start() {  
 **mWaiterThread**.start();  
 connect();  
}

*// send a connect command***private void** connect() {  
 AdbMessage message = **new** AdbMessage();  
 message.set(AdbMessage.***A\_CNXN***, AdbMessage.***A\_VERSION***, AdbMessage.***MAX\_PAYLOAD***, **"host::\0"**);  
 message.write(**this**);  
}

#### WaiterThread

@Override  
**public void** run() {  
 *// start out with a command read* BulkMessage currentCommand = **new** BulkMessage();  
 BulkMessage currentData = **null**;  
 *//* ***FIXME error checking***

*//封装了usbRequest.setClientData和queue，请求读adb协议的command header数据段*currentCommand.readCommand(getInRequest());  
  
 **while** (**true**) {  
 log(**"runing"**);  
 **synchronized** (**this**) {  
 **if** (**mStop**) {  
 **return**;  
 }  
 }

*//阻塞等待request被响应，*  
 UsbRequest request = **mDeviceConnection**.requestWait();  
 **if** (request == **null**) {  
 **break**;  
 }  
  
 BulkMessage message = (BulkMessage) request.getClientData();  
 request.setClientData(**null**);  
 BulkMessage messageToDispatch = **null**;  
  
 **if** (message == currentCommand) {  
 **int** dataLength = message.getDataLength();  
 *// read data if length > 0* **if** (dataLength > 0) {

**//请求读adb协议的command data数据段**  
 message.readData(getInRequest(), dataLength);  
 currentData = message;  
 } **else** {  
 messageToDispatch = message;  
 }  
 currentCommand = **null**;

*//只需关注自己提交的、而不是其他endpoint的 command data request*  
 } **else if** (message == currentData) {  
 messageToDispatch = message;  
 currentData = **null**;  
 }  
*//command data字段 准备解析并处理*  
 **if** (messageToDispatch != **null**) {  
 *// queue another read first*

*//先提交一次read command header, 为下一次接收做准备* currentCommand = **new** BulkMessage();  
 currentCommand.readCommand(getInRequest());  
  
 *// then dispatch the current message* dispatchMessage(messageToDispatch);  
 }  
  
 *// put request back into the appropriate pool*

*//in or out request用完了 重新放回List中；* **if** (request.getEndpoint() == **mEndpointOut**) {  
 releaseOutRequest(request);  
 } **else** {  
 **synchronized** (**mInRequestPool**) {  
 **mInRequestPool**.add(request);  
 }  
 }  
 }  
}

### dispatchMessage

*// dispatch a message from the device***void** dispatchMessage(BulkMessage message) {  
 **int** command = message.getCommand();  
 **switch** (command) {  
 **case** BulkMessage.A\_SYNC:  
 log(**"got A\_SYNC"**); *//收到同步command*  
 **break**;  
 **case** BulkMessage.A\_CNXN: *//收到connect response，处理下*  
 handleConnect(message);  
 **break**;  
 **case** BulkMessage.A\_OPEN:  
 **case** BulkMessage.A\_OKAY:  
 **case** BulkMessage.A\_CLSE:  
 **case** BulkMessage.A\_WRTE:  
 AdbSocket socket = getSocket(message.getArg1());*//收到ok response*  
 **if** (socket == **null**) {  
 log(**"ERROR socket not found"**);  
 } **else** {

*//device往host write,即 host接收数据,这里的数据就是logcat的消息；*   
 socket.handleMessage(message);  
 }  
 **break**;  
 }  
}

### 协议

## Dump

**public void** dump(IndentingPrintWriter pw) {  
 **synchronized** (**mLock**) {  
 pw.println(**"USB Host State:"**);  
 **for** (String name : **mDevices**.keySet()) {  
 pw.println(**" "** + name + **": "** + **mDevices**.get(name));  
 }  
 }  
}

---------------------

作者：kc专栏

来源：CSDN

原文：https://blog.csdn.net/kc58236582/article/details/54691334

版权声明：本文为博主原创文章，转载请附上博文链接！

## [libusbhost](https://www.jianshu.com/p/b267c5cedfa9)

libusbhost主要提供与usb设备通信的接口

struct usb\_device \*usb\_device\_open(const char \*dev\_name)

---打开一个usb设备，在/dev/bus/usb/下

void usb\_device\_close(struct usb\_device \*device)

void usb\_descriptor\_iter\_init(struct usb\_device \*device, struct usb\_descriptor\_iter \*iter)

struct usb\_descriptor\_header \*usb\_descriptor\_iter\_next(struct usb\_descriptor\_iter \*iter)

--获取descriptor

int usb\_device\_claim\_interface(struct usb\_device \*device, unsigned int interface)

----claim一个interface用于通讯

int usb\_device\_release\_interface(struct usb\_device \*device, unsigned int interface)

int usb\_device\_bulk\_transfer(struct usb\_device \*device, --------传输数据

int endpoint,

void\* buffer,

int length,

unsigned int timeout)

int usb\_device\_control\_transfer(struct usb\_device \*device, ----------控制指令

int requestType,

int request,

int value,

int index,

void\* buffer,

int length,

unsigned int timeout)

在java代码中可以通过一下文件中提供的接口来访问usb设备。

frameworks/base/core/java/android/hardware/usb/UsbDeviceConnection.java

## 定制方案

**mHostBlacklist**

**mHostBlacklist** = context.getResources().getStringArray(  
 com.android.internal.R.array.config\_usbHostBlacklist);

## 参考

android usb host 读写USB设备

<https://blog.csdn.net/u013815973/article/details/24501583>

Android Usb在framework的相关源码分析

<https://www.jianshu.com/p/51809085e9cc>

android USB通信

<https://www.jianshu.com/p/e2e57cddac6a>

adbdevice介绍

<https://blog.csdn.net/anribras/article/details/50833269>

adb 通信协议

<https://blog.csdn.net/evilcode/article/details/6904200>

# Host-Serial

java->jni->Serial驱动，虽然也可以正常读写，这样做缺点是

1、如果突然串口不能正常通信可能会导致app崩溃，

2、这个串口只能被当前的app使用，其他app 不能使用。

3、违背正常Android设计初衷，Android标准流程是

java->service->jni->hal->serial驱动。

所以我在项目中使用了Android源码中的demo

会获取目前存在可以使用串口列表。这个根据自己需要添加。

我的就使用一个串口3

<string-array translatable="false" name="config\_serialPorts">

<item>"/dev/ttyS3"</item>

</string-array>

## SerialChat

frameworks\base\tests\SerialChat\AndroidManifest.xml

<**uses-permission android:name="android.permission.SERIAL\_PORT"**/>

**mSerialManager** = (SerialManager)getSystemService(Context.***SERIAL\_SERVICE***);

### 打开和关闭

@Override  
**public void** onResume() {  
 **super**.onResume();  
  
 String[] ports = **mSerialManager**.getSerialPorts();  
 **if** (ports != **null** && ports.**length** > 0) {  
 **try** {  
 **mSerialPort** = **mSerialManager**.openSerialPort(ports[0], 115200);  
 **if** (**mSerialPort** != **null**) {  
 **new** Thread(**this**).start();  
 }  
 } **catch** (IOException e) {  
 }  
 }  
  
}

@Override  
**public void** onDestroy() {  
 **if** (**mSerialPort** != **null**) {  
 **try** {  
 **mSerialPort**.close();  
 } **catch** (IOException e) {  
 }  
 **mSerialPort** = **null**;  
 }  
 **super**.onDestroy();  
}

### 发送和接收

**mOutputBuffer**.put(bytes);  
**mSerialPort**.write(**mOutputBuffer**, bytes.**length**);

**public void** run() {  
 Log.*d*(***TAG***, **"run"**);  
 **int** ret = 0;  
 **byte**[] buffer = **new byte**[1024];  
 **while** (ret >= 0) {  
 **try** {  
 Log.*d*(***TAG***, **"calling read"**);  
 **mInputBuffer**.clear();  
 ret = **mSerialPort**.read(**mInputBuffer**);  
 Log.*d*(***TAG***, **"read returned "** + ret);  
 **mInputBuffer**.get(buffer, 0, ret);  
 } **catch** (IOException e) {  
 Log.*e*(***TAG***, **"read failed"**, e);  
 **break**;  
 }  
  
 **if** (ret > 0) {   
 String text = **new** String(buffer, 0, ret);  
 Log.*d*(***TAG***, **"chat: "** + text);  
   
 }  
 }  
 Log.*d*(***TAG***, **"thread out"**);  
}

## 参考

<https://blog.csdn.net/lb5761311/article/details/80618834>

Android串口Serial服务解析

Android USB转串口通信

<https://www.jianshu.com/p/524d1009c069>

<https://github.com/HelloHuDi/usb-with-serial-port>

<https://www.wanandroid.com/blog/show/2239>

<https://blog.csdn.net/lnc2003/article/details/80606524>

<http://www.zhimengzhe.com/Androidkaifa/378520.html>

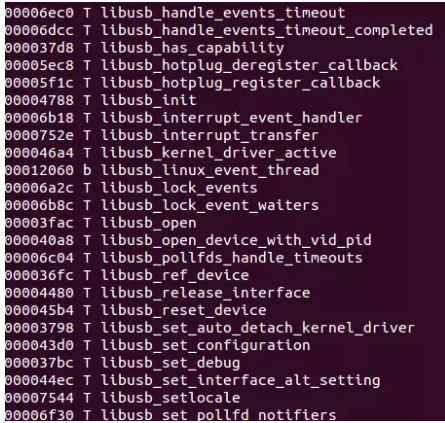
https://www.cnblogs.com/hackfun/p/7683512.html

**libusb**

android系统已经一致了

external.libusb

ibusb1.0.x主要语法



## 编译

<https://github.com/libusb/libusb>

# 六.参考

<http://scottmaxiao.github.io/AOA.html>

<https://source.android.com/devices/accessories/protocol>

<https://developer.android.com/guide/topics/connectivity/usb/accessory>

<https://source.android.com/devices/accessories/custom>

<https://blog.csdn.net/gaojinshan/article/details/12012363>

<https://blog.csdn.net/yingzhao80/article/details/45511351>

# app demo

原生demo

相关实例：

frameworks/base/libs/usb/tests/AccessoryChat/AndroidManifest.xml

\frameworks\base\tests\AccessoryDisplay\source\src\com\android\accessorydisplay\source\SourceActivity.java

cts\apps\CtsVerifier\src\com\android\cts\verifier\usb\UsbAccessoryTestActivity.java

<**activity android:name=".usb.UsbAccessoryTestActivity"  
 android:label="@string/usb\_accessory\_test"  
 android:configChanges="keyboardHidden|orientation|screenSize"  
 android:launchMode="singleTop"**>

<**activity  
 android:name=".main.activity.DJIAoaActivity"  
 android:configChanges="orientation|screenSize|keyboardHidden|keyboard"** >  
 <**intent-filter**>  
 <**action android:name="android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED"** />  
 </**intent-filter**>  
  
 <**meta-data  
 android:name="android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED"  
 android:resource="@xml/accessory\_filter"** />  
</**activity**>

<**resources**>  
 <**usb-accessory manufacturer="Google, Inc." model="AccessoryChat" type="Sample Program" version="1.0"** />  
</**resources**>

*<?***xml version="1.0" encoding="utf-8"***?>*<**resources**>  
 <**usb-accessory model="T600" manufacturer="DJI"**/>  
</**resources**>

# dumpsys usb

dumpsys usb

# 实战问题

对端传入null，导致系统异常

# AOA透传方案

**setprop sys.usb.config** accessory**,adb**

# Usb Debug启动流程

代码在/framework/base/service/com/android/server/usb/UsbService.java里去初始化一个叫UsbDeviceManager的类，相关文件也在同一个目录。

在UsbService.java中

public UsbService(Context context) {

mContext = context;

final PackageManager pm = mContext.getPackageManger();

....

if (new File("/sys/class/android\_usb").exists())

{

mDeviceManager = new UsbDeviceManager();

}

...

}

中的new UsbDeviceManager()就是关键代码。我的项目中由于/sys/class目录下没有android\_usb所以UsbDeviceManager初始化函数没有执行，所以会无法启动ADB。刚才/sys/class/android\_usb应该是一个和驱动有关的目录，这里不做详细介绍。　为什么UsbDeviceManager能够启动ADB呢？额。。。那是因为他监听了settings的数据库，然后做了响应动作。我们来看下UsbDeviceManager.java中做了神马事情。

private class AdbSettingsObserver extends ContentObserver {

...

@Override

boolean enable = (Settings.Global.getInt(mContextResolver, Settings.Global.ADB\_ENABLED, 0) > 0)

mHandler.sendMessage(MSG\_ENABLE\_ADB, enable);

}

而后续代码中注册了数据库的监听mContentResolver.registerContentResolver(Settings.Global.getUriFor(Settings.Global.ADB\_ENABLED), false, new AdbSettingsObserver);

来监听最上面提到的Settings.Global\_ADB\_ENABLED这个数据库字段的改变来产生相应动作。

具体动作就是对android properties属性进行设置，关键字段是persist.sys.usb.config，如果启动ADB，则通过SystemProperties.get("persisit.sys.usb.config", "adb");来启动ADB。

你可以手动通过setprop在终端中设置相应的值先试试看看。前提是你必须有root权限。

## Ref

[USB各种模式 解释](http://www.cnblogs.com/zxc2man/p/5610812.html)

[Android的USB系统简单分析之一](https://www.jianshu.com/p/b267c5cedfa9)

[Linux下使用USB模拟ACM串口设备](http://blog.csdn.net/mapeng892020/article/details/54095037)

[Android Ethernet从上至下解析一](http://www.bkjia.com/Androidjc/1045168.html)

[android 5.1 Ethernet开发相关](https://www.jianshu.com/p/b52cda6c0b58)

[android5.1 增加ethernet设置(DHCP与Static ip)-UI修改](http://blog.csdn.net/hclydao/article/details/50972932)