Kernal 与 Framework层交互 UEventObserver；插入与拔出USB设备，事件监听以及上报.UEventObserver

**涉及到的类文件**：

* android\_os\_UEventObserver.cpp  
  ./frameworks/base/core/jni/android\_os\_UEventObserver.cpp
* UEventObserver.java  
  ./frameworks/base/core/java/android/os/UEventObserver.java

# Todo

Uevent原理

# UEventObserver

**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***);

## startObservering

//this参数，当onUEvent回调时，则回调到注册的这个Observer中。

**public final void** startObserving(String match) {  
 **if** (match == **null** || match.isEmpty()) {  
 **throw new** IllegalArgumentException(**"match substring must be non-empty"**);  
 }  
  
 **final** UEventThread t = *getThread*();  
 t.addObserver(match, **this**);  
}

接下来，startObserving方法转到了UEventThread中。

## UEventThread

UEventThread是个线程是UEventObserver的内部类，run方法中是个死循环，不断地监听UEvent，当有事件从Kernal上报时，则通过Handler一步步上传。

UEventThread 在UEventObserver中以单例模式存在

### addObserver

**public void** addObserver(String match, UEventObserver observer) {  
 **synchronized** (**mKeysAndObservers**) {  
 **mKeysAndObservers**.add(match);  
 **mKeysAndObservers**.add(observer);  
 *nativeAddMatch*(match);  
 }  
}

### run()

@Override  
**public void** run() {  
 *nativeSetup*();  
 **while** (**true**) {  
 String message = *nativeWaitForNextEvent*();  
 **if** (message != **null**) {  
 sendEvent(message);  
 }  
 }  
}

### nativeWaitForNextEvent

framework/base/core/jni/android\_os\_UEventObserver.cpp

### sendEvent

**private void** sendEvent(String message) {  
 **synchronized** (**mKeysAndObservers**) {  
 **final int** N = **mKeysAndObservers**.size();  
 **for** (**int** i = 0; i < N; i += 2) {  
 **final** String key = (String)**mKeysAndObservers**.get(i);  
 **if** (message.contains(key)) {  
 **final** UEventObserver observer =  
 (UEventObserver)**mKeysAndObservers**.get(i + 1);  
 **mTempObserversToSignal**.add(observer);  
 }  
 }  
 }  
  
 **if** (!**mTempObserversToSignal**.isEmpty()) {  
 **final** UEvent event = **new** UEvent(message);  
 **final int** N = **mTempObserversToSignal**.size();  
 **for** (**int** i = 0; i < N; i++) {  
 **final** UEventObserver observer = **mTempObserversToSignal**.get(i);  
 observer.onUEvent(event);  
 }  
 **mTempObserversToSignal**.clear();  
 }  
}

在UEventThread中，有两个ArrayList，一个是mKeysAndObservers，虽然是个List但是它扮演的是Map的角色，里面的match (String) , Observer(Object) 一一对应，这里就有疑问了？为何不直接使用map呢？map不正好是key-value对应的吗？ 我的认为是，list里面match (String) 与Obsever的对应是存在重复的情况。但是map中是不允许key重复的。这应该是没有使用Map的原因吧。另外一个是mTempObserversToSignal ，作为临时变量

### Jni