android-uevent 简记

2017-02-09 | 分类 驱动杂记

简述

sysfs 是 Linux userspace 和 kernel 进行交互的一个媒介。通过 sysfs，userspace 可以主动去读写 kernel 的一些数据，同样的， kernel 也可以主动将一些“变化”告知给 userspace。也就是说，通过sysfs，userspace 和 kernel 的交互，本质上是双向的。

userspace 通过 sysfs 访问 kernel 数据的方法，便是大名鼎鼎的 show() / store() 方法：只要在 kernel 提供了对应的 show() / store() 方法，用户便可以通过 shell 用户，cd 进入到相应的目录，使用 cat / echo 操作对应的文件节点即可。而 kernel ，通过 sysfs 将一些 kernel 的“变化”“告知”给 userspace 则是通过 uevent 的方式。

一般来说，Kernel 会发送一个字符串给 userspace，然后 userspace 来解析处理该字符串，比如android7.0上 HDMI 热插拔的 event 字符串： change@/devices/virtual/switch/hdmi 。 该字符串的路径为 “/sys/devices/virtual/switch/hdmi/change” ，届时 userspace 的监听线程去读取该文件节点即可。

在 Linux-3.x 上是基于 NetLink 来实现的。其实现思路是，首先在内核中调用 `netlink_kernel_create()` 函数创建一个socket套接字；当有事件发生的时候，则通过 `kobject_uevent()` 最终调用 `netlink_broadcast_filtered()` 向 userspace 发送数据。如果同时在 userspace ，有在监听该事件，则两相一合，kernel 的“变化”，userspace 即刻知晓。

Kernel

kernel ，关于 uevent 的实现代码，大约可参考文件 `kobject_uevent.c` ，其简要调用如下：

```
kobject_uevent(&drv->p->kobj, KOBJ_ADD);
kobject_uevent_env(kobj, action, NULL);
retval = netlink_broadcast_filtered(uevent_sock, skb,0, 1, GFP_KERNEL,kobj_bcast_filter,kobj);
```

其中，`kobject_uevent(struct kobject *kobj, enum kobject_action action)` 中的 action 对应着以下几种：

```
  KOBJ_ADD,
  KOBJ_REMOVE,
  KOBJ_CHANGE,
  KOBJ_MOVE,
  KOBJ_ONLINE,
  KOBJ_OFFLINE,
```

而 `kobject_uevent()` 其实就是直接调用了 `kobject_uevent_env()` 函数。一切的操作，将在该函数中完成，比如 kset uevent ops (struct kset_uevent_ops)的获取、字符串的填充组合、netlink message 的发送等。

其中， kset_uevent_ops 有以下几种：

```
  slab_uevent_ops
  bus_uevent_ops
  device_uevent_ops
  gfs2_uevent_ops
  module_uevent_ops
```

这些 uevent ops 在 `start_kernel()` 就会被注册。

Userspace

此处仅记述 android 的学习，理论上，非 android 的实现原理应该也是一样的。 android 实现则是按照 android 的体系架构，java 文件通过 jni 到 hal 层来实现的 userspace 监听。

Android

在高通平台的 android 7.0 版本上，Android java 提供了一个 `UEventObserver` 类。在该类中有一个事件线程 `UEventThread`，该线程中将重新实现了一个 `run()` 方法：

```
@Override
    public void run() {
```

```
        nativeSetup();

        while (true) {
            String message = nativeWaitForNextEvent();
            if (message != null) {
                if (DEBUG) {
                    Log.d(TAG, message);
                }
                sendEvent(message);
            }
        }
    }
```

其中的 `nativeSetup()` 和 `nativeWaitForNextEvent()` 即是通过 JNI 来实现的： `nativeSetup()` 创立绑定 socket 套接字；`nativeWaitForNextEvent()` 则是通过调用`recv()`函数监听套接字事件。

那么如何在 android-java 使用该类呢？下面以 BatteryService 为例：

```
public BatteryService(Context context) {
    // watch for invalid charger messages if the invalid_charger switch exists
    if (new File("/sys/devices/virtual/switch/invalid_charger/state").exists()) {
        UEventObserver invalidChargerObserver = new UEventObserver() {
            @Override
            public void onUEvent(UEvent event) {
                final int invalidCharger = "1".equals(event.get("SWITCH_STATE")) ? 1 : 0;
                synchronized (mLock) {
                    if (mInvalidCharger != invalidCharger) {
                        mInvalidCharger = invalidCharger;
                    }
                }
            }
        };
        invalidChargerObserver.startObserving(
                "DEVPATH=/devices/virtual/switch/invalid_charger");
    }
}
```

第一步： new 一个 UEventObserver()；
第二步： startObserving()；

其发生的调用过程如下：

```
  startObserving()
  addObserver()
  nativeAddMatch()
```

`nativeAddMatch()` 依然是通过 JNI 来实现的，其目的是为了将 `startObserving()` 的参数增加到匹配序列中，当内核发送具有该参数的数据时，就返回匹配成功，然后调用 BatteryService 的onUEvent函数。
以上函数，大约可参考文件 UEventObserver.java 和 BatteryService.java。

JNI

在 JNI 层为 uevent 提供了4个封装：

```
static void nativeSetup(JNIEnv *env, jclass clazz);
static jstring nativeWaitForNextEvent(JNIEnv *env, jclass clazz);
static void nativeAddMatch(JNIEnv* env, jclass clazz, jstring matchStr);
static void nativeRemoveMatch(JNIEnv* env, jclass clazz, jstring matchStr);
```

`nativeSetup()` 调用 `uevent_init()`创建绑定套接字；
`nativeWaitForNextEvent()` 调用 `uevent_next_event()` 循环接收套接字数据；
`nativeAddMatch()` 添加需要监听的字符串到 `gMatches` 全局变量；
`nativeRemoveMatch()` 做 `nativeAddMatch` 逆操作；

以上内容，在android7.0 上可参考文件 android_os_UEventObserver.cpp

HAL

在 HAL 层，最主要的就是以下两个函数：

```
int uevent_init();
int uevent_next_event(char* buffer, int buffer_length);
```

它们详细实现如下：

```
int uevent_init()
{
    struct sockaddr_nl addr;
    int sz = 64*1024;
    int s;

    memset(&addr, 0, sizeof(addr));
    addr.nl_family = AF_NETLINK;
    addr.nl_pid = getpid();
    addr.nl_groups = 0xffffffff;

    s = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);
    if(s < 0)
        return 0;

    setsockopt(s, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz));

    if(bind(s, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
        close(s);
        return 0;
    }

    fd = s;
    return (fd > 0);
}
```

以上函数就是Linux编程中最基本的套接字操作；

```
int uevent_next_event(char* buffer, int buffer_length)
{
    while (1) {
        struct pollfd fds;
        int nr;

        fds.fd = fd;
        fds.events = POLLIN;
        fds.revents = 0;
        nr = poll(&fds, 1, -1);

        if(nr > 0 && (fds.revents & POLLIN)) {
            int count = recv(fd, buffer, buffer_length, 0);
            if (count > 0) {
                struct uevent_handler *h;
                pthread_mutex_lock(&uevent_handler_list_lock);
                LIST_FOREACH(h, &uevent_handler_list, list)
                    h->handler(h->handler_data, buffer, buffer_length);
                pthread_mutex_unlock(&uevent_handler_list_lock);

                return count;
            }
        }
    }

    // won't get here
    return 0;
}
```

以上的实现中，采用了 `poll()` 函数 + `recv()` 函数的方式实现了对事件的监听。其中， `poll()` 和 `select()` 类似，在一定的条件下可以互相替用； `recv()` 相当于 `read()` 函数，其有阻塞和非阻塞两种用法。是否阻塞，需要使用函数 `setsockopt()` 来设置

套接字的属性。

以上内容，在 android 7.0 上可以参看 hardware/ 目录下的 uevent.c 文件。

优化

从网上看到了一点资料，说是在 uevent 这个部分还可以优化的。基本的思路就是，把收不到的和永远不会使用到的 uevent 去掉，不让它在 kernel 发出来。相关文章链接如下： Udev 内核机制(kobject_uevent) 性能优化

over

上一篇 下一篇