Linux 设备模型浅析之 uevent 篇

Linux 设备模型，仅仅看理论介绍，比如 LDD3 的第十四章，会感觉太抽象不易理解，而通过阅读内核代码就更具体更易理解，所以结合理论介绍和内核代码阅读能够更快速的理解掌握 linux 设备模型。这一序列的文章的目的就是在于此，看这些文章之前最好能够仔细阅读 LDD3 的第十四章。uevent，即 user space event，就是内核向用户空间发出的一个事件通知，使得应用程序能有机会对该 event 作出反应，udev 及 mdev(busybox)就是这种应用程序。阅读这篇文章之前，最好先阅读文章《Linux 设备模型浅析之设备篇》和《Linux 设备模型浅析之驱动篇》。

一、在《Linux 设备模型浅析之设备篇》中介绍过 device_add()例程，其用于将一个 device 注册到 device model，其中调用了 **kobject_uevent**(&dev->kobj, KOBJ_ADD)例程向用户空间发出 KOBJ_ADD 事件并输出环境变量，以表明一个 device 被添加了。在《Linux 设备模型浅析之设备篇》中介绍过 rtc_device_register()例程 ，其最终调用 device_add()例程添加了一个 rtc0 的 device，我们就以它为例子来完成 uevent 的分析。让我们看看 kobject_uevent()这个例程的代码，如下：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action)
{
        return kobject_uevent_env(kobj, action, NULL);
}
```

它又调用了 **kobject_uevent_env()**例程，部分代码如下：

```
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,
                       char *envp_ext[])
{
        struct kobj_uevent_env *env;
        const char *action_string = kobject_actions[action];  // 本例是“add”命令
        const char *devpath = NULL;
        const char *subsystem;
        struct kobject *top_kobj;
        struct kset *kset;
        struct kset_uevent_ops *uevent_ops;
        u64 seq;
        int i = 0;
        int retval = 0;

        pr_debug("kobject: '%s' (%p): %s\n",
                 kobject_name(kobj), kobj, __func__);

        /* search the kset we belong to */
        top_kobj = kobj;
        /* 找到其所属的 kset 容器，如果没找到就从其父 kobj 找，一直持续下去，直到父 kobj 不存在 */
        while (!top_kobj->kset && top_kobj->parent)
```

```
        top_kobj = top_kobj->parent;

if (!top_kobj->kset) {
        pr_debug("kobject: '%s' (%p): %s: attempted to send uevent "
                 "without kset!\n", kobject_name(kobj), kobj,
                 __func__);
        return -EINVAL;
}

/* 在本例中是 devices_kset 容器，详细介绍可参照《Linux 设备模型浅析之设备篇》，后面将列出 devices_kset 的定义 */
kset = top_kobj->kset;
uevent_ops = kset->uevent_ops;       // 本例中 uevent_ops = &device_uevent_ops

/* 回调 uevent_ops->filter ()例程，本例中是 dev_uevent_filter()例程，主要是检查是否 uevent suppress，后面分析 */
/* skip the event, if the filter returns zero. */
if (uevent_ops && uevent_ops->filter)
        if (!uevent_ops->filter(kset, kobj)) { // 如果不成功，即 uevent suppress，则直接返回
                pr_debug("kobject: '%s' (%p): %s: filter function "
                         "caused the event to drop!\n",
                         kobject_name(kobj), kobj, __func__);
                return 0;
        }

/* 回调 uevent_ops-> name ()，本例中是 dev_uevent_name()例程，获取 bus 或 class 的名字，本例中 rtc0 不存在 bus，所以是 class 的名字“rtc”，后面分析 */
/* originating subsystem */
if (uevent_ops && uevent_ops->name)
        subsystem = uevent_ops->name(kset, kobj);
else
        subsystem = kobject_name(&kset->kobj);
if (!subsystem) {
        pr_debug("kobject: '%s' (%p): %s: unset subsystem caused the "
                 "event to drop!\n", kobject_name(kobj), kobj,
                 __func__);
        return 0;
}

// 获得用于存放环境变量的 buffer
/* environment buffer */
env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);
if (!env)
        return -ENOMEM;

/* 获取该 kobj 在 sysfs 的路径，通过遍历其父 kobj 来获得，本例是/sys/devices/platform/s3c2410-rtc/rtc/rtc0 */
/* complete object path */
```

```
devpath = kobject_get_path(kobj, GFP_KERNEL);
if (!devpath) {
        retval = -ENOENT;
        goto exit;
}

// 添加 ACTION 环境变量，本例是“add”命令
/* default keys */
retval = add_uevent_var(env, "ACTION=%s", action_string);
if (retval)
        goto exit;
// 添加 DEVPATH 环境变量，本例是/sys/devices/platform/s3c2410-rtc/rtc/rtc0
retval = add_uevent_var(env, "DEVPATH=%s", devpath);
if (retval)
        goto exit;

// 添加 SUBSYSTEM 环境变量，本例中是“rtc”
retval = add_uevent_var(env, "SUBSYSTEM=%s", subsystem);
if (retval)
        goto exit;

/* keys passed in from the caller */
if (envp_ext) {          // 为 NULL，不执行
        for (i = 0; envp_ext[i]; i++) {
                retval = add_uevent_var(env, "%s", envp_ext[i]);
                if (retval)
                        goto exit;
        }
}

// 回调 uevent_ops->uevent()，本例中是 dev_uevent()例程，输出一些环境变量，后面分析
/* let the kset specific function add its stuff */
if (uevent_ops && uevent_ops->uevent) {
        retval = uevent_ops->uevent(kset, kobj, env);
        if (retval) {
                pr_debug("kobject: '%s' (%p): %s: uevent() returned "
                         "%d\n", kobject_name(kobj), kobj,
                         __func__, retval);
                goto exit;
        }
}

/*
 * Mark "add" and "remove" events in the object to ensure proper
 * events to userspace during automatic cleanup. If the object did
 * send an "add" event, "remove" will automatically generated by
 * the core, if not already done by the caller.
 */
if (action == KOBJ_ADD)
```

```
        kobj->state_add_uevent_sent = 1;
    else if (action == KOBJ_REMOVE)
        kobj->state_remove_uevent_sent = 1;

    /* 增加 event 序列号的值，并输出到环境变量的 buffer。该系列号可以从/sys/kernel/
    uevent_seqnum 属性文件读取，至于 uevent_seqnum 属性文件及/sys/kernel/目录是怎样产
    生的，后面会分析 */
    /* we will send an event, so request a new sequence number */
    spin_lock(&sequence_lock);
    seq = ++uevent_seqnum;
    spin_unlock(&sequence_lock);
    retval = add_uevent_var(env, "SEQNUM=%llu", (unsigned long long)seq);
    if (retval)
        goto exit;

    /* 如果配置了网络，那么就会通过 netlink socket 向用户空间发送环境标量，而用户空间
    则通过 netlink socket 接收，然后采取一些列的动作。这种机制目前用在 udev 中，也就是
    pc 机系统中，后面会分析*/
#if defined(CONFIG_NET)
    /* send netlink message */
    /* 如果配置了 net，则会在 kobject_uevent_init()例程中将全局比昂俩 uevent_sock 初试化
    为 NETLINK_KOBJECT_UEVENT 类型的 socket。*/
    if (uevent_sock) {
        struct sk_buff *skb;
        size_t len;

        /* allocate message with the maximum possible size */
        len = strlen(action_string) + strlen(devpath) + 2;
        skb = alloc_skb(len + env->buflen, GFP_KERNEL);
        if (skb) {
            char *scratch;

            /* add header */
            scratch = skb_put(skb, len);
            sprintf(scratch, "%s@%s", action_string, devpath);

            /* copy keys to our continuous event payload buffer */
            for (i = 0; i < env->envp_idx; i++) {
                len = strlen(env->envp[i]) + 1;
                scratch = skb_put(skb, len);
                strcpy(scratch, env->envp[i]);
            }

            NETLINK_CB(skb).dst_group = 1;
            retval = netlink_broadcast(uevent_sock, skb, 0, 1,
                                       GFP_KERNEL); // 广播
        } else
            retval = -ENOMEM;
    }
```

```
#endif
        /* 对于嵌入式系统来说，busybox 采用的是 mdev，在系统启动脚本 rcS 中会使用 echo /
        sbin/mdev > /proc/sys/kernel/hotplug 命令，而这个 hotplug 文件通过一定的方法映射到了 u
        event_helper[]数组，所以 uevent_helper[] = “/sbin/mdev” 。所以对于采用 busybox 的嵌
        入式系统来说会执行里面的代码，而 pc 机不会。也就是说内核会 call 用户空间
        的/sbin/mdev 这个应用程序来做动作，后面分析 */
        /* call uevent_helper, usually only enabled during early boot */
        if (uevent_helper[0]) {
                char *argv [3];

                // 加入到环境变量 buffer
                argv [0] = uevent_helper;
                argv [1] = (char *)subsystem;
                argv [2] = NULL;
                retval = add_uevent_var(env, "HOME=/");
                if (retval)
                        goto exit;
                retval = add_uevent_var(env,
                                        "PATH=/sbin:/bin:/usr/sbin:/usr/bin");
                if (retval)
                        goto exit;

                // 呼叫应用程序来处理， UMH_WAIT_EXEC 表明等待应用程序处理完
                retval = call_usermodehelper(argv[0], argv,
                                             env->envp, UMH_WAIT_EXEC);
        }

exit:
        kfree(devpath);
        kfree(env);
        return retval;
}
```

代码中，

1. devices_kset 容器指针定义在 drivers/base/core.c 中，在同文件里的 devices_init()例程中初始化，devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL)。显然初始化后 devices_kset ->uevent_ops = &device_uevent_ops。而 device_uevent_ops 结构体定义如下：

```
static struct kset_uevent_ops device_uevent_ops = {
        .filter =       dev_uevent_filter,
        .name =         dev_uevent_name,
        .uevent =       dev_uevent,
}
```

通过该结构体的定义，就可以知道上面分析的一些回调例程的出处了。

2. dev_uevent_filter()例程可以让程序发送 uevent 事件之前做些检查和过滤，然后再决定是否发送 uevent 事件，其代码定义如下：

```
static int dev_uevent_filter(struct kset *kset, struct kobject *kobj)
{
        struct kobj_type *ktype = get_ktype(kobj);
```

```
        if (ktype == &device_ktype) {          // 做个检测
                struct device *dev = to_dev(kobj);
                if (dev->uevent_suppressha) //  可以通过设置这个变量为 1 来阻止发送 uevent 事件
                        return 0;
                if (dev->bus)
                        return 1;
                if (dev->class) //  bus 或 class 如果都没设置，那么也不会发送 uevent 事件
                        return 1;
        }
        return 0;
}
```

3. dev_uevent_name()例程用于获取 bus 或 class 的 name，bus 优先，其代码定义如下：

```
static const char *dev_uevent_name(struct kset *kset, struct kobject *kobj)
{
        struct device *dev = to_dev(kobj);

        if (dev->bus)  // 如果设置了 bus，则返回 bus 的 name
                return dev->bus->name;
        if (dev->class) // 如果没有设置 bus 而设置了 class ，则返回 class 的 name
                return dev->class->name;
        return NULL; // 否则，返回 NULL
}
```

4. dev_uevent()例程用于输出一定的环境变量，其代码定义如下：

```
static int dev_uevent(struct kset *kset, struct kobject *kobj,
                      struct kobj_uevent_env *env)
{
        struct device *dev = to_dev(kobj);
        int retval = 0;

        /* add the major/minor if present */
        if (MAJOR(dev->devt)) {     // 本例中 rtc0 有 devt，所以会输出下面的环境变量
                add_uevent_var(env, "MAJOR=%u", MAJOR(dev->devt));
                add_uevent_var(env, "MINOR=%u", MINOR(dev->devt));
        }

        // 本例中没有设置 type
        if (dev->type && dev->type->name)
                add_uevent_var(env, "DEVTYPE=%s", dev->type->name);

        // 本例中没有设置 driver
        if (dev->driver)
                add_uevent_var(env, "DRIVER=%s", dev->driver->name);

        /* have the bus specific function add its stuff */
        /* 本例中没有设置 bus ，若果设置了 platform_bus_type，则调用 platform_uevent()例程，
        可参考 platform.c */
        if (dev->bus && dev->bus->uevent) {
                retval = dev->bus->uevent(dev, env);
```

```
        if (retval)
                pr_debug("device: '%s': %s: bus uevent() returned %d\n",
                         dev_name(dev), __func__, retval);
    }

    // 本例中设置了 class ，是 rtc class，但是没有实现 class->dev_uevent()例程
    /* have the class specific function add its stuff */
    if (dev->class && dev->class->dev_uevent) {
        retval = dev->class->dev_uevent(dev, env);
        if (retval)
                pr_debug("device: '%s': %s: class uevent() "
                         "returned %d\n", dev_name(dev),
                         __func__, retval);
    }

    /* 本例中没有设置 type
    /* have the device type specific fuction add its stuff */
    if (dev->type && dev->type->uevent) {
        retval = dev->type->uevent(dev, env);
        if (retval)
                pr_debug("device: '%s': %s: dev_type uevent() "
                         "returned %d\n", dev_name(dev),
                         __func__, retval);
    }

    return retval;
}
```

下面开始就分析/sys/kernel 目录如何生成的，以及该目录下有些什么文件夹和文件。

二、在 kernel/ksysfs.c 的 **ksysfs_init()**例程中初始化了全局指针 kernel_kobj，并生成/sys/kernel 目录，代码如下：

```
static int __init ksysfs_init(void)
{
    int error;

    // 获得一个 kobj，无 parent，故生成/sys/kernel 目录
    kernel_kobj = kobject_create_and_add("kernel", NULL);
    if (!kernel_kobj) {
        error = -ENOMEM;
        goto exit;
    }
    // 在/sys/kernel 目录下生成一些属性文件（包含在属性组里）
    error = sysfs_create_group(kernel_kobj, &kernel_attr_group);
    if (error)
        goto kset_exit;

    if (notes_size > 0) {
        notes_attr.size = notes_size;
```

```
                // 生成/sys/kernel/notes 二进制属性文件，可用来读取二进制 kernel .notes section
                error = sysfs_create_bin_file(kernel_kobj, &notes_attr);
                if (error)
                        goto group_exit;
        }

        /* 生成/sys/kernel/uids 目录和/sys/kernel/uids/0 目录以及/sys/kernel/uids/0/cpu_share 属性文
        件，后两者都是针对 root_user 的 */
        /* create the /sys/kernel/uids/ directory */
        error = uids_sysfs_init();
        if (error)
                goto notes_exit;

        return 0;

notes_exit:
        if (notes_size > 0)
                sysfs_remove_bin_file(kernel_kobj, &notes_attr);
group_exit:
        sysfs_remove_group(kernel_kobj, &kernel_attr_group);
kset_exit:
        kobject_put(kernel_kobj);
exit:
        return error;
}
```

代码中，

1. kernel_attr_group 的定义如下：

```
static struct attribute_group kernel_attr_group = {
        .attrs = kernel_attrs,
}
```

而 kernel_attrs 的代码如下：

```
static struct attribute * kernel_attrs[] = {
#if defined(CONFIG_HOTPLUG)
        &uevent_seqnum_attr.attr,    // 这个之前提过，用于获取 event 序列号
        &uevent_helper_attr.attr,    // 这个之前也提过，用于存取用户提供的程序
#endif
#ifdef CONFIG_PROFILING
        &profiling_attr.attr,
#endif
#ifdef CONFIG_KEXEC
        &kexec_loaded_attr.attr,
        &kexec_crash_loaded_attr.attr,
        &vmcoreinfo_attr.attr,
#endif
        NULL
}
```

2. uevent_seqnum_attr 是通过 KERNEL_ATTR_RO(uevent_seqnum)定义的，也就是说，生成为 /sys/kernel/uevent_seqnum 可读的名属性文件，读取的方法是 uevent_seqnum_show()例程，其

代码如下：

```
static ssize_t uevent_seqnum_show(struct kobject *kobj,
                                  struct kobj_attribute *attr, char *buf)
{
        // 将 uevent_seqnum 全局标量的值读取到 buf，最终输出到用户空间
        return sprintf(buf, "%llu\n", (unsigned long long)uevent_seqnum);
}
```

3. uevent_helper_attr 是通过 KERNEL_ATTR_RW(uevent_helper)定义的，也就是说，生成名为 /sys/kernel/uevent_helper 可读写的属性文件，其作用与/proc/sys/kernel/hotplug 相同，最终是读写 uevent_helper[]数组。读写的方法分别是 uevent_helper_show()和 uevent_helper_store()例程，前者的代码如下：

```
static ssize_t uevent_helper_show(struct kobject *kobj,
                                  struct kobj_attribute *attr, char *buf)
{
        // 将数组 uevent_helper[]中的字符读取到 buf，最终输出到用户空间
        return sprintf(buf, "%s\n", uevent_helper);
}
```

后者的代码如下：

```
static ssize_t uevent_helper_store(struct kobject *kobj,
                                   struct kobj_attribute *attr,
                                   const char *buf, size_t count)
{
        if (count+1 > UEVENT_HELPER_PATH_LEN) // 作个判读，字符串长度不能超标
                return -ENOENT;
        memcpy(uevent_helper, buf, count);  // 拷贝用户传过来的字符串到 uevent_helper 数组
        uevent_helper[count] = '\0';
        if (count && uevent_helper[count-1] == '\n')
                uevent_helper[count-1] = '\0';
        return count;
}
```

三、通过前面的分析可知，要实现 hotplug 机制，需要有用户空间的程序配合才行。对于 pc 机的 linux 系统，采用的是 udevd 服务程序，其通过监听 NETLINK_KOBJECT_UEVENT 获得内核发出的 uevent 事件和环境变量，然后再查找匹配的 udev rules，根据找到的 rules 做动作，udev 的具体实现原理可参照网上的一些文章。在《Linux 设备模型浅析之设备篇》中讲过，在每个注册的 device 文件夹下会生成一个 uevent 属性文件，其作用就是实现手动触发 hotplug 机制。可以向其中写入“add”和“remove”等命令，以添加和移除设备。在系统启动后注册了很多 device，但用户空间还没启动，所以这些事件并没有处理，udevd 服务启动后，会扫描/sys 目录里所有的 uevent 属性文件，向其写入"add”命令，从而触发 uevent 事，这样 udevd 服务程序就有机会处理这些事件了。在嵌入式系统中使用的是 mdev，是 udev 的简化版本，在启动脚本 rcS 中会有这样一句命令/sbin/mdev -s，其作用就是刚刚讲到的，扫描/sys 目录里所有的 uevent 属性文件，向其写入"add”命令，触发 uevent 事件，从而 mdev 有机会处理这些事件。

从上面的分析可以看出，每当内核注册设备或驱动时都会产生 uevent 事件，这样用户空间的 udev 或 mdev 就有机会捕捉到这些事件，根据匹配的规则作一定的处理，比如在/dev 目录下生成设备节点或使用 modprobe 加载驱动程序，等等。从而实现自动生成设备节点、加载驱动程序等等这些热拔插机制。