Vold框架分析

吴观平

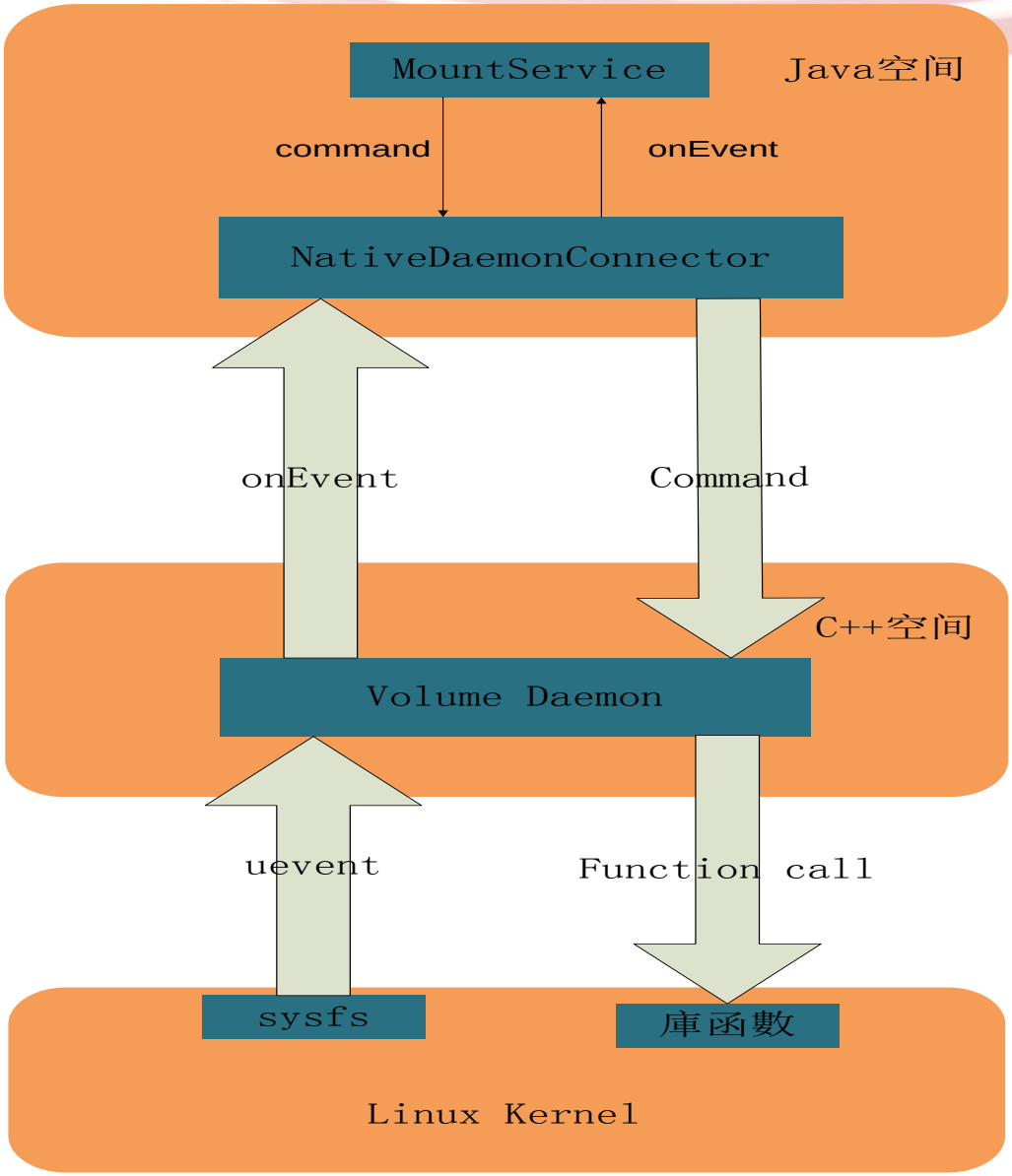
Vold 简介

- Volume Daemon
- Linux 标准udev
- 基于内核sysfs
- 监听uevent

Vold 功能

- 内核态与用户态相互沟通守护进程

  Android 的volume服务主要是用来管理usb/sd卡等外部存储设备。平台可以对外部存储设备进行操作和轮询状态，当外部存储设备状态发生变化时，volume 服务也会实时报告平台


MountService
Java空间
command
onEvent
NativeDaemonConnector
onEvent
Command
C++空间
Volume Daemon
uevent
Function call
sysfs
庫函数
Linux Kernel

Vold 启动

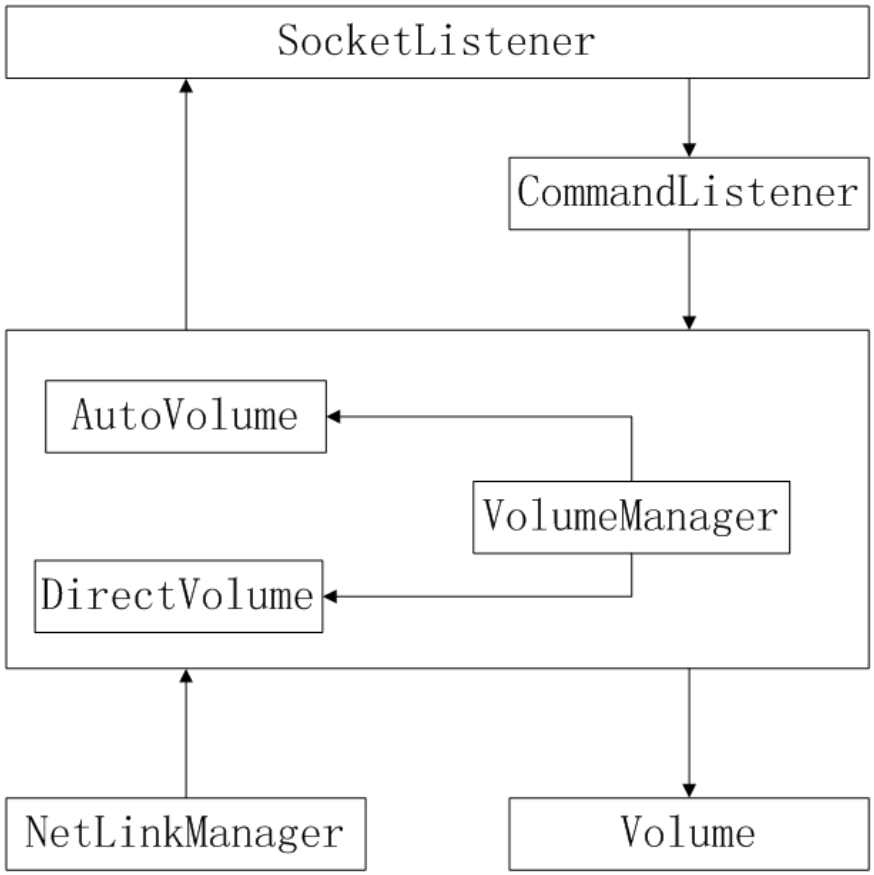
init.rc中配置了volume daemon,

```
service vold /system/bin/vold
    class core
    socket vold stream 0660 root mount
    ioprio be 2
```

这里启动vold,并且创建一个socket,用来与Framework层中的服务通信。

Vold 内部架构

内部流程大致分为：创建连结、引导和事件处理。
内部框架图如下：



创建连结

在vold作为一个守护进程，一方面接受驱动的信息，并把信息传给应用层；另一方面接受上层的命令并完成相应。所以这里的连结一共有两条：

1)vold socket: 负责vold与framework层的信息传递；

```
/*
 * Create the connection to vold with a maximum queue of twice the
 * amount of containers we'd ever expect to have. This keeps an
 * "asec list" from blocking a thread repeatedly.
 */
mConnector = new NativeDaemonConnector(this, "vold", MAX_CONTAINERS * 2, VOLD_TAG, 25);
```

2)sysfs uevent的socket: 接收kernel driver中发出的信息；

```
/* Create our singleton managers */
if (!(vm = VolumeManager::Instance())) {
    SLOGE("Unable to create VolumeManager");
    exit(1);
};

if (!(nm = NetlinkManager::Instance())) {
    SLOGE("Unable to create NetlinkManager");
    exit(1);
};


cl = new CommandListener();
vm->setBroadcaster((SocketListener *) cl);
nm->setBroadcaster((SocketListener *) cl);
```

引导

vold启动时，对现有外设存储设备的处理。

首先这里需要载入vold.fstab，并解析该文件

dev_mount sdcard /storage/sdcard0 auto /devices/platform/msm_sdcc.1/mmc_host

dev_mount命令：	dev_mount
<label>标签	sdcard
<mount_point>挂载点：	/storage/sdcard0
<part>子分区个数：	auto
<sysfs_path1...>设备在sysfs文件系统下的路径(可多个)：	/devices/platform/msm_sdcc.1/mmc_host

需要注意的是：

1)子分区的数目可以为auto，表示只有一个子分区。子分区数目也可以为任意大于0的一个整数。

2)个参数间不能有空格，应该以tab制表符为参数的间隔，原因是android对vold.fstab的解析是以”\t”为标识，从而得到各个参数。

事件处理

这里通过对两个连结的监听，完成对动态事件的处理，以及对上层应用操作的响应。

我们来具体分析一下代码过程。

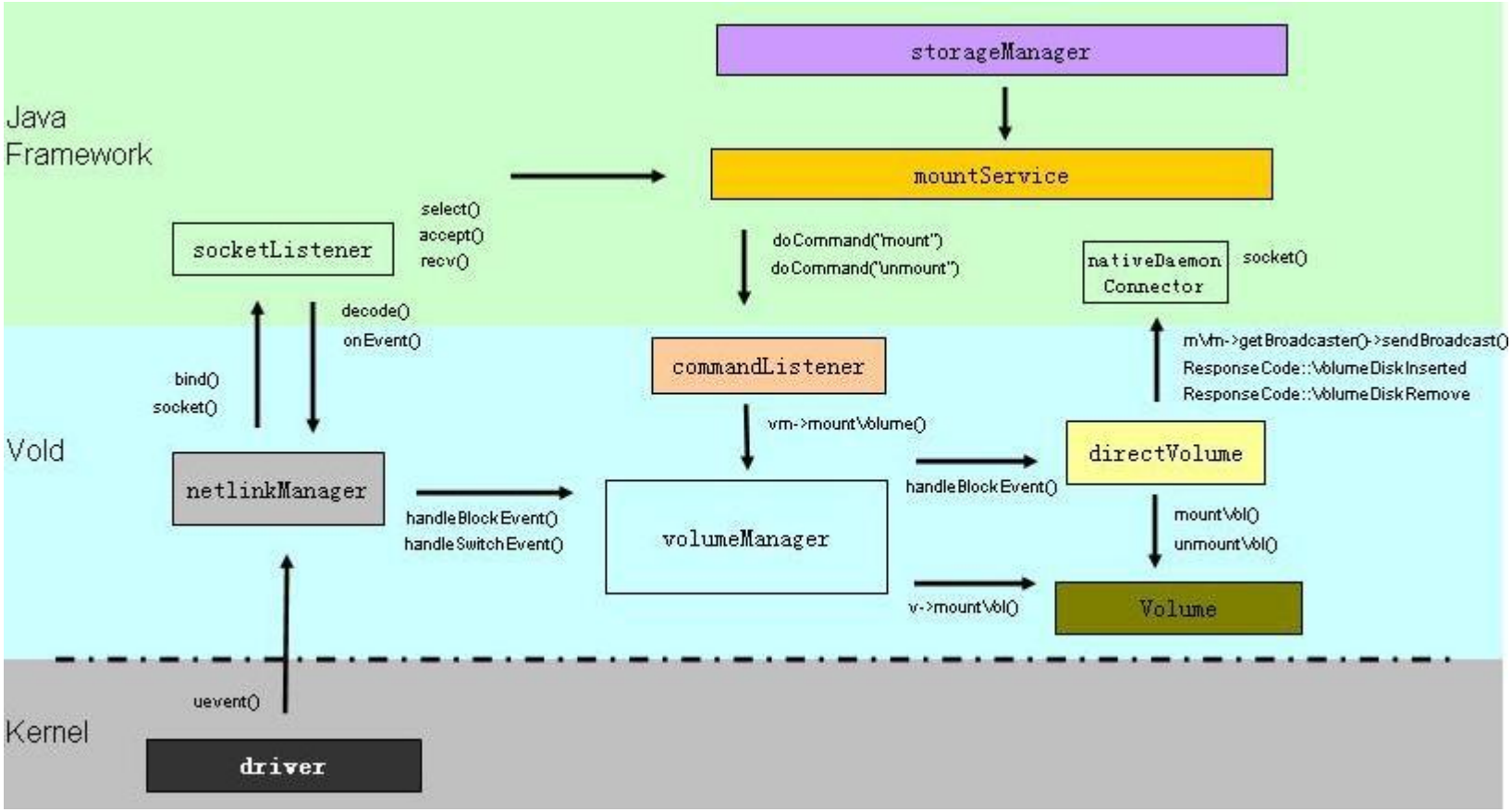
1）kernel发出uevent

2）framework层发出命令

总体架构

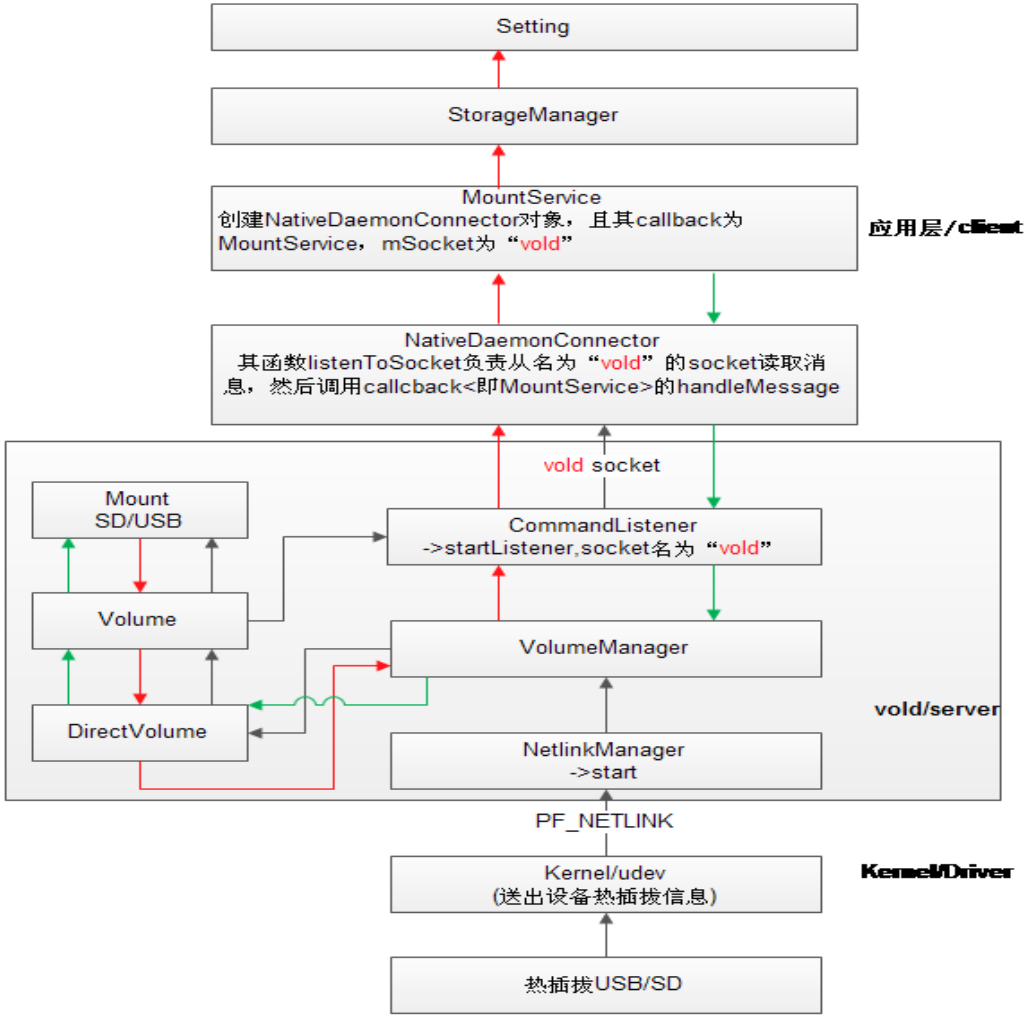
Vold服务由volumeManager统一管控，它将具体任务分别分派给netlinkManager, commandListener, directVolume, Volume去完成。

Vold服务向下通过socket机制与底层驱动交互，向上通过JNI, intent, socket, doCommand等机制与Java Framework交互。


Java
Framework
Vold
Kernel
storageManager
mountService
socketListener
select()
accept()
recv()
doCommand("mount")
doCommand("unmount")
nativeDaemon
Connector
socket()
decode()
onEvent()
bind()
socket()
commandListener
m\vm->getBroadcaster()->sendBroadcast()
ResponseCode::VolumeDiskInserted
ResponseCode::VolumeDiskRemove
vm->mount\volume()
directVolume
netlinkManager
handleBlockEvent()
handleSwitchEvent()
volumeManager
handleBlockEvent()
mount\vol()
unmount\vol()
v->mount\vol()
Volume
uevent()
driver

Android热插拔事件处理流程图

Android热插拔事件处理流程如下图所示：



黑色箭头：表示插入SD/USB后事件传递以及SD/USB挂载
红色箭头：表示挂载成功后的消息传递流程
绿色箭头：表示MountService发出挂载/卸载设备的命令

组成

1. NetlinkManager:
2. VolumeManager:
3. DirectVolume:
4. Volume:
5. CommandListener:
6. NativeDaemonConnector:
7. MountService:
8. StorageManaer:

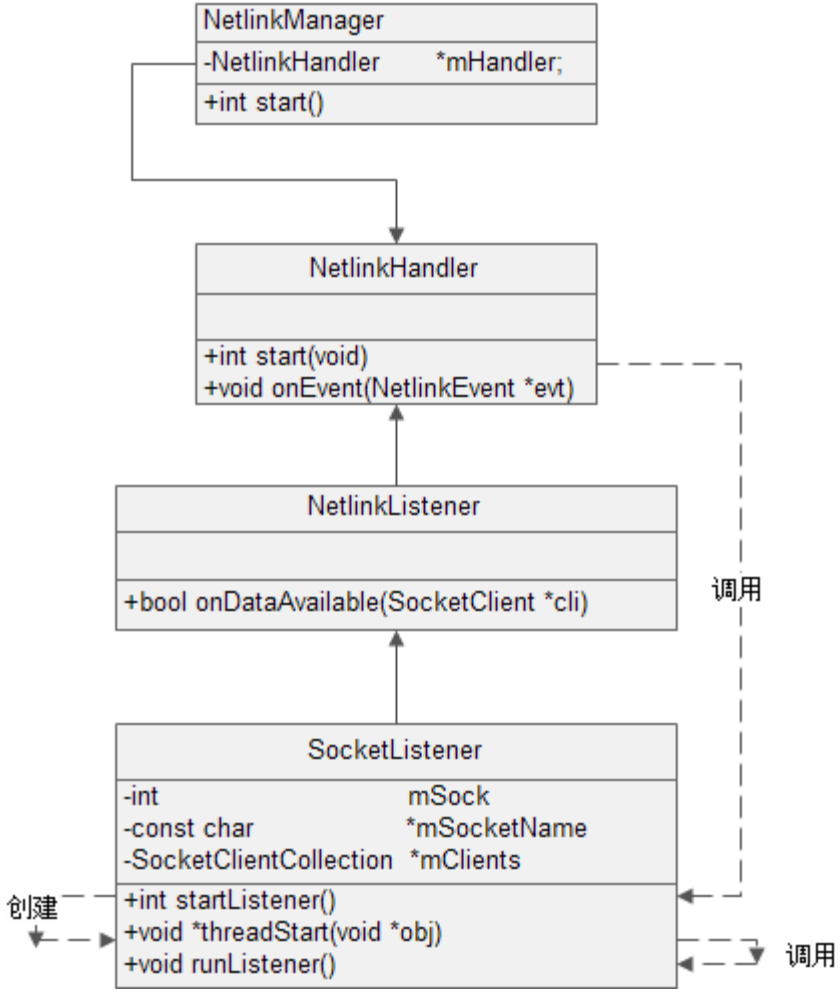
Vold用户态

NetlinkManager

NetlinkManager负责与Kernel交互，通过PF_NETLINK来现。

Vlod启动代码如下(/system/vold/main.cpp)

NetlinkManager的家族关系如下所示：



与Vold相关的Kernel态

用户态创建的netlink sock被kernel保存在：nl_table[sk->sk_protocol].mc_list

Kernel态创建的netlink sock被kernel保存在：uevent_sock_list，上面的sk->sk_protocol为uevent_sock_list的协议，二者只有协议一致才可以发送。

在用户态的socket创建方式（/system/vold/NetlinkManager.cpp）：

```
if ((mSock = socket(PF_NETLINK,

SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0)
{
    SLOGE("Unable to create uevent socket: %s",
strerror(errno));
    return -1;
}
```

在Kernel的socket创建方式(/kernel/lib/kobject_uevent.c)：

```
static int uevent_net_init(struct net *net)
{
        struct uevent_sock *ue_sk;

        ue_sk = kzalloc(sizeof(*ue_sk), GFP_KERNEL);
        if (!ue_sk)
           return -ENOMEM;

        ue_sk->sk = netlink_kernel_create(net, NETLINK_KOBJECT_UEVENT,
                                          1, NULL, NULL, THIS_MODULE);
        if (!ue_sk->sk) {
           printk(KERN_ERR
                 "kobject_uevent: unable to create netlink socket!\n");
           kfree(ue_sk);
           return -ENODEV;
        }
        mutex_lock(&uevent_sock_mutex);
        list_add_tail(&ue_sk->list, &uevent_sock_list);
        mutex_unlock(&uevent_sock_mutex);
        return 0;
}
```

此sock被创建之后，被增加到全局变量uevent_sock_list列表中，下面的分析围绕此列表进行。

Thanks

for more information, please visit:
http://www.thundersoft.com/

contact us:
biz@thundersoft.com
+86-10-62662686

Address:
4th floor, Taixiang Building 1A#, Longxiang Road, Haidian District Beijing, China, 100191