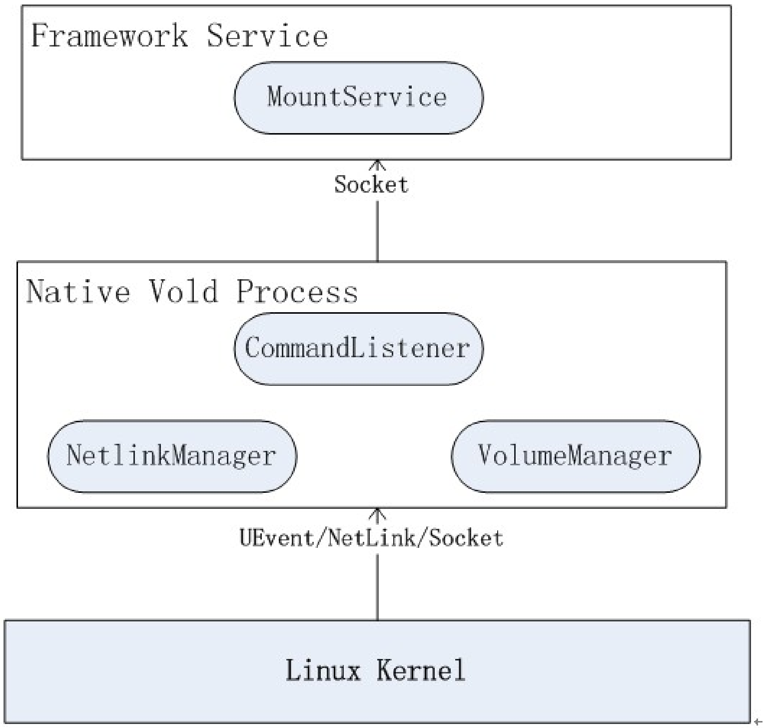
Vold工作流程分析学习

一 Vold工作机制分析

vold进程：管理和控制Android平台外部存储设备，包括SD插拨、挂载、卸载、格式化等；

vold进程接收来自内核的外部设备消息。

Vold框架图如下：



Vold接收来自内核的事件，通过netlink机制。

Netlink 是一种特殊的 socket；

Netlink 是一种在内核与用户应用间进行双向数据传输的非常好的方式，用户态应用使用标准的socket API 就可以使用 netlin 功能；

Netlink是一种异步通信机制，在内核与用户态应用之间传递的消息保存在socket缓存队列中；

内核通过Netlink发送uEvent格式消息给用户空间程序；外部设备发生变化，Kernel发送uevent消息。

二 Vold进程启动过程

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

vold进程执行过程：

\system\vold\main.cpp

```
int main()
{
        VolumeManager *vm;
        CommandListener *cl;
        NetlinkManager *nm;
        //创建vold设备文件夹
        mkdir("/dev/block/vold", 0755);

        //初始化Vold相关的类实例 single
        vm = VolumeManager::Instance();
        nm = NetlinkManager::Instance();

        //CommandListener 创建vold socket监听上层消息
        cl = new CommandListener();
        vm->setBroadcaster((SocketListener *) cl);
        nm->setBroadcaster((SocketListener *) cl);

        //启动VolumeManager
        vm->start();

        //根据配置文件/etc/vold.fstab 初始化VolumeManager
        process_config(vm);

        //启动NetlinkManager socket监听内核发送uevent
        nm->start();

        //向/sys/block/目录下所有设备uevent文件写入“add\n”,
        //触发内核sysfs发送uevent消息
        coldboot("/sys/block");

        //启动CommandListener监听vold socket
        cl->startListener();

        // Eventually we'll become the monitoring thread
        while(1) {
            sleep(1000);
        }

        exit(0);
}
```

process_config解析vold.fstab文件：

```
static int process_config(VolumeManager *vm) {
        //打开vold.fstab的配置文件
        fp = fopen("/etc/vold.fstab", "r")
        //解析vold.fstab 配置存储设备的挂载点
    while(fgets(line, sizeof(line), fp)) {
        const char *delim = " \t";
        char *type, *label, *mount_point, *part, *mount_flags, *sysfs_path;

        type = strtok_r(line, delim, &save_ptr)
        label = strtok_r(NULL, delim, &save_ptr)
        mount_point = strtok_r(NULL, delim, &save_ptr)
              //判断分区 auto没有分区
        part = strtok_r(NULL, delim, &save_ptr)
        if (!strcmp(part, "auto")) {
             //创建DirectVolume对象 相关的挂载点设备的操作
           dv = new DirectVolume(vm, label, mount_point, -1);
        } else {
           dv = new DirectVolume(vm, label, mount_point, atoi(part));
```

```
        }
                //添加挂载点设备路径
        while ((sysfs_path = strtok_r(NULL, delim, &save_ptr))) {
            dv->addPath(sysfs_path)
        }
              //将DirectVolume 添加到VolumeManager管理
              vm->addVolume(dv);
    }

    fclose(fp);
    return 0;
}
```

vold.fstab文件：

导出一个我的手机里面的vold.fstab文件 内容：

```
dev_mount sdcard /mnt/sdcard emmc@fat /devices/platform/goldfish_mmc.0 /devices/platform/mtk-sd.0/mmc_host

dev_mount external_sdcard /mnt/sdcard/external_sd auto /devices/platform/goldfish_mmc.1 /devices/platform/mtk-sd.1/mmc_host
```

vold.fstab格式是:

type   label   mount_point part   sysfs_path   sysfs_path

sysfs_path可以有多个 part指定分区个数，如果是auto没有分区

三 Vold中各模块分析

在vold进程main函数中创建了很多的类实例，并将其启动。

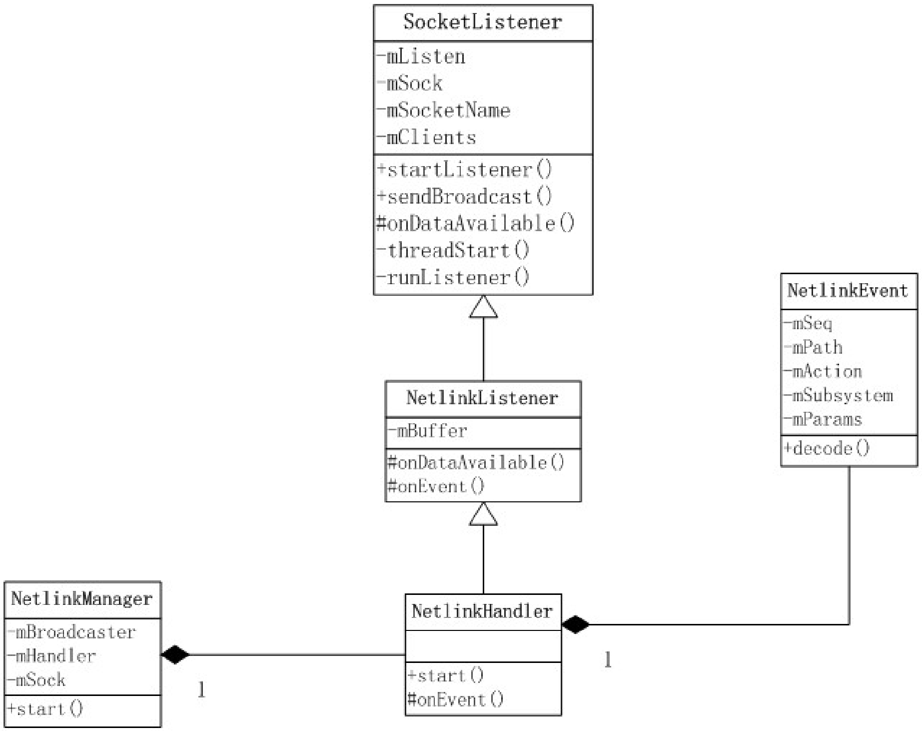
```
int main()
{
        ……
        vm->start();
        nm->start();
        cl->startListener();
}
```

这些类对象之间是如何的，还需要现弄清楚每个类的职责和工作机制。

1 NetlinkManager模块

NetlinkManager模块接收从Kernel发送的Uevent消息，解析转换成NetlinkEvent对象；再将此NetlinkEvent对象传递给Volum
处理。

此模块相关的类结构：



下面从start开始，看起如何对Kernel的Uevent消息进行监控的。

NetlinkManager start：

```
int NetlinkManager::start() {
    //netlink使用的socket结构
    struct sockaddr_nl nladdr;

    //初始化socket数据结构
    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    nladdr.nl_pid = getpid();
    nladdr.nl_groups = 0xffffffff;
    //创建socket PF_NETLINK类型
    mSock = socket(PF_NETLINK,SOCK_DGRAM,NETLINK_KOBJECT_UEVENT);
    //配置socket 大小
    setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz);
    setsockopt(mSock, SOL_SOCKET, SO_PASSCRED, &on, sizeof(on);
    //bindsocket地址
    bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr);

    //创建NetlinkHandler 传递socket标识，并启动
    mHandler = new NetlinkHandler(mSock);
    mHandler->start();
    return 0;
}
```

NetlinkHandler start：

```
int NetlinkHandler::start() {
      //父类startListener
      return this->startListener();
}
```

NetlinkListener start：

```
int SocketListener::startListener() {
    //NetlinkHandler mListen为false
    if (mListen && listen(mSock, 4) < 0) {
        return -1;
    } else if (!mListen){
        //mListen为false 用于netlink消息监听
        //创建SocketClient作为SocketListener 的客户端
        mClients->push_back(new SocketClient(mSock, false, mUseCmdNum));
    }

    //创建匿名管道
    pipe(mCtrlPipe);
    //创建线程执行函数threadStart    参this
    pthread_create(&mThread, NULL, SocketListener::threadStart, this);
}
```

线程监听Kernel netlink发送的UEvent消息：

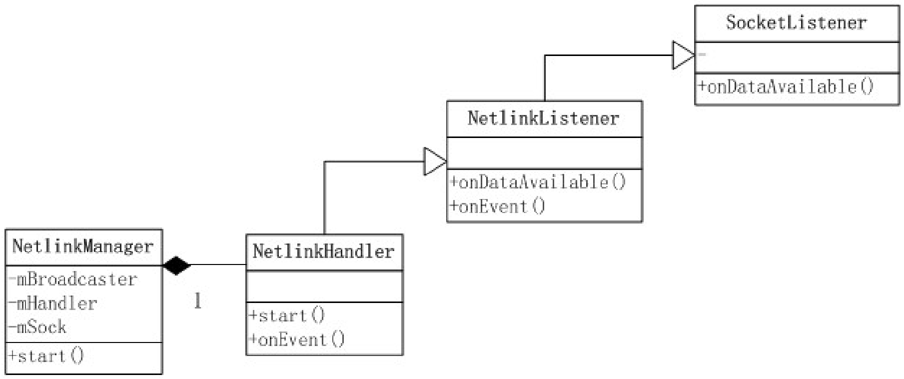
```
void *SocketListener::threadStart(void *obj) {

    //参数转换
    SocketListener *me = reinterpret_cast<SocketListener *>(obj);

    me->runListener();

    pthread_exit(NULL);

    return NULL;

}
```

SocketListener 线程消息循环：

```
void SocketListener::runListener() {
        //SocketClient List
    SocketClientCollection *pendingList = new SocketClientCollection();

    while(1) {
        fd_set read_fds;
        //mListen 为false
        if (mListen) {
            max = mSock;
            FD_SET(mSock, &read_fds);
        }
```

```
        //加入一组文件描述符集合 选择fd最大的max
        FD_SET(mCtrlPipe[0], &read_fds);
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
            int fd = (*it)->getSocket();
            FD_SET(fd, &read_fds);
            if (fd > max)
                max = fd;
        }
        pthread_mutex_unlock(&mClientsLock);

        //监听文件描述符是否变化
        rc = select(max + 1, &read_fds, NULL, NULL, NULL);
        //匿名管道被写，退出线程
        if (FD_ISSET(mCtrlPipe[0], &read_fds))
            break;
        //mListen 为false
        if (mListen && FD_ISSET(mSock, &read_fds)) {
                //mListen 为ture 表示正常监听socket
            struct sockaddr addr;
            do {
                //接收客户端连接
                c = accept(mSock, &addr, &alen);
            } while (c < 0 && errno == EINTR);

            //此处创建一个客户端SocketClient加入mClients列表中，异步延迟处理
            pthread_mutex_lock(&mClientsLock);
            mClients->push_back(new SocketClient(c, true, mUseCmdNum));
            pthread_mutex_unlock(&mClientsLock);
        }

        /* Add all active clients to the pending list first */
        pendingList->clear();
        //将所有有消息的Client加入到pendingList中
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
            int fd = (*it)->getSocket();
            if (FD_ISSET(fd, &read_fds)) {
                pendingList->push_back(*it);
            }
        }
        pthread_mutex_unlock(&mClientsLock);

        //处理所有消息
        while (!pendingList->empty()) {
            it = pendingList->begin();
            SocketClient* c = *it;
            pendingList->erase(it);
               //处理有数据发送的socket 虚函数
            if (!onDataAvailable(c) && mListen) {
               //mListen为false
            }
        }
    }
}
```

Netlink消息处理：



在消息循环中调用onDataAvailable处理消息，onDataAvailable是个虚函数，NetlinkListener重写了此函数。

NetlinkListener onDataAvailable消息处理：

```
bool NetlinkListener::onDataAvailable(SocketClient *cli)
{
    //获取socket id
    int socket = cli->getSocket();
    //接收netlink uevent消息
    count = TEMP_FAILURE_RETRY(uevent_kernel_multicast_uid_recv(
                             socket, mBuffer, sizeof(mBuffer), &uid));
    //解析uevent消息为NetlinkEvent消息
    NetlinkEvent *evt = new NetlinkEvent();
    evt->decode(mBuffer, count, mFormat);

    //处理NetlinkEvent onEvent虚函数
    onEvent(evt);
}
```

将接收的Uevent数据转化成NetlinkEvent数据，调用onEvent处理，NetlinkListener子类NetlinkHandler重写了此函数。

NetlinkHandler NetlinkEvent数据处理：

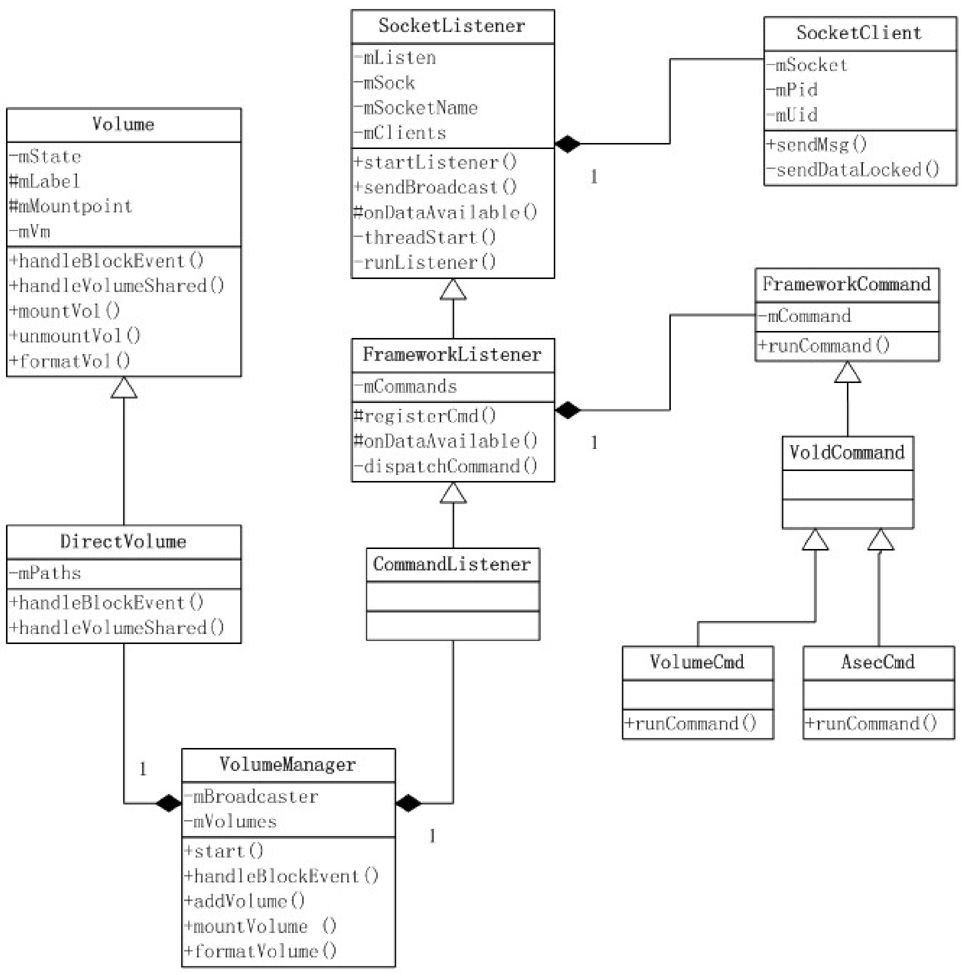
```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {

    //获取VolumeManager实例
    VolumeManager *vm = VolumeManager::Instance();

    //设备类型
    const char *subsys = evt->getSubsystem();



    //将消息传递给VolumeManager处理
    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt);

    }

}
```

NetlinkManager通过NetlinkHandler将接收到Kernel内核发送的Uenvet消息，

转化成了NetlinkEvent结构数据传递给VolumeManager处理。

2 VolumeManager模块

此模块管理所有挂载的设备节点以及相关操作执行；下面是VolumeManager模块类结构图：



DirectVolume：一个实体存储设备在代码中的抽象。

SocketListenner：创建线程，监听socket。

这里VolumeManager构造的SocketListenner与NetlinkManager构造的SocketListenner有所不同的：

NetlinkManager构造的SocketListenner：Kernel与Vold通信；

VolumeManager构造的SocketListenner：Native Vold与Framework MountService 通信；

VolumeManager构造的SocketListenner，由vold进程main函数中创建的CommandListener：

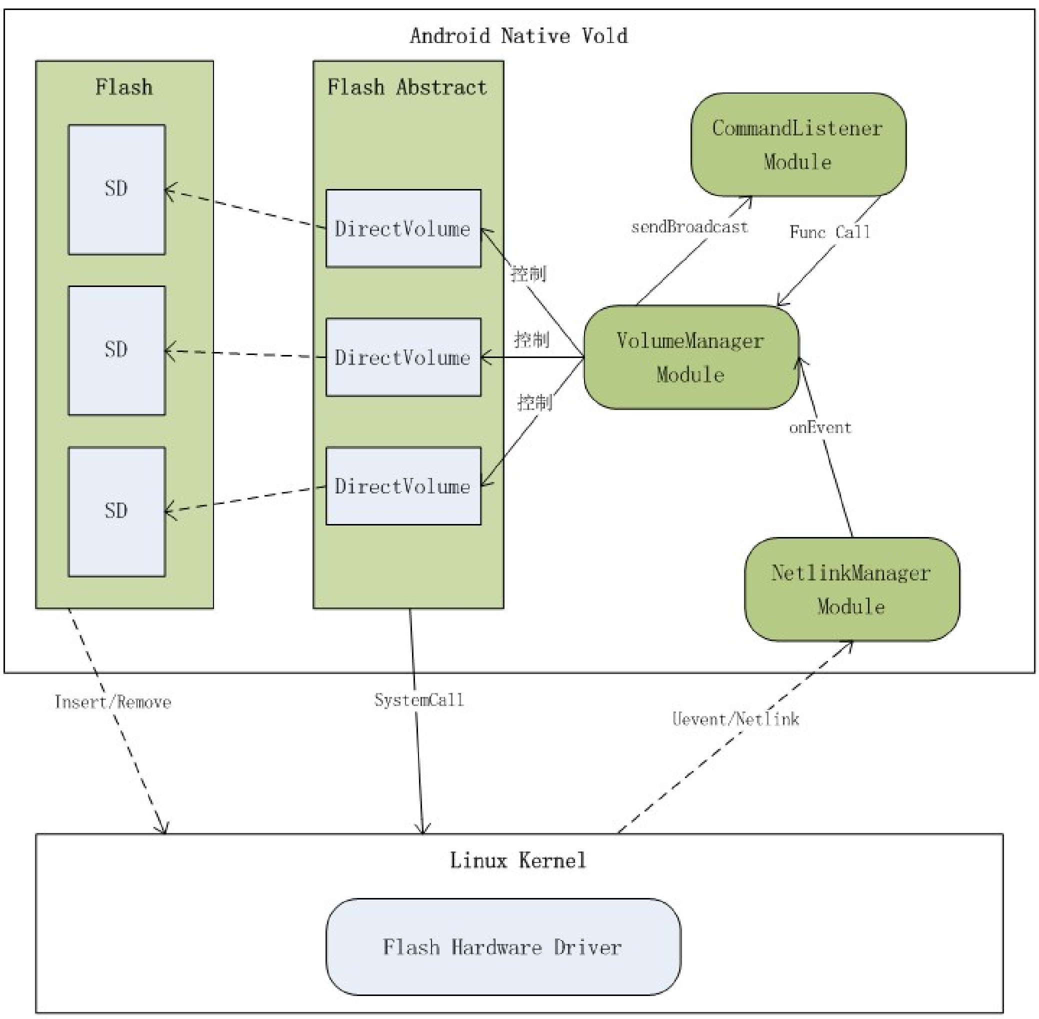
```
int main() {
        ……

        CommandListener *cl;
        cl = new CommandListener();

       vm->setBroadcaster((SocketListener *) cl);

       //启动CommandListener监听
       cl->startListener();

}
```

VolumeManager工作流程：

```
//从main函数中的start开始：
int VolumeManager::start() {
    return 0;
}
```

NetlinkManager接收到Kernel通过netlink发送的Uevent消息，转化成了NetlinkEvent消息，再传递给了VolumeMana

NetlinkManager与VolumeManager交互流程图：



VolumeManager处理消息 handleBlockEvent：

从NetlinkManager到VolumeManager代码过程

函数执行从onEvent到handleBlockEvent：

```
    void NetlinkHandler::onEvent(NetlinkEvent *evt) {
        ……
            //将消息传递给VolumeManager处理
        if (!strcmp(subsys, "block")) {
            vm->handleBlockEvent(evt);
        }
    }
    void VolumeManager::handleBlockEvent(NetlinkEvent *evt) {
        //有状态变化设备路径
       const char *devpath = evt->findParam("DEVPATH");

       //遍历VolumeManager中所管理Volume对象（各存储设备代码抽象）
       for (it = mVolumes->begin(); it != mVolumes->end(); ++it) {
           if (!(*it)->handleBlockEvent(evt)) {
               hit = true;
               break;
           }
    }
}
```

将消息交给各个Volume对象处理：DirectVolume

从VolumeManager到所管理的Volume对象

这里的Volume为其派生类DirectVolume。

```
int DirectVolume::handleBlockEvent(NetlinkEvent *evt)
{
    //有状态变化设备路径
    const char *dp = evt->findParam("DEVPATH");
    PathCollection::iterator  it;
    for (it = mPaths->begin(); it != mPaths->end(); ++it) {
        //匹配 设备路径
        if (!strncmp(dp, *it, strlen(*it))) {
            int action = evt->getAction();
            const char *devtype = evt->findParam("DEVTYPE");
            //动作判断
            if (action == NetlinkEvent::NlActionAdd) {
                int major = atoi(evt->findParam("MAJOR"));
                int minor = atoi(evt->findParam("MINOR"));
                char nodepath[255];
                //设备节点路径名称
                snprintf(nodepath,sizeof(nodepath), "/dev/block/vold/%d:%d",
                         major, minor);

                //创建设备节点
                createDeviceNode(nodepath, major, minor);
                if (!strcmp(devtype, "disk")) {
                    //添加disk
                    handleDiskAdded(dp, evt);
                } else {
```

```
                    //添加分区
                    handlePartitionAdded(dp, evt);
                }
            } else if (action == NetlinkEvent::NlActionRemove) {

            } else if (action == NetlinkEvent::NlActionChange) {

            } else {
                SLOGW("Ignoring non add/remove/change event");
            }
            return 0;
        }
    }
}
```

为什么要让VolumeManager中的每一个Volume对象都去处理SD状态变换消息，

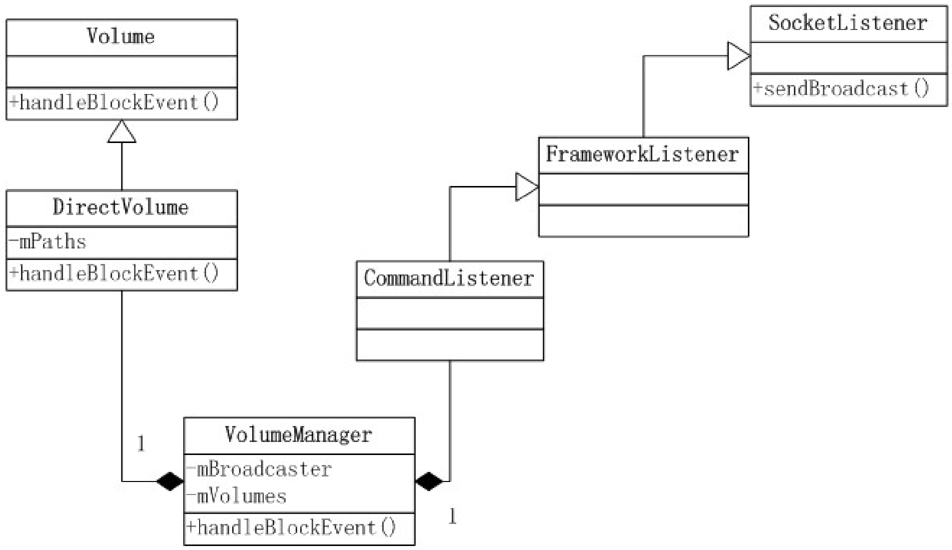
每一个Volume可能对应多个Path；即一个挂载点对应多个物理设备。

抽象存储设备DirectVolume 动作状态变化处理：

```
void DirectVolume::handleDiskAdded(const char *devpath, NetlinkEvent *evt) {
    //主次设备号
    mDiskMajor = atoi(evt->findParam("MAJOR"));
    mDiskMinor = atoi(evt->findParam("MINOR"));

    //设备分区情况
    const char *tmp = evt->findParam("NPARTS");
   mDiskNumParts = atoi(tmp);

    if (mDiskNumParts == 0) {
        //没有分区，Volume状态为Idle
        setState(Volume::State_Idle);
    } else {
        //有分区未加载，设置Volume状态Pending
        setState(Volume::State_Pending);
    }
    //格式化通知msg："Volume sdcard /mnt/sdcard disk inserted (179:0)"
    char msg[255];
    snprintf(msg, sizeof(msg), "Volume %s %s disk inserted (%d:%d)",
             getLabel(), getMountpoint(), mDiskMajor, mDiskMinor);
    //调用VolumeManager中的Broadcaster——>CommandListener 发送此msg
    mVm->getBroadcaster()->sendBroadcast(ResponseCode::VolumeDiskInserted,
                                         msg, false);
}
```

消息通知Framework层存储设备状态变化：

类继承关系：



发送消息通知Framework层是在SocketListener中完成；

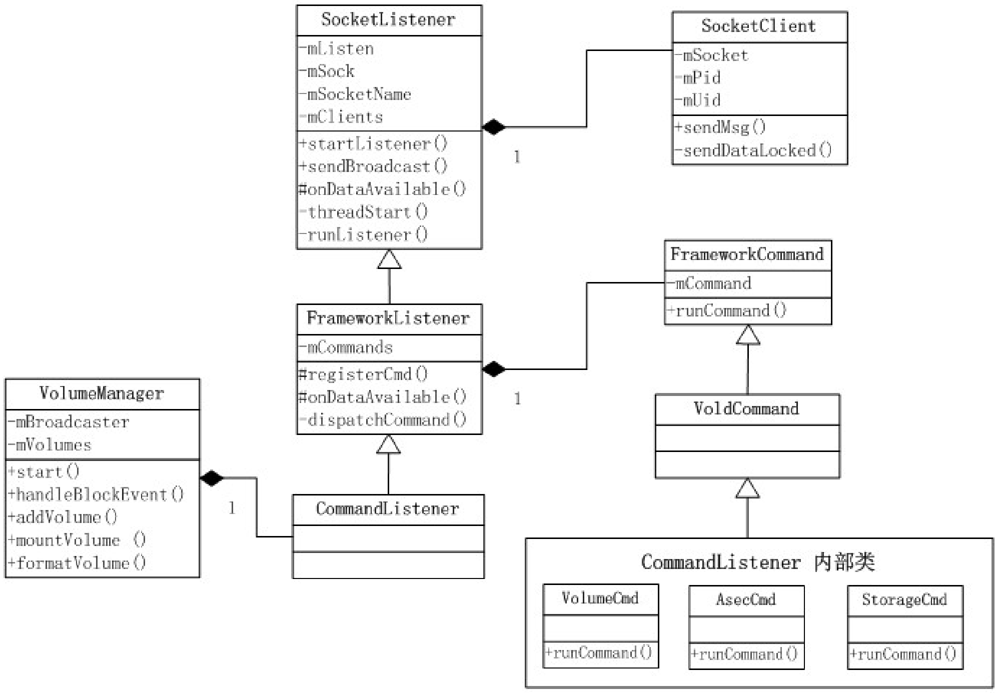
```
void SocketListener::sendBroadcast(int code, const char *msg, bool addErrno)
{
    pthread_mutex_lock(&mClientsLock);
    //遍历所有的消息接收时创建的Client SocketClient
    // SocketClient将消息通过socket ("vold") 通信
    for (i = mClients->begin(); i != mClients->end(); ++i) {
        (*i)->sendMsg(code, msg, addErrno, false);
    }
    pthread_mutex_unlock(&mClientsLock);
}
```

这里工作的SocketListener是VolumeManager的，SocketListener的派生类CommandListener，用来与Framework交互的，监听Socket消息。通过VolumeManager中调用sendBroadcast，与CommandListener模块进行交互。由此需要清楚CommandListener模块工作流程。

3 CommandListener模块

CommandListener监听Socket，使Vold与Framework层进行进程通信；

其相关类继承结构图如下：



CommandListener工作流程：

```
int main()
{
       VolumeManager *vm;
       CommandListener *cl;
       NetlinkManager *nm;

      //CommandListener 创建vold socket监听上层消息
       cl = new CommandListener();

       //作为VolumeManager与NetlinkManager的Broadcaster
       vm->setBroadcaster((SocketListener *) cl);
       nm->setBroadcaster((SocketListener *) cl);

      //启动CommandListener监听
       cl->startListener();
       ……
}
```

CommandListener实例的创建：构造函数

CommandListener构造函数：

```
    CommandListener::CommandListener() :
                 FrameworkListener("vold", true) {
```

```
    //注册Framework发送的相关命令 Command模式

    registerCmd(new DumpCmd());

    registerCmd(new VolumeCmd());

    registerCmd(new AsecCmd());

    registerCmd(new ObbCmd());

    registerCmd(new StorageCmd());

    registerCmd(new XwarpCmd());

    registerCmd(new CryptfsCmd());

}
```

FrameworkListener构造函数：

```
FrameworkListener::FrameworkListener(const char *socketName, bool withSeq) :

                            SocketListener(socketName, true, withSeq) {

    mCommands = new FrameworkCommandCollection();

    mWithSeq = withSeq;

}
```

注册Command：

```
void FrameworkListener::registerCmd(FrameworkCommand *cmd) {
    mCommands->push_back(cmd);
}
```

SocketListener构造函数：

```
SocketListener::SocketListener(const char *socketName, bool listen, bool useCmdNum) {

    //mListen = true 正常的socket监听
    mListen = listen;

    //socket 名称"vold"
    mSocketName = socketName;

    mSock = -1;

    mUseCmdNum = useCmdNum;
```

```
    //初始化锁
    pthread_mutex_init(&mClientsLock, NULL);

    //构造Listener Client List
    mClients = new SocketClientCollection();

}
```

CommandListener启动 startListener：

```
int SocketListener::startListener() {
    //mSocketName = “Vold”
    mSock = android_get_control_socket(mSocketName);

    //NetlinkHandler mListen为true 监听socket
    if (mListen && < 0) {
        return -1;
    } else if (!mListen){
        mClients->push_back(new SocketClient(mSock, false, mUseCmdNum));
    }

    //创建匿名管道
    pipe(mCtrlPipe);
    //创建线程执行函数threadStart 参数this
    pthread_create(&mThread, NULL, SocketListener::threadStart, this);
}

void *SocketListener::threadStart(void *obj) {
    SocketListener *me = reinterpret_cast<SocketListener *>(obj);
    me->runListener();
}

void SocketListener::runListener() {
    //SocketClient List
    SocketClientCollection *pendingList = new SocketClientCollection();

    while(1) {
        fd_set read_fds;
                //mListen 为true
        if (mListen) {
            max = mSock;
            FD_SET(mSock, &read_fds);
        }
                //加入一组文件描述符集合 选择fd最大的max select有关
        FD_SET(mCtrlPipe[0], &read_fds);
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
            int fd = (*it)->getSocket();
            FD_SET(fd, &read_fds);
            if (fd > max)
                max = fd;
        }
        pthread_mutex_unlock(&mClientsLock);

        //监听文件描述符是否变化
        rc = select(max + 1, &read_fds, NULL, NULL, NULL);
        //匿名管道被写，退出线程
        if (FD_ISSET(mCtrlPipe[0], &read_fds))
            break;

        //mListen 为true
```

```
        if (mListen && FD_ISSET(mSock, &read_fds)) {
            //mListen 为ture 表示正常监听socket
            struct sockaddr addr;
            do {
                c = accept(mSock, &addr, &alen);
            } while (c < 0 && errno == EINTR);

            //创建一个客户端SocketClient，加入mClients列表中 到异步延迟处理
            pthread_mutex_lock(&mClientsLock);
            mClients->push_back(new SocketClient(c, true, mUseCmdNum));
            pthread_mutex_unlock(&mClientsLock);
        }

        /* Add all active clients to the pending list first */
        pendingList->clear();
        //将所有有消息的Client加入到pendingList中
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
            int fd = (*it)->getSocket();
            if (FD_ISSET(fd, &read_fds)) {
                pendingList->push_back(*it);
            }
        }
        pthread_mutex_unlock(&mClientsLock);

        /* Process the pending list, since it is owned by the thread,*/
        while (!pendingList->empty()) {
            it = pendingList->begin();
            SocketClient* c = *it;
               //处理有数据发送的socket
            if (!onDataAvailable(c) && mListen) {
               //mListen为true
               ……
            }
        }
    }
}
```

CommandListener启动的线程监听Socket消息，接收到的消息处理onDataAvailable。

CommandListener父类FrameworkCommand重写了此函数。

CommandListener监听Socket消息处理：

```
bool FrameworkListener::onDataAvailable(SocketClient *c) {
    char buffer[255];
    //读取socket消息
    len = TEMP_FAILURE_RETRY(read(c->getSocket(), buffer, sizeof(buffer)));
    for (i = 0; i < len; i++) {
        if (buffer[i] == '\0') {
            //根据消息内容 派发命令
            dispatchCommand(c, buffer + offset);
            offset = i + 1;
        }
    }
    return true;
}

void FrameworkListener::dispatchCommand(SocketClient *cli, char *data) {
   char *argv[FrameworkListener::CMD_ARGS_MAX];
    //解析消息内容 命令 参数
    ……
```

```
    //执行对应的消息
    for (i = mCommands->begin(); i != mCommands->end(); ++i) {
        FrameworkCommand *c = *i;
        //匹配命令
        if (!strcmp(argv[0], c->getCommand())) {
            //执行命令
            c->runCommand(cli, argc, argv);
            goto out;
        }
    }
out:
        return;
}
```

Command执行处理：以VolumeCommand为例

```
CommandListener::VolumeCmd::VolumeCmd() :
                 VoldCommand("volume") {
}

int CommandListener::VolumeCmd::runCommand(SocketClient *cli,
                                    int argc, char **argv) {

    //获取VolumeManager实例
    VolumeManager *vm = VolumeManager::Instance();
    //Action判断 传递给VolumeManager处理
    if (!strcmp(argv[1], "list")) {
        return vm->listVolumes(cli);
    } else if (!strcmp(argv[1], "debug")) {
        vm->setDebug(!strcmp(argv[2], "on") ? true : false);
    } else if (!strcmp(argv[1], "mount")) {
        rc = vm->mountVolume(argv[2]);
    } else if (!strcmp(argv[1], "unmount")) {
        rc = vm->unmountVolume(argv[2], force, revert);
    } else if (!strcmp(argv[1], "format")) {
        rc = vm->formatVolume(argv[2]);
    } else if (!strcmp(argv[1], "share")) {
        rc = vm->shareVolume(argv[2], argv[3]);
    } else if (!strcmp(argv[1], "unshare")) {
        rc = vm->unshareVolume(argv[2], argv[3]);
    } else if (!strcmp(argv[1], "shared")) {
       ……

    return 0;
}
```

CommandListener使用Command模式。

CommandListener接收到来自Framework层得消息，派发命令处理，再传递给VolumeManager处理。

VolumeManager中Action处理：

```
int VolumeManager::unmountVolume(const char *label) {
    //查找Volume
```

```
    Volume *v = lookupVolume(label);
    //Volume执行动作
    return v-> unmountVol ();
}
//VolumeAction处理：
int Volume::unmountVol(bool force, bool revert) {
    doUnmount(Volume::SEC_STG_SECIMGDIR, force);
        ……
}

int Volume::doUnmount(const char *path, bool force) {
    ……
    //systemcall
    umount(path);
}
```

整个Vold处理过程框架图如下：