# Vold分析

Last edited by **caoquanli** 1 month ago

# vold整个流程详细分析

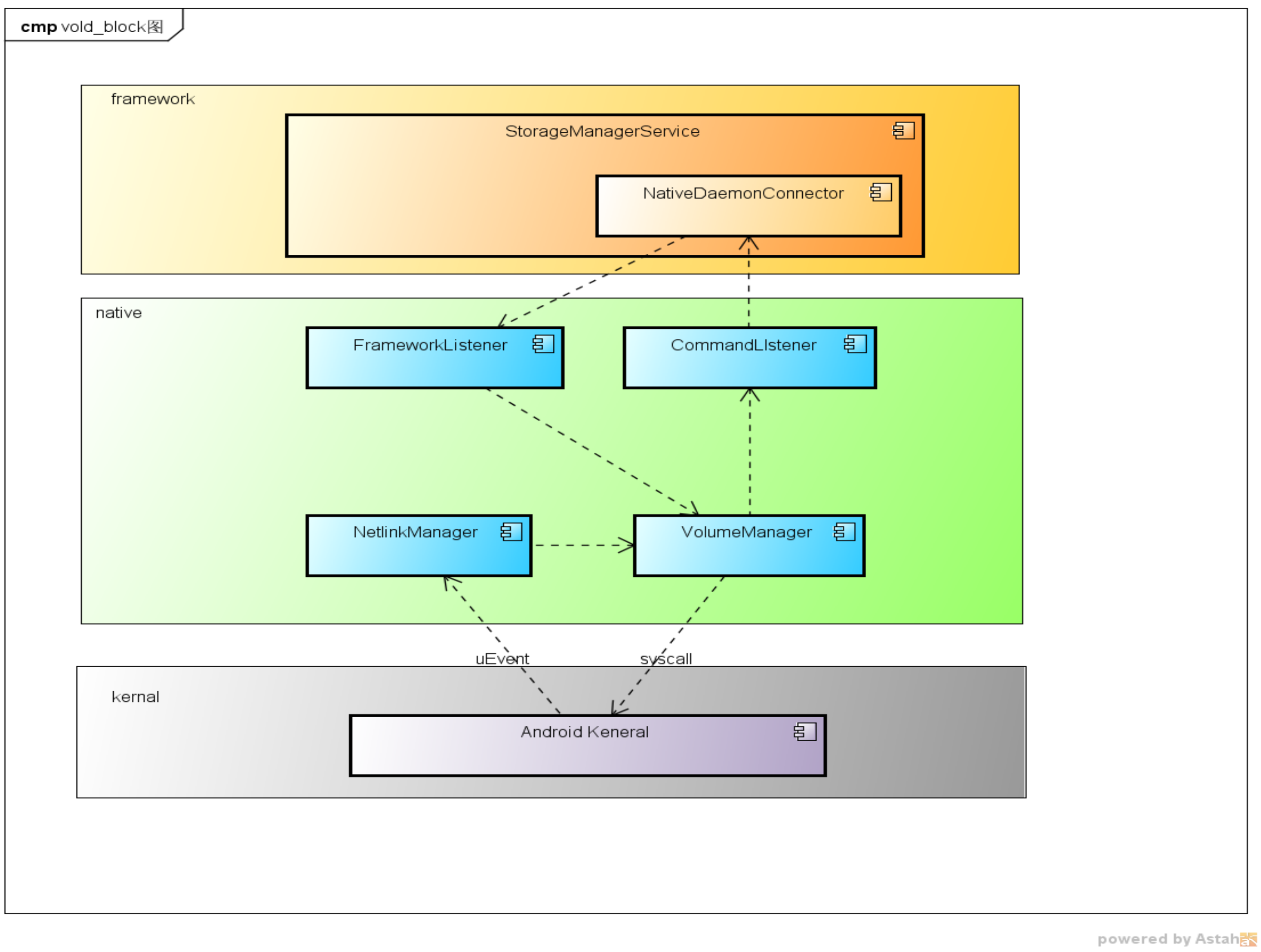
Table of Contents



liubaoyin < liubaoyin@iauto.com> v1.0 , 2019-1-25

本章开篇以StorageManagerService讲起，分为五大模块详细分析存储系统的架构以及流程，核心就是Vold与StorageManagerService之间的交互。

Vold整体框架如下图：



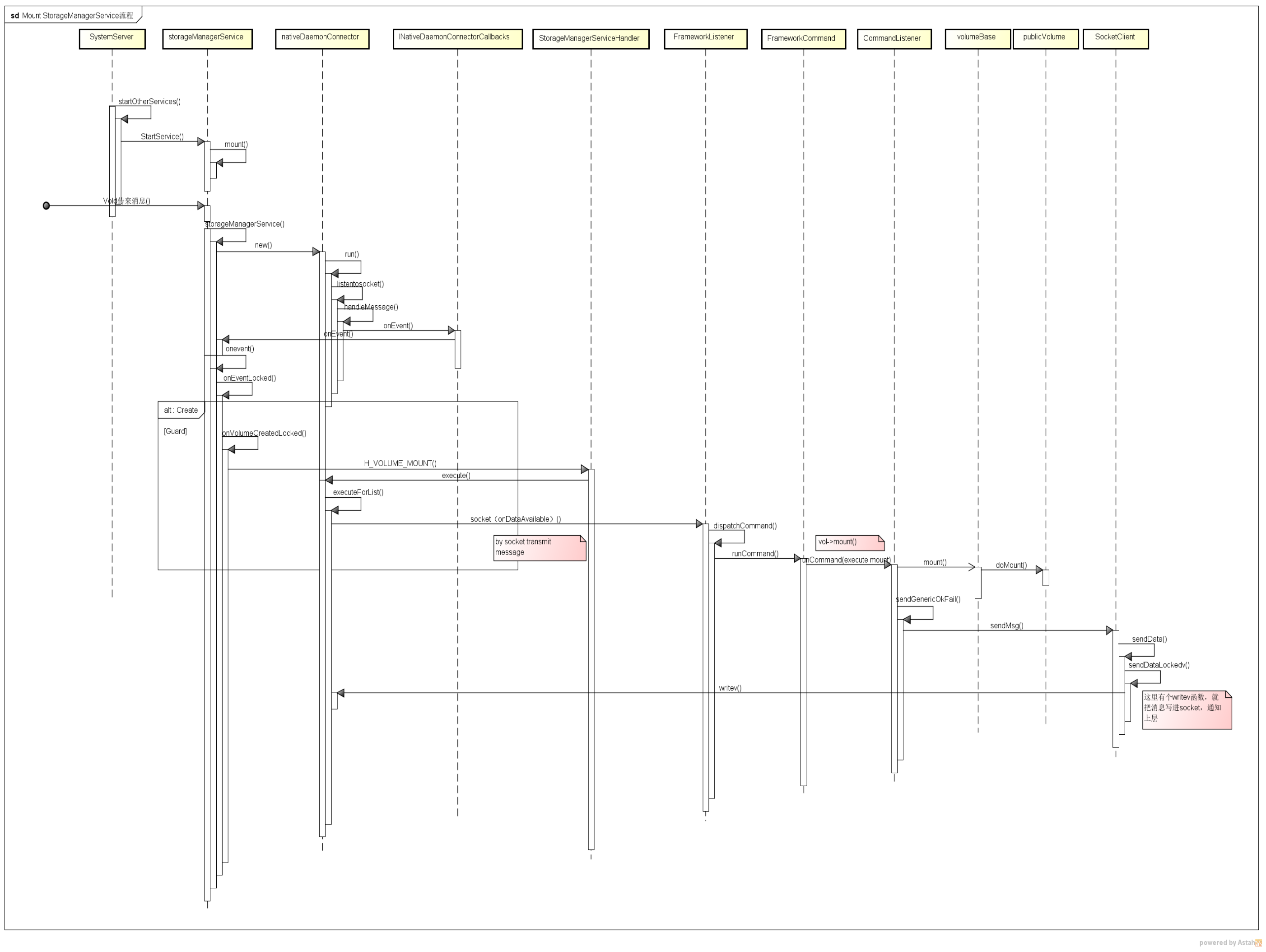

从上图中我们可以简单介绍一下架构

- Linux Kernel：通过NetLink以uevent的形式向Vold的NetlinkManager发送Uevent事件，触发Vold工作
- NetlinkManager：接收来自Kernel的Uevent事件，再转发给VolumeManager去处理
- VolumeManager：接收来自NetlinkManager的事件，再转发给CommandListener进行处理
- CommandListener：接收来自VolumeManager的事件，通过socket通信方式发送给MountService

- StorageManagerService：接收来自CommandListener的事件然后处理以及给vold发送处理请求

# 第一章 启动StorageManagerService

StorageManagerService整体时序如下图：




## 1.1节 System_server启动service

有关system_server的启动这里不再再赘述，想了解的同学可一上网搜一下，本章就从system_server启动StorageManagerService开始说。 在Android7.0之前这个service都叫MountService，相信大家对于这个service都不陌生，这个service是Android系统专门用来管理存储设备的，是framework层与vold通信的门户。直接看代码。

SystemServer.java

```
private void startOtherServices() {
    ...
    if (mFactoryTestMode != FactoryTest.FACTORY_TEST_LOW_LEVEL) {
            if (!disableStorage &&
                    !"0".equals(SystemProperties.get("system_init.startmountservice"))) {
                traceBeginAndSlog("StartStorageManagerService");
                try {
                    /*
                     * NotificationManagerService is dependant on StorageManagerService,
                     * (for media / usb notifications) so we must start StorageManagerSer
vice first.
                     */
                    mSystemServiceManager.startService(STORAGE_MANAGER_SERVICE_CLASS);
                    storageManager = IStorageManager.Stub.asInterface(
                            ServiceManager.getService("mount"));
                } catch (Throwable e) {
                    reportWtf("starting StorageManagerService", e);
                }
                traceEnd();

                traceBeginAndSlog("StartStorageStatsService");
                try {
                    mSystemServiceManager.startService(STORAGE_STATS_SERVICE_CLASS);
                } catch (Throwable e) {
                    reportWtf("starting StorageStatsService", e);
                }
                traceEnd();
            }
```

这里的STORAGE_MANAGER_SERVICE_CLASS就是com.android.server.usage.StorageStatsService$Lifecycle； 到这里就去看看StorageManagerService中的Lifecycle对象。 onStart()

```
public void onStart() {
     mStorageManagerService = new    StorageManagerService(getContext());
```

```
     publishBinderService("mount", mStorageManagerService);
     mStorageManagerService.start();
    }
```

很明显了这个Service就是从这里起的。 有关这个Lifecycle，这个玩意是android现在已经封装好的一个用来描述Activity等的生命周期的架构。可能很多人不理解，我也不理解，就当他是表示生命周期的吧。

## 1.2 StorageManagerService初始化

service已经起来了，到这里就去看StorageManagerService的构造函数了。这个构造函数很重要，做了很多东西。

```
    public StorageManagerService(Context context) {
        sSelf = this;

        mContext = context;
        mCallbacks = new Callbacks(FgThread.get().getLooper());
        mLockPatternUtils = new LockPatternUtils(mContext);

        // XXX: This will go away soon in favor of IMountServiceObserver
        mPms = (PackageManagerService) ServiceManager.getService("package");

        HandlerThread hthread = new HandlerThread(TAG);
        hthread.start();
        mHandler = new StorageManagerServiceHandler(hthread.getLooper());

        // Add OBB Action Handler to StorageManagerService thread.
        mObbActionHandler = new ObbActionHandler(IoThread.get().getLooper());

        // Initialize the last-fstrim tracking if necessary
        File dataDir = Environment.getDataDirectory();
        File systemDir = new File(dataDir, "system");
        mLastMaintenanceFile = new File(systemDir, LAST_FSTRIM_FILE);
        if (!mLastMaintenanceFile.exists()) {
            // Not setting mLastMaintenance here means that we will force an
            // fstrim during reboot following the OTA that installs this code.
            try {
                (new FileOutputStream(mLastMaintenanceFile)).close();
            } catch (IOException e) {
                Slog.e(TAG, "Unable to create fstrim record " + mLastMaintenanceFile.getP
ath());
            }
        } else {
            mLastMaintenance = mLastMaintenanceFile.lastModified();
        }

        mSettingsFile = new AtomicFile(
                new File(Environment.getDataSystemDirectory(), "storage.xml"));

        synchronized (mLock) {
            readSettingsLocked();
        }

        LocalServices.addService(StorageManagerInternal.class, mStorageManagerInternal);

        /*
         * 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_TA
G, 25,
                null);
        mConnector.setDebug(true);
        mConnector.setWarnIfHeld(mLock);
        mConnectorThread = new Thread(mConnector, VOLD_TAG);

        // Reuse parameters from first connector since they are tested and safe
        mCryptConnector = new NativeDaemonConnector(this, "cryptd",
                MAX_CONTAINERS * 2, CRYPTD_TAG, 25, null);
        mCryptConnector.setDebug(true);
        mCryptConnectorThread = new Thread(mCryptConnector, CRYPTD_TAG);

        final IntentFilter userFilter = new IntentFilter();
        userFilter.addAction(Intent.ACTION_USER_ADDED);
```

```
        userFilter.addAction(Intent.ACTION_USER_REMOVED);
        mContext.registerReceiver(mUserReceiver, userFilter, null, mHandler);

        synchronized (mLock) {
            addInternalVolumeLocked();
        }

        // Add ourself to the Watchdog monitors if enabled.
        if (WATCHDOG_ENABLE) {
            Watchdog.getInstance().addMonitor(this);
        }
    }
```

一步步分析他都干了些啥:

- 1.创建了mCallbacks回调方法，他也是个handler，他用的looper是名为android.fg的FgThread线程的looper。
- 2.创建了StorageManagerServiceHandler，new了一个hthread，与这个handler进行绑定。
- 3.创建obb操作的handler，mObbActionHandler，用线程名为android.io的IoThread的looper。
- 4.创建NativeDaemonConnector对象，与vold的通信都是通过他来操作的。
- 5.创建并启动VoldConnector线程。
- 6.创建并启动CryptdConnector线程。 这里一下子起了三个线程，并且用了两个系统线程，任务繁重啊。

这个callback方法就是一个handler，他在一开始new一个RemoteCallbackList<>对象，这里面其实就是一个死亡通知回调，当binder死了之后，他会回调binderDied方法通知用户。

```
public class RemoteCallbackList<E extends IInterface> {
    private static final String TAG = "RemoteCallbackList";

    /*package*/ ArrayMap<IBinder, Callback> mCallbacks
            = new ArrayMap<IBinder, Callback>();
    private Object[] mActiveBroadcast;
    private int mBroadcastCount = -1;
    private boolean mKilled = false;
    private StringBuilder mRecentCallers;

    private final class Callback implements IBinder.DeathRecipient {
        final E mCallback;
        final Object mCookie;

        Callback(E callback, Object cookie) {
            mCallback = callback;
            mCookie = cookie;
        }

        public void binderDied() {
            synchronized (mCallbacks) {
                mCallbacks.remove(mCallback.asBinder());
            }
            onCallbackDied(mCallback, mCookie);
        }
    }
    }
```

### 1.3 NativeDaemonConnector

下面就到了看NativeDaemonConnector的时候了，这个对象中我们能够了解到nativie的消息是怎么传到framework层，并且framework层是怎么给native层抛消息的。

```
NativeDaemonConnector(INativeDaemonConnectorCallbacks callbacks, String socket,
            int responseQueueSize, String logTag, int maxLogSize, PowerManager.WakeLock w
l,
            Looper looper) {
        mCallbacks = callbacks;
        mSocket = socket;
        mResponseQueue = new ResponseQueue(responseQueueSize);
        mWakeLock = wl;
        if (mWakeLock != null) {
            mWakeLock.setReferenceCounted(true);
        }
        mLooper = looper;
```

```
        mSequenceNumber = new AtomicInteger(0);
        TAG = logTag != null ? logTag : "NativeDaemonConnector";
        mLocalLog = new LocalLog(maxLogSize);
    }
```

这玩意里面也搞了很多东西，首先注意下那个ResponseQueue，这个queue就是存放命令消息的，最大空间是五百，超过就会阻塞。而且呢这个玩意也是实现runnerable接口的，所以看看他的run方法，里面就干了两件事，new了一个handler处理消息的，还有一个listenToSocket()方法。handler这个方法最中会callback到StorageManagerService中的onEvent()方法中。而listenerToSocket方法就是真正接收native消息的了，就是一个socket客户端。

```
    private void listenToSocket() throws IOException {
        LocalSocket socket = null;

        try {
            socket = new LocalSocket();
            LocalSocketAddress address = determineSocketAddress();

            socket.connect(address);

            InputStream inputStream = socket.getInputStream();
            synchronized (mDaemonLock) {
                mOutputStream = socket.getOutputStream();
            }

            mCallbacks.onDaemonConnected();

            FileDescriptor[] fdList = null;
            byte[] buffer = new byte[BUFFER_SIZE];
            int start = 0;

            while (true) {
                int count = inputStream.read(buffer, start, BUFFER_SIZE - start);
                if (count < 0) {
                    loge("got " + count + " reading with start = " + start);
                    break;
                }
                fdList = socket.getAncillaryFileDescriptors();

                // Add our starting point to the count and reset the start.
                count += start;
                start = 0;

                for (int i = 0; i < count; i++) {
                    if (buffer[i] == 0) {
                        // Note - do not log this raw message since it may contain
                        // sensitive data
                        final String rawEvent = new String(
                                buffer, start, i - start, StandardCharsets.UTF_8);

                        boolean releaseWl = false;
                        try {
                            final NativeDaemonEvent event =
                                    NativeDaemonEvent.parseRawEvent(rawEvent, fdList);

                            log("RCV <- {" + event + "}");

                            if (event.isClassUnsolicited()) {
                                // TODO: migrate to sending NativeDaemonEvent instances
                                if (mCallbacks.onCheckHoldWakeLock(event.getCode())
                                        && mWakeLock != null) {
                                    mWakeLock.acquire();
                                    releaseWl = true;
                                }
                                Message msg = mCallbackHandler.obtainMessage(
                                        event.getCode(), uptimeMillisInt(), 0, event.getR
awEvent());
                                if (mCallbackHandler.sendMessage(msg)) {
                                    releaseWl = false;
                                }
                            } else {
                                mResponseQueue.add(event.getCmdNumber(), event);
                            }
                        } catch (IllegalArgumentException e) {
                            log("Problem parsing message " + e);
```

```
                    } finally {
                        if (releaseWl) {
                            mWakeLock.release();
                        }
                    }

                    start = i + 1;
                }
            }

            if (start == 0) {
                log("RCV incomplete");
            }

            // We should end at the amount we read. If not, compact then
            // buffer and read again.
            if (start != count) {
                final int remaining = BUFFER_SIZE - start;
                System.arraycopy(buffer, start, buffer, 0, remaining);
                start = remaining;
            } else {
                start = 0;
            }
        }
    } catch (IOException ex) {
        loge("Communications error: " + ex);
        throw ex;
    } finally {
        synchronized (mDaemonLock) {
            if (mOutputStream != null) {
                try {
                    loge("closing stream for " + mSocket);
                    mOutputStream.close();
                } catch (IOException e) {
                    loge("Failed closing output stream: " + e);
                }
                mOutputStream = null;
            }
        }

        try {
            if (socket != null) {
                socket.close();
            }
        } catch (IOException ex) {
            loge("Failed closing socket: " + ex);
        }
    }
```

仔细看看代码相信大家都能理解他是怎么接收消息的了。先建立一个socket连接，然后就是接收event了，如果响应码在[600,700]区间内，就交给cCallbackHandler去处理，如果没有就添加到mResponseQueue中，用ResponseQueue.add()方法，这个玩意再去看看，他会把一些未处理的pendind命令去保存好。

以上就是StorageManagerService接收socket信息的过程。

### 1.4 启动Service

然后按照生命周期的流程，就走到life中的onBootPhase()[启动阶段]中了。

```
public void onBootPhase(int phase) {
            if (phase == SystemService.PHASE_ACTIVITY_MANAGER_READY) {
                mStorageManagerService.systemReady();
            } else if (phase == SystemService.PHASE_BOOT_COMPLETED) {
                mStorageManagerService.bootCompleted();
            }
        }
```

就是起了一个方法，systemReady()，这个方法会给Handler发送一个消息H_SYSTEM_READY，去handleMessage中去看看这个message会干什么，调用handleSystemReady()方法，这里面很简单，

```
private void handleSystemReady() {
        initIfReadyAndConnected();
        resetIfReadyAndConnected();
```

```
        // Start scheduling nominally-daily fstrim operations
        MountServiceIdler.scheduleIdlePass(mContext);
    }
```

就干了这几件事。里面具体的东西可以看看源码，这里就不再细说了。 现在回到listentoSocket方法去看看他是怎么处理消息的，对于响应码在[600,700]区间内时，上面代码已经展示过，用mCallbackhandler去处理，然后也是给他发个消息，去看看handleMessage，直接就是调用mCallbacks的onEvent方法：

```
    public boolean handleMessage(Message msg) {
        final String event = (String) msg.obj;
        final int start = uptimeMillisInt();
        final int sent = msg.arg1;
        try {
            if (!mCallbacks.onEvent(msg.what, event, NativeDaemonEvent.unescapeArgs(even
t))) {
                log(String.format("Unhandled event '%s'", event));
            }
        } catch (Exception e) {
            loge("Error handling '" + event + "': " + e);
        } finally {
            if (mCallbacks.onCheckHoldWakeLock(msg.what) && mWakeLock != null) {
                mWakeLock.release();
            }
            final int end = uptimeMillisInt();
            if (start > sent && start - sent > WARN_EXECUTE_DELAY_MS) {
                loge(String.format("NDC event {%s} processed too late: %dms", event, star
t - sent));
            }
            if (end > start && end - start > WARN_EXECUTE_DELAY_MS) {
                loge(String.format("NDC event {%s} took too long: %dms", event, end - sta
rt));
            }
        }
        return true;
    }
```

这个onEvent方法是INativeDaemonConnectorCallbacks中的方法，去看看发现他是空的，然后和回过头去看StorageManagerService，发现他是INativeDaemonConnectorCallbacks的子类，所以实际就是回到了StorageManagerService中。去看他的onEcent方法，很简单就是调用到了onEventLocked方法。他就会根据相应码采取不同的操作去处理来自下层的消息，以VOLUME_CREATE为例，相应码650，我们去看看他的操作，在这里有有部分很重要，当framework层收到这样的响应码，会涉及到framework层向native层发消息的操作。仔细研究一下，看下一章。

## 第二章 Framework层发消息流程

上面说到当请求码是650时，会调用到onVolumeCreatedLocked方法

```
private void onVolumeCreatedLocked(VolumeInfo vol) {
        if (mPms.isOnlyCoreApps()) {
            Slog.d(TAG, "System booted in core-only mode; ignoring volume " + vol.getId
());
            return;
        }

        if (vol.type == VolumeInfo.TYPE_EMULATED) {
            final StorageManager storage = mContext.getSystemService(StorageManager.clas
s);
            final VolumeInfo privateVol = storage.findPrivateForEmulated(vol);

            if (Objects.equals(StorageManager.UUID_PRIVATE_INTERNAL, mPrimaryStorageUuid)
                    && VolumeInfo.ID_PRIVATE_INTERNAL.equals(privateVol.id)) {
                Slog.v(TAG, "Found primary storage at " + vol);
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_PRIMARY;
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_VISIBLE;
                mHandler.obtainMessage(H_VOLUME_MOUNT, vol).sendToTarget();

            } else if (Objects.equals(privateVol.fsUuid, mPrimaryStorageUuid)) {
                Slog.v(TAG, "Found primary storage at " + vol);
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_PRIMARY;
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_VISIBLE;
                mHandler.obtainMessage(H_VOLUME_MOUNT, vol).sendToTarget();
            }
            } else if (vol.type == VolumeInfo.TYPE_PUBLIC) {
```

```
            // TODO: only look at first public partition
            if (Objects.equals(StorageManager.UUID_PRIMARY_PHYSICAL, mPrimaryStorageUuid)
                    && vol.disk.isDefaultPrimary()) {
                Slog.v(TAG, "Found primary storage at " + vol);
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_PRIMARY;
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_VISIBLE;
            }

            // Adoptable public disks are visible to apps, since they meet
            // public API requirement of being in a stable location.
            if (vol.disk.isAdoptable()) {
                vol.mountFlags |= VolumeInfo.MOUNT_FLAG_VISIBLE;
            }

            vol.mountUserId = mCurrentUserId;
            mHandler.obtainMessage(H_VOLUME_MOUNT, vol).sendToTarget();

        } else if (vol.type == VolumeInfo.TYPE_PRIVATE) {
            mHandler.obtainMessage(H_VOLUME_MOUNT, vol).sendToTarget();
}
```

可以看到不管是检测到是什么类型的volume他都会给mHandler发送一个H_VOLUME_MOUNT消息，呐，再去handlerMessage去看看他是怎么处理的

```
case H_VOLUME_MOUNT: {
                    final VolumeInfo vol = (VolumeInfo) msg.obj;
                    if (isMountDisallowed(vol)) {
                        Slog.i(TAG, "Ignoring mount " + vol.getId() + " due to policy");
                        break;
                    }
                    try {
                        mConnector.execute("volume", "mount", vol.id, vol.mountFlags,
                                vol.mountUserId);
                    } catch (NativeDaemonConnectorException ignored) {
                    }
```

这地方又回到NativeDaemonConnector中了，调用了他的的execute()方法。这个方法最终会调用到executeForList()中,还是把代码弄出来看看。

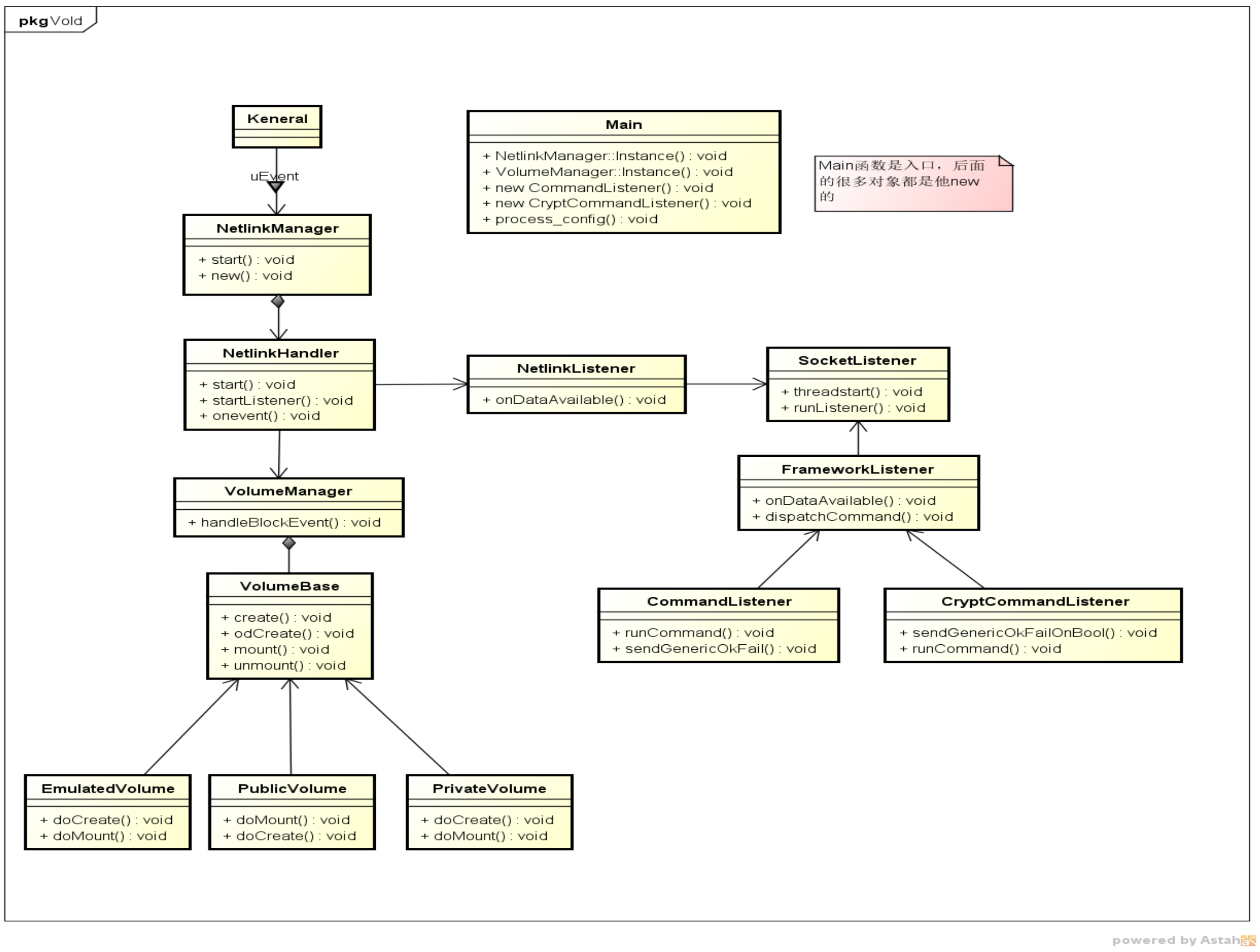
```
    public NativeDaemonEvent[] executeForList(long timeoutMs, String cmd, Object... args)
            throws NativeDaemonConnectorException {
        if (mWarnIfHeld != null && Thread.holdsLock(mWarnIfHeld)) {
            Slog.wtf(TAG, "Calling thread " + Thread.currentThread().getName() + " is hol
ding 0x"
                    + Integer.toHexString(System.identityHashCode(mWarnIfHeld)), new Thro
wable());
        }

        final long startTime = SystemClock.elapsedRealtime();

        final ArrayList<NativeDaemonEvent> events = Lists.newArrayList();

        final StringBuilder rawBuilder = new StringBuilder();
        final StringBuilder logBuilder = new StringBuilder();
        final int sequenceNumber = mSequenceNumber.incrementAndGet();

        makeCommand(rawBuilder, logBuilder, sequenceNumber, cmd, args);

        final String rawCmd = rawBuilder.toString();
        final String logCmd = logBuilder.toString();

        log("SND -> {" + logCmd + "}");

        synchronized (mDaemonLock) {
            if (mOutputStream == null) {
                throw new NativeDaemonConnectorException("missing output stream");
            } else {
                try {
                    mOutputStream.write(rawCmd.getBytes(StandardCharsets.UTF_8));
                } catch (IOException e) {
                    throw new NativeDaemonConnectorException("problem sending command",
e);
                }
            }
```

```
        }

        NativeDaemonEvent event = null;
        do {
            event = mResponseQueue.remove(sequenceNumber, timeoutMs, logCmd);
            if (event == null) {
                loge("timed-out waiting for response to " + logCmd);
                throw new NativeDaemonTimeoutException(logCmd, event);
            }
            if (VDBG) log("RMV <- {" + event + "}");
            events.add(event);
        } while (event.isClassContinue());

        final long endTime = SystemClock.elapsedRealtime();
        if (endTime - startTime > WARN_EXECUTE_DELAY_MS) {
            loge("NDC Command {" + logCmd + "} took too long (" + (endTime - startTime) +
"ms)");
        }

        if (event.isClassClientError()) {
            throw new NativeDaemonArgumentException(logCmd, event);
        }
        if (event.isClassServerError()) {
            throw new NativeDaemonFailureException(logCmd, event);
        }

        return events.toArray(new NativeDaemonEvent[events.size()]);
```

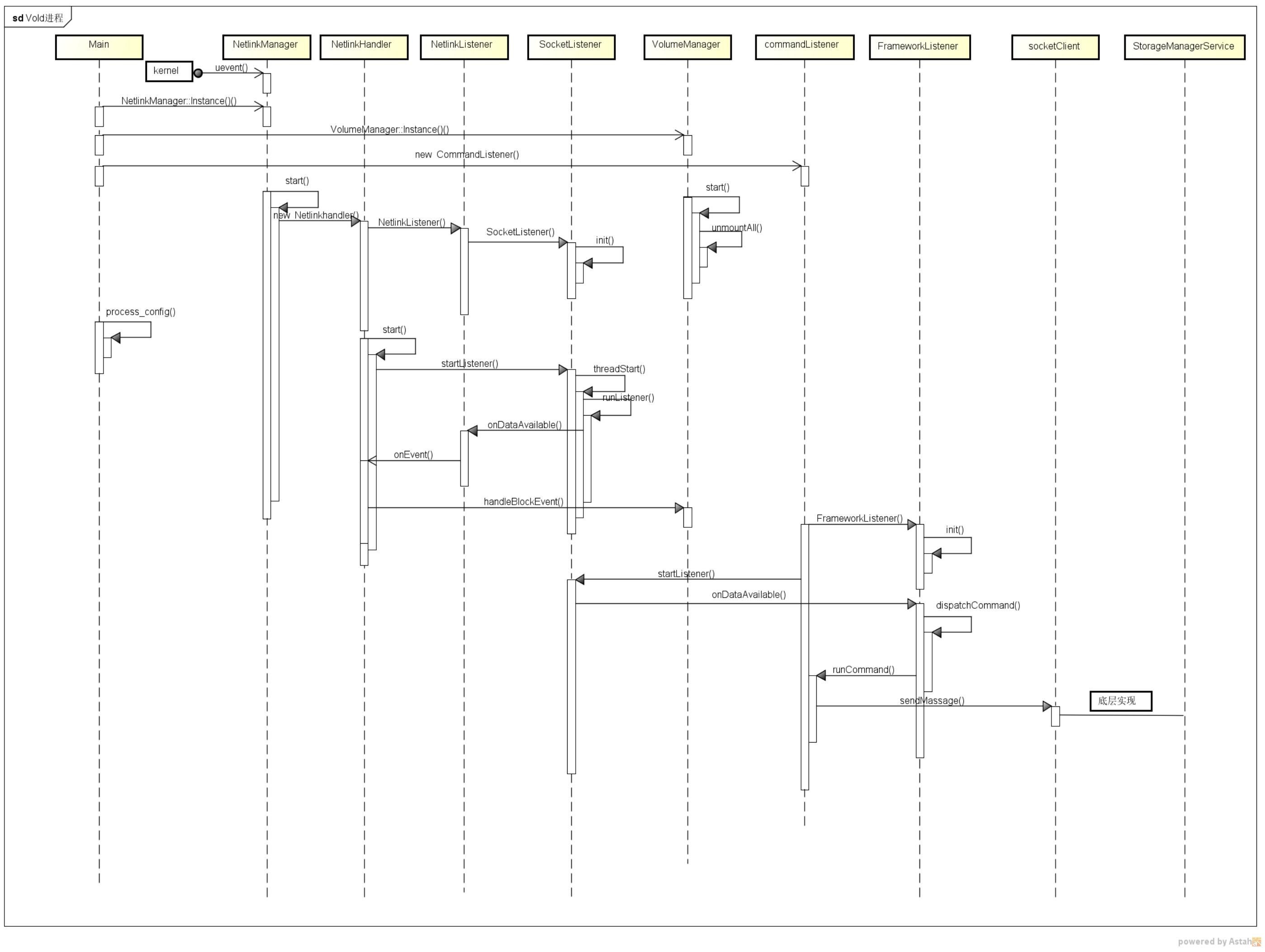
呐，里面有个 mOutputStream.write操作，这个就是给socket中写信息的，还有mResponseQueue.remove，这个就是处理信息的，队列操作，想了解看看代码。现在这地方就相当与给native层发消息了。

## 第三章 Vold模块启动流程

Vold 整体类关系：




Vold整体时序图：




vold（Volume Daemoon）这个玩意大家都不会陌生，native层的存储管理都是他来处理。先分析他的工作原理以及启动。 vold他是在开机加载init.rc的时候起来的，

```
on post-fs-data
    # We chown/chmod /data again so because mount is run as root + defaults
    chown system system /data
    chmod 0771 /data
    # We restorecon /data in case the userdata partition has been reset.
    restorecon /data

    # Make sure we have the device encryption key.
    start vold//启动vold服务
    installkey /data
```

native世界，起来了那就去他的main函数中看看，代码有点多这里就不列举了，感兴趣的同学去system/vold/main.cpp看看，大概说一说，他会创建并起四个对象，

- VolumeManager
- NetlinkManager （NetlinkHandler）
- CommandListener
- CryptCommandListener

还有一个process_config方法，这个就是解析fastab文件的。 接下来就说说起来的那几个类。在说这几个类之前先说一下类关系。这部分错综复杂很容易理不清。

基本类关系在本章开始已经展示出来了。

上面已经说了vold的启动，下面就以vold接收到新设备为例讲一下vold部分的挂载流程。

### 3.1 kernel发出uEvent

kernel上报事件给用户采用的是netlink方式，这个是一种的特殊的socket，传送的消息是暂存在socket接收缓存中，并不被接收者立即处理，是一种异步通信机制，而syscall呵ioctl都是同步通信机制。kernel使用大量的netlink与native层进行通信，比如u盘的插入，就会产生uEvent(User Space event)，这里说一下这个uEvent，uEvent是Kobject的一部分，当Kobject状态改变时通知用户空间程序程序，对于kobject_action包括KOBJ_ADD，KOBJ_REMOVE，KOBJ_CHANGE，KOBJ_MOVE，KOBJ_ONLINE，KOBJ_OFFLINE，当发送任何一种action都会引发Kernel发送Uevent消息。

### 3.2 NetlinkManager

这里得接收ENetlink的event是封装好的NetlinkManager，这个对象是在main函数中起来的，上面已经说过现在直接看他的start方法，

```
int NetlinkManager::start() {
    struct sockaddr_nl nladdr;
    int sz = 64 * 1024;
```

```
    int on = 1;

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

    if ((mSock = socket(PF_NETLINK, SOCK_DGRAM | SOCK_CLOEXEC,
            NETLINK_KOBJECT_UEVENT)) < 0) {
        SLOGE("Unable to create uevent socket: %s", strerror(errno));
        return -1;
    }

    // When running in a net/user namespace, SO_RCVBUFFORCE is not available.
    // Try using SO_RCVBUF first.
    if ((setsockopt(mSock, SOL_SOCKET, SO_RCVBUF, &sz, sizeof(sz)) < 0) &&
        (setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz)) < 0)) {
        SLOGE("Unable to set uevent socket SO_RCVBUF/SO_RCVBUFFORCE option: %s", strerror
(errno));
        goto out;
    }

    if (setsockopt(mSock, SOL_SOCKET, SO_PASSCRED, &on, sizeof(on)) < 0) {
        SLOGE("Unable to set uevent socket SO_PASSCRED option: %s", strerror(errno));
        goto out;
    }

    if (bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
        SLOGE("Unable to bind uevent socket: %s", strerror(errno));
        goto out;
    }

    mHandler = new NetlinkHandler(mSock);
    if (mHandler->start()) {
        SLOGE("Unable to start NetlinkHandler: %s", strerror(errno));
        goto out;
    }

    return 0;

out:
    close(mSock);
    return -1;
}
```

很明显建立了socket的端口，并于kernal建立了连接，连接建立好了，当kernel那边发过来event这边就会去处理，在代码中他new了一个NetlinkHandler，直接start，这个地方就调到socketListener中了，不知道大家有没有发现NetLinkListener是继承自SocketListener的，so,this一指，指到父类中去了，看SocketListener的startListener方法

### 3.3 SocketListener

```
int SocketListener::startListener(int backlog) {

    if (!mSocketName && mSock == -1) {
        SLOGE("Failed to start unbound listener");
        errno = EINVAL;
        return -1;
    } else if (mSocketName) {
        if ((mSock = android_get_control_socket(mSocketName)) < 0) {
            SLOGE("Obtaining file descriptor socket '%s' failed: %s",
                 mSocketName, strerror(errno));
            return -1;
        }
        SLOGV("got mSock = %d for %s", mSock, mSocketName);
        fcntl(mSock, F_SETFD, FD_CLOEXEC);
    }

    if (mListen && listen(mSock, backlog) < 0) {
        SLOGE("Unable to listen on socket (%s)", strerror(errno));
        return -1;
    } else if (!mListen)
        mClients->push_back(new SocketClient(mSock, false, mUseCmdNum));

    if (pipe(mCtrlPipe)) {
        SLOGE("pipe failed (%s)", strerror(errno));
```

```
        return -1;
    }

    if (pthread_create(&mThread, NULL, SocketListener::threadStart, this)) {
        SLOGE("pthread_create (%s)", strerror(errno));
        return -1;
    }

    return 0;
```

看后面又起了个线程，专门用来监听socket，thread→threadStart，这个方法直接建立了一个socketListener对象，去执行runListener，这个方法的大部分代码就不放了就方最关键的部分，

```
if (!onDataAvailable(c)) {
                release(c, false);
            }
            c->decRef();
```

调到onDataAvaiable方法去了，这个就要跑去他的子类去看这个方法了，NetlinkListener→onDataAvailable()，这个方法首先获取socket信息，然后new了一个NetlinkEvent对象，把他塞到OnEvent方法中去执行，这个方法在NetlinkHandler中，

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();

    if (!subsys) {
        SLOGW("No subsystem found in netlink event");
        return;
    }

    const char *tmpAction = evt->findParam("ACTION");
    const char *tmpName = evt->findParam("DEVNAME");
    const char *tmpDevType = evt->findParam("DEVTYPE");
    const char *tmpDevPath = evt->findParam("DEVPATH");
    const char *tmpInterface = evt->findParam("INTERFACE");

    SLOGW("subsys is %s\n", subsys);
    SLOGW("tmpAction is %s\n", tmpAction);
    SLOGW("tmpName is %s\n", tmpName);
    SLOGW("tmpDevType is %s\n", tmpDevType);
    SLOGW("tmpDevPath is %s\n", tmpDevPath);
    SLOGW("tmpInterface is %s\n", tmpInterface);

    if (!strcmp(subsys, "usb")
        && tmpDevType && !strcmp(tmpDevType, "usb_device")) {
        if (evt->getAction() == NetlinkEvent::Action::kAdd) {
            SLOGW("NetlinkHandler usb_device:%s","add");
            vm->handleUsbdevice("add", tmpDevPath);
        }else if (evt->getAction() == NetlinkEvent::Action::kRemove) {
            SLOGW("NetlinkHandler usb_device:%s","remove");
            vm->handleUsbdevice("remove", tmpDevPath);
        }
    }

    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt);
    }
}
```

### 3.4 VolumeManager::handleBlockEvent

这部分代码粘一下，很重要，大家仔细分下，他实例了一个VolumeManager，最终还是调到VolumeManager中去了，执行他的handleBlockEvent（这部分都是以新增USB为例子来说的）。

```
switch (evt->getAction()) {
    case NetlinkEvent::Action::kAdd: {
        for (const auto& source : mDiskSources) {
            if (source->matches(eventPath)) {
                // For now, assume that MMC and virtio-blk (the latter is
                // emulator-specific; see Disk.cpp for details) devices are SD,
                // and that everything else is USB
                int flags = source->getFlags();
```

```
                if (major == kMajorBlockMmc
                    || (android::vold::IsRunningInEmulator()
                    && major >= (int) kMajorBlockExperimentalMin
                    && major <= (int) kMajorBlockExperimentalMax)) {
                    flags |= android::vold::Disk::Flags::kSd;
                } else {
                    flags |= android::vold::Disk::Flags::kUsb;
                }

                auto disk = new android::vold::Disk(eventPath, device,
                        source->getNickname(), flags);
                disk->create();
                mDisks.push_back(std::shared_ptr<android::vold::Disk>(disk));
                break;
            }
        }
        break;
    }
```

当action是add时，走上面的流程，new了一个disk对象，走到了他的create方法中。

```
status_t Disk::create() {
    CHECK(!mCreated);
    mCreated = true;
    notifyEvent(ResponseCode::DiskCreated, StringPrintf("%d", mFlags));
    notifyPath();
    readMetadata();
    readPartitions();
    return OK;
}
```

关键就是中间的那个notifyEvent了。去看看

```
void Disk::notifyEvent(int event, const std::string& value) {
    VolumeManager::Instance()->getBroadcaster()->sendBroadcast(event,
            StringPrintf("%s %s", getId().c_str(), value.c_str()).c_str(), false);
}
```

呐搞了一个“广播”发出去了，调到socketListener中去了，执行他的sendBroadcast方法，这里面就是sendMsg了，之后就到了SocketClient中去了，调用sendData，然后到sendDataLockedv，这个方法就是给frameWork发消息的最终手段了

```
for (;;) {
        ssize_t rc = TEMP_FAILURE_RETRY(
            writev(mSocket, iov + current, iovcnt - current));

        if (rc > 0) {
            size_t written = rc;
            while ((current < iovcnt) && (written >= iov[current].iov_len)) {
                written -= iov[current].iov_len;
                current++;
            }
            if (current == iovcnt) {
                break;
            }
            iov[current].iov_base = (char *)iov[current].iov_base + written;
            iov[current].iov_len -= written;
            continue;
        }
```

就是给socket中write信息，然后frameWork层作为服务器去接收。接收到之后的流程上面已经说过了这里就到这了。

## 第四章 Vold接收上层消息处理流程

上面分析过framework层接收到创建消息会在给vold下发消息，去让vold去实现真正的挂载操作。上面已经分析了下发流程，下面就来分析一下他的读取以及实现流程。

### 4.1 FrameworkListener

接收来自framework的消息主要就是FrameworkListener在读，这个对象是在CommandListener中注册的，他也是CommandListener的父类，更是SocketListener的子类，执行他的onDataAvailable

```
bool FrameworkListener::onDataAvailable(SocketClient *c) {
    char buffer[CMD_BUF_SIZE];
    int len;

    len = TEMP_FAILURE_RETRY(read(c->getSocket(), buffer, sizeof(buffer)));
    if (len < 0) {
        SLOGE("read() failed (%s)", strerror(errno));
        return false;
    } else if (!len) {
        return false;
    } else if (buffer[len-1] != '\0') {
        SLOGW("String is not zero-terminated");
        android_errorWriteLog(0x534e4554, "29831647");
        c->sendMsg(500, "Command too large for buffer", false);
        mSkipToNextNullByte = true;
        return true;
    }

    int offset = 0;
    int i;

    for (i = 0; i < len; i++) {
        if (buffer[i] == '\0') {
            /* IMPORTANT: dispatchCommand() expects a zero-terminated string */
            if (mSkipToNextNullByte) {
                mSkipToNextNullByte = false;
            } else {
                dispatchCommand(c, buffer + offset);
            }
            offset = i + 1;
        }
```

这里面就是一个read一直在读socket中的数据，拿到数据（命令）之后就dispatchCommand

```
    for (i = mCommands->begin(); i != mCommands->end(); ++i) {
        FrameworkCommand *c = *i;

        if (!strcmp(argv[0], c->getCommand())) {
            if (c->runCommand(cli, argc, argv)) {
                SLOGW("Handler '%s' error (%s)", c->getCommand(), strerror(errno));
            }
            goto out;
        }
    }
```

到这里就是走到FrameworkCommand中的runCommand中去了，这个runCommand在哪呢，这种知道不到方法就去他的.h文件看看，跑到CommandListener中去了

## 4.2 CommandListener

到这里了，根据command，就那上面的volume创建来说，这里走到volume分支，再看他的mount部分

```
else if (cmd == "mount" && argc > 2) {
        // mount [volId] [flags] [user]
        std::string id(argv[2]);
        auto vol = vm->findVolume(id);
        if (vol == nullptr) {
            return cli->sendMsg(ResponseCode::CommandSyntaxError, "Unknown volume", fals
e);
        }

        int mountFlags = (argc > 3) ? atoi(argv[3]) : 0;
        userid_t mountUserId = (argc > 4) ? atoi(argv[4]) : -1;

        vol->setMountFlags(mountFlags);
        vol->setMountUserId(mountUserId);

        int res = vol->mount();
        if (mountFlags & android::vold::VolumeBase::MountFlags::kPrimary) {
            vm->setPrimary(vol);
        }
        return sendGenericOkFail(cli, res);
```

看那个mount方法，这里会走到VolumeManager中去，他有个自身属性，VolumeBase，在根据volume种类，对于USB来说，它就是PublicVolume，new一个这个方法，所以之后的mount操作就交给他了

### 4.3 PublicVolume

mount 然后到doMount

```
status_t PublicVolume::doMount() {
    // TODO: expand to support mounting other filesystems
    readMetadata();

    if (mFsType != "vfat") {
        LOG(ERROR) << getId() << " unsupported filesystem " << mFsType;
        return -EIO;
    }

    if (vfat::Check(mDevPath)) {
        LOG(ERROR) << getId() << " failed filesystem check";
        return -EIO;
    }

    // Use UUID as stable name, if available
    std::string stableName = getId();
    if (!mFsUuid.empty()) {
        stableName = mFsUuid;
    }

    mRawPath = StringPrintf("/mnt/media_rw/%s", stableName.c_str());

    mFuseDefault = StringPrintf("/mnt/runtime/default/%s", stableName.c_str());
    mFuseRead = StringPrintf("/mnt/runtime/read/%s", stableName.c_str());
    mFuseWrite = StringPrintf("/mnt/runtime/write/%s", stableName.c_str());

    setInternalPath(mRawPath);
    if (getMountFlags() & MountFlags::kVisible) {
        setPath(StringPrintf("/storage/%s", stableName.c_str()));
    } else {
        setPath(mRawPath);
    }

    if (fs_prepare_dir(mRawPath.c_str(), 0700, AID_ROOT, AID_ROOT)) {
        PLOG(ERROR) << getId() << " failed to create mount points";
        return -errno;
    }

    if (vfat::Mount(mDevPath, mRawPath, false, false, false,
            AID_MEDIA_RW, AID_MEDIA_RW, 0007, true)) {
        PLOG(ERROR) << getId() << " failed to mount " << mDevPath;
        return -EIO;
    }

    if (getMountFlags() & MountFlags::kPrimary) {
        initAsecStage();
    }

    if (!(getMountFlags() & MountFlags::kVisible)) {
        // Not visible to apps, so no need to spin up FUSE
        return OK;
    }

    if (fs_prepare_dir(mFuseDefault.c_str(), 0700, AID_ROOT, AID_ROOT) ||
            fs_prepare_dir(mFuseRead.c_str(), 0700, AID_ROOT, AID_ROOT) ||
            fs_prepare_dir(mFuseWrite.c_str(), 0700, AID_ROOT, AID_ROOT)) {
        PLOG(ERROR) << getId() << " failed to create FUSE mount points";
        return -errno;
    }

    dev_t before = GetDevice(mFuseWrite);

    if (!(mFusePid = fork())) {
        if (getMountFlags() & MountFlags::kPrimary) {
            if (execl(kFusePath, kFusePath,
                    "-u", "1023", // AID_MEDIA_RW
                    "-g", "1023", // AID_MEDIA_RW
                    "-U", std::to_string(getMountUserId()).c_str(),
```

```
                    "-w",
                    mRawPath.c_str(),
                    stableName.c_str(),
                    NULL)) {
                PLOG(ERROR) << "Failed to exec";
            }
        } else {
            if (execl(kFusePath, kFusePath,
                    "-u", "1023", // AID_MEDIA_RW
                    "-g", "1023", // AID_MEDIA_RW
                    "-U", std::to_string(getMountUserId()).c_str(),
                    mRawPath.c_str(),
                    stableName.c_str(),
                    NULL)) {
                PLOG(ERROR) << "Failed to exec";
            }
        }

        LOG(ERROR) << "FUSE exiting";
        _exit(1);
    }

    if (mFusePid == -1) {
        PLOG(ERROR) << getId() << " failed to fork";
        return -errno;
    }

    while (before == GetDevice(mFuseWrite)) {
        LOG(VERBOSE) << "Waiting for FUSE to spin up...";
        usleep(50000); // 50ms
    }
    /* sdcardfs will have exited already. FUSE will still be running */
    TEMP_FAILURE_RETRY(waitpid(mFusePid, nullptr, WNOHANG));

    return OK;
}
```

相信到这里大家就熟悉了，挂载的目录都是我们熟悉的目录，绝体的挂载操作就是这里了。这里呢执行完了就会在回到CommandListener中的runCommand，代码在上一小节执行sendGenericOkFail，

```
int CommandListener::sendGenericOkFail(SocketClient *cli, int cond) {
    if (!cond) {
        return cli->sendMsg(ResponseCode::CommandOkay, "Command succeeded", false);
    } else {
        return cli->sendMsg(ResponseCode::OperationFailed, "Command failed", false);
    }
}
```

逻辑很简单，更据挂载完的cond，判断是否挂载成功，发送不同的响应码，sendMsg上面也走过了，这里就不再走了。挂载完了，又回到framework层去了。

## 第五章 StorageManagerService 处理挂载成功请求

这里又跑到NativeDaemonConnector中去了，流程上面已经介绍过了，不再赘述，就看看收到挂载成功的响应吧，在接收到挂载成功的消息后，handleMassage时会给onEvent发送VOLUME_STATE_CHANGED消息，当收到这个消息时

```
 case VoldResponseCode.VOLUME_STATE_CHANGED: {
                if (cooked.length != 3) break;
                final VolumeInfo vol = mVolumes.get(cooked[1]);
                if (vol != null) {
                    final int oldState = vol.state;
                    final int newState = Integer.parseInt(cooked[2]);
                    vol.state = newState;
                                        Log.d(TAG,"oldState and newState: " + oldState +
"$$" + newState);
                    onVolumeStateChangedLocked(vol, oldState, newState);
                }
                break;
```

调用onVolumeStateChangedLocked方法，代码这这里也不看了，和简单，也是发个消息去处理，然后在VolumeInfo中根据state去获取一个Intent，然后就是把这个广播打包，调用sendBroadcastAsUser，方法发送出去，这个广播是啥呢，就是我们在MediaScanner中经常所说的mount通知（ACTION_MEDIA_MOUNTED），这个广播就是在这里发出去的。

## 第六章 总结

在本章中，大体的介绍了一下Vold与StroageManagerService的工作原理以及基本架构，这部分内容对研究vold帮助还是有一些的，本章内容都是我通过阅读源码以及找网上的资料做出来的整理，内容有点长，错误也不少，望指正，大家一起学习。