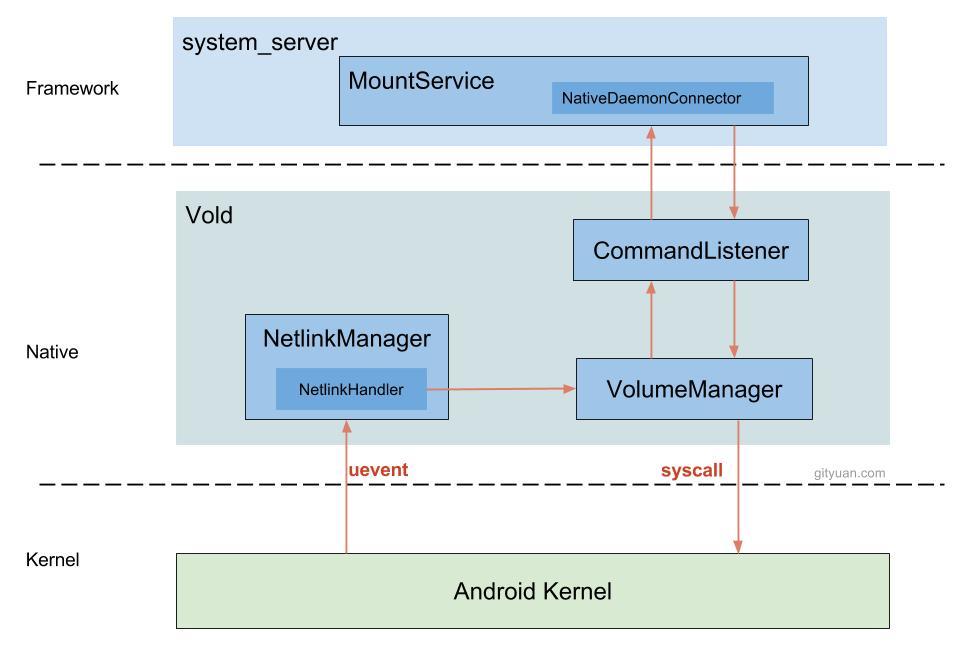
**Vold通信机制在Android9.0架构变化及挂载流程简析**

Vold是Volume Daemon的缩写，它是Android平台中外部存储系统的管控中心，是管理和控制Android平台外部存储设备的后台进程。其功能主要包括：SD卡的插拔事件检测、SD卡挂载、卸载、格式化等。

Android9之前的版本,vold和framework通信是通过socket方式.我们先看一下Android9之前的架构图:



在Android9.0的版本上,vold和Framework的通信方式是binder通信.vold的服务不在init.rc中启动,它会在自己的一个rc文件里配置.位置位于system/vold/vold.rc

service vold /system/bin/vold \

--blkid\_context=u:r:blkid:s0 --blkid\_untrusted\_context=u:r:blkid\_untrusted:s0 \

--fsck\_context=u:r:fsck:s0 --fsck\_untrusted\_context=u:r:fsck\_untrusted:s0

class core

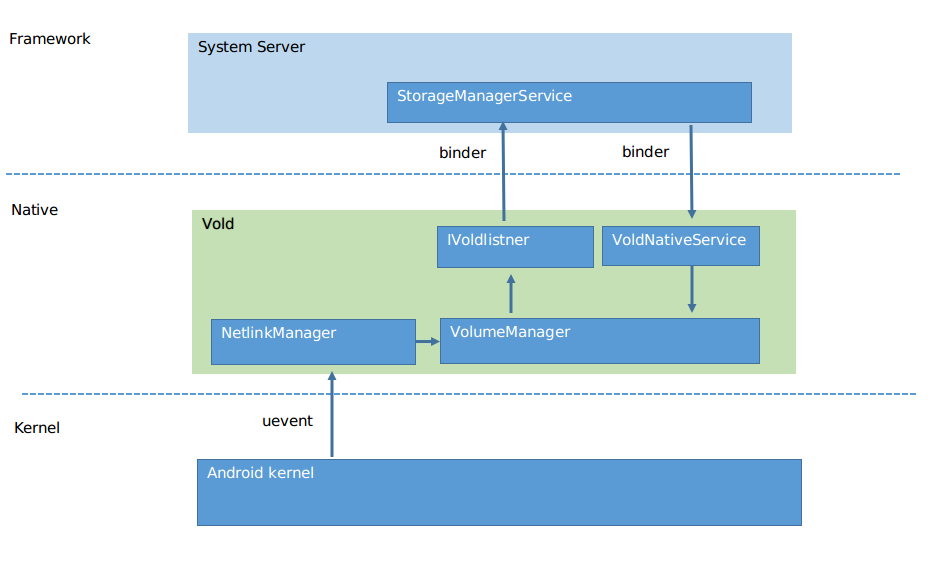
ioprio be 2

writepid /dev/cpuset/foreground/tasks

shutdown critical

group root reserved\_disk

被init进程启动后，将调用system/vold/main.cpp中的main函数,然后初始化VolumeManager,NetlinkManager等等.初始化Vold服务.在分析vold的流程的时候,我们先看一下vold在Android9.0上的架构图.



从两张图的对比,我们可以很清晰的看到vold的架构变化.我们先看看Android9.0新增的一些文件.

/system/vold/binder/android/os/

IVold.aidl

IVoldListener.aidl

IVoldTaskListener.aidl

这几个文件是binder调用的aidl文件,我们先分析一下vold和framework是如何建立联系的.

整体思路就是StorageManagerService建立和vold的binder通信,然后设置设置一个listner来监听vold消息.详细分析如下:

1.在开机过程中,StorageManagerService会被加载和初始化.StorageManagerService的初始化.StorageManagerService由SystemServer启动

private void connect() {

IBinder binder = ServiceManager.getService("storaged");

if (binder != null) {

try {

binder.linkToDeath(new DeathRecipient() {

@Override

public void binderDied() {

Slog.w(TAG, "storaged died; reconnecting");

mStoraged = null;

connect();

}

}, 0);

} catch (RemoteException e) {

binder = null;

}

}

if (binder != null) {

mStoraged = IStoraged.Stub.asInterface(binder);

} else {

Slog.w(TAG, "storaged not found; trying again");

}

binder = ServiceManager.getService("vold");

if (binder != null) {

try {

binder.linkToDeath(new DeathRecipient() {

@Override

public void binderDied() {

Slog.w(TAG, "vold died; reconnecting");

mVold = null;

connect();

}

}, 0);

} catch (RemoteException e) {

binder = null;

}

}

上面这段代码主要是在StorageManagerService初始化的时候建立和vold的binder通信,并设置binder死亡后重新连接机制.

2.设置binder通信机制

if (binder != null) {

mVold = IVold.Stub.asInterface(binder);

try {

mVold.setListener(mListener);

} catch (RemoteException e) {

mVold = null;

Slog.w(TAG, "vold listener rejected; trying again", e);

}

} else {

Slog.w(TAG, "vold not found; trying again");

}

在上面的代码中,mVold其实是IVold对象,是一个代理类,VoldNativeService.cpp这个文件,是IVold的具体实现.通过binder机制,获取mVold对象,从而实现binder通信.例如当mVold这个代理对象去调用mount方法,通过binder机制最终调用到VoldNativeService的mount方法.从而实现跨进程通信.

mVold.setListener(mListener).mListener是IVoldListener的具体实现.我们先看一下代码:

private final IVoldListener mListener = new IVoldListener.Stub() {

@Override

public void onDiskCreated(String diskId, int flags) {

synchronized (mLock) {

final String value = SystemProperties.get(StorageManager.PROP\_ADOPTABLE);

switch (value) {

case "force\_on":

flags |= DiskInfo.FLAG\_ADOPTABLE;

break;

case "force\_off":

flags &= ~DiskInfo.FLAG\_ADOPTABLE;

break;

}

mDisks.put(diskId, new DiskInfo(diskId, flags));

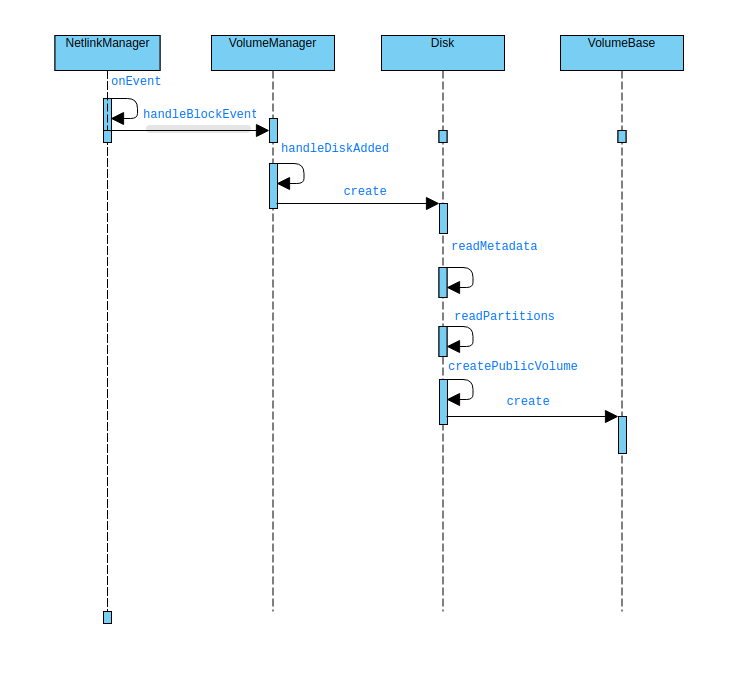
}

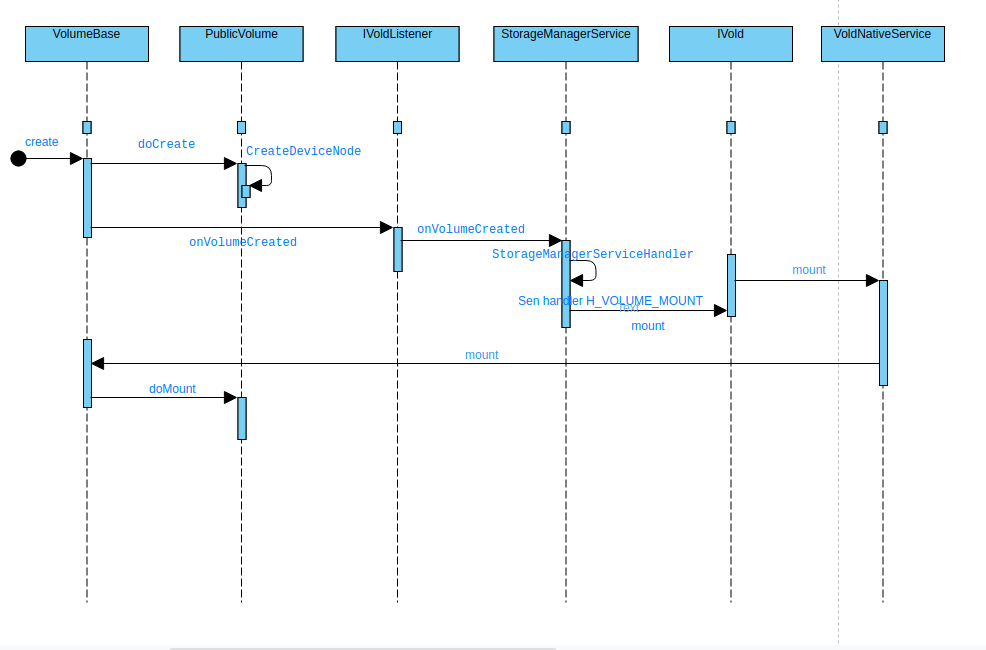
}

这等于设置了一个监听,当vold进程中相关event改变时,就会通过IVoldListener这个代理,通知StorageManagerService处理,并通知上层.

这样,StorageManagerService和Vold进程启动并建立和通信.

下面我们分析一下设备在Android9.0的挂载流程:先看几张时序图:





下面从代码的角度去分析设备加载流程:

UEvent:在linux平台上，uevent给系统软件提供设备事件，设备节点的权限管理等等，它由kernel发出。通过socket与udev守护进程通讯（systemd-udevd.service），

在sysfs下的很多kobject下都有uevent属性，它主要用于内核与udev(自动设备发现程序)之间的一个通信接口，应用软件，也可以直接使用该uevent文件做作为一

个驱动接口。uevent是具有一定格式的字符串，包含一些内容，如：action: {add, change, remove,} devpath设备节点号{major,minor}等。通过对uevent的解析，就能得到kernel发生了那些设备事件。

我们知道Vold服务初始化的时候执行main函数,会初始化NetlinkManager,执行start方法,其中会创建一个NetlinkHandler,当底层Kernel发送消息Uevent,经过一系列处理,最终调用VolumeManager的handleBlockEvent处理消息.NetlinkManager启动后就是创建一个可以接收Kernel消息的socket，并以此socket构建并启动NetlinkHandler。   
void NetlinkHandler::onEvent(NetlinkEvent\* evt) {

VolumeManager\* vm = VolumeManager::Instance();

const char\* subsys = evt->getSubsystem();

if (!subsys) {

LOG(WARNING) << "No subsystem found in netlink event";

return;

}

if (std::string(subsys) == "block") {

vm->handleBlockEvent(evt);

}

}

void VolumeManager::handleBlockEvent(NetlinkEvent\* evt) {

std::lock\_guard<std::mutex> lock(mLock);

if (mDebug) {

LOG(DEBUG) << "----------------";

LOG(DEBUG) << "handleBlockEvent with action " << (int)evt->getAction();

evt->dump();

}

std::string eventPath(evt->findParam("DEVPATH") ? evt->findParam("DEVPATH") : "");

std::string devType(evt->findParam("DEVTYPE") ? evt->findParam("DEVTYPE") : "");

if (devType != "disk") return;

int major = std::stoi(evt->findParam("MAJOR"));

int minor = std::stoi(evt->findParam("MINOR"));

dev\_t device = makedev(major, minor);

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 if (eventPath.find("ufs") != std::string::npos) {

flags |= android::vold::Disk::Flags::kSd;

flags |= android::vold::Disk::Flags::kUfsCard;

} else {

flags |= android::vold::Disk::Flags::kUsb;

}

auto disk =

new android::vold::Disk(eventPath, device, source->getNickname(), flags);

handleDiskAdded(std::shared\_ptr<android::vold::Disk>(disk));

break;

}

}

break;

}

在handleBlockEvent里会首先new一个新的Disk出来,然后执行handleDiskAdded函数,这个函数是处理设备的挂载,初始化我们新new出来的Disk设备.在这个函数里会真正的开始创建并初始化Disk.

void VolumeManager::handleDiskAdded(const std::shared\_ptr<android::vold::Disk>& disk) {

// For security reasons, if secure keyguard is showing, wait

// until the user unlocks the device to actually touch it

if (mSecureKeyguardShowing) {

LOG(INFO) << "Found disk at " << disk->getEventPath()

<< " but delaying scan due to secure keyguard";

mPendingDisks.push\_back(disk);

} else {

disk->create();

mDisks.push\_back(disk);

}

}

status\_t Disk::create() {

CHECK(!mCreated);

mCreated = true;

auto listener = VolumeManager::Instance()->getListener();

if (listener) listener->onDiskCreated(getId(), mFlags);

readMetadata();

readPartitions();

return OK;

}

在Disk.cpp的create方法中,通过VolumeManager的实例对象获得binder通信时建立的Listener监听对象,然后通过这个Listener的函数onDiskCreated通知上层service磁盘已经创建. 然后执行readMetadata()和readPartitiongs()检查文件格式和读取分区列表.在函数readPartittions()中,最终调用createPublicVolume()函数去创建设备的Volume信息.

void Disk::createPublicVolume(dev\_t device) {

auto vol = std::shared\_ptr<VolumeBase>(new PublicVolume(device));

if (mJustPartitioned) {

LOG(DEBUG) << "Device just partitioned; silently formatting";

vol->setSilent(true);

vol->create();

vol->format("auto");

vol->destroy();

vol->setSilent(false);

}

mVolumes.push\_back(vol);

vol->setDiskId(getId());

vol->create();

}

vol -> create()实际上调用VolumeBase的create方法.由于PublicVolume是VolumeBase的子类,且具体实现VolumeBase的方法,这里其实调用的是PublicVolume的doCreate方法.

status\_t VolumeBase::create() {

CHECK(!mCreated);

mCreated = true;

status\_t res = doCreate();

auto listener = getListener();

if (listener) {

listener->onVolumeCreated(getId(), static\_cast<int32\_t>(mType), mDiskId, mPartGuid);

}

setState(State::kUnmounted);

return res;

}

这里在创建完volume信息后,同样获得binder通信建立起来时候的Listener对象,通知上层StorageManagerService创建volume.

根据时序图的流程,下面开始分析StorageManagerService的流程,当通过binder机制通知上层volume创建后,StorageManagerService的onVolumeCreate会收到消息.最终会调用onVolumeCreatedLocked函数

@Override

public void onVolumeCreated(String volId, int type, String diskId, String partGuid) {

synchronized (mLock) {

final DiskInfo disk = mDisks.get(diskId);

final VolumeInfo vol = new VolumeInfo(volId, type, disk, partGuid);

mVolumes.put(volId, vol);

onVolumeCreatedLocked(vol);

}

}

@GuardedBy("mLock")

private void onVolumeCreatedLocked(VolumeInfo vol) {

if (mPmInternal.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.class);

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();

} else if (vol.type == VolumeInfo.TYPE\_STUB) {

vol.mountUserId = mCurrentUserId;

mHandler.obtainMessage(H\_VOLUME\_MOUNT, vol).sendToTarget();

} else {

Slog.d(TAG, "Skipping automatic mounting of " + vol);

}

}

发送设备挂载信息,是通过StorageManagerService中的内部类StorageManagerServiceHandler进行发送的.

case H\_VOLUME\_MOUNT: {

final VolumeInfo vol = (VolumeInfo) msg.obj;

if (isMountDisallowed(vol)) {

Slog.i(TAG, "Ignoring mount " + vol.getId() + " due to policy");

break;

}

mount(vol);

break;

}

@Override

public void mount(String volId) {

enforcePermission(android.Manifest.permission.MOUNT\_UNMOUNT\_FILESYSTEMS);

final VolumeInfo vol = findVolumeByIdOrThrow(volId);

if (isMountDisallowed(vol)) {

throw new SecurityException("Mounting " + volId + " restricted by policy");

}

mount(vol);

}

private void mount(VolumeInfo vol) {

try {

mVold.mount(vol.id, vol.mountFlags, vol.mountUserId);

} catch (Exception e) {

Slog.wtf(TAG, e);

}

}

根据vold与framework的binder通信机制.mVold.mount在VoldNativeService中处理. 这样代码逻辑又从framework层转移到了native.

binder::Status VoldNativeService::mount(const std::string& volId, int32\_t mountFlags,

int32\_t mountUserId) {

ENFORCE\_UID(AID\_SYSTEM);

CHECK\_ARGUMENT\_ID(volId);

ACQUIRE\_LOCK;

auto vol = VolumeManager::Instance()->findVolume(volId);

if (vol == nullptr) {

return error("Failed to find volume " + volId);

}

vol->setMountFlags(mountFlags);

vol->setMountUserId(mountUserId);

int res = vol->mount();

if (res != OK) {

return translate(res);

}

if ((mountFlags & MOUNT\_FLAG\_PRIMARY) != 0) {

res = VolumeManager::Instance()->setPrimary(vol);

if (res != OK) {

return translate(res);

}

}

return translate(OK);

}

实际上调用了调用VolumeBase的mount函数

调用子类的doMount()函数,其实调用PublicVolume::doMount函数.

status\_t VolumeBase::mount() {

if ((mState != State::kUnmounted) && (mState != State::kUnmountable)) {

LOG(WARNING) << getId() << " mount requires state unmounted or unmountable";

return -EBUSY;

}

setState(State::kChecking);

status\_t res = doMount();

setState(res == OK ? State::kMounted : State::kUnmountable);

return res;

}

status\_t PublicVolume::doMount() {

readMetadata();

if (mFsType == "vfat" && vfat::IsSupported()) {

if (vfat::Check(mDevPath)) {

LOG(ERROR) << getId() << " failed filesystem check";

return -EIO;

}

} else if (mFsType == "exfat" && exfat::IsSupported()) {

if (exfat::Check(mDevPath)) {

LOG(ERROR) << getId() << " failed filesystem check";

return -EIO;

}

} else if (mFsType == "ntfs" && ntfs::IsSupported()) {

if (ntfs::Check(mDevPath)) {

LOG(ERROR) << getId() << " failed filesystem check";

return -EIO;

}

} else {

LOG(ERROR) << getId() << " unsupported filesystem " << mFsType;

return -EIO;

}

至此,设备的挂载结束.