Android 6.0 Reboot 流程源代码分析

Shutdown 跟 reboot流程很类似，所以这里以reboot分析：

reboot的类型：

1、手动长按power键选择reboot;

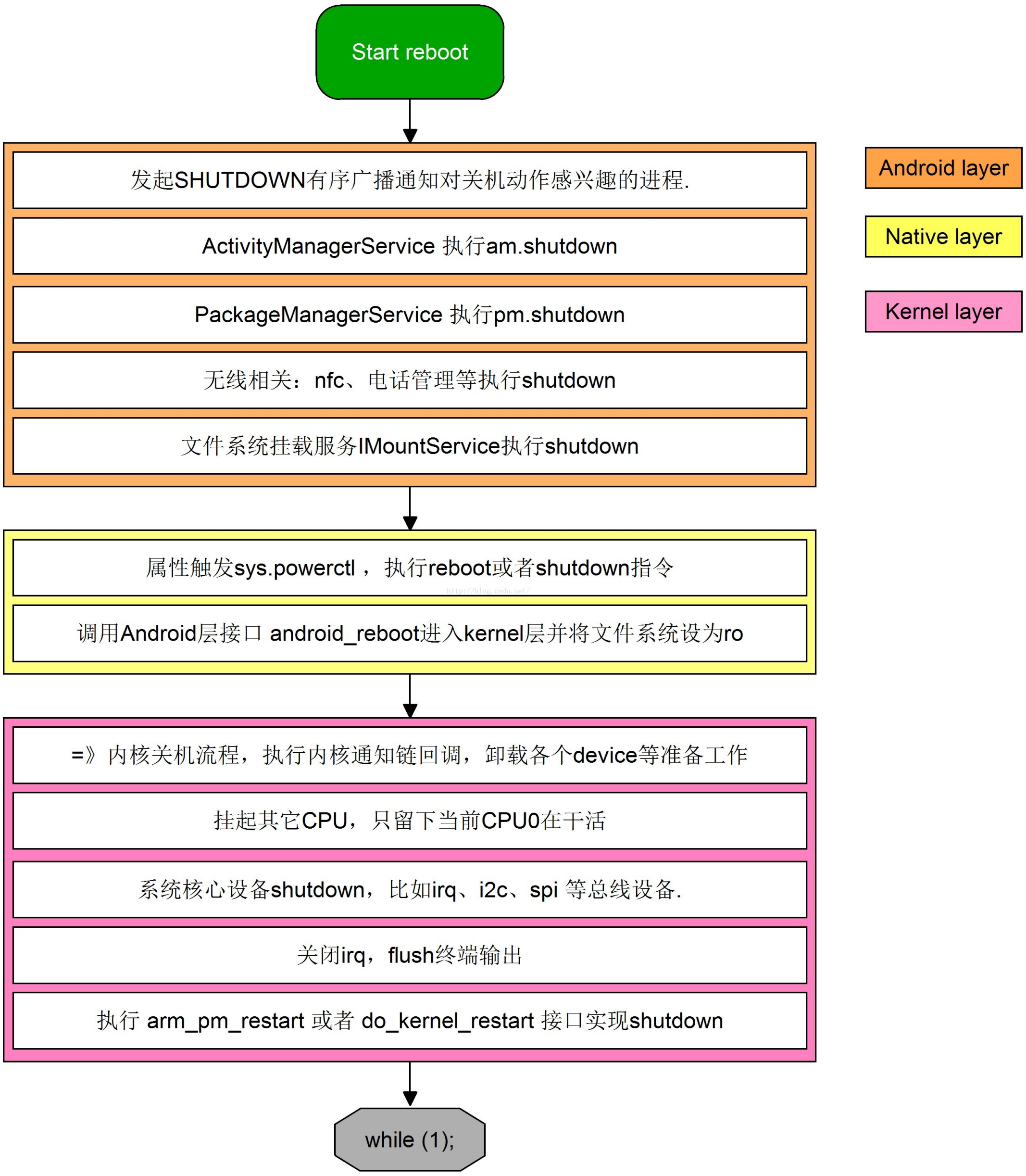
2、adb reboot;

3、手动长按power键11s触发reboot;

4、BUG_ON(1)，触发kernel panic流程reboot;

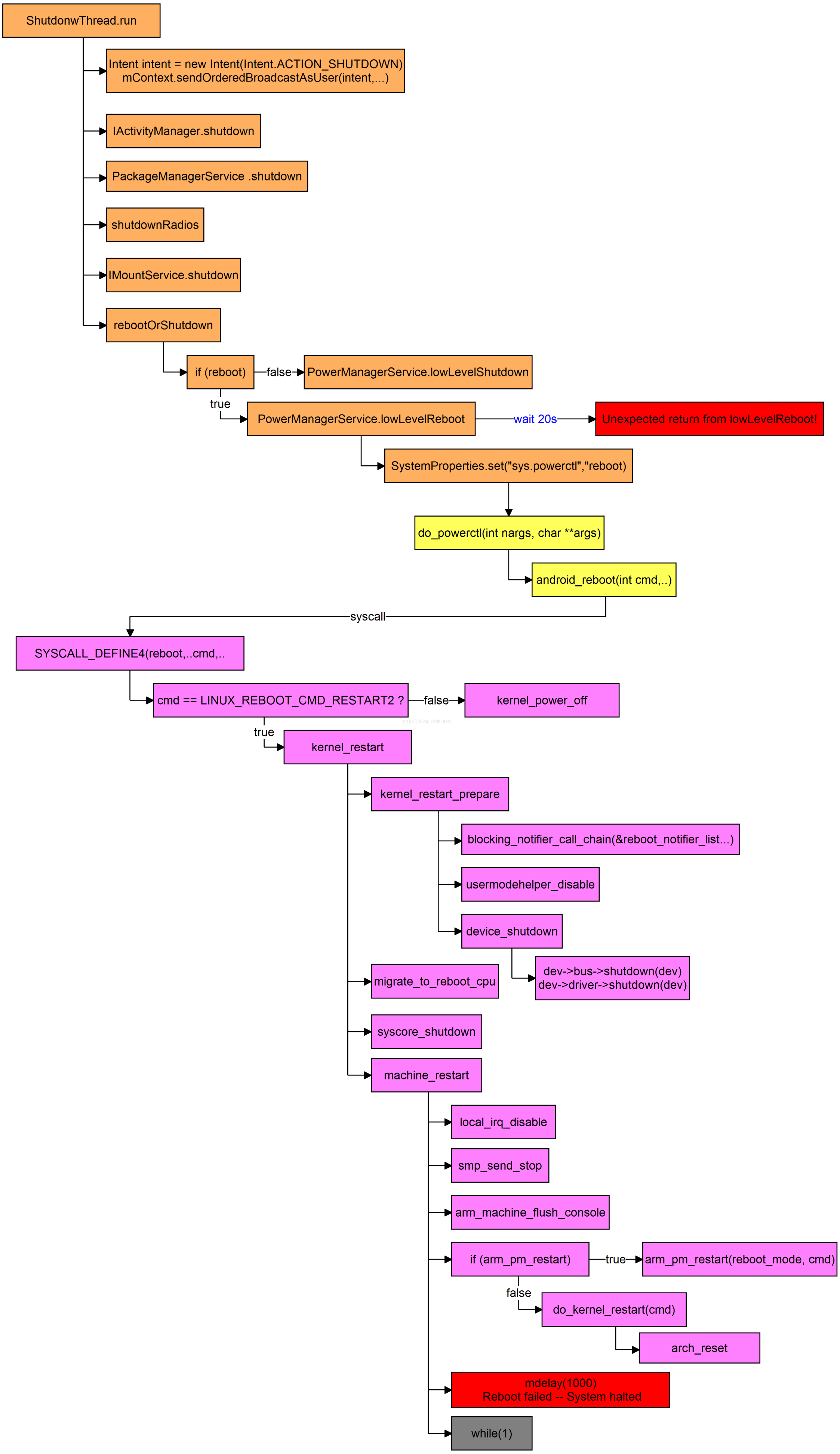
上面1、2、4的本质上代码跑的是一样的，3 是直接触发hardware实现，下面主要分析第1类正常的关键源码流程。

关机逻辑流程总图：



Start reboot
发起SHUTDOWN有序广播通知对关机动作感兴趣的进程.
ActivityManagerService 执行am.shutdown
PackageManagerService 执行pm.shutdown
无线相关：nfc、电话管理等执行shutdown
文件系统挂载服务IMountService执行shutdown
属性触发sys.powerctl， 执行reboot或者shutdown指令
调用Android层接口 android_reboot进入kernel层并将文件系统设为ro
=》内核关机流程，执行内核通知链回调，卸载各个device等准备工作
挂起其它CPU，只留下当前CPU0在干活
系统核心设备shutdown，比如irq、i2c、spi 等总线设备.
关闭irq，flush终端输出
执行 arm_pm_restart 或者 do_kernel_restart 接口实现shutdown
while (1);
Android layer
Native layer
Kernel layer

源代码路径流程图：


ShutdonwThread.run
Intent intent = new Intent(Intent.ACTION_SHUTDOWN)
mContext.sendOrderedBroadcastAsUser(intent,...)
IActivityManager.shutdown
PackageManagerService .shutdown
shutdownRadios
IMountService.shutdown
rebootOrShutdown
if (reboot)
false
PowerManagerService.lowLevelShutdown
true
PowerManagerService.lowLevelReboot
wait 20s
Unexpected return from lowLevelReboot!
SystemProperties.set("sys.powerctl","reboot)
do_powerctl(int nargs, char **args)
android_reboot(int cmd,..)
syscall
SYSCALL_DEFINE4(reboot,..cmd,..
cmd == LINUX_REBOOT_CMD_RESTART2 ?
false
kernel_power_off
true
kernel_restart
kernel_restart_prepare
blocking_notifier_call_chain(&reboot_notifier_list...)
usermodehelper_disable
device_shutdown
dev->bus->shutdown(dev)
dev->driver->shutdown(dev)
migrate_to_reboot_cpu
syscore_shutdown
machine_restart
local_irq_disable
smp_send_stop
arm_machine_flush_console
if (arm_pm_restart)
true
arm_pm_restart(reboot_mode, cmd)
false
do_kernel_restart(cmd)
arch_reset
mdelay(1000)
Reboot failed -- System halted
while(1)

一、首先看Android层.

1、长按power键选择reboot必定走以下接口：

下面开始进入reboot 前的准备工作，大致分为发起**有序shutdown广播、执行Activity、安装包管理、无线相关、挂载服务**等组件的shutdown工作.

```
ShutdownThread.java
public void run() {
//SHUTDOWN有序广播结果接受
        BroadcastReceiver br = new BroadcastReceiver() {
            @Override public void onReceive(Context context, Intent intent) {
                actionDone();
            }
        };
...
//发送SHUTDOWN有序广播，注意是同步的，如果被阻塞将会block住main thread.
        Intent intent = new Intent(Intent.ACTION_SHUTDOWN);
        intent.addFlags(Intent.FLAG_RECEIVER_FOREGROUND);
        mContext.sendOrderedBroadcastAsUser(intent,
                UserHandle.ALL, null, br, mHandler, 0, null, null);

//等待有序广播全部处理完成,也就是等上面的br.onReveive回调.
        synchronized (mActionDoneSync) {
            while (!mActionDone) {
...
                try {
                    mActionDoneSync.wait(Math.min(delay, PHONE_STATE_POLL_SLEEP_MSEC));
                } catch (InterruptedException e) {
                }
            }
        }
...
//ActivityManagerService执行shutdown操作，写一些相关状态(比如battery)记录到文件.
        final IActivityManager am =
            ActivityManagerNative.asInterface(ServiceManager.checkService("activity"));
        if (am != null) {
            try {
                am.shutdown(MAX_BROADCAST_TIME);
            } catch (RemoteException e) {
            }
        }
...
//安装包管理服务执行shutdonw，将当前的packageName写入data/system目录文件中.
        final PackageManagerService pm = (PackageManagerService)
            ServiceManager.getService("package");
        if (pm != null) {
            pm.shutdown();
        }
...

//无线相关执行shutdown，比如nfc、电话服务相关等.
        shutdownRadios(MAX_RADIO_WAIT_TIME);
...

//挂载服务卸载完成回调
        IMountShutdownObserver observer = new IMountShutdownObserver.Stub() {
            public void onShutDownComplete(int statusCode) throws RemoteException {
                actionDone();
            }
        };

//执行文件系统挂载服务卸载
        synchronized (mActionDoneSync) {
            try {
                final IMountService mount = IMountService.Stub.asInterface(
                        ServiceManager.checkService("mount"));
                    mount.shutdown(observer);
            } catch (Exception e) {
                Log.e(TAG, "Exception during MountService shutdown", e);
            }

// 等待卸载完成，也就是等上面的 observer.onShutDownComplete执行完
            while (!mActionDone) {
            ...
                try {
                    mActionDoneSync.wait(Math.min(delay, PHONE_STATE_POLL_SLEEP_MSEC));
                } catch (InterruptedException e) {
                }
            }
        }
...
//准备工作完成，进入正式reboot流程
        rebootOrShutdown(mContext, mReboot, mRebootReason);
    }
```

继续分析准备工作后的reboot流程，主要要干的事情就是把shutdown或者reboot的command从Java层传到native层的reboot接口.

```
public static void rebootOrShutdown(final Context context, boolean reboot, String reason) {
//如果是重启的话就执行lowLevelReboot，否则就执行lowLevelShutdown接口
        if (reboot) {
            PowerManagerService.lowLevelReboot(reason);
        } else if (SHUTDOWN_VIBRATE_MS > 0 && context != null) {
//如果是关机命令，则会振动500ms提示
            Vibrator vibrator = new SystemVibrator(context);
            try {
                vibrator.vibrate(SHUTDOWN_VIBRATE_MS, VIBRATION_ATTRIBUTES);
            } catch (Exception e) {
            }
//等500ms待vib完成再进入shutdown.
            try {
                Thread.sleep(SHUTDOWN_VIBRATE_MS);
            } catch (InterruptedException unused) {
            }
        }
        PowerManagerService.lowLevelShutdown();
    }

public static void lowLevelReboot(String reason) {
...
//使用属性服务传入cmd触发reboot的 Action
    SystemProperties.set("sys.powerctl", "reboot," + reason);

//等待20s，也就是说20s内需要关机完成
    try {
        Thread.sleep(20 * 1000L);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

 //下面这条log很关键！！！，如果这条log打出来了，就说明关机失败了，需要找原因了...
    Slog.wtf(TAG, "Unexpected return from lowLevelReboot!");
}

//如果执行的是shutdown则走执行下面的cmd
public static void lowLevelShutdown() {
    SystemProperties.set("sys.powerctl", "shutdown");
}
```

Java层关机流程分析到此结束，进入native层，我们知道，属性服务贯穿整个Android系统可以很方便的触发各种Action、启动服务等，那么这里的SystemProperties.set("sys.powerctl", "reboot," + reason)到底干了什么事情呢？这个需要从init.rc找答案(属性服务触发实现机制暂不讨论)。搜索sys.powerctl关键字：

```
./rootdir/init.rc:544:on property:sys.powerctl=*
./rootdir/init.rc:545:    powerctl ${sys.powerctl}
```

这是一个on 的action，意思是当sys.powerctl的值改变时，执行powerctl命令，参数就是${sys.powerctl}，此处就是上面的reboot,那么具体是什么呢？搜索powerctl会发现：

```
./init/keywords.h:17:int do_powerctl(int nargs, char **args);
./init/keywords.h:79:    KEYWORD(powerctl,    COMMAND, 1, do_powerctl)
```

很显然其实就是代表的do_powerctl函数！简单来说就是执行SystemProperties.set("sys.powerctl", "reboot," + reason) 函数的时候其实就是会最终下面的函数：

```
int do_powerctl(int nargs, char **args)
{
    char command[PROP_VALUE_MAX];
...

    res = expand_props(command, args[1], sizeof(command));
...
    if (strncmp(command, "shutdown", 8) == 0) {
        cmd = ANDROID_RB_POWEROFF;
        len = 8;
    } else if (strncmp(command, "reboot", 6) == 0) {
        cmd = ANDROID_RB_RESTART2;
        len = 6;
    } else {
        ERROR("powerctl: unrecognized command '%s'\n", command);
        return -EINVAL;
    }
...
//很简单，就是解析出要下发哪一个cmd，这里显然就是ANDROID_RB_RESTART2了，接着
//调用android层最后一个函数接口
    return android_reboot(cmd, 0, reboot_target);
}

int android_reboot(int cmd, int flags UNUSED, const char *arg)
{
    int ret;

// 将缓冲区数据写回磁盘,保证数据同步.
    sync();

//把filesystem置为read only，不允许proc再往里面写东西.
    remount_ro();

//下面就是reboot的system call进入内核空间了:

    switch (cmd) {
        case ANDROID_RB_RESTART:
            ret = reboot(RB_AUTOBOOT);
            break;

        case ANDROID_RB_POWEROFF:
            ret = reboot(RB_POWER_OFF);
            break;

        case ANDROID_RB_RESTART2:
            ret = syscall(__NR_reboot, LINUX_REBOOT_MAGIC1, LINUX_REBOOT_MAGIC2,
                           LINUX_REBOOT_CMD_RESTART2, arg);
            break;

        default:
            ret = -1;
    }

    return ret;
}
```

二、Android层关机流程分析完成，进入内核层分析，执行系统调用后进入kernel层系统调用入口：（系统调用是用户程序请求内核服务的标准形式，这里我们不去关注其具体实现）

```
SYSCALL_DEFINE4(reboot, int, magic1, int, magic2, unsigned int, cmd,
  void __user *, arg)
{
...
//忽略前头一堆各种检查细节，关注reboot流程主线.

//互斥锁，保证当前就一个CPU在执行此路径.
 mutex_lock(&reboot_mutex);
 switch (cmd) {
 case LINUX_REBOOT_CMD_RESTART:
  kernel_restart(NULL);
  break;
...
 case LINUX_REBOOT_CMD_HALT:
  kernel_halt();
  do_exit(0);
  panic("cannot halt");

 case LINUX_REBOOT_CMD_POWER_OFF:
  kernel_power_off();
  do_exit(0);
  break;

 case LINUX_REBOOT_CMD_RESTART2:
  ret = strncpy_from_user(&buffer[0], arg, sizeof(buffer) - 1);
  if (ret < 0) {
   ret = -EFAULT;
   break;
  }
  buffer[sizeof(buffer) - 1] = '\0';
//进入内核restart入口函数
  kernel_restart(buffer);
  break;
...
 mutex_unlock(&reboot_mutex);
 return ret;
}
```

kernel_restart 函数要干的事情主要分为几部分：

```
void kernel_restart(char *cmd)
{
//kernel关机准备工作.
 kernel_restart_prepare(cmd);

//挂起其他cpu的工作，只留下当前cpu干活
 migrate_to_reboot_cpu();

//核心设备执行shutdown，比如PM,irq，usb等.
 syscore_shutdown();
 if (!cmd)
  pr_emerg("Restarting system\n");
 else
  pr_emerg("Restarting system with command '%s'\n", cmd);
 kmsg_dump(KMSG_DUMP_RESTART);

//执行各个体系结构相关的关机、restart操作实现
 machine_restart(cmd);
}
```

kernel_restart_prepare 分析，主要干了两件事情：发通知给感兴趣的dev + 执行dev卸载

```
void kernel_restart_prepare(char *cmd)
{
//发cmd给通知链中对SYS_RESTART感兴趣的设备，执行nofifier回调.
 blocking_notifier_call_chain(&reboot_notifier_list, SYS_RESTART, cmd);
 system_state = SYSTEM_RESTART;

//用户模式 disable ?
 usermodehelper_disable();

//设备卸载
 device_shutdown();
}
```

这里需要重点分析下**device_shutdown**函数，如果该函数stuck，会导致无法关机.

```
void device_shutdown(void)
{
 struct device *dev, *parent;

//自旋锁，关抢断.
 spin_lock(&devices_kset->list_lock);
 /*
  * Walk the devices list backward, shutting down each in turn.
  * Beware that device unplug events may also start pulling
  * devices offline, even as the system is shutting down.
  */
 while (!list_empty(&devices_kset->list)) {
//从device链表使用"内核中经典大法-从实例找容器方式" 遍历各个dev
  dev = list_entry(devices_kset->list.prev, struct device,
    kobj.entry);
  /*
   * hold reference count of device's parent to
   * prevent it from being freed because parent's
   * lock is to be held
   */

//激活parent dev和dev，这get,put名字起的容易让人误解，汗..
  parent = get_device(dev->parent);
  get_device(dev);
  /*
   * Make sure the device is off the kset list, in the
   * event that dev->*->shutdown() doesn't remove it.
   */

//把dev从kobj.entry容器中删除
  list_del_init(&dev->kobj.entry);
  spin_unlock(&devices_kset->list_lock);

  /* hold lock to avoid race with probe/release */
  if (parent)
   device_lock(parent);
  device_lock(dev);

//阻止任何的runtime相关的dev挂起
  /* Don't allow any more runtime suspends */
  pm_runtime_get_noresume(dev);

//这个pm runtime相关函数很复杂，暂时没看懂要干什么，汗..
  pm_runtime_barrier(dev);

//执行各个对关机感兴趣dev的shutdown回调函数
  if (dev->bus && dev->bus->shutdown) {
   if (initcall_debug)
    dev_info(dev, "shutdown\n");
   dev->bus->shutdown(dev);
  } else if (dev->driver && dev->driver->shutdown) {
   if (initcall_debug)
    dev_info(dev, "shutdown\n");
   dev->driver->shutdown(dev);
  }

  device_unlock(dev);
  if (parent)
   device_unlock(parent);

//告诉dev，你现在可以挂起了.
  put_device(dev);
  put_device(parent);

  spin_lock(&devices_kset->list_lock);

 }
 spin_unlock(&devices_kset->list_lock);
}
```

下面进入执行**真正的关机**操作：

```
void machine_restart(char *cmd)
{
//关闭中断
 local_irq_disable();

//停掉别的cpu，只留下当前执行的cpu（smp：多对称处理器结构<Symmetrical Multi-Processing>）
 smp_send_stop();


 /* Flush the console to make sure all the relevant messages make it
  * out to the console drivers */
 arm_machine_flush_console();


//arm_pm_restart 是函数指针，指向各个体系结构和芯片厂商具体的restart入口，传参给pmic执行restart或者shutdwon的动作。
//比如高通8937项目对于的就是：do_msm_restart，mtk 6580对应的就是跑默认的接口：
 if (arm_pm_restart)
  arm_pm_restart(reboot_mode, cmd);
 else
  do_kernel_restart(cmd);

//等1s时间，若1s后打印出下面的log就说明shutdwon失败了，正常情况就已经断电关机了.
 mdelay(1000);

 /* Whoops - the platform was unable to reboot. Tell the user! */
 printk("Reboot failed -- System halted\n");
 local_irq_disable();

//如果跑到这里就说明关机失败了.
 while (1);
}
```

Android reboot流程整体比较简单，到此分析完.