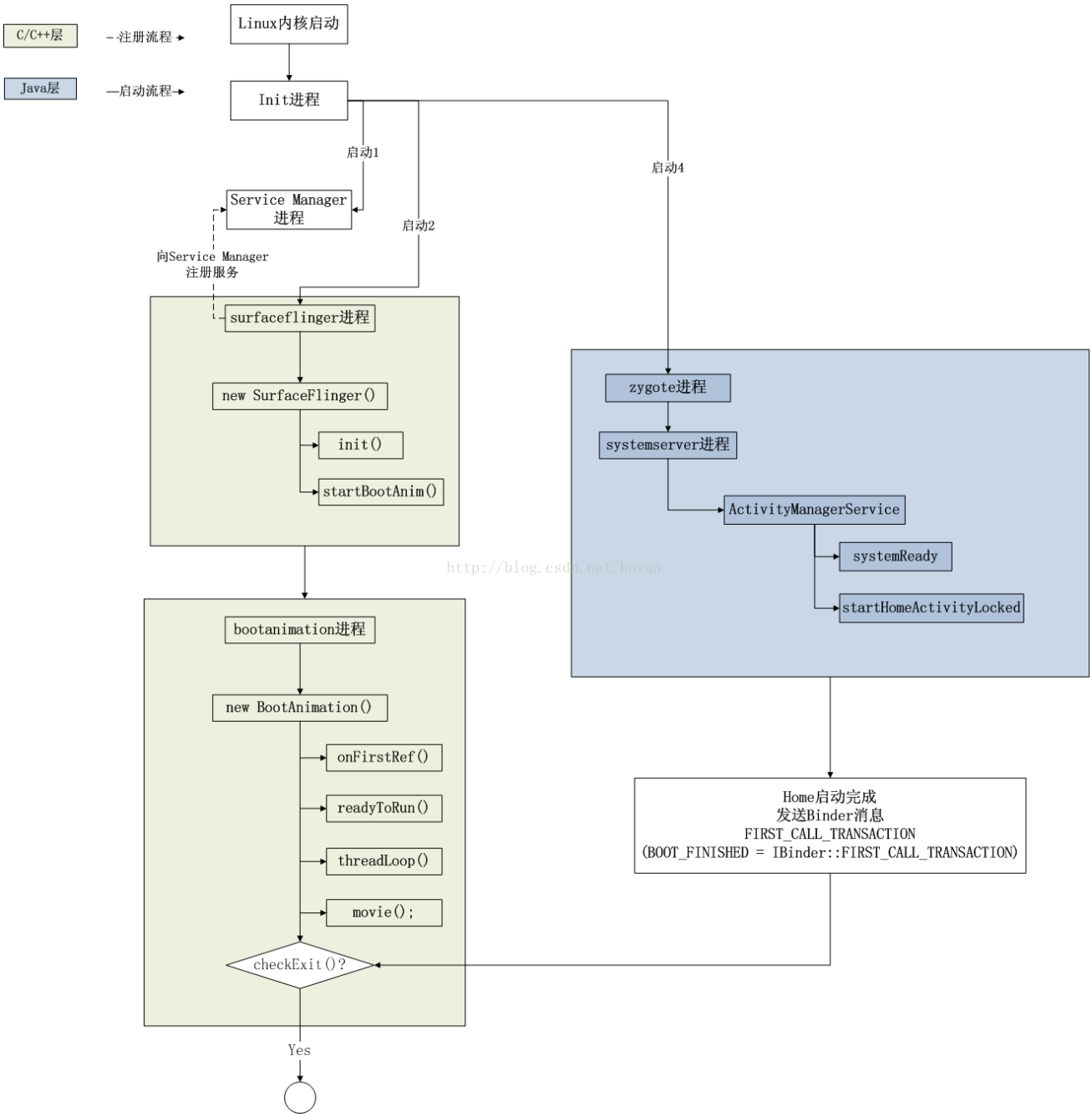
深入解析bootanimation启动流程

注：该讲解中出现的源码是基于高通平台Android 7.1源码

Android启动过程大致流程一般是先加载bootloader，然后启动kernel，启动init进程，加载ServiceManager，紧接着是 开机动画，最后到我们的Home界面，当然其中肯定还有很多细致的东西我们这里就不多讲了，今天我们主要剖析的是开机动画大致流程，详解分析下开机动画的启动，运行，结束。

先来张形象点的流程图



首先了解下SurfaceFlinger，这是一个非常重要的服务进程，基本上所有UI面板的绘制肯定都跟它有关，所以开机动画的启动播放肯定离不开它，它是由init进程通过解析其对应的init.rc脚本来启动surfaceFlinger服务。

看下Surfaceflinger对应的init.rc文件

```
service surfaceflinger /system/bin/surfaceflinger
    class core
    user system
    group graphics drmrpc readproc
    onrestart restart zygote
    writepid /dev/stune/foreground/tasks
```

接下来看下SurfaceFlinger的入口main函数干了什么，main函数在Main_surfaceflinger.cpp中

Main_surfaceflinger.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)

```
int main(int, char**) {
    signal(SIGPIPE, SIG_IGN);
    // When SF is launched in its own process, limit the number of
    // binder threads to 4.
    ProcessState::self()->setThreadPoolMaxThreadCount(4);//将SurfaceFlinger的线程池大小设为4

    // start the thread pool
    //构造获取ProcessState的实例，并启动线程池
    sp<ProcessState> ps(ProcessState::self());
    ps->startThreadPool();

    // instantiate surfaceflinger
    /*获取一个SurfaceFlinger的实例flinger*/
    sp<SurfaceFlinger> flinger = DisplayUtils::getInstance()->getSFInstance();

    setpriority(PRIO_PROCESS, 0, PRIORITY_URGENT_DISPLAY);

    set_sched_policy(0, SP_FOREGROUND);

    ......

    // initialize before clients can connect
    flinger->init();//调用SurfaceFlinger的初始化函数函数init

    // publish surface flinger
    sp<IServiceManager> sm(defaultServiceManager());
    sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false);//将flinger添加到
ServiceManager中

    ......

    // run surface flinger in this thread
    flinger->run();//调用SurfaceFlinger的run方法

    return 0;
}
```

接下来我们讲解下SurfaceFlinger构造启动相关的这几个函数：

1. SurfaceFlinger构造：sp<SurfaceFlinger> flinger = DisplayUtils::getInstance()->getSFInstance();

源文件:DisplayUtils.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)

DisplayUtils的getInstance构造获取一个DisplayUtils的实例

```
DisplayUtils* DisplayUtils::getInstance() {
    if(sDisplayUtils == NULL) {
        sDisplayUtils = new DisplayUtils();
    }
    return sDisplayUtils;
}
```

getSFInstance方法中会**new**一个SurfaceFlinger实例，sUseExtendedImpls默认为false

```
SurfaceFlinger* DisplayUtils::getSFInstance() {
    if(sUseExtendedImpls) {
        return new ExSurfaceFlinger();
    } else {
        return new SurfaceFlinger();
    }
```

接下来看下SurfaceFlinger类和它构造函数

```
//SurfaceFlinger.h (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)
class SurfaceFlinger : public BnSurfaceComposer,
                       private IBinder::DeathRecipient,
                       private HWComposer::EventHandler
{
```

```
......
}
```
从上面代码可以看出SurfaceFlinger继承了BnSurfaceComposer，DeathRecipient， EventHandler这三个类

```
//SurfaceFlinger.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger
SurfaceFlinger::SurfaceFlinger()
    :   BnSurfaceComposer(),
        mTransactionFlags(0),
        mTransactionPending(false),
        mAnimTransactionPending(false),
        mLayersRemoved(false),
        mRepaintEverything(0),
        mRenderEngine(NULL),
        mBootTime(systemTime()),
        mBuiltinDisplays(),
        mVisibleRegionsDirty(false),
        mGeometryInvalid(false),
        mAnimCompositionPending(false),
        mDebugRegion(0),
        mDebugDDMS(0),
        mDebugDisableHWC(0),
        mDebugDisableTransformHint(0),
        mDebugInSwapBuffers(0),
        mLastSwapBufferTime(0),
        mDebugInTransaction(0),
        mLastTransactionTime(0),
        mBootFinished(false),
        mForceFullDamage(false),
        mPrimaryDispSync("PrimaryDispSync"),
        mPrimaryHWVsyncEnabled(false),
        mHWVsyncAvailable(false),
        mHasColorMatrix(false),
        mHasPoweredOff(false),
        mFrameBuckets(),
        mTotalTime(0),
        mLastSwapTime(0)
{
    ALOGI("SurfaceFlinger is starting");
    ......
}
```
SurfaceFlinger的构造函数可以看出主要是一大堆的成员变量的初始化，还有一些属性值的获取。这里我们主要关注和注意的是SurfaceFlinger的构造函数中调用了父类的构造函数BnSurfaceComposer();

所以我们看下BnSurfaceComposer这个类
```
//ISurfaceComposer.h (la.um.5.6\linux\android\frameworks\native\include\gui)
class BnSurfaceComposer: public BnInterface<ISurfaceComposer> {
public:
    enum {
        // Note: BOOT_FINISHED must remain this value, it is called from
        // Java by ActivityManagerService.
        BOOT_FINISHED = IBinder::FIRST_CALL_TRANSACTION,
        ......
        SET_ACTIVE_COLOR_MODE,
    };

    virtual status_t onTransact(uint32_t code, const Parcel& data,
            Parcel* reply, uint32_t flags = 0);
};
```
BnSurfaceComposer是一个BnInterface类，本地服务接口类，可想而知，肯定还有供客户端使用的BpSurfaceComposer

好了，SurfaceFlinger初始化过程先说到这个。

1. SurfaceFlinger初始化：flinger->init();

先看下init的源码实现
```
//SurfaceFlinger.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)
void SurfaceFlinger::init() {
    ALOGI(  "SurfaceFlinger's main thread ready to run. "
```

```
            "Initializing graphics H/W...");

    {
    ......
    // start boot animation
    startBootAnim();//重要入口，开机动画启动入口函数
    ALOGV("Done initializing");
}
```

以上是init函数的相关代码，这么一大堆主要是绘制相关类的初始化，我们需要关心的是startBootAnim()这个函数，我们来看下这个函数的具体实现

```
//SurfaceFlinger.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)

void SurfaceFlinger::startBootAnim() {
    // start boot animation
    property_set("service.bootanim.exit", "0");
    property_set("ctl.start", "bootanim");
}
```

startBootAnim()函数这里主要是设置了两个属性值：

“service.bootanim.exit”这个属性值是用来标记开机动画的开始和退出，0表示开始，1表示退出

“ctl.start”设置成“bootanim”表示通过属性服务(property Service)的ctl.start命令来启动bootanim进程（/bin/bootanim）

关于属性服务的相关知识这里就不细讲了麻烦查阅相关资料

现在基本知道bootanim是由SurfaceFlinger启动的，接下来我们看下bootanim这个进程 先看下它的init.rc脚本

```
//Bootanim.rc (la.um.5.6\linux\android\frameworks\base\cmds\bootanimation)
service bootanim /system/bin/bootanimation
    class core
    user media
    group graphics audio
    disabled
    oneshot
    writepid /dev/stune/top-app/tasks
```

从rc脚本可以看出，bootanim默认是disable状态的，所以init进程解析rc阶段并没有启动它

接下来看下bootanim的入口main函数

```
Bootanimation_main.cpp (la.um.5.6\linux\android\frameworks\base\cmds\bootanimation)
int main()
{
    setpriority(PRIO_PROCESS, 0, ANDROID_PRIORITY_DISPLAY);

    char value[PROPERTY_VALUE_MAX];
    property_get("debug.sf.nobootanimation", value, "0");
    int noBootAnimation = atoi(value);
    ALOGI_IF(noBootAnimation,  "boot animation disabled");
    if (!noBootAnimation) {

        //构造初始化ProcessState函数
        sp<ProcessState> proc(ProcessState::self());

        //启动线程池
        ProcessState::self()->startThreadPool();

        // create the boot animation object
        //构造一个BootAnimation实例
        sp<BootAnimation> boot = new BootAnimation();

        //将主线程本身加入线程池当中
        IPCThreadState::self()->joinThreadPool();

    }
    return 0;
}
```

我们重点看下sp<BootAnimation> boot = new BootAnimation();

boot变量的前缀是sp，就说明它是一个强引用，sp是相对于wp而言，也就是还有个弱引用，sp和wp在内核和底层代码中非常常见，其作用是通过引用计数来控制对象的生命周期。

简述RefBase、sp和wp

RefBase是Android中所有对象的始祖，类似于MFC中的CObject及Java中的Object对象。在Android中，RefBase结合sp和wp，实现了一套通过引用计数的方法来控制对象生命周期的机制。就如我们想像的那样，这三者的关系非常暧昧。初次接触Android源码的人往往会被那个随处可见的sp和wp搞晕了头。 什么是sp和wp呢？其实，sp并不是我开始所想的smart pointer（C++语言中有这个东西），它真实的

意思应该是strong pointer，而wp则是weak pointer的意思。我认为，Android推出这一套机制可能是模仿Java，因为Java世界中有所谓weak reference之类的东西。sp和wp的目的，就是为了帮助健忘的程序员回收new出来的内存。 说明　我还是喜欢赤裸裸地管理内存的分配和释放。不过，目前sp和wp的使用已经深入到Android系统的各个角落，想把它去掉真是不太可能了。
我们这里需要阐述的东西是，当一个对象初始化是被sp修饰时，那么回调onFirstRef方法，换句话说onFirstRef（）属于其父类RefBase，该函数在强引用sp新增引用计数时调用，什么意思？就是当 有sp包装的类初始化的时候调用。 感兴趣的同学可以去阅读sp，wp的相关源码
我们具体看下BootAnimation类以及它的构造函数

```
class BootAnimation : public Thread, public IBinder::DeathRecipient
{
public:
                BootAnimation();
    virtual     ~BootAnimation();
......
}
```

构造方法如下

```
BootAnimation::BootAnimation() : Thread(false), mClockEnabled(true), mTimeIsAccurate(false),
        mTimeCheckThread(NULL) {
    mSession = new SurfaceComposerClient();

    // If the system has already booted, the animation is not being used for a boot.
    mSystemBoot = !property_get_bool(BOOT_COMPLETED_PROP_NAME, 0);
}

//mSession的定义如下，注意它也是个sp修饰的变量
sp<SurfaceComposerClient>       mSession;
```

BootAnimation继承了Thread类和DeathRecipient类
在它的构造函数中构造了一个类型为SurfaceComposerClient的成员变量mSession，
这里其实相当于new了一个SurfaceFlinger的代理对象跟便于跟SurfaceFlinger来通信
我们可以具体看下SurfaceComposerClient这个类
[SurfaceComposerClient.cpp (la.um.5.6\linux\android\frameworks\native\libs\gui)]

```
SurfaceComposerClient::SurfaceComposerClient()
    : mStatus(NO_INIT), mComposer(Composer::getInstance())
{
}
```

因为mSession是sp对象，所以初始化SurfaceComposerClient时也会调用onFirstRef函数

```
void SurfaceComposerClient::onFirstRef() {
    sp<ISurfaceComposer> sm(ComposerService::getComposerService());
    if (sm != 0) {
        sp<ISurfaceComposerClient> conn = sm->createConnection();
        if (conn != 0) {
            mClient = conn;
            mStatus = NO_ERROR;
        }
    }
}
```

看下sm是通过ComposerService::getComposerService()来构造初始化

```
/*static*/ sp<ISurfaceComposer> ComposerService::getComposerService() {
    ComposerService& instance = ComposerService::getInstance();
    Mutex::Autolock _l(instance.mLock);
    if (instance.mComposerService == NULL) {
        ComposerService::getInstance().connectLocked();
        assert(instance.mComposerService != NULL);
        ALOGD("ComposerService reconnected");
    }
    return instance.mComposerService;
}
```

getComposerService中调用ComposerService实例的connectLocked方法

```
void ComposerService::connectLocked() {
    const String16 name("SurfaceFlinger");
    while (getService(name, &mComposerService) != NO_ERROR) {
        usleep(250000);
    }
    assert(mComposerService != NULL);
```

```
    // Create the death listener.
    class DeathObserver : public IBinder::DeathRecipient {
        ComposerService& mComposerService;
        virtual void binderDied(const wp<IBinder>& who) {
            ALOGW("ComposerService remote (surfaceflinger) died [%p]",
                  who.unsafe_get());
            mComposerService.composerServiceDied();
        }
     public:
        DeathObserver(ComposerService& mgr) : mComposerService(mgr) { }
    };

    mDeathObserver = new DeathObserver(*const_cast<ComposerService*>(this));
    IInterface::asBinder(mComposerService)->linkToDeath(mDeathObserver);
}
```

connectLocked函数中通过getService方法来获取SurfaceFlinger服务代理对象，并保存到mComposerService变量中，这里还设置了SurfaceFlinger对象死亡监听

现在回过头来，也就说在BootAnimation的构造函数中主要绑定链接了SurfaceFlinger服务，便于后续跟SurfaceFlinger的通信交互

接下来是调用BootAnimation中的onFirstRef函数，这就是为什么前面我们要讲sp的作用，不然你根本不知道BootAnimation初始化后接下来怎么走

```
[BootAnimation.cpp (la.um.5.6\linux\android\frameworks\base\cmds\bootanimation)]
void BootAnimation::onFirstRef() {
    status_t err = mSession->linkToComposerDeath(this);
    ALOGE_IF(err, "linkToComposerDeath failed (%s) ", strerror(-err));
    if (err == NO_ERROR) {
        run("BootAnimation", PRIORITY_DISPLAY);
    }
}
```

在onFirstRef方法中先注册对SurfaceFlinger的死亡监听

linkTocomposerDeath的作用是当surfaceflinger死掉时，BootAnimation就会得到通知,会回调binderDied函数

```
void BootAnimation::binderDied(const wp<IBinder>&)
{
    // woah, surfaceflinger died!
    ALOGD("SurfaceFlinger died, exiting...");

    // calling requestExit() is not enough here because the Surface code
    // might be blocked on a condition variable that will never be updated.
    kill( getpid(), SIGKILL );
    requestExit();
    audioplay::destroy();
}
```

这里可以看出一旦SurfaceFlinger死了，bootanim进程也会死，media也会跟着去

然后调用了run方法，一开始我看到这里也是蒙了下，不知道到了这里再继续往哪走

其实看下BootAnimation的定义就知道，它继承了Thread类

```
#include <utils/Thread.h>
```

这个Thread指的是libutils中的Thread

所以接下来我们需要简单介绍下libutils中Thread类的run方法

```
[Threads.cpp (la.um.5.6\linux\android\system\core\libutils)]
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    ......
    // hold a strong reference on ourself
    mHoldSelf = this;
    mRunning = true;
    bool res;
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    } else {
        res = androidCreateRawThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    }
    ......
```

```
}
```
mCanCallJava默认为false，所以走的是androidCreateRawThreadEtc分支，注意它传了一个_threadLoop的参数

```
int androidCreateRawThreadEtc(android_thread_func_t entryFunction,
                              void *userData,
                              const char* threadName __android_unused,
                              int32_t threadPriority,
                              size_t threadStackSize,
                              android_thread_id_t *threadId)
{
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    ......
    errno = 0;
    pthread_t thread;
    //通过pthread_create来创建线程
    int result = pthread_create(&thread, &attr,
                    (android_pthread_entry)entryFunction, userData);
    ......
    if (threadId != NULL) {
        *threadId = (android_thread_id_t)thread; // XXX: this is not portable
    }
    return 1;
}
```

线程函数_threadLoop介绍
无论一分为二是如何处理的，最终都会调用线程函数_threadLoop，
莫非_threadLoop会有什么操作吗？下面我们来看：
[-->Thread.cpp]

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();


#if HAVE_ANDROID_OS
    self->mTid = gettid();
#endif


    bool first = true;


    do {//进入一个do...while循环
        bool result;
        if (first) {
            first = false;
           //self代表继承Thread类的对象，第一次进来时将调用readyToRun，看看是否准备好。
          self->mStatus = self->readyToRun();
            result = (self->mStatus == NO_ERROR);


            if (result && !self->mExitPending) {
                 result = self->threadLoop();
            }
        } else {
           /*
           调用子类实现的threadLoop函数，注意这段代码运行在一个do-while循环中。
             这表示即使我们的threadLoop返回了，线程也不一定会退出。
          */
            result = self->threadLoop();
        }
   /*
    线程退出的条件：
    1）result 为false。这表明，如果子类在threadLoop中返回false，线程就可以
```

```
        退出。这属于主动退出的情况，是threadLoop自己不想继续干活了，所以返回false。
        读者在自己的代码中千万别写错threadLoop的返回值。
        2）mExitPending为true，这个变量可由Thread类的requestExit函数设置，这种
        情况属于被动退出，因为由外界强制设置了退出条件。
       */
            if (result == false || self->mExitPending) {
                self->mExitPending = true;
                self->mLock.lock();
                self->mRunning = false;
                self->mThreadExitedCondition.broadcast();
                self->mLock.unlock();
                break;//退出循环
            }
            strong.clear();
            strong = weak.promote();
        } while(strong != 0);

        return 0;
    }
```

关于_threadLoop，我们就介绍到这里。请读者务必注意下面一点：
threadLoop运行在一个循环中，它的返回值可以决定是否退出线程。
所以说，当调用调用Thread的run方法时，会依次调用其子类的readyToRun方法，然后调用threadLoop方法
好了，那么接下来我们看BootAnimation中的readyToRun方法

```
status_t BootAnimation::readyToRun() {
    ......
    // create the native surface
    sp<SurfaceControl> control = session()->createSurface(String8("BootAnimation"),
            dinfo.w, dinfo.h, PIXEL_FORMAT_RGB_565);

    SurfaceComposerClient::openGlobalTransaction();
    control->setLayer(0x40000000);
    SurfaceComposerClient::closeGlobalTransaction();

    sp<Surface> s = control->getSurface();

    ......
    mFlingerSurface = s;

    // If the device has encryption turned on or is in process
    // of being encrypted we show the encrypted boot animation.
    char decrypt[PROPERTY_VALUE_MAX];
    property_get("vold.decrypt", decrypt, "");

    bool encryptedAnimation = atoi(decrypt) != 0 || !strcmp("trigger_restart_min_framework", decrypt);

    if (encryptedAnimation && (access(getAnimationFileName(IMG_ENC), R_OK) == 0)) {
        mZipFileName = getAnimationFileName(IMG_ENC);
    }
    else if (access(getAnimationFileName(IMG_OEM), R_OK) == 0) {
        mZipFileName = getAnimationFileName(IMG_OEM);
    }
    else if (access(getAnimationFileName(IMG_SYS), R_OK) == 0) {
        mZipFileName = getAnimationFileName(IMG_SYS);
    }
    return NO_ERROR;
}
```

这里我们所需要关心的是最后几行代码，getAnimationFileName，通过这个函数来获取
不同路径的开机动画，然后将文件名保存到mZipFileName中

```
const char *BootAnimation::getAnimationFileName(ImageID image)
{
    const char *fileName[3] = { OEM_BOOTANIMATION_FILE,
            SYSTEM_BOOTANIMATION_FILE,
            SYSTEM_ENCRYPTED_BOOTANIMATION_FILE };

    // Load animations of Carrier through regionalization environment
    if (Environment::isSupported()) {
        Environment* environment = new Environment();
```

```
        const char* animFile = environment->getMediaFile(
                Environment::ANIMATION_TYPE, Environment::BOOT_STATUS);
        ALOGE("Get Carrier Animation type: %d,status:%d",
Environment::ANIMATION_TYPE,Environment::BOOT_STATUS);
        if (animFile != NULL && strcmp(animFile, "") != 0) {
           return animFile;
        }else{
           ALOGD("Get Carrier Animation file: %s failed", animFile);
        }
        delete environment;
    }else{
           ALOGE("Get Carrier Animation file,since it's not support carrier");
    }

    return fileName[image];
}
```

当前主要几个路径下的开机动画

```
static const char OEM_BOOTANIMATION_FILE[] = "/oem/media/bootanimation.zip";
static const char SYSTEM_BOOTANIMATION_FILE[] = "/system/media/bootanimation.zip";
static const char SYSTEM_ENCRYPTED_BOOTANIMATION_FILE[] = "/system/media/bootanimation-encrypted.zip";
```

接下来是调用BootAnimation中的threadLoop方法了

```
bool BootAnimation::threadLoop()
{
    bool r;
    // We have no bootanimation file, so we use the stock android logo
    // animation.
    if (mZipFileName.isEmpty()) {
        r = android();//如果mZipFileName为空就调用默认android开机动画
    } else {
        r = movie();//如果mZipFileName调用自定义开机动画
    }

    eglMakeCurrent(mDisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
    eglDestroyContext(mDisplay, mContext);
    eglDestroySurface(mDisplay, mSurface);
    mFlingerSurface.clear();
    mFlingerSurfaceControl.clear();
    eglTerminate(mDisplay);
    IPCThreadState::self()->stopProcess();
    return r;
}
```

重点看下movie（）函数实现

```
bool BootAnimation::movie()
{
    //加载开机动画
    Animation* animation = loadAnimation(mZipFileName);
    if (animation == NULL)
        return false;

     ......

     ......

    //播放开机动画
    playAnimation(*animation);

    if (mTimeCheckThread != NULL) {
        mTimeCheckThread->requestExit();
        mTimeCheckThread = NULL;
    }

    //释放动画资源
    releaseAnimation(animation);

    if (clockTextureInitialized) {
        glDeleteTextures(1, &mClock.name);
    }
```

```
    return false;
}
```

这里重点看下playAnimation

```
bool BootAnimation::playAnimation(const Animation& animation)
{
    const size_t pcount = animation.parts.size();
    nsecs_t frameDuration = s2ns(1) / animation.fps;
    const int animationX = (mWidth - animation.width) / 2;
    const int animationY = (mHeight - animation.height) / 2;

    // TINNO BEGIN
    // Modified by zhiqin.lin,for Poweron and poweroff ring tone switch, 20161121.
    #ifdef FEATURE_BOOT_ANIMOTION_SOUND
    ALOGE("playBackgroundMusic entry");
    playBackgroundMusic();
    #endif
    // TINNO END.

    for (size_t i=0 ; i<pcount ; i++) {
        const Animation::Part& part(animation.parts[i]);
        const size_t fcount = part.frames.size();
        glBindTexture(GL_TEXTURE_2D, 0);

        // Handle animation package
        if (part.animation != NULL) {
            playAnimation(*part.animation);
            if (exitPending())
                break;
            continue; //to next part
        }

        for (int r=0 ; !part.count || r<part.count ; r++) {
            // Exit any non playuntil complete parts immediately
            if(exitPending() && !part.playUntilComplete)
                break;

            // only play audio file the first time we animate the part
            if (r == 0 && part.audioData && playSoundsAllowed()) {
                ALOGD("playing clip for part%d, size=%d", (int) i, part.audioLength);
                audioplay::playClip(part.audioData, part.audioLength);
            }

            glClearColor(
                    part.backgroundColor[0],
                    part.backgroundColor[1],
                    part.backgroundColor[2],
                    1.0f);

            for (size_t j=0 ; j<fcount && (!exitPending() || part.playUntilComplete) ; j++) {
                const Animation::Frame& frame(part.frames[j]);
                nsecs_t lastFrame = systemTime();

                if (r > 0) {
                    glBindTexture(GL_TEXTURE_2D, frame.tid);
                } else {
                    if (part.count != 1) {
                        glGenTextures(1, &frame.tid);
                        glBindTexture(GL_TEXTURE_2D, frame.tid);
                        glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
                        glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
                    }
                    initTexture(frame);
                }

                const int xc = animationX + frame.trimX;
                const int yc = animationY + frame.trimY;
                Region clearReg(Rect(mWidth, mHeight));
```

```
                clearReg.subtractSelf(Rect(xc, yc, xc+frame.trimWidth, yc+frame.trimHeight));
                if (!clearReg.isEmpty()) {
                    Region::const_iterator head(clearReg.begin());
                    Region::const_iterator tail(clearReg.end());
                    glEnable(GL_SCISSOR_TEST);
                    while (head != tail) {
                        const Rect& r2(*head++);
                        glScissor(r2.left, mHeight - r2.bottom, r2.width(), r2.height());
                        glClear(GL_COLOR_BUFFER_BIT);
                    }
                    glDisable(GL_SCISSOR_TEST);
                }
                // specify the y center as ceiling((mHeight - frame.trimHeight) / 2)
                // which is equivalent to mHeight - (yc + frame.trimHeight)
                glDrawTexiOES(xc, mHeight - (yc + frame.trimHeight),
                              0, frame.trimWidth, frame.trimHeight);
                if (mClockEnabled && mTimeIsAccurate && part.clockPosY >= 0) {
                    drawTime(mClock, part.clockPosY);
                }

                eglSwapBuffers(mDisplay, mSurface);

                nsecs_t now = systemTime();
                nsecs_t delay = frameDuration - (now - lastFrame);
                //ALOGD("%lld, %lld", ns2ms(now - lastFrame), ns2ms(delay));
                lastFrame = now;

                if (delay > 0) {
                    struct timespec spec;
                    spec.tv_sec  = (now + delay) / 1000000000;
                    spec.tv_nsec = (now + delay) % 1000000000;
                    int err;
                    do {
                        err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &spec, NULL);
                    } while (err<0 && errno == EINTR);
                }

                checkExit();//检测是否退出动画
            }

            usleep(part.pause * ns2us(frameDuration));

            // For infinite parts, we've now played them at least once, so perhaps exit
            if(exitPending() && !part.count)
                break;
        }

    }

    // Free textures created for looping parts now that the animation is done.
    for (const Animation::Part& part : animation.parts) {
        if (part.count != 1) {
            const size_t fcount = part.frames.size();
            for (size_t j = 0; j < fcount; j++) {
                const Animation::Frame& frame(part.frames[j]);
                glDeleteTextures(1, &frame.tid);
            }
        }
    }

    // we've finally played everything we're going to play
    audioplay::setPlaying(false);
    audioplay::destroy();

    return true;
}
```

这里主要是循环显示开机动画，包括开机铃声，最后通过checkExit（）来循环检测是否可以退出开机动画

```
void BootAnimation::checkExit() {
```

```
    // Allow surface flinger to gracefully request shutdown
    char value[PROPERTY_VALUE_MAX];
    property_get(EXIT_PROP_NAME, value, "0");
    int exitnow = atoi(value);

    if(mp != NULL){
        ALOGE("linzhiqin# mp.isPlaying() = %d\n",mp->isPlaying());
        if (!mp->isPlaying()) {
            isMPlayerCompleted = true;
        }
    } else {
        isMPlayerCompleted = true;
    }

    ALOGE("linzhiqin# isMPlayerCompleted = %d\n",isMPlayerCompleted);
    if (exitnow && isMPlayerCompleted) {
        requestExit();
    }
}
```

这里主要是获取EXIT_PROP_NAME的属性值来判断是否为1退出

```
static const char EXIT_PROP_NAME[] = "service.bootanim.exit";
```

所以接下来我们要搞清楚什么地方将"service.bootanim.exit"设置成1了

当launcher应用程序主线程跑起来后，如果主线程处于空闲，就会向ActivityManagerService发送一个activityIdle的消息。

应用程序主线程是ActivityThread.java来描述的，activityIdle是这个类来实现的

```
private class Idler implements MessageQueue.IdleHandler {
...
                IActivityManager am = ActivityManagerNative.getDefault();
    ...
                        try {
                            am.activityIdle(a.token, a.createdConfig, stopProfiling);
                            a.createdConfig = null;
                        } catch (RemoteException ex) {
                            // Ignore
                        }
                    ....
    }
```

上面的ActivityManagerNavtive.getDefault()得到am ActivityManagerProxy对应的客户端的实现 那么am.activityIdle()就是ActivityManagerProxy里的函数，如下

```
public void activityIdle(IBinder token, Configuration config, boolean stopProfiling)
            throws RemoteException
    {
        ......
        mRemote.transact(ACTIVITY_IDLE_TRANSACTION, data, reply, IBinder.FLAG_ONEWAY);//发送
ACTIVITY_IDLE_TRANSACTION

        ......
    }
```

发送了ACTIVITY_IDLE_TRANSACTION的进程间通信，这个消息被ActivityManagerNative接收处理了。

```
case ACTIVITY_IDLE_TRANSACTION: {//收到消息
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            Configuration config = null;
            if (data.readInt() != 0) {
                config = Configuration.CREATOR.createFromParcel(data);
            }
            boolean stopProfiling = data.readInt() != 0;
            if (token != null) {
                activityIdle(token, config, stopProfiling);//这个函数在ActivityManagerService被重写
            }
            reply.writeNoException();
            return true;
        }
```

然后这里的activityIdle肯定是ActivityManagerService实现的

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
@Override
    public final void activityIdle(IBinder token, Configuration config, boolean stopProfiling) {
```

```
        final long origId = Binder.clearCallingIdentity();
        synchronized (this) {
            ActivityStack stack = ActivityRecord.getStackLocked(token);
            if (stack != null) {
                ActivityRecord r =
                        mStackSupervisor.activityIdleInternalLocked(token, false, config);
                if (stopProfiling) {
                    if ((mProfileProc == r.app) && (mProfileFd != null)) {
                        try {
                            mProfileFd.close();
                        } catch (IOException e) {
                        }
                        clearProfilerLocked();
                    }
                }
            }
        }
        Binder.restoreCallingIdentity(origId);
    }
```

调用activityIdleInternalLocked函数，在下面实现

frameworks/base/services/java/com/android/server/am/ActivityStackSupervisor.java

```
    final ActivityRecord activityIdleInternalLocked(final IBinder token, boolean fromTimeout,
            Configuration config) {
        ....

        if (enableScreen) {
            mService.enableScreenAfterBoot();//调ActivityManagerService类的enableScreenAfterBoot()函数
         }
....
        if (activityRemoved) {
            resumeTopActivitiesLocked();
        }

        return r;
    }
```

来到frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
void enableScreenAfterBoot() {
        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_ENABLE_SCREEN,
                SystemClock.uptimeMillis());
        mWindowManager.enableScreenAfterBoot();//调WindowManagerService类里的enableScreenAfterBoot()函数

        synchronized (this) {
            updateEventDispatchingLocked();
        }
    }
```

来到frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

```
public void enableScreenAfterBoot() {
     ....

        performEnableScreen();
    }
```

performEnableScreen()实现

```
public void performEnableScreen() {


.....
                    surfaceFlinger.transact(IBinder.FIRST_CALL_TRANSACTION, // BOOT_FINISHED
                                            data, null, 0);
   ....
    }
```

这里WMS又跟SurfaceFlinger进程进行远程调用通信，传递了一个IBinder.FIRST_CALL_TRANSACTION消息类型过去

然后我们看下SurfaceFlinger的Bn端怎么接收处理的这个消息

```
class BnSurfaceComposer: public BnInterface<ISurfaceComposer> {
public:
    enum {
        // Note: BOOT_FINISHED must remain this value, it is called from
```

```
        // Java by ActivityManagerService.
        BOOT_FINISHED = IBinder::FIRST_CALL_TRANSACTION,
        CREATE_CONNECTION,
        ......
        }
    virtual status_t onTransact(uint32_t code, const Parcel& data,
            Parcel* reply, uint32_t flags = 0);
}
```

从BnSurfaceComposer的定义可以看出FIRST_CALL_TRANSACTION对应的是BOOT_FINISHED枚举消息类型,我们接下来看看BnSurfaceComposer类中对onTransact的实现中对BOOT_FINISHED的处理

```
status_t BnSurfaceComposer::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CREATE_CONNECTION: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> b = IInterface::asBinder(createConnection());
            reply->writeStrongBinder(b);
            return NO_ERROR;
        }
        .....
        case BOOT_FINISHED: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            bootFinished();
            return NO_ERROR;
        }
        ......
}
```

BOOT_FINISHED对应的case中调用了bootFinished（）方法

[SurfaceFlinger.cpp (la.um.5.6\linux\android\frameworks\native\services\surfaceflinger)]

```
void SurfaceFlinger::bootFinished()
{
    const nsecs_t now = systemTime();
    const nsecs_t duration = now - mBootTime;
    ALOGI("Boot is finished (%ld ms)", long(ns2ms(duration)) );
    mBootFinished = true;

    // wait patiently for the window manager death
    const String16 name("window");
    sp<IBinder> window(defaultServiceManager()->getService(name));
    if (window != 0) {
        window->linkToDeath(static_cast<IBinder::DeathRecipient*>(this));
    }

    // stop boot animation
    // formerly we would just kill the process, but we now ask it to exit so it
    // can choose where to stop the animation.
    property_set("service.bootanim.exit", "1");

    const int LOGTAG_SF_STOP_BOOTANIM = 60110;
    LOG_EVENT_LONG(LOGTAG_SF_STOP_BOOTANIM,
                   ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
}
```

这里最关键的一句property_set("service.bootanim.exit", "1"); 好吧，千回万转终于找到梦中人了，原来是这里将"service.bootanim.exit"设为1了

，然后bootanimation进程的checkExit()检测到就退出进程，停止播放，大功告成。