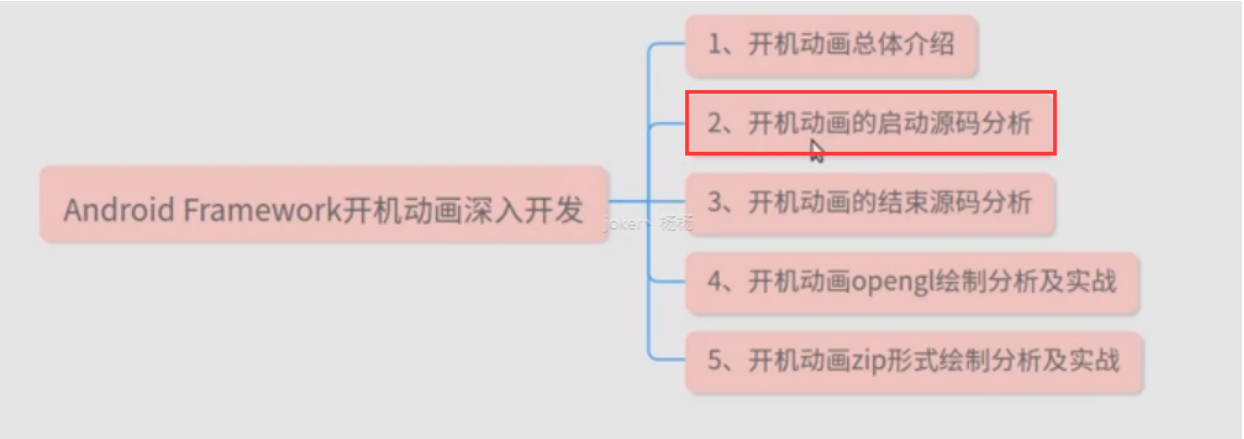
18、安卓开机动画BootAnimation启动源码分析



代码路径介绍哦：

bootanimation	frameworks/base/cmds/bootanimation/
surfaceflinger	frameworks/native/services/surfaceflinger/
init	system/core/init/

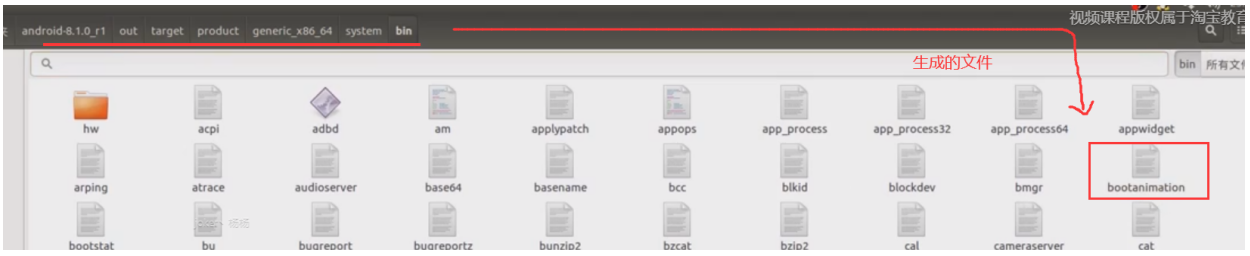
```
ting: ~/Desktop/Aosp/android-8.1.0_r1/frameworks/base/cmds/bootanimation
drwxr-xr-x  2 ting ting  4096 6月   8 00:32 iot/
ting@Ting:~/Desktop/Aosp/android-8.1.0_r1/frameworks/base/cmds/bootanimation$ cat Android.mk
bootanimation_CommonCFlags = -DGL_GLEXT_PROTOTYPES -DEGL_EGLEXT_PROTOTYPES
bootanimation_CommonCFlags += -Wall -Werror -Wunused -Wunreachable-code

# bootanimation executable
# ============================================================

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_CFLAGS += ${bootanimation_CommonCFlags}

LOCAL_SHARED_LIBRARIES := \
    libOpenSLES \
    libandroidfw \
    libbase \
    libbinder \
    libbootanimation \
    libcutils \
    liblog \
    libutils \

LOCAL_SRC_FILES:= \
    BootAnimationUtil.cpp \

ifeq ($(PRODUCT_IOT),true)
LOCAL_SRC_FILES += \
    iot/iotbootanimation_main.cpp \
    iot/BootAction.cpp

LOCAL_SHARED_LIBRARIES += \
    libandroidthings \
    libbase \
    libbinder

LOCAL_STATIC_LIBRARIES += cpufeatures

else

LOCAL_SRC_FILES += \
    bootanimation_main.cpp \
    audioplay.cpp \

endif  # PRODUCT_IOT

LOCAL_MODULE:= bootanimation

LOCAL_INIT_RC := bootanim.rc
```

查看mk文件

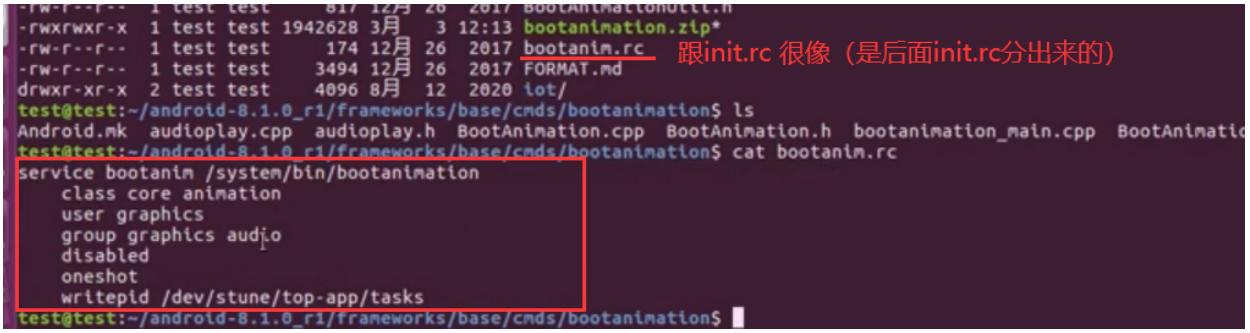
Android.mk生成的模块名字

经过在这里编译之后，就会在

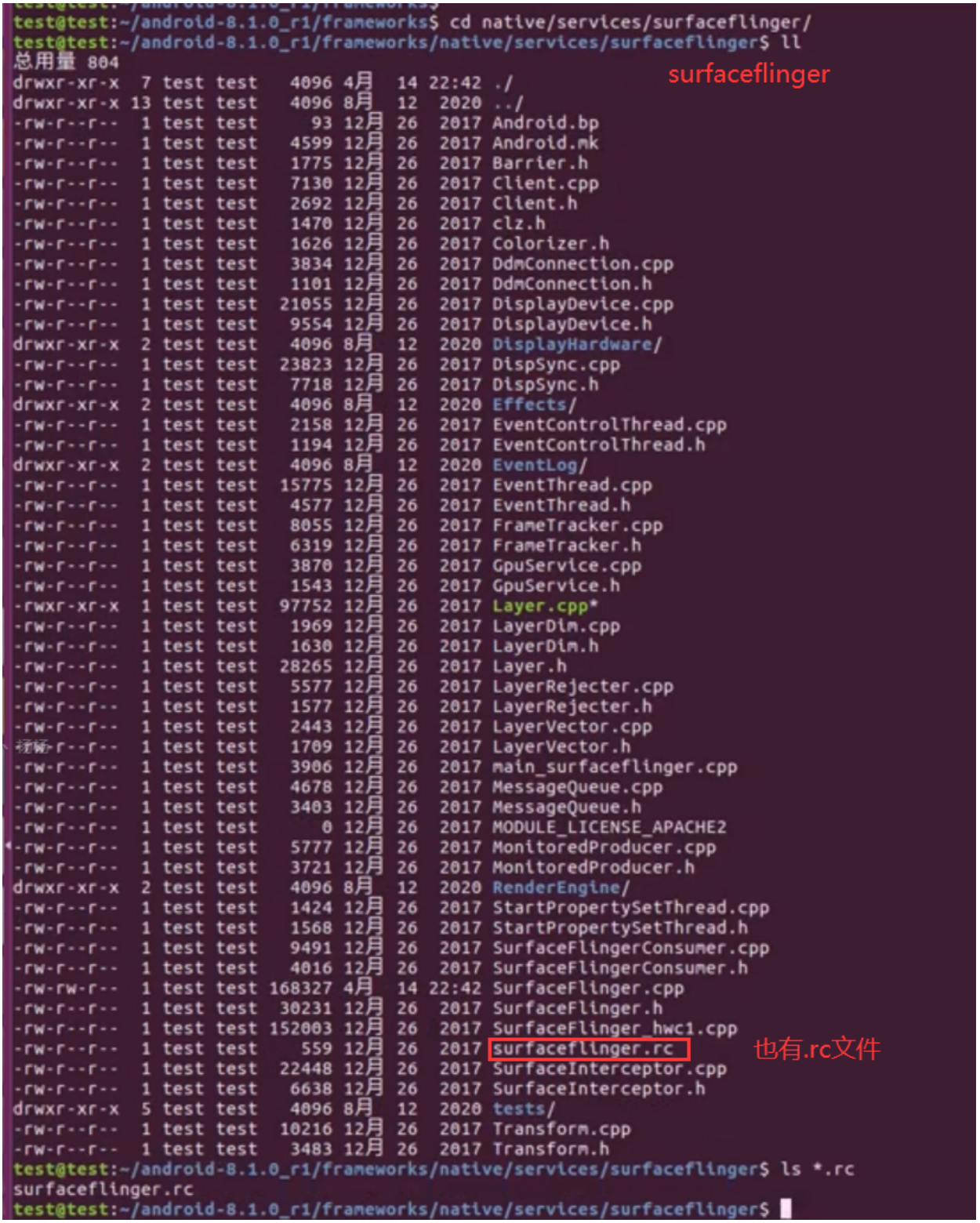
out/target/product/genic_x86_64/system/bin下生成二进制文件：



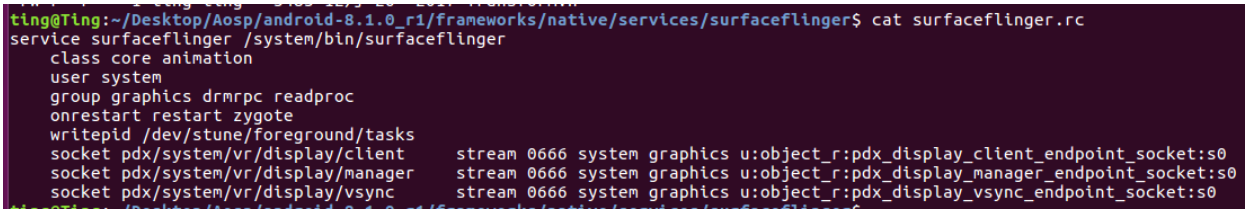
目录有个bootanim.rc:



在看下surfaceflinger:



surfaceflinger.rc



后面在分析init.rc

分析启动流程

```
启动流程详细分析：
内核起来后会启动第一个进程，即init进程。

init进程会根据init.rc配置启动surfaceflinger进程。


    service surfaceflinger /system/bin/surfaceflinger
        class main
        user system
        group graphics drmrpc
        onrestart restart zygote


surfaceflinger进程便启动了，跟着就会跑进程的main()函数。

 frameworks/native/services/surfaceflinger/main_surfaceflinger.cpp

    int main(int argc, char** argv) {
    ....

        // instantiate surfaceflinger
        sp<SurfaceFlinger> flinger = new SurfaceFlinger();//创建surfaceflinger服务实例

    ....
        flinger->init();

        // publish surface flinger
        sp<IServiceManager> sm(defaultServiceManager());
        sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false);//注册到service manager里

        // run in this thread
        flinger->run();//开跑

        return 0;
    }


首先new一个SurfaceFlinger实例，然后init，然后run
```

在init.rc和surfaceflinger文件都有这个启动的代码，那么都会执行吗？

不会的。里面有个参数是disable，就是说默认不会启动这个服务。

启动需要用到property_set，init进程会通过epoll一直监听property_set调用所触发的事件

参考：

https://blog.csdn.net/suofeng1234/article/details/52047561

```
test@test:~/android-8.1.0_r1/frameworks/base/cmds/bootanimation$ cat bootanim.rc
service bootanim /system/bin/bootanimation
    class core animation
    user graphics
    group graphics audio
    disabled
    oneshot
    writepid /dev/stune/top-app/tasks
```

就是说在init.rc解析时 是不会调用的

接着看flinger->init():

```
init()
    // Inform native graphics APIs whether the present timestamp is supported:
    if (getHwComposer().hasCapability(
            HWC2::Capability::PresentFenceIsNotReliable)) {
        mStartPropertySetThread = new StartPropertySetThread(false);
    } else {
        mStartPropertySetThread = new StartPropertySetThread(true);
    }
创建了一个线程

启动了线程
    if (mStartPropertySetThread->Start() != NO_ERROR) {
        ALOGE("Run StartPropertySetThread failed!");
    }

    ALOGV("Done initializing");
}
```

```
初始化graphics之后，mStartPropertySetThread()播放开机动画。//注意已经不是以前的startBootAnim方法

StartPropertySetThread如下定义：

StartPropertySetThread::StartPropertySetThread(bool timestampPropertyValue):
        Thread(false), mTimestampPropertyValue(timestampPropertyValue) {}

status_t StartPropertySetThread::Start() {
    return run("SurfaceFlinger::StartPropertySetThread", PRIORITY_NORMAL);
}

bool StartPropertySetThread::threadLoop() {
    // Set property service.sf.present_timestamp, consumer need check its readiness
    property_set(kTimestampProperty, mTimestampPropertyValue ? "1" : "0");
    // Clear BootAnimation exit flag
    property_set("service.bootanim.exit", "0");//关键属性
    // Start BootAnimation if not started
    property_set("ctl.start", "bootanim");//关键属性
    // Exit immediately
    return false;
}
```

在初始化过程中了，创建了属性设置线程，并且调用了start方法跑线程。

这个类是继承Thread，是个线程类

```
class StartPropertySetThread : public Thread {
// Boot animation is triggered via calls to "property_set()" which can block
// if init's executing slow operation such as 'mount_all --late' (currently
// happening 1/10th with fsck)  concurrently. Running in a separate thread
// allows to pursue the SurfaceFlinger's init process without blocking.
// see b/34499826.
// Any property_set() will block during init stage so need to be offloaded
// to this thread. see b/63844978.
public:
    StartPropertySetThread(bool timestampPropertyValue);
    status_t Start();
private:
    virtual bool threadLoop();
    static constexpr const char* kTimestampProperty = "service.sf.present_timestamp";
    const bool mTimestampPropertyValue;
};

}

#endif // ANDROID_STARTBOOTANIMTHREAD_H
```

接着看Start():

```
#include <cutils/properties.h>
#include "StartPropertySetThread.h"

namespace android {

StartPropertySetThread::StartPropertySetThread(bool timestampPropertyValue):
        Thread(false), mTimestampPropertyValue(timestampPropertyValue) {}

status_t StartPropertySetThread::Start() {
    return run("SurfaceFlinger::StartPropertySetThread", PRIORITY_NORMAL);
}

bool StartPropertySetThread::threadLoop() {
    // Set property service.sf.present_timestamp, consumer need check its readiness
    property_set(kTimestampProperty, mTimestampPropertyValue ? "1" : "0");
    // Clear BootAnimation exit flag
    property_set("service.bootanim.exit", "0");
    // Start BootAnimation if not started
    property_set("ctl.start", "bootanim");
    // Exit immediately
    return false;
}

} // namespace android
```

最终会执行到这里

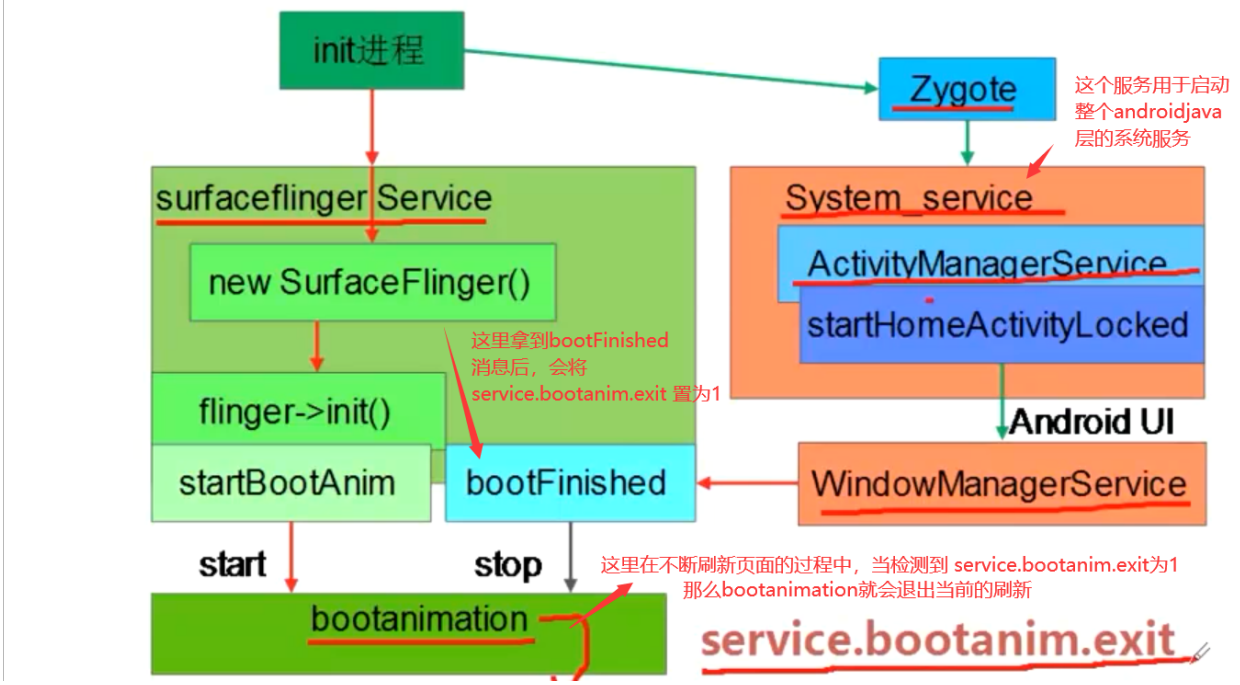
0表示开始播放动画

设置BootAnimation 退出的标志

启动BootAnimation(如果没有启动的话)

也就是当service.bootanim.exit = 1 时，就会停止刷新：

bootanim启动框架图：



property_set("ctl.start", "bootanim"); 这一句话就启动了服务，怎么启动起来的?

这样bootanim进程就会启动?凭什么设置了一个属性就启动了？那么下面我们来看，/system/core/init/init.cpp ，在看init进程的init.cpp的main函数中：

```
int main(int argc, char** argv) {
.....

    property_load_boot_defaults();
    export_oem_lock_status();
    start_property_service(); //start_property_service
    set_usb_controller();
}
```

下面来来看看start_property_service方法，在/system/core/init/property_service.cpp：
main函数中start_property_service()，在这个函数中注册一个epoll handle 的机制 register_epoll_handler()：

```
666  void start_property_service() {
667      property_set("ro.property_service.version", "2");
668
669      property_set_fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLOEXEC | SOCK_NONBLOCK,
670                                      0666, 0, 0, NULL);
671      if (property_set_fd == -1) {
672          PLOG(ERROR) << "start_property_service socket creation failed";
673          exit(1);
674      }
675
676      listen(property_set_fd, 8);
677
678      register_epoll_handler(property_set_fd, handle_property_set_fd);
679  }
```

init.cpp 启动了属性服务：

```
    property_load_boot_defaults();
    export_oem_lock_status();
    start_property_service();
    set_usb_controller();

    const BuiltinFunctionMap function_map;
    Action::set_function_map(&function_map);
```

init.cpp

查看该方法

```
void start_property_service() {
    property_set("ro.property_service.version", "2");
                                        创建了socket
    property_set_fd = CreateSocket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLOEXEC | SOCK_NONBLOCK,
                                   false, 0666, 0, 0, nullptr, sehandle);
    if (property_set_fd == -1) {
        PLOG(ERROR) << "start_property_service socket creation failed";
        exit(1);
    }

    listen(property_set_fd, 8);   进行监听

    register_epoll_handler(property_set_fd, handle_property_set_fd);
}          注册epoll handler
```

有消息来就会回调这个方法

查看回调方法：里面会接收到这个socket，然后进行一些变量初始化，并通过RecvUint32 将获取到的命令传入到了cmd变量中，然后进行case 判断，最后调用handle_property_set方法：

```
SocketConnection socket(s, cr);
uint32_t timeout_ms = kDefaultSocketTimeout;

uint32_t cmd = 0;
if (!socket.RecvUint32(&cmd, &timeout_ms)) {
    PLOG(ERROR) << "sys_prop: error while reading command from the socket";
    socket.SendUint32(PROP_ERROR_READ_CMD);
    return;
}

switch (cmd) {
case PROP_MSG_SETPROP: {
    char prop_name[PROP_NAME_MAX];
    char prop_value[PROP_VALUE_MAX];

    if (!socket.RecvChars(prop_name, PROP_NAME_MAX, &timeout_ms) ||
        !socket.RecvChars(prop_value, PROP_VALUE_MAX, &timeout_ms)) {
      PLOG(ERROR) << "sys_prop(PROP_MSG_SETPROP): error while reading name/value from the socket";
      return;
    }

    prop_name[PROP_NAME_MAX-1] = 0;
    prop_value[PROP_VALUE_MAX-1] = 0;

    handle_property_set(socket, prop_value, prop_value, true);
    break;
  }
```

handle_property_set:

```
static void handle_property_set(SocketConnection& socket,
                                const std::string& name,
                                const std::string& value,
                                bool legacy_protocol) {
  const char* cmd_name = legacy_protocol ? "PROP_MSG_SETPROP" : "PROP_MSG_SETPROP2";
  if (!is_legal_property_name(name)) {
    LOG(ERROR) << "sys_prop(" << cmd_name << "): illegal property name \"" << name << "\"";
    socket.SendUint32(PROP_ERROR_INVALID_NAME);
    return;
  }

  struct ucred cr = socket.cred();
  char* source_ctx = nullptr;
  getpeercon(socket.socket(), &source_ctx);

  if (android::base::StartsWith(name, "ctl.")) {
    if (check_control_mac_perms(value.c_str(), source_ctx, &cr)) {
      handle_control_message(name.c_str() + 4, value.c_str());
      if (!legacy_protocol) {
        socket.SendUint32(PROP_SUCCESS);
      }
    } else {
      LOG(ERROR) << "sys_prop(" << cmd_name << "): Unable to " << (name.c_str() + 4)
                 << " service ctl [" << value << "]"
                 << " uid:" << cr.uid
                 << " gid:" << cr.gid
                 << " pid:" << cr.pid;
      if (!legacy_protocol) {
        socket.SendUint32(PROP_ERROR_HANDLE_CONTROL_MESSAGE);
      }
    }
```

判断是否是合法的属性名字

if判断是否是控制命令，如果是检查是否有该权限

最后调用

这里说一下property_set方法：

```
uint32_t property_set(const std::string& name, const std::string& value) {
    if (name == "selinux.restorecon_recursive") {
        return PropertySetAsync(name, value, RestoreconRecursiveAsync);
    }

    return PropertySetImpl(name, value);
}
```

如果属性名字是selinux.restorecon_recursive是一个递归存储的意思，就异步进行设置

通常调用这个方法

PropertySetImpl：

```
static uint32_t PropertySetImpl(const std::string& name, const std::string& value) {
    size_t valuelen = value.size();

    if (!is_legal_property_name(name)) {
        LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: bad name";
        return PROP_ERROR_INVALID_NAME;
    }

    if (valuelen >= PROP_VALUE_MAX) {
        LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: "
                   << "value too long";
        return PROP_ERROR_INVALID_VALUE;
    }

    prop_info* pi = (prop_info*) __system_property_find(name.c_str());
    if (pi != nullptr) {
        // ro.* properties are actually "write-once".
        if (android::base::StartsWith(name, "ro.")) {
            LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: "
                       << "property already set";
            return PROP_ERROR_READ_ONLY_PROPERTY;
        }

        __system_property_update(pi, value.c_str(), valuelen);
    } else {
        int rc = __system_property_add(name.c_str(), name.size(), value.c_str(), valuelen);
        if (rc < 0) {
            LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: "
                       << "__system_property_add failed";
            return PROP_ERROR_SET_FAILED;
        }
```

判断是否是合法的属性名字

是否属性条目超过最大数量 （92）

通过属性名字找到 属性的节点（是个链表）

如果找到了，进入if语句

如果属性名是以"ro."开头，那么已经设置过了

接着进行属性值的更新（修改）

如果没有找到这个属性对应的节点，就执行添加(add)

接着看如果是以".ctl"开头的控制命令会执行handle_control_message:

```
File  Edit  View  Search  Terminal  Help
ting@Ting:~/Desktop/Aosp/android-8.1.0_r1/system/core/init$ grep "handle_control
_message" ./ -rn
./init.cpp:209:void handle_control_message(const std::string& msg, const std::st
ring& name) {
./property_service.cpp:427:        handle_control_message(name.c_str() + 4, value.
c_str());
./init.h:35:void handle_control_message(const std::string& msg, const std::strin
g& arg);
ting@Ting:~/Desktop/Aosp/android-8.1.0_r1/system/core/init$
```

找到该api的定位位置 init.cpp

先看传入的数据:

```
if (android::base::StartsWith(name, "ctl.")) {
  if (check_control_mac_perms(value.c_str(), source_ctx, &cr)) {
    handle_control_message(name.c_str() + 4, value.c_str());
    if (!legacy_protocol) {
      socket.SendUint32(PROP_SUCCESS);
    }
  } else {
    LOG(ERROR) << "sys.prop(" << cmd_name << "): Unable to " << (name.c_str() + 4)
```

传入的名字+4的地址的位置和value

可以看到传入进去的参数是过滤掉了"ctl." 这4个字符的地址:

```
bool StartPropertySetThread::threadLoop() {
    // Set property service.sf.present_timestamp, consumer need check its readiness
    property_set(kTimestampProperty, mTimestampPropertyValue ? "1" : "0");
    // Clear BootAnimation exit flag
    property_set("service.bootanim.exit", "0");
    // Start BootAnimation if not started
    property_set("ctl.start", "bootanim");
    // Exit immediately
    return false;
}
```

name value

也就是进行了服务启动：

```
}

void handle_control_message(const std::string& msg, const std::string& name) {
    Service* svc = ServiceManager::GetInstance().FindServiceByName(name);
    if (svc == nullptr) {
        LOG(ERROR) << "no such service '" << name << "'";
        return;
    }

    if (msg == "start") {
        svc->Start();
    } else if (msg == "stop") {
        svc->Stop();
    } else if (msg == "restart") {
        svc->Restart();
    } else {
        LOG(ERROR) << "unknown control msg '" << msg << "'";
    }
}
```

"start" "Bootanim"

首先通过SM拿到name也就是value是个服务名称，通过传入name，拿到了name所代表的服务引用Bootanim

根据msg执行对应的方法，这里就是执行svc->Start()启动服务

停止服务

重启服务

错误的msg

就会根据这个脚本开始启动服务：

```
File Edit View Search Terminal Help
ting@Ting:~/Desktop/Aosp/android-8.1.0_r1/frameworks/base/cmds/bootanimation$ ca
t bootanim.rc
service bootanim /system/bin/bootanimation
    class core animation
    user graphics
    group graphics audio
    disabled
    oneshot
    writepid /dev/stune/top-app/tasks
ting@Ting:~/Desktop/Aosp/android-8.1.0_r1/frameworks/base/cmds/bootanimation$
```

1.为什么要在surfaceflinger初始化来启动bootanimation?

因为bootanimation的绘画，是依赖surfaceflinger来操作的，surfaceflinger就是进行绘画操作的。包括手机上的一切绘制，都是需要

surfaceflinger来实现的。所以要在surfaceflinger里面来启动。

2.bootanim服务都没有启动，为什么能在SM拿到?

因为init.rc里面有bootanimation的定义，因此在init进程执行parse_config()时，会将该进程服务添加到service_list中，所以bootanimation应用是存在的。然后如果找到了服务，就会调用service_start启动服务

```
把service.bootanim.exit属性设为0，这个属性bootanimation进程里会周期检查，=1时就退出动画，这里=0表示要播放动画。
后面通过ctl.start的命令启动bootanimation进程，动画就开始播放了。

 下面来到bootanimation的实现


frameworks/base/cmds/bootanimation/bootanimation_main.cpp

    int main(int argc, char** argv)
    {


            sp<ProcessState> proc(ProcessState::self());
            ProcessState::self()->startThreadPool();

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

            IPCThreadState::self()->joinThreadPool();//binder线程池，与surfaceflinger通信用的。

        }
        return 0;
    }
```

总结：