每日一问 | Binder是如何做到跨进程权限控制的?

每日一问 | Binder是如何做到跨进程权限控制的?

鸿洋　2023-06-07 21:01

在framework的代码中，经常看到如下的权限检测的代码：

```
public boolean performDexOptMode(@NonNull Computer snapshot, String packageName,
        boolean checkProfiles, String targetCompilerFilter, boolean force,
        boolean bootComplete, String splitName) {
    if (!PackageManagerServiceUtils.isSystemOrRootOrShell()
            && !isCallerInstallerForPackage(snapshot, packageName)) {
        throw new SecurityException("performDexOptMode");
    }

    int flags = (checkProfiles ? DexoptOptions.DEXOPT_CHECK_FOR_PROFILES_UPDATES : 0)
            | (force ? DexoptOptions.DEXOPT_FORCE : 0)
            | (bootComplete ? DexoptOptions.DEXOPT_BOOT_COMPLETE : 0);
    return performDexOpt(new DexoptOptions(packageName, REASON_CMDLINE,
            targetCompilerFilter, splitName, flags));
}


/**
 * Check if the Binder caller is system UID, root's UID, or shell's UID.
 */
public static boolean isSystemOrRootOrShell() {
    final int uid = Binder.getCallingUid();
    return uid == Process.SYSTEM_UID || uid == Process.ROOT_UID || uid == Process.SHELL_UID;
}
```

Binder.getCallingUid()字面理解是获取调用方的uid，但是这个代码是目标进程调用的，如何通过一个静态方法调用，就拿到调用方的uid呢?

原问题出处：每日一问 | Binder是如何做到跨进程权限控制的? :https://wanandroid.com/wenda/show/26624

1.代码解答

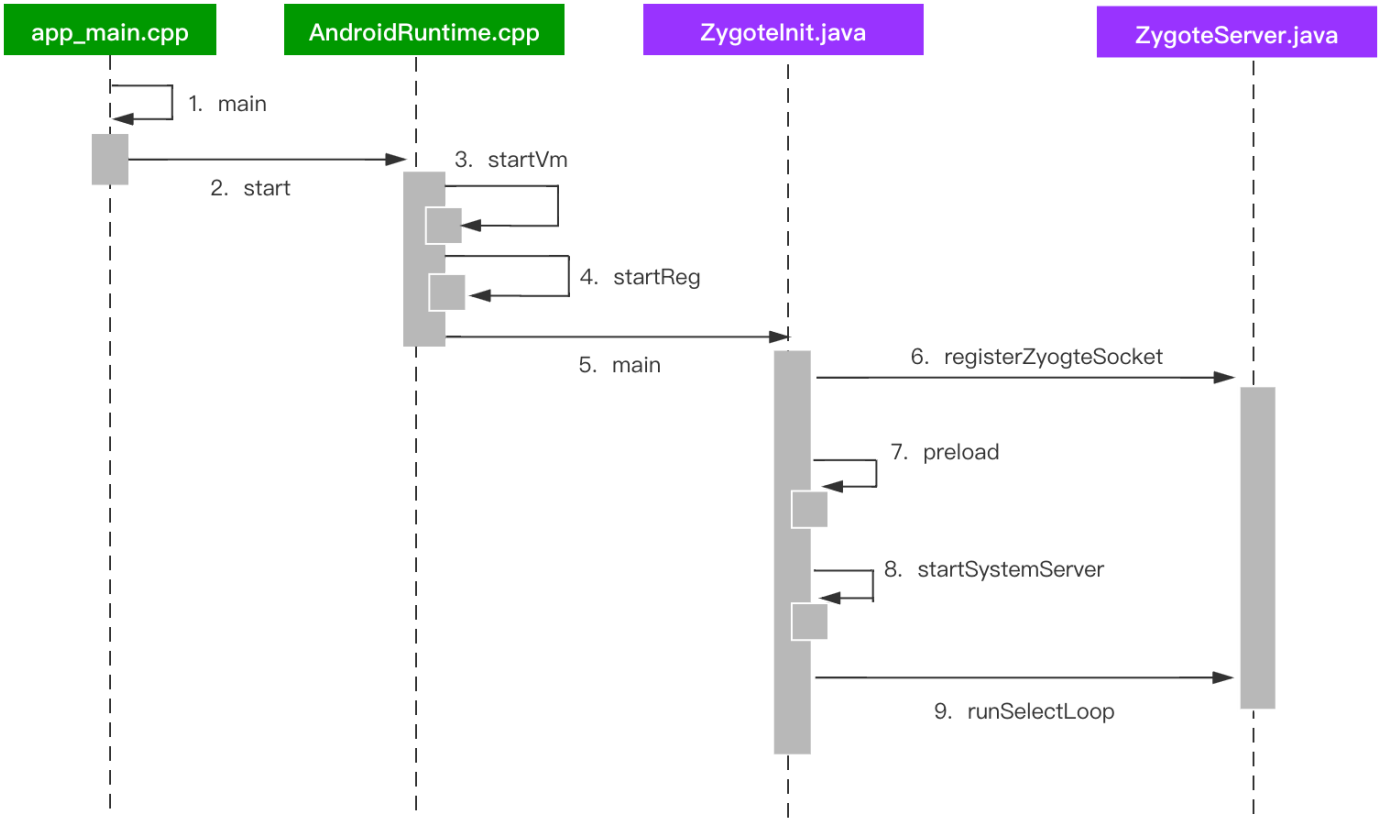
```
// frameworks/base/services/core/java/com/android/server/pm/PackageManagerServiceUtils.java
public class PackageManagerServiceUtils {
    ...
    /**
     * Check if the Binder caller is system UID, root's UID, or shell's UID.
     */
    public static boolean isSystemOrRootOrShell() {
        final int uid = Binder.getCallingUid();
        return uid == Process.SYSTEM_UID || uid == Process.ROOT_UID || uid == Process.SHELL_UID;
    }
    ...
}
```

Binder.getCallingUid()字面理解是获取调用方的uid，但是这个代码是目标进程调用的，如何通过一个静态方法调用，就拿到调用方的uid呢?

这里看看Binder.getCallingUid()的代码：

```
// frameworks/base/core/java/android/os/Binder.java
public class Binder implements IBinder {
    ...
    /**
     * Return the Linux UID assigned to the process that sent you the
     * current transaction that is being processed. This UID can be used with
     * higher-level system services to determine its identity and check
     * permissions. If the current thread is not currently executing an
     * incoming transaction, then its own UID is returned.
     */
    @CriticalNative
    public static final native int getCallingUid();
    ...
}
```

发现getCallingUid()方法是Native函数，那么它是动态注册到那个cpp类的方法呢？其实在Android系统启动过程中，Zygote进程就已经动态注册的该方法，Zygote进程是由Init进程通过init.zygote.rc文件而创建的，zygote所对应的可执行程序app_process，所对应的源文件frameworks/base/cmds/app_process/app_main.cpp。



上图的第4步startReg()方法就是Zygote注册Native的方法入口：

```
// frameworks/base/core/jni/AndroidRuntime.cpp

// 4.extern关键词修饰此函数表示：在别处定义的，要在此处引用。
extern int register_android_os_Binder(JNIEnv* env);

// RegJNIRec数组
static const RegJNIRec gRegJNI[] = {
    ...
    // 3
    REG_JNI(register_android_os_Binder),
    ...
}

//宏定义REG_JNI
//REG_JNI(register_android_os_Binder).mProc(env)等价于执行了register_android_os_Binder(env)方法
#ifdef NDEBUG
    #define REG_JNI(name)      { name }
    struct RegJNIRec {
        int (*mProc)(JNIEnv*);
    };

/*
 * Register android native functions with the VM.
 * 注册Android的Native函数
 */
/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    ...
    // 1.注册gRegJNI数组的Native方法
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
      ...
    }
    ...
}

static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
{
    // 遍历gRegJNI数组，其含有REG_JNI(register_android_os_Binder)对象
    for (size_t i = 0; i < count; i++) {
        // 2.REG_JNI(register_android_os_Binder).mProc(env)
        // 等价于执行了register_android_os_Binder(env)方法
        if (array[i].mProc(env) < 0) {
        ...
        }
    }
    return 0;
}
```

综上所述，流程最终会调用register_android_os_Binder(JNIEnv* env)方法，这个函数定义在frameworks/base/core/jni/android_util_Binder.cpp里：

```
// frameworks/base/core/jni/android_util_Binder.cpp

// JNI函数数组
static const JNINativeMethod gBinderMethods[] = {
     /* name, signature, funcPtr */
     ...
     // 3.Native方法getCallingUid()映射android_os_Binder_getCallingUid()方法
    // @CriticalNative
    { "getCallingUid", "()I", (void*)android_os_Binder_getCallingUid },
    ...
}

static jint android_os_Binder_getCallingUid()
{   // 4.调用IPCThreadState.getCallingUid()
    return IPCThreadState::self()->getCallingUid();
}

static int int_register_android_os_Binder(JNIEnv* env)
{   ...
    // 2.通过注册Native方法的辅助函数动态注册gBinderMethods数组的JNIMethod
    return RegisterMethodsOrDie(
        env, kBinderPathName,
        gBinderMethods, NELEM(gBinderMethods));
}

int register_android_os_Binder(JNIEnv* env)
{
    // 1.调用int_register_android_os_Binder()
    if (int_register_android_os_Binder(env) < 0)
    ...
}
```

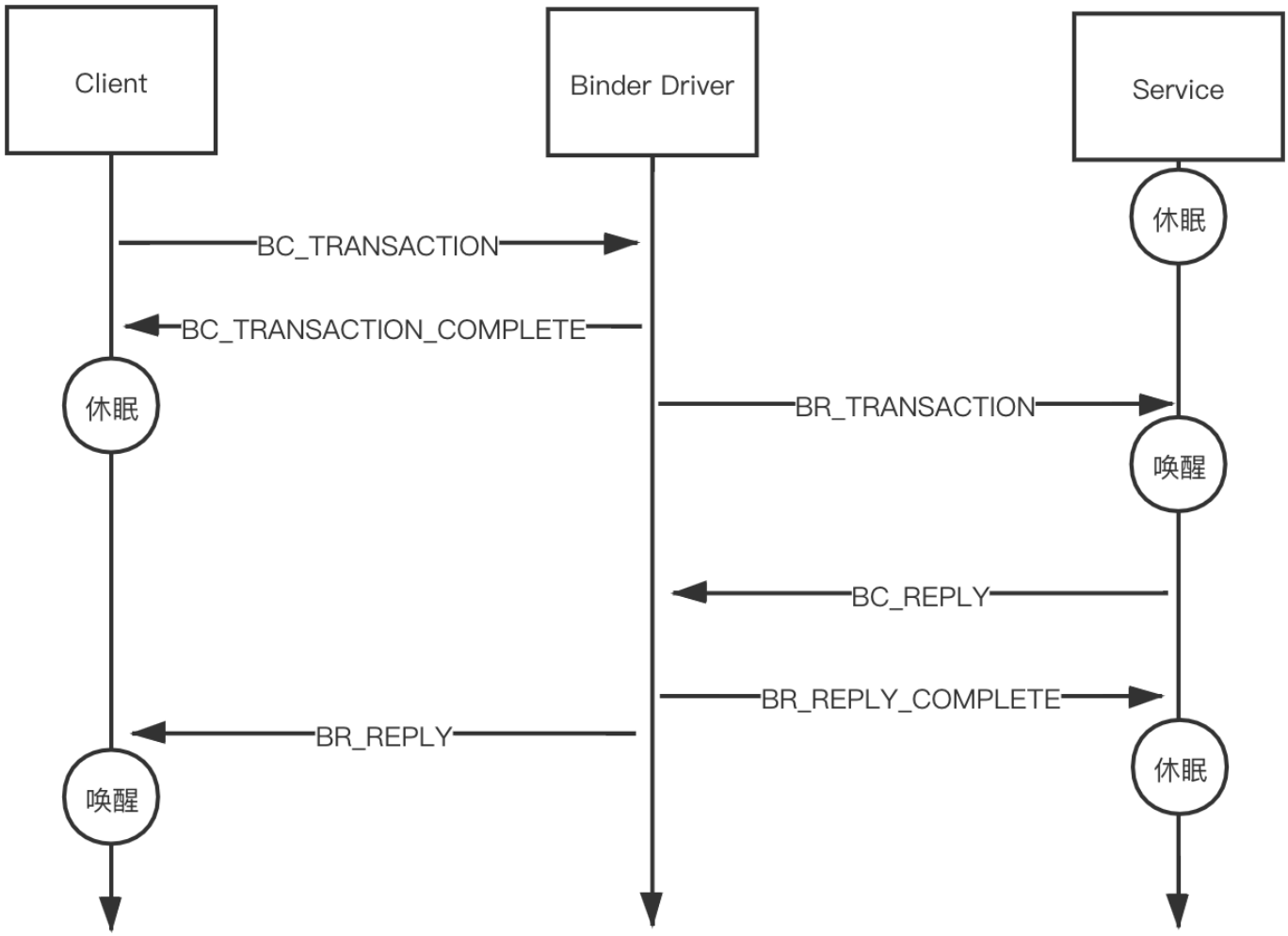
综上所述，兜兜转转Binder.getCallingUid()从Java层最终会调用IPCThreadState.getCallingUid()的Native层：

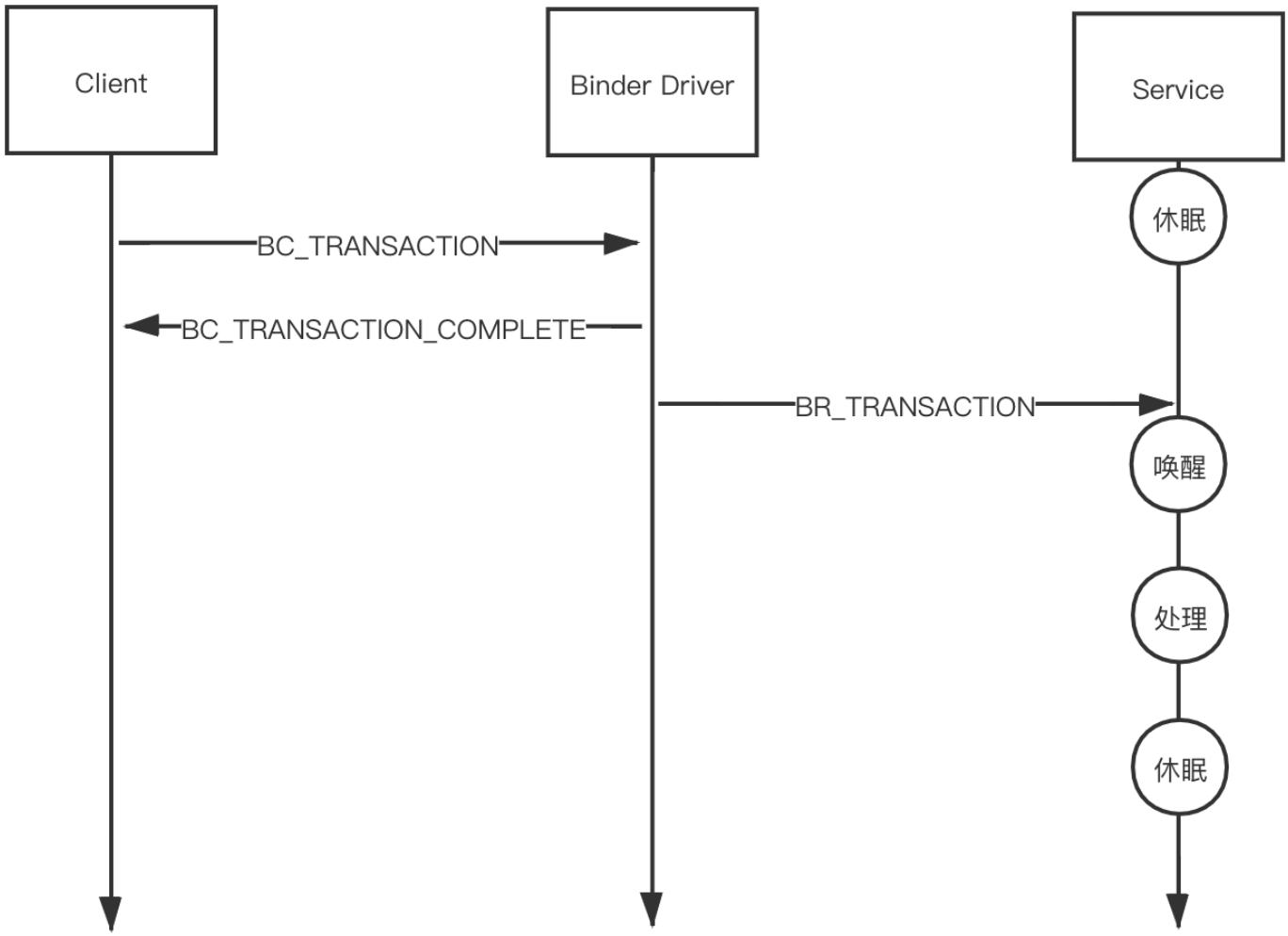
```
// frameworks/native/libs/binder/IPCThreadState.cpp
uid_t IPCThreadState::getCallingUid() const
{
    ...
    return mCallingUid;
}
```

正如代码所展示的，IPCThreadState.getCallingUid()直接返回mCallingUid字段，写入mCallingUid引用多个方法里，发现在executeCommand()方法：

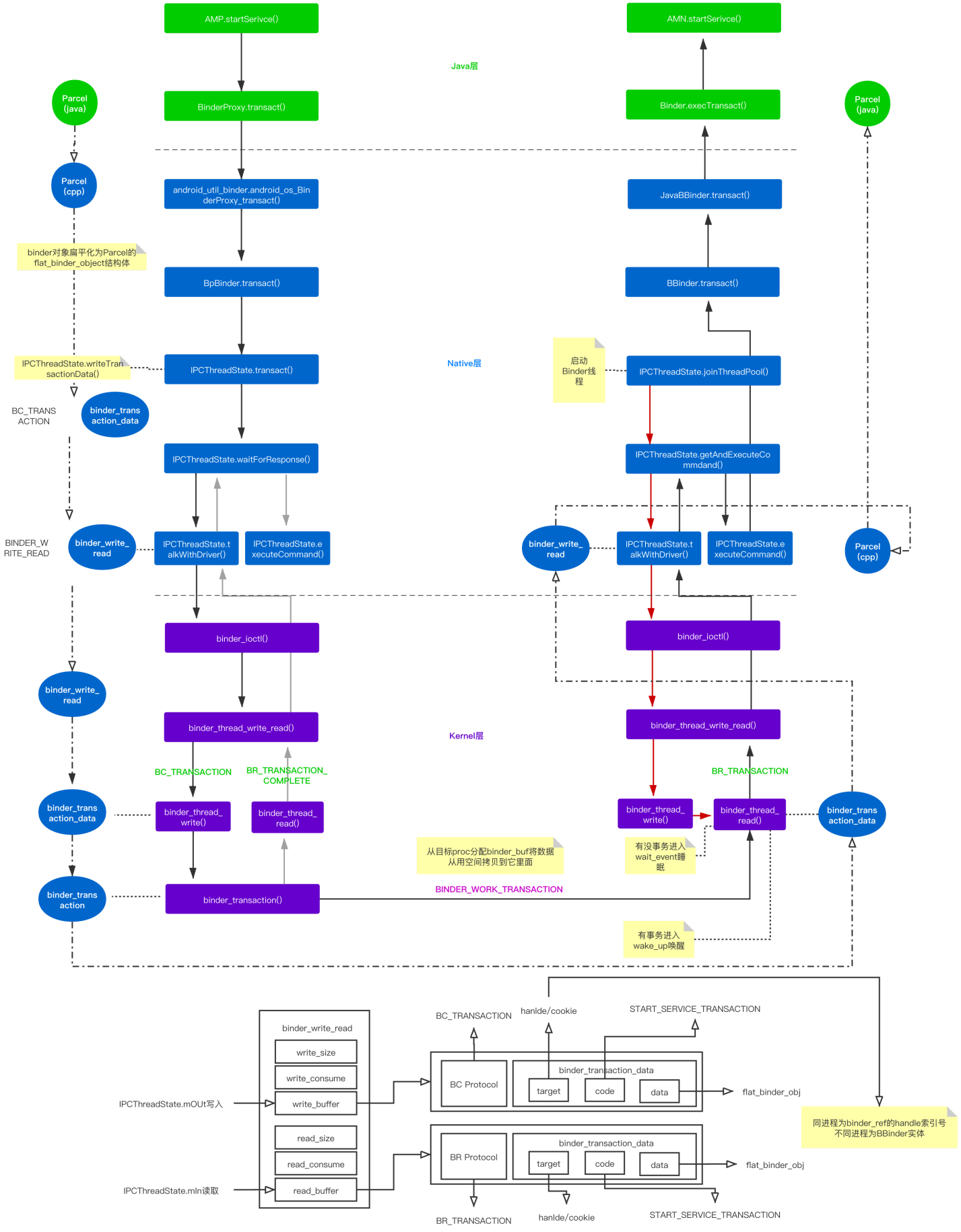
```
// frameworks/native/libs/binder/IPCThreadState.cpp
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    ...
    switch ((uint32_t)cmd) {
    ...
    case BR_TRANSACTION:
        {
            binder_transaction_data_secctx tr_secctx;
            binder_transaction_data& tr = tr_secctx.transaction_data;
            // 1.从mIn的Parcel对象读取binder_transaction_data
            if (cmd == (int) BR_TRANSACTION_SEC_CTX) {
                result = mIn.read(&tr_secctx, sizeof(tr_secctx));
            } else {
                result = mIn.read(&tr, sizeof(tr));
                tr_secctx.secctx = 0;
            }
            ...
            // 2.mCallingUid赋值为binder_transaction_data的sender_euid
            mCallingUid = tr.sender_euid;
        }
    }
    ...
}
```

由上面代码可知，在IPCThreadState.executeCommand()执行解析Binder驱动BR_TRANSACTION的ioctl系统调用命令返回的结果时，mCallingUid赋值为binder_transaction_data的sender_euid，根据名字应该是调用方的uid，下面是Binder非oneway和oneway调用的简单流程图：


Client
Binder Driver
Service
休眠
BC_TRANSACTION
BC_TRANSACTION_COMPLETE
休眠
BR_TRANSACTION
唤醒
BC_REPLY
BR_REPLY_COMPLETE
BR_REPLY
唤醒
休眠



从上面2张图可知，Binder非oneway和oneway调用都会执行Binder驱动BR_TRANSACTION的ioctl系统调用，这里看看下面完整的Binder调用的简单流程图，以startService()的Binder调用为例：



Binder复杂的程度远远不能就用一张图来说得清楚，这里只是让大家知道个大概的流程，从图中得知，在Kernel层的Binder驱动里的binder_thread_read()方

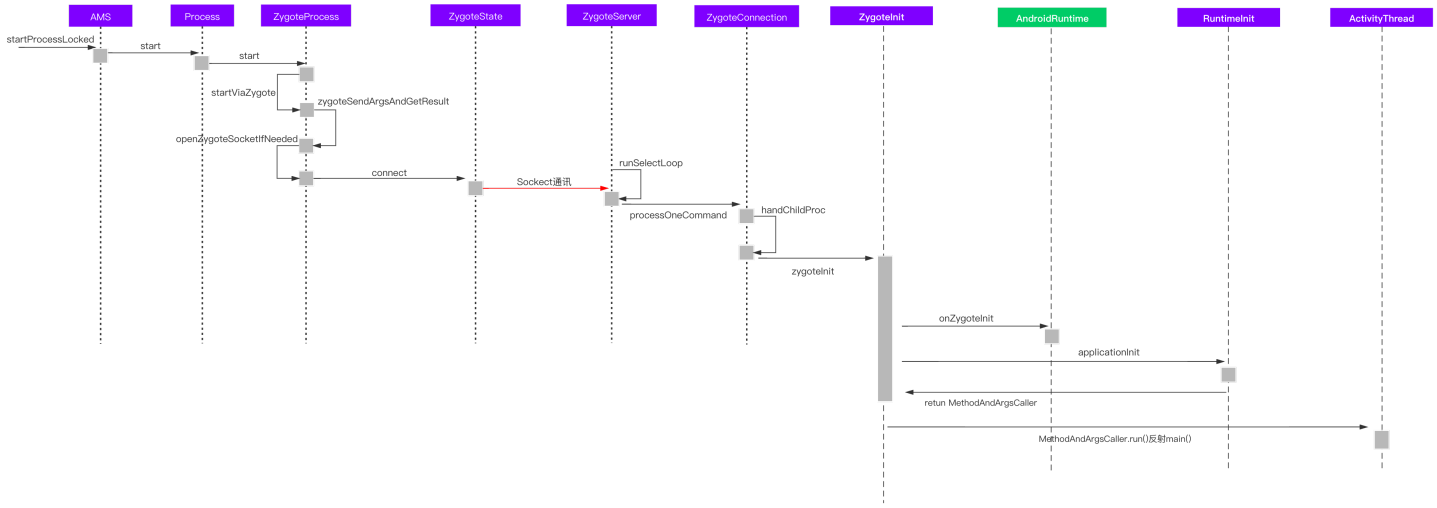
法处理BR_TRANSACTION的ioctl系统调用命令：

```
// 注意这是Kernel代码，AOSP把Kernel分离出来了
// common/drivers/android/binder.c
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              binder_uintptr_t binder_buffer, size_t size,
                              binder_size_t *consumed, int non_block)
{
        ...
        while (1) {
            uint32_t cmd;
            struct binder_transaction_data_secctx tr;
            // 当BR_TRANSACTION调用的时候，trd就是要发送给Service端的数据内容
            struct binder_transaction_data *trd = &tr.transaction_data;
            struct binder_work *w = NULL;
            // 当BR_TRANSACTION调用的时候，t就是Client端发送过来的数据内容
            struct binder_transaction *t = NULL;
            ...
            switch (w->type) {
                case BINDER_WORK_TRANSACTION: {
                    ...
                    t = container_of(w, struct binder_transaction, work);
                } break;
            }
            //只有BR_TRANSACTION,BR_REPLY才会往下执行
            if (!t)
                continue;
            ...
            trd->code = t->code;
            trd->flags = t->flags;
            // 设置binder_transaction_data的sender_euid
            // 为Client端发送过来的数据内容里uid
            trd->sender_euid = from_kuid(current_user_ns(), t->sender_euid);
    }
}
```

由上面的流程图可知，binder_thread_read()是从binder_transaction()执行进来的，在它里sender_euid字段被赋值：

```
// common/drivers/android/binder.c
static void binder_transaction(struct binder_proc *proc,
                               struct binder_thread *thread,
                               struct binder_transaction_data *tr, int reply,
                               binder_size_t extra_buffers_size)
{
    ...
    // 调用task_euid()方法传入binder_pro的task_struct的tsk获取euid
        t->sender_euid = task_euid(proc->tsk);
    ...
}
// common/drivers/android/binder_internal.h
struct binder_proc {
    ...
        struct task_struct *tsk;
    ...
}
```

如果熟悉Linux kernel进程的管理和调度机制的话，应该就非常熟悉task_struct结构体就是进程描述符了，它包含了一个进程所需的所有信息。因为在Client端发送数据到Service端的场景下，上面代码binder_transaction()传入的参数*proc就是Client端的binder_proc对象，binder驱动层的每一个binder_proc结构体都与用户空间的一个用于binder通信的进程一一对应。



每个App在启动前必须先创建一个进程，该进程是由Zygote进程fork出来，进程具有独立的资源空间，用于承载app上运行的各种Activity/Service等组件,进程在创建时候打开Binder驱动，然后才能进行Binder IPC通讯，如startActivity()调用流程AMS先检测目标进程是否启动，若没则创建目标进程。这里看一下上图的关键方法onZygoteInit()会进入Native层调用AndroidRuntime.cpp注册关联的方法：

```
// frameworks/base/core/jni/AndroidRuntime.cpp
    // 全局静态AndroidRuntime对象指针，在构造函数里赋值并保证进程单例
    static AndroidRuntime* gCurRuntime = NULL;
    ...
    static void com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env, jobject clazz)
    {
        //1.调用AndroidRuntime的纯虚函数onZygoteInit()
        gCurRuntime->onZygoteInit();
    }
    ...
```

在AndroidRuntime的头文件中的onZygoteInit()定义为纯虚函数，其子类AppRuntime实现在frameworks/base/cmds/app_process/app_main.cpp里定义：

```
// frameworks/base/cmds/app_process/app_main.cpp
    ...
    class AppRuntime : public AndroidRuntime
    {
    public:
        AppRuntime(char* argBlockStart, const size_t argBlockLength)
            : AndroidRuntime(argBlockStart, argBlockLength)
            , mClass(NULL){}
    ...
        virtual void onZygoteInit()
        {
            //1.创建sp智能指针引用的ProcessState对象
            // 在其作用域结束后将自动释放
            sp<ProcessState> proc = ProcessState::self();
            ...
        }
    }
// frameworks/native/libs/binder/ProcessState.cpp
    ...
    const char* kDefaultDriver = "/dev/binder";
    ...
    sp<ProcessState> ProcessState::self()
    {
        return init(kDefaultDriver, false /*requireDefault*/);
    }
    // sp智能指针引用的ProcessState进程单例
    static sp<ProcessState> gProcess;
    ...
    sp<ProcessState> ProcessState::init(const char *driver, bool requireDefault)
    {   ...
        // 2.创建sp智能指针引用的ProcessState对象
        gProcess = sp<ProcessState>::make(driver);
        ...
    }
    ...
    ProcessState::ProcessState(const char* driver)
      : mDriverName(String8(driver)),
       ... {
        // 3.构造方法里调用open_driver执行打开Binder驱动逻辑
        base::Result<int> opened = open_driver(driver);
        ...
    }
    ...
    static base::Result<int> open_driver(const char* driver) {
        // 4.open系统调用打开/dev/binder驱动设备
        int fd = open(driver, O_RDWR | O_CLOEXEC);
    }
```

这里看一下打开Binder驱动的代码：

```
// 注意这是Kernel代码，AOSP把Kernel分离出来了
// common/drivers/android/binder.c
static int binder_open(struct inode *nodp, struct file *filp)
{
        struct binder_proc *proc, *itr;
    ...
    // 这里的current表示的是当前的进程，
        proc->tsk = current->group_leader;
}
```

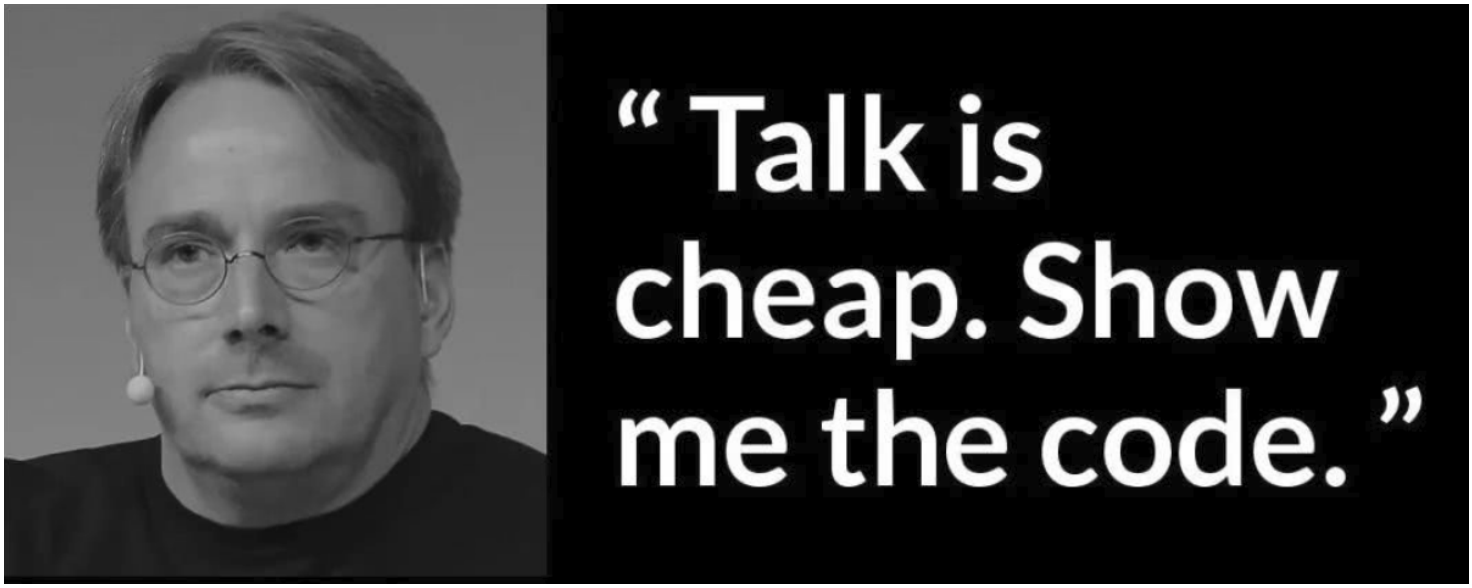
这样Client进程打开Binder驱动时调用binder_open()，就将当前进程的task_struct进程描述符绑定到了对应的binder_proc的tsk字段里。也就能获取Client进程的uid了。

2.总结

由于笔者水平有限仅给各位提供参考，希望能够抛砖引玉，也欢迎各位在评论区留言交流意见。有同学看到这些觉得难，其实，这并不奇怪，一般写应用界面根本就不会接触到，因为这需要了解Framework基本的知识才行，如Android系统启动的流程、进程的创建、四大组件与AMS的流程。我认为学习Framework层需要具备以下基础：

- 1.c/c++基础
- 2.Linux内存基础（大抵理解kernel进程的管理和调度机制，内存管理和内存寻址，I/O驱动设备(字符设备、块设备、网络设备)和调度机制等）

如果没有以上这2个必备的基础，你在看Binder源码很可能停留在表面，当然前面所说的也只不过是整个Binder的冰山一角而已。学习新东西往往有共性所在，正如你会Jetpack Compose，那么你也会70%的Flutter了。不管看多少书，更重要的是自己思考，动手重复的实践！也许这个过程很耗时间，但是，这个不断以代码去验证自己的某些猜想的过程。



> 空谈误国，实干兴邦。

3.构建优雅的Android Kernel内核和AOSP平台代码阅读环境

3.1 阅读AOSP源码的方式

工具	描述
Vim等文本工具	打开速度快，不用索引源码，不支持代码跳转。
Source Insight	界面朴素，索引速度较慢，代码跳转不完整，仅支持Window平台。
在线源码网站(如androidxref)	不用repo下载整个源码文件，但因为源码版权问题，不能阅读Android10之后的版本。
IDE	贴合实际开发环境，代码跳转、调试友好，IntelliJ/Android Studio支持Java源码环境，CLion支持Native源码环境。

在2023年的当下，笔者不再建议使用Source Insight来阅读本地源码(PS：类似Eclipse开发视感)，这里探究用AOSP源码提供的脚本工具生成使用于Android Studio和CLion项目来构建源码环境：

脚本工具	描述	缺点	
IDEGen	生成android.ipr项目配置文件和android.iml依赖文件， AndroidStudio打开android.ipr即可导入AOSP。	仅支持生成Java源码环境， 生成project未配置JDK依赖， 因为索引整个AOSP项目导入Android Studio构建非常非常非常卡顿， 需要手动排除不要的模块目录才能减轻卡顿问题。	/developme
Soong	可指定选定的模块生成CMakeLists.txt适用于CLion的项目配置文件， CLion打开即可导入AOSP。	仅支持生成Native源码环境，用.mk文件编译 (Make构建源码)的模块不会生成CMakeLists.txt文件， 需要手动编写.bp(Soong构建)文件编译模块， 编译出来的是各个模块单独的CMakeLists.txt， 需要手动创建项目总CMakeLists.txt把各个模块合并。	/build/soon
AIDEGen	自动完成指定模块的所有依赖，并自动拉起指定IDE导入项目。	如果指定的Native模块依赖很多、很大，工作耗时长	/tools/asuit

3.2 AOSP源码环境搭建准备工作

3.2.1 系统硬件配置

配置项	内容	说明
系统	Ubuntu18.04(官方指定)	自 2021 年 6 月 22 日起，不再支持在 Windows 或 MacOS 上进行构建， 而且在Android12或之前版本， AOSP使用的平台API在较新版本MacOS废弃编译不通过 (如:exteral目录下的第三库使用C的sprintf()函数)，可以通过虚拟机方案(VMware Fusion，Paralles Desktop)解决系统问题。
硬盘	至少250G起步，分区必须是Ext4文件系统， 不然编译产物含有" * : - / ?"的特殊命名文件不会被识别，Android Kernel(common-android13-5.15) 编译后大小为28G左右，AOSP(android-13.0.0_r41) 编译后大小为195G。	对于硬盘焊死在主板不支持拓展电脑，可以外接移动硬盘(Ext4文件系统)解决。
内存	16G起步	对于虚拟机上分配内存或者本机内存小于16G或者情况， 可以在Ubunt调整swap分区大小（默认2G，建议16G起步）。

更多详情可以参考官方资料：
软硬件要求:https://source.android.google.cn/docs/setup/start/requirements?hl=zh-cn

3.2.2 Ubuntu安装与配置依赖软件

笔者使用的2016年MacBook Pro 15寸高配版本，故这里以虚拟机方案解决Ubuntu系统问题。
因为Paralles Desktop是收费的，故这里使用VMware Workstation Player个人版免费版本:https://www.vmware.com/cn/products/workstation-player.html，注册VMware账号即可拿到license进行安装。

因为Ubuntu官方镜像在国内下载速度较慢，故这里使用阿里云镜像站提供的ubuntu-18.04-desktop-amd64.iso：https://mirrors.aliyun.com/oldubuntu-releases/releases/18.04.5/ubuntu-18.04-desktop-amd64.iso

接下来将ubuntu-18.04-desktop-amd64.iso用VMware Workstation Player导入即可。

如果插上了移动硬盘但虚拟机识别不了，请打开虚拟机设置->usb和蓝牙，选择对应的UBS的协议版本即可：

因为笔者的2016年MacBook Pro 15寸硬盘只有512G，但目前可用空间也就200G远远不能满足编译AOSP和Android Kernel的空间大小，所以这里选择外接移动硬盘，***注意：要把移动硬盘格式化成Ext4格式***

接下来打开终端执行apt命令安装所需要的软件包：

```
# 参考官网：https://source.android.google.cn/docs/setup/start/initializing?hl=zh-cn#installing-required-packages-ubuntu-1804
sudo apt-get install git-core gnupg flex bison build-essential zip curl zlib1g-dev libc6-dev-i386 libncurses5 lib32ncurses5-dev x11pr
```

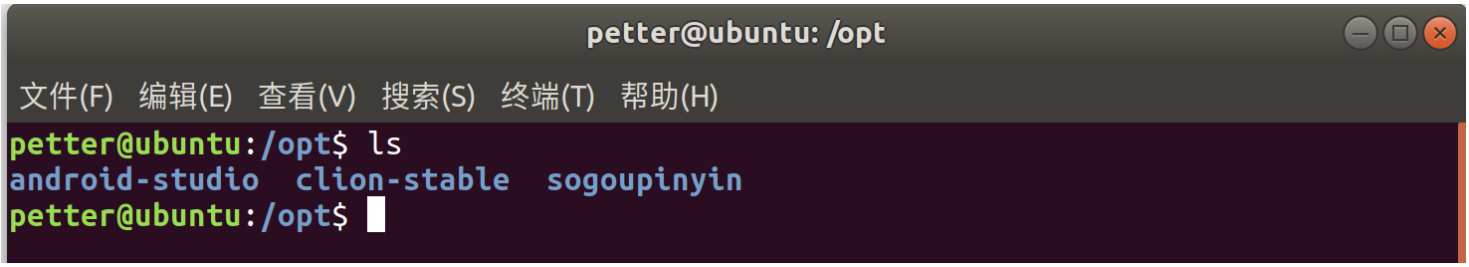
因为生成构建源码至少需要16G+的内存大小，所以在虚拟机调整swap大小：

```
# 1.设置swap大小为16G
sudo dd if=/dev/zero of=/tmp/swapfile bs=1G count=16
#  2.修改空间权限
sudo chmod 600 /tmp/swapfile
#  3.格式化空间
sudo mkswap /tmp/swapfile
#  4.启用Swap
sudo swapon /tmp/swapfile
#  5.永久启用虚拟空间
sudo vim /etc/fstab
#  在文件最末尾加入下面内容保存即可
/tmp/swapfile swap swap defaults 0 0
# 6.查看内存设置是否生效
free -m
```

最后就是下载AndroidStudio和CLion2020版本(PS：2020以后版的较新版本导入生成的Native项目Cmake会构建失败)到opt目录（PS：从snap软件商店下载，后面AIDEgen不能自动识别到它们的安装路径导致不能自动启动导入生成的项目代码），这里以安装CLion为例 ***(AndroidStudio同理，但安装完需要打开并安装SDK等相关配置)***：

1.从CLion官网下载压缩包:https://download.jetbrains.com/cpp/CLion-2020.3.4.tar.gz

2.创建opt目录，把CLion压缩包解压到clion-stable。(ps:只有命名为'clion-stable'才会被AIDEgen会自动识别拉起CLion，而AndroidStudio安装包要解压在opt/android-studio目录)

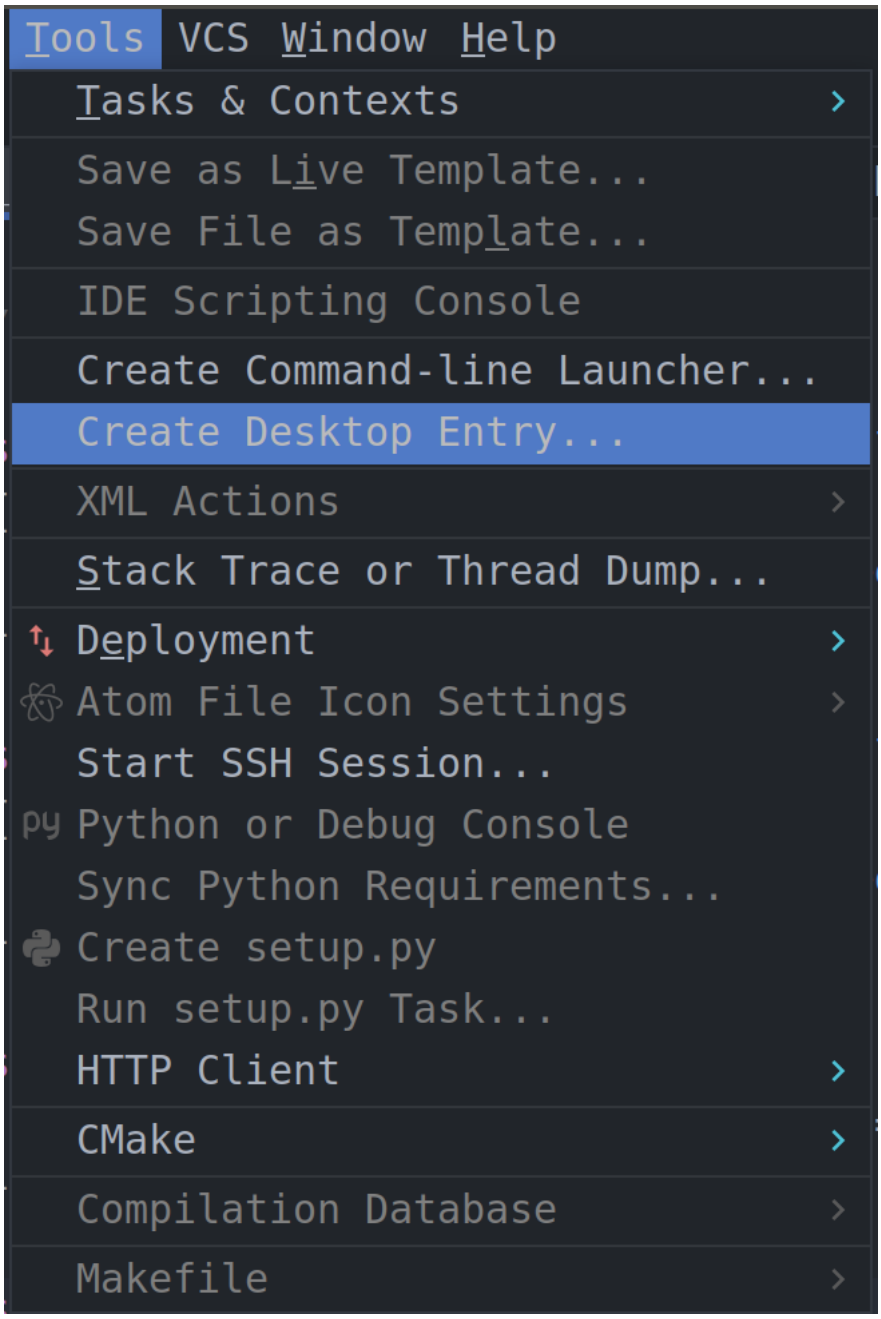



```
sudo mkdir /opt/clion-stable
sudo sudo tar -zxvf CLion-2020.3.4.tar.gz -C /opt/clion-stable
```

3.启动CLion并创建桌面快捷方式

```
#  启动CLion
sh /opt/clion-stable/bin/clion.sh
```

在完成CLion配置后，随便新建个项目进入主界面，在顶部菜单栏选择即可:Tools -> Create Desktop Entry



3.2.3 下载AOSP分支源码

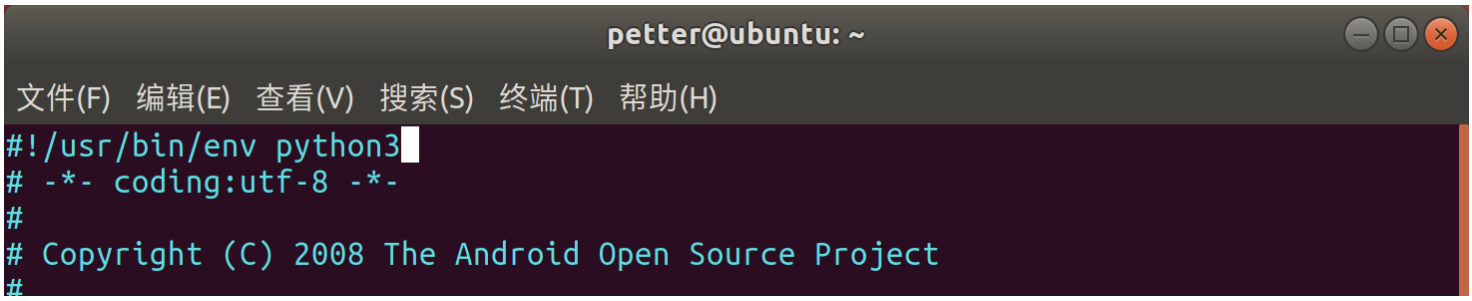
首先要在终端配置git的用户名和邮箱：

```
git config --global user.name "你的用户名"
git config --global user.email "您的邮箱地址"
```

然后就是安装Repo(PS：这是下载源码的工具)：参考官方Repo的资料:https://source.android.google.cn/docs/setup/download?hl=zh-cn#repo

```
mkdir ~/bin
PATH=~/bin:$PATH
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
vim ~/bin/repo
```

将文件中首行的'#!/usr/bin/env python'改为'#!/usr/bin/env python3'，这样Repo就以Python3执行，而不会报错或者执行不兼容的Python2。



最后再配置Python3别名和Repo镜像源地址的环境变量：

```
vim ~/.bashrc

# 将下面两行插入文件最末端
# Python3别名
alias python='python3'
# Repo镜像源地址设置为清华大学开源软件镜像站源
export REPO_URL='https://mirrors.tuna.tsinghua.edu.cn/git/git-repo/'

# 保存修改退出后，要执行source命令配置才生效
source ~/.bashrc
```

最后就是在移动硬盘创建工作目录进行拉取下载AOSP代码，这里不是拉取master分支，而是android-13.0.0_r41分支，这样拉取单个分支代码占用存储空间会小很多，更多分支参考官网：https://source.android.google.cn/docs/setup/about/build-numbers?hl=zh-cn#source-code-tags-and-builds，笔者在拉过程中网络下载速度平均30M/s，峰值60M/s，可能是清华大学开源软件镜像站限制了网速发挥不了千兆网络宽度的速度，约1小时18分左右就完成下载，但下载完成Repo自动执行checkout代码又花了15分钟左右，整个过程花了1个半小时左右，所以这个时间长短取决于您的网络带宽和电脑的CPU的核心数和硬盘、内存的读写性能。

```
# 以下命令是在移动硬盘打开终端执行操作
mkdir aosp && cd aosp && repo init -u https://aosp.tuna.tsinghua.edu.cn/platform/manifest -b android-13.0.0_r41 && repo sync
```

3.2.4 AOSP的模块概念

AOSP的模块：文件目录下有 Android.bp(Soong构建配置文件) 或者 Android.mk(GUN Make构建配置文件)即可视为模块：

```
source build/envsetup.sh
# 将AOSP所有模块的名字打印输出写入到文件查看(直接在终端输出打印会因为太多而显示不完整。)
allmod >> mods.txt

# 获取模块在源码的路径
pathmod framework
# 下面一行是输出framework模块在源码的路径
/media/petter/AOSP/AOSP_13/frameworks/base

pathmod libbinder
# 下面一行是输出binder模块在源码的路径
/media/petter/AOSP/AOSP_13/frameworks/native/libs/binder
```

envsetup.sh脚本封装了很多有用的命令，'allmod'查看AOSP所有模块，'pathmod'查看模块在源码的路径，可以执行'hmm'查看更多命令的用法。

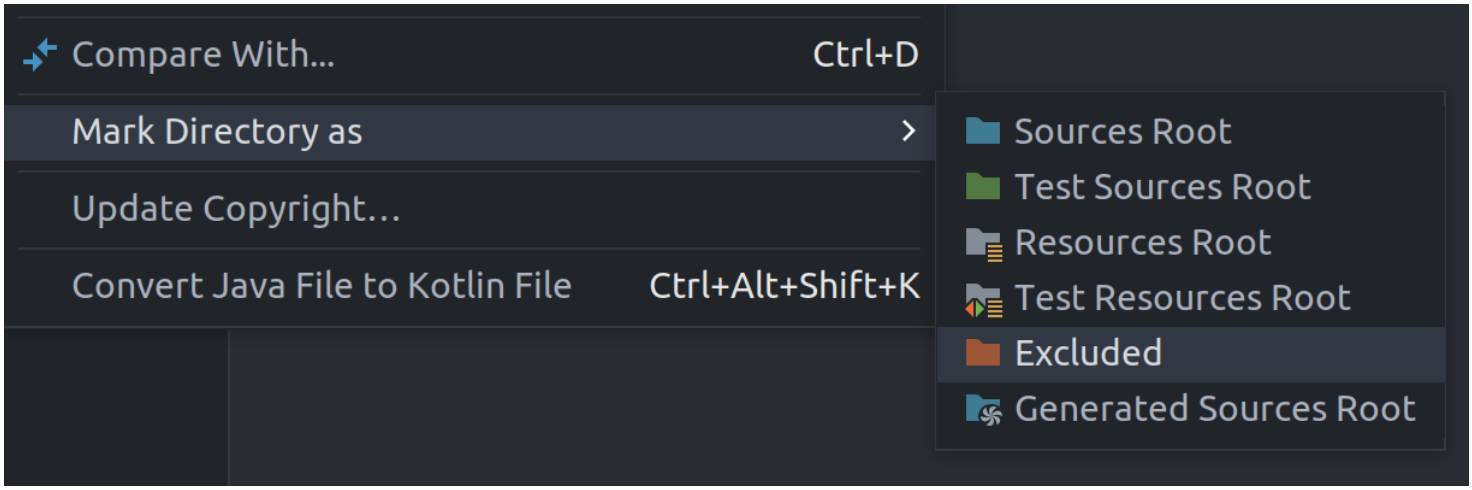
3.3 IDEGen生成AOSP源码Java环境（不推荐）

```
# 以下命令是移动硬盘下载源码的目录打开终端执行操作
apt install default-jdk -y && source build/envsetup.sh && make idegen && . development/tools/idegen/idegen.sh
```

执行上面的命令也是十分的耗时的，当成功后在AOSP代码根目录下可以找到 android.iml 和 android.ipr 两个文件。打开Android Studio，点击File --> Open，选中生成的 android.ipr 文件即可。但是整个源码的体积很大，如果全部导入会非常非常卡顿，可以排除一些代码，可以在android.iml下修改：

```
<content url="file://$MODULE_DIR$">
    <excludeFolder url="file://$MODULE_DIR$/bionic" />
    <excludeFolder url="file://$MODULE_DIR$/bootable" />
    <excludeFolder url="file://$MODULE_DIR$/build" />
    <excludeFolder url="file://$MODULE_DIR$/cts" />
    <excludeFolder url="file://$MODULE_DIR$/dalvik" />
    <excludeFolder url="file://$MODULE_DIR$/developers" />
    <excludeFolder url="file://$MODULE_DIR$/development" />
    <excludeFolder url="file://$MODULE_DIR$/device" />
    <excludeFolder url="file://$MODULE_DIR$/docs" />
    <excludeFolder url="file://$MODULE_DIR$/external" />
    <excludeFolder url="file://$MODULE_DIR$/hardware" />
    <excludeFolder url="file://$MODULE_DIR$/kernel" />
    <excludeFolder url="file://$MODULE_DIR$/out" />
    <excludeFolder url="file://$MODULE_DIR$/pdk" />
    <excludeFolder url="file://$MODULE_DIR$/platform_testing" />
    <excludeFolder url="file://$MODULE_DIR$/prebuilts" />
    <excludeFolder url="file://$MODULE_DIR$/sdk" />
    <excludeFolder url="file://$MODULE_DIR$/system" />
    <excludeFolder url="file://$MODULE_DIR$/test" />
    <excludeFolder url="file://$MODULE_DIR$/toolchain" />
    <excludeFolder url="file://$MODULE_DIR$/tools" />
    <excludeFolder url="file://$MODULE_DIR$/.repo" />
</content>
```

也可以在AndroidStudio选择目录手动排除：



最后就是点击顶部菜单栏File -> Project Structure配置项目对应的JDK、SDK了。

3.4 Soong生成AOSP源码Native环境（不推荐）

```
source build/envsetup.sh && export SOONG_GEN_CMAKEFILES=1 && export SOONG_GEN_CMAKEFILES_DEBUG=1
# 这是编译生成整个AOSP各个模块的CMakeList.txt,
# '-j8'指定的执行生成任务的线程数为8个，可根据自己电脑的cup参数指定
# make -j8

# 编译特定路径的模块
make frameworks/native/libs/binder
```

Soong编译模块期间顺带着把Native代码依赖关系记录到一个文件CMakeLists.txt里，上命令执行非常耗时，生成的CMakeLists.txt文件在对应的路径：'out/development/ide/clion/frameworks/native/libs/binder/libbinder-arm-android/CMakeLists.txt'，其中'/frameworks/native/libs/binder'是指对应模块在源码的路径，'libbinder-arm-android'是代表lib<模块名>-架构名-<平台名>意思。更多参考官方Soong文档:https://source.android.google.cn/docs/setup/build?hl=zh-cn

```
# out/development/ide/clion/frameworks/native/libs/binder/libbinder-arm-android/CMakeLists.txt
project(libbinder)
set(ANDROID_ROOT /media/petter/AOSP/AOSP_13)

set(CMAKE_C_COMPILER "${ANDROID_ROOT}/prebuilts/clang/host/linux-x86/clang-r450784d/bin/clang")
set(CMAKE_CXX_COMPILER "${ANDROID_ROOT}/prebuilts/clang/host/linux-x86/clang-r450784d/bin/clang++")
list(APPEND
     SOURCE_FILES
    ${ANDROID_ROOT}/frameworks/native/libs/binder/Binder.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/BpBinder.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/BufferedTextOutput.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/Debug.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/FdTrigger.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/IInterface.cpp
    ${ANDROID_ROOT}/frameworks/native/libs/binder/IMemory.cpp
# 省略...
```

用CLion打开刚生成的CMakeLists.txt即可，打开后目录结构是扁平的，需要修改工程根目录，在顶部菜单栏选择Tools -> CMake -> Change Project Root，以上面为例选择源码frameworks/native/libs/binder目录即可。如果您想引导多个模块，只能手动创建CMakeLists.txt把对应生成模块的CMakeLists.txt添加进来：

```
cmake_minimum_required(VERSION 3.6)
project(native)
add_subdirectory(services/surfaceflinger)
add_subdirectory(libs/ui/libui-arm64-android)
add_subdirectory(libs/gui/libgui-arm64-android)
```

3.5 AIDEGen 生成AOSP源码Java\Native环境（推荐）

AIDEGen是在2020开始释放出现在AOSP源码中，帮助开发者自动完成项目在IDE上的模块依赖、SDK、JDK等环境配置，不再需要开发者手动配置，注意的是从Android 10开始支持生成Java项目，而Native项目则从Android 11开始才完全支持，在完成任务后自动启动指定的IDE并导入项目代码。

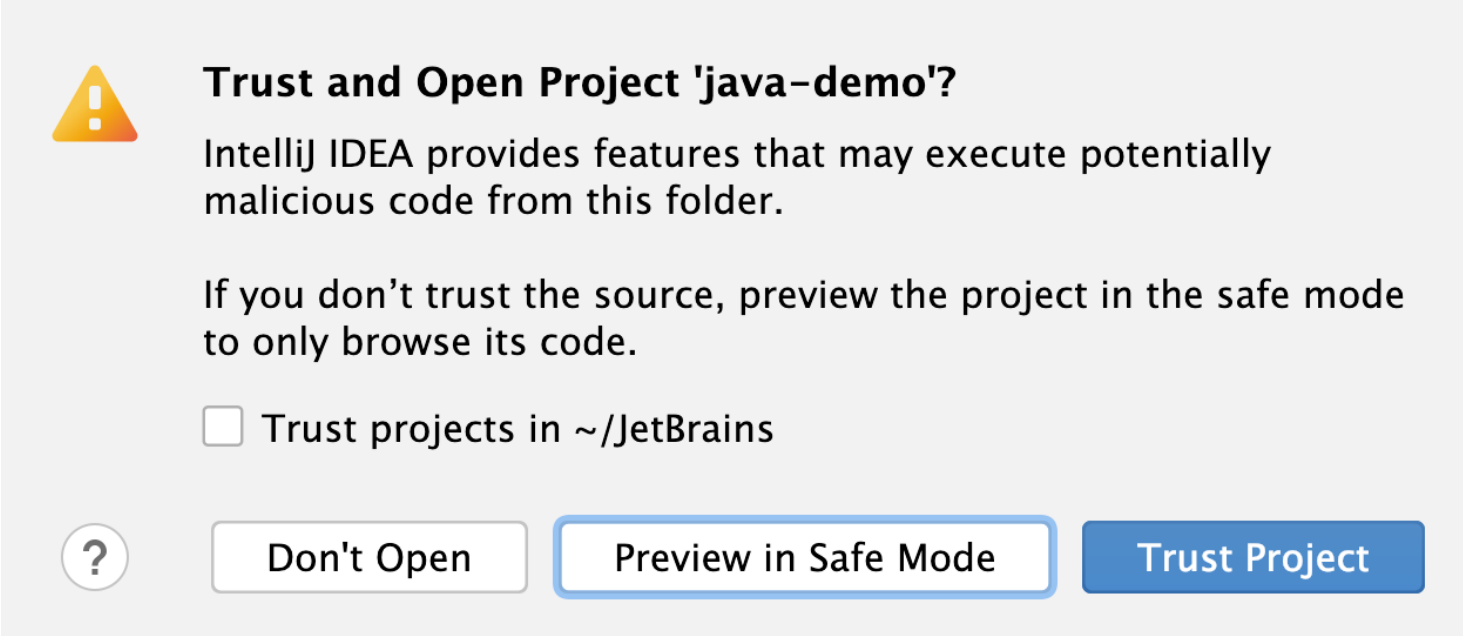
```
# ‘sdk_pc_x86_64-userdebug’表示要构建目标为64位的pc模拟器版本，
# 更多参考官方:https://source.android.google.cn/docs/setup/build/building?hl=zh-cn#choose-a-target
source build/envsetup.sh && lunch sdk_pc_x86_64-userdebug

# 生成framework模块的java源码环境，并启动AndroidStudio导入项目
aidegen framework -i s
# 也可以指定多个路径构建，下面是Binder相关的native层源码环境，并启动CLion导入项目
aidegen frameworks/base/core/jni frameworks/native/libs/binder -i c
```

如果您之前已经使用aidegen生成过模块，可以添加‘-s’跳过部分过程加速任务，更多Aidegen的参数说明如下：

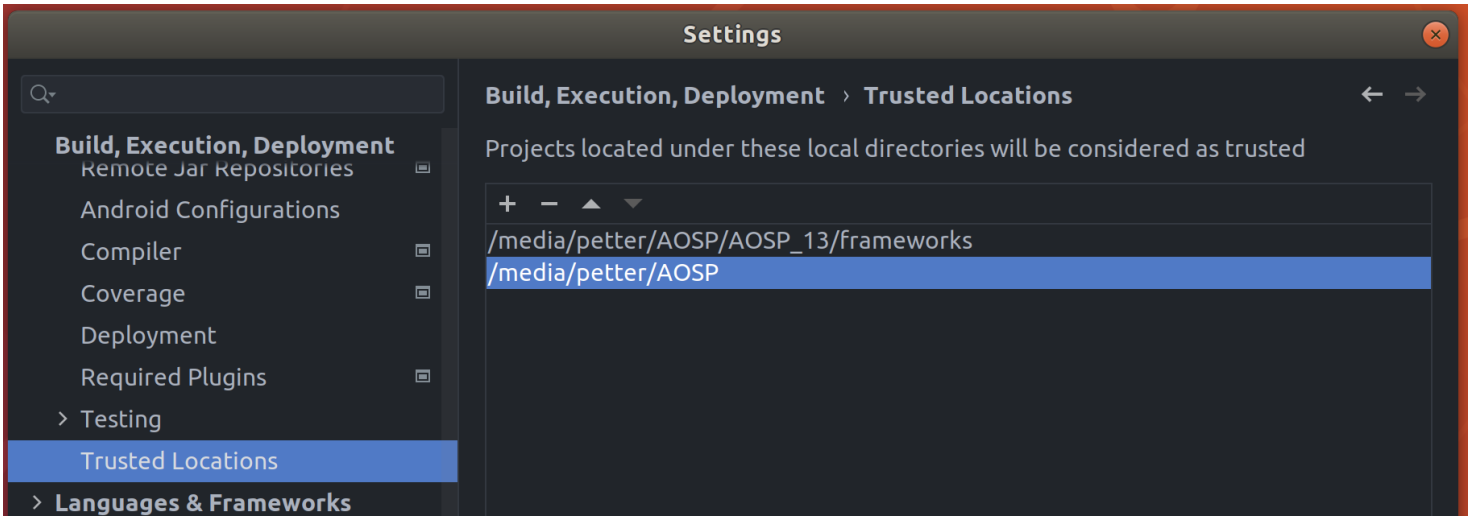
参数	完整参数	描述
-d	--depth	指定源码模块的深度值
-i	--ide	启动IDE的类型, j=IntelliJ s=Android Studio e=Eclipse c=CLion v=VS Code
-p	--ide-path	指定用户安装IDE的路径.
-n	--no_launch	完成任务后不启动IDE导入源码.
-r	--config-reset	重置AIDEGen已保存的配置
-s	--skip-build	跳过构建jar包和模块
-v	--verbose	在执行任务时显示DEBUG级别的log
-a	--android-tree	生成整个AOSP的IDE项目文件
-e	--exclude-paths	排除指定目录
-l	--language	使用特定语言启动IDE,j=java c=C/C++ r=Rust
-h	--help	帮助

3.6 最佳实践

如果您完成Ubuntu软硬件，AndroidStudio、CLion、Git、Repo、Python配置，到了用Repo拉取源码这最耗时的步骤，首先在Android Studio、CLion的设置受信任项目的路径，这样AIDEgen启动它们是就不弹出这个窗口：



打开IDE设置菜单：Build、Execution、Deployment -> Trusted Locations，设置AOSP路径即可



最后执行以下命令就会完成源码拉取、编译依赖、启动IDE的任务，如果您的电脑性能好、宽带网速可以，一觉醒来您就发现可以AndroidStudio、CLion查看Framework源码了。

```
mkdir aosp && cd aosp && repo init -u https://aosp.tuna.tsinghua.edu.cn/platform/manifest -b android-13.0.0_r41 && repo sync && sourc
```

3.7 构建Android Kernel内核源码环境

因为AOSP把Kernel内核分离开，若要查看Kernel层的模块代码(如Binder驱动)，则需要单独下载Kernel分支了，而不同的设备有不同的内核分支，这里下载的是通用内核common-android13-5.15分支。更多参考官网:https://source.android.google.cn/docs/setup/build/building-kernels?hl=zh-cn

```
mkdir kernel && cd kernel && repo init -u https://mirrors.tuna.tsinghua.edu.cn/git/AOSP/kernel/manifest -b common-android13-5.15 && r
```

上面命令执行通过repo拉取代码，编译，然后使用‘gen_compile_commands.py’脚本在根目录生成‘compile_commands.json’的依赖索引文件，直接用CLion打开即可阅读Kernel内核的代码了。

3.8 支持正版软件

VScode在AIGEGen也支持java、Native项目导入，但要安装对应的Java、Camke、C、C++，CQuery插件，相对繁琐并索引代码速度慢，而Kernel内核源码也能通过clangd插件构建索引，总而言之，CLion导入Native项目却非常省心而高效，但CLion是收费的：

1.若资金允许请购买正版：https://www.jetbrains.com.cn/clion/buy/#personal
2.学生凭学生证可免费申请正版授权：https://www.jetbrains.com/shop/eform/students
3.创业公司可5折购买正版授权：https://www.jetbrains.com/shop/eform/startup