编译系统

# Todo

资源引用问题

添加系统级res资源包：<https://www.jianshu.com/p/89c3ee73e4c4>

Letv: <http://blog.163.com/zhou_411424/blog/static/197362156201311135301754>

QA: https://my.oschina.net/kingguary/blog/177502

我是说运行时的问题

<https://blog.csdn.net/u013686019/article/details/77140477>

Android平台Overlay机制

5.2.16. LOCAL\_OVERRIDES\_PACKAGEs为何要删除才会有呢？？？

unbutu-编译源码

<https://www.jianshu.com/p/329b43272d09>

JACK编译器配置

<https://blog.csdn.net/zhudaozhuan/article/details/51842072>

user版和debug系统如何区别工作的

搭建源码学习环境，提高团队技术能力

# 前言

开源适配（产生背景）问题 : 众所周知，Android 是一个开源的操作系统。Android 的源码中包含了大量的开源项目以及许多的模块。不同产商的不同设备对于 Android 系统的定制都是不一样的。

如何将这些项目和模块的编译统一管理起来，如何能够在不同的操作系统上进行编译，如何在编译时能够支持面向不同的硬件设备，不同的编译类型，且还要提供面向各个产商的定制扩展，是非常有难度的。

（Android） **Build系统**：是主要由 **Make 文件（**最主要**）**，Shell 脚本以及 Python 脚本组成的一部分Android 源码。用来编译 Android 系统，Android SDK 以及相关文档。解决开源适配问题

学习作用：

对于 Android 平台开发人员来说，本文可以帮助你熟悉你每天接触到的构建环境。

对于其他开发人员来说，本文可以作为一个 GNU Make 的使用案例，学习这些成功案例，可以提升我们的开发经验。

关于如何获取 Android 源码，[请参照 Android Source 官方网站](http://source.android.com/source/downloading.html)

Build 系统中最主要的处理逻辑都在 Make 文件中，而其他的脚本文件只是起到一些辅助作用，本文先搭建编译环境。再重点讨论 Make 文件中的内容，然后在说shell脚本。

# ubutu-编译源码

重要网址清华大学镜像AndroidSource：<https://mirrors.tuna.tsinghua.edu.cn/help/AOSP/>

l[inux下安装7z命令及7z命令的使](https://www.cnblogs.com/yiwd/p/3649094.html)用

## 编译环境

操作系统要求在[AOSP开源](https://link.jianshu.com?t=https:/android.googlesource.com/)中,

| **Android版本** | **编译要求的Ubuntu最低版本** |
| --- | --- |
| Android 6.0至AOSP master | Ubuntu 14.04 |

JDK版本要求除了操作系统版本这个问题外,我们还需要关注JDK版本问题,为了方便,同样我们也列出的不同Android版本的源码需要用到的JDK版本:

| **Android版本** | **编译要求的JDK版本** |
| --- | --- |
| AOSP的Android主线 | OpenJDK 8 |
| Android 5.x至android 6.0 | OpenJDK 7 |

更具体的可以参看[Google源码编译要求](https://link.jianshu.com/?t=https://source.android.com/source/requirements.html)

下面是Ubuntu14.04中的依赖设置:

sudo apt-get install gnupg flex bison gperf build-essential \

zip curl zlib1g-dev gcc-multilib g++-multilib libc6-dev-i386 \

lib32ncurses5-dev x11proto-core-dev libx11-dev lib32z-dev ccache \

libgl1-mesa-dev libxml2-utils xsltproc unzip

。

## 工具下载

### 下载 repo 工具:

mkdir ~/bin

PATH=~/bin:$PATH

curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo

chmod a+x ~/bin/repo

必须要有网络哦

### 下载openJDK

### 初始化包

由于首次同步需要下载 24GB 数据，过程中任何网络故障都可能造成同步失败，我们强烈建议您使用初始化包进行初始化。

下载 https://mirrors.tuna.tsinghua.edu.cn/aosp-monthly/aosp-latest.tar，下载完成后记得根据 checksum.txt 的内容校验一下。

由于所有代码都是从隐藏的 .repo

目录中 checkout 出来的，所以我们只保留了 .repo

目录，下载后解压 再 repo sync

一遍即可得到完整的目录。

使用方法如下:

wget https://mirrors.tuna.tsinghua.edu.cn/aosp-monthly/aosp-latest.tar # 下载初始化包

tar xf aosp-latest.tarcd AOSP # 解压得到的 AOSP 工程目录# 这时 ls 的话什么也看不到，因为只有一个隐藏的 .repo 目录

repo sync # 正常同步一遍即可得到完整目录# 或 repo sync -l 仅checkout代码

## 编译错误

### Out of memory error.

AILED: /bin/bash out/target/common/obj/JAVA\_LIBRARIES/framework\_intermediates/dex-dir/classes.dex.rsp

Out of memory error (version 1.2-rc4 'Carnac' (298900 f95d7bdecfceb327f9d201a1348397ed8a843843 by android-jack-team@google.com)).

GC overhead limit exceeded.

Try increasing heap size with java option '-Xmx<size>'.

Warning: This may have produced partial or corrupted output.

ninja: build stopped: subcommand failed.

make: \*\*\* [ninja\_wrapper] 错误 1

在控制台执行以下命令:

export JACK\_SERVER\_VM\_ARGUMENTS="-Dfile.encoding=UTF-8 -XX:+TieredCompilation -Xmx4096m"out/host/linux-x86/bin/jack-admin kill-serverout/host/linux-x86/bin/jack-admin start-server

错误

log:

Copy: apicheck (out/host/linux-x86/obj/EXECUTABLES/apicheck\_intermediates/apicheck)

Install: out/host/linux-x86/bin/apicheck

Checking API: checkapi-last

(unknown): error 17: Field android.app.Notification.FLAG\_SHOW\_LIGHTS has changed value from 1 to 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have tried to change the API from what has been previously released in

an SDK. Please fix the errors listed above.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

解决办法有2：1、make clean--> make update-api --make ； 2、要么手动添加相应版本的api值，使编译通过。

。

# Make框架

整个 Build 系统中的 Make 文件可以分为三类：

## 核心目录/build/core

第一类是 Build 系统核心文件，此类文件定义了整个 Build 系统的框架，而其他所有 Make 文件都是在这个框架的基础上编写出来的。

图 1 是 Android 源码树的目录结构，Build 系统核心文件全部位于 /build/core（本文所提到的所有路径都是以 Android 源码树作为背景的，“/”指的是源码树的根目录，与文件系统无关）目录下。

图 1. Android 源码树的目录结构



### 入口文件/Makefile

整个 Build 系统的入口文件是源码树根目录下名称为“Makefile”的文件，当在源代码根目录上调用 make 命令时，make 命令首先将读取该文件。

Makefile 文件的内容只有一行：“include build/core/main.mk”。该行代码的作用很明显：包含 build/core/main.mk 文件。在 main.mk 文件中又会包含其他的文件，其他文件中又会包含更多的文件，这样就引入了整个 Build 系统。

这些 Make 文件间的包含关系是相当复杂的，图 3 描述了这种关系，该图中黄色标记的文件（且除了 $开头的文件）都位于 build/core/ 目录下。

图 4. 主要的 Make 文件及其包含关系



表 2. 主要的 Make 文件的说明

| **文件名** | **说明** |
| --- | --- |
| **main.mk** | 最主要的 Make 文件，该文件中首先将对编译环境进行检查，同时引入其他的 Make 文件。另外，该文件中还定义了几个最主要的 Make 目标，例如 droid，sdk，等（参见后文“Make 目标说明”）。 |
| **help.mk** | 包含了名称为 help 的 Make 目标的定义，该目标将列出主要的 Make 目标及其说明。 |
| **pathmap.mk** | 将许多头文件的路径通过名值对的方式定义为映射表，并提供 include-path-for 函数来获取。例如，通过 $(call include-path-for, frameworks-native)便可以获取到 framework 本地代码需要的头文件路径。 |
| **envsetup.mk** | 配置 Build 系统需要的环境变量，例如：TARGET\_PRODUCT，TARGET\_BUILD\_VARIANT，HOST\_OS，HOST\_ARCH 等。 当前编译的主机平台信息（例如操作系统，CPU 类型等信息）就是在这个文件中确定的。 另外，该文件中还指定了各种编译结果的输出路径。 |
| **combo/select.mk** | 根据当前编译器的平台选择平台相关的 Make 文件。 |
| **dumpvar.mk** | 在 Build 开始之前，显示此次 Build 的配置信息。 |
| **config.mk** | 整个 Build 系统的配置文件，最重要的 Make 文件之一。该文件中主要包含以下内容：   * 定义了许多的常量来负责不同类型模块的编译。 * 定义编译器参数以及常见文件后缀，例如 .zip,.jar.apk。 * 根据 BoardConfig.mk 文件，配置产品相关的参数。 * 设置一些常用工具的路径，例如 flex，e2fsck，dx。 |
| **definitions.mk** | 最重要的 Make 文件之一，在其中定义了大量的函数。这些函数都是 Build 系统的其他文件将用到的。例如：my-dir，all-subdir-makefiles，find-subdir-files，sign-package 等，关于这些函数的说明请参见每个函数的代码注释。 |
| **distdir.mk** | 针对 dist 目标的定义。dist 目标用来拷贝文件到指定路径。 |
| **dex\_preopt.mk** | 针对启动 jar 包的预先优化。 |
| **pdk\_config.mk** | 顾名思义，针对 pdk（Platform Developement Kit）的配置文件。 |
| ${ONE\_SHOT\_MAKEFILE} | ONE\_SHOT\_MAKEFILE 是一个变量，当使用“mm”编译某个目录下的模块时，此变量的值即为当前指定路径下的 Make 文件的路径。 |
| ${subdir\_makefiles} | 各个模块的 Android.mk 文件的集合，这个集合是通过 Python 脚本扫描得到的。 |
| **post\_clean.mk** | 在前一次 Build 的基础上检查当前 Build 的配置，并执行必要清理工作。 |
| **legacy\_prebuilts.mk** | 该文件中只定义了 GRANDFATHERED\_ALL\_PREBUILT 变量。 |
| **Makefile** | 被 main.mk 包含，该文件中的内容是辅助 main.mk 的一些额外内容。 |

### config.mk

Android 源码中包含了许多的模块，模块的类型有很多种，例如：Java 库，C/C++ 库，APK 应用，以及可执行文件等 。并且，Java 或者 C/C++ 库还可以分为静态的或者动态的，库或可执行文件既可能是针对设备（本文的“设备”指的是 Android 系统将被安装的设备，例如某个型号的手机或平板）的也可能是针对主机（本文的“主机”指的是开发 Android 系统的机器，例如装有 Ubuntu 操作系统的 PC 机或装有 MacOS 的 iMac 或 Macbook）的。不同类型的模块的编译步骤和方法是不一样，为了能够一致且方便的执行各种类型模块的编译，在 config.mk 中定义了许多的常量，这其中的每个常量描述了一种类型模块的编译方式，这些常量有：

* BUILD\_HOST\_STATIC\_LIBRARY
* BUILD\_HOST\_SHARED\_LIBRARY
* BUILD\_STATIC\_LIBRARY
* BUILD\_SHARED\_LIBRARY
* BUILD\_EXECUTABLE
* BUILD\_HOST\_EXECUTABLE
* BUILD\_PACKAGE
* BUILD\_PREBUILT
* BUILD\_MULTI\_PREBUILT
* BUILD\_HOST\_PREBUILT
* BUILD\_JAVA\_LIBRARY
* BUILD\_STATIC\_JAVA\_LIBRARY
* BUILD\_HOST\_JAVA\_LIBRARY

通过名称大概就可以猜出每个变量所对应的模块类型。（在模块的 Android.mk 文件中，只要包含进这里对应的常量便可以执行相应类型模块的编译。对于 Android.mk 文件的编写请参见后文：“添加新的模块”。）

这些常量的值都是另外一个 Make 文件的路径，详细的编译方式都是在对应的 Make 文件中定义的。这些常量和 Make 文件的是一一对应的，对应规则也很简单：常量的名称是 Make 文件的文件名除去后缀全部改为大写然后加上“BUILD\_”作为前缀。例如常量 BUILD\_HOST\_PREBUILT 的值对应的文件就是 host\_prebuilt.mk。

这些 Make 文件的说明如表 3 所示：

表 3. 各种模块的编译方式的定义文件

| **文件名** | **说明** |
| --- | --- |
| **host\_static\_library.mk** | 定义了如何编译主机上的静态库。 |
| **host\_shared\_library.mk** | 定义了如何编译主机上的共享库。 |
| **static\_library.mk** | 定义了如何编译设备上的静态库。 |
| **shared\_library.mk** | 定义了如何编译设备上的共享库。 |
| **executable.mk** | 定义了如何编译设备上的可执行文件。 |
| **host\_executable.mk** | 定义了如何编译主机上的可执行文件。 |
| **package.mk** | 定义了如何编译 APK 文件。 |
| **prebuilt.mk** | 定义了如何处理一个已经编译好的文件 ( 例如 Jar 包 )。 |
| **multi\_prebuilt.mk** | 定义了如何处理一个或多个已编译文件，该文件的实现依赖 prebuilt.mk。 |
| **host\_prebuilt.mk** | 处理一个或多个主机上使用的已编译文件，该文件的实现依赖 multi\_prebuilt.mk。 |
| **java\_library.mk** | 定义了如何编译设备上的共享 Java 库。 |
| **static\_java\_library.mk** | 定义了如何编译设备上的静态 Java 库。 |
| **host\_java\_library.mk** | 定义了如何编译主机上的共享 Java 库。 |

不同类型的模块的编译过程会有一些相同的步骤，例如：编译一个 Java 库和编译一个 APK 文件都需要定义如何编译 Java 文件。因此，表 3 中的这些 Make 文件的定义中会包含一些共同的代码逻辑。为了减少代码冗余，需要将共同的代码复用起来，复用的方式是将共同代码放到专门的文件中，然后在其他文件中包含这些文件的方式来实现的。这些包含关系如图 5 所示。由于篇幅关系，这里就不再对其他文件做详细描述（其实这些文件从文件名称中就可以大致猜出其作用）。

## 产品目录/device/sony/

### 简述

第二类是针对某个产品（一个产品可能是某个型号的手机或者平板电脑）的 Make 文件，这些文件通常位于 device 目录下，**该目录下又以公司名以及产品名分为两级目录**，图 2 是 device 目录下子目录的结构。对于一个产品的定义通常需要一组文件，这些文件共同构成了对于这个产品的定义。例如，**/device/sony/it26** 目录下的文件共同构成了对于 Sony LT26 型号手机的定义。



### 在 Build 系统中添加新产品

当我们要开发一款新的 Android 产品的时候，我们首先就需要在 Build 系统中添加对于该产品的定义。

在 Android Build 系统中对产品定义的文件通常位于 device 目录下（另外还有一个可~~以定义产品的目录是 vender~~ 目录，这是个历史遗留目录，Google 已经建议不要在该目录中进行定义，而应当选择 device 目录）。device 目录下根据公司名以及产品名分为二级目录，这一点我们在概述中已经提到过。

通常，对于一个产品的定义通常至少会包括四个文件：AndroidProducts.mk，产品版本定义文件，BoardConfig.mk 以及 verndorsetup.sh。下面我们来详细说明这几个文件。

#### AndroidProducts.mk: /device/corp/<product-name>/

AndroidProducts.mk：该文文件中的内容很简单，其中只需要定义一个变量，名称为“PRODUCT\_MAKEFILES”，该变量的值为产品版本定义文件名的列表，例如：

|  |
| --- |
| PRODUCT\_MAKEFILES := \  $(LOCAL\_DIR)/full\_stingray.mk \  $(LOCAL\_DIR)/stingray\_emu.mk \  $(LOCAL\_DIR)/kp001.mk |

#### kp001.mk(keypad)

产品版本定义文件：顾名思义，该文件中包含了对于特定产品版本的定义。该文件可能不只一个，因为同一个产品可能会有多种版本（例如，面向中国地区一个版本，面向美国地区一个版本）。该文件中可以定义的变量以及含义说明如表 6 所示：

表 6. 产品版本定义文件中的变量及其说明

| **常量** | **说明** |
| --- | --- |
| **PRODUCT\_NAME** | 最终用户将看到的完整产品名，会出现在“关于手机”信息中。 |
| **PRODUCT\_MODEL** | 产品的型号，这也是最终用户将看到的。 |
| **PRODUCT\_LOCALES** | 该产品支持的地区，以空格分格，例如：en\_GB de\_DE es\_ES fr\_CA。 |
| **PRODUCT\_PACKAGES** | 该产品版本中包含的 APK 应用程序，以空格分格，例如：Calendar Contacts。 |
| **PRODUCT\_DEVICE** | 该产品的工业设计的名称。 |
| **PRODUCT\_MANUFACTURER** | 制造商的名称。 |
| **PRODUCT\_BRAND** | 该产品专门定义的商标（如果有的话）。 |
| **PRODUCT\_PROPERTY\_OVERRIDES** | 对于商品属性的定义。 |
| **PRODUCT\_COPY\_FILES** | 编译该产品时需要拷贝的文件，以“源路径 : 目标路径”的形式。 |
| **PRODUCT\_OTA\_PUBLIC\_KEYS** | 对于该产品的 OTA 公开 key 的列表。 |
| **PRODUCT\_POLICY** | 产品使用的策略。 |
| **PRODUCT\_PACKAGE\_OVERLAYS** | 指出是否要使用默认的资源或添加产品特定定义来覆盖。 |
| **PRODUCT\_CONTRIBUTORS\_FILE** | HTML 文件，其中包含项目的贡献者。 |
| **PRODUCT\_TAGS** | 该产品的标签，以空格分格。 |

通常情况下，我们并不需要定义所有这些变量。Build 系统的已经预先定义好了一些组合，它们都位于 /build/target/product 下，每个文件定义了一个组合，我们只要继承这些预置的定义，然后再覆盖自己想要的变量定义即可。例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # 继承 full\_base.mk 文件中的定义  $(call inherit-product, $(SRC\_TARGET\_DIR)/product/full\_base.mk)  # 覆盖其中已经定义的一些变量  PRODUCT\_NAME := full\_lt26  PRODUCT\_DEVICE := lt26  PRODUCT\_BRAND := Android  PRODUCT\_MODEL := Full Android on LT26 |

#### BoardConfig.mk

* 该文件用来配置硬件主板，它其中定义的都是设备底层的硬件特性。例如：该设备的主板相关信息，Wifi 相关信息，还有 bootloader，内核，radioimage 等信息。对于该文件的示例，请参看 Android 源码树已经有的文件。

这里定义语言只显示中文和英文就不太好了。。。

#### vendorsetup.sh

* 该文件中作用是通过 add\_lunch\_combo 函数在 lunch 函数中添加一个菜单选项。该函数的参数是产品名称加上编译类型，中间以“-”连接，
* 例如：add\_lunch\_combo full\_lt26-userdebug。/build/envsetup.sh 会扫描所有 device 和 vender 二 级目 录下的名称 为"vendorsetup.sh"文件，并根据其中的内容来确定 lunch 函数的 菜单选项。

在配置了以上的文件之后，便可以编译出我们新添加的设备的系统镜像了。

add\_lunch\_combo kp001-$PROJECT\_VARIANT

首先，调用“source build/envsetup.sh”该命令的输出中会看到 Build 系统已经引入了刚刚添加的 vendorsetup.sh 文件。

然后再调用“lunch”函数，该函数输出的列表中将包含新添加的 vendorsetup.sh 中添加的条目。然后通过编号或名称选择即可。

最后，调用“make -j8”来执行编译即可。

## Make apk 基本原理

本节从mmm packages/app/Settings 执行开始，分析一下其执行原理

[Android Apk 编译原理解析](http://www.txtlxg.com/228/xiaosayidao/article/details/75096276)

# Make文件系统

## ro.build.type

*user:性能好,Log/Debug信息少,相当于是正式版*

*//eng:Log/Debug信息相对多*

*//userdebug:debug强*

编译类型

在针对特定产品进行编译时，如果能在最终发布版本的基础上进行细微修改，通常会非常有用。在模块定义中，模块可以通过 LOCAL\_MODULE\_TAGS 指定标记，这些标记可以是以下一个或多个值：optional（默认）、debug、eng。

1、如果某个模块通过 LOCAL\_MODULE\_TAGS 指定标记，标记默认为 eng。仅当编译选项为eng时，系统才会安装该模块。

2、如果某个模块通过 LOCAL\_MODULE\_TAGS 指定标记，标记默认为 debug。仅当编译选项为eng/user-debug时，系统才会安装该模块。

3、如果某个模块没有通过 LOCAL\_MODULE\_TAGS 指定标记，则其标记默认为 optional。仅当 PRODUCT\_PACKAGES 的产品配置需要可选模块时，系统才会安装可选模块。

详细介绍

Android源码编译选项eng、user、userdebug是由Android.mk文件中的LOCAL\_MODULE\_TAGS配置项来决定的。其一般形式如下：

LOCAL\_MODULE\_TAGS := user eng optional test

各项具体说明如下：

1、user：只有在user版本时该模块才被编译进去；

2、eng：只有在eng版本时该模块才被编译进去；

3、test：只有在tests版本时该模块才被编译进去；

4、optional：在所有版本中都编译该模块进去

Android编译选项中当前已定义的编译类型：

| **user** |  | **userdebug** | **eng** |
| --- | --- | --- | --- |
| release版本  这是旨在用作最终版本配置步骤的编译类型。 |  | 部分debug版本 | debug版本 |
| 仅安装标签为 user 的模块 |  | 安装标签为 user、debug 的模块 | 安装标签为 user、debug、eng 的模块 |
| 除了带有标记的模块之外，还会根据产品定义文件安装相应模块 |  | 除了带有标记的模块之外，还会根据产品定义文件安装相应模块 | 除了带有标记的模块之外，还会根据产品定义文件安装相应模块 |
| 设定属性 ro.secure=1，打开安全检查功能 |  | 设定属性 ro.secure=1，打开安全检查功能 | 设定属性 ro.secure=0，关闭安全检查功能 |
| 设定属性 ro.debuggable=0，关闭应用调试功能 |  | 设定属性 ro.debuggable=1，启用应用调试功能 | 设定属性 ro.debuggable=1，启用应用调试功能 |
|  |  |  | 设定属性 ro.kernel.android.checkjni=1，启用 JNI 调用检查 |
| 默认关闭 adb 功能 |  | 默认打开 adb 功能 | 默认打开 adb 功能 |
| 打开 Proguard 混淆器 |  | 打开 Proguard 混淆器 | 关闭 Proguard 混淆器 |
| 打开 DEXPREOPT 预先编译优化 |  | 打开 DEXPREOPT 预先编译优化 | 关闭 DEXPREOPT 预先编译优化 |

如果目标构建变体为 user，则 adb 无法获取 root 权限。

# 情景分析

## 系统如何指定默认lib库

android编译自己 内置的jar做法，android编译jar做法

http://www.bkjia.com/Androidjc/971505.html

**build/core/java.mk**

1. LOCAL\_JAVA\_LIBRARIES := bla-bla Ted-New-Lib

上述代码中的Ted-New-Lib就是新增的lib.

注意该文件中, 存在多个对LOCAL\_JAVA\_LIBRARIES赋值的位置, 除了最后一个排序的, 其他都得这样修改

**2. build/target/product/core\_minimal.mk**

**Makefile代码  [收藏代码](javascript:void())**

1. # The order of PRODUCT\_BOOT\_JARS matters.
2. PRODUCT\_BOOT\_JARS := \
3. ...
4. Ted-New-Lib
5. ...

这部分代码放在minimal下不是optional的, 可以根据需求, 放在其他几个core\_xxx.mk中.

**3. frameworks/base/Android.mk**

**Makefile代码  [收藏代码](javascript:void())**

1. ...
2. LOCAL\_JAVA\_LIBRARIES := bla-bla Ted-New-Lib
3. ...

# Make目标（基本框架）



## Make系统

### 一次编译

Android 系统的编译环境目前只支持 Ubuntu 以及 Mac OS 两种操作系统。[关于编译环境的构建方法请参见](http://source.android.com/source/initializing.html)，打开控制台之后转到 Android 源码的根目录，然后执行如清单 1 所示的三条命令即可先看下面几条指令，相信编译过Android源码的人都再熟悉不过的。

清单 1. 编译 Android 系统

|  |  |
| --- | --- |
| 1  2  3 | $ source build/envsetup.sh //初始化编译环境，包括后面的lunch和make指令  $ lunch full-eng //指定此次编译的目标设备以及编译类型  $ make –j12////开始编译，默认为编译整个系统，其中-j12代表的是编译的job数量为12。 |

编译Android源码的第一步是 source build/envsetup.sh，其中source命令就是用于运行shell脚本命令，功能等价于”.”，因此该命令也等价于. build/envsetup.sh。在文件envsetup.sh声明了**当前会话终端**可用的命令。作用是初始化编译环境，并引入一些辅助的 Shell 函数，这其中就包括第二步使用 lunch 函数

第二行命令“lunch full-eng”是调用 lunch 函数，并指定参数为“full-eng”。lunch 函数的参数用来指定此次编译的目标设备以及编译类型。在这里，这两个值分别是“full”和“eng”。“full”是 Android 源码中已经定义好的一种产品，是为模拟器而设置的。而编译类型会影响最终系统中包含的模块，关于编译类型将在表 7 中详细讲解。

如果调用 lunch 函数的时候没有指定参数，那么该函数将输出列表以供选择，该列表类似图 3 中的内容（列表的内容会根据当前 Build 系统中包含的产品配置而不同，具体参见后文“添加新的产品”），此时可以通过输入编号或者名称进行选择。

3. lunch 函数的输出



第三行命令“make -j8”才真正开始执行编译。make 的参数“-j”指定了同时编译的 Job 数量，这是个整数，该值通常是编译主机 CPU 支持的并发线程总数的 1 倍或 2 倍（例如：在一个 4 核，每个核支持两个线程的 CPU 上，可以使用 make -j8 或 make -j16）。在调用 make 命令时，如果没有指定任何目标，则将使用默认的名称为“droid”目标，该目标会编译出完整的 Android 系统镜像。

第三行命令“make -j8”才真正开始执行编译。make 的参数“-j”指定了同时编译的 Job 数量，这是个整数，该值通常是编译主机 CPU 支持的并发线程总数的 1 倍或 2 倍（例如：在一个 4 核，每个核支持两个线程的 CPU 上，可以使用 make -j8 或 make -j16）。在调用 make 命令时，如果没有指定任何目标，则将使用默认的名称为“droid”目标，该目标会编译出完整的 Android 系统镜像。

### make droid

如果在源码树的根目录直接调用“make”命令而不指定任何目标，则会选择默认目标：“droid”（在 main.mk 中定义）。因此，这和执行“make droid”效果是一样的。

droid 目标将编译出整个系统的镜像。从源代码到编译出系统镜像，整个编译过程非常复杂。这个过程并不是在 droid 一个目标中定义的，而是 droid 目标会依赖许多其他的目标，这些目标的互相配合导致了整个系统的编译。

图 6 描述了 droid 目标所依赖的其他目标：

图 6. droid 目标所依赖的其他 Make 目标



表 4. droid 所依赖的其他 Make 目标的说明

| **名称** | **说明** |
| --- | --- |
| **apps\_only** | 该目标将编译出当前配置下不包含 user，userdebug，eng 标签（关于标签，请参见后文“添加新的模块”）的应用程序。 |
| **droidcore** | 该目标仅仅是所依赖的几个目标的组合，其本身不做更多的处理。 |
| **dist\_files** | 该目标用来拷贝文件到 /out/dist 目录。 |
| **files** | 该目标仅仅是所依赖的几个目标的组合，其本身不做更多的处理。 |
| **prebuilt** | 该目标依赖于 $(ALL\_PREBUILT)，$(ALL\_PREBUILT)的作用就是处理所有已编译好的文件。 |
| $(modules\_to\_install) | modules\_to\_install 变量包含了当前配置下所有会被安装的模块（一个模块是否会被安装依赖于该产品的配置文件，模块的标签等信息），因此该目标将导致所有会被安装的模块的编译。 |
| $(modules\_to\_check) | 该目标用来确保我们定义的构建模块是没有冗余的。 |
| $(INSTALLED\_ANDROID\_INFO\_TXT\_TARGET) | 该目标会生成一个关于当前 Build 配置的设备信息的文件，该文件的生成路径是：out/target/product/<product\_name>/android-info.txt |
| **systemimage** | 生成 system.img。 |
| $(INSTALLED\_BOOTIMAGE\_TARGET) | 生成 boot.img。 |
| $(INSTALLED\_RECOVERYIMAGE\_TARGET) | 生成 recovery.img。 |
| $(INSTALLED\_USERDATAIMAGE\_TARGET) | 生成 userdata.img。 |
| $(INSTALLED\_CACHEIMAGE\_TARGET) | 生成 cache.img。 |
| $(INSTALLED\_FILES\_FILE) | 该目标会生成 out/target/product/<product\_name>/ installed-files.txt 文件，该文件中内容是当前系统镜像中已经安装的文件列表。 |

其他目标

Build 系统中包含的其他一些 Make 目标说明如表 5 所示：

### Build输出/out

所有的编译产物都将位于 /out 目录下，该目录下主要有以下几个子目录：

/out/host/：该目录下包含了针对主机的 Android 开发工具的产物。即 SDK 中的各种工具，例如：emulator，adb，aapt 等。

/out/target/common/：该目录下包含了针对设备的共通的编译产物，主要是 Java 应用代码和 Java 库。

/out/target/product/<product\_name>/：包含了针对特定设备的编译结果以及平台相关的 C/C++ 库和二进制文件。其中，<product\_name>是具体目标设备的名称。

/out/dist/：包含了为多种分发而准备的包，通过“make disttarget”将文件拷贝到该目录，默认的编译目标不会产生该目录。

#### 镜像文件

Build 的产物中最重要的是三个镜像文件，它们都位于 /out/target/product/<product\_name>/ 目录下。

这三个文件是：

system.img：包含了 Android OS 的系统文件，库，可执行文件以及预置的应用程序，将被挂载为根分区。

ramdisk.img：在启动时将被 Linux 内核挂载为只读分区，它包含了 /init 文件和一些配置文件。它用来挂载其他系统镜像并启动 init 进程。

userdata.img：将被挂载为 /data，包含了应用程序相关的数据以及和用户相关的数据。

## make **sdk**

make PRODUCT-sdk-sdk 命令，编译一个新的sdk出来

**自己将源代码中的@hide去掉，然后编译一个sdk来替换默认的sdk**

**和make sdk的区别**

lunch完直接make

SDK的编译和模块编译非常相似,其命令如下:   
首先是初始化编译环境:source build/envsetup.sh

* 1

接下来是设定编译目标lunch sdk-eng

* 1

最后通过以下命令编译SDK make sdk

source build/envsetup.sh

lunch sdk-eng

make

android.jar 切记这个特例，这货是make sdk生成的，多方整合，别以为也可以找到对应目录，木有的！还有就是这个jar很实用的，很多时候我们想用AS直接调运系统的hide API等，自己编译一个就能派上用场啦！

如何导出系统的android.jar呢，编译响应模块？

3 在ubuntu命令行终端使用如下命令进行编译：

make clean

make update-api

make PRODUCT-sdk-sdk

最后一行也可以用如下两个命令代替：

make –j4

make sdk

4编译结果

4.1android.jar文件所在的目录为

/out/target/common/obj/PACKAGING/android\_jar\_intermediates/android.jar并在/out/target/common/obj/JAVA\_LIBRARIES/android\_stubs\_current\_intermediates/src目录下重新以package形式组织所有生成到android.jar中的源代码。

   4.2 SDK所在目录为

out/host/linux-x86/sdk/，生成的sdk文件目录和压缩包为：

android-sdk\_eng.xxx\_linux-x86

android-sdk\_eng.xxx\_linux-x86.zip(xxx为自己android源码所在的文件名)

## 其他主要 Make 目标

表 5. 其他主要 Make 目标

| **Make 目标** | **说明** |
| --- | --- |
| **make clean** | 执行清理，等同于：rm -rf out/。 |
| **make sdk** | 编译出 Android 的 SDK。 |
| **make clean-sdk** | 清理 SDK 的编译产物。 |
| **make update-api** | 更新 API。在 framework API 改动之后，需要首先执行该命令来更新 API，公开的 API 记录在 frameworks/base/api 目录下。  <http://www.cnblogs.com/douzhanshen/p/3243442.html>  在更目录下执行  比如添加了一个类，增加了对外的方法 |
| **make dist** | 执行 Build，并将 MAKECMDGOALS 变量定义的输出文件拷贝到 /out/dist 目录。 |
| **make all** | 编译所有内容，不管当前产品的定义中是否会包含。 |
| **make help** | 帮助信息，显示主要的 make 目标。 |
| **make snod** | 从已经编译出的包快速重建系统镜像。 |
| **make libandroid\_runtime** | 编译所有 JNI framework 内容。 |
| **Make framework** | 编译所有 Java framework 内容。 |
| **Make services** | 编译系统服务和相关内容。 |
| **make <local\_target>** | 编译一个指定的模块，local\_target 为模块的名称。 |
| **make clean-<local\_target>** | 清理一个指定模块的编译结果。 |
| **makedump-products** | 显示所有产品的编译配置信息，例如：产品名，产品支持的地区语言，产品中会包含的模块等信息。 |
| **makePRODUCT-xxx-yyy** | 编译某个指定的产品。 |
| **makebootimage** | 生成 boot.img |
| **makerecoveryimage** | 生成 recovery.img |
| **makeuserdataimage** | 生成 userdata.img |
| **makecacheimage** | 生成 cache.img |

## Make子模块

| **编译指令** | **解释** |
| --- | --- |
| m | 在源码树的根目录执行编译 |
| mm | 编译当前路径下所有模块，但不包含依赖 |
| mmm [module\_path] | 编译指定路径下所有模块，但不包含依赖 |
| mma | 编译当前路径下所有模块，且包含依赖 |
| mmma [module\_path] | 编译指定路径下所有模块，且包含依赖 |
| make [module\_name] | 无参数，则表示编译整个Android代码 |

下面列举部分模块的编译指令：

| **模块** | **make命令** | **mmm命令** |
| --- | --- | --- |
| init | make init | mmm system/core/init |
| zygote | make app\_process | mmm frameworks/base/cmds/app\_process |
| system\_server | make services | mmm frameworks/base/services  编译frameworks/base的mk不是递归编译关系，其子模块需要单独mmm，比如services模块，修改了pm之类，必须mmm frameworks/base/services  支持release/debug |
| java framework.jar | make framework | mmm frameworks/base  如果编译失败：需要再编译一次framework-res  [务必保证系统为debug版本，否则不生效的！！](https://www.zhihu.com/question/40425628)  编译framwork，有些依赖不好处理  <https://www.zhihu.com/question/40425628>  1.在平台编译的时候，依赖framework，没有编译 |
| framework资源 | make framework-res | mmm frameworks/base/core/res |
| jni framework | make libandroid\_runtime | mmm frameworks/base/core/jni |
| binder | make libbinder | mmm frameworks/native/libs/binder |

上述mmm命令同样适用于mm/mma/mmma，编译系统采用的是增量编译，只会编译发生变化的目标文件。当需要重新编译所有的相关模块，则需要编译命令后增加参数-B，比如make -B [module\_name]，或者 mm -B [module\_path]。

有些依赖库，直接从手机里面拉取出来，是可以的

## Make apk

### main manifest 需要制定sdk版本号

Main manifest has <uses-sdk android:targetSdkVersion='1'> but library uses targetSdkVersion='25'

Note: main manifest lacks a <uses-sdk android:targetSdkVersion> declaration, which defaults to value minSdkVersion or 1.

<uses-sdk

android:minSdkVersion="17"

android:targetSdkVersion="25" />

----------------------------------------------------------------

### 不得使用硬编码，否则过不了make本地化检查

out/target/common/obj/JAVA\_LIBRARIES/dpad-widget\_intermediates/aar/res/layout/common\_dlg1.xml:17: error: Error: This attribute must be localized. (at 'text' with value 'Title').

[android强制实行多语言](https://blog.csdn.net/qinyun188/article/details/7466973)

### AAR导入方法

**Tips:**

* 对于m、mm、mmm、mma、mmma这些命令的实现都是通过make方式来完成的。
* mmm/mm编译的效率很高，而make/mma/mmma编译较缓慢；
* make/mma/mmma编译时会把所有的依赖模块一同编译，但mmm/mm不会;
* 建议：首次编译时采用make/mma/mmma编译；当依赖模块已经编译过的情况，则使用mmm/mm编译。
* make clean：执行清理操作，等价于 rm -rf out/
* make update-api：更新API，在framework API改动后需执行该指令，Api记录在目录frameworks/base/api；
* mmm + 记事本编译常见的坑，错误的import！

## 导出系统class.jar

系统jar包主要目录：/system/framework，以framework.jar为例。

注：309k的jar包，直接从系统pull出来，发现都是309k的一个jar，只有摘要并没有class，不能满足系统应用的jar依赖。

309k的jar：309k的jar是一个经过编译优化的包，WITH\_ DEX\_PREOPT=false,mm 模块就好

### 不需要hide api

直接用系统未经过编译优化的原始jar包就好了。从网上对应版本下载都可以的

### 需要hide api:

需要调用hide 的api的话，mmm编译的文件会生成在 优化后的文件在

目录out/target/common/obj/JAVA\_LIBRARIES/

比如framework.jar对应的是framework\_intermediates/classes.jar

core-libart.jar对应的是core-libart\_intermediates/classes.jar

# 预置Apk方法

<https://www.jianshu.com/p/9af0a34d3eed>

## 用户可以卸载，恢复出厂设置时不能恢复？

LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_DATA\_APPS)

## APK使得用户可以卸载，并且恢复出厂设置时能够恢复

LOCAL\_MODULE\_PATH := $(TARGET\_OUT)/vendor/operator/app

# MAKE 语法

在Android系统中，所有的应用都是以apk的形式存在，以系统应用为例

mmm packages/app/Settings

我们带着这个命令来了解apk的mmm编译原理

大致会分为如下几个步骤：

step1. 通过source 命令，读入envsetup.sh里面定义的各种命令，比如mm、mma、mmm、godir、croot等，方便我们在当前终端进行相关的命令输入。

step2.lunch将要编译的产品，生成产品相关的参数配置。

source build/envsetup.sh；

lunch

step3. 开始执行编译命令，生成目标的依赖关系。

mmm packages/app/Settings

另外一点是在make文件中可以指定具体的manifest文件：   
**LOCAL\_MANIFEST\_FILE := <manifest file path>**

## 模块目录Android.mk

第三类是针对某个模块（关于模块后文会详细讨论）的 Make 文件。整个系统中，包含了大量的模块，每个模块都有一个专门的 Make 文件，这类文件的名称统一为“Android.mk”，该文件中定义了如何编译当前模块。Build 系统会在整个源码树中扫描名称为“Android.mk”的文件并根据其中的内容执行模块的编译。

### 添加新的模块

关于“模块”的说明在上文中已经提到过，这里不再赘述。

在源码树中，一个模块的所有文件通常都位于同一个文件夹中。为了将当前模块添加到整个 Build 系统中，每个模块都需要一个专门的 Make 文件，该文件的名称为“Android.mk”。Build 系统会扫描名称为“Android.mk”的文件，并根据该文件中内容编译出相应的产物。

需要注意的是：在 Android Build 系统中，编译是以模块（而不是文件）作为单位的，每个模块都有一个唯一的名称，一个模块的依赖对象只能是另外一个模块，而不能是其他类型的对象。对于已经编译好的二进制库，如果要用来被当作是依赖对象，那么应当将这些已经编译好的库作为单独的模块。对于这些已经编译好的库使用 BUILD\_PREBUILT 或 BUILD\_MULTI\_PREBUILT。例如：当编译某个 Java 库需要依赖一些 Jar 包时，并不能直接指定 Jar 包的路径作为依赖，而必须首先将这些 Jar 包定义为一个模块，然后在编译 Java 库的时候通过模块的名称来依赖这些 Jar 包。

下面，我们就来讲解 Android.mk 文件的编写：

Android.mk 文件通常以以下两行代码作为开头：

|  |  |
| --- | --- |
| 1  2 | LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS) |

这两行代码的作用是：

1. 设置当前模块的编译路径为当前文件夹路径。
2. 清理（可能由其他模块设置过的）编译环境中用到的变量。

为了方便模块的编译，Build 系统设置了很多的编译环境变量。要编译一个模块，只要在编译之前根据需要设置这些变量然后执行编译即可。它们包括：

* LOCAL\_SRC\_FILES：当前模块包含的所有源代码文件。
* LOCAL\_MODULE：当前模块的名称，这个名称应当是唯一的，模块间的依赖关系就是通过这个名称来引用的。
* LOCAL\_C\_INCLUDES：C 或 C++ 语言需要的头文件的路径。
* LOCAL\_STATIC\_LIBRARIES：当前模块在静态链接时需要的库的名称。
* LOCAL\_SHARED\_LIBRARIES：当前模块在运行时依赖的动态库的名称。
* LOCAL\_CFLAGS：提供给 C/C++ 编译器的额外编译参数。
* LOCAL\_JAVA\_LIBRARIES：当前模块依赖的 Java 共享库。
* LOCAL\_STATIC\_JAVA\_LIBRARIES：当前模块依赖的 Java 静态库。
* LOCAL\_PACKAGE\_NAME：当前 APK 应用的名称。
* LOCAL\_CERTIFICATE：签署当前应用的证书名称。
* LOCAL\_MODULE\_TAGS：当前模块所包含的标签，一个模块可以包含多个标签。标签的值可能是 debug, eng, user，development 或者 optional。其中，optional 是默认标签。标签是提供给编译类型使用的。不同的编译类型会安装包含不同标签的模块，关于编译类型的说明如表 7 所示：

表 7. 编译类型的说明

| **名称** | **说明** |
| --- | --- |
| **eng** | 默认类型，该编译类型适用于开发阶段。 当选择这种类型时，编译结果将：   * 安装包含 eng, debug, user，development 标签的模块 * 安装所有没有标签的非 APK 模块 * 安装所有产品定义文件中指定的 APK 模块 |
| **user** | 该编译类型适合用于最终发布阶段。 当选择这种类型时，编译结果将：   * 安装所有带有 user 标签的模块 * 安装所有没有标签的非 APK 模块 * 安装所有产品定义文件中指定的 APK 模块，APK 模块的标签将被忽略 |
| **userdebug** | 该编译类型适合用于 debug 阶段。 该类型和 user 一样，除了：   * 会安装包含 debug 标签的模块 * 编译出的系统具有 root 访问权限 |

表 3 中的文件已经定义好了各种类型模块的编译方式。所以要执行编译，只需要引入表 3 中对应的 Make 文件即可（通过常量的方式）。例如，要编译一个 APK 文件，只需要在 Android.mk 文件中，加入“include $(BUILD\_PACKAGE)

除此以外，Build 系统中还定义了一些便捷的函数以便在 Android.mk 中使用，包括：

$(call my-dir)：获取当前文件夹路径。

$(call all-java-files-under, <src>)：获取指定目录下的所有 Java 文件。

$(call all-c-files-under, <src>)：获取指定目录下的所有 C 语言文件。

$(call all-Iaidl-files-under, <src>) ：获取指定目录下的所有 AIDL 文件。

$(call all-makefiles-under, <folder>)：获取指定目录下的所有 Make 文件。

$(call intermediates-dir-for, <class>, <app\_name>, <host or target>, <common?> )：获取 Build 输出的目标文件夹路径。

清单 2 和清单 3 分别是编译 APK 文件和编译 Java 静态库的 Make 文件示例：

## 命令说明

### framework-res-package-target

#.PHONY: 指定framework-res-package-target为伪目标  
# define a global intermediate target that other module may depend on.  
.PHONY: framework-res-package-target  
framework-res-package-target: $(LOCAL\_BUILT\_MODULE)

### LOCAL\_AAPT\_FLAGS

# Tell aapt to create "extending (non-application)" resource IDs,

# since these resources will be used by many apps.

LOCAL\_AAPT\_FLAGS := -x

#告诉应用编译的资源可以被其他应用程序使用

LOCAL\_AAPT\_FLAGS += --private-symbols com.android.internal

### LOCAL\_AAPT\_FLAG

Sdfg

1. # 定义jar包 aar包的路径 :后面跟的是相对Android.mk文件的相对路径
2. LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES := supportinternal:lib/support-annotations-22.2.1-sources.jar \
3. supportannmtions:lib/internal\_impl-22.2.1.jar \
4. support4:lib/support-v4-22.2.1.aar \

LOCAL\_AAPT\_FLAGS := --auto-add-overlay

LOCAL\_AAPT\_FLAGS += --extra-packages com.tomorrow\_p.simple

### LOCAL\_CFLAGS

提供给 C/C++ 编译器的额外编译参数。

LOCAL\_CFLAGS := -Werror

-Wall 打开gcc的所有警告

-Werror，它要求gcc将所有的警告当成错误进行处理

### LOCAL\_EXPORT\_PACKAGE\_RESOURCES

#允许app的资源被其它模块使用

#默认为null

# Create package-export.apk, which other packages can use to get

# PRODUCT-agnostic resource data like IDs and type definitions.

LOCAL\_EXPORT\_PACKAGE\_RESOURCES := true

### LOCAL\_MODULE\_TAGS

#指定该模块的编译版本为optional  
#user: 指该模块只在user版本下才编译  
#eng: 指该模块只在eng版本下才编译  
#tests: 指该模块只在tests版本下才编译  
#optional:指该模块在所有版本下都编译  
LOCAL\_MODULE\_TAGS := optional

### LOCAL\_PATH := $(call my-dir)

#把该Makefile所在的目录设定为当前目录   
LOCAL\_PATH := $(call my-dir)

1. # 每个Android.mk文件必须以定义LOCAL\_PATH为开始。它用于在开发tree中查找源文件。

2. # my-dir 则由Build System提供。返回包含Android.mk的目录路径。固定写法

### include $(CLEAR\_VARS)

清除编译规则模板使用的变量的值。在./build/core/clear\_vars.mk中定义

1. # CLEAR\_VARS 变量由Build System提供。并指向一个指定的GNU Makefile，负责清理多余的LOCAL\_xxx.

2. # 这个清理动作是必须的，因为所有的编译控制文件由同一个GNU Make解析和执行，其变量是全局的

# 所以清理后才能避免相互影响 比如LOCAL\_MODULE, LOCAL\_SRC\_FILES, LOCAL\_STATIC\_LIBRARIES

include $(BUILD\_EXECUTABLE)

可执行程序

将你列出的Source编译成一个可执行Native程序

### LOCAL\_NO\_STANDARD\_LIBRARIES

#不是标准的java库  
LOCAL\_NO\_STANDARD\_LIBRARIES := true

### LOCAL\_STATIC\_JAVA\_LIBRARIES

为引用的静态库(静态库是需要编译进apk的)

### LOCAL\_STATIC\_JAVA\_AAR\_LIBRARIES := support4

1. # 定义引用的aar包：名字自定义，后面会针对名字进行路径说明

# aar包内如果有jar包，需要导出引用比如V4包有jar包

### LOCAL\_JAVA\_LIBRARIES

为非静态库(非静态库是Android系统自带的库)

### LOCAL\_JARJAR\_RULES

包名映射

LOCAL\_JARJAR\_RULES := $(LOCAL\_PATH)/jarjar-rules.txt

可以先看jarjar-rules.txt文件的内容：

rule org.bouncycastle.\*\* com.android.@0

rule com.android.org.chromium.\*\* com.test.android.org.chromium.@1

The build will change all org.bouncycastle to com.android.org.bouncycastle. Therefore, in your classes which are dependent on the library produced, the import statements should look like: \

import com.android.org.bouncycastle...

change all com.android.org.chromium to com.test.android.org.chromium

rule com.squareup.\*\* com.android.@1

rule okio.\*\* [com.android.okhttp.okio.@1](mailto:com.android.okhttp.okio.@1)

<https://blog.csdn.net/FengShanChuiChui/article/details/72329104>

### LOCAL\_SRC\_FILES

1. # 定义src文件路径
2. LOCAL\_SRC\_FILES := $(call all-subdir-Java-files) $(call all-java-files-under, src) $(call all-renderscript-files-under, src)

### LOCAL\_RESOURCE\_DIR

Hh

1. # 定义res文件路径

LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res

### LOCAL\_PACKAGE\_NAME

1. # 编译后名称
2. LOCAL\_PACKAGE\_NAME := SysemUI

### LOCAL\_OVERRIDES\_PACKAGES

此变量可以使其他的模块不加入编译，替代编译包！

1.LOCAL\_OVERRIDES\_PACKAGES := privison

2.并在out目录下把Launcher2.apk删除掉，这样系统就不会将原生的Launcher编译进去，

#说明是不让privison这个apk编译到系统。 也不会生成到(TARGET\_OUT)/priv-app目录下。

当然,另外的方法。也可以在mk文件中编译开关中将相应Launcher注释掉

### LOCAL\_PRIVILEGED\_MODULE

Sdfg

1. # 预置到 /system/pri-app/;不添加此属性，编译后的路径为/system/app/SystemUI

2. # 在同样具有系统权限情况下，/system/pri-app下的apk权限比/system/app下大

3. LOCAL\_PRIVILEGED\_MODULE := true

### LOCAL\_MODULE\_PATH

1. # 编译生成目标的安装路径

2. # 在build/core/envsetup.mk中定义;

TARGET\_ROOT\_OUT：表示根文件系统;T

ARGET\_OUT：表示system文件系统;

TARGET\_OUT\_DATA：表示data文件系统。

3. # 预置的apk用户可以卸载,恢复出厂设置不可恢复

4. LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_DATA\_APPS)

5. # 预置的apk用户可以卸载,恢复出厂设置可恢复

LOCAL\_MODULE\_PATH := $(TARGET\_OUT)/vendor/operator/app

LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_JAVA\_LIBRARIES)

### LOCAL\_CERTIFICATE := platform

#为APK指定签名具有系统权限

#签名有四种类型:testkey, platform, shared, media 默认为testkey

LOCAL\_CERTIFICATE := platform

Fgf

1. # 定义生成的apk使用的key
2. # 指定平台签名,生成Settings.apk文件之后,系统会使用如下两个文件对Settings.apk进行签名,这个系统设置就拥有系统权限了
3. # (源码根目录)MSM8939\_AndroidL/build/target/product/security/platform.pk8
4. # (源码根目录)MSM8939\_AndroidL/build/target/product/security/platform.x509.pem
5. # Settings光使用Platform签名还不行,需要在AndroidManifest.xml文件**package**="xxx"后指定共享用户ID,并将添加属性: coreApp="true",才会变成内核应用
6. # 任何一个Android应用,如果使用前面的配置,并在Android源代码中使用mm/mmm命令进行编译,其权限就会和Settings完全一样
7. # 每一个版本(包括同一版本不同分支)Android源代码所带的Platform签名都不一样,使用不同的签名会导致apk程序认证不通过,尽管system/app下已存在对应apk
8. # 设置coreApp="true"和android:sharedUserId="android.uid.system"的apk将用于系统级的权限(不需要root权限),Android采用了Linux的ACL权限管理方式
9. LOCAL\_CERTIFICATE := platform

### LOCAL\_PROGUARD\_FLAG\_FILES := proguard.flags

指定混淆标识文件

LOCAL\_PROGUARD\_ENABLED := disabled

### LOCAL\_MANIFEST\_FILE := $(LOCAL\_PATH)/AndroidManifest.xml

1. # 指明合并后的AndroidManifest.xml的路径 （一般不用指定）
2. LOCAL\_MANIFEST\_FILE := $(LOCAL\_PATH)/AndroidManifest.xml

### include $(BUILD\_MULTI\_PREBUILT)

1. # 因引入jar包或aar包，须加上这句

include $(BUILD\_MULTI\_PREBUILT)

### include $(call all-makefiles-under,$(LOCAL\_PATH))

完整的

### Include文件夹

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. include frameworks/opt/setupwizard/library/common-full-support.mk
2. include frameworks/base/packages/SettingsLib/common.mk

其中setupwizard/navigationbar和setupwizard/library/和SettingsLib是包含res的工程库,因此不能导入jar包

### include $(BUILD\_PACKAGE)

#执行BUILD\_PACKAGE。开始编译APK  
#BUILD\_PACKAGE定义在config.mk:  
#BUILD\_PACKAGE:= $(BUILD\_SYSTEM)/package.mk

  # Tell it to build an APK-->直译（告诉它来建立一个APK）

### ONE\_SHOT\_MAKEFILE

**# Use the following include to make our test apk.**

ifeq (,$(ONE\_SHOT\_MAKEFILE))

include $(call all-makefiles-under,$(LOCAL\_PATH))

endif

### TARGET\_OUT\_JAVA\_LIBRARIES

最终生成在/system/framework下面

### Android.mk

Df

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. LOCAL\_PATH := $(call my-dir)
3. include $(CLEAR\_VARS)
5. LOCAL\_STATIC\_JAVA\_LIBRARIES := supportinternal supportannotions
7. LOCAL\_STATIC\_JAVA\_AAR\_LIBRARIES := support4
9. # 定义src文件路径
10. LOCAL\_SRC\_FILES := $(call all-subdir-Java-files) $(call all-java-files-under, src) $(call all-renderscript-files-under, src)
11. # 定义res文件路径
12. LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res
14. LOCAL\_PACKAGE\_NAME := SysemUI
15. LOCAL\_PRIVILEGED\_MODULE := **true**
16. # 编译生成目标的安装路径
17. # 在build/core/envsetup.mk中定义;TARGET\_ROOT\_OUT：表示根文件系统;TARGET\_OUT：表示system文件系统;TARGET\_OUT\_DATA：表示data文件系统。
18. # 预置的apk用户可以卸载,恢复出厂设置不可恢复
19. LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_DATA\_APPS)
20. # 预置的apk用户可以卸载,恢复出厂设置可恢复
21. LOCAL\_MODULE\_PATH := $(TARGET\_OUT)/vendor/operator/app
23. # 定义生成的apk使用的key
25. LOCAL\_CERTIFICATE := platform
26. # 指定混淆标识文件
27. LOCAL\_PROGUARD\_FLAG\_FILES := proguard.flags
28. # 定义jar包 aar包的路径 :后面跟的是相对Android.mk文件的相对路径
29. LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES := supportinternal:lib/support-annotations-22.2.1-sources.jar \
30. supportannmtions:lib/internal\_impl-22.2.1.jar \
31. support4:lib/support-v4-22.2.1.aar \
33. include $(CLEAR\_VARS)
35. # 因为aar包的存在，这个主要是用于合并AndroidManifest.xml 和res资源文件
36. LOCAL\_AAPT\_FLAG := \
37. --auto-add-overlay \
38. --extra-**package** android.support.v4 \
39. --extra-**package** com.openthos.launcher.openthoslauncher
41. # 指明合并后的AndroidManifest.xml的路径 （一般不用指定）
42. LOCAL\_MANIFEST\_FILE := $(LOCAL\_PATH)/AndroidManifest.xml
44. # 因引入jar包或aar包，须加上这句
45. include $(BUILD\_MULTI\_PREBUILT)
47. include $(call all-makefiles-under,$(LOCAL\_PATH))

## 如何引用aar包

分成3步

1. 编译aar包：

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. include $(CLEAR\_VARS)
2. LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES := <aar alias>:libpath/<lib file>.aar
3. include $(BUILD\_MULTI\_PREBUILT)
4. 引用aar包

LOCAL\_STATIC\_JAVA\_AAR\_LIBRARIES:= <aar alias>

1. 资源声明

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. LOCAL\_AAPT\_FLAGS := \
2. --auto-add-overlay \
3. --extra-packages <aar **package** name>

## Android.mk添加第三方jar

F

LOCAL\_PATH:= $(call my-dir)

include $(CLEAR\_VARS)

LOCAL\_JAVA\_LIBRARIES := bouncycastle conscrypt telephony-common

LOCAL\_STATIC\_JAVA\_LIBRARIES := android-support-v4 android-support-v13 jsr305 fc909 quciifplib \

umeng

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := \

$(call all-java-files-under, src) \

src/com/android/settings/EventLogTags.logtags

LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res

LOCAL\_SRC\_FILES += \

src/com/android/location/XT/IXTSrv.aidl \

src/com/android/location/XT/IXTSrvCb.aidl

LOCAL\_PACKAGE\_NAME := Settings

LOCAL\_CERTIFICATE := platform

LOCAL\_PRIVILEGED\_MODULE := true

LOCAL\_OVERRIDES\_PACKAGES := Email

LOCAL\_PROGUARD\_ENABLED := disabled

LOCAL\_PROGUARD\_FLAG\_FILES := proguard.flags

include frameworks/opt/setupwizard/navigationbar/common.mk

include $(BUILD\_PACKAGE)

**# Use the following include to make our test apk.**

ifeq (,$(ONE\_SHOT\_MAKEFILE))

include $(call all-makefiles-under,$(LOCAL\_PATH))

endif

include $(CLEAR\_VARS)

LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES := umeng:libs/umeng-analytics-v6.0.1.jar

**#冒号前面为jar别名，后面为jar文件的实际路径**

include $(BUILD\_MULTI\_PREBUILT)

## framework添加第三方jar

## 举例

Gh

更多案例

<https://blog.csdn.net/qq_25804863/article/details/48669003>

### make logtags

在指定子目录下查找所有.logtags文件

SRC\_FILES := $(call all-logtags-files-under,src)

$1:指定子目录名称src/com/android/settings/EventLogTags.logtags

1. LOCAL\_PATH:= $(call my-dir)
2. include $(CLEAR\_VARS)
4. LOCAL\_SRC\_FILES := \
5. $(call all-logtags-files-under, src)
7. LOCAL\_MODULE := settings-logtags
9. include $(BUILD\_STATIC\_JAVA\_LIBRARY)

make的out目录

python脚本build/tools/java-event-log-tags.py则负责将EventLogTags.logtags以及调用转化为java文件，或者是将java文件中的writeEvent调用转为标准的java调用，以及生成system/etc/event-log-tags文件(对于使用来说，分析这个python文件没有多大意义)。   
  以frameworks/base/services/core/java/com/android/server/am/EventLogTags.logtags文件为例，该文件编译过程中通过python脚本生成的对应java文件在out目录中：

out/target/common/obj/JAVA\_LIBRARIES/services.core\_intermediates/src/com/android/server/am/EventLogTags.java

* 1

  下面看看logcat -b events是怎么读取并显示event log的：

### 编译一个 APK 文件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  # 获取所有子目录中的 Java 文件  LOCAL\_SRC\_FILES := $(call all-subdir-java-files)  # 当前模块依赖的静态 Java 库，如果有多个以空格分隔  LOCAL\_STATIC\_JAVA\_LIBRARIES := static-library  # 当前模块的名称  LOCAL\_PACKAGE\_NAME := LocalPackage  # 编译 APK 文件  include $(BUILD\_PACKAGE) |

### 编译一个 Java 的静态库

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)    # 获取所有子目录中的 Java 文件  LOCAL\_SRC\_FILES := $(call all-subdir-java-files)    # 当前模块依赖的动态 Java 库名称  LOCAL\_JAVA\_LIBRARIES := android.test.runner    # 当前模块的名称  LOCAL\_MODULE := sample    # 将当前模块编译成一个静态的 Java 库  include $(BUILD\_STATIC\_JAVA\_LIBRARY) |

### 引用系统的库 liblog

LOCAL\_SHARED\_LIBRARIED += liblog

# 编译会从/out/targer/product/fiber-a31st/obj/lib/liblog.so找对应的库

### Framework-res

<https://blog.csdn.net/mingli198611/article/details/7057584>

# Include resources generated by system RenderScript files.

framework\_GENERATED\_SOURCE\_DIR := $(call intermediates-dir-for,JAVA\_LIBRARIES,framework,,COMMON)/src

framework\_RenderScript\_STAMP\_FILE := $(framework\_GENERATED\_SOURCE\_DIR)/RenderScript.stamp

#LOCAL\_RESOURCE\_DIR := $(framework\_GENERATED\_SOURCE\_DIR)/renderscript/res $(LOCAL\_PATH)/res

include $(BUILD\_PACKAGE)

# Make sure the system .rs files get compiled before building the package-export.apk.

# $(resource\_export\_package): $(framework\_RenderScript\_STAMP\_FILE)

## Android mk 文件配置签名

我们知道在 build/target/product/security 目录中有四组默认签名供，Android.mk在编译APK使用：

* 1、testkey：普通APK，默认情况下使用。
* 2、platform：该APK完成一些系统的核心功能。经过对系统中存在的文件夹的访问测试，   
  这种方式编译出来的APK所在进程的UID为system。
* 3、shared：该APK需要和home/contacts进程共享数据。
* 4、media：该APK是media/download系统中的一环。

举例说明一下。

系统中所有使用android.uid.system作为共享UID的APK，

都会首先在manifest节点中增加android:sharedUserId="android.uid.system"，

然后在Android.mk中增加LOCAL\_CERTIFICATE := platform。可以参见Settings等

系统中所有使用android.uid.shared作为共享UID的APK，

都会在manifest节点中增加android:sharedUserId="android.uid.shared"，

然后在Android.mk中增加LOCAL\_CERTIFICATE := shared。可以参见Launcher等

系统中所有使用android.media作为共享UID的APK，

都会在manifest节点中增加android:sharedUserId="android.media"，

然后在Android.mk中增加LOCAL\_CERTIFICATE := media。可以参见Gallery等。

## REF

[**Android.mk文件语法规范（Android.mk File）**](https://blog.csdn.net/smfwuxiao/article/details/8530742)

## MAKE APK特殊需求

1. 禁止硬编码：资源本地化问题
2. Preference禁止key用@string方式引用，menifest禁止android

# Android.mk分析

## framework-res.apk

/frameworks/base/core/res/Android.mk

目标模块：framework-res.apk

### **特点**

# 1.支持应用程序编译的资源能够被其它应用使用

# 2.该APK具有系统权限

# 3.能够指定一个全局的依赖于LOCAL\_BUILT\_MODULE的中间目标

### 脚本解释

把该Makefile所在的目录设定为当前目录   
LOCAL\_PATH := $(call my-dir)   
#清除编译规则模板使用的变量的值。在./build/core/clear\_vars.mk中定义   
include $(CLEAR\_VARS)   
#不是标准的java库   
LOCAL\_NO\_STANDARD\_LIBRARIES := true   
#设定包名或者生成的apk的名称   
LOCAL\_PACKAGE\_NAME := framework-res   
#为APK指定签名具有系统权限   
#签名有四种类型:testkey, platform, shared, media 默认为testkey   
LOCAL\_CERTIFICATE := platform   
#告诉应用编译的资源可以被其他应用程序使用

# Tell aapt to create "extending (non-application)" resource IDs,   
# since these resources will be used by many apps.

LOCAL\_AAPT\_FLAGS := -x

私有资源说明

LOCAL\_AAPT\_FLAGS += --private-symbols com.android.internal

#optional:指该模块在所有版本下都编译  
LOCAL\_MODULE\_TAGS := optional

LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_JAVA\_LIBRARIES)//最终生成在/system/framework下面

#允许app的资源被其它模块使用

LOCAL\_EXPORT\_PACKAGE\_RESOURCES := true

### 自定义framework-res资源包

传统的Android系统只有一个framework-res.apk资源包，第三方厂商在进行rom定制时会直接修改framework res资源，达到美化目的。但是这种方式跟原生资源的耦合度过高，在系统迁移或者framework ui移植时需要进行人工merge，工作量巨大。通过为Android添加一个新的独立的资源包，可以将厂商定制资源独立出来，可移植、可维护性非常好。

做一个custom-framework-res.apk

### 收益

### 实施

#### 新建自定义资源包 工程

涉及模块:./build

Framework/base:

Package/app

##### “res\_custemer”文件夹

在framework/base/core/下新建名为“res\_custemer”文件夹,结构如下



##### 编写AndroidManifest.xml文件

Dfg

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="custemer" coreApp="true" android:sharedUserId="android.uid.system"

android:sharedUserLabel="@null">

<eat-comment />

<protected-broadcast android:name="android.intent.action.SCREEN\_OFF" />

<permission android:name="android.permission.ADVANCED\_WIDGET\_API"

android:protectionLevel="normal" />

<application android:process="system"

android:persistent="true"

android:hasCode="false"

android:label="@null"

android:allowClearUserData="false"

android:killAfterRestore="false"

android:icon="@null">

</application>

</manifest>

"custemer" 为资源包名

##### 编写Android.mk文件

LOCAL\_PATH := $(call my-dir)

include $(CLEAR\_VARS)

# include apicheck.mk later, we need the build pass to prepare the first version

# include $(LOCAL\_PATH)/apicheck.mk

LOCAL\_PACKAGE\_NAME := custemer-res

LOCAL\_CERTIFICATE := platform

LOCAL\_AAPT\_FLAGS := -x3

# Tell aapt to build resource in utf16(the ROM will be enlarged),

# in order to save RAM size for string cache table

ifeq (yes,strip$(MTK\_GMO\_RAM\_OPTIMIZE))

LOCAL\_AAPT\_FLAGS += --utf16

endif

LOCAL\_NO\_CUSTEMERRES := true

LOCAL\_MODULE\_TAGS := optional

# Install this alongside the libraries.

LOCAL\_MODULE\_PATH := $(TARGET\_OUT\_JAVA\_LIBRARIES)

# Create package-export.apk, which other packages can use to get

# PRODUCT-agnostic resource data like IDs and type definitions.

LOCAL\_EXPORT\_PACKAGE\_RESOURCES := true

include $(BUILD\_PACKAGE)

# define a global intermediate target that other module may depend on.

.PHONY: gome-res-package-target

gome-res-package-target: $(LOCAL\_BUILT\_MODULE)

"LOCAL\_AAPT\_FLAGS := -x3 " value 的定义需要与 "frameworks\base\libs\androidfw\ResourceTypes.cpp"中 所定义的resource ID 一致

#### 添加编译依赖关系

我只是定制一个主题有必要换么，后面蓝牙开关窗口

添加系统级res资源包

<https://www.jianshu.com/p/89c3ee73e4c4>

<http://blog.163.com/zhou_411424/blog/static/197362156201311135301754>

### 运行时适配

### 引用

Freamework自引用：

Java引用：可以引用

Xml方式

编译

密码

### 编译运行

但是运行，但是为何不生效呢

先不纠结了，回退吧

在debug，去掉编译优化的环境下。

1.首次引入了几个color资源ID，发现不能启动系统了，提示资源找不到，reboot和恢复出厂设置都没有用，烧录mkandroid得到的boot.img,recovery.img system.img后才正常

2.没有新增id，mm之后push到分区里面是ok的

如何提高定制效率呢

新建资源没有生效

Adb reboot 也没有用

1. 恢复工厂设置
2. 此时删除一下 /data/dalvik-cache

## framework.jar

### 编译验证

1.关闭编译优化：/device/XXX/XXX/BoardConfig.mk

WITH\_DEXPREOPT := false

2.第一次更新framework.jar, WITH\_ DEX\_PREOPT设为禁用false，编译大包loader/parameter/uboot/trust/misc/res/system.img/kernel.img/recovery.img/boot.img/等等所有镜像！,用刷机工具android TOOL烧入固件。单纯的烧录system.img/ system.img/kernel.img/recovery.img/boot.img/会造成无法启动系统，停留在kernel 的log上面,最小img集合还没有找到

3.第N次更新，后面保持WITH\_ DEX\_PREOPT为false,

adb push framework.jar /system/framework/ framework.jar

adb shell stop

adb shell start

### 依赖framework-res

No rule to make target `out/target/common/obj/APPS/framework-res\_intermediates/src/R.stamp

分析：没有编译出framework-res

解决：mmm frameworks\base\core\res

<https://blog.csdn.net/zhangjm123/article/details/7848293>

# Android平台Overlay机制

Android overlay 机制允许在不修改packages中apk的情况下，来自定义 framework和package中的资源文件，实现资源的定制。来达到显示不同的**UI得目的（如MIUI）**。以下几类能够通过该机制定义：

(1)，Configurations (string, bool, bool-array)

(2)，Localization (string, string-array)

(3)，UI Appearance (color, drawable, layout, style, theme, animation)

(4)，Raw resources (audio, video, xml)

More: <http://developer.android.com/guide/topics/resources/available-resources.html>

## 为产品添加Overlay目录

### Product Overlays与Device Overlays

Sdfg

有两种不同的overaly目录定义，来影响最终的效果:

PRODUCT\_PACKAGE\_OVERLAYS: used by a particular product

DEVICE\_PACKAGE\_OVERLAYS: used several products that share a common device model

如果包含同一资源，那么 PRODUCT\_PACKAGE\_OVERLAYS 将覆盖 DEVICE\_PACKAGE\_OVERLAYS 中的, 这两个定义如下:

build/core/package.mk (Line: 93)

package\_resource\_overlays := $(strip \

$(wildcard $(foreach dir, $(PRODUCT\_PACKAGE\_OVERLAYS), \

$(addprefix $(dir)/, $(LOCAL\_RESOURCE\_DIR)))) \

$(wildcard $(foreach dir, $(DEVICE\_PACKAGE\_OVERLAYS), \

$(addprefix $(dir)/, $(LOCAL\_RESOURCE\_DIR)))))

PRODUCT\_PACKAGE\_OVERLAYS & DEVICE\_PACKAGE\_OVERLAYS 功能是一样的，只是优先级不一样：PRODUCT\_PACKAGE\_OVERLAYS 优先于 DEVICE\_PACKAGE\_OVERLAYS。

### 1.2 改变makefile来添加overlays的编译项

为了添加一个overlay目录, 需要修改产品的makefile：

(for example: device/vendor-name/device-name/product-name.mk)

添加以下几行:

PRODUCT\_PACKAGE\_OVERLAYS := device/vendor-name/device-name/product-name/overlay

$(PRODUCT\_PACKAGE\_OVERLAYS)

Or：

DEVICE\_PACKAGE\_OVERLAYS := device/vendor-name/device-name/common/overlay

$(DEVICE\_PACKAGE\_OVERLAYS)

(如: device/vendor-name/device-name/device\_base.mk)中添加：

LOCAL\_PATH := device/vendor-name/device-name

DEVICE\_PACKAGE\_OVERLAYS := $(LOCAL\_PATH)/overlay

如果要定义多个overlays目录，需要用空格隔开。如果有多个目录，并且都包含同一资源的定义，那么将使用第一个定义的目录中的资源。

### 1.3 在overlay目录下创建资源文件

想覆盖Android系统自带package中资源文件, 那么在overlay目录下必须包含和要替换package相同的路径, 该路径是Android源码目录的相对路径.

For example, 如果我们想要替换以下目录的资源文件:

packages/apps/Settings/res/

那么在overlay目录下面必须创建一样的目录:

....../overlay目录/packages/apps/Settings/res/

然后放入想要替换的资源（必须和系统package相同路径和文件名）。

注意:

(1)，For color, bool, string, array, style/theme types, the resource values are identifed by their keys, so for these types, there is no need to put the resources in a file with the same name as in the original base package.

(2)，For layout, animation, picture drawables and raw types, the resources are indentifed by their file name, and overlay for these resources should keep the file name same as in the base packages.

## TODO

Overlay

https://my.oschina.net/kingguary/blog/160190

<https://www.jianshu.com/p/9304089c513d>

## 在APK中检测资源

## REF

[Android平台Overlay机制](https://blog.csdn.net/wh_19910525/article/details/39254815)

# AS环境配置

Android Studio 工程

在项目根目录下新建 Android.mk 文件，

<http://zhuanlan.51cto.com/art/201705/540532.htm>

# 系统应用环境配置方案

关键字：系统应用 系统设置 android7.1 android stdio settings

## 概述

### 行业现状

芯片公司：MTK/RK/intel, ui定制非常少，并且主要是处理系统底层，一般用source insight等文本工具

手机厂商：hw/小米/酷派， 目前一般都用AS，各种导入包/依赖/AAPT编译问题，如何导入和解决相关的编译问题/网上参考资料较少。一般是专门的sdk团队负责搭建，主要编译framework各种jar，暴露hide方法。

迁移到AS环境下有一定的工作量：比如系统设置在android7.0又多依赖了settingslib，然后product字段/资源重载机制as的aapt又不支持，

### 开发环境方案对比

直接把文件夹拖入eclipse/android studio，本质和记事本效率其实差别不大

|  |  |  |
| --- | --- | --- |
| 生命周期 | 记事本环境方案 | AS环境方案 |
| 1.编码 | 工具：  vi/sourceinsight/notepad/  文档结构不合理：res和src文档结构不合理，修改资源文件和代码文件非常不方便  语法助手：几乎没有 | 文档结构：  语法助手：自动：import，layout实时预览，代码自动补全，代码语法检查，重构变量/方法等等等等。。 |
| 2.编译 | linux的mm命令  语法问题出错：回到1 | 自动 |
| 3.安装 | com控制台的adb install | 自动 |
| 4.运行 | 手动点击/或者脚本启动 | 自动 |
| 5.调试 | 业务逻辑问题：回到1： 插入Log.d代码，然后2->3->4  Ui问题：回到1->2->3->4->5 | 直接依附进程，断点调试 |
| 小结 |  |  |
| 效率指数 | ：1 | 3-5 |
| 环境配置难度 | ：0  不用任何配置 | 环境配置难度：7d  很复杂，Settings为例，依赖10个左右的jar包，3个其他工程，需要编译系统sdk，消除as的aapt和linux的aapt差异 |
| 系统覆盖率 | 100%，应用层，系统层，底层都ok | 20%，只支持部分apk，framwork/native/kernel/driver无法支持 |

目前我们对settings和systemui ui定制多，采用as符合当前业务和效率的需要

### 代码基本框架

由于android系统更新频繁，应用定制必须考虑兼容性问题，做到强移植性，减少如android7.0迁移到8.0的工作量。

### 方案目标

目前我们对settings和systemui ui定制多，采用as符合当前业务和效率的需要。结合目前的协作和工作环境，并且都是远程登陆到服务器工作。该方案需要实现以下目标。

* [as](http://1.as/)环境：在as环境下，可以正常编译，充分利用ide的高效功能，提取系统class.jar，系统签名
* 兼容多种编译方式：支持mk和as模块编译，支持系统全量编译，尽量减少两种编译方式代码的差异性
* 便捷的git操作：团队协作，合理gitignore，可以从服务器代码git pull到本地编写代码
* 可移植性强：便于从android7.1直接迁移到android8.0

### AS说明

Android Studio 3.1.2

Build #AI-173.4720617, built on April 14, 2018

JRE: 1.8.0\_152-release-1024-b02 amd64

JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Windows 7 6.1

## AS本地源码环境搭建-Settings依赖分析

基本思想：

初手直接建立git init->优先保证可以as编译删文件->逐个恢复文件为线索，导入jar包。

高手可以直接一步到位，走mk分析为线索

### 查找Settings依赖库(Android.mk)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_PATH:= $(call my-dir)
2. include $(CLEAR\_VARS)
4. LOCAL\_SRC\_FILES := \
5. $(call all-logtags-files-under, src)
7. LOCAL\_MODULE := settings-logtags
9. include $(BUILD\_STATIC\_JAVA\_LIBRARY)
11. # Build the Settings APK
12. include $(CLEAR\_VARS)
14. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common
15. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
16. android-support-v4 \
17. android-support-v13 \
18. android-support-v7-recyclerview \
19. android-support-v7-preference \
20. android-support-v7-appcompat \
21. android-support-v14-preference \
22. jsr305 \
23. settings-logtags
25. LOCAL\_MODULE\_TAGS := optional
27. LOCAL\_SRC\_FILES := \
28. $(call all-java-files-under, src) \
29. $(call all-java-files-under, src-kpad)
31. LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res \
32. $(LOCAL\_PATH)/res-kpad \
33. frameworks/support/v7/preference/res \
34. frameworks/support/v14/preference/res \
35. frameworks/support/v7/appcompat/res \
36. frameworks/support/v7/recyclerview/res
38. LOCAL\_PACKAGE\_NAME := Settings
39. LOCAL\_CERTIFICATE := platform
40. LOCAL\_PRIVILEGED\_MODULE := **true**
42. LOCAL\_PROGUARD\_FLAG\_FILES := proguard.flags
44. LOCAL\_AAPT\_FLAGS := --auto-add-overlay \
45. --extra-packages android.support.v7.preference:android.support.v14.preference:android.support.v17.preference:android.support.v7.appcompat:android.support.v7.recyclerview
47. ifneq ($(INCREMENTAL\_BUILDS),)
48. LOCAL\_PROGUARD\_ENABLED := disabled
49. LOCAL\_JACK\_ENABLED := incremental
50. LOCAL\_DX\_FLAGS := --multi-dex
51. LOCAL\_JACK\_FLAGS := --multi-dex **native**
52. endif
54. include frameworks/opt/setupwizard/library/common-full-support.mk
55. include frameworks/base/packages/SettingsLib/common.mk
57. include $(BUILD\_PACKAGE)
59. # Use the following include to make our test apk.
60. ifeq (,$(ONE\_SHOT\_MAKEFILE))
61. include $(call all-makefiles-under,$(LOCAL\_PATH))
62. endif

#### LOCAL\_STATIC\_JAVA\_LIBRARIES

为引用的静态库(静态库是需要编译进apk的)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
2. android-support-v4 \
3. android-support-v13 \
4. android-support-v7-recyclerview \
5. android-support-v7-preference \
6. android-support-v7-appcompat \
7. android-support-v14-preference \
8. jsr305 \
9. settings-logtags

#### LOCAL\_JAVA\_LIBRARIES

为非静态库(非静态库是Android系统自带的库)

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common

#### Include文件夹

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. include frameworks/opt/setupwizard/library/common-full-support.mk
2. include frameworks/base/packages/SettingsLib/common.mk

其中setupwizard/navigationbar和setupwizard/library/和SettingsLib是包含res的工程库,因此不能导入jar包

#### 小结

从Android.mk文件得出结论是:

Settings依赖上述的三个工程(带有res的工程)

静态库

1. android-support-v4 \
2. android-support-v13 \
3. android-support-v7-recyclerview \
4. android-support-v7-preference \
5. android-support-v7-appcompat \
6. android-support-v14-preference \
7. jsr305 \
8. settings-logtags

动态库:

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. bouncycastle core-oj telephony-common ims-common

### SettingsLib

这个模块比较简单

#### comom

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021) @commom.mk

1. LOCAL\_DEX\_PREOPT := **true**
2. ifeq ($(LOCAL\_USE\_AAPT2),**true**)
3. LOCAL\_STATIC\_ANDROID\_LIBRARIES += \
4. android-support-annotations \
5. android-support-v4 \
6. SettingsLib
7. **else**
8. LOCAL\_RESOURCE\_DIR += $(call my-dir)/res
9. LOCAL\_AAPT\_FLAGS += --auto-add-overlay --extra-packages com.android.settingslib
10. LOCAL\_STATIC\_JAVA\_LIBRARIES += \
11. android-support-annotations \
12. android-support-v4 \
13. SettingsLib
14. endif

#### android.mk

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. LOCAL\_PATH := $(call my-dir)
2. include $(CLEAR\_VARS)
3. LOCAL\_USE\_AAPT2 := **true**
4. LOCAL\_MODULE := SettingsLib
5. LOCAL\_DEX\_PREOPT := **false**
6. ANDROID\_COMPILE\_WITH\_JACK := **false**
7. #WITH\_DEXPREOPT := false
8. LOCAL\_SHARED\_ANDROID\_LIBRARIES := \
9. android-support-v4 \
10. android-support-v7-recyclerview \
11. android-support-v7-preference \
12. android-support-v7-appcompat \
13. android-support-v14-preference
14. LOCAL\_RESOURCE\_DIR := $(LOCAL\_PATH)/res
15. LOCAL\_JAR\_EXCLUDE\_FILES := none
16. LOCAL\_SRC\_FILES := $(call all-java-files-under, src)
17. include $(BUILD\_STATIC\_JAVA\_LIBRARY)

fdg

### 查找Setupwizard依赖库(Android.mk)

上文分析到还依赖于setupwizard

1. include frameworks/opt/setupwizard/library/common-full-support.mk

具体文件如下：

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. #
2. # Include this make file to build your application against this module.
3. #
4. # Make sure to include it after you've set all your desired LOCAL variables.
5. # Note that you must explicitly set your LOCAL\_RESOURCE\_DIR before including this file.
6. #
7. # For example:
8. #
9. #   LOCAL\_RESOURCE\_DIR := \
10. #        $(LOCAL\_PATH)/res
11. #
12. #   include frameworks/opt/setupwizard/library/common-full-support.mk
13. #
15. # Check that LOCAL\_RESOURCE\_DIR is defined
16. ifeq (,$(LOCAL\_RESOURCE\_DIR))
17. $(error LOCAL\_RESOURCE\_DIR must be defined)
18. endif
20. # Add --auto-add-overlay flag if not present
21. ifeq (,$(findstring --auto-add-overlay, $(LOCAL\_AAPT\_FLAGS)))
22. LOCAL\_AAPT\_FLAGS += --auto-add-overlay
23. endif
25. # Include setup wizard library, if not already included
26. ifeq (,$(findstring setup-wizard-lib-full-support,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
27. LOCAL\_RESOURCE\_DIR += \
28. $(call my-dir)/main/res \
29. $(call my-dir)/eclair-mr1/res \
30. $(call my-dir)/full-support/res
31. LOCAL\_AAPT\_FLAGS += --extra-packages com.android.setupwizardlib
32. LOCAL\_STATIC\_JAVA\_LIBRARIES += setup-wizard-lib-full-support
33. endif
35. ## Include transitive dependencies below
37. # Include support-v7-appcompat, if not already included
38. ifeq (,$(findstring android-support-v7-appcompat,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
39. LOCAL\_RESOURCE\_DIR += frameworks/support/v7/appcompat/res
40. LOCAL\_AAPT\_FLAGS += --extra-packages android.support.v7.appcompat
41. LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v7-appcompat
42. endif
44. # Include support-v7-recyclerview, if not already included
45. ifeq (,$(findstring android-support-v7-recyclerview,$(LOCAL\_STATIC\_JAVA\_LIBRARIES)))
46. LOCAL\_RESOURCE\_DIR += frameworks/support/v7/recyclerview/res
47. LOCAL\_AAPT\_FLAGS += --extra-packages android.support.v7.recyclerview
48. LOCAL\_STATIC\_JAVA\_LIBRARIES += android-support-v7-recyclerview
49. endif

#### 资源LOCAL\_RESOURCE\_DIR

依赖的资源文件如下

1. LOCAL\_RESOURCE\_DIR += \
2. $(call my-dir)/main/res \
3. $(call my-dir)/eclair-mr1/res \
4. $(call my-dir)/full-support/res
5. = frameworks/support/v7/appcompat/res
6. frameworks/support/v7/recyclerview/res

#### LOCAL\_STATIC\_JAVA\_LIBRARIES

setup-wizard-lib-full-support

android-support-v7-appcompat

android-support-v7-recyclerview

#### setup-wizard-lib-full-support模块分析

setup-wizard-本身又依赖setup-wizard-lib-full-support，这个在android,mk定义了该模块

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. ##
2. # Build the full-support library, which includes RecyclerView and any other support libraries as
3. # they are integrated.
4. #
5. include $(CLEAR\_VARS)
7. LOCAL\_AAPT\_FLAGS := --auto-add-overlay \
8. --extra-packages android.support.v7.appcompat \
9. --extra-packages android.support.v7.recyclerview
10. LOCAL\_MANIFEST\_FILE := main/AndroidManifest.xml
11. LOCAL\_MODULE := setup-wizard-lib-full-support
12. LOCAL\_RESOURCE\_DIR := \
13. $(LOCAL\_PATH)/main/res \
14. $(LOCAL\_PATH)/eclair-mr1/res \
15. $(LOCAL\_PATH)/full-support/res \
16. frameworks/support/v7/appcompat/res \
17. frameworks/support/v7/recyclerview/res
18. LOCAL\_SDK\_VERSION := current
19. LOCAL\_SRC\_FILES := $(call all-java-files-under, main/src eclair-mr1/src full-support/src)
20. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
21. android-support-v4 \
22. android-support-v7-appcompat \
23. android-support-v7-recyclerview
25. include $(BUILD\_STATIC\_JAVA\_LIBRARY)

综上：setup-wizard-lib-full-support的AndroidManifest 是main/AndroidManifest.xml，源码目录为 main/src eclair-mr1/src full-support/src，资源文件为main/res /eclair-mr1/res full-support/res

，依赖jar为android-support-v4/android-support-v7-appcompat/android-support-v7-recyclerview

#### 小结

common-full-support依赖setup-wizard-lib-full-support

### 依赖关系图



## 导入对应的jar和源码

### 关于support.jar

Fg

*/\* implementation files('framework-libs/android-support-design.jar')  
 implementation files('framework-libs/android-support-v4.jar')  
 implementation files('framework-libs/android-support-v7-appcompat.jar')  
 implementation files('framework-libs/android-support-v7-recyclerview.jar')  
 implementation files('framework-libs/android-support-v13.jar')\*/  
  
 //for we need res ,must* compile **'com.android.support:design:25.4.0'** compile **'com.android.support:support-v4:25.4.0'** compile **'com.android.support:appcompat-v7:25.4.0'** compile **'com.android.support:recyclerview-v7:25.4.0'** compile **'com.android.support:support-v13:25.4.0'**

### **导入SettingsLib源码**

源码位置：frameworks/base/packages/SettingsLib/

然后设置jdk1.7

根据import报错,

1. framework方法缺失：需要framework.jar,通过Use Libraries方式导入framework.jar(同时需要配置优先使用这个库， （framework.jar通过系统make模块编译方式得到）。

2.根据报错libcore/icu：找到对应源码,导入core-libart-classes.jar （也是通过系统make模块编译方式得到），Gradle配置

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. provided files('../framework\_libs/core-libart-classes.jar')

### 导入setupwizard/navigationbar源码

源码路径: /frameworks/opt/setupwizard/navigationbar

并设置成Is Library

### 导入setupwizard/library源码

源码路径: frameworks/opt/setupwizard/library

### **Settings**

#### **导入framework.jar**

同上

#### **导入静态库**

android-support直接在AS下用远程仓库获得，注意版本号和系统保持一致

1. LOCAL\_STATIC\_JAVA\_LIBRARIES := \
2. android-support-v4 \
3. android-support-v13 \
4. android-support-v7-recyclerview \
5. android-support-v7-preference \
6. android-support-v7-appcompat \
7. android-support-v14-preference \
8. jsr305 \

jsr305.jar通过系统make得到，

settings-logtags

#### **导入动态库**

一下库都需要通过系统编译得到，源码目录不做展开了

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/79765186) [copy](https://blog.csdn.net/qq1028850792/article/details/79765186)

1. LOCAL\_JAVA\_LIBRARIES := bouncycastle core-oj telephony-common ims-common

telephony-common.jar主要是基带相关的包，导入包含了hide的jar，否则会如下报错：

错误: 找不到符号符号: 变量 BEARER\_BITMASK位置: 类 Carriers

错误: 找不到符号符号: 变量 EXTRA\_KEY\_ALERT\_TITLE位置: 接口 Phone

## 编译

这里描述AS编译过程产生的问题

### AS下的特殊配置

#### Found item String more than one time

#### gesture\_fingerprint\_swipe是空文件异常

这个是手势定义，都是空文件，另外还有[gesture\_ambient\_lift](https://android.googlesource.com/platform/packages/apps/Settings/+/12e4beb%5E%21/)等

在make环境ok，但是as不行。

解决办法：重名为.png

#### 如何导入多个framework下jar包

比如要添加framework.jar 和core.jar。在项目的build.gradle 最后添加上面这段代码，options.compilerArgs.add （），在括号中填写真实的路径，有多个jar包就用分号隔开。

**[html]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. allprojects {
2. repositories {
3. jcenter()
4. }
5. gradle.projectsEvaluated {
6. tasks.withType(JavaCompile) {
7. options.compilerArgs **<<** "-Xlint:unchecked" **<<** "-Xlint:deprecation"
8. options.compilerArgs.add('-Xbootclasspath/p:framework\_libs/telephony-common.jar;framework\_libs/core.jar')
9. }
10. }
11. }

#### android自带的属性规范

<item name="@\*android: not found

<https://www.niwoxuexi.com/blog/android/article/310.html>

因为已经不支持@开头使用android自带的属性，我们只要把@符号删掉就可以了

<https://stackoverflow.com/questions/47033594/android-studio-3-0-error-style-attribute-androidattr-windowenteranimation-n>

'@\*android:attr/regularColor' not found 同理

在color.xml把@\*改为@可以干活？systemui

### Jdk7格式

Android7采用的java1.8，AS肯定是可以用1.7的，这个我们需要改过来

### buildToolsVersion 异常

报错：Execution failed for task ':app:transformClassesWithDesugarForDebug'

解决办法：Open your build.gradle and use below

android {

compileSdkVersion 26

buildToolsVersion "26.0.1"

Then **Clean-Rebuild-Run** [.MORE](https://stackoverflow.com/questions/45833937/execution-failed-for-task-apptransformclasseswithdesugarfordebug-error-on-andro?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa)

### DexArchiveMergerException: r8.errors.CompilationError: Program type already present:android.Manifest

clean your project

then rebuild

<https://stackoverflow.com/questions/47079710/dexarchivemergerexception-with-android-studio-3-0>

provided files(**'../framework\_libs/classes-jarjar.jar'**)

**classes-jarjar.jar 和supprot-v14重合了？**

### <item name="@android:preferenceStyle

<**item name="@android:preferenceStyle"**>@style/Preference</**item**>  
<**item name="@android:editTextPreferenceStyle"**>@style/EditTextPreference</**item**>  
<**item name="@android:dropdownPreferenceStyle"**>@style/Preference.DropDown.Material</**item**>  
<**item name="@android:preferenceFragmentStyle"**>@style/PreferenceFragmentStyle</**item**>

### EventLogTags找不到

Make 编译这个class吧

## 安装

### PARSE\_FAILED\_MANIFEST\_MALFORMED

可以编译出apk了，但是此时无法安装

C:\Users\key.guan\Desktop>adb install -r -d F:\K\1-src\3399\test\dpad-settings\build\outputs\apk\debug\dpad-settings-debug.apk

adb: failed to install F:\K\1-src\3399\test\dpad-settings\build\outputs\apk\debug\dpad-settings-debug.apk: Failure [INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED: Fai

led parse during installPackageLI: /data/app/vmdl360143393.tmp/base.apk (at Binary XML file line #153): <meta-data> requires an android:value or android:resource attribute]

Failure [INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED: Failed parse during installPackageLI: /data/app/vmdl566395868.tmp/base.apk (at Binary XML file line #3421): <meta-data> requires an android:value or android:resource attribute]

INSTALL\_PARSE\_FAILED\_MANIFEST\_MALFORMED meta-data android:resource

|  |  |  |
| --- | --- | --- |
|  | 3437 |  |
|  |  |  |
| 1000 | 3434 |  |
| 1500 | 3432 |  |
| 2000 | 3433 |  |
| 2500 | 3431 |  |
| 2700 | 3434 |  |
| 2800 | 3432 |  |
|  |  |  |
| 2850 | 3437 |  |
| 2900 | 3437 |  |
| 2918 | 3437 |  |
| 3005 | 3437 |  |
|  |  |  |

### signatures do not match

问题：Failure [INSTALL\_FAILED\_UPDATE\_INCOMPATIBLE: Package com.android.settings signatures do not match the previously installed version; ignoring!]

Adb Failure [INSTALL\_FAILED\_TEST\_ONLY

分析：没有系统签名！

解决：系统签名

### base.apk code is missing

问题：Failure [INSTALL\_FAILED\_INVALID\_APK: Package couldn't be installed in /data/app/com.android.settings-1: Package /data/app/com.android.settings-1/base.apk code is missing]

分析：AS的编译机制，系统可能PMS对于apk的解析处于一个错误的状态

解决：$ adb shell pm uninstall com.android.settings

DELETE\_FAILED\_INTERNAL\_ERROR

Error while Installing APK

重启系统就好！

## 运行

### DrawerLayout MeasureSpec.EXACTLY

[drawerlayout必须给一个确定的大小不能用wrap\_content](https://blog.csdn.net/mikogodzd/article/details/50164669)

触发方法： 系统设置->关于手机

#### 问题分析

发生在DeviceInfoSettingsActivity，其他Activity不会发生该问题原因在于风格**，但是为何mm方式下不会崩溃？**

**android:layout\_width="match\_parent"  
android:layout\_height="match\_parent"**

1. 设死长宽，修改为1920\*1080可以规避，不会报错，但是异常
2. DeviceInfoSettingsActivity成弹窗风格，**android:theme="@style/Theme.SubSettingsDialogWhenLarge"**

在使用ToolBar+DrawerLayout的时候，总是报如下的错误：

**android:layout\_width="match\_parent"  
android:layout\_height="match\_parent"**

**https://blog.csdn.net/qq\_38859786/article/details/72852950**

#### 问题解决

直接删除修改风格

，但是此时

### SeekBar$OnSeekBarChangeListener)' on a null object reference

为何原生的没有？NPE

### SecurityException: Package com.android.settings is currently frozen!

adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

显示异常。

### Must specify preferenceTheme in theme

### @android:attr/preferenceFragmentStyle' not found

getActivity().getTheme().resolveAttribute(  
 android.support.v7.preference.R.attr.***preferenceTheme***, tv, **true**);

## 系统应用基本架构

### kpad定制架构

dpad-core

通信协议

## 待优化

### 目前远程编译很慢

方案1：分析as慢的原因，从源头解决问题

方案2：Git可以拉取到本地D盘么

配置ssh

然后push git review 框架

方案3：制作一个win脚本。每次拷贝文件pull/push脚本吧，

方案4：登陆远程unbunttu界面，安装android stadio？

保留:m目前的样子 编译mk\_settings 脚本

### [Android Studio under Network mound -](https://stackoverflow.com/questions/30151183/android-studio-under-network-mound)

https://stackoverflow.com/questions/30151183/android-studio-under-network-mound

## REF

[使用Android Studio开发系统级应用](https://www.jianshu.com/p/3a4b5f1c09c1)

[**Android.mk文件学习笔记**](https://blog.csdn.net/lzh398651363/article/details/73527245)

[**Android系统源码Settings导入eclipse**](https://blog.csdn.net/qq_25804863/article/details/48669477)

**[Android Studio 导入系统 jar包](https://www.cnblogs.com/bluestorm/p/6744140.html)**

## TASK

[Android AAPT详解](https://www.jianshu.com/p/8d691b6bf8b4)

<https://www.jianshu.com/p/85c8ce13fcad?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation>

android6.0 ：<https://gitee.com/lkuan/HelloAndroid/tree/master/AOSPLib/Android-6.0.0_r1/JAVA_LIBRARIES>

# 系统应用AS环境配置

## PackageInstaller

LOCAL\_STATIC\_JAVA\_LIBRARIES

android-support-v4 \  
android-support-v7-recyclerview \  
android-support-v7-preference \  
android-support-v7-appcompat \  
android-support-v14-preference \  
android-support-v17-preference-leanback \  
android-support-v17-leanback \  
xz-java

# SystemUI AS环境搭建



## 模块分析

LOCAL\_PATH:= $(call my-dir)  
  
include $(CLEAR\_VARS)  
  
LOCAL\_MODULE := SystemUI-proto-tags  
  
LOCAL\_SRC\_FILES := $(call all-proto-files-under,src) \  
 src/com/android/systemui/EventLogTags.logtags  
  
LOCAL\_PROTOC\_OPTIMIZE\_TYPE := nano  
LOCAL\_PROTO\_JAVA\_OUTPUT\_PARAMS := optional\_field\_style=accessors  
  
include $(BUILD\_STATIC\_JAVA\_LIBRARY)

$(call all-Iaidl-files-under, src)

librockchip\_pip\_jni

代码安装？

frameworks/support/v7/recyclerview/res \

frameworks/support/v17/leanback/res

## 系统导出

framework-protos \  
SystemUI-proto-tags

Keyguard \

android-support-v14-preference \  
android-support-v17-leanback \

等

## 编译

aka com.android.settings:color/transparent) not found.

**@android:style/Theme.DeviceDefault.Light**

error: resource android:color/quaternary\_device\_default\_settings is private.

导入Keyguaard的res，运行去重复的string

手动删除字符串

JDK配置

compileOptions {  
 sourceCompatibility JavaVersion.***VERSION\_1\_8*** targetCompatibility JavaVersion.***VERSION\_1\_8***}

K[eyguard开始被编译为一个jar包，被SystemUI静态导入](http://www.cnblogs.com/cascle/p/7053688.html)

[Keyguard分析](http://www.cnblogs.com/cascle/p/7053688.html)

[以SystemUI为例，如果需要在SystemUI中引入第三方ja](http://www.voidcn.com/article/p-pcdsvywq-vc.html)

[Android系统源码framework SystemUI导入eclipse编译](https://blog.csdn.net/qq_25804863/article/details/48669667)

AS直接支持MK研究！需要构建环境，更麻烦吧

### 错误: 程序包com.google.protobuf.nano不存在

com.android.framework.protobuf.nano

<https://stackoverflow.com/questions/22247951/android-protobuf-nano-documentation>

### 错误: 程序包IRecentsSystemUserCallbacks不存在

sourceSets {  
 main {  
 aidl.srcDirs = [**'SystemUI/src'**]  
 res.srcDirs = [**'SystemUI/res'**, **'SystemUI/dji/main/res'**, **'SystemUI/dji/test/res'**,**'Keyguard-res'**,**"src/main/res"**]  
 java.srcDirs = [**'SystemUI/src'**, **'SystemUI/dji/main/java'**,**'src/main/java'**,**'SystemUI/dji/test/java'**,**"src/main/java"**]  
 manifest.srcFile **'SystemUI/AndroidManifest.xml'** }  
}

http://www.cnblogs.com/bluestorm/p/6826622.html

### Attr/frameColo

### r more than one time

## 运行

### AutoReinflateContainer

android.view.InflateException: Binary XML file line #72: Binary XML file line #42: Error inflating class com.android.systemui.AutoReinflateContainer

Caused by: android.view.InflateException: Binary XML file line #42: Error inflating class com.android.systemui.AutoReinflateContainer

Toolbar.setNavigationIcon on a null object

### android.content.res.Resources$NotFoundExceptio

# DocumentsUI

资源管理器

## 配置

dependencies {  
 compile **'com.google.guava:guava:23.0-android'**

## 编译

**tools:replace="android:label"**

error: style attribute 'attr/actionBarWidgetTheme (aka com.android.DocumentsUI:attr/actionBarWidgetTheme)' not found.

Message{kind=ERROR, text=error: style attribute 'attr/actionBarWidgetTheme

## 安装

### [INSTALL\_FAILED\_CONFLICTING\_PROVIDER

adb shell pm install -t -r "/data/local/tmp/com.android.DocumentsUI"

Failure [INSTALL\_FAILED\_CONFLICTING\_PROVIDER: Package couldn't be installed in /data/app/com.android.DocumentsUI-1: Can't install because provider name com.android.documentsui.recents (in package com.android.DocumentsUI) is already used by com.android.documentsui]

### Failure [INSTALL\_FAILED\_TEST\_ONLY:

# Gallery2

## 导入库

### Guva

https://github.com/google/guava/wiki/Release23

### Android 视频编辑解析库MP4Parser

使用此库可以实现多个音频视频的追加合并，将视频的音频和视频进行分离替换等，同时还支持了字幕的添加，使用时候需要注意音视频格式。主要功能有：

* 将音频视频混合到一个mp4文件中
* 将编码格式相同的录音进行拼接合并: Different resolutions和Different frame-rates是不能进行拼接合并操作的。
* 添加或者修改元数据
* 通过省略帧缩短录音

[实例参考](https://www.jianshu.com/p/c87ada9b0f65)，仓库版本列表[**mvnrepository**](https://mvnrepository.com/artifact/com.googlecode.mp4parser/isoparser)

系统源码提示 参考：[GIT](https://github.com/sannies/mp4parser/issues/277)，但是已经找不到对应的版本了。

导入方案1【失败】：远程依赖，**1.1.22，1.0.6，***1.0-RC-27*

*// compile 'com.googlecode.mp4parser:isoparser:1.1.21'* compile **'org.mp4parser:streaming:1.9.27'** compile **'org.mp4parser:muxer:1.9.27'** compile **'org.mp4parser:isoparser:1.9.27'**

导入方案2【pass】：只好源码依赖之！！！！！！！！！！

[**Android 视频编辑解析库MP4Parser**](https://www.jianshu.com/p/c87ada9b0f65)

[**mvnrepository**](https://mvnrepository.com/artifact/com.googlecode.mp4parser/isoparser)

[Android(java方法）上实现mp4的分割和拼接](http://www.cnblogs.com/android-blogs/p/5711171.html)

#### E/BitmapUtils: createVideoThumbnail

java.lang.reflect.InvocationTargetException

at java.lang.reflect.Method.invoke(Native Method)

at com.android.gallery3d.common.BitmapUtils.createVideoThumbnail(BitmapUtils.java:199)

### XMPCore

配置源码路径就好了

### RenderScript

Renderscript是Android操作系统上的一套API。它基于异构计算思想（指使用不同类型指令集和体系架构的计算单元组成系统的计算方式），专门用于密集型计算，尤其是图像处理、计算机图形、计算机视觉。允许开发者以较少的代码实现功能复杂且性能优越的应用程序。

RenderScript可在设备上所有可用的处理器上并行执行，例如多核CPU、GPU、DSP，所以开发者可以专心写处理算法，而不需要关心调度和负载平衡的问题。

本文通过写一个简单的例子（[源码链接](https://link.jianshu.com/?t=https://github.com/kevin-nazgul/AndroidMisc)）来解释怎么使用RenderScript，例子中将彩色的图片转换为黑白的，为了测试RenderScript的性能，加上手指滑动的效果，如下图：



#### 工程配置

RenderScript的api从Android 3.0（Api level 11）开始系统自带了，在低于3.0的机器上，可以通过support包来使用RenderScript。但是还是推荐都使用support的RenderScript，因为RenderScript可能会有bug，使用support包能够及时得到更新。本文均使用support包中的Api。

要使用RenderScript，除了要导入一个jar包外，还需要复制一些so文件，如果使用gradle的话，就方便一些，只需要在build.gradle文件中增加两句：

defaultConfig {

minSdkVersion 8

targetSdkVersion 16

**renderscriptTargetApi 18**

**renderscriptSupportModeEnabled true** }

jh  **renderscriptTargetApi**：设置生成的字节码的版本，推荐使用最高的API Level，并且设置renderscriptSupportModeEnabled为true。这个选项的合法值从11到最新发布的API Level。如果你的最低SDK的版本和这个值不一样，这个值就会被忽略，在编译的时候，这个值会被设置了最低的SDK版本。

 **renderscriptSupportModeEnabled**：指定在运行的设备不是target version的时候，生成的字节码需要回滚到兼容的版本

#### 脚本语言

为了能在多个平台执行，RenderScript要求开发者使用Rs的脚本语言来实现计算的代码，脚本代码采用了c99语法，所以看起来和C语言很像。代码要求放在.rs文件中，文件需要放在 <project\_root>/src/ 中，脚本包含脚本的入口、函数和变量：

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. #pragma version(1)
2. #pragma rs java\_package\_name(com.winomtech.androidmisc.rs)
4. **void** root(**const** uchar4 \*in, uchar4 \*out, uint32\_t x, uint32\_t y) {
5. out->r = out->g = out->b = (in->r + in->g + in->b) / 3;
6. }
8. **void** init() {
9. }

#pragma version(1) 声明脚本中使用的Rs的版本。#pragma rs java\_package\_name 声明脚本生成的Java层的代码所在的包名

*root*（Kernel）函数是rs脚本默认的入口，这种入口函数在rs中称为Kernel函数，默认的Kernel函数名为*root*，且必须返回*void*，并且有以下的参数：

* 指向rs脚本输入输出的内存的指针。在Android 3.2(Api Level 13)或者更早的版本，这两个参数都需要，Android 4.0(Api Level 14)及以后的版本要求一个或两个。

下面的参数是可选的，但是如果你选择使用它们，则都需要提供：

* rs脚本用来实现计算的附加数据的指针，可以是原始类型的指针，也可以是复杂的结构体的指针。
* 附加数据的大小

从Android 4.1 (Api level 16)开始，可以自己定义kernel函数的参数，而不必和前面描述参数及返回值一样。并且可以在同一个脚本中定义多个kernel函数，但是需要在自定义的kernel函数前面增加 \_\_attribute\_\_((kernel))。例如，下面是一个kernel函数，接收两个 uint32\_t 类型的参数，返回一个uchar4：

**[cpp]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. uchar4 \_\_attribute\_\_((kernel)) myFun(uint32\_t x, uint32\_t y) {
2. ...
3. }

参数类型uchar4是rs中一类数据类型，称为Vector。Vector通常是普通类型接着2、3、4，比如：float4, int3, double2, ulong4。Vector类型中的成员可以使用多种风格来访问：变量名接着一个点，再接着：

* 字符x、y、z和w
* 字符r、g、b和a
* 字符s或S，随后接着从0开始的下标

uint32\_t x、uint32\_t y两个参数是当前执行到的下标，这两个参数是可选的，最多可以有三维x、y、z，类型必须为uint32\_t。

init函数是一个可选的函数，在kernel函数执行前，init函数会被执行一次，可以在init里面做一些初始化的工作，比如初始化变量。

#### 脚本编译

脚本在执行之前会有两次编译：在android编译的过程中，会发生第一次编译，生成LLVM的字节码；当应用在设备中执行的时候，会发生第二次编译，生成设备的机器码。

#### Java层的接口

因为脚本最后会被编译为机器码，为了方便Java层设置数据和调用接口，rs在android编译的时候，会生成一个java文件，名字为ScriptC\_script\_name，比如Gray.rs文件，会自动生成一个ScriptC\_Gary.java。下面代码是上面rs脚本生成的内容：

生成的代码继承于ScriptC，其中有一些变量和函数名都有\_root的后缀，这个后缀的root其实就是kernel函数的名字，如果一个rs脚本中有多个kernel函数，比如有一个名为test的kernel函数，那么就有一些函数和变量名有\_test后缀。

getKernelID\_root这个是返回root的id，这个id是RenderScript框架所使用的。有时候需要在一个Kernel函数执行完后，再执行另外一个时，就需要使用到它。

forEach\_root函数是用来调用Kernel函数执行的入口，在执行之前，会检查输入和输出是否和脚本中的类型一致，如果不一致直接抛出异常，同时还会检查输入和输出的维度是否一致。

#### 执行

下面代码是缩减后的代码，仅作讲解使用，代码的主要作用是用输入的图片生成一张黑白的图片，并且显示在界面上：

Bitmap mInBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.renderscript\_input); Bitmap mOutBitmap = Bitmap.createBitmap(mInBitmap.getWidth(), mInBitmap.getHeight(), Bitmap.Config.ARGB\_8888); RenderScript mRenderScript = RenderScript.create(getActivity()); ScriptC\_Gray mGrayScript = new ScriptC\_Gray(mRenderScript); Allocation mAllocationIn = Allocation.createFromBitmap(mRenderScript, mInBitmap); Allocation mAllocationOut = Allocation.createFromBitmap(mRenderScript, mOutBitmap); mGrayScript.forEach\_root(mAllocationIn, mAllocationOut); mAllocationOut.copyTo(mOutBitmap); mImageView.setImageBitmap(mOutBitmap);

##### RenderScript

Fg

这个类用来访问RenderScript的上下文，管理RenderScript的初始化、资源、以及生命周期的。在任何其他RS对象使用之前，需要创建一个RenderScript对象，并且需要缓存这个对象，以方便后续使用。当进程完成使用RenderScript的任务的时候，需要调用releaseAllContexts。

通过调用RenderScript.create静态方式，可以得到RenderScript的对象。在Api 23的时候，多次调用create得到的会是同样的对象，所以能够得到同样的上下文，以及同样的配置。在Api 23之前，每次调用create将生成一个新的上下文。

##### ScriptC\_Gray

##### Allocation

fgs应用程序在Android虚拟机上使用RenderScript，但是实际的rs脚本在native层执行，并且需要访问Android虚拟机分配的内存。为此，就需要将虚拟机层分配的内存和RenderScript runtime连接起来，这个过程称为绑定，然后RenderScript runtime就可以方便的使用它需要的内存了，而不需要显式的分配。最终的结果就好像你在C中调用了malloc一样。

为了支持这种内存分配的方式，提供了一组API给Android VM来分配内存，类似于malloc函数的功能。这些类本质是描述应该怎么分配内存以及执行真正的分配。为了更好的理解这些类是怎么工作的，我们看看它们与malloc的关系：

array = (int \*)malloc(sizeof(int)\*10);

这个malloc调用可以分为两个部分：分配的内存单元的大小（sizeof(int)），分配的单元的个数。Android框架提供了类似于这两部分的类，以及类似malloc的类。

Element类代表malloc中的分配单元，例如浮点类型或者一个结构体。Type类对Element和分配的个数进行包装，可以将Type想象成Element的数组。而Allocation类基于给定的Type执行真正的内存分配，以及访问分配的内存。

Type由5个维度组成：X、Y、Z、LOD(level of detail)和Faces(of a cube map)。可以设置X、Y、Z为任意正整数，但是要限制在可用内存的范围内。如果某个维度的值为0，表示没有那个维度，例如：x=10、y=1、z=0表示二维。LOD和Faces用boolean值来表示是否出现了。

Allocation.createFromBitmap是对分配操作的封装，会根据Bitmap的配置来指定对于的Element的值，以及Type的值，并且用Bitmap的宽高来指定x、y的值。

##### forEach\_root

forEach\_root中两个参数分别指定输入和输出，RenderScript会根据输入参数中的维度，遍历整个数组，对每个数值都调用一次Kernel函数，所以rs文件中的in、out、x、y是根据当前遍历的位置的不同而不同。

##### 变量

在前面的rs文件中，只指定了输入和输出，当需要外部传入多个参数时，就需要使用rs的变量了。本文的例子中，除了要指定输入输出，还需要指定从哪个位置开始是需要变灰的，所以我们在rs脚本中增加一个变量：

#pragma version(1)

#pragma rs java\_package\_name(com.winomtech.androidmisc.rs)

int gPos;

...

再编译下android工程，我们再看生成的java代码，会多出来两个方法：

public synchronized void set\_gPos(int v) {

setVar(mExportVarIdx\_gPos, v);

mExportVar\_gPos = v;

}

public int get\_gPos() {

return mExportVar\_gPos;

}

java层就可以通过这两个接口来访问rs中的gPos变量了

#### 总结

RenderScript用起来还算挺简单的，当然也主要是本文实现功能简单。当然RenderScript的作用还是很容易看出来的，可以写一个长循环，然后看cpu使用，在我的手机上面，4个核心可以跑满，而如果使用jni的话，很明显只会在一个核心上面跑。

RenderScript :简单而快速的图像处理，不用复杂的OpenCL就能利用好你手机的GPU计算能力，核心在于写到rs脚本文件

Go

resources.srcDirs = ['gen']

renderscript.srcDirs = ['src']

[RenderScript的基础使用](https://www.jianshu.com/p/b7142cb70b04)

[RenderScript 代码实例](https://www.helplib.com/Java_API_Classes/article_59409)

[How to Use the Renderscript Support Library with Gradle](https://futurestud.io/tutorials/how-to-use-the-renderscript-support-library-with-gradle-based-android-projects)

#### make编译

##### make配置

LOCAL\_SRC\_FILES := \

$(call all-renderscript-files-under, src)

##### 输入



##### 输出

out/target/common/obj/APPS/Gallery2\_intermediates/src/renderscript/



#### AS配置

AS工程依赖



Rs依赖？貌似搞不定呢

java依赖

java.srcDirs = [**'Gallery2/src'**, ,**"src/mp4parser/isoparser/src/main/java"**]

远程仓库依赖？

### so共享库

/system/lib64下自带了

libjni\_eglfence libjni\_filtershow\_filters libjni\_jpegstream

LOCAL\_JNI\_SHARED\_LIBRARIES := libjni\_eglfence libjni\_filtershow\_filters libjni\_jpegstream

## gradle编译

### Error: Found item String/no\_storage more than one time

运行脚本：fixGallery2String.py

### filtershow\_state\_button\_background: Error: The file name must end with .xml or .png

命名为.png

### unknown element <permission> found.,

fg

<**permission android:name="com.android.gallery3d.filtershow.permission.WRITE"  
 android:protectionLevel="signature"** />

移动位置

### attribute 'com.android.gallery3d:max\_width' not found.

error: attribute 'com.android.gallery3d:max\_width' not found.

Message{kind=ERROR, text=error: attribute 'com.android.gallery3d:max\_width' not found., sources=[F:\K\1-src\3399\dpad-gallery2\Gallery2\res\layout\filtershow\_activity.xml:75], original message=, tool name=Optional.of(AAPT)}

applicationId **'com.android.gallery3d'**

## 模块覆盖

LOCAL\_OVERRIDES\_PACKAGES := Gallery Gallery3D GalleryNew3D

但是系统的Gallery Gallery3D GalleryNew3D貌似都没有，是为了兼容旧版本吧？？？

## 源码编译

### org.apache.http.legacy

<https://blog.csdn.net/zxw136511485/article/details/53185485>

原来是Android 6.0不再支持 Apache HTTP client， 建议使用 HttpURLConnection 代替。如果还想要继续使用 Apache HTTP client 的，请在build.gradle中添加下面的代码

1. android {
2. useLibrary 'org.apache.http.legacy'
3. }

# bluetooth

LOCAL\_JNI\_SHARED\_LIBRARIES := libbluetooth\_jni

改为源码依赖就好了

LOCAL\_JAVA\_LIBRARIES := javax.obex telephony-common libprotobuf-java-micro services.net

javax.obex：framework/base下源码依赖

telephony-common jar依赖

libprotobuf-java-micro 不处理不行啊，必须处理

services.net 源码

LOCAL\_STATIC\_JAVA\_LIBRARIES := com.android.vcard bluetooth.mapsapi sap-api-java-static android-support-v4 services.net

com.android.vcard：源码

bluetooth.mapsapi：源码

sap-api-java-static：

hardware/ril???不处理

android-support-v4：online

# 常用脚本

## 去除验证限制

adb root

adb disable-verity

adb reboot

## 重启zgost

adb root

adb remount

adb shell stop

adb shell start

## 恢复出厂设置

恢复出厂设置，删除虚拟机资源文件缓存

需要root权限

App层: sendBroadcast(new Intent("Android.intent.action.MASTER\_CLEAR"));

Adb层:

adb root

adb shell am broadcast -a android.intent.action.MASTER\_CLEAR

echo --wipe\_all > /cache/recovery/command

reboot recovery

# 编译优化

新加入一个app，mmm或者全包编译都会包异常。Rm –rf out 再编译一次就好

With\_

## 无dex文件

在android7.0平台，编译了新模块，直接adb install无法安装，是不会通过的！

问题分析：默认开启了编译优化，导致每次编译的apk，odex文件都被抽取出来了，解压apk目录如下：无dex代码，只有资源文件。



然而标准的APK的目录应该是这样的



**解决办法1：**

需要都更新对应apk和odex文件

需要push system/framework/oat/arm64/xx.odex那么system/framework/xx.jar有必要再推送么？

但是Settings始终没有验证成功为何？

adb disable-verity

**[python]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. adb root
2. adb remount
3. adb uninstall com.android.settings
4. adb push Settings.apk /system/priv-app/Settings/
5. adb push Settings.odex /system/priv-app/Settings/oat/arm64/
6. adb shell sync
7. adb shell am force-stop com.android.settings
8. adb shell am start com.android.settings
9. pause;

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. 06-24 20:11:48.327 8667-8667/? E/AndroidRuntime: FATAL EXCEPTION: main
2. Process: com.android.settings, PID: 8667
3. java.lang.RuntimeException: Unable to instantiate application com.android.settings.SettingsApplication: java.lang.ClassNotFoundException: Didn't find **class** "com.android.settings.SettingsApplication" on path: DexPathList[[zip file "/system/priv-app/Settings/Settings.apk"],nativeLibraryDirectories=[/system/priv-app/Settings/lib/arm64, /system/priv-app/Settings/Settings.apk!/lib/arm64-v8a, /system/lib64, /vendor/lib64, /system/lib64, /vendor/lib64]]
4. at android.app.LoadedApk.makeApplication(LoadedApk.java:802)
5. at android.app.ActivityThread.handleBindApplication(ActivityThread.java:5496)
6. at android.app.ActivityThread.-wrap2(ActivityThread.java)
7. at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1580)
8. at android.os.Handler.dispatchMessage(Handler.java:102)
9. at android.os.Looper.loop(Looper.java:154)
10. at android.app.ActivityThread.main(ActivityThread.java:6238)
11. at java.lang.reflect.Method.invoke(Native Method)
12. at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:929)
13. at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:819)
14. Caused by: java.lang.ClassNotFoundException: Didn't find **class** "com.android.settings.SettingsApplication" on path: DexPathList[[zip file "/system/priv-app/Settings/Settings.apk"],nativeLibraryDirectories=[/system/priv-app/Settings/lib/arm64, /system/priv-app/Settings/Settings.apk!/lib/arm64-v8a, /system/lib64, /vendor/lib64, /system/lib64, /vendor/lib64]]
15. at dalvik.system.BaseDexClassLoader.findClass(BaseDexClassLoader.java:56)
16. at java.lang.ClassLoader.loadClass(ClassLoader.java:380)
17. at java.lang.ClassLoader.loadClass(ClassLoader.java:312)
18. at android.app.Instrumentation.newApplication(Instrumentation.java:992)
19. at android.app.LoadedApk.makeApplication(LoadedApk.java:796)
20. at android.app.ActivityThread.handleBindApplication(ActivityThread.java:5496)
21. at android.app.ActivityThread.-wrap2(ActivityThread.java)
22. at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1580)
23. at android.os.Handler.dispatchMessage(Handler.java:102)
24. at android.os.Looper.loop(Looper.java:154)
25. at android.app.ActivityThread.main(ActivityThread.java:6238)
26. at java.lang.reflect.Method.invoke(Native Method)
27. at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:929)
28. at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:819)
29. Suppressed: java.io.IOException: No original dex files found **for** dex location /system/priv-app/Settings/Settings.apk
30. at dalvik.system.DexFile.openDexFileNative(Native Method)
31. at dalvik.system.DexFile.openDexFile(DexFile.java:367)
32. at dalvik.system.DexFile.<init>(DexFile.java:112)
33. at dalvik.system.DexFile.<init>(DexFile.java:77)
34. at dalvik.system.DexPathList.loadDexFile(DexPathList.java:359)
35. at dalvik.system.DexPathList.makeElements(DexPathList.java:323)
36. at dalvik.system.DexPathList.makeDexElements(DexPathList.java:263)
37. at dalvik.system.DexPathList.<init>(DexPathList.java:126)
38. at dalvik.system.BaseDexClassLoader.<init>(BaseDexClassLoader.java:48)
39. at dalvik.system.PathClassLoader.<init>(PathClassLoader.java:64)
40. at com.android.internal.os.PathClassLoaderFactory.createClassLoader(PathClassLoaderFactory.java:43)
41. at android.app.ApplicationLoaders.getClassLoader(ApplicationLoaders.java:58)
42. at android.app.LoadedApk.createOrUpdateClassLoaderLocked(LoadedApk.java:520)
43. at android.app.LoadedApk.getClassLoader(LoadedApk.java:553)
44. at android.app.ActivityThread.getTopLevelResources(ActivityThread.java:1901)
45. at android.app.LoadedApk.getResources(LoadedApk.java:766)
46. at android.app.ContextImpl.<init>(ContextImpl.java:2039)
47. at android.app.ContextImpl.createAppContext(ContextImpl.java:1984)
48. at android.app.ActivityThread.handleBindApplication(ActivityThread.java:5413)
49. ... 8 more

**解决办法2：**

关闭编译优化：/device/rockship/kpad/BoardConfig.mk

WITH\_DEXPREOPT := false



# Build-Shell

## 代码搜索grep

| **搜索指令** | **解释** |  |
| --- | --- | --- |
| cgrep | 所有**C/C++**文件执行搜索操作 |  |
| jgrep | 所有**Java**文件执行搜索操作 | 搜索所有Java代码中包含zygote所在文件  jgrep zygote |
| ggrep | 所有**Gradle**文件执行搜索操作 |  |
| mangrep [keyword] | 所有**AndroidManifest.xml**文件执行搜索操作 | 搜索所有AndroidManifest.xml文件中的launcher关键字所在文件的具体位置，指令  mangrep launcher |
| mgrep [keyword] | 所有**Android.mk**文件执行搜索操作 |  |
| sepgrep [keyword] | 所有**sepolicy**文件执行搜索操作 | 搜索所有system\_app的selinux权限信息  sepgrep system\_app |
| resgrep [keyword] | 所有本地res/\*.xml文件执行搜索操作 |  |
| sgrep [keyword] | 所有source源文件执行搜索操作 |  |

上述指令用法最终实现方式都是基于grep指令，各个指令用法格式：

xgrep [keyword] //x代表的是上表的搜索指令

**Tips:** Android源码非常庞大，直接采用grep来搜索代码，不仅方法笨拙、浪费时间，而且搜索出很多无意义的混淆结果。根据具体需求，来选择合适的代码搜索指令，能节省代码搜索时间，提高搜索结果的精准度，方便定位目标代码。

## 导航指令

| **导航指令** | **解释** |  |
| --- | --- | --- |
| croot | 切换至Android根目录 | 当进入源码层级很深后，需要返回到根目录，使用croot一条指令完成  另外cd - 指令可用于快速切换至上次目录 |
| cproj | 切换至工程的根目录 | 当每次修改完某个文件后需要编译时，执行cproj后会跳转到当前模块的根目录，也就是Android.mk文件所在目录，然后再执行mm指令，即可编译目标模块 |
| godir [filename] | 跳转到包含某个文件的目录 | 经常在不同的模块之间修改常用到的，但是这个并不好用  因为结果太多了。。 |

## 2.4 信息查询

| **查询指令** | **解释** |  |
| --- | --- | --- |
| hmm | 查询所有的指令help信息 |  |
| **findmakefile** | 查询当前目录所在工程的Android.mk文件路径 | 可以在src/com/android向父目录查找 |
| print\_lunch\_menu | 查询lunch可选的product |  |
| **printconfig** | 查询各项编译变量值 | 当不太确认自己在哪个产品上的时候，可以通过这个来查询 |
| gettop | 查询Android源码的根目录 |  |
| gettargetarch | 获取TARGET\_ARCH值 |  |

# 常用工具

## SecureCRT

### 记住上次连接

### 超时自动断开的问题

解决办法：

Options->Session Options->Terminal->Anti-idle->勾选Send protocol NO-OP

(中文版：选项->会话选项->终端->反空闲->发送协议NO-OP)

后面的设置时间默认的是60秒，只要小于自动断开连接的时限就可以了。

### 语法高亮



设置语法高亮，多色显示

Options -> Session Options -> Emulation (Terminal)  
其中Terminal选择 【Xterm】，勾选【ANSI Color】和【Select an alternate keyboard emulation】  
设置完成，重新连接。

# REF

[理解Android编译命令](http://gityuan.com/2016/03/19/android-build/)

[理解 Android Build 系统](https://www.ibm.com/developerworks/cn/opensource/os-cn-android-build/)

# TASK

[android“设置”里的版本号](http://blog.csdn.net/wh_19910525/article/details/8286027)

frameworks/base/api

hide实现/系统api保护原理

# QA

**编译系统支持Gradle么？**