编译系统

1. LOCAL\_DEX\_PREOPT := false

# 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，本文先重点讨论 Make 文件中的内容，然后在说shell脚本。

# 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目标



## 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：

## 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。 |
| **makeframework** | 编译所有 Java framework 内容。 |
| **makeservices** | 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 | 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！

## 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。

## 导出api update-api

## **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

## make系统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

## 常见问题

### 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>

# 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

### LOCAL\_PATH := $(call my-dir)

Dsfhf

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)

g

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

### 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\_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！

LOCAL\_OVERRIDES\_PACKAGES := privison

### 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;TARGET\_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\_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\_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\_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)

  # 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

### 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

## 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

# 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？**