[**Android高版本P/Q/R源码编译指南**](https://www.cnblogs.com/jiusibuiu/p/14235212.html)

Android各个版本的对应关系如下:

* Android 5.x (Lollipop)简称Android L版本
* Android 6.0 (MarshMallow) 简称Android M版本
* Android 7.x (Nougat)简称Android N版本
* Android 8.x (Oreo)简称Android O版本
* Android 9.0 (Pie)简称Android P版本
* Android 10.0 (Q)简称Android Q版本
* Android 11.0 （R)简称Android R版本

并且这里还有一点需要特别注意，本文演示的Android R版本是以高通平台为基准进行的。

**一.Android编译环境的构建以及常见命令**

  俗话说天时地利人和缺一不可，而这其中的地利翻译过来说的就是环境因素了，人的成长离不开环境因素，而我们的Android编译也离不开编译环境的构建！虽然我们本篇博客的主题是Android源码编译指南，但是我们还是有必要抽出一个章节来简单说明下Android编译环境的构建和初始化过程，以及初始化完毕后常见的命令。

**1.1 Android编译环境的构建**

虽然Android的版本一直在迭代着，但是Android编译环境的构建步骤还是比较良心的依然没有多大的变化(注意这里的措辞，只是步骤)，对于有过Android源码开发经验的读者来说是再为熟悉不过的了，通常是如下二部曲:

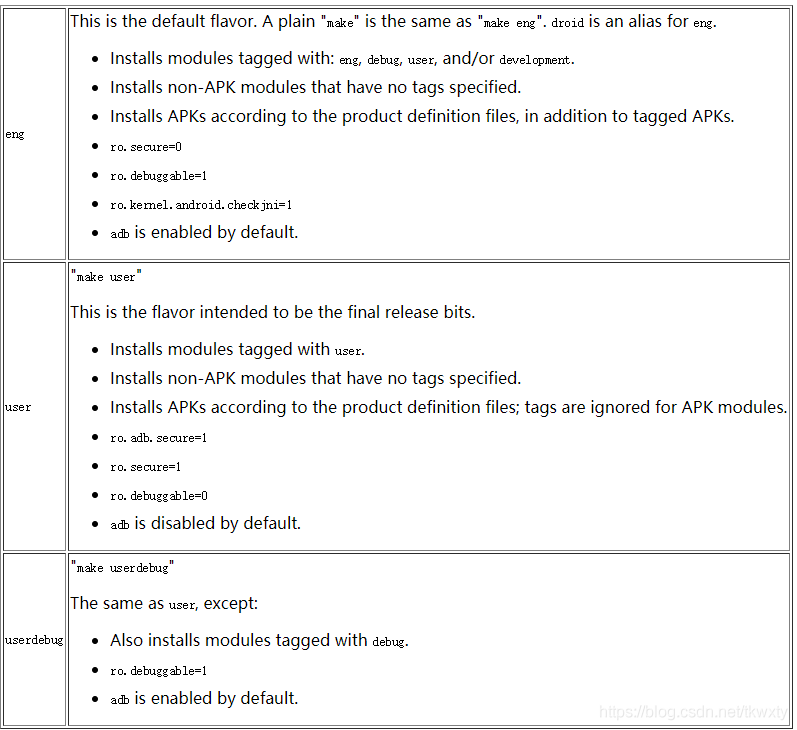
$ source build/envsetup.sh

$ lunch aosp-eng

虽然各位对上述的命令应该已经烂熟于心了，但是这里我还是简单说明一下:

* 第一行命令”source build/envsetup.sh”引入了build/envsetup.sh 脚本。该脚本的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 aosp-eng”是调用 lunch 函数，并指定参数为”aosp-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”aosp”和”eng”。”aosp”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。

这里补充一点对Android的源码编译类型简单说明一下，它可以分为如下三种功能，每种类型的特点如下:



**1.2 Android编译各种常见命令**

在编译环境初始化完成后，我们就可以使用各种各种编译环境提供的指令和make编译命令族来开启Android的构建之旅了，这里我简单的总结了下，我们在Android编译中可能会用到的编译环境提供的指令和make编译命令族，如下:

**1.2.1 常见的Android命令指令**

| **指令** | **说明** |
| --- | --- |
| croot | 切到Android源码树的根目录(当你深入Android源码树的子目录，想回到根目录的时候此命令就非常实用了) |
| m | 相当于在源码树的根目录执行make，并且该命令不一定要在根目录下执行 |
| mm | 编译当前目录路径下的所有模块(包括include进来的，但是不包括存在依赖关系模块) |
| mma | 编译当前目录路径下的所有模块(包括include进来的，且包括存在依赖关系模块) |
| mmm[module\_path] | 编译指定目录路径下的所有模块(包括include进来的，但是不包括存在存在依赖关系模块) |
| mmma[module\_path] | 编译指定目录路径下的所有模块(包括include进来的，包括存在存在依赖关系模块) |
| cgrep | 对C/C++文件执行grep(即grep的时候只搜寻C/C++文件类型，注意这里也包括.h文件类型) |
| jgrep | 对Java文件执行grep(即grep的时候只搜寻Java文件类型) |
| resgrep | 在所有res/*.xml文件上执行 grep即grep的时候只搜寻res/*.xml文件类型) |
| printconfig | 显示当前Android编译的相关配置信息 |
| add\_lunch\_combo | 在lunch命令的的菜单中添加一个条目 |

这里我们对上述表格中的不包括存在依赖关系模块::  
1.依赖关系模块这个要怎么说呢，这里我们举个栗子！譬如模块A的编译需要依赖模块B，此时的B是一个so库。  
2.假如我们通过mm或者mmm编译模块A的时候，此时B模块还没有编译那么此时就会报错  
3.假如我们使用的是mma或者mmma编译模块A，假如依赖的模块B还没有编译，那么会先将模块B编译OK，然后编译模块A(当然这里只是举栗子，可能A还依赖C,D同理也会先编译)

**1.2.2 make编译命令族**

Android的Build编译系统处理常见的make单命令之外，还提供了其它的一系列make命令族，这里我们简单过下:

| **指令** | **说明** |
| --- | --- |
| make update-api | 更新API文件，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 |
| make | Android默认系统编译指令，会编译出整个系统的所有镜像(其实质最终执行的是make droid) |
| make droid | 同上 |
| make sdk | 编译出Android的SDK开发套件 |
| make clean-sdk | 清理SDK的编译产物 |
| make dist | 执行整个编译，并将 MAKECMDGOALS变量定义的输出文件拷贝到 /out/dist目录下， 这个命令在实际中用的比较少 |
| make all | 编译所有内容，不管当前产品的定义中是否会包含,官方解释如下: builds everything make droid does,plus everything whose LOCAL\_MODULE\_TAGS do not include the “droid” tag. The build server runs this to make sure that everything that is in the tree and has an Android.mk builds. |
| make help | 帮助信息命令，显示当前Android版本主要支持的make命令 |
| make snod | 从已经编译出的包快速构建系统镜像(譬如你重新单独编译了某个模块，然后想快速进行打包到system.img，可以使用此命令加快速度) |
| make clean-$(LOCAL\_MODULE) | Let you selectively clean one target. For example, you can type make clean-libutils and t will delete libutils.so and all of the intermediate files. 即清理掉一个指定模块的编译结果和中间产物 |
| make clean-$(LOCAL\_PACKAGE\_NAME) | Let you selectively clean one target. For example, you can type make clean-Home and it will clean just the Home app… 即清理掉一个指定模块的编译结果和中间产物 |
| make clean | deletes all of the output and intermediate files for this configuration. This is the same as rm -rf out/<configuration>/ 通常删除的是整个Android源码工程的out/\*目录 |
| make clobber | deletes all of the output and intermediate files for all configurations. This is the same as rm -rf out/. 这个命令在实际中，应用得比较少 |
| make dataclean | deletes contents of the data directory inside the current combo directory. This is especially useful on the simulator and emulator, where the persistent data remains present between builds. 这个命令在实际中应用得也比价少 |
| make installclean | 当我们在执行切换编译目标时可以执行make installclean，用以清除之前编译生成的文件，但是又不会将整个out目录清空，这样可以加快编译目标的构建速度 |
| make LOCAL\_MODULE | 编译一个指定的模块，LOCAL\_MODULE 为模块的名称，这种编译方法通常运用在整个Android工程没有构建，但是想快速编译一个模块时可以使用，可以加快单个模块构建速度 |
| make targets | will print a list of all of the LOCAL\_MODULE names you can make. |
| make libandroid\_runtime | 编译所有JNI framework内容。 |
| make framework | 编译所有Javaframework内容(做Android framework开发的小伙们对这条命令应该是再熟悉不过的了)。 |
| make services | 编译系统服务和相关内容 |
| make bootimage | 编译生成boot.img |
| make recoveryimage | 编译生成recovey.img |
| make cacheimage | 编译生成cache.img |
| make systemimage | 编译生成system.img |
| make vendorimage | 编译生成vendor.img |
| make superimage | 编译生成superi.img |

对上述的make命令有几点需要注意:  
1.可能在不同的Android版本有不同表现，且有的可能已经不支持了  
2.读者最好对于每个make编译命令，自行使用一番，然后慢慢品尝

**二.Android编译的发展史简介**

  有过一定Android开发经验的读者应该知道Android最初是用Android.mk配置来编译源码的(这里的Android.mk本质上有点类似Makefile文件)。但是随着Android版本的迭代，源码工程文件越来越大，包含的模块越来越多，而以Android.mk组织的项目编译花费的时间越来越多。面对这个严峻的问题，Android的妈咪谷歌终于在在Android7.0开始引入了ninja编译系统。相对于make来说ninja在大的项目管理中速度和并行方面有突出的优势，因此Google采用了ninja来取代之前使用的make。由于Android.mk的数量巨大且复杂，不可能把所有的Android.mk改写成ninja的构建规则，因此Google搞了个kati工具，用于将Androd.mk转换成ninja的构建规则文件build.ninja，再使用ninja来进行构建工作。

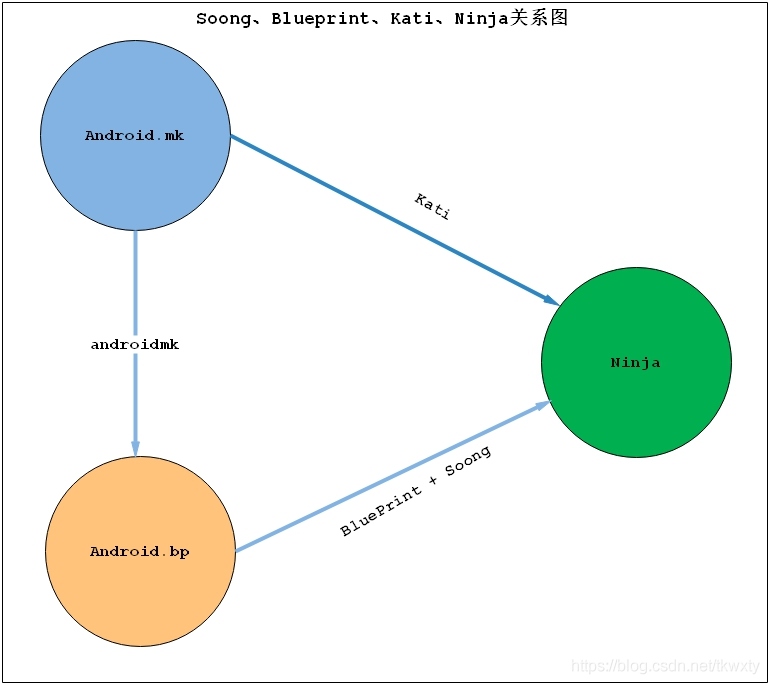
Android编译的发展依然没有停止进化，果不其然Android8.0开始，Google引入了Android.bp文件来替代之前的Android.mk文件，Android.bp只是纯粹的配置文件，不包括分支、循环等流程控制，本质上就是一个json配置文件。同时还引入Soong这个工具，用于将Android.bp转换为ninja的构建规则文件build.ninja，再使用ninja来进行构建工作。但之前的模块全部是用Android.mk来定义的，google不可能一下子把所有模块都修改成Android.bp，只能逐步替换。无论是Android.mk还是Android.bp最后都是转化成ninja的构建规则，再进行编译的。

如果你对上述的概述，还是觉得太麻烦了，这里我们整体来概括一下Android build系统随着Android版本相应的发展演变过程:

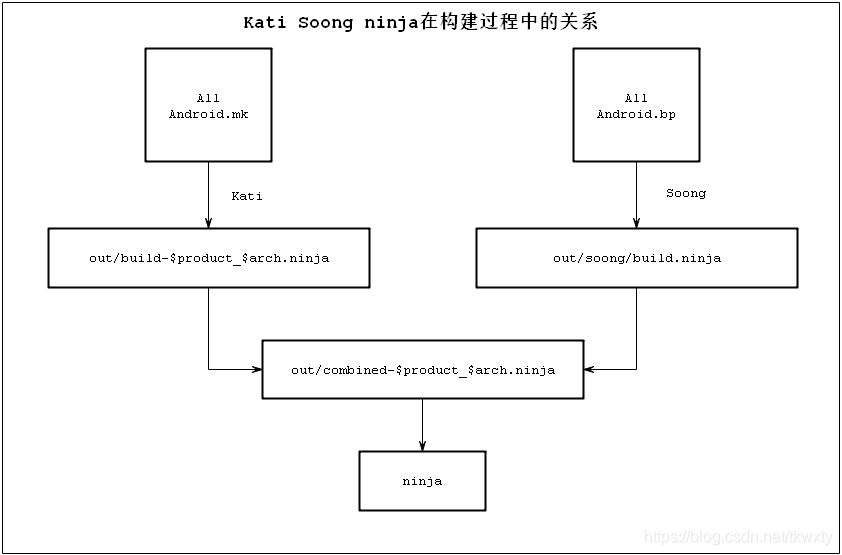
* Android 7.0引入ninja和kati
* Android 8.0使用Android.bp来替换Android.mk，引入Soong
* Android 9.0强制使用Android.bp

**2.1 Soong、Blueprint、Kati、Ninja关系**

前面一顿咔咔，我们简单介绍了Android编译系统的范展示，其中突然一下子冒出了许多的概念，这里我们先暂且不对其中涉及的概念讲述，我们先说说Soong、Blueprint、Kati、Ninja之间的关系，如下:



上图是整体的关系图，同时在Android源码工程构建过程中的转换关系如下:



如果对上述的关系还是没有捯饬清楚的，我们再来说说，说说:

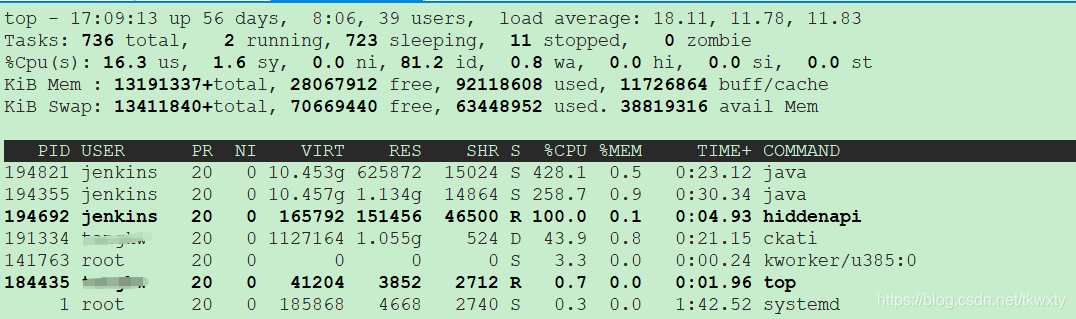
* 首先通过Kati将Android.mk转换成ninja格式的文件
* 通过androidmk将将Android.mk转换成Android.bp，但是只针对没有分支、循环等流程控制的Android.mk才有效，如果对于有控制流的就必须手动了具体可以想见博客[Android.bp正确姿势添加宏控制编译指南](https://blog.csdn.net/tkwxty/article/details/105111218)
* 通过Blueprint+ Soong将Android.bp转换成ninja格式的文件

不容易啊，这里我们对涉及到Ninja, kati, Soong, bp关系搞清楚了(各种三角恋)！那么关于它们的概念，接下来我们也得简单介绍介绍，安排上才行！

**2.2 Kati简介**

Kati是专为Android开发的一个基于Golang和C++的工具，主要功能是把Android中的Android.mk文件转换成 Ninja文件。代码路径是build/kati，编译后的产物是ckati。

Kati代码是开源的，可以把它clone下来，如果感兴趣可以查看下其实现原理

这里我们构建一个通过Android.mk配置的LOCAL\_MODULE模块，然后通过top命令就可以查看在编译的过程中执行了ckati的命令。  


**2.3 Ninja简介**

ninja是一个编译框架，会根据相应的ninja格式的配置文件进行编译，但是ninja文件一般不会手动修改，而是通过将Android.bp文件转换成ninja格文件来编译。

**2.4 Android.bp简介**

Android.bp的出现就是为了替换Android.mk文件。而bp跟mk文件不同，它是纯粹的配置，没有分支、循环等流程控制，不能做算数逻辑运算。如果需要控制逻辑，那么只能通过Go语言编写。Android的妈咪谷歌为了让开发者能更加的快速掌握Android.bp特意提供了androidmk命令(关于它的详细介绍可以参见博客[Android.bp入门指南之Android.mk转换成Android.bp](https://blog.csdn.net/tkwxty/article/details/104411520)，这里就不过多的戏说了)用于Android.mk转换成Android.bp使用，如下转换命令:

$ androidmk Android.mk > Android.bp

**2.5 Blueprint和Soong构建编译系统**

**2,5.1 Soong简介**

Soong类似于之前的Makefile编译系统的核心，负责提供Android.bp语义解析，并将之转换成Ninja文件。Soong还会编译生成一个androidmk命令，用于将Android.mk文件转换为Android.bp文件，不过这个转换功能仅限于没有分支、循环等流程控制的Android.mk才有效。

**2.5.2 Blueprint简介**

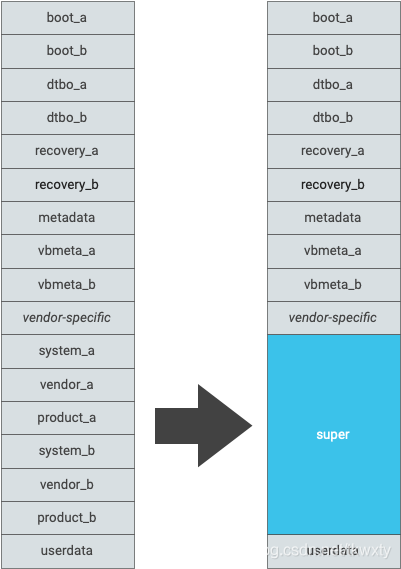
Blueprint是生成、解析Android.bp的工具，是Soong的一部分。Soong负责Android编译而设计的工具，而Blueprint只是解析文件格式，Soong解析内容的具体含义。Blueprint和Soong都是由Golang写的项目，从Android 7.0，prebuilts/go/目录下新增Golang所需的运行环境，在编译时使用。并且因为Soong和Blueprint是Google谷歌为Android.bp特别定制的工具，所以不需要要摘出来单独来操作。

**三.高版本P/Q/R源码编译**

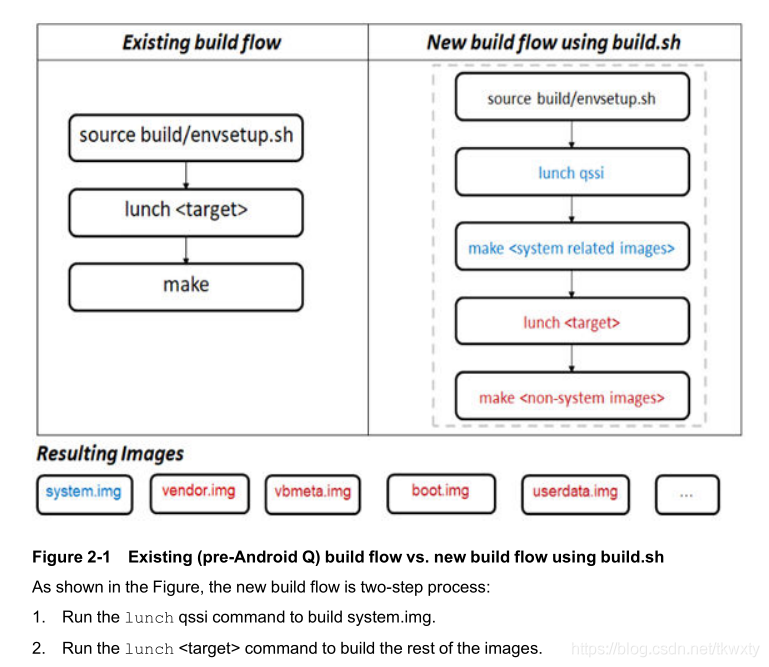
  通过前面的章节我们了解Android编译环境的基本构建和编译的发展史，那么本章节将重点分析Android O之后高阶版本的编译的不同之处。并且本文的博客前年也有说到是以高通版本的Android为基线的。所以在开始本章节的博客前，有两个知识点需要提前介绍下，一个是Android Q以及之后的动态分区，以及qssi的概念！

**3.1 Android动态分区**

动态分区是Android的用户空间分区系统。使用此分区系统，您可以在无线下载(OTA)更新期间创建、销毁分区或者调整分区大小。借助动态分区，供应商无需担心各个分区（例如system、vendor和product）的大小。取而代之的是，设备分配一个super分区，其中的子分区可动态地调整大小。单个分区映像不再需要为将来的OTA预留空间。相反，super中剩余的可用空间还可用于所有动态分区(关于动态分区详见谷歌官方[Android实现动态分区](https://source.android.google.cn/devices/tech/ota/dynamic_partitions/implement?hl=nl))。



**3.2 什么是QSSI**

QSSI 是 Qualcomm Single System Image 的缩写，并且高通平台从Android Q开始支持。并且其编译也和Android原生编译有差别，其差别如下:  


**3.3 具有QSSI特性Android关键的整体编译流程**

通过前面看到QSSI特性的固件编译流程也和通用版本的有一定的区别，这里的编译分为两种模式，第一种Android的标准编译模式，另外一种就是高通提供的编译脚本。

这里需要注意的的是通用版本的Android还是可以直接通过make相关的分区进行直接编译的，譬如make superimage或者直接执行make编译

**3.3.1 通过Android内置make命令编译**

source build/envsetup.sh

* 编译 system.img  
  lunch qssi-userdebug  
  make target-files-package
* 编译除system.img外的其他img  
  lunch xx-userdebug  
  make target-files-package

**3.2.2 高通提供的build.sh脚本进行编译**

* 编译所有img，包括system和其它img

source build/envsetup.sh

lunch XX-userdebug

./build.sh dist -j32

* 编译system.img，产物在qssi目录下

source build/envsetup.sh

lunch xx-userdebug

./build.sh dist qssi\_only -j32

* 编译super.img

source build/envsetup.sh

lunch xx-userdebug

./build.sh dist merge\_only -j32

* 编译其它img，例如vendorimage，如果不指定会编译其它所有img，产物在XX目录下

source build/envsetup.sh

lunch xx-userdebug

./build.sh vendorimage dist target\_only -j32

**3.4 非QSSI特性的整体编译流程**

非QSSI特性的编译流程,依然和以前的版本Android编译变化不大，通常是如下的步骤:

source build/envsetup.sh

lunch xx-userdebug

make

**3.5**[**http://jintianxuesha.com/?cate=12**](http://jintianxuesha.com/?cate=12)**动态分区刷机的方法**

Android Q版本以及以上将system和vendor分区合并为super分区，无法通过adb reboot bootloader模式单独刷动态分区里面的img，例如system,vendor,product,odm，只能刷super.img和其他的，但是fastboot模式下可以单独刷动态分区里面的img，其方法如下:

#推荐进入fastboot模式刷机:

adb reboot fastboot

fastboot getvar is-userspace

is-userspace: yes

Finished. Total time: 0.002s

fastboot flash vendor vendor.img

fastboot flash system system.img

fastboot flash vbmeta vbmeta.img

fastboot flash vbmeta\_system vbmeta\_system.img

#fastbootd是用户空间的代码，因为动态的逻辑分区只能在应用空间识别

1.如果是在linux下fastboot刷机出现权限问题，需要将fastboot的所有者属性改成root  
sudo chown root:root fastboot  
sudo chmod +s fastboot  
2.如果是在windows环境下使用fastboot，很大概率可能不识别fastboot，此时推荐下载360的手机助手借助它安装对应的驱动，这样就能进行相关的识别了，此处是个人经验

**3.6.Framework编译**

现在Android R之上的Framework的编译已经和之前有所不同，具体参见下面解释:

* Android R之前单独编译framework和service命令为:

make -j8 framework

make -j8 services

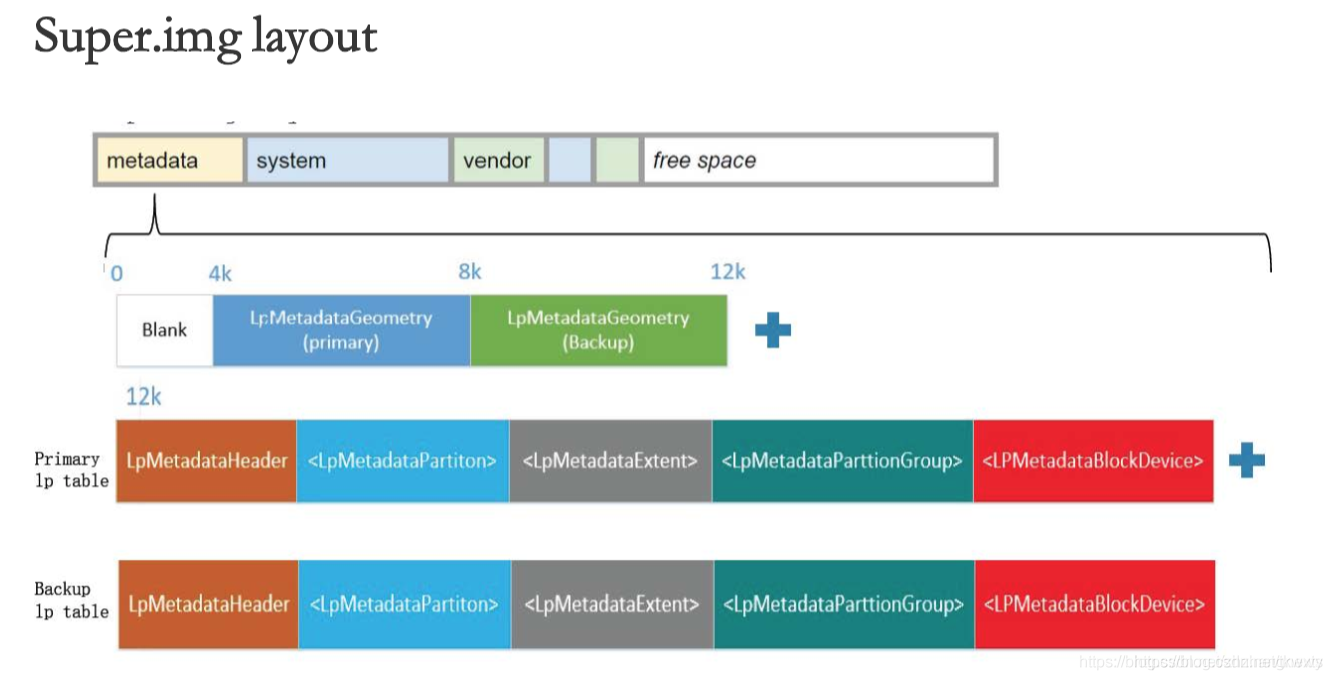
* Android R之后的命令为:

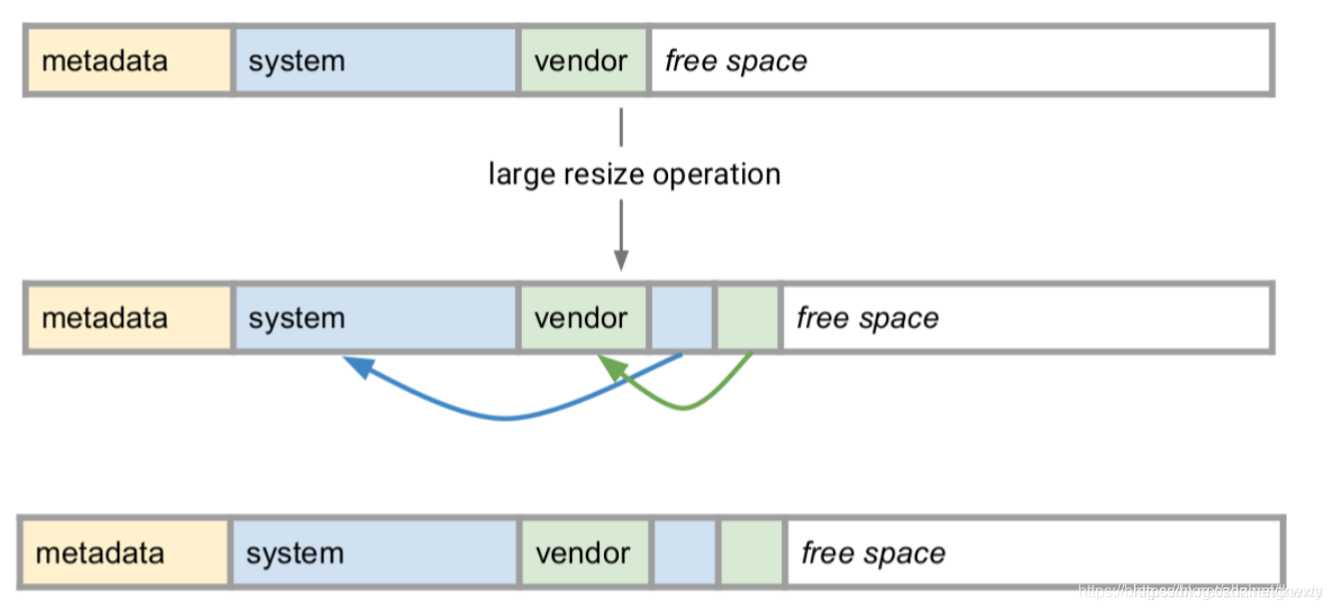
make -j8 framework-minus-apex

make -j8 services

**四.Android为啥要引入动态分区**

  在前面我们简单说了下动态分区的概念，即在Android Q以及以后得编译包中，我们找不到了对应的system,vendor等img文件，但是多了一个super.img，system，vendor，product，ODM合并为super分区，这个就是动态分区了。简单来说就是为了在ota的时候能够灵活创建分区和修改分区大小，将system,vendor,odm,product合并成super分区，并在super分区上预留出一定量的free space，这样就可以动态调整这些分区的大小，解决了ota的时候分区不足，以及调整分区的风险.。

  
当OTA升级之后，需要重新调整分区大小：



**从函数说起**

大家都习惯看从头，从构建目标讲起的，导致每篇文档熟的都是前面的部分。很多教程也都是想办法能够观其大略，从整体上给大家一个思路。比如《深入理解Android内核设计思想》的第4章，比如《Android内核剖析》的第18章，比如《深入解析Android 5.0系统》的第2章。

于是我打算反其道而行之，先从调用函数开始讲。

**最后一招:shell函数**

我们最先把最后看家的绝招列出来吧，shell函数，可以用来执行shell命令。一切用之后讲到的函数解决不了的问题，都可以靠shell函数调用外部功能来解决。  
不过，调用shell需要启动新进程，影响性能，只要有其它方法建议就不要使用它。

比如，我们想要列出当前目录下有哪些cpp文件，makefile可以这样写：

files := $(shell ls \*.cpp)

all :

@echo -n "The files is:"

@echo $(files)

.PHONY : all

**拼接字符串**

在Makefile中用途相当广的就是拼接字符串，join函数就是干这事儿的。  
格式：**$(join 串1 , 串2)**也可以是多个串一起拼：$(join 串1 串2 串3, 串4 串5 串6)  
多个串就是1和4拼，2和5拼。。。总之是逗号前的和逗号后的拼。

**去空格**

有事儿没事儿，去去空格总不是坏事儿。  
$(strip 字符串)

**文件路径操作函数**

在实际的Makefile开发中，经常遇到要处理变量中存储的路径名。

先来个提纲：

* 有一个不知道哪里来的文件名，想取它的路径，用dir
* 有一个不知道哪里来的文件名，想只取它的文件名，用notdir
* 如果只想要扩展名，判断是什么类型的，用suffix
* 如果想不要扩展名，取出来拼个别的扩展名上去，比如xxx.c，只要xxx，将来拼个xxx.o或者就是xxx之类的，用basename
* 想给文件添加个扩展名,用addsuffix
* 想要把文件搞到另一个路径上去，拼个目录名上去，用addprefix

**取一个文件的目录路径函数dir**

例：

.PHONY : all2

oatfile := out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

result := $(dir $(oatfile))

all2 :

@echo -n "The result is: "

@echo $(result)

输出：The result is: out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/

**取文件名函数**

例：

.PHONY : all2

oatfile := out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

result\_dir := $(dir $(oatfile))

result\_notdir := $(notdir $(oatfile))

all2 :

@echo "The result is: "

@echo $(result\_dir)

@echo $(result\_notdir)

输出：  
The result is:  
out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/  
package.odex

**取文件扩展名**

例：

.PHONY : all2

oatfile := out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

result\_dir := $(dir $(oatfile))

result\_notdir := $(notdir $(oatfile))

result\_suffix := $(suffix $(oatfile))

all2 :

@echo "The result is: "

@echo $(result\_dir)

@echo $(result\_notdir)

@echo $(result\_suffix)

输出：  
The result is:  
out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/  
package.odex  
.odex

**取文件基本名**

如果只用basename函数，是连路径都有的，只去除掉了扩展名的名字。不过我们刚学过notdir，一起用下就是了。

例：

.PHONY : all2

oatfile := out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

result\_suffix := $(suffix $(oatfile))

result\_basename := $(basename $(oatfile))

result\_basename2 := $(basename $(notdir $(oatfile)))

all2 :

@echo "The result is: "

@echo $(result\_suffix)

@echo $(result\_basename)

@echo $(result\_basename2)

输出：  
The result is:  
.odex  
out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package  
package

**给名字加后缀**

使用addsuffix函数，$(addsuffix 文件名,扩展名)  
我们举个例子说明：现在想把oat文件压缩成.tar.gz：

.PHONY : all2

oatfile := out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

result\_basename := $(basename $(oatfile))

compress\_oat := tar cfvz $(addsuffix .tar.gz , $(result\_basename)) $(oatfile)

all2 :

@echo "The result is: "

@echo $(compress\_oat)

输出如下：  
The result is:  
tar cfvz out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.tar.gz out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

**给文件名加前缀**

这个用于换路径, addprefix。一个前缀可以后边都加上。

例：

.PHONY : all3

oatfile2 := system.odex music.odex Contacts.odex

result\_oatfile2 = $(addprefix /data/dalvik-cache/, $(oatfile2))

all3:

@echo $(result\_oatfile2)

输出：  
/data/dalvik-cache/system.odex /data/dalvik-cache/music.odex /data/dalvik-cache/Contacts.odex

**直接做字符串替换**

如果我们不想拆开加去的这么麻烦，有一个简易的方法是直接做字符串替换。  
subst函数的定义如下：$(subst 源串,目标串,要做替换的字符串)

例，我们把刚才将.odex扩展名换成.tag.gz扩展名，并拼成一个tar命令的makefile重写一下：

.PHONY : all4

oatfile\_targz := $(subst $(suffix $(oatfile)),.tar.gz,$(oatfile))

compress\_oat := tar cfvz $(oatfile\_targz) $(oatfile)

all4 :

@echo "The result is: "

@echo $(compress\_oat)

输出如下：  
The result is:  
tar cfvz out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.tar.gz out/target/product/ali6753\_65t\_m0/obj/APPS/MusicFX\_intermediates/oat/arm64/package.odex

**过滤函数**

过滤函数有两种：

* filter是符合条件的留下
* filter-out是符合条件的去除掉

我们看一个例子：

all8:

@echo $(filter-out default interpreter jit optimizing,xoc)

@echo $(filter-out default interpreter jit optimizing,default)

输出结果：

$ make all8

xoc

filter-out default interpreter jit optimizing,default这一句，因为default在列表中，所以被过滤掉了，变成一个空串。  
filter-out default interpreter jit optimizing,xoc：这句因为xoc不在过滤列表之中，所以留下了。

**单词的处理**

文件列表，参数列表等等可以看成对单词的处理

比如下面的例子，我们想要查MAKEFILE\_LIST中的最后一个文件，可以这样写：

my-dir = $(call parent-dir,$(lastword $(MAKEFILE\_LIST)))

针对单词处理，有下面的常用函数：

* firstword:取第一个单词
* lastword:取最后一个单词
* words:统计一共有多少个单词
* word:取第n个单词
* wordlist:取单词的子集

word的取值是从1开始，不能取0。

我们看一个例子：

SETTINGS\_ART\_DST := out/target/product/6753\_doov\_l5\_64\_m/system/priv-app/Settings/oat/arm64/Settings.odex

.PHONY : all10

all10:

@echo "Install: $@"

$(eval SETTINGS\_ART\_DST\_LIST := $(subst /, ,$(SETTINGS\_ART\_DST)))

@echo $(words $(SETTINGS\_ART\_DST\_LIST))

@echo $(word 1,$(SETTINGS\_ART\_DST\_LIST))

@echo $(word 2,$(SETTINGS\_ART\_DST\_LIST))

@echo $(wordlist 5,10,$(SETTINGS\_ART\_DST\_LIST))

输出如下：

$ make all10

Install: all10

10

out

target

system priv-app Settings oat arm64 Settings.odex