**第四章 构建系统**

上一章的目标是通过自定义AOSP开发尽可能快地开始运行。此时您无需关闭本书，也可以开始挖掘和修改您的AOSP树，以满足您的需求。所有您需要做的测试您的修改是重建AOSP，再次启动模拟器，如果需要，使用ADB将其重新启动。但是，如果您希望最大化您的努力，您可能需要深入了解Android的构建系统。

虽然它是模块化的，Android的构建系统是相当复杂的，不像任何主流的构建系统;至少主流的系统没有一个用于开源项目。具体来说，它以相当非常规的方式使用make，并且不提供任何类型的基于菜单配置的配置（或等同于此）。Android非常有自己的构建模式，需要一些时间来习惯。所以准备一两杯好咖啡-事情要变得严肃起来了。

像AOSP的其余部分一样，构建系统是一个移动目标。所以在以下信息长时间保持有效时，您应该注意AOSP 版本中使用的更改。

**与其他构建系统的比较**

在开始解释Android构建系统的工作原理之前，首先要强调与Android应用程序的不同之处。首先，与大多数基于make的构建系统不同，Android构建系统不依赖递归makefile。例如，与Linux内核不同，没有顶级的makefile将递归地调用子目录的makefile。相反，有一个脚本可以探索所有目录和子目录，直到找到一个Android.mk文件，然后停止它，并且不会浏览该文件位置下的子目录，除非Android.mk找到指示构建系统。请注意，Android不依赖Makefile文件。相反，它是指定如何构建本地“模块”的Android.mk文件。

另一种Android特性是构建系统的配置方式。虽然我们大多数人习惯于基于内核式menuconfig或GNU autotools（即autoconf，automake等）的系统，Android依赖于一组变量，这些变量可以通过'envsetup.sh'和'lunch'来动态设置为shell环境的一部分，或者在buildspec.mk文件中静态提前定义。对于新手来说，这似乎也是一个惊喜 - Android的构建系统的可配置性水平相当有限。因此，当您可以指定要为其构建AOSP的目标的属性时，在一定程度上，在最终的AOSP中默认应包含哪些应用程序，没有办法让您启用或禁用大多数功能，因为可能 menuconfig。例如，您不能决定您不希望电源管理支持，或者您不希望位置服务默认启动。

此外，构建系统不会在与源文件相同的位置生成对象文件或任何类型的中间输出。例如，您不会在.c源文件中树中找到.o文件。实际上，任何一个输出都没有使用现有的AOSP目录。相反，构建系统创建一个out/目录，它存储它生成的所有东西。因此，make clean与rm -rf out/非常相似。换句话说，删除out/目录会删除任何已构建的内容。

我们开始更详细地探索之前关于构建系统的最后一件事，就是与GNU make有很大关联。更确切的说是3.81版；即使是最新的3.82版如果不更新的话也无法和大多数的AOSP版本一起工作。实际上，构建系统很大程度上依赖于许多GNU特定于特定的功能，如define，include和ifndef指令。

**体系结构(Architecture)**

如图4-1所示，构建系统的入口点是build/core/目录中的main.mk文件，它通过顶层makefile调用，如前所述。build/core/目录实际上包含了大量的构建系统，我们将从那里介绍关键文件。请再次记住，Android构建系统将所有内容都放到单个makefile中; 它不是递归的。因此，您看到的每个.mk文件最终都将成为单个巨大makefile的一部分，其中包含构建系统中所有部分的规则。



图4-1 Android构建系统

**配置(Configuration)**

构建系统的首要任务之一是通过包含config.mk来引入构建配置。可以通过使用envsetup.sh和lunch命令或在顶层目录中提供buildspec.mk文件来配置构建。在这两种情况下，需要设置以下一些变量.

**TARGET\_PRODUCT**

Android的构建风格。例如，每个风格可以包括一组不同的应用程序或区域 设置或构建树的不同部分。看看在build/target/product/中的 AndroidProducts.mk文件包含的各种单一产品的.mk文件，如 Android2.3下的device/samsung/crespo/, and device/htc/passion/目录。 在Android4.2的情况下，请查看device/asus/biter/和device/ samsung/amgnuro/而不是Crespo和Passion。值包括以下内容：

**generic**

“vanilla”的种类，你可以得到的AOSP最基本的部分。

**full**

“all dressed”的类型，大多数应用程序和主要语言环境启用。

**full\_crespo**

与‘full’相同，但是是为Crespo（Samsung Nexus S）准备的。 **full\_grouper**

与‘full’相同，但是是为Grouper (Asus Nexus 7)准备的。

**sim**

Android模拟器.即使这在Android2.3/Gingerbread中可用，但此目标已被 删除，并不在Android4.2/Jelly Bean中。

**sdk**

SDK

**TARGET\_BUILD\_VARIANT**

选择要安装的模块。每个模块应该在其Android.mk中设置一个 LOCAL\_MODULE\_TAGS变量,至少包含以下之一：user,debug,eng,

tests,optional,samples.通过选择该变量，您将告诉构建系统应该包括哪 些模块子集-唯一的例外是这些规则不适用的包（即生成.apk文件的模块）。 特别来说：

eng:包括标记为user，debug或eng的所有模块。

Userdebug:包括标记为用户和调试的两个模块。

User:仅包含标记为用户的模块

**TARGET\_BUILD\_TYPE**

说明是否使用特殊构建标志，或在代码中定义DEBUG变量。这里可能的值是 ‘release’或'debug'。最值得注意的是，frameworks/base/Android.mk 文件在frameworks/base/core/ config/debug和 frameworks/base/core/config/ndebug之间选择，这取决于该变量是否设置 为'debug'。前者导致ConfigBuildFlags。DEBUG将Java常量设置为true， 而后者会将 其设置为false。例如，部分系统服务中的某些代码以DEBUG为 条件。通常，TARGET\_BUILD\_TYPE设置为'release'。

**TARGET\_TOOLS\_PREFIX**

默认情况下，构建系统将使用其在prebuilt/目录(Android4.2下是 prebuilts/目录)之下附带的交叉开发工具链之一。但是，如果您希望使用 其他工具链，则可以将此值设置为指向其位置。

**OUT\_DIR**

默认情况下，构建系统将所有构建输出都放入out/目录中。你可以使用此 变量提供备用的输出目录。

**BUILD\_ENV\_SEQUENCE\_NUMBER**

如果您使用模板build/buildspec.mk.default来创建自己的build spec.mk文件，则该值将被正确设置。但是，如果您使用较早的AOSP版本创 建buildspec.mk，并尝试在将来的AOSP版本中使用它，其中包含对其构建 系统的重要更改，因此，一个不同的值，这个变量将作为一个安全网。这将 导致构建系统通知您，您的buildspec.mk文件与您的构建系统不匹配。

除了选择要建立的AOSP的哪些部分以及与其建立的哪些选项之外，构建系统还需要了解其建立的目标。这是通过一个BoardConfig.mk文件提供的，这将指定要提供给内核的命令行，应该加载内核的基地址，或最适合板卡CPU（TARGET\_ARCH\_VARIANT）的指令集版本。看看build/target/board/一组每个目标的目录，每个目录都包含一个BoardConfig.mk文件。还可以看看AOSP中包含的各种device/\*/TARGET\_DEVICE/BoardConfig.mk文件。后者比前者更丰富，因为它们包含更多的硬件特定信息。设备名称（即TARGET\_DEVICE）是从为配置中的TARGET\_PRODUCT集提供的product.mk文件中指定的PRODUCT\_DEVICE派生的。例如，在Android2.3/Gingerbread中，device/samsung/crespo/AndroidProducts.mk包括device/samsung/crespo/full\_crespo.mk，将PRODUCT\_DEVICE设置为crespo。因此，构建系统在device/\*/crespo/中找到一个BoardConfig.mk，在该位置恰好有一个。Android4.2/Jelly Bean中的device/asus/grouper/full\_grouper.mk中的PRODUCT\_DEVICE设置为'grouper'，从而将构建系统指向device/\*/grouper/BoardConfig.mk。

关于配置的最后一块谜题是用于构建Android的CPU特定选项。对于ARM，这些包含在build/core/combo/arch/arm/armv\*.mk中，TARGET\_ARCH\_VARIANT确定要使用的实际文件。每个文件列出用于构建C / C ++文件的特定于CPU的交叉编译器和交叉链接器标志。它们还包含许多ARCH\_ARM\_HAVE\_\*变量，使得AOSP的其他部分能够基于目标CPU中是否找到给定的ARM功能有条件地构建代码。

**envsetup.sh**

现在，您了解构建系统需要的配置输入种类，我们可以更详细地讨论envsetup.sh的作用。顾名思义，envsetup.sh实际上是为Android设置一个构建环境。但它只是部分工作。主要地，它定义了一系列可用于任何类型的AOSP工作的shell命令：

$ **cd ~/android/aosp-2.3.x**

$ **. build/envsetup.sh**

$ **help**

在4.2/Jelly Bean中，hmm已经取代了help，并且提供给您的命令集已经扩展了：

$ **cd ~/android/aosp-4.2**

$ **. build/envsetup.sh**

$ **hmm**

你可能会发现croot和godir命令对于遍历树很有用。鉴于使用Java及其要求将包存储在与相应的完全限定包名称的每个子部分具有相同层次结构的目录树中的要求相当深入。例如，com.foo.bar包的文件部分必须存储在com/foo/bar/目录下。因此，在AOSP的顶级目录下面找到自己的7到10个目录并不罕见，并且像cd ../../../ ...一样乏味的得以返回到目录的上部。

m和mm也是非常有用的，因为它们允许您分别从顶层构建，无论您在哪里，或只是构建在当前目录中找到的模块。例如，如果您对Launcher进行了修改，并且在packages/apps/Launcher2中，则可以通过键入mm而不是cd回到顶级并键入make来重建该模块。请注意，mm不会重建整个树，因此即使依赖模块已更改，也不会重新生成AOSP镜像。然而m会做那个。尽管如此，mm可以用来测试您的本地更改是否打破构建，直到您准备好重新生成完整的AOSP。

虽然在线帮助没有提到'lunch'，它是由envsetup.sh定义的命令之一。如果你使用‘lunch’是没有任何参数，它会显示一些潜在的选择。

$ **lunch**

这些选择不是静态的。最依赖于当前envsetup.sh运行的AOSP中的内容。事实上，它们使用脚本定义的add\_lunch\_combo（）函数单独添加。例如，在2.3 / Gingerbread中，envsetup.sh默认添加generic-eng和simulator：

*# add the default one here*

*add\_lunch\_combo generic-eng*

*# if we're on linux, add the simulator. There is a special case*

*# in lunch to deal with the simulator*

*if [ "$(uname)" = "Linux" ] ; then*

*add\_lunch\_combo simulator*

*fi*

在4.2 / Jelly Bean中，模拟器不再是有效的目标，而envsetup.sh则改为：

*# add the default one here*

*add\_lunch\_combo full-eng*

*add\_lunch\_combo full\_x86-eng*

*add\_lunch\_combo vbox\_x86-eng*

*add\_lunch\_combo full\_mips-eng*

envsetup.sh还包括可以找到的所有供应商提供的脚本。以下是2.3/ Gingerbread中的实现方法

# Execute the contents of any vendorsetup.sh files we can find.

for f in `/bin/ls vendor/\*/vendorsetup.sh vendor/\*/build/vendorsetup.sh device/\*

/\*/vendorsetup.sh 2> /dev/null`

do

echo "including $f"

. $f

done

unset f

以下是4.2 / Jelly Bean中的实现方法：

# Execute the contents of any vendorsetup.sh files we can find.

for f in `/bin/ls vendor/\*/vendorsetup.sh vendor/\*/\*/vendorsetup.sh device/\*/\*/v

endorsetup.sh 2> /dev/null`

do

echo "including $f"

. $f

done

unset f

在Android2.3中，device/Samsung/Crespo/vendorsetup.sh文件为例

add\_lunch\_combo full\_crespo-userdebug

类似的在Android4.2中，device/asus/grouper/vendorsetup.sh文件是这样的

add\_lunch\_combo full\_grouper-userdebug

那么这就成为你最近看到的菜单。请注意，菜单要求您选择组合。在基本上，这是TARGET\_PRODUCT和TARGET\_BUILD\_VARIANT的组合，但是在Android2.3中有模拟器这个例外。该菜单提供默认组合，但其他组合仍然有效，可以作为参数在命令行中传递给'lunch'。例如在Android2.3中，你可以这样做:

$ **lunch generic-user**

============================================

PLATFORM\_VERSION\_CODENAME=REL

PLATFORM\_VERSION=2.3.4

TARGET\_PRODUCT=generic

TARGET\_BUILD\_VARIANT=user

TARGET\_SIMULATOR=false

TARGET\_BUILD\_TYPE=release

TARGET\_BUILD\_APPS=

TARGET\_ARCH=arm

HOST\_ARCH=x86

HOST\_OS=linux

HOST\_BUILD\_TYPE=release

BUILD\_ID=GINGERBREAD

============================================

$ **lunch full\_crespo-eng**

============================================

PLATFORM\_VERSION\_CODENAME=REL

PLATFORM\_VERSION=2.3.4

TARGET\_PRODUCT=full\_crespo

TARGET\_BUILD\_VARIANT=eng

TARGET\_SIMULATOR=false

TARGET\_BUILD\_TYPE=release

TARGET\_BUILD\_APPS=

TARGET\_ARCH=arm

HOST\_ARCH=x86

HOST\_OS=linux

HOST\_BUILD\_TYPE=release

BUILD\_ID=GINGERBREAD

============================================

一旦‘lunch’完成了一个’generic-eng‘组合的运行，它将在当前shell中设置表4-1中描述的环境变量，为构建系统提供所需的配置信息。

当然，如果你厌倦了总是输入build/envsetup.sh和‘lunch’，你需要做的就是将build/buildspec.mk.default复制到顶层目录中，重命名为buildspec.mk，然后编辑它匹配通过运行这些命令设置的配置。该文件已包含您需要提供的所有变量;这只是一个取消对相应行的注释和适当设置值的问题。一旦你做完了，你所要做的就是去AOSP的目录并直接调用make。你可以跳过envsetup.sh和lunch。

**功能定义(Function Definitions)**

由于构建系统相当大(build/core/ 单独存在40多个.mk文件)，因此可以尽可能多地重用代码。这就是为什么构建系统在definitions.mk文件中定义了大量函数的原因。该文件实际上是构建系统中最大的一个，大约60KB，在2.3版本中makefile代码大约有1,800行约140个函数。在4.2版本构建系统中仍然是最大的文件，大小为73KB，170个函数，以及约2,100行的makefile代码。函数提供各种操作，包括文件查找(例如，all-makefiles-under和all-c-files-under)，转换(例如，transform-c-to-o和transform-java-to-classes.jar)，复制(例如，copy-file-to-target)和实用工具(例如，my-dir)。

这些功能不仅在构建系统组件的其余部分中使用，作为其核心库，但它们有时也直接用在模块的Android.mk文件中使用。以下是计算器应用的Android.mk的示例代码片段：

LOCAL\_SRC\_FILES := $(call all-java-files-under, src)

虽然完全描述definitions.mk不在本书的范围之内，你应该很容易自己去探索它。如果没有别的，大部分功能之前都有一个解释他们做什么的注释。以下是2.3 / Gingerbread的一个例子：

###########################################################

## Find all of the java files under the named directories.

## Meant to be used like:

## SRC\_FILES := $(call all-java-files-under,src tests) ###########################################################

define all-java-files-under

$(patsubst ./%,%, $(shell cd $(LOCAL\_PATH) ; \

find $(1) -name "\*.java" -and -not -name ".\*"))

endef

**主要编译方法(Main Make Recipes)**

在这一点上，您可能会想知道哪些好东西实际上是生成的。各种镜像如RAM磁盘生成或SDK如何放在一起，嗯，我希望你不要抱怨，但我一直保持最好的到最后。所以不用多说，看看在build/core/（不是顶级的）Makefile。该文件以看似无害开始：

# Put some miscellaneous rules here

但不要被愚弄，这是一些最好的肉。下面是Android2.3生成RAM磁盘的代码段：

# ---------------------------------------------------------------

# the ramdisk

INTERNAL\_RAMDISK\_FILES := $(filter $(TARGET\_ROOT\_OUT)/%, \

$(ALL\_PREBUILT) \

$(ALL\_COPIED\_HEADERS) \

$(ALL\_GENERATED\_SOURCES) \

$(ALL\_DEFAULT\_INSTALLED\_MODULES))

BUILT\_RAMDISK\_TARGET := $(PRODUCT\_OUT)/ramdisk.img

# We just build this directly to the install location.

INSTALLED\_RAMDISK\_TARGET := $(BUILT\_RAMDISK\_TARGET)

$(INSTALLED\_RAMDISK\_TARGET): $(MKBOOTFS) $(INTERNAL\_RAMDISK\_FILES) | $(MINIGZIP)

$(call pretty,"Target ram disk: $@")

$(hide) $(MKBOOTFS) $(TARGET\_ROOT\_OUT) | $(MINIGZIP) > $@

而这里是创建用于在同一AOSP版本中检查空中更新（OTA）的证书包的代码段 :

# ---------------------------------------------------------------

# Build a keystore with the authorized keys in it, used to verify the

# authenticity of downloaded OTA packages.

#

# This rule adds to ALL\_DEFAULT\_INSTALLED\_MODULES, so it needs to come

# before the rules that use that variable to build the image.

ALL\_DEFAULT\_INSTALLED\_MODULES += $(TARGET\_OUT\_ETC)/security/otacerts.zip

$(TARGET\_OUT\_ETC)/security/otacerts.zip: KEY\_CERT\_PAIR :=

$(DEFAULT\_KEY\_CERT\_PAIR)

$(TARGET\_OUT\_ETC)/security/otacerts.zip: $(addsuffix .x509.pem,

$(DEFAULT\_KEY\_CERT\_PAIR))

$(hide) rm -f $@

$(hide) mkdir -p $(dir $@)

$(hide) zip -qj $@ $<

.PHONY: otacerts

otacerts: $(TARGET\_OUT\_ETC)/security/otacerts.zip

显然，这里有很多可以举例的，但是看看*Makefile*有关如何创建以下内容的信息：

* Properties (including the target’s /default.prop and /system/build.prop).
* RAM disk.
* Boot image (combining the RAM disk and a kernel image).
* NOTICE files: These are files required by the AOSP’s use of the Apache Software License (ASL). Have a look at the ASL for more information about NOTICE files.
* OTA keystore.
* Recovery image.
* System image (the target’s /system directory).
* Data partition image (the target’s /data directory).
* OTA update package.
* SDK.

不过，有些事情不在这个文件中：

**内核镜像**(**Kernel images)**

不要寻找建立这些的任何规则。官方AOSP版本没有内核部分 - 附录E中列 出的一些第三方项目，然而，实际上将包内核源直接分配到它们分发的AOSP 中。相反，您需要为您的目标找到一个Androidized内核，与AOSP分开构 建，并将其提供给AOSP。您可以在device/中的设备中找到一些示例。例如， 在2.3/Gingerbread中，device/samsung/crespo/包含一个内核映像(文件 叫做kernel)和一个用于Crespo的WiFi的可加载模块(bcm4329.ko file)。 这两个都是在AOSP之外构建的，并以二进制形式复制到树中以便与构建的 其余部分一起包含。

**NDK**

虽然构建NDK的代码在AOSP中，但它与AOSP的build/中的构建系统完全不 同。 相反，NDK的构建系统是在ndk/build/中。我们将讨论如何构建NDK。

**CTS**

构建CTS的规则在build/core/tasks/cts.mk中。

**Cleaning**

正如我前面提到的，make clean是基本等同于擦除out/目录。clean的目标本身在main.mk中定义。然而，还有其他清理目标。最值得注意的是，在cleanbuild.mk中定义的installclean会在您更改TARGET\_PRODUCT，TARGET\_BUILD\_VARIANT或PRODUCT\_LOCALES时自动调用。所以，如果我最开始为Android2.3构建了generic-eng联合体并在其后使用lunch更改联合体为full-eng，那在下一次我开始make时，其中的一些构建输出会自动的被installclean修剪：

$ **make -j16**

============================================

PLATFORM\_VERSION\_CODENAME=REL

PLATFORM\_VERSION=2.3.4

TARGET\_PRODUCT=full

TARGET\_BUILD\_VARIANT=eng

...

============================================

与clean相反，installclean不会清除整个的out/目录。相反，只需要重新构建在组合配置更改的那部分。还有一个clobber目标，基本上是和clean一样。

**模块构建模板(Module Build Templates)**

我刚刚描述的是构建系统的架构和其核心组件的机制。阅读完毕后，您应该从上而下的角度对Android的构建方式有更好的了解。然而，很少有这一点渗透到AOSP模块的Android.mk文件的级别。该系统实际上已经被构建，使得模块构建配方与构建系统的内部结构几乎是独立的。相反，提供构建模板，以便模块作者可以适当地构建其模块。每个模板都针对特定类型的模块量身定制，模块作者可以使用一组记录的变量，所有这些都以LOCAL\_为前缀，以调制模板的行为和输出。当然，模板和底层支持文件（主要是base\_rules.mk）与构建系统的其余部分密切相关，以正确处理每个模块的构建输出。但这是模块的作者是看不见的。

这些模板本身与build/core/中的构建系统的其余部分位于相同的位置。Android.mk通过include指令访问它们。以下是一个例子：

include $(BUILD\_PACKAGE)

如您所见，Android.mk文件实际上并不包含名称为.mk的模板。相反，它们包括一个设置为相应的.mk文件的变量。表4-2提供了可用模块模板的完整列表。

这些构建模板是Android.mk通常相当轻量化:

以下是以Android2.3中的Service Manager在(frameworks/base/cmds/servicemanager/)的Android.mk为例：

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

include $(CLEAR\_VARS)

LOCAL\_SHARED\_LIBRARIES := liblog

LOCAL\_SRC\_FILES := service\_manager.c binder.c

LOCAL\_MODULE := servicemanager

ifeq ($(BOARD\_USE\_LVMX),true)

LOCAL\_CFLAGS += -DLVMX

Endif

include $(BUILD\_EXECUTABLE)

以下是Android2.3的桌面时钟(packages/app/DeskClock/)的例子：

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

include $(CLEAR\_VARS)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := $(call all-java-files-under, src)

LOCAL\_PACKAGE\_NAME := DeskClock

LOCAL\_OVERRIDES\_PACKAGES := AlarmClock

LOCAL\_SDK\_VERSION := current

include $(BUILD\_PACKAGE)

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

如您所见，两个模块中基本上使用相同的结构，尽管它们提供非常不同的输入，并产生非常不同的输出。还要注意Desk Clock的Android.mk的最后一行，它基本上包括所有子目录的Android.mk文件。如前所述，构建系统在层次结构中查找第一个makefile，并不会在找到目录的目录下面的任何子目录中查找，因此需要手动调用它们。很显然，这得代码只是出来在底层去寻找所有的makefile。但是，AOSP的某些部分将根据配置明确列出子目录或有条件地选择它们。

[http://source.android.com](https://source.android.com/)上的文档用于提供所有LOCAL\_ \*变量的详尽列表及其含义和用途。不幸的是，在撰写本文时，此列表已不再可用。然而，build/core/build-system.html文件包含该列表的早期版本，您应该引用该文件，直到最新列表再次可用。以下是一些最常遇到的LOCAL\_ \*变量：

**LOCAL\_PATH**

通常通过调用$(call my-dir)来提供当前模块源的路径。

**LOCAL\_MODULE**

属于此模块的构建输出的名称。实际的文件名或输出及其位置将取决于 您包含的构建模板。例如，如果设置为foo，并且您构建一个可执行文 件，则最终的可执行文件将是一个名为foo的命令，它将被放在目标的 /system/bin/目录中。如果LOCAL\_MODULE设置为libfoo，并且包括 BUILD\_SHARED\_LIBRARY而不是BUILD\_EXECUTABLE，构建系统将生成 libfoo.so并将其放在/system/lib/中。请注意，您在此处提供的名称 对于正在构建的特定模块类（即构建模板类型）必须是唯一的。例如， 不能有两个libfoo.so库。预计在将来某个时候，模块名称必须是全局 唯一的（即跨所有模块类）。

**LOCAL\_SRC\_FILES**

用于构建模块的源文件。您可以通过使用构建系统定义的函数之一来提 供这些功能，比如桌面时钟使用all-java-files-under，或者您可以像 Service Manager一样显式列出文件。

**LOCAL\_PACKAGE\_NAME**

与所有其他模块不同，app使用此变量而不是LOCAL\_MODULE提供其名称， 您可以通过比较前面显示的两个Android.mk文件来看到。

**LOCAL\_SHARED\_LIBRARIES**

使用它来列出您的模块所依赖的所有库。如前所述，Service Manager 对liblog的依赖使用此变量进行指定。

**LOCAL\_MODULE\_TAGS**

如前所述，这允许您控制此模块构建的TARGET\_BUILD\_VARIANT。通常， 这应该被设置为optional

**LOCAL\_MODULE\_PATH**

使用它来覆盖您正在构建的模块类型的默认安装位置。

了解更多LOCAL\_ \*变量的一个好方法是查看AOSP中现有的Android.mk文件。此外，clear\_vars.mk包含已清除的变量的完整列表。所以虽然没有给你每个的含义，但它肯定列出了所有变量。

此外，除了影响全局AOSP的清洁目标之外，每个模块都可以通过提供CleanSpec.mk来定义自己的清理规则，就像模块提供的Android.mk文件一样。不像后者，不过前者不是必需的。默认情况下，构建系统具有每种类型的模块的清除规则。但是，您可以在CleanSpec.mk中指定自己的规则，以防您的模块的构建在构建系统默认情况下不会生成，因此通常不会知道如何清理。

**输出(output)**

现在我们已经看到了构建系统的工作原理，以及模块使用的构建模板，我们来看看它在out/中创建的输出。在相当高的层次上，构建输出分为三个阶段，两种模式，一种用于主机，一种用于目标:

1. 中间体是使用模块源生成的。这些中间体的格式和位置取决于模块的来源。例如它们可能是基于C/C ++代码的.o文件，也可能是基于Java的代码的.jar文件。
2. 中间体由构建系统用于创建实际的二进制文件和软件包：以.o文件为例，将它们链接到一个实际的二进制文件中。
3. 二进制程序和程序包被组合在一起构建系统所请求的最终输出。例如，二进制文件被复制到包含root和/system文件系统的目录中，并且生成这些文件系统的镜像以便在实际的设备上使用。

out/主要分为两个目录，反映其操作模式：host/和target/。在每个目录中，您将找到一些包含构建过程中生成的各种中间体的obj/目录。大多数这些存储在前面提到的名为“BUILD\_\*”宏的子目录中，并在构建系统的操作过程中提供了特定的补充目的：

* EXECUTABLES/
* JAVA\_LIBRARIES/
* SHARED\_LIBRARIES/
* STATIC\_LIBRARIES/
* APPS/
* DATA/
* ETC/
* KEYCHARS/
* PACKAGING/
* NOTICE\_FILES/
* include/
* lib/

您可能最感兴趣的目录是out/target/product/PRODUCT\_DEVICE/。这就是相应产品配置的.mk中定义的PRODUCT\_DEVICE的输出镜像所在的位置。表4-3说明了该目录的内容。

请查看第2章，重新了解根文件系统，/system和/data。实际上，当内核引导时，它将挂载RAM磁盘映像并执行内部的/init。这个二进制文件依次运行/init.rc脚本，它会将/system和/data镜像挂载在各自的位置。我们将在第6章中回到根文件系统布局和引导时的系统操作。

**构建方法(Build Recipes)**

考虑到构建系统的架构和功能，让我们来看看一些最常见的，有些不太常见的构建方法。我们只会轻点使用每个方法的结果，但您应该有足够的信息来开始。

**默认的droid构建**

早些时候，我们经历了一些简单的make命令，但从来没有真正解释过默认目标。当你纯粹运行make时，就像你输入下面命令一样：

$ **make droid**

droid实际上是main.mk中定义的默认目标。您通常不需要手动指定此目标。我在这里提供它的完整形式，所以你知道它存在。

**查看构建命令(seeing the build commands)**

当您构建AOSP时，您会注意到它并不会显示它正在运行的命令。相反，它只打印出每个步骤的摘要。如果要查看其所做的一切，例如gcc命令行，请将showcommands目标添加到命令行：

$ **make showcommands**

结合我在上一节中所解释的内容，上诉命令与以下内容相同：

$ **make droid showcommands**

正如你在使用这种情况时会很快注意到的，它会产生大量的输出，因此很难跟踪。但是，如果要分析用于构建AOSP的实际命令，则可能希望将标准输出和标准错误保存到文件中：

$ **make showcommands > aosp-build-stdout 2> aosp-build-stderr**

您还可以执行此操作将所有输出合并到单个文件中：

$ **make showcommands 2>&1 | tell build.log**

有些还说，他们更喜欢使用nohup命令：

$ **nohup make showcommands**

**构建适用于Linux和Mac OS的SDK**

官方的Android SDK可以在[http://developer.android.com](https://developer.android.com/index.html)上找到。然而，您可以使用AOSP构建自己的SDK，例如，如果扩展了核心API以展示新功能，并希望将结果分发给开发人员，以便他们可以从新的API中受益。为此，您需要选择一个特殊的组合：

$ **. build/envsetup.sh**

$ **lunch sdk-eng**

$ **make sdk**

一旦这些命令执行完，SDK在out/host/linux-x86/sdk/(linux下构建)和out/host/darwin-x86/sdk/(Mac下构建)下将有两个副本，一个ZIP文件，非常像在[http://developer.android.com](http://developer.android.com/)上分发的版本，一个未压缩并可以使用。

假设您已经使用[http://developer.android.com](http://developer.android.com/)上的说明配置Eclipse进行Android开发，您需要执行两个额外的步骤来使用您新建的SDK。首先，您需要告诉Eclipse新SDK的位置。要实现这个目的，请转到窗口→首选项→Android，在“SDK位置”框中输入新SDK的路径，然后单击“确定”。另外，由于在撰写本文时并不完全清楚的原因，您还需要转到Window→Android SDK Manager，取消选择所有可能选择的项目，除了前两个在“工具”下，然后单击“安装2个软件包...”完成之后，您将能够使用新的SDK创建新项目，并访问您在其中公开的任何新API。如果您不这样做第二步，您将能够创建新的Android项目，但没有一个将正确解析Java库，因此将永远不会构建。

**构建适用于Windows的SDK**

构建Windows SDK的指令与Linux和Mac OS略有不同：

$ **. build/envsetup.sh**

$ **lunch sdk-eng**

$ **make win\_sdk**

结果输出将在/host/windows/sdk/中。

**构建CTS**

如果你想建立CTS，你不需要使用envsetup.sh或lunch。你可以直接输入：

$ **make cts**

cts命令包含自己的在线帮助。以下是Android2.3/Gingerbread的相应示例输出：

$ **cd out/host/linux-x86/bin/**

$ **./cts**

$ **cts\_host > help**

$ **cts\_host > ls --plan**

一旦你有一个运行目标，如模拟器，你可以启动测试套件，它将使用adb在目标上来运行测试：

$ **./cts start --plan CTS**

**构建NDK**

如前所述，NDK有自己的独立构建系统，具有自己的安装和帮助系统，您可以这样调用：

$ **cd ndk/build/tools**

$ **export ANDROID\_NDK\_ROOT=aosp-root/ndk**

$ **./make-release --help**

当您准备构建NDK时，可以按以下方式调用make-release，并且看到其相当明显的警告：

$ **./make-release**

IMPORTANT WARNING !!

This script is used to generate an NDK release package from scratchfor the following host platforms: linux-x86

This process is EXTREMELY LONG and may take SEVERAL HOURS on a dual-core machine. If you plan to do that often, please read docs/DEVELOPMENT.TXT that provides instructions on how to do that more easily.

Are you sure you want to do that [y/N]

**y**

Downloading toolchain sources...

...

**更新API**

构建系统具有保护措施来防止你修改了AOSP的核心API。如果你这样做了，默认情况下，构建将失败，并显示如下警告：

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

You have tried to change the API from what has been previously approved.

To make these errors go away, you have two choices:

1) You can add "@hide" javadoc comments to the methods, etc. listed in the errors above.

2) You can update current.xml by executing the following command:

make update-api

To submit the revised current.xml to the main Android repository,

you will need approval.

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

make: \*\*\* [out/target/common/obj/PACKAGING/checkapi-current-timestamp] Error 38

make: \*\*\* Waiting for unfinished jobs....

如错误信息提示，为了继续构建，您需要执行以下操作：

$ m**ake update-api**

...

Install: out/host/linux-x86/framework/apicheck.jar

Install: out/host/linux-x86/framework/clearsilver.jar

Install: out/host/linux-x86/framework/droiddoc.jar

Install: out/host/linux-x86/lib/libneo\_util.so

Install: out/host/linux-x86/lib/libneo\_cs.so

Install: out/host/linux-x86/lib/libneo\_cgi.so

Install: out/host/linux-x86/lib/libclearsilver-jni.so

Copying: out/target/common/obj/JAVA\_LIBRARIES/core\_intermediates/ emma\_out/lib/classes-jarjar.jar

Install: out/host/linux-x86/framework/dx.jar

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

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

Copying: out/target/common/obj/JAVA\_LIBRARIES/bouncycastle\_interm ediates/emma\_out/lib/classes-jarjar.jar

Copying: out/target/common/obj/JAVA\_LIBRARIES/ext\_intermediates/e mma\_out/lib/classes-jarjar.jar

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

Copying: out/target/common/obj/JAVA\_LIBRARIES/core-junit\_intermed iates/emma\_out/lib/classes-jarjar.jar

Copying: out/target/common/obj/JAVA\_LIBRARIES/framework\_intermedi ates/emma\_out/lib/classes-jarjar.jar

Copying current.xml

下次开始make时，您不会再收到有关API更改的错误信息。显然，在这一点上，您不再与官方API兼容，因此不太可能被Google认证为“Android”设备。

**构建一个独立模块**

到目前为止，我们已经研究了构建整个树。您还可以构建单个模块。以下是如何要求构建系统构建Launcher2模块（即主屏幕）：

$ **make Launcher2**

您也可以单独clean模块：

$ **make clean-Launcher2**

如果要强制构建系统重新生成系统映像以包含更新的模块，则可以将snod目标添加到命令行：

$ **make Launcher2 snod**

**脱离树的构建**

如果您希望针对AOSP及其仿生库构建代码，但不希望将其纳入AOSP，您可以使用如以下类似的makefile来完成工作

# Paths and settings

TARGET\_PRODUCT = generic

ANDROID\_ROOT = /home/karim/android/aosp-2.3.x

BIONIC\_LIBC = $(ANDROID\_ROOT)/bionic/libc

PRODUCT\_OUT = $(ANDROID\_ROOT)/out/target/product/$(TARGET\_PRODUCT)

CROSS\_COMPILE = \ $(ANDROID\_ROOT)/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/a rm-eabi-

# Tool names

AS = $(CROSS\_COMPILE)as

AR = $(CROSS\_COMPILE)ar

CC = $(CROSS\_COMPILE)gcc

CPP = $(CC) -E

LD = $(CROSS\_COMPILE)ld

NM = $(CROSS\_COMPILE)nm

OBJCOPY = $(CROSS\_COMPILE)objcopy

OBJDUMP = $(CROSS\_COMPILE)objdump

RANLIB = $(CROSS\_COMPILE)ranlib

READELF = $(CROSS\_COMPILE)readelf

SIZE = $(CROSS\_COMPILE)size

STRINGS = $(CROSS\_COMPILE)strings

STRIP = $(CROSS\_COMPILE)strip

export AS AR CC CPP LD NM OBJCOPY OBJDUMP RANLIB READELF \

SIZE STRINGS STRIP

# Build settings

CFLAGS = -O2 -Wall -fno-short-enums

HEADER\_OPS = -I$(BIONIC\_LIBC)/arch-arm/include \

-I$(BIONIC\_LIBC)/kernel/common \

-I$(BIONIC\_LIBC)/kernel/arch-arm

LDFLAGS = -nostdlib -Wl,-dynamic-linker,/system/bin/linker \

$(PRODUCT\_OUT)/obj/lib/crtbegin\_dynamic.o \

$(PRODUCT\_OUT)/obj/lib/crtend\_android.o \

-L$(PRODUCT\_OUT)/obj/lib -lc –ldl

# Installation variables

EXEC\_NAME = example-app

INSTALL = install

INSTALL\_DIR = $(PRODUCT\_OUT)/system/bin

# Files needed for the build

OBJS = example-app.o

# Make rules

all: example-app

.c.o: $(CC) $(CFLAGS) $(HEADER\_OPS) -c $<

example-app: ${OBJS}

$(CC) -o $(EXEC\_NAME) ${OBJS} $(LDFLAGS)

install: example-app

test -d $(INSTALL\_DIR) || $(INSTALL) -d -m 755 $(INSTALL\_DIR)

$(INSTALL) -m 755 $(EXEC\_NAME) $(INSTALL\_DIR)

clean:

rm -f \*.o $(EXEC\_NAME) core

distclean:

rm -f \*~

rm -f \*.o $(EXEC\_NAME) core

在这种情况下，您不需要关心envsetup.sh或lunch。你可以直接输入'魔法咒语'：

$ **make**

显然，这不会将您的二进制文件添加到AOSP生成的任何镜像中。即使install的目标在将目标文件系统从NFS上卸载才有效，而且有效值只在debug是才是有效的，这是makefile被认为是有用的。在某种程度上，也可以认为使用这样一个makefile实际上是适得其反的，因为它远远超过了将这个代码作为AOSP的模块部分添加时产生的相当于Android.mk的复杂性。

不过，这种技巧可以有其用途。例如，在某些情况下，修改由相当大的代码库使用的常规构建系统可能有意义，以在其外部的AOSP上构建该项目;另一种方法是将项目复制到AOSP中，并创建Android.mk文件来重现其原始常规构建系统的机制，这可能是本身的一个实质性的工作。

**递归构建**

如果你真的想要，你可以自己修改一个makefile来构建一个基于递归makefile的组件，而不是使用Android.mk文件来重现相同的功能，如上一节所述。例如，附录E中提到的几个AOSP分支将内核源包括在AOSP的顶层，并修改AOSP的主makefile以调用内核的现有构建系统。

**AOSP的基本技巧**

你买这本书最有可能的原因是：你修改AOSP来适用于你的需求。在接下来的几页中，我们会寻找一些你想要尝试的最明显的技巧。当然，我们只是在这里设置与构建系统相关的部分，这是您可能希望开始的部分。

**添加一个设备**

添加自定义设备很可能是您阅读本书的原因列表中最重要的项目之一（如果不是最重要的）。我要告诉你如何做，所以你可能想要标记这个部分。当然，我其实只是向你展示这个作品的构建方面。将Android移植到新硬件方面还有很多步骤。但是，将新设备添加到构建系统中一定是您首先做的事情之一。幸运的是，这件事比较简单。

为了本次练习的目的，假设您为名为ACME的公司工作，并且您负责提供其最新的小发明：CoyotePad，旨在成为玩所有鸟类游戏的最佳平台。让我们开始在device/中为我们的新设备创建一个条目

$ **cd ~/android/aosp-2.3.x**

$ **. build/envsetup.sh**

$ **mkdir -p device/acme/coyotepad**

$ **cd device/acme/coyotepad**

我们在这里需要的第一件事就是创建一个AndroidProducts.mk文件来描述可以为CoyotePad构建的各种AOSP产品：

PRODUCT\_MAKEFILES := $(LOCAL\_DIR)/full\_coyotepad.mk

虽然我们可以描述几个产品（参见build/target/product/AndroidProd ucts.mk作为示例），典型的情况是指定一个，在这种情况下，它在full\_coyotepad.mk中描述：

$(call inherit-product, $(SRC\_TARGET\_DIR)/product/languages\_full.mk)

# If you're using 4.2/Jelly Bean, use full\_base.mk instead of full.mk

$(call inherit-product, $(SRC\_TARGET\_DIR)/product/full.mk)

DEVICE\_PACKAGE\_OVERLAYS :=

PRODUCT\_PACKAGES +=

PRODUCT\_COPY\_FILES +=

PRODUCT\_NAME := full\_coyotepad

PRODUCT\_DEVICE := coyotepad

PRODUCT\_MODEL := Full Android on CoyotePad, meep-meep

值得仔细看看这个makefile。首先，我们使用‘inherit-product’功能来告诉构建系统将其他产品描述作为我们的基础。这使我们能够建立在其他人的工作上，而不必从头开始指定我们想要包括的AOSP。languages\_full.mk将拉入大量的区域设置，并且full.mk将确保我们获得与使用full-eng组合一样构建的同一组模块

关于其他变量：

DEVICE\_PACKAGE\_OVERLAYS

允许我们指定一个目录，该目录将构成将应用到AOSP源的重叠的基础， 从而允许我们用设备特定资源替换默认的包资源。例如，如果您想为 Launcher2 或其他应用程序设置自定义布局或颜色，则会发现此功能非 常有用。我们将在下一节中介绍如何使用它。

PRODUCT\_PACKAGES

允许我们指定要包含此产品的软件包，以及我们已经继承的产品中指定 的软件包。例如，如果您有定制化app，二进制文件或位于 device/acme/coyotepad/中的库，您需要将它们添加到这里，以便将它 们包含在生成的最终镜像中。注意使用+ =符号。它允许我们附加到变量 中的现有值，而不是替换其内容。

PRODUCT\_COPY\_FILES

允许我们列出我们想要看到的复制到目标文件系统的特定文件以及需要 复制到的位置。每对的目的/源被冒号‘:’分割，每队之前用空格'space' 分开。这对于配置文件和预构建的二进制文件（如固件映像或内核模块） 非常有用。

PRODUCT\_NAME

TARGET\_PRODUCT，您可以通过选择lunch组合或将其作为组合参数的一 部分传递给lunch，如下所示：

$ lunch full\_coyotepad-eng

PRODUCT\_DEVICE

给到客户的实际成品的名称。TARGET\_DEVICE来自此变量。 PRODUCT\_DEVICE必须匹配device/acme/中的一个条目，因为这是构建查 找相应的BoardConfig.mk的位置。在这种情况下， 该变量与我们已经 存在的目录的名称相同。

PRODUCT\_MODEL

设置中“关于手机”部分“型号”中提供的本产品的名称。该变量实际 上被存储为设备上可访问的ro.product.model全局属性。

版本4.2/Jelly Bean还包括通常设置为Android的PRODUCT\_BRAND。然后，该变量的值可用作ro.product.brand全局属性。后者由堆栈的某些部分用于基于设备的供应商的动作。现在我们已经描述了该产品，我们还必须通过BoardConfig.mk文件提供有关设备正在使用的板卡的一些信息：

TARGET\_NO\_KERNEL := true

TARGET\_NO\_BOOTLOADER := true

TARGET\_CPU\_ABI := armeabi

BOARD\_USES\_GENERIC\_AUDIO := true

USE\_CAMERA\_STUB := true

这是一个很瘦小的BoardConfig.mk，并确保我们实际构建成功。对于该文件的现实版本，可以查看Android2.3的device/samsung/crespo/BoardConfigCommon.mk或者Android4.2

的device/asus/grouper/BoardConfigCommon.mk。

您还需要提供一个常规的Android.mk来构建您可能已经包含在此设备目录中的所有模块：

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

include $(CLEAR\_VARS)

ifneq ($(filter coyotepad,$(TARGET\_DEVICE)),)

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

endif

它实际上是首选的操作方式，将所有设备特定的app，二进制文件和库放在设备的目录中，而不是全局的AOSP的其余部分。如果您在这里添加模块，不要忘了还将它们添加到我们前面描述过的PRODUCT\_PACKAGES中。如果您只是将它们放在这里并提供有效的Android.mk文件，那么它们将被构建，但它们不会在最终的镜像中。

如果您有多个产品共享相同的软件包，您可能需要创建一个包含共享软件包的device/acme/common/目录。你可以在Android4.2的device/generic/目录中看到一个例子。在同一版本中，您还可以查看device/samsung/maguro/device.mk如何从device/samsung/tuna/device.mk继承一个设备基于另一个设备的示例。

最后，让我们将envsetup.sh中添加的设备添加到lunch中来关闭该循环。为此，您需要在设备的目录中添加vendorsetup.sh：

add\_lunch\_combo full\_coyotepad-eng

您还需要确保它是可执行的，如果它是可操作的：

$ **chmod 755 vendorsetup.sh**

现在我们可以回到AOSP的根目录并将我们最新的ACME CoyotePad来展开追逐：

$ **croot**

$ **. build/envsetup.sh**

$ **lunch**

You're building on Linux

Lunch menu... pick a combo:

1. generic-eng

2. simulator

3. full\_coyotepad-eng

4. full\_passion-userdebug

5. full\_crespo4g-userdebug

6. full\_crespo-userdebug

Which would you like? [generic-eng] **3**

============================================

PLATFORM\_VERSION\_CODENAME=REL

PLATFORM\_VERSION=2.3.4

TARGET\_PRODUCT=full\_coyotepad

TARGET\_BUILD\_VARIANT=eng

TARGET\_SIMULATOR=false

TARGET\_BUILD\_TYPE=release

TARGET\_BUILD\_APPS=

TARGET\_ARCH=arm

HOST\_ARCH=x86

HOST\_OS=linux

HOST\_BUILD\_TYPE=release

BUILD\_ID=GINGERBREAD

============================================

$ **make -j16**

正如你看到的，AOSP以及可以识别我们的新设备并打印出了正确的信息。构建完成后，我们还将提供与其他任何AOSP构建相同类型的输出，除了它将是特定产品的目录。

另外，请看system/中的build.prop文件。它包含可在运行时在目标上提供的各种全局属性，并且与我们的配置和构建相关。

你可以想象，这里还有很多事情要做，以确保AOSP在我们的硬件上运行。但是前面的步骤给了我们起点。然而，通过隔离单个目录中的单板特定更改，此配置将简化将CoyotePad的附加支持添加到下一个版本的AOSP中。实际上，它只是将相应的目录复制到新的AOSP的device/目录，并调整其中的代码以使用新的API。

**添加一个app**

添加一个应用程序到你的板子上是比较直接的。试着将用Eclipse和默认的SDK创建一个‘HelloWorld!’app作为起点。Eclipse中的所有新的Android项目都默认是'Hello World!'.然后将该应用程序从Eclipse工作区复制到其目的地：

$ **cp -a ~/workspace/HelloWorld ~/android/aosp-2.3.x/device/acme/coyotepad/**

然后，您必须在aosp-root/device/acme/coyotepad/HelloWorld/中创建一个Android.mk文件来构建该应用程序：

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

include $(CLEAR\_VARS)

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := $(call all-java-files-under, src)

LOCAL\_PACKAGE\_NAME := HelloWorld

include $(BUILD\_PACKAGE)

鉴于我们将此模块标记为可选项，默认情况下不会包含在AOSP构建中。要包括它，您需要将其添加到CoyotePad的full\_coyotepad.mk中列出的PRODUCT\_PACKAGES。

如果，不仅是添加您的应用程序到你的板子上，您想在全球范围内为AOSP生成的所有产品以及现有常用应用程序添加默认应用程序，您需要将其放在packages/app中，而不是您的板子目录下。您还需要修改内置的.mk文件，如aosp-root/build/target/product/core.mk，以便默认构建您的应用程序。不过建议不要这样做，因为它不是很便携，因为它需要你对每个新的AOSP版本进行修改。如前所述，最好尽可能多地在device/acme/coyotepad中进行自定义修改。

**添加一个app覆盖**

有时您实际上并不想添加一个应用程序，而是修改AOSP中默认的现有应用程序。这就是应用程序的覆盖。覆盖是AOSP中包含的一种机制，允许设备制造商更改所提供的资源(例如app)，而无需实际修改AOSP中包含的原始资源.要使用此功能，您必须创建一个覆盖树，并告知构建系统。重叠式最简单位置在特定于设备的目录中，例如我们在上一节中创建的目录：

$ **cd device/acme/coyotepad/**

$ **mkdir overlay**

要告诉构建系统考虑到这个覆盖，我们需要修改我们的full\_coyotepad.mk：

DEVICE\_PACKAGE\_OVERLAYS := device/acme/coyotepad/overlay

在这一点上，尽管如此，我们的覆盖没有多少。假设我们要修改一些Launcher2的默认字符串。 我们可以这样做：

$ mkdir -p overlay/packages/apps/Launcher2/res/values

$ cp aosp-root/packages/apps/Launcher2/res/values/strings.xml overlay/packages/apps/Launcher2/res/values/

然后，您可以修剪本地strings.xml以仅覆盖所需的字符串。最重要的是，您的设备将有一个具有自定义字符串的Launcher2，但默认的Launcher2仍将具有原始字符串。因此，如果有人依赖于您用于构建另一个产品的相同的AOSP源，那么它们仍然会获得原始字符串。当然，您可以替换大多数资源，包括图像和XML文件。只要将文件放置在与AOSP中相同的层次结构中，而不是device/acme/coyotepad/overlay/中，它们将被构建系统考虑在内。

**添加一个Native的工具或者后台进程**

像上面添加一个应用程序的例子一样，您可以添加您的自定义本地工具和后台在device/acme/coyotepad/的子目录。显然，您需要在包含代码的目录中提供一个Android.mk来构建该模块：

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

include $(CLEAR\_VARS)

LOCAL\_MODULE := hello-world

LOCAL\_MODULE\_TAGS := optional

LOCAL\_SRC\_FILES := hello-world.cpp

LOCAL\_SHARED\_LIBRARIES := liblog

include $(BUILD\_EXECUTABLE)

在应用程序的情况下，您还需要确保hello-world是CoyotePad的PRODUCT\_PACKAGES的一部分。

如果您打算将全局的二进制文件全部添加到所有产品构建中，而不是仅在本地添加到您的主板上，则需要知道树中有多个本地工具和后台进程所在的位置。这些里是最重要的：

system/core/ 和 system/

定制Android二进制文件，意在在Android Framework外部使用或独立 的部分。

frameworks/base/cmds/

与Android Framework紧密耦合的二进制文件。例如，这是Service Manager和installd的位置。

external/

由导入到AOSP的外部项目生成的二进制文件。例如，strace在这里。

从上面列出的代码生成二进制代码的位置，您还需要将其添加为全局.mk文件之一，如aosp-root/build/target/product/core.mk。然而，如上所述，不建议使用全局添加，因为它们不能轻易转移到较新的AOSP版本。

**添加一个本地库**

像app和二进制文件一样，您还可以为您的主板添加本机库。假设，如上所述，构建库的源位于device/acme/coyotepad/的子目录中，您将需要一个Android.mk来构建库：

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

include $(CLEAR\_VARS)

LOCAL\_MODULE := libmylib

LOCAL\_MODULE\_TAGS := optional

LOCAL\_PRELINK\_MODULE := false

LOCAL\_SRC\_FILES := $(call all-c-files-under,.)

include $(BUILD\_SHARED\_LIBRARY)

要使用此库，您必须将其添加到由Android.mk文件列出的任何二进制文件所依赖的库中：

LOCAL\_SHARED\_LIBRARIES := libmylib

您还可能需要将相关标头添加到与放置库相同位置的include/目录中，以便需要链接到库的代码可以找到这些标题，例如device/acme/coyotepad/include/。

如果您想将库全局应用于所有AOSP构建，而不仅仅是您的设备，那么您需要更多关于树通常在库中找到的各种位置的信息。首先，您应该知道，与二进制文件不同，在单个模块中使用了很多库，但是不会适用其他地方的库。因此，这些库通常将被放置在该模块的代码中，而不是位于系统范围内使用库的通常位置。后者通常在以下位置：

system/core/

系统许多部分使用的库，包括Android Framework之外的一些。这就是 liblog所在的地方。

frameworks/base/libs/

库文件与框架密切相关。 这是libbinder的地方。

frameworks/native/libs/

在Android4.2中，许多在Android2.3中frameworks/base/libs/下的 库已被移出并进入frameworks/native/libs/。

external/

由外部项目生成的库导入AOSP。 OpenSSL的libssl在这里。

类似地，代替使用CoyotePad特定的包含目录，您可以使用全局目录，如system/core/include/ or frameworks/base/include/ 或者4.2版中frameworks/base/include/.同样，如前所述，您应该仔细检查是否真正需要这样的全球添加，因为当您尝试将设备移植到下一个版本的Android时，它们将是额外的工作量。