|  |
| --- |
| Elvonzion.WX |
| Android |
|  |
|  |
| **elvonzion@gmail.com** |
| **2012/8/12** |

|  |
| --- |
|  |

**目 录**

[1 系统结构 1](#_Toc390804245)

[1.1 系统架构 1](#_Toc390804246)

[1.1.1 系统架构 1](#_Toc390804247)

[1.1.2 AOSP结构 1](#_Toc390804248)

[1.2 build system 3](#_Toc390804249)

[1.2.1 构造规则 3](#_Toc390804250)

[1.2.2 配置流程 7](#_Toc390804251)

[1.2.3 编译配置 12](#_Toc390804252)

[1.2.4 产品配置 17](#_Toc390804253)

[1.2.5 模块配置 26](#_Toc390804254)

[1.2.6 系统预构造 32](#_Toc390804255)

[1.2.7 多模块构造 35](#_Toc390804256)

[1.2.8 单模块构造 39](#_Toc390804257)

[1.2.9 系统盘构造 46](#_Toc390804258)

[1.2.10 系统安装 47](#_Toc390804259)

[1.2.11 通用规则 50](#_Toc390804260)

[1.3 系统结构 55](#_Toc390804261)

[1.3.1 sp 55](#_Toc390804262)

[1.4 运行结构 56](#_Toc390804263)

[1.4.1 ServiceManger 56](#_Toc390804264)

[1.4.2 goldfish 57](#_Toc390804265)

[1.4.3 Dalvik 57](#_Toc390804266)

[1.4.4 Ashmem 57](#_Toc390804267)

[1.4.5 Binder 58](#_Toc390804268)

[1.4.6 Logger 59](#_Toc390804269)

[1.4.7 Launcher 60](#_Toc390804270)

[1.5 系统构造 61](#_Toc390804271)

[1.5.1 init 61](#_Toc390804272)

[1.5.2 property 63](#_Toc390804273)

[1.5.3 debug 67](#_Toc390804274)

[1.5.4 toolbox 68](#_Toc390804275)

[1.6 HAL 70](#_Toc390804276)

[1.6.1 HAL加载 70](#_Toc390804277)

[1.6.2 HAL构造 72](#_Toc390804278)

[1.7 JNI 76](#_Toc390804279)

[1.7.1 函数映射 76](#_Toc390804280)

[1.7.2 变量类型 77](#_Toc390804281)

[1.7.3 ENV 77](#_Toc390804282)

[1.7.4 JNI编译 77](#_Toc390804283)

[1.7.5 JNI构造 78](#_Toc390804284)

[2 Android模块 80](#_Toc390804285)

[2.1 camera 80](#_Toc390804286)

[2.1.1 framework 81](#_Toc390804287)

[2.1.2 类结构 86](#_Toc390804288)

[2.1.3 代码位置 86](#_Toc390804289)

[2.1.4 callback 87](#_Toc390804290)

[2.1.5 HAL 88](#_Toc390804291)

[2.1.6 预览（ViewFinder/Preview） 90](#_Toc390804292)

[2.1.7 拍照(takePicture) 93](#_Toc390804293)

[2.1.8 录像(Recording) 95](#_Toc390804294)

[2.1.9 虚拟 camera 96](#_Toc390804295)

[2.1.10 usb camera 96](#_Toc390804296)

[2.2 Surface 99](#_Toc390804297)

[2.2.1 功能 99](#_Toc390804298)

[2.2.2 框架结构 99](#_Toc390804299)

[2.2.3 图层 101](#_Toc390804300)

[2.2.4 使用方式 101](#_Toc390804301)

[2.2.5 Overlay 103](#_Toc390804302)

[2.2.6 HAL 104](#_Toc390804303)

[2.3 media 106](#_Toc390804304)

[2.3.1 stagefright 106](#_Toc390804305)

[2.3.2 meidaprofiles 106](#_Toc390804306)

# 系统结构

## 系统架构

### 系统架构



1. HAL(Hardware Abstraction Layer)

### AOSP结构

The Android Open Source Project (AOSP) maintains a complete software stack to be ported by OEMs and other device implementors and run on their own hardware.

|  |  |
| --- | --- |
| bionic | C库以及C运行时支持源码(libc,libm,libdl,动态linker) |
| bootable | 启动引导相关代码 |
| build | 系统编译规则,generic等基础开发包 |
| cts | Android兼容性测试套件标准 |
| dalvik | dalvik JAVA虚拟机模块源码 |
| development | 应用程序开发相关 |
| device | 设备厂商的支持源码 |
| external | 外部开源模块源码 |
| frameworks | 核心框架（java及C++实现） |
| hardware | 硬件抽象层源码 |
| libcore | 核心库，支持dalvik |
| ndk | 本地代码的开发工具，用来包含应用程序的本地代码 |
| out | 存放编译产生的文件 |
| packages | 应用程序包 |
| prebuilt | 预编译 |
| sdk | sdk及模拟器 |
| system | 文件系统库，应用及组件(C实现) |

1. NDK

Android NDK 是在SDK前面又加上了“原生”二字，即Native Development Kit，因此又被Google称为“NDK”。

在此之前，Android平台的第三方应用程序均是依靠基于Java的Dalvik特制虚拟机进行开发的。原生SDK的公布可以让开发者更加直接的接触Android系统资源，并使用传统的C或C++语言编写程序，并在程序封包文件（.apks）中直接嵌入原生库文件。

不过，Google也表示，使用原生SDK编程相比Dalvik虚拟机也有一些劣势，比如程序更加复杂，兼容性难以保障，无法访问Framework API，Debug难度更大等。开发者需要自行斟酌使用。

Android程序运行在Dalvik虚拟机中，NDK允许用户使用类似C / C++之类的原生代码语言执行部分程序。

NDK包括了：

1. 从C / C++生成原生代码库所需要的工具和build files。
2. 将一致的原生库嵌入可以在Android设备上部署的应用程序包文件（application packages files ，即.apk文件）中。
3. 支持所有未来Android平台的一些列原生系统头文件和库

## build system

### 构造规则

1. 构造方法

android的构造系统的帮助文档可参考build/core/build-system.html。



1. 设置构造环境<line 8>

source build/envsetup.sh

1. 选择构造的产品项<9>

lunch

1. 执行构造< line 10>

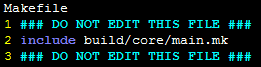
make

1. 版本信息写入< line 15~21>

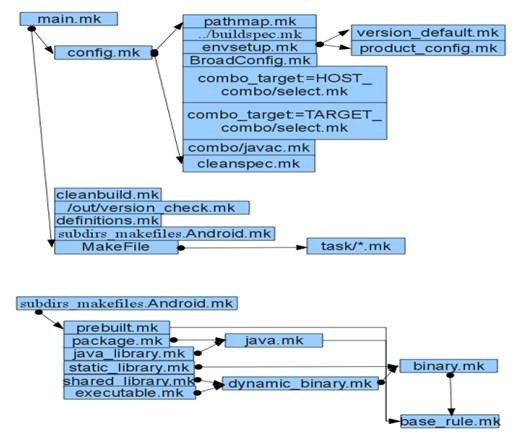
如果需要版本信息记录，可通过mount方式在生成的目标rootfs中写入版本信息。

1. 构造运行结构

直接执行make时，会解析默认Makfile。内容如下：



默认Makefile是直接加载main.mk实现对具体编译规则的引入， 加载关系如下：



1. 系统配置的构造

系统配置安装完成不同功能将配置规则定义在不同的Makeifle中，各makefile实现的功能如下。

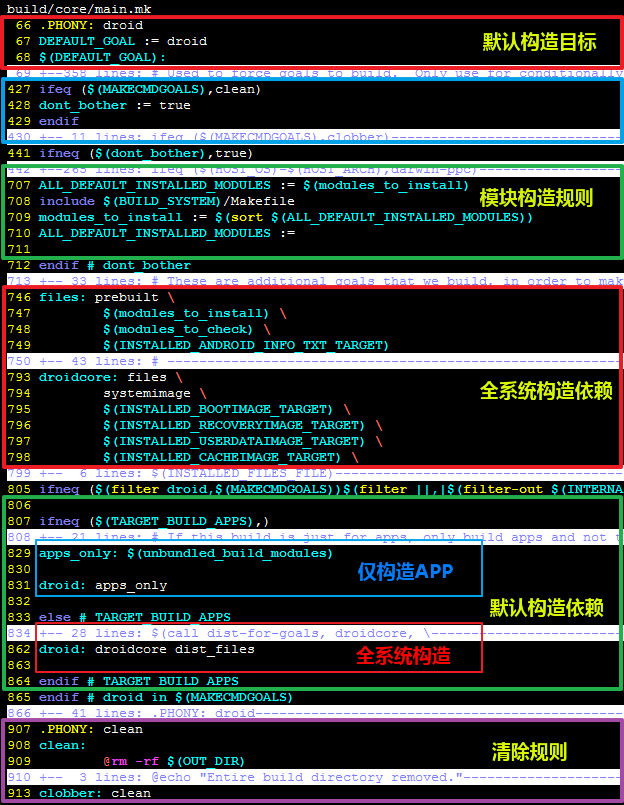
|  |  |
| --- | --- |
| 文件 | 功能 |
| help.mk |  |
| config.mk | 编译配置(在main.mk和envsetup.sh中引用) |
| versions\_checked.mk | 版本规则 |
| definitions.mk | 通用变量，函数定义 |
| dex\_preopt.mk |  |
| pdk\_config.mk | PDK编译规则 |

定义方式如下：



1. 构造规则的组成

当make不同目标时，则会使用该目标对应的规则对目标进行构造。通过对目标解析，实现不同目的的构造，实现如下：



其中各目标或变量意义如下：

|  |  |
| --- | --- |
| 目标/变量 | 含义 |
| droid | 默认构造目标 |
| droidcore | 构造android系统模块的目标 |
| dist\_files |  |
| apps\_only | 构造应的用目标 |
| files | 构造android系统依赖的文件 |
| modules\_to\_install | 需要构造的模块文件列表 |
| systemimage | 系统镜像 |
| clean | 清除系统的目标规则 |

1. NDK构造

NDK的构造规则位于ndk/build目录下。

cd ndk/build/tools

export ANDROID\_NDK\_ROOT=NDK源码的目录

./make-release –help

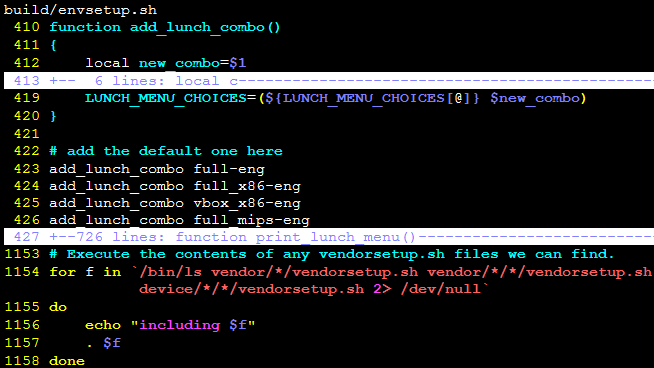
1. 构造过程中用到的工具

|  |  |
| --- | --- |
| 命令名 | 功能 |
| acp | 复制文件 |
| adbs | 查看crash问题 |
| apicheck | 检测API兼容性，合法性(framework/base/api/) |
| apriori | 实现prelink(bionic/linker/README.TXT) |
| atree | 按照xxx.atree文件的描述对文件进行操作 |
| buildinfo.sh | 创建xxx.prop文件 |
| check\_builds.sh | 查看发布版本之间变化 |
| check\_prereq | 用于设备ota升级 |
| compare\_fileslist.py | 版本比较 |
| dump-package-stats | 查看jar/apk文件信息 |
| event\_log\_tags.py | 处理event-log-tags的内容 |
| fileslist.py | 列出指定目录下所有文件及大小 |
| findleaves.py | 在指定目录中查找指定文件 |
| fixlinebreaks.sh | 把windows中的换行改为linux |
| fs\_config | 列出指定文件(夹)的权限 |
| fs\_get\_stats | 获取文件的简单stats |
| kcm | key character map |
| mktarball.sh | 打包 |
| signapk | 对jar包签名 |
| soslim | 用于编译 |
| warn.py | 解析系统编译log |

### 配置流程

系统配置是通过使用source命令运行 build/envsetup.sh实现。执行后，脚本中定义的函数将导入到当前环境，比如：hmm，croot，lunch(设置编译环境)，mm(在模块所在目录下编译)，mmm(使用模块所在路径作为参数进行编译)等。实现如下：

1. 增加产品项



1. <423~426>增加系统默认的产品项
2. <1154~1158>增加所有厂家的产品项

在默认目录(vendor和device) 下搜索所有厂家的设置脚本vendorsetup.sh，并使用source命令(缩写为“.”符号)执行。通过调用add\_lunch\_combo方法添加不同厂家的不同产品。如下增加了2各产品项：



1. 产品项的格式

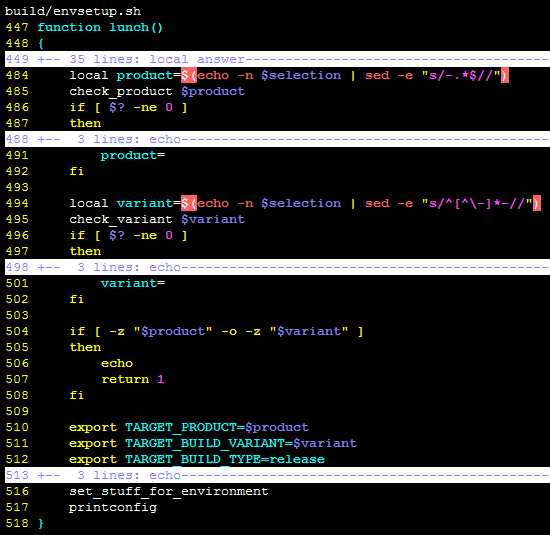
产品项的描述格式为xxx-yyy，其中xxx表示产品名称(product)，yyy表示产品类型(variant)。比如：full-eng表示产品名称为full，版本为eng。

产品名称可自由定义，产品类型定义如下：

|  |  |
| --- | --- |
| variant | 含义 |
| eng | 工程机 |
| user | 最终用户机 |
| userdebug | 调试测试机 |
| tests | 测试机 |
| optional | 所有版本 |

1. 选择构造的产品项

使用lunch方法可实现从已添加的产品项中(通过add\_lunch\_combo增加)选择某一个作为当前构造的产品。并进行相应的配置，实现如下：



1. 获取产品项的名称和版本

其中<484>行用于将脚本传入的参数selection进行解析，比如为nt72668-eng时，则product为nt72668，variant为eng.

最后导出产品编译配置：

TARGET\_PRODUCT=xxx

TARGET\_BUILD\_VARIANT=yyy

TARGET\_SIMULATOR=false

TARGET\_BUILD\_TYPE=release

其中：

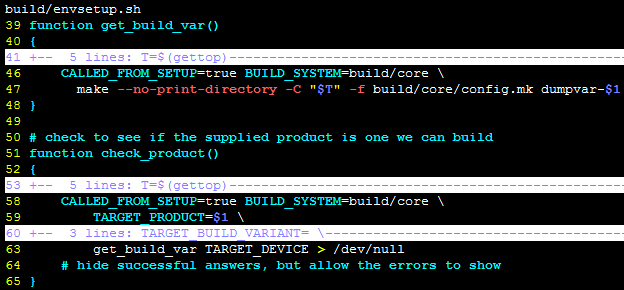
当模块makefile中LOCAL\_MODULE\_TAGS的定义与当前选择的产品版本TARGET\_BUILD\_VARIANT相同时该模块才进行构造，否则不进行构造，一般该值指定为optional(表示任何版本)，如下：



1. 获取产品项的的设备类型

lunch中调用check\_product方法实现对当前选择的产品类型进行相关环境变量的设置，从而获取TARGET\_DEVICE变量的值。

通过make build/core/config.mk实现系统配置的载入，并通过构造dump-%目标实现对系统配置的获取，方法如下：

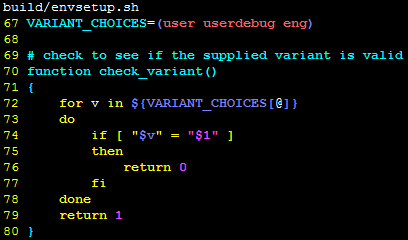


<58>设置TARGET\_PRODUCT为当前选择的产品项名称。

<63>调用get\_build\_var获取当前产品项对应的TARGET\_DEVICE的值，<41>get\_build\_var通过make dumpvar-XXX获取XXX变量的值。

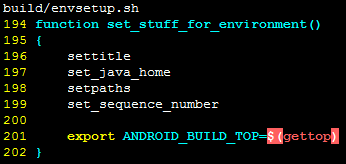
1. 检测版本类型合法性

lunch中调用check\_variant方法实现对选中的产品版本进行合法性检测，实现如下，即检测版本类型是否在可选类型user ,userdebug,eng中。



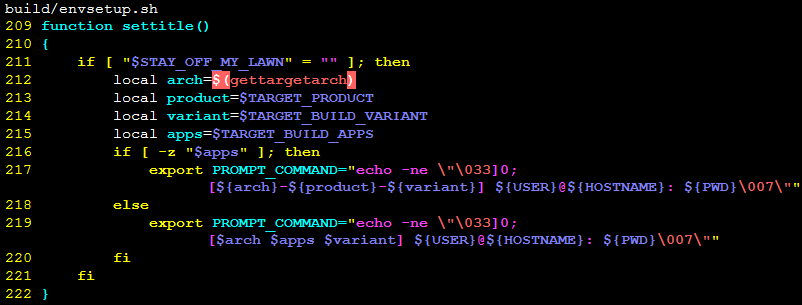
1. 配置系统环境

当产品项对应的产品类型确定，则调用set\_stuff\_for\_environment方法实现对系统运行环境的配置。

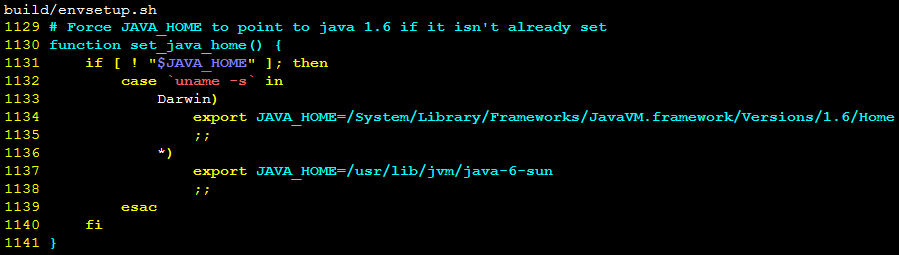


1. 设置命令行调用跟踪命令

通过settitle实现对PROMPT\_COMMAND变量设置实现对命令调用的跟踪，则在shell中调用命令时同时会伴随此命令的调用。这里设置的跟踪命令如下：

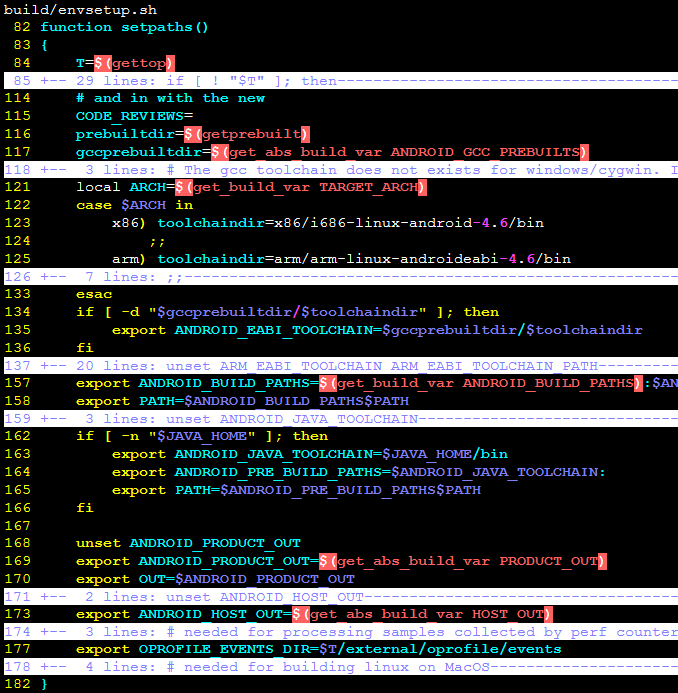
1

1. 设置Java vm的目录



1. 设置编译工具目录

主要定义了交叉编译工具的路径，以及java vm的路径，如下：

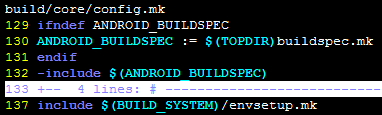


### 编译配置

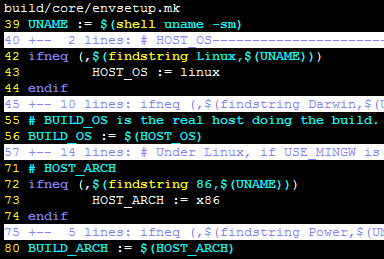
编译配置规则定义在config.mk文件中，该文件在main.mk和envsetup.sh都有引用，用于获取系统的编译配置。其实现的配置如下：

1. 编译环境配置

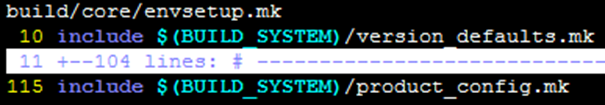
在config.mk文件加载了envsetup.mk，该makefile定义了编译环境的处理规则。



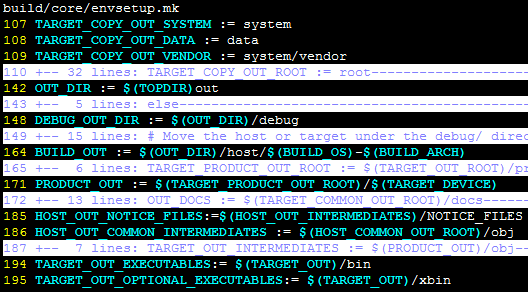
1. HOST 编译环境配置



1. 产品配置 (详见产品配置章节)



1. 编译目录配置



1. 目标设备的编译输出目录配置

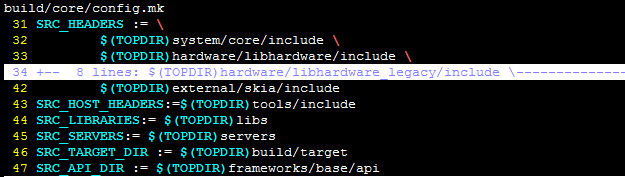
|  |  |  |  |
| --- | --- | --- | --- |
| 目标设备编译输出：**一级目录** out/target/product/xxxx | | | |
| **二级目录** | | **三级目录** | |
| /system | TARGET\_OUT | \_EXECUTABLES | /bin |
| /system/vendor | TARGET\_OUT\_VENDOR | \_OPTIONAL\_EXECUTABLES | /xbin |
| /data | TARGET\_OUT\_DATA | \_SHARED\_LIBRARIES | /lib |
|  |  | \_JAVA\_LIBRARIES | /framework |
|  |  | \_APPS | /app |
|  |  | \_ETC | /etc |
| /root | TARGET\_ROOT\_OUT | \_BIN | /bin |
|  |  | \_SBIN | /sbin |
|  |  | \_ETC | /etc |
|  |  | \_USR | /usr |
| /sysloader | TARGET\_SYSLOADER | \_OUT | / |
| /installer | TARGET\_INSTALLER | \_ROOT\_OUT | /root |
|  |  | \_SYSTEM\_OUT | /root/system |
|  |  | \_DATA\_OUT | /data |
|  |  | \_OUT | / |

1. 主机端编译输出目录配置

|  |  |  |  |
| --- | --- | --- | --- |
| 主机设备编译输出：**一级目录** out/host | | | |
| **二级目录** | | **三级目录** | |
| /linux-x86 | HOST\_OUT | \_EXECUTABLES | /bin |
| /linux-mips |  | \_SHARED\_LIBRARIES | /lib |
| **…** |  | \_JAVA\_LIBRARIES | /framework |
|  |  | \_SDK\_ADDON | /sdk\_addon |
|  |  | \_INTERMEDIATES | /obj |
|  |  | \_HEADERS | /obj/include |
|  |  | \_INTERMEDIATE\_LIBRARIES | /obj/lib |

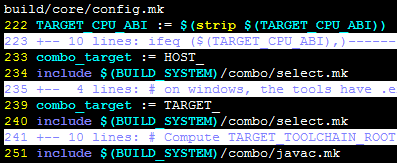
1. 基础模块配置

各模块在构造时可能会引用基础模块中一些源码，常用的基础模块如下：

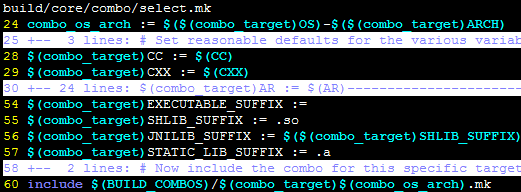


1. 编译工具配置

编译工具分为HOST以及TARGET两大类。在配置TARGET的编译工具前需要保证当前产品的板级配置(详见产品配置章节)已经确定。



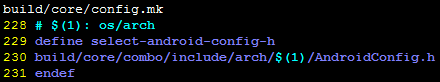
1. 编译工具配置文件(通用)



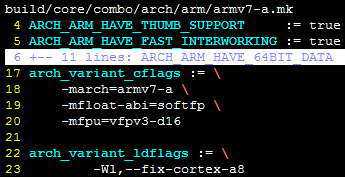
1. 不同平台类型的编译选项(以ARM平台为例)



其中获取android配置头文件AndroidConfig.h：

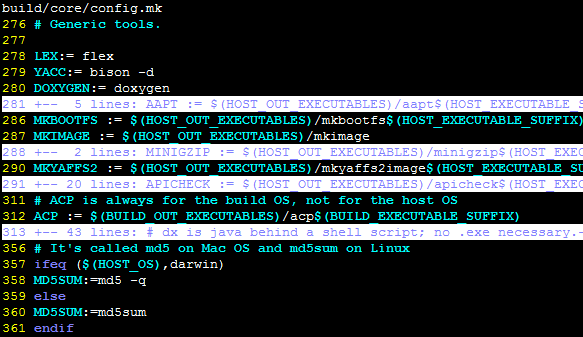


1. 不同架构的编译选项(以armv7架构为例)



1. 通用工具配置

通用工具属于HOST端在构造相关模块时使用的工具，在系统构造前首先会构造他们，其路径配置如下：



其中：

MKBOOTFS用于记录根文件系统的文件列表；

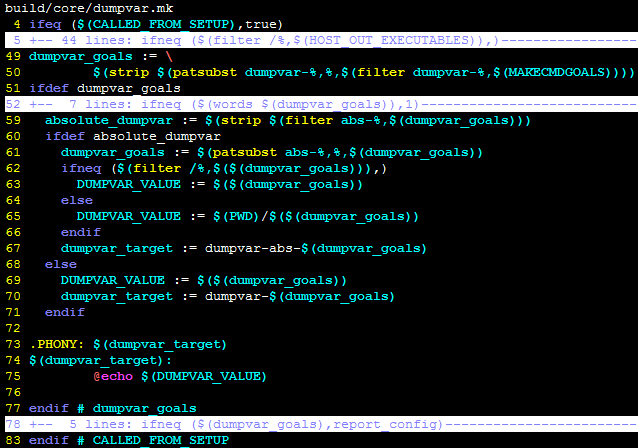
MKIMAGE用于将根文件系统的文件列表转化为ramdisk。

1. 打印配置项的值

当编译配置后，可通过dump-%规则可获取对应配置的值，该规则定义在dumpvar.mk文件，通过在config.mk中include该文件实现。



dumpvar-XXX规则定义如下：

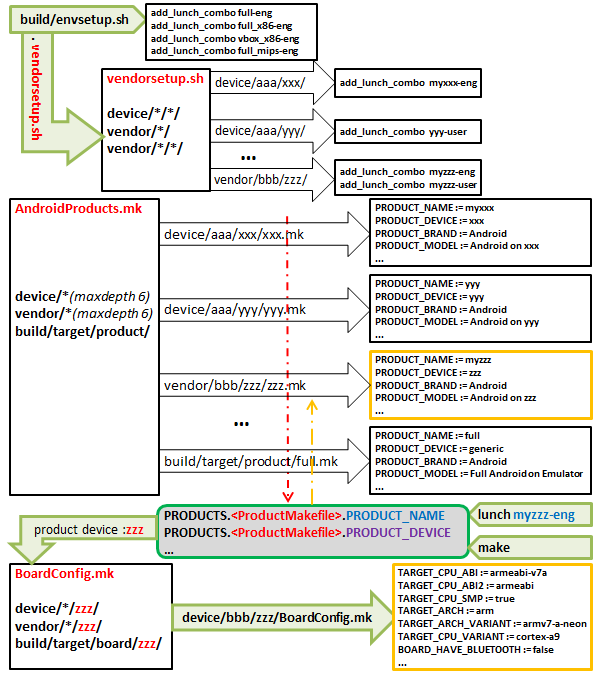


执行后打印DUMPVAR\_VALUE变量，即目标名对应的变量值。

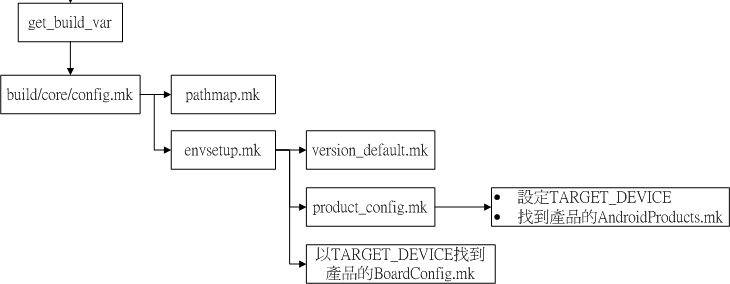
比如：对于dumpvar-TARGET\_DEVICE则输出TARGET\_DEVICE对应的值。

### 产品配置

产品模块是针对特定产品的模块，在进行编译配置前需要先完成产品配置，产品配置的流程如下：



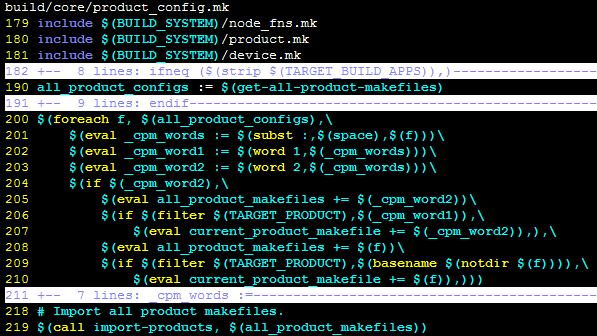
产品配置规则定义在product\_config.mk中，通过将该文件加载到编译环境配置envsetup.mk中实现对产品的置的获取，其在编译配置的位置如下：



实现步骤如下：

1. 载入所有产品配置文件

所有产品的默认配置文件均为AndroidProducts.mk，通过将所有产品的配置文件加载，从而得到将所有产品的配置信息。实现如下：

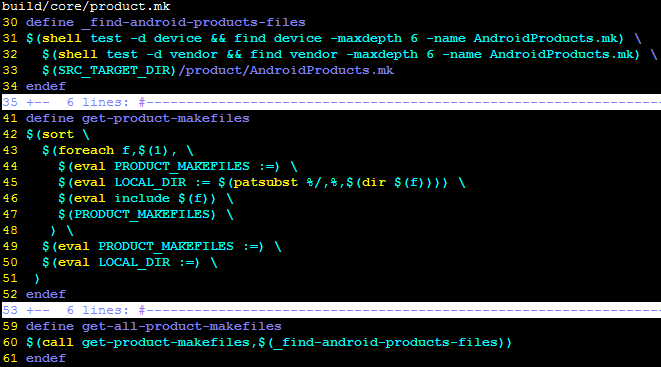


1. 载入产品配置的通用规则

<179~181>将定义了产品配置规则的makefile包含到product\_config.mk。

1. 获取所有产品的配置文件

<190>get-product-makefiles用于获取默认的product配置文件AndroidProducts.mk并将其加载，其实现位于product.mk文件中，如下：



<31~32>：查找所有默认的product配置文件，通过搜索verdor，device，以及build/target目录，得到所有名为AndroidProducts.mk的文件。

<42~51>：加载所有AndroidProducts.mk，其中在加载该文件前需要对列表变量PRODUCT\_MAKEFILES清空，该变量用于记录定义了产品配置的Makefile文件列表，具体定义在AndroidProducts.mk中实现。

1. 配置特定产品的参数

每个产品需要将其配置信息定义到存在于PRODUCT\_MAKEFILES变量中的makefile。从而实现配置定义的规则导入实现。其加载到默认的产品配置文件实现对其所在模块的配置。

1. 设置产品配置文件

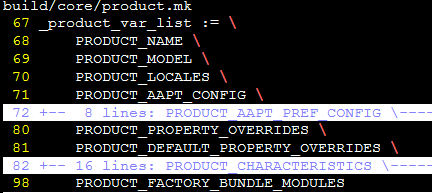
在默认配置的product配置文件中将定义产品配置的文件加入PRODUCT\_MAKEFILES文件列表中，如下：



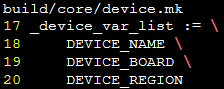
1. 需要定义的产品配置项

不同产品在实现各自产品配置时，需要对一系列描述产品的配置项进行设置，每个配置项对应一个变量，这些变量的定义必需在PRODUCT\_MAKEFILES文件中。标准配置项对应的变量名列表如下：

产品相关的配置项：

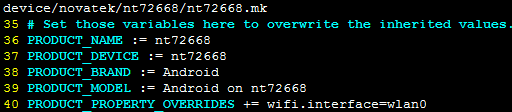


设备相关的配置项：



1. 定义产品配置项

PRODUCT\_MAKEFILES中一般需要定义的最小配置项如下：

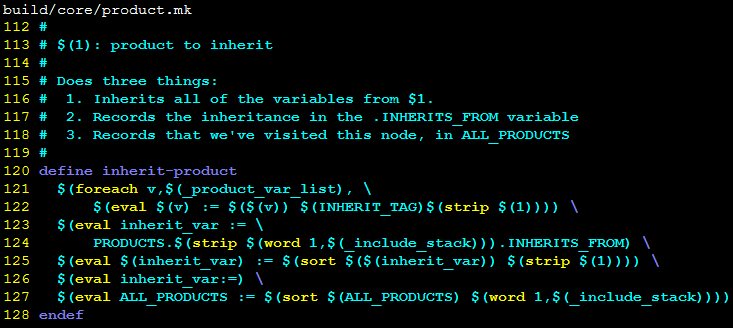


1. 配置项继承赋值

除过直接在PRODUCT\_MAKEFILES中定义配置外，也可以通过调用inherit-product方法增加定义在其他文件的产品配置项。其配置项的值形如“变量值 @inherit:文件”的继承赋值格式。



继承变量的实现如下：



<121~122>将变量的值增加继承标识：变量名:= $(**变量对应的值** @inherit:变量所在文件路径)，如果所在文件不存在变量定义，则对应值部分为空，只剩@inherit:变量所在文件。

<123~126>更新变量列表inherit\_var，每次调用将增加一个列表项，其值为PRODUCTS.文件路径.INHERITS\_FROM。

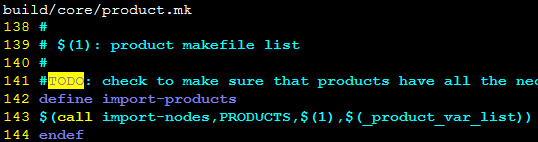
<127>更新文件列表ALL\_PRODUCTS，每次调用将增加一个列表项，其值为文件路径。

1. 创建所有产品的配置信息

通过对所有产品的配置文件进行解析，将配置信息按照不同产品项进行分类，并创建新变量，用于全局范围内对不同产品的配置项进行引用。

1. 产品配置项信息收集

在product\_config.mk中调用import-products方法创建PRODUCTS列表和一系列形如“PRODUCTS.配置文件路径.参数名”的配置变量。实现如下：



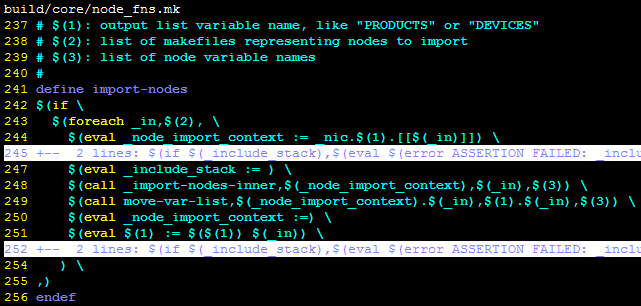
<143>$1为待解析的配置文件， \_product\_var-list为需要提取的变量名列表(import-nodes的实现参考配置规则章节)。

PRODUCTS列表用于记录所有产品的配置定义文件(即PRODUCT\_MAKEFILES).

配置变量则用于记录不同产品的配置。实现了将不同产品的配置按照其定义所在文件进行分类。

1. 创建特定产品配置项变量的方法

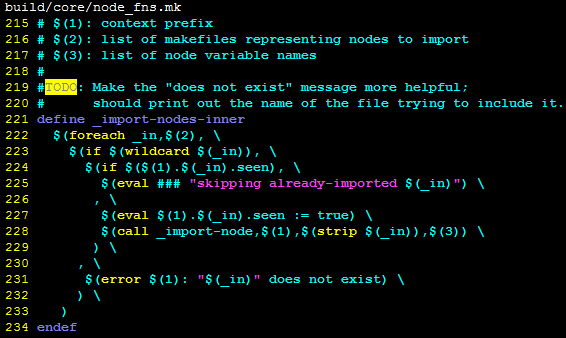
通过参数中传入的node列表名($1)，待解析的产品配置文件($2)，变量名列表($3)，实现变量列表(“列表名.变量名”)，和配置文件列表(“列表名”)的创建：



<244>\_node\_import\_context := **\_nic.列表名.[[文件]]**

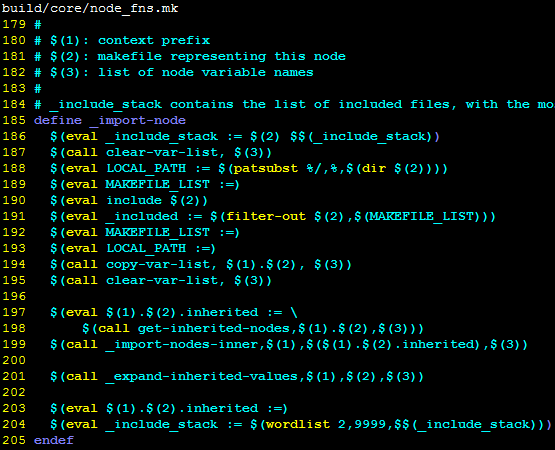
<247>清空\_include\_stack(用于记录import\_node正在处理的文件)

<248>导入node



对导入node的文件创建一个名为“**\_nic.列表名.[[文件]]**.**文件**.seen”变量并设置为true，如果该文件没有进行过导入，则执行\_import\_node，参数为：**\_nic.列表名.[[文件]], 文件, 变量列表**。

<228>递归查找node并导入



<186>将文件（makefile文件）放到\_include\_stack列表(记录该函数递归调用)

<187>将变量的值置为空，即:“变量 := ”

<188>获取文件路径

<189>清除make记录的makefile路径

**<190>载入文件**

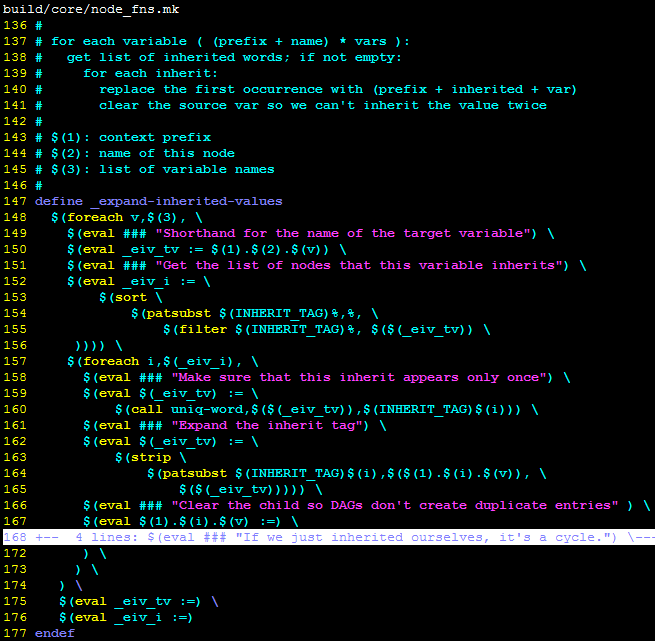
<191>获取该文件中include的其他makefile

<194>创建临时变量并赋值：**\_nic.列表名.[[文件]].文件.变量:= 变量对应的值**，其中**变量的值**定义在对应配置文件中。

<197>如果<**194>**新建变量**对应的值**包含以“@inherit:”为前缀的字符串(该前缀后为makefile文件路径)，则将**对应的值**去掉前缀以及之后的文件路径，只将值放入 **\_nic.列表名.[[文件]] .文件.inherited**列表，即通过inherit-product方法更新的某mk文件中变量对应的值。

**<199>递归执行\_import-nodes-inner直到**<224>条件成立，即**\_nic.列表名.[[文件]]**.**文件**.seen为真则停止递归**。**

<201>展开node并去除重复inherit的变量。通过双循环进行消除重复。



<150>\_eiv\_tv变量名初始化为 **\_nic.列表名.[[文件]] .文件.**变量。

<152>\_eiv\_i:= [\_eiv\_tv某个变量名对应值中的“@inherit](mailto:.文件.变量对应的值@inherit):”后的文件路径列表，即某个变量名的继承变量在多个配置文件中。

<157~173> 查询所有配置文件，去除@inherit变量名中对应值相同的变量名。即如果两个继承变量对应的值相同，则只保留一个。

<159>使用uniq-word方法将\_eiv\_tv变量现有的值(第一个参数) 逐个在所有含有继承变量的配置文件(第二个参数)中进行比较，如果配置文件中该变量没有定义或者定义为空(相同个数为0或1)，则使用\_eiv\_tv变量中现有的值。如果配置文件中该变量已经定义为非空(即文件相同，值也相同)，则将值合并起来。

<162>将当前\_eiv\_tv变量值替换为uniq-word处理后的值。

<249>修改变量名前缀

将临时变量的前缀**\_nic.列表名.[[文件]].文件**替换为**列表名.文件**，即创建类似“**列表名**.**文件**.变量”的新变量。

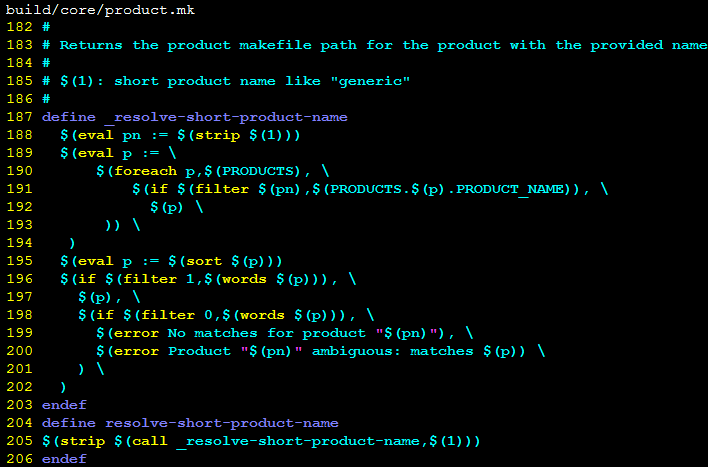
<251>将解析的文件放入列表变量中，列表变量为**列表名**，比如PRODUCTS。

1. 获取当前产品的配置信息

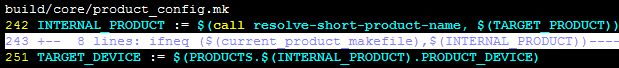
当所有产品的配置变量创建后，则使用resolve-short-product-name方法获取指定产品名所对应的配置文件，并记录在INTERNAL\_PRODUCT变量中，然后通过配置文件可获取指定产品的其他配置信息。

1. 型获取对应Product的配置文件

通过传入product的名称从而获取其配置文件的路径。



1. 获取当前构造的产品其对应的设备类型



<242>通过Product的配置信息TARGET\_PRODUCT(在lunch时选择确定的)的值得到该产品所在配置文件路径。

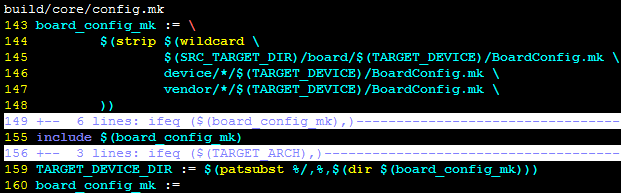
<251>通过配置文件路径获取到指定产品的其他配置(此处为device配置)。

1. 对当前产品进行板级配置

在获取到当前构造的产品配置后，则通过当前产品的TARGET\_DEVICE配置完成对应的板级配置。

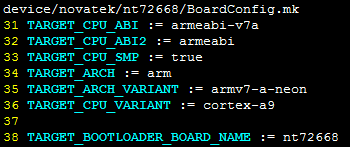
1. 加载板级配置

默认的板级配置文件为BoardConfig.mk，其位于device或vender目录下以并当前PRODUCT\_DEVICE的值命名的子目录下。



1. 板级配置的实现

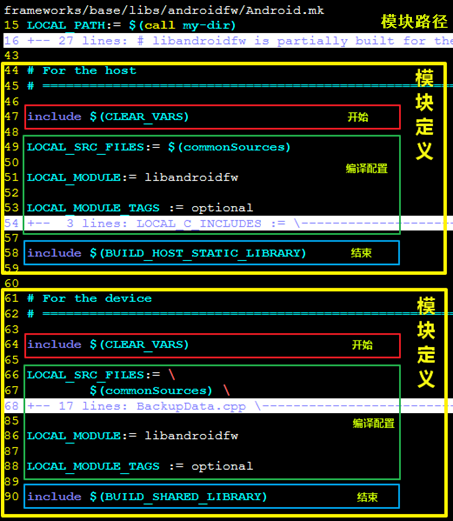
板级配置主要配置一些用于决定当前编译工具的信息，如下：



### 模块配置

1. 模块定义

每个module的定义以include $(CLEAR\_VARS)开始，以include(BUILD\_XXX) 结束(XXX由模块类型确定)。其中一个makefile中可以定义多个模块。

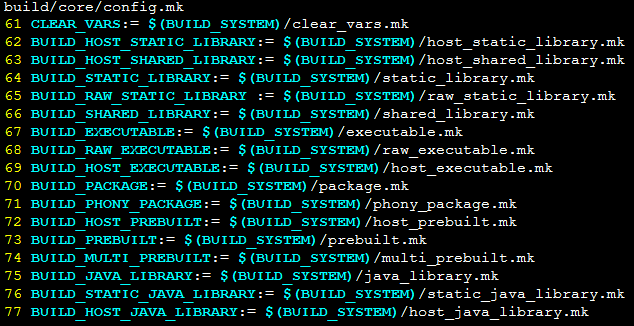


相关配置如下：

|  |  |
| --- | --- |
| 需要配置的变量 | 含义 |
| LOCAL\_PATH | 模块路径(必需定义在CLEAR\_VARS 之前) |
| LOCAL\_UNINSTALLABLE\_MODULE | 模块不需要安装(不定义则表示需要安装) |
| LOCAL\_MODULE\_PATH | 模块安装路径 |
| LOCAL\_MODULE | 模块名称 |
| LOCAL\_MODULE\_TAGS | 模块的版本类型(即lunch选择的variant) |
| LOCAL\_SRC\_FILES | 模块需要加入编译的源码文件 |
| LOCAL\_GENERATED\_SOURCES | 模块配置中自动生成的文件(动态创建的文件) |
| LOCAL\_C\_INCLUDES | 编译时默认头文件路径 |
| LOCAL\_CFLAGS | 编译时加入的cflag |
| LOCAL\_LDLIBS | 模块依赖的库 |
| LOCAL\_COPY\_HEADERS | 模块源码目录下需要复制的头**文件** |
| LOCAL\_COPY\_HEADERS\_TO | 模块编译目录下存放头文件的**子目录** |

1. 不同类型模块的编译规则

针对不同类型的模块将构造规则定义在不同的makefile文件中，各具体的模块通过include方式将其对应类型的规则载入从而实现本模块的编译，如下：



1. 模块构造规则分类

模块配置规则分散定义在不同makefile中，各模块的Makeifle通过将其依赖的规则加入自己的规则中实现具体模块的编译，如下(依赖方向🡪)：

|  |  |  |  |
| --- | --- | --- | --- |
| 模块类型 | | C/ Java语言 | 基础规则 |
| host\_executable.mk | | binary.mk | base\_rule.mk |
| executable.mk | dynamic\_binary.mk |
| shared\_library.mk |
| static\_library.mk | |
| package.mk | | java.mk |
| java\_library.mk | |

1. 各构造规则适用的模块类型

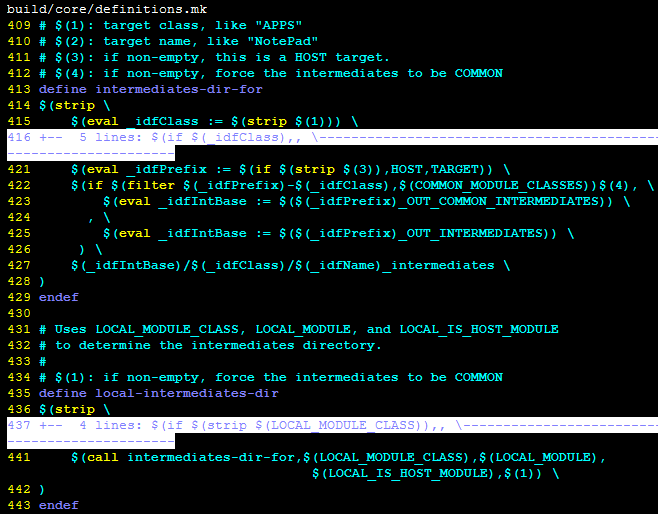
|  |  |
| --- | --- |
| Makefile文件 | 功能 |
| CLEAR\_VAR | 清除配置，每个模块定义开始必须加载该规则 |
| BUILD\_HOST\_XXX | 编译host 上运行的模块，没有则表示target |
| BUILD\_XXX\_STATIC\_LIBRARY | 编译静态库模块 |
| BUILD\_XXX\_SHARED\_LIBARY | 编译共享库模块 |
| BUILD\_XXX\_EXECUTABLE | 编译可执行文件模块 |
| BUILD\_XXX\_JAVA | 编译java指令类型模块 |
| BUILD\_PACKAGE | 编译apk模块 |
| BUILD\_XXX\_PREBUILT | 预编译规则 |

1. 模块编译中间结果输出目录

通过模块类型，模块名，host/target类型，是否为commm模块最终确定输出目录。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 一级目录 | | 二级目录 | 三级目录 |
| 参数 | 是否  HOST | 是否通用模块 | 模块类型 | 模块名 |
| 目录名 | HOST\_ | OUT\_COMMON\_INTERMEDIATES | SHARED\_LIBRARIES | 模块名\_intermediates |
| STATIC\_LIBRARIES |
| TARGET\_ | OUT\_INTERMEDIATES | EXECUTABLES |
| JAVA\_LIBRARIES |

使用local-intermediates-dir方法可获取当前模块中间结果的输出目录，该方法会自动获取模块的本地配置来确定输出目录。具体如下：

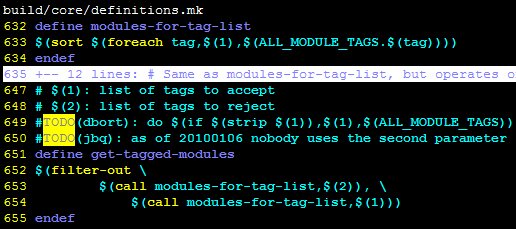


<422>如果通用模块列表中包含名为“HOST-模块名 (TARGET-模块名) ”的文件，或者参数指定输出为通用模块，则使用通用模块的路径。即：如果传入了一个参数则一定为通用模块，如果无参数传入且通用模块列表中也没有则表示输出到非通用模块。

1. 通用模块列表

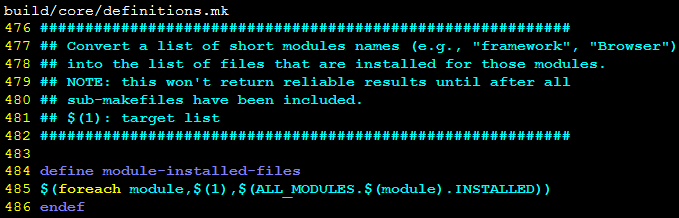


1. 模块安装路径
2. 获取指定版本的模块

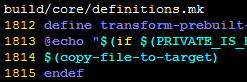


1. 获取模块对应的配置文件

通过模块名获取对应的模块文件。

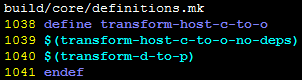


1. 预处理模块安装

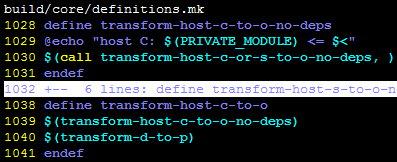


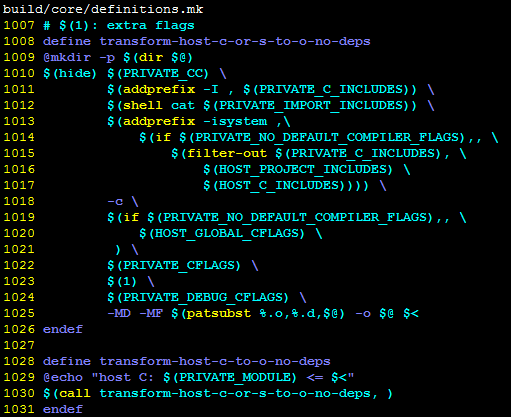
1. 模块编译命令
2. c源码编译为obj

使用transform-host-c-to-o用于将模块编译为对应平台的object文件。



编译源文件

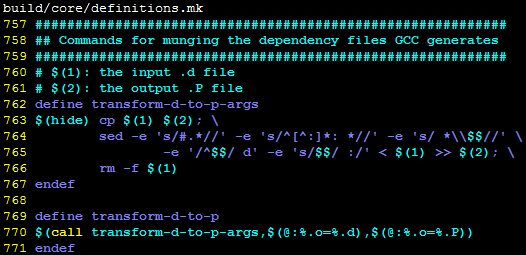




<1010>通过模块配置的编译器来创建object。

<1025>-MD –MF表示同时输出依赖关系文件到$@同名的.d文件。

创建依赖文件



<763>将每个object文件编译生成的依赖.d复制一份为.P。

<764>将.d文件具有某些特征的行删除，附加到.P文件之后。

清除“#”开始的字串，即删除“#test”之类的字串；

清除非“:”起始，后跟“:”和任意空格的文本，即删除“test:: ”之类的字串；

清除行尾为任意空格后为“$”结束的字串；

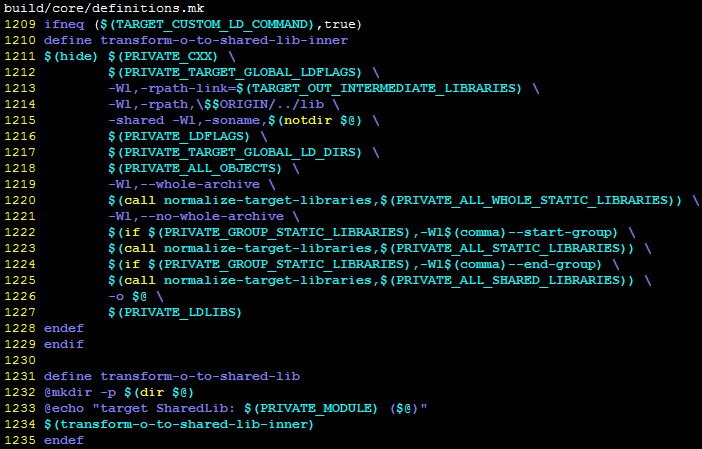
删除空行；

在行末加入“:”

<766>删除.d文件。

1. obj链接为共享库

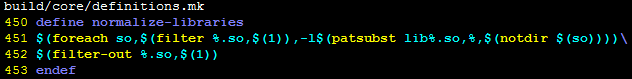
使用transform-o-to-shared-lib用于将模块编译的中间object文件链接为共享库库。



<1213>指定进行编译连接时其依赖的库所在目录。

<1214>指定运行时load查找依赖库所在模块。

<1223>normalize-target-libraries用于指定链接库的参数-lxxx，实现如下：



### 系统预构造

1. 预构造模块的makefile

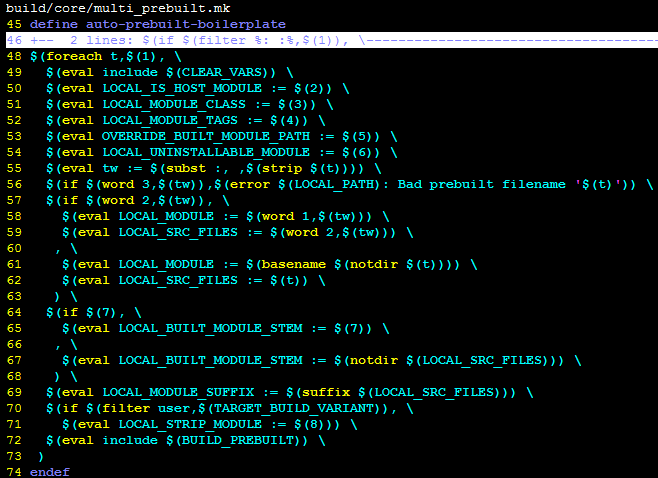
针对是否host，预编译模块使用的规则如下：

|  |  |
| --- | --- |
| host预编译规则 | target预编译规则 |
| host\_prebuilt.mk | prebuilt.mk |
| multi\_prebuilt.mk(按类型分派) |  |
| prebuilt.mk |  |

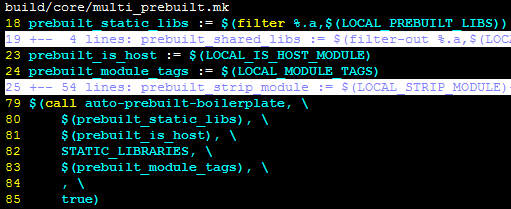
1. host模块规则



1. 增加多种类型的预构造模块



比如：增加静态连接库的实现：

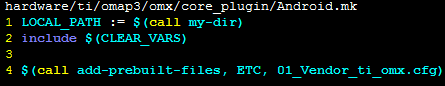


1. 安装预构造文件

有些文件已经存在于当前源码中，则可以直接安装，其增加方法如下：

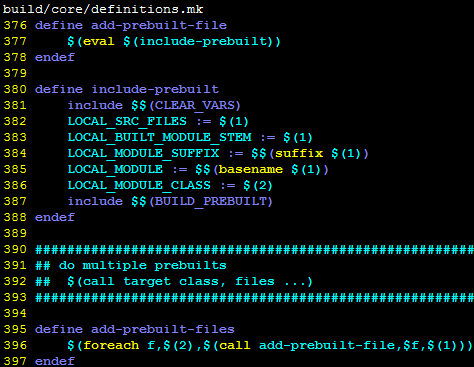
1. 增加预构造文件的方法

模块调用add-prebuilt-files则可将传入的文件安装到对应类型的目录。比如：



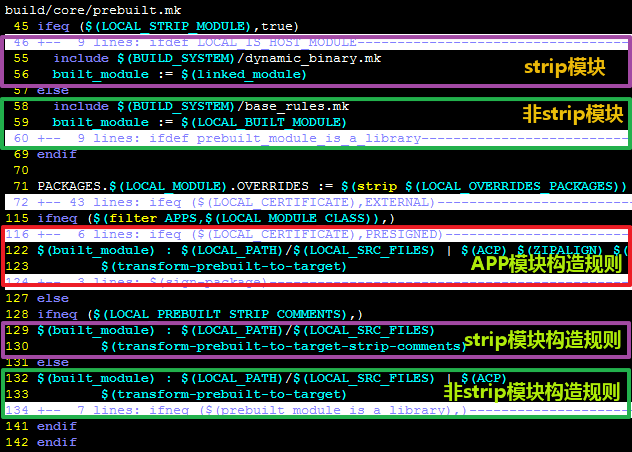
<4>将01\_Vendor\_ti\_omx.cfg文件安装到ETC类型对应的目录下。

1. 增加预构造文件的规则



<396>将$2中的文件安装到$1对应的目录。

1. 安装规则的实现

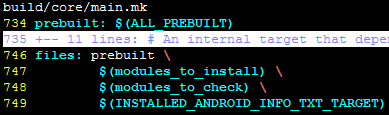


1. 安装预构造模块

在构造系统前部分模块需要提前构造，然后进行安装，其安装方法如下：

1. 增加预构造模块

对于需要优先构造的模块，可通过将其加入ALL\_PREBUILT列表实现。



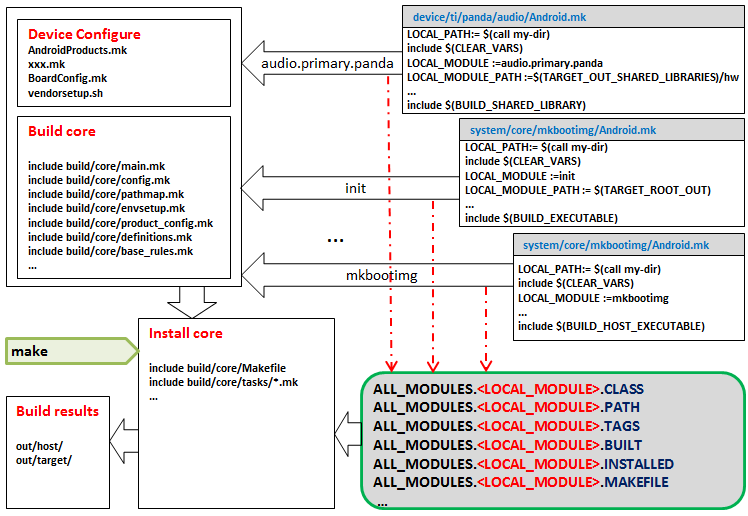
1. 安装规则的实现

需要由模块自身的makefile实现，通过调用transform-prebuilt-to-target实现复制到目标目录。



### 多模块构造

多模块构造时，各makefile通过include方式加载到一起，其运行结构如下：



构造步骤如下：

1. 加载各模块的构造规则

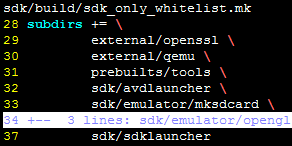
模块的构造规则以及配置规则定义在其自身的配置文件(Android.mk)中，当执行make时，需要先加载各模块的makefile，步骤如下：

1. 获取各模块所在目录

将需要编译的模块路径加入subdir列表变量中，如下：

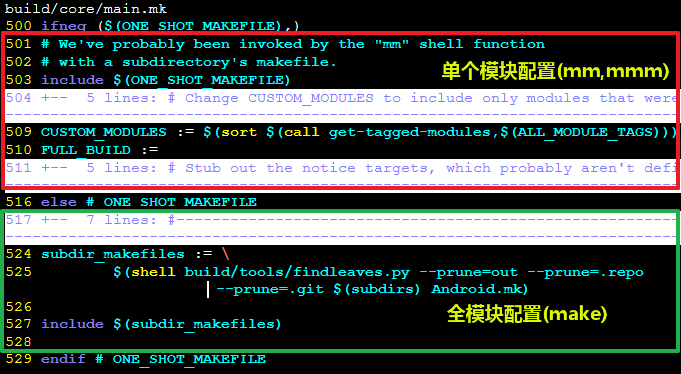


<454>SDK模块的路径配置定义在sdk/目录下，比如：



1. 加载各模块的Makefile

当配置了subdirs后，则加载目录下的Makefile。



<503>加载从mm，mmm命令传入的ONE\_SHOT\_MAKEFILES。

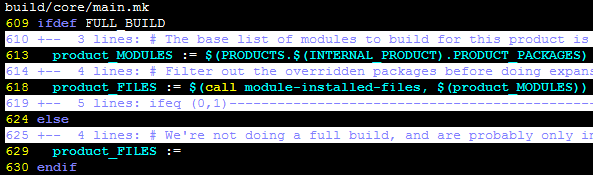
<524~527>从subdir加载Android.mk文件(详见模块构造规则定义章节)。

1. 添加指定产品模块

当进行全编译时，会重新构造系统特有的产品模块。

1. 产品模块的获取

通过从产品配置中获取到INTERNAL\_PRODUCT值从所有产品的注册信息中获取到当前产品的模块列表。

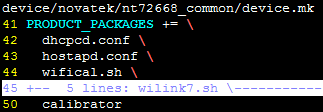


<613>从产品模块列表中获取当前产品特有的模块。

<618>筛选出产品模块中**仅需要构造**的模块，并加入product\_FILES。其中需要构造的模块记录在ALL\_MODULES.$(模块ID).**INSTALLED**列表中。

1. 增加产品模块的方法

对于特定产品的模块需要将模块ID或模文件名添加到PRODUCT\_PACKAGES列表，如下：

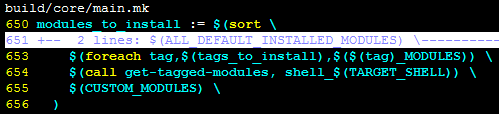


<42~44>这几个文件属于直接安装，但不需构造的模块，。

<50>需要构造的模块ID，其信息在模块注册时会记录在**INSTALLED**列表。

1. 获取所有需要构造的模块

当构造模块时，则将与当前构造的版本相同的模块加入构造列表，如下：



<654>将从ALL\_MODULE\_TAGS.$(tag)列表中筛选出指定版本的模块。

1. 加载单模块构造规则

单个模块的构造规则定义在其自身的配置文件，当构造modules\_to\_install中各个模块时，则会使用对应的规则进行构造，。

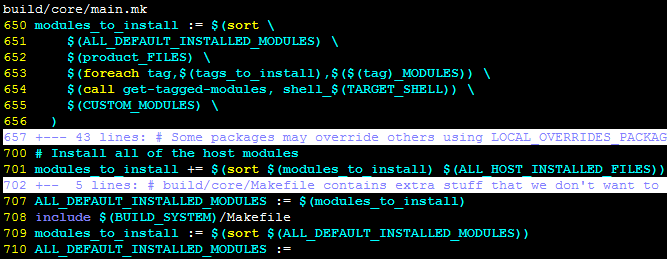


各模块分属于不同构造层次，上层模块在构造前需要先构造其依赖的底层的块，层次关系如下(低层模块🡪上层模块)：

Arch(arm,x86) 🡪 Board(ti,samsung) 🡪  Device  🡪 Product

1. 加载系统构造规则

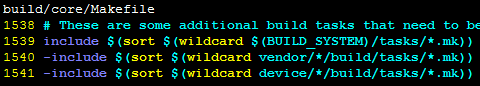
当所有存在modules\_to\_install中的模块构造后，则进行最终的系统构造，其规则定义在Makefile中。如下：



<707>ALL\_DEFAULT\_INSTALLED\_MODULES为已构造完成的模块列表，该列表在系统构造时用于构造最终的系统(详见系统构造)。

1. 加载其他模块构造规则

当所有模块构造后，会构造系统的辅助模块，比如cts，api兼容性检测等。



makfiel的功能：

|  |  |
| --- | --- |
| makefile | 功能 |
| apicheck.mk | 判断api是否符合AOSP的规范 |
| cts.mk | 构造cts测试命令(make cts)，编译的命令位于HOST\_OUT目录 |
| ide.mk | IDE开发环境 |

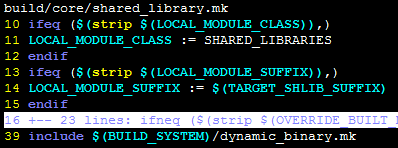
### 单模块构造

1. 设置模块输出文件路径

模块文件名由输出路径和文件名组成，文件又由STEM和SUFFIX组合而成。以共享库的规则为例，如下：

1. 设置模块文件的类型和后缀

如果模块配置中没有定义后缀LOCAL\_MODULE\_SUFFIX，则使用默认后缀(与TARGET\_SHLIB\_SUFFIX配置相同)。



后面部分定义在dynamic\_binary.mk中。

1. 设置模块文件名(不含路径)

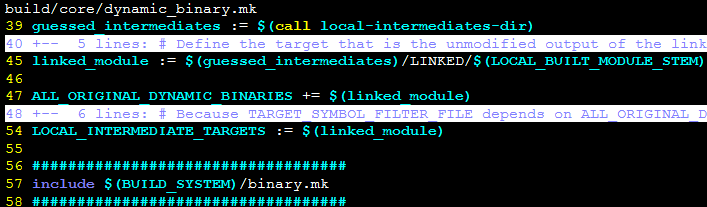
如果模块配置中没有定义模块对应的文件名(不带后缀)LOCAL\_MODULE\_STEM，则使用默认文件名（与模块名相同）。

模块的文件名(带后缀全名) LOCAL\_INSTALLED\_MODULE\_STEM最终由STEM和SUFFIX组合。



1. 设置模块中间生成文件的输出目录

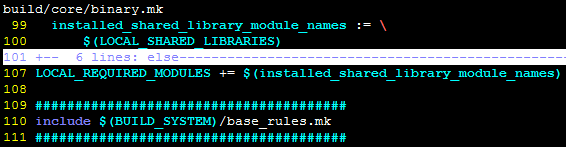
使用local-intermediates-dir方法获取模块的输出路径，并在该目录下建立子目录LINKED(链接名与STEM相同)。



<47>更新共享库列表。

<54>初始化LOCAL\_INTERMEDIATE\_TARGETS为模块链接名，其构造规则定义在binary.mk中。

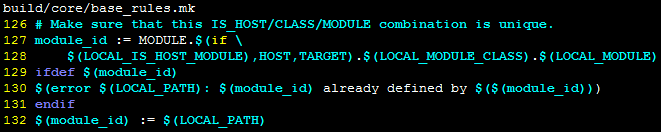
1. 创建模块信息



1. 模块唯一性检测

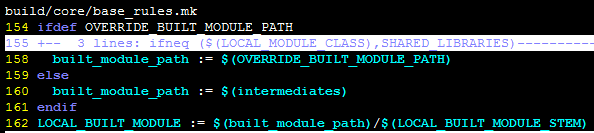
通过将每个模块的配置信息组合，并将该组合定义为一个新的变量，当发现该变量重复定义时说明模块也被重复定义。

该变量名为<模块运行端类型>.<模块文件类型>.<模块ID>，实现如下：



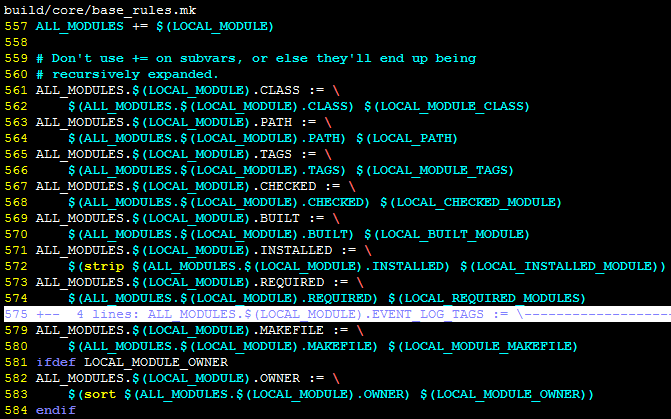
1. 设置模块编译后的路径(intermediate directory)

模块编译路径和安装路径一般不一样，编译后的路径一般会复制到安装路径。



1. 注册模块信息

通过创建以LOCAL\_MODULE为索引的变量实现。

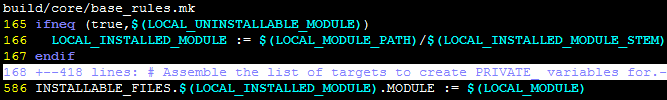


ALL\_MODULES是记录所有模块ID的列表。

以ALL\_MODULES.$LOCAL\_MODULE为前缀的变量用于记录该模块的信息，各信息的功能如下。

|  |  |
| --- | --- |
| 模块信息名 | 功能 |
| CLASS | 文件类型 |
| PATH | 源码所在的目录 |
| TAGS | 编译类型 |
| BUILT | 编译后路径，即intermediate directory |
| INSTALLED | 安装路径，即复制BUILT的路径 |

1. 创建模块的安装信息(final path)



<165>如果模块需要安装，则创建LOCAL\_INSTALLED\_MODULE变量。

<586>更新用于记录所有安装模块ID的列表。

其中LOCAL\_MODULE\_PATH为模块安装的目录，比如：

对于目标设备的模块：

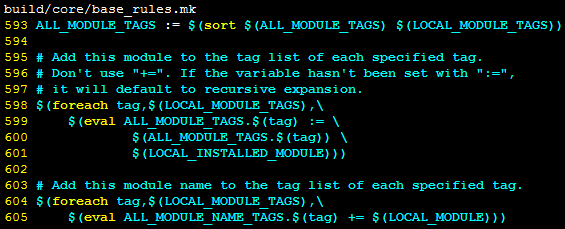


则模块最终会复制到根文件系统的system/lib/hw目录下。

对于主机端的模块：

1. 模块版本分类

通过模块配置的tag将其分类到不同模块列表，以便于在多模块构造时筛选出指定版本的模块。

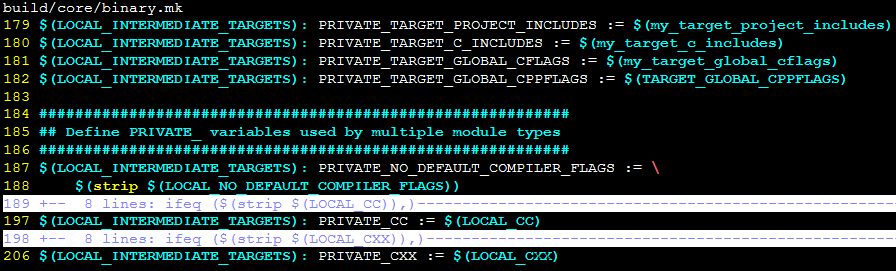


<599>创建以tag分类的模块安装名称列表，并加入当前模块的**安装名称**。

<605>创建以tag分类的模块ID表，并加入当前模块的ID。

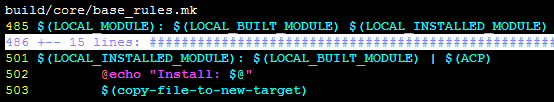
1. 配置模块编译参数

定义PRIVATE\_前缀的模式变量，设置模块编译需要的工具，以及参数，保证这些配置只在构造本地模块时有效。



1. 编译规则定义

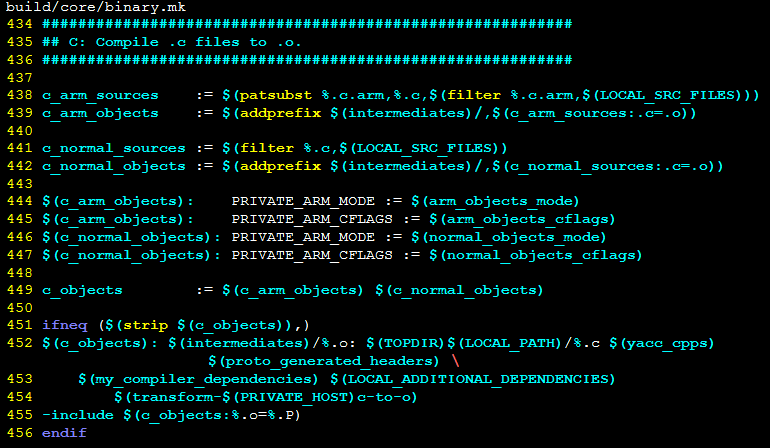
在系统构造时，其ALL\_DEFAULT\_INSTALLED\_MODULES中包含的模块的编译规则如下：



在系统构造时，其依赖的LOCAL\_BUILT\_MODULE需要先完成构造，构造步骤如下。

1. 源文件构造object规则

从模块配置的源码文件列表中获取其对应的obj文件，不同源文件构造obj的规则不同，其中c语言产生的.o规则如下。

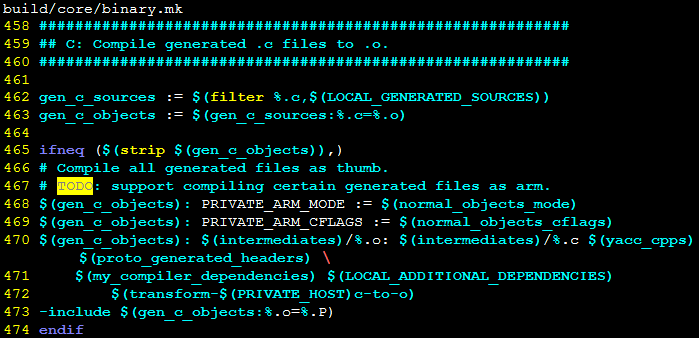


<438>LOCAL\_SRC\_FILES表示源文件在模块目录下已经存在定义。

<454>编译工具调用，该工具通过调用PRIVATE\_CC实现obj和依赖文件的生产。

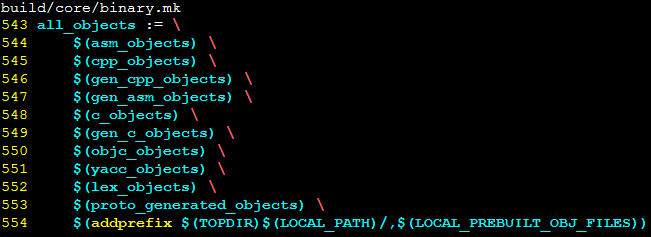
1. 自动生成的源文件构造object规则

如果源文件是在LOCAL\_GENERATED\_SOURCES中定义的，则表示源文件是在编译模块时自动生成的。自动生成的文件构造需要在源文件编译后，再编译到模块中，如下：



1. 获取编译过程中的obj文件

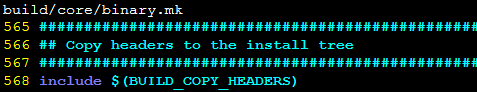
all\_objects中存放当前模块在编译所有的obj文件。



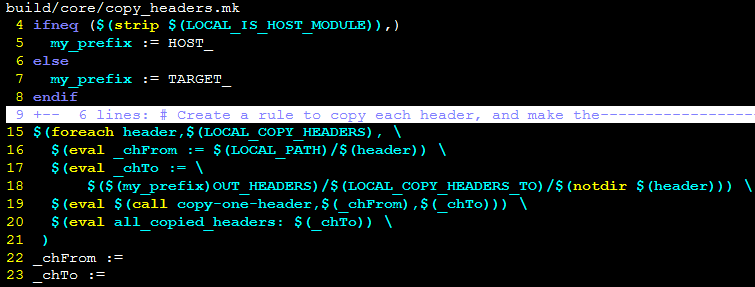
其中，gen\_xxx\_objects前缀的object表示源文件与xxx\_objects源文件类型相同，但是自动生成的。

1. 复制模块的头文件

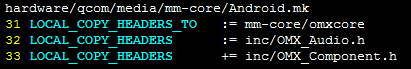
在模块构造时，如果模块的某些头文件在构造其他模块时需要被引用，则可将这些头文件定义到LOCAL\_COPY\_HEADERS列表中。这些文将件复制到安装目录的头文件目录(TARGET\_OUT\_HEADERS)下的LOCAL\_COPY\_HEADERS\_TO子目录。



实现如下：

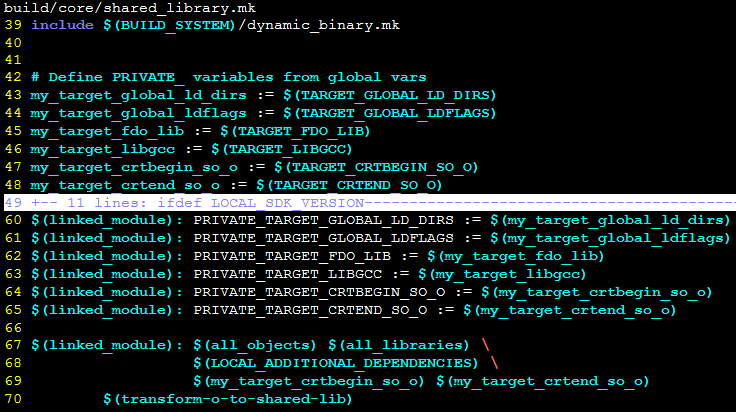


增加复制头文件的方法：



1. 模块构造规则

当模块的源文件对应的object构造完成后，则生成最终的模块文件。

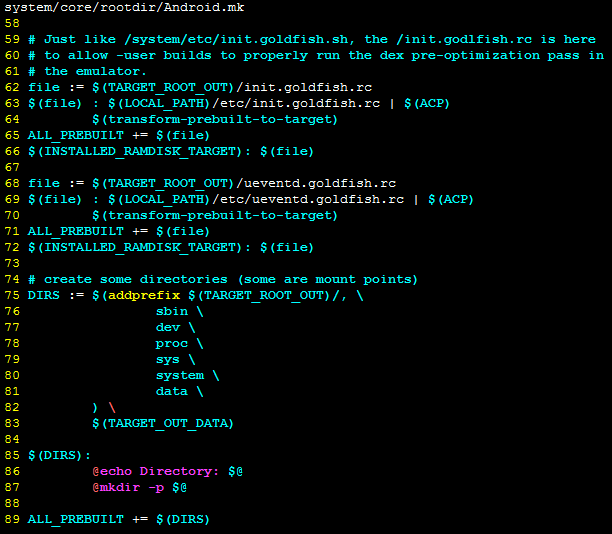


<70> 使用transform-o-to-shared-lib将obj链接为一个模块。

### 系统盘构造

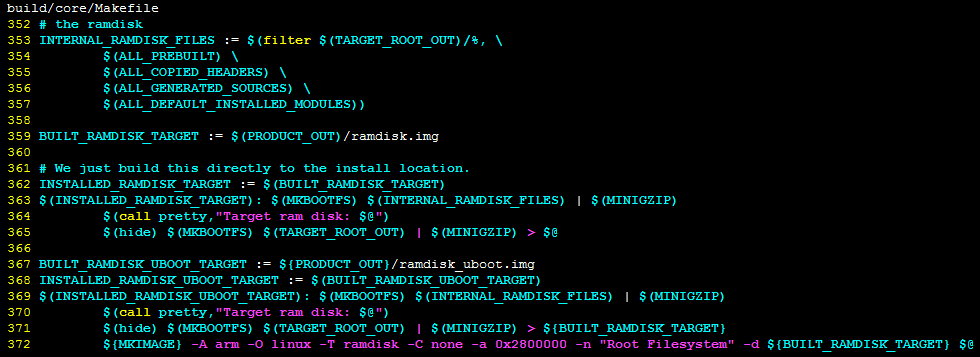
1. root文件系统构造

root文件系统给系统提供基础的linux的根文件系统。其实际对应于编译输出目录下的root文件夹。



1. ramdisk构造

root文件系统会制作为ramdisk。



其中：

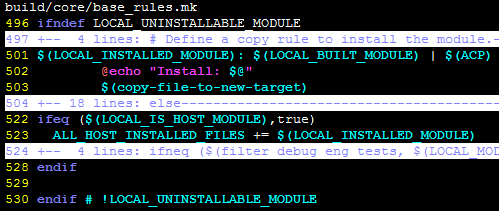
<365,371>用于创建root目录下的文件路径列表。

<372>用于将对应的文件路径列表制作为ramdisk.

### 系统安装

1. 安装已构造的模块

当各模块编译完后，则开始构造INSTALLED\_MODULE，通过将编译后的模块复制到配置的目录下完成安装，复制方法如下：



<501> BUILT\_MODULE为模块编译完成后的结果，INSTALLED\_MODULE为需要安装的模块。

<503>直接复制第一个依赖到目标位置。

<523>更新主机端的安装模块列表。

1. 安装产品模块的依赖文件

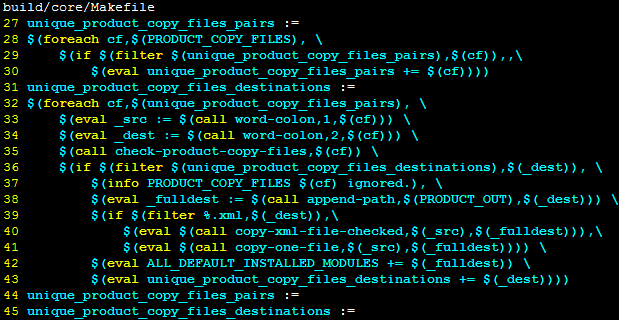
部分模块运行时需要一些依赖文件，这些文件一般放在模块对应的目录下，当构造模块时可将其复制到根文件系统目录下。

1. 增加复制文件的方法

通过将文件增加到PRODUCT\_COPY\_FILES列表中即可实现，格式为“源文件：目标路径”，如下：



1. 复制规则

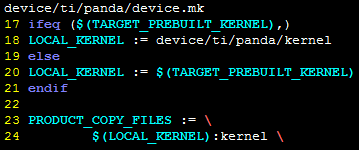


<28>去除PRODUCT\_COPY\_FILES列表中重复项。

<32>复制PRODUCT\_COPY\_FILES列表中文件。

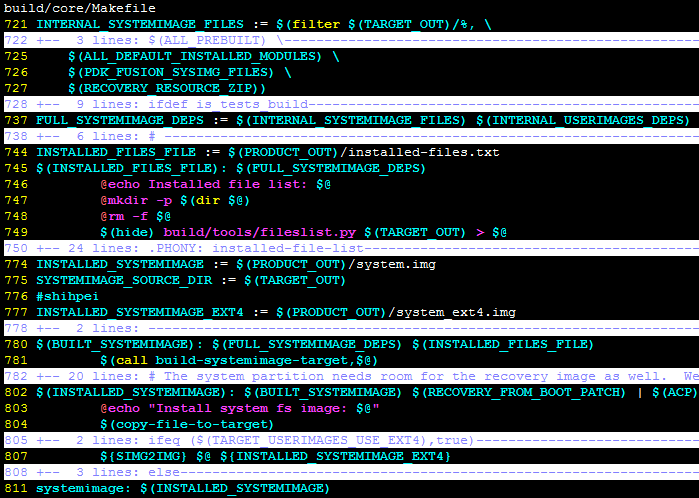
1. 安装内核

因为内核的构造是独立于android的，其安装的方法与安装模块依赖文件的方法相同，如下：



1. 构造系统镜像

当所有模块编译完后，则会将安装好的模块制作为系统镜像用于目标设备的烧录或升级。



需要构造的依赖模块

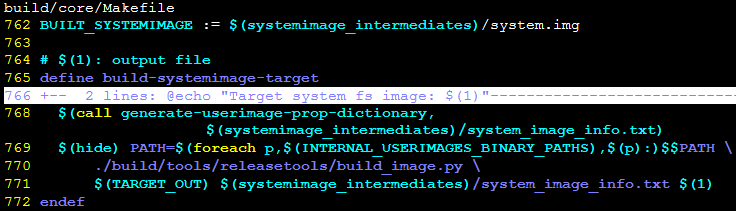
systemimage🡪BUILT\_SYSTEMIMAGE<780：系统镜像>

🡪 FULL\_SYSTEMIMAGE\_DEPS<737：镜像依赖的文件>

🡪 INTERNAL\_SYSTEMIMAGE\_FILES<721：列表>

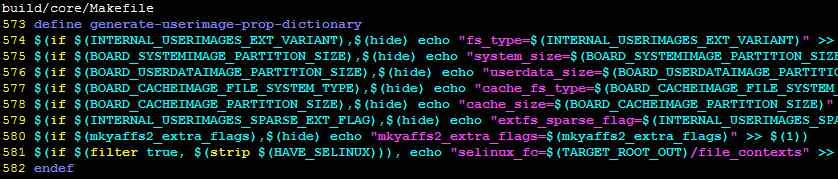
🡪ALL\_DEFAULT\_INSTALLED\_MODULES(725:模块文件)

系统镜像是将系统依赖的所有文件制作为文件系统镜像，制作方法如下：



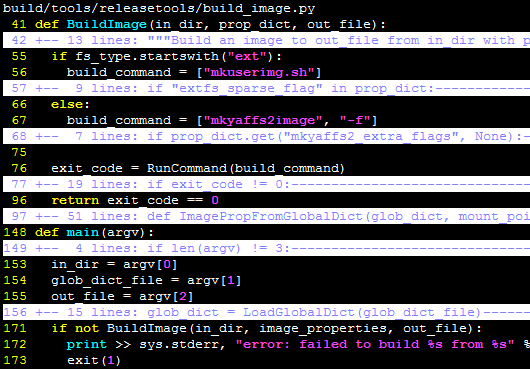
1. 创建镜像配置文件<768>

将制作镜像的参数写入system\_image\_info.txt文件。具体实现如下：



1. 制作镜像的实现<770>

调用build\_image.py将TARGET\_OUT(即system目录)作为镜像文件的源目录，将system\_image\_info.txt作为镜像参数文件,实现镜像文件BUILT\_SYSTEMIMAGE的制作，如下：



<55~67>根据镜像类型使用mkuserimg.sh或mkyaffs2img程序完成镜像制作。

1. 安装系统映像导入<804>

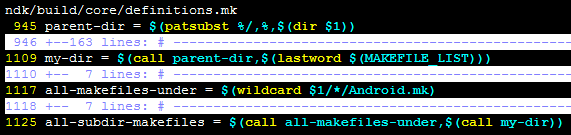
将制作好的BUILT\_SYSTEMIMAGE复制到INSTALLED\_SYSTEMIMAGE。

1. 镜像导入<807>

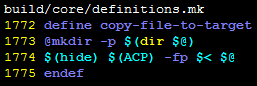
将INSTALLED\_SYSTEMIMAGE导入INSTALLED\_SYSTEMIMAGE\_EXT4中，实现system.img写入system\_ext4.img。

### 通用规则

1. 文件处理
2. 常用路径

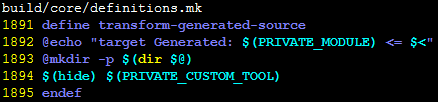


1. 文件复制(将$<复制到$@目录下，即将第一个依赖复制到目标)

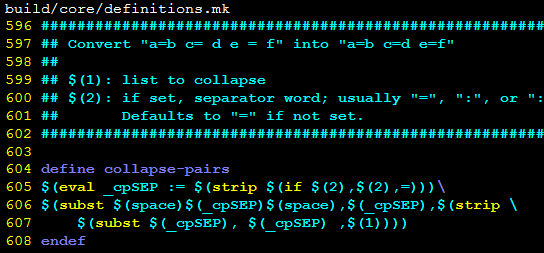


1. 使用用户工具复制

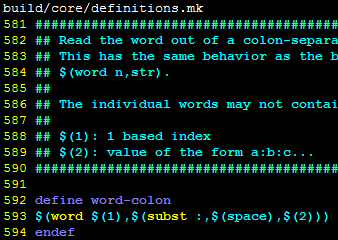
在执行前需要配置用户工具。



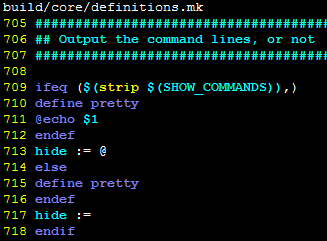
1. 字符串处理
2. 赋值语句去空格



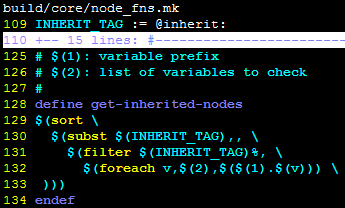
1. colon字符串分割



1. 杂项
2. 命令输出控制

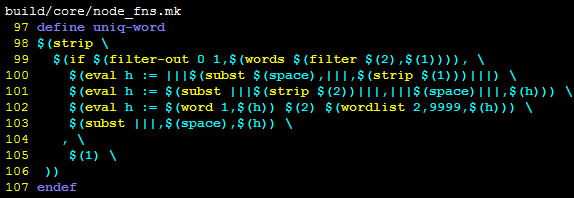


1. node处理规则
2. inherit定义



如果$(1).加在$(2)某个变量名后，其对应新变量的值如果前缀为“@inherit:”，则去掉新变量对应值的前缀并返回。

1. 去除重复



<99>获取$(1)与$(2)的相同的部分,计算相同部分单词(空格分开)的个数，如果只有一个相同或者没有任何相同(小于2)则仅保留$(1)。如果相同单词个数>=2，则继续。(假设$1为value file, $2为value file)。

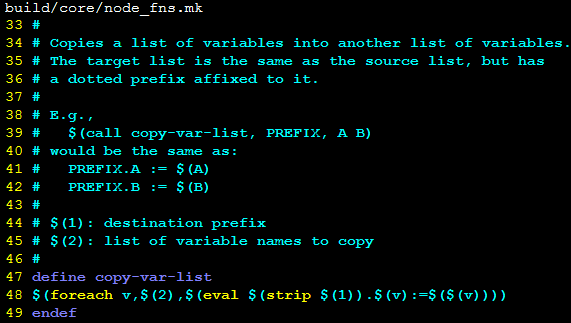
<100>将$(1)各单词之间的空白分隔改为||| value ||| file |||。

<101>将和$(2) (即||| value file |||)中相同的部分替换为空格。

<102>提取替换后的第一个单词和$(2)和第二个之后单词。

<103>去掉|||。(最后还是重复的 file value file value，？？？)

1. 复制变量



创建新变量：前缀.变量 := 变量对应的值

1. 移动变量



创建新变量：新前缀.变量 := 旧前缀.变量 对应的值，并清除旧前缀.变量。

1. 镜像制作
2. 镜像导入

通过simg2img可将镜像写入目标设备中。

使用方法：

mkdir sys

/simg2img system.img sys.raw

sudo mount -t ext4 -o loop sys.raw sys/

Then you have all your system partition mouned in 'sys' and you can modify whatever you want in 'sys'. For example de-odex apks and framework jars.

## 系统结构

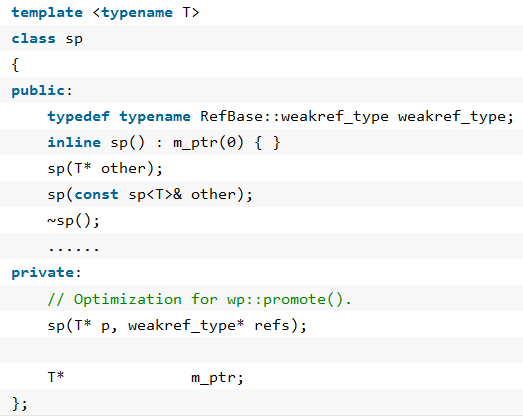
### sp

指针类是对一个普通指针的封装，通过RefBase类维护该指针的引用计数，当引用为零时则自动释放该指针指向的内存。

1. 指针类分为 sp(strong pointer), wp(weak point)，sp和wp都是一个模板类，使用sp<T>或者wp<T>, 类T必需满足：
2. 该类是虚基类RefBase的子类或间接子类
3. 该类必须定义虚析构函数。如virtual ~MyClass();
4. 源码位置：

frameworks/base/include/utils/RefBase.h

frameworks/base/libs/utils/RefBase.cpp



1. wp<T>类型不能直接对类型T进行操作，要想对T进行某种操作，必需把wp升级为sp指针，使用promote()来实现升级：

wp<T> weakp= new T();

sp<T> t = weakp.promote();

1. wp可能会在弱引用计数不为0的情况下被销毁，如果T使用extendObjectLifetime(OBJECT\_LIFETIME\_WEAK) 改变生命周期，则只有当强引用和弱引用的计数都为0时，T对象才会被销毁。

## 运行结构

### ServiceManger

### goldfish

Goldfish是一个虚拟cpu，是一种ARM处理器。Android模拟器通过运行它来运行arm926t指令集（arm926t属于armv5构架）。

它的核心内容存放在：arch/arm/mach-goldfish,默认编译配置文件为： goldfish\_defconfig，编译生成的linux内核镜像在android的模拟器(qemu)中使用。启动模拟器时，Linux Kernel镜像默认使用：prebuilt/android-arm/kernel目录下的kernel-qemu文件。

### Dalvik

### Ashmem

Ashmem的含义为：Anonymous Shared Memory 匿名共享内存，通过内核的机制，为用户空间程序提供分配内存的机制。

1. Ashmem设备节点名称为/dev/ashmem，主设备号为10（Misc Driver），次设备号动态生成。
2. Ashmem的代码路径：

kernel/include/linux/ashmem.h

kernel/mm/ashmem.c

1. 在用户空间C libutil库对Asheme封装并提供接口：

system/core/include/cutils/ashmem.h

system/core/libcutils/ashmem-dev.c

system/core/libcutils/ashmem-host.c

### [Binder](http://mynvt/deptsites/20101227A/A001/Shared%20Documents/Android/System/Android_Binder.ppt)

1. 内核驱动

Android的Binder驱动程序为用户层程序提供了IPC（进程间通信）的支持，Android整个系统的运行依赖Binder驱动。

1. Binder设备节点名称为/dev/binder，主设备号为10（Misc Driver），次设备号动态生成。
2. binder的代码路径：

kernel/include/linux/binder.h

kernel/drivers/misc/binder.c

1. 在用户空间libutil工具库和ServiceManager守护进程调用Binder接口提供对整个系统的支持：

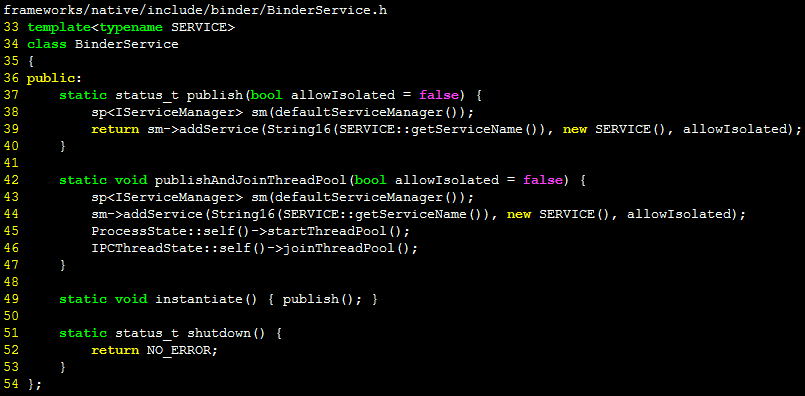
frameworks/base/cmds/servicemanager/

frameworks/base/include/utils/

frameworks/base/libs/utils/

1. BinderService

模板类，Service通过继承BinderService实现和client的交互。



### Logger

Android的Logger驱动程序为用户层程序提供Log的支持，这个驱动作为

一个工具来使用。

1. Logger有三个设备节点：

/dev/log/main

/dev/log/event

/dev/log/radio

主设备号为10（Misc Driver）

次设备号动态生成

1. Logger驱动的代码路径：

kernel/include/linux/logger.h

kernel/drivers/misc/logger.c

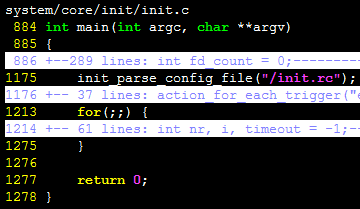
1. 在用户空间logcat程序调用Logger驱动：

system/core/logcat/

### Launcher

## 系统构造

### init



init.rc主要定义了两种类型的事件。

1. services

服务是指那些需要在系统初始化时就启动或退出时自动重启的程序.

1. 语法：

service <name> <pathname> [ <argument> ]\*

<option>

<option>

...

1. options

选项是用来修改服务的。它们影响如何及何时运行这个服务.

|  |  |
| --- | --- |
| 选项 | 描述 |
| critical | 据设备相关的关键服务，如果在4分钟内，此服务重复启动了4次，那么设备将会重启进入还原模式。 |
| disabled | 服务不会自动运行，必须显式地通过服务器来启动。 |
| setenv <name> <value> | 设置环境变量 |
| socket <name> <type> <perm> [ <user> [ <group> ] ] | 在/dev/socket/下创建一个unix domain的socket，并传递创建的文件描述符fd给服务进程.  type必须为dgram/stream/seqpacket.  用户名和组名默认为0 |
| user <username> | 在执行此服务之前先切换用户名。当前默认为root. |
| group <groupname> [ <groupname> ]\* | 类似于user,切换组名 |
| oneshot | 当此服务退出时不会自动重启. |
| class <name> | 给服务指定一个类属,这样方便操作多个服务同时启动或停止.默认情况下为default. |
| onrestart | 当服务重启时执行一条指令， |

1. action

动作表示了一组命令(commands)组成.动作包含一个触发器，决定了何时执行这个动作。当触发器的条件满足时，这个动作会被加入到已被执行的队列尾。如果此动作在队列中已经存在，那么它将不会执行.

1. 语法：

on <trigger>

<command>

<command>

…

1. trigger

触发器用来描述一个触发条件，当这个触发条件满足时可以执行动作.

|  |  |
| --- | --- |
| 触发器 | 描述 |
| boot | 当init程序执行，并载入/init.conf文件时触发. |
| <name>=<value> | 当属性名对应的值设置为指定值时触发. |
| device-added-<path> | 当添加设备时触发. |
| device-removed-<path> | 当设备移除时触发. |
| service-exited-<name> | 当指定的服务退出时触发 |

1. command

表示可执行程序。

### [property](http://blog.csdn.net/androiddeveloper_lee/article/details/6625267)

属性系统作为android的一个服务运行，用于管理系统配置和状态。所有这些配置和状态都是属性。每个属性是一个键值对（key/value pair），其类型都是字符串。

从功能上看，属性与windows系统的注册表非常相似。许多android应用程序和库直接或者间接地依赖此特性，以决定它们的运行时行为。例如，adbd进程查询属性服务已确认当前是否运行在模拟器环境中。另一个例子是java.io.File.pathSpearator，其返回存储于属性服务中的值。

1. 属性文件路径

Android编译时收集的各种property记录在build.prop文件。

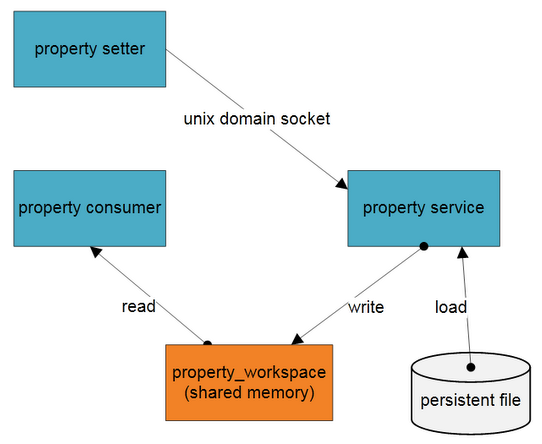
文件路径：out/target/product/<board>/system/



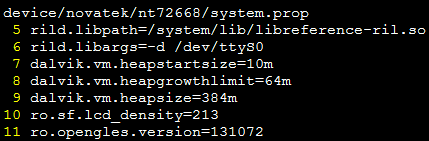
1. 属性前缀

前缀必须用system\core\init\property\_service.c中定义的前缀 ，进行系统属性设置的程序也必须有相应的进程权限。

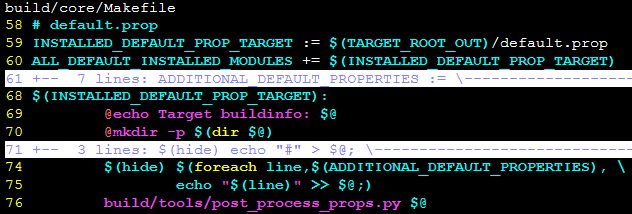
1. “ro.”开头，为只读属性。
2. “persist.”开头，当设置这个属性时，其值也将写入/data/property。
3. “net.”开头，当设置这个属性时，“net.change”属性将会自动设置，以加入到最后修改的属性名。
4. 属性系统结构



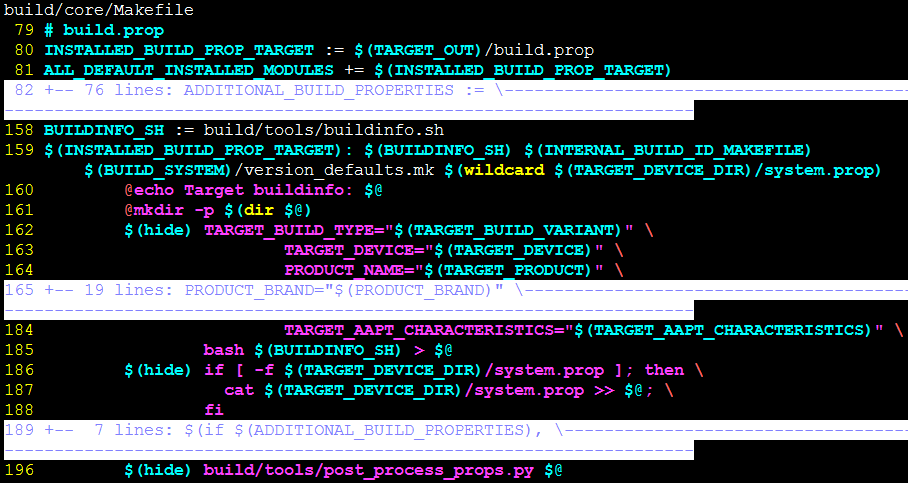
1. shared memory是所有属性记录的存储所在。
2. service进程负责从永久文件中加载属性记录并将它们保存在共享内存中。setter进程将共享内存加载到其自身的虚拟地址空间，但其不能直接写该内存。当setter试图增加或者更新一个属性时，它将该属性通过unix domain socket发送至service而实现属性写入共享内存和永久文件中。
3. consumer进程将共享内存加载到其自身的虚拟地址空间并直接访问这些属性。
4. 构造流程
5. 系统 prop文件定义



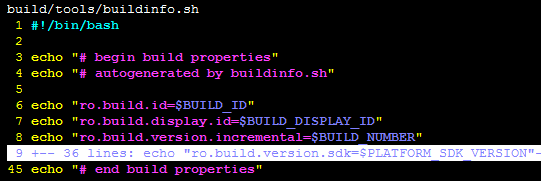
1. 默认prop文件创建



1. 编译prop文件创建

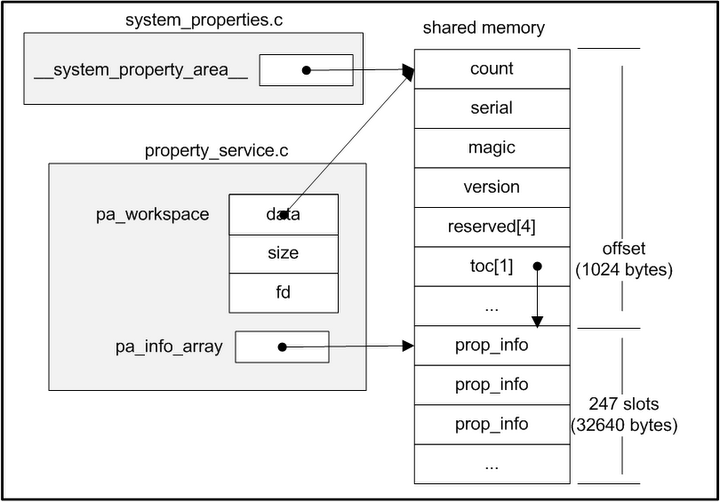


构造buildinfo.sh的定义：



1. 初始化流程
2. 创建共享内存区

init进程首先创建一个shared memory区域，并保存一个指向该区域的描述符fd。init进程将该区域通过使用了MAP\_SHARED标志的mmap映射至它自身的虚拟地址空间，这样，任何该区域对于所有进程都是可见的。fd和区域大小被存储在一个名为ANDROID\_PROPERTY\_WORKSPACE的变量中。任何其他进程，比如consumer和setter将使用这个变量来获得fd和尺寸，这样它们就能mmap这个区域到它们自身的虚拟地址空间中。该共享内存区域如下图所示。



1. 加载属性文件

init进程将从下列文件加载属性：

/default.prop

/system/build.prop

/system/default.prop

/data/local.prop

1. 启动property service

创建unix domain socket：/dev/socket/property\_service，调用poll来等待该socket上的连接事件。

1. 更新property

当初始化libc（bionic/libc/bionic/libc\_common.c \_\_libc\_init\_common 函数），将从环境变量中返回fd和尺寸，并映射共享内存到其自身的地址空间（bionic/libc/bionic/system\_properties.c \_\_system\_properties\_init 函数）。之后通过libcutils可以像读取普通内存那样为consumer读取属性。

1. 访问方式
2. native code

当编写本地应用程序时，可以使用property\_get和property\_set 这两个API来读取/设置属性。要使用它们，我们需要include cutils/properties.h，并链接libcutils库。

1. java code

在Java包（java.lang.System）中提供有System.getProperty和System.setProperty方法。但值得注意的是，尽管这两个API在语义上等同native函数，但其将数据存储于完全不同的位置。实际上，dalvik VM使用一个哈希表来存储这些属性。所以，用这两个API存储的属性各自独立的，java不能存取native属性，反之亦然。

然而Android有一个内部隐藏类（@hide，对SDK不可见）android.os.SystemProperties来操纵native属性。其通过jni来存取native属性库。

1. shell脚本

Android提供getprop和setprop命令行工具来获取和更新属性。其依赖libcutils实现。

getprop <属性名>

setprop <属性名><<属性值>

### debug

1. 打开调试信息

#undef NDEBUG //打开LOGV/LOGI/LOGD

#define LOG\_NDEBUG 0 //打开LOGV

#define LOG\_NIDEBUG 0 //打开LOGI

#define LOG\_NDDEBUG 0 //打开LOGD

1. 查看调试信息

需要打印时调用即可，即可输出类名、函数名和位于的行数。其他信息可以依葫芦画瓢，方便查找。

在使用串口调试的时候可以用logcat来查看。

-s 设置过滤器，例如指定 '\*:s'

-f <filename> 输出到文件，默认情况是标准输出。

-r [<kbytes>] Rotate log every kbytes. (16 if unspecified). Requires -f

-n <count> Sets max number of rotated logs to <count>, default 4

-v <format> 设置log的打印格式, <format> 是下面的一种:

brief process tag thread raw time threadtime long

-c 清除所有log并退出

-d 得到所有log并退出 (不阻塞)

-g 得到环形缓冲区的大小并退出

-b <buffer> 请求不同的环形缓冲区 ('main' (默认), 'radio', 'events')

-B 输出log到二进制中。

过滤器表达式的格式是tag:priority ... ，其中tag是标记， priority是最小的优先级， 该标记标识的所有大于等于指定优先级的消息被写入日志。也可以在一个过滤器表达式中提供多个这样的过滤，它们之间用空格隔开。

优先级标识符(小🡪大)：

V(Verbose),D(Debug),I(Info),W(Warn),E(Error),F( Fatal),S(Silent)

1. 依赖文件

使用时andorid.mk中需要加入

LOCAL\_SHARED\_LIBRARIES += \

libcutils libutils

LOCAL\_LDLIBS += -llog

源文件中需要引用#include "utils/Log.h"

1. printf打印

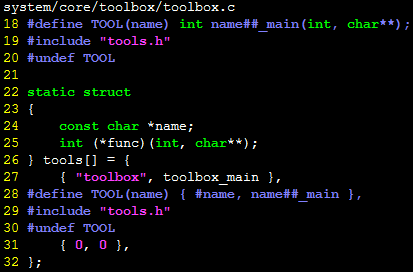
将可执行文件的路径作为logwrapper的参数，在init.rc中这样写：

service adbd /system/bin/logwrapper /sbin/adbd

### toolbox

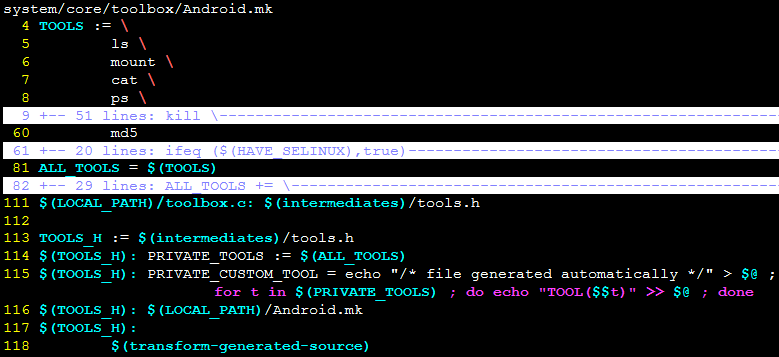
toolbox为Android系统上使用的linux命令工具，类似于busybox。

1. 工具定义



1. 头文件构造规则

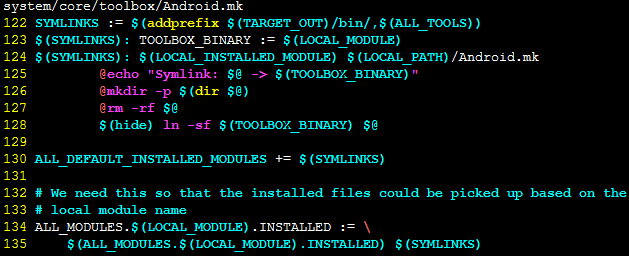
tool.h文件是根据makefile配置自动生成的，其构造规则如下：



<118>最终调用<115>定义的工具创建tool.h.

1. 安装规则

安装的方法是根据工具名创建软连接，方法如下：

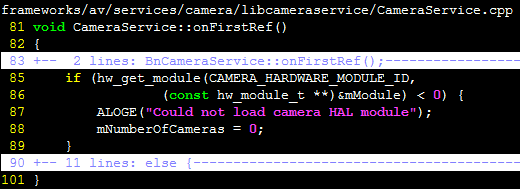


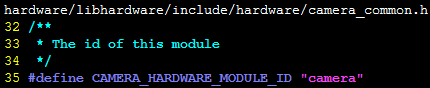
## HAL

### HAL加载

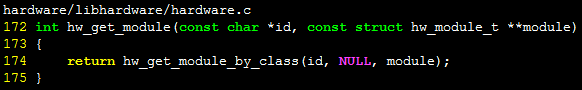
1. 加载的时机

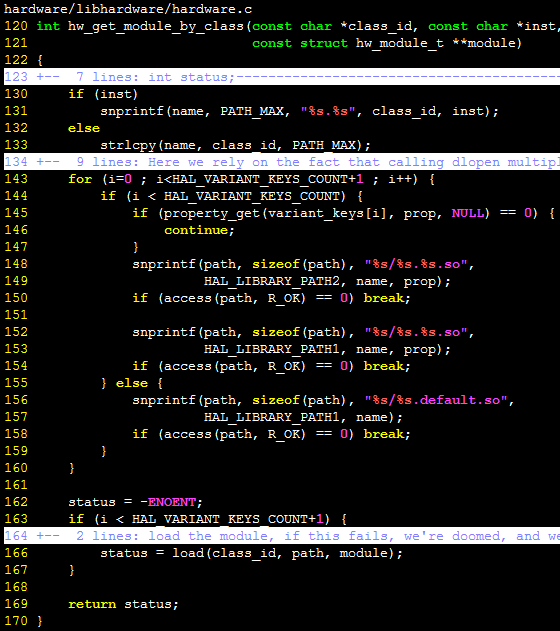
一般通过在模块第一次引用时调用get\_module载入。



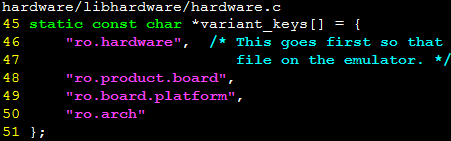


1. hw\_get\_module实现

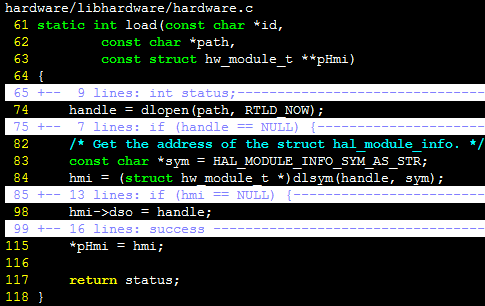




查询顺序：



1. load的实现



### HAL构造

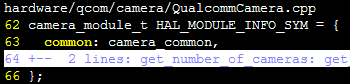
1. HAL moudle信息定义

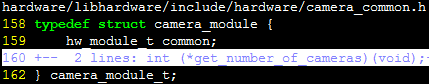
每个hardware module必需有一个HAL\_MODULE\_INFO\_SYM的变量，并且必需以hw\_module\_t作为该变量的开始(类似LKM的modinfo)。

1. HAL模块信息

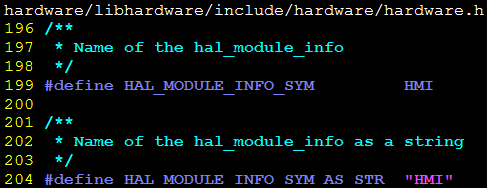


定义示例(camera hal)：

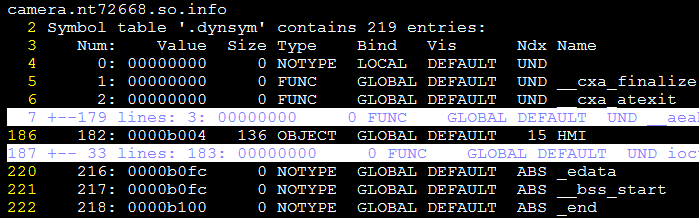




1. 变量名称定义

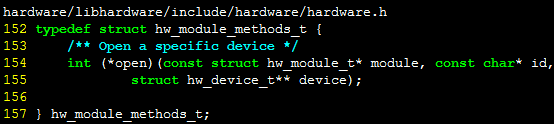


比如查看hal的so文件：

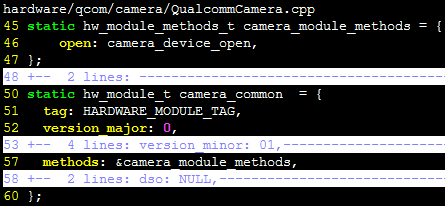


1. 模块方法定义

模块方法为HAL\_MODULE\_INFO\_SYM的hw\_module\_t的一个成员，用于存放模块的通用操作方法(目前只有打开模块的方法)。定义为struct hw\_module\_methods\_t(类似LKM 的modinfo中init/cleanup)。



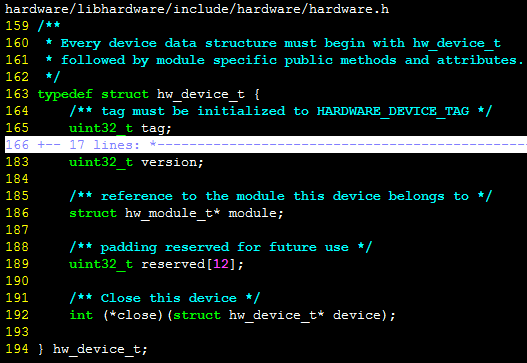
定义示例(camera hal)：



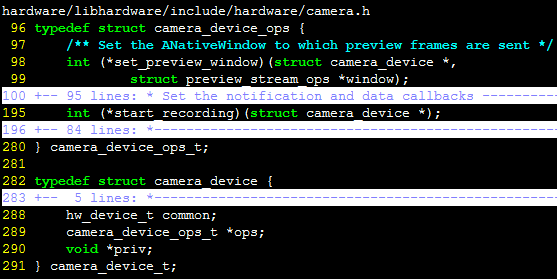
1. 设备描述信息

设备描述信息存储了对应hardware的接口方法，在hal加载时动态生成，由hal的代码实现初始化(类似LKM的private指针)。

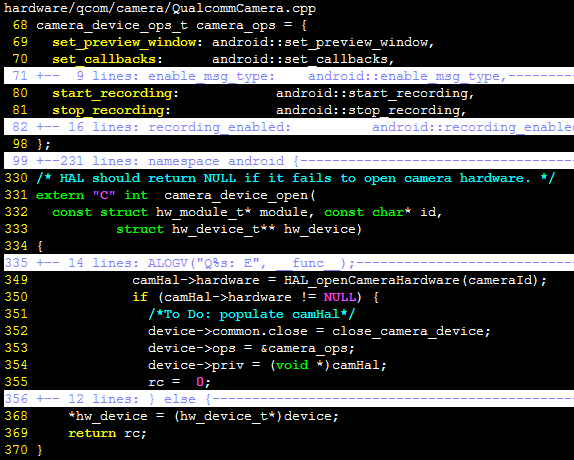
每个具体的设备描述信息必需以struct hw\_device\_t成员开始：



定义示例(camera hal)：

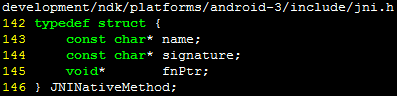


初始化示例(camera hal)：



## JNI

### 函数映射

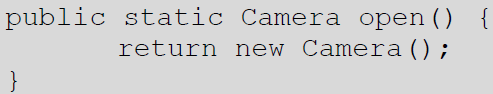


1. name：Java中函数的名字。
2. signature：描述函数的参数和返回值的字符串。字符串的格式：“(T)T”，其中T表示类型标识符。
3. 括号内部的T表示函数参数类型，最后一个T表示函数返回类型。
4. T标识符意义

|  |  |
| --- | --- |
| 标识符 | 类型 |
| V | void |
| Z | jboolean |
| I | jint |
| J | jlong |
| D | jdouble |
| F | jfloat |
| B | jbyte |
| C | jchar |
| S | jshort |
| [I | jintArray |
| Ljava/lang/String; | String jstring |
| Ljava/net/Socket; | Socket jobject |

1. 对象类型：以"L"开头，以";"结尾，各层次之间用"/" 隔开，如果对象位于一个嵌入类，则用"$"作为分隔符。
2. 数组类型：以"["开始，"["的重复个数表示数组维数，如"[[[D"表示“double[][][]”。
3. 对于没有参数的函数，则省略括号内部的T。
4. 多个参数按照从左到右的顺序依次描述。
5. fnPtr：是函数指针，指向C函数。
6. JNI调用用方式
7. 调用JAVA类；

比如：获得一个 android.hardware.camera类的实例。



1. 直接调用native代码。

比如：调用android\_hardware\_camera.cpp的代码。



### 变量类型

1. 不透明类型

struct \_jobject;  
typedef struct \_jobject \*jobject;  
struct \_jfieldID;  
typedef struct \_jfieldID \*jfieldID;  
struct \_jmethodID;  
typedef struct \_jmethodID \*jmethodID;  
  
So, the types are defined as pointers to opaque (undefined) structs, in order to get better compile-time type checking

### ENV

C++为类,使用的是env, C为指针,使用的是(\*env)，

### JNI编译

1. JNI代码位置

framework/base/core/jni/文件夹下的文件都被编译进 libandroid\_runtime.so公共库中。

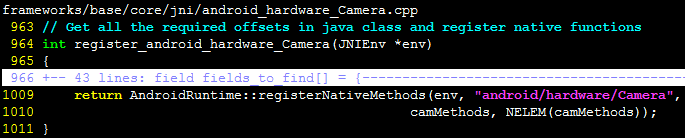
1. 动态库位置

显式的动态库通过hw\_get\_module -> dlopen打开，库文件必须放在文件系统的 system/lib/hw/下。

### JNI构造

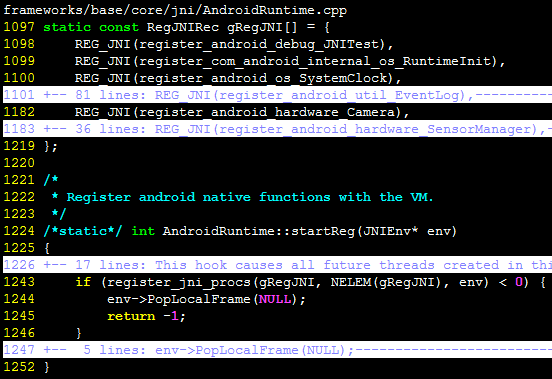
1. 定义

定义JNI的实现代码完成java和c之间的交互。

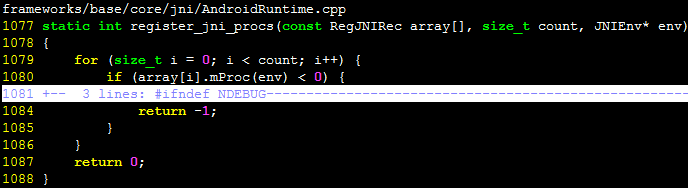


1. 注册

统一调用JNI的注册接口。

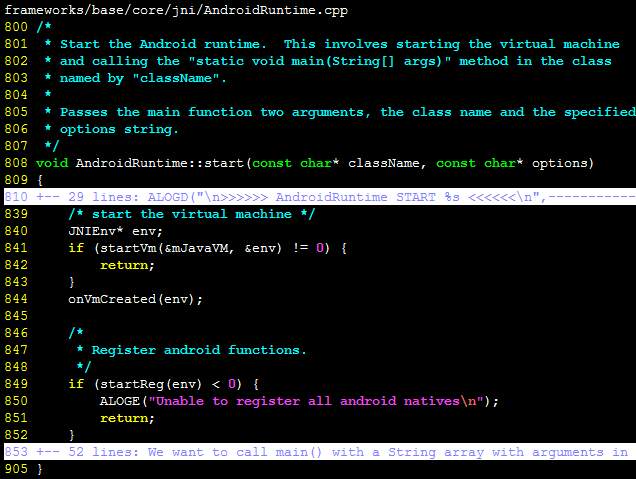


调用各HAL模块实现的JNI注册函数。



1. 启动

VM启动时，在当前的VM中注册JNI。



# Android模块

## camera

Camera是Andriod的视频输入系统，包含取景器（viewfinder）、录像（Recording）和拍照的功能。Camera部分框架部分包含在ui库中，中间层的实现是CameraService，通过调用下层的Camera HAL来实现功能。

Android2.3.4中CameraHardwareInterface是一个纯虚基类,实际的Camera HAL层只需要实现CameraHardwareInterface中定义的所有的接口即可.

在Android2.3.4中CameraService的Android.mk中有如下:

ifeq ($(USE\_CAMERA\_STUB), true)

LOCAL\_STATIC\_LIBRARIES += libcamerastub

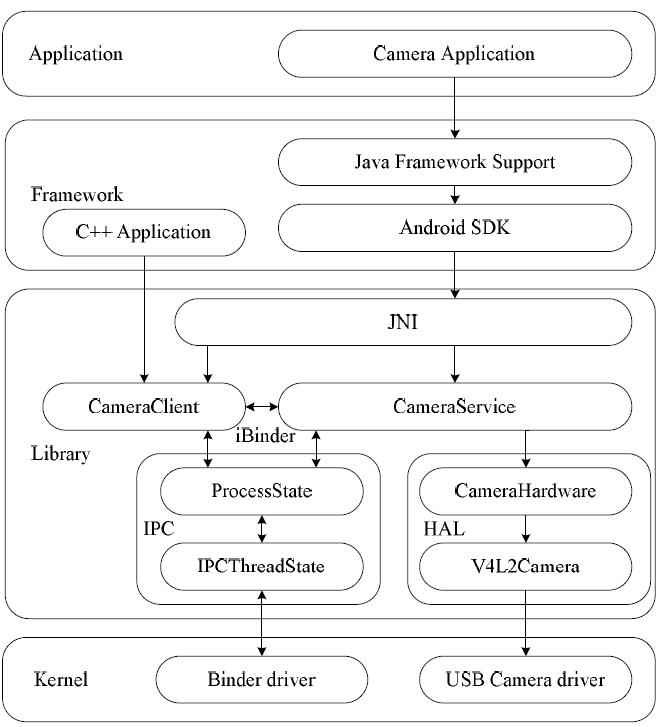
else

LOCAL\_SHARED\_LIBRARIES += libcamera

endif

在Android4.0中, CameraHardwareInterface不再是一个简单的纯虚接口类, 而是将所有的接口实现都转接到了camera\_device\_t对象中.

### framework



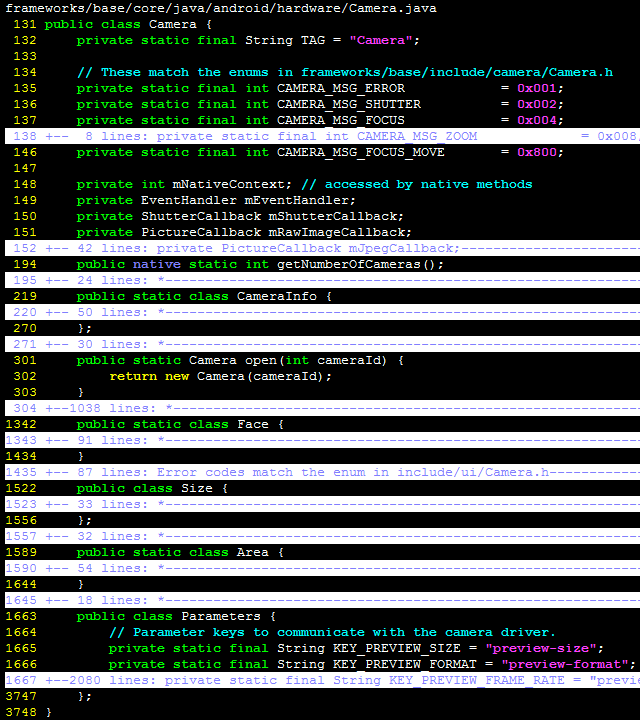
1. app

通过调用Framework层的android.hardware.camera类来实现具体的功能由JAVA 代码实现，最终打包成APK。原生Camera应用代码在Camera.java(packages\apps\camera\src\com\android\camera。

1. framework

提供android.hardware.camera类给app使用。由JAVA代码实现，通过JNI机制调用native代码实现的具体功能。

1. 源码：frameworks/base/core/java/android/hardware/Camera.java。



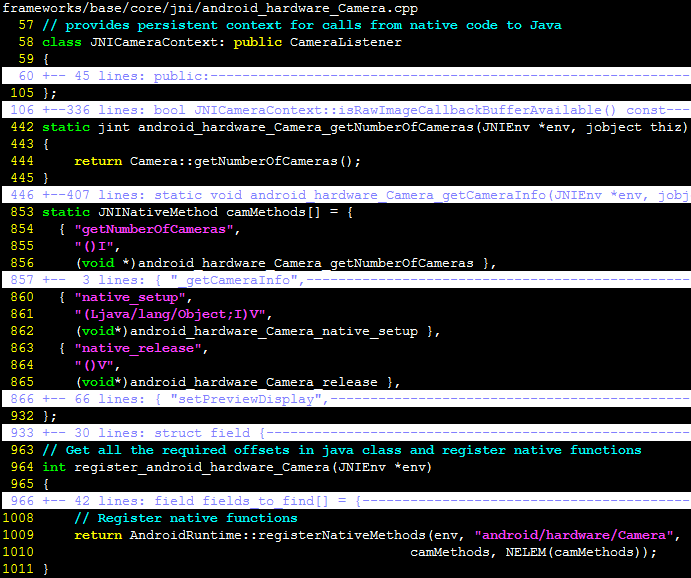
1. app参数传递

参数以map的方式组织存储，使用Parameters类中形如“preview-size=640X480;preview-format=yuv422sp;”格式的字符串传到service端。

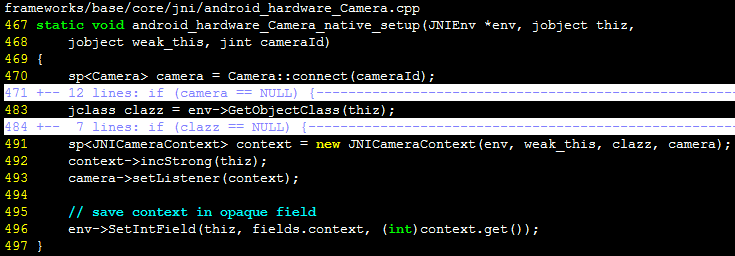
1. JNI

实现framework中JAVA类android.hardware.Camera的本地代码调用。

1. 源码： frameworks/base/core/jni/android\_hardware\_Camera.cpp。



1. 通过register\_android\_hardware\_Camera(JNIEnv \*env)函数(cpp)将native函数注册到虚拟机中，以供framework层的JAVA 代码调用。
2. 通过native\_setup/native\_release函数(java)打开/关闭camera。



1. native函数通过调用libcamera\_client.so中的**Camera**类实现具体的功能。
2. client

client相关的类源码位置：frameworks/av/camera/,各类实现文件如下：

**Camera**类-->Camera.cpp,Camera.h

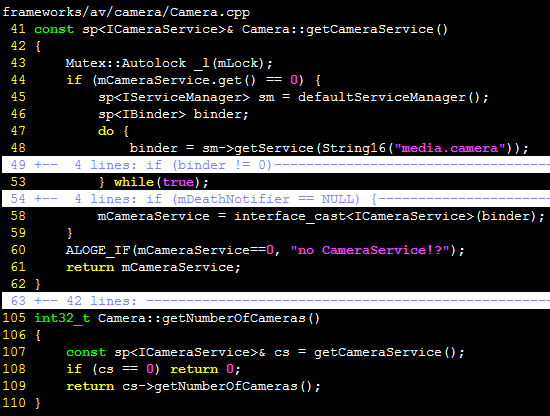
CameraParameters类-->CameraParameters.cpp,CameraParameters.h

Icamera类-->ICamera.cpp,ICamera.h

IcameraClient类-->ICameraClient.cpp,ICameraClient.h

IcameraService类-->ICameraService.cpp,ICameraService.h

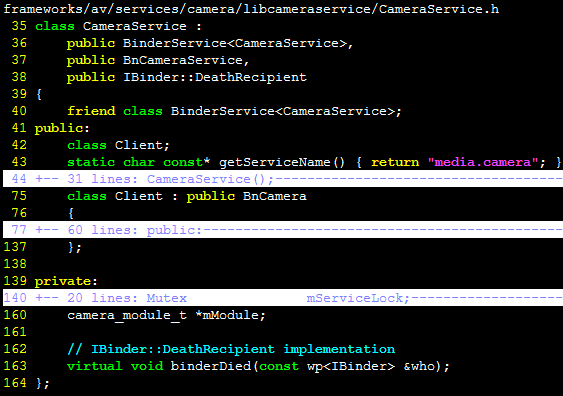
1. Icamera、IcameraClient、IcameraService三个类是按照Binder IPC通信要求的框架实现的，用来与service端交互来实现具体功能。
2. CameraParameters 接收android.hardware.camera::Parameters类为参数，解析与格式化所有的参数设置。
3. Camera继承自BnCameraClient🡪ICameraClient。



1. service

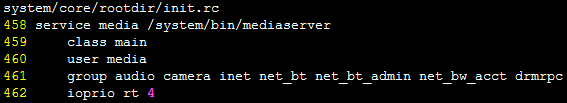
service相关的类源码位置：frameworks/av/services/camera/libcameraservice，编译入libcameraservice.so,如下：

1. CameraService类，继承自BnCameraService🡪ICameraService。

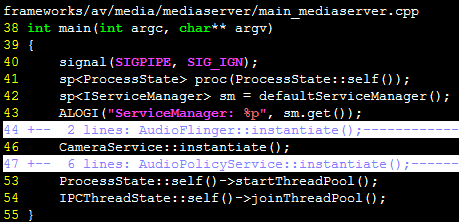


1. CameraService::Client类，继承自BnCamera🡪ICamera，通过调用HAL层来实现具体的功能。
2. Service启动

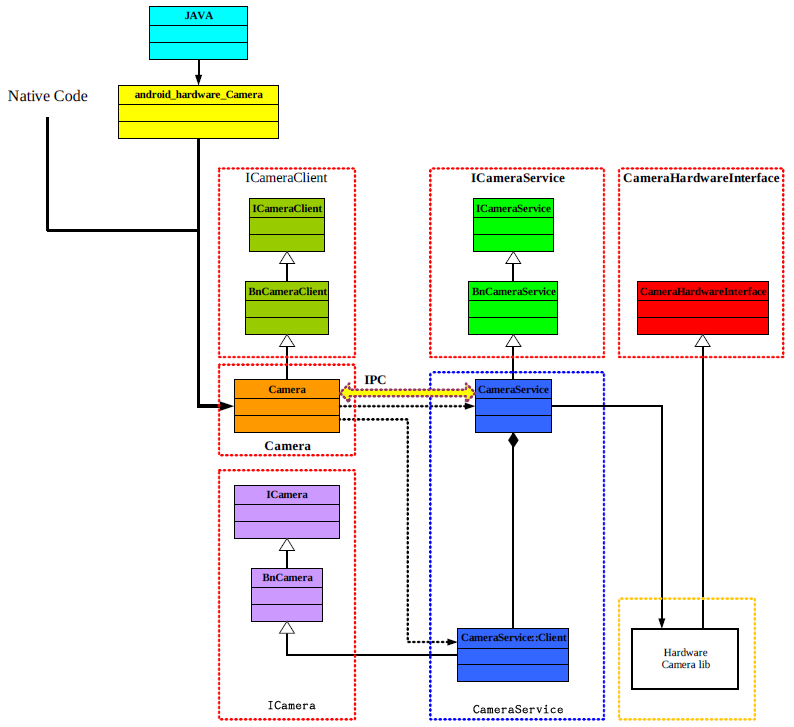
在init.rc中有如下代码用于执行/system/bin/mediaserver，启动多媒体服务进程。



mediaserver的实现中创建了CameraService实例，名称为“media.camera”。



### 类结构



### 代码位置

1. Android 4.0 (ICS)
2. Android Framework

Java: framwworks/base/core/java/android/hardware

JNI: frameworks/base/core/jni

1. Camera Service

frameworks/base/services/camera/libcameraservice/

1. IBinder Interfaces

frameworks/base/include/camera/ICamera.h

1. IBinder Implementation

frameworks/base/libs/camera/ICamera.cpp etc

1. Camera HAL Interface

frameworks/base/services/camera/libcameraservice/CameraHardwareInterface.h

1. Camera HAL

hardware/<vendor>/camera

1. Android 4.2 (Jelly Bean)
2. Android Framework

Java: frameworks/base/core/java/android/hardware

JNI: frameworks/base/core/jni

1. Camera Service

frameworks/av/services/camera/libcameraservice/

1. IBinder Interfaces

frameworks/av/include/camera/ICamera.h etc.

1. IBinder Implementation

frameworks/av/camera/ICamera.cpp etc.

1. Camera HAL Interface

frameworks/av/services/camera/libcameraservice/CameraHardwareInterface.h

1. Camera HAL

hardware/<vendor>/camera

### callback

1. camera\_notify\_callback

用来传递一些消息: 如快门消息，自动对焦完成消息，出错消息，拍照编码结束等, 参数@msg\_type可以是：

CAMERA\_MSG\_ERROR

CAMERA\_MSG\_SHUTTER

CAMERA\_MSG\_FOCUS

CAMERA\_MSG\_ZOOM

CAMERA\_MSG\_FOCUS\_MOVE

1. camera\_data\_callback

该函数用于从camera HAL返回数据，返回的数据可以是：

preview的数据帧，

preview的元数据，

以及压缩为jpeg格式的帧数据

参数@msg\_type可以是：

CAMERA\_MSG\_PREVIEW\_FRAME

CAMERA\_MSG\_PREVIEW\_METADATA

CAMERA\_MSG\_COMPRESSED\_IMAGE

CAMERA\_MSG\_RAW\_IMAGE

CAMERA\_MSG\_RAW\_IMAGE\_NOTIFY

CAMERA\_MSG\_POSTVIEW\_FRAME

参数@data 用来返回图像数据

参数@metadata 用来返回图像的元数据

1. camera\_data\_timestamp\_callback

该回调函数用来从Camera HAL返回帧数据以及对应的时间戳，这个callback用于录像的场景。

@timestamp：生成帧数据的时间戳

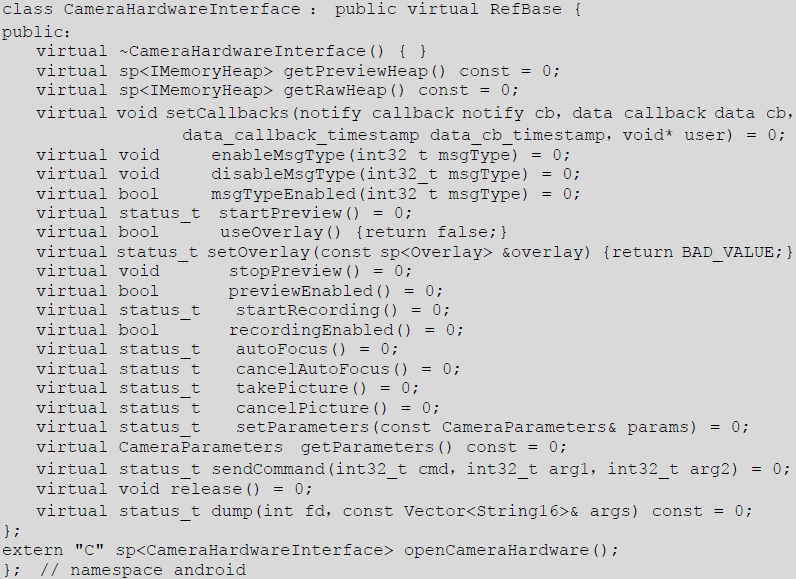
@msg\_type: 取值为CAMERA\_MSG\_VIDEO\_FRAME

@data：用来返回帧数

### HAL

Android通过HAL层来保证底层硬件（驱动）改变时，只需修改对应的HAL层代码，frameWork层与app的都不用改变。

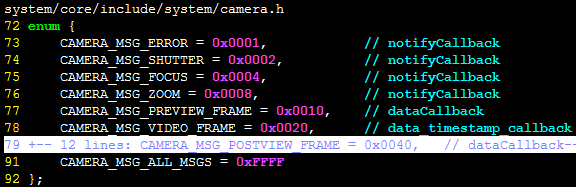
1. HAL一般编译为一个动态库libcamera.so（动态库名字可以改，只需要与Android.mk一致即可)。
2. HAL通过被Libcameraservice.so::CameraService::Client调用而实现具体camera功能，HAL需要实现HAL接口类，该类的定义文件如下：

frameworks/base/include/camera/CameraHardwareInterface.h

1. HAL对设备的访问

打开对应的设备文件（如/dev/video0），并通过ioctrl访问camera driver。一般通过V4L2 command从linux kernel中的camera driver得到视频数据。然后交给surface(overlay)显示或者保存为文件。

1. 如果使用映射内核内存的方式（V4L2\_MEMORY\_MMAP），构建预览的内存MemoryHeapBase需要从V4L2驱动程序中得到内存指针；如果使用用户空间内存的方式（V4L2\_MEMORY\_USERPTR），MemoryHeapBase中开辟的内存是在用户空间建立的；
2. 在预览的线程中，使用VIDIOC\_DQBUF调用阻塞等待视频帧的到来，处理完成后使用VIDIOC\_QBUF调用将帧内存再次压入队列，等待下一帧的到来。
3. 由于video0 权限不够会造成open 打开设备文件/dev/video0失败，因此需要在Android源代码/system/core/init/device.c中修改权限为660。
4. HAL需要响应的消息



### 预览（ViewFinder/Preview）

应用程序Camera 调用startpreview 接口，利用底层的Binder 机制，通过客户端CameraClient 通知服务 端CameraService，服务端通过内部类调用硬件抽象层中提供的功能，完成具体的操作。

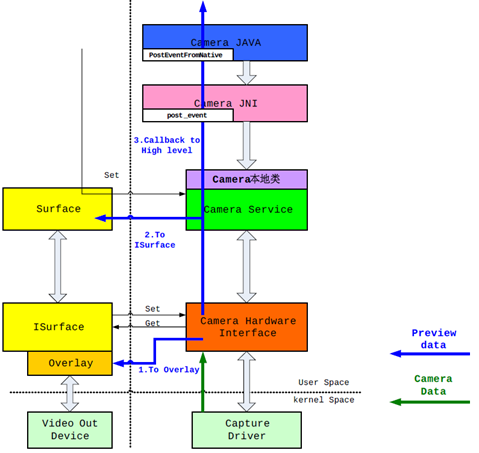
硬件抽象层的CameraHardware 创建previewthread 线程和Loop 线程， 前者通过GrabPreveiwFrame 接口取得V4L2Camera 采集摄像头的一帧数据，放入指定缓冲区，并通过convert 将数据转换为RGB565 ，之后向服务端发送取景消息

CAMERA\_MSG\_PREVIEW\_FRAME，从而实现了底层数据到缓冲区ISurface 的映射；后者将硬件设备采集的数据通过postBuffer不断的传递给显示缓冲区framebuffer，完成取景显示。

1. 数据流

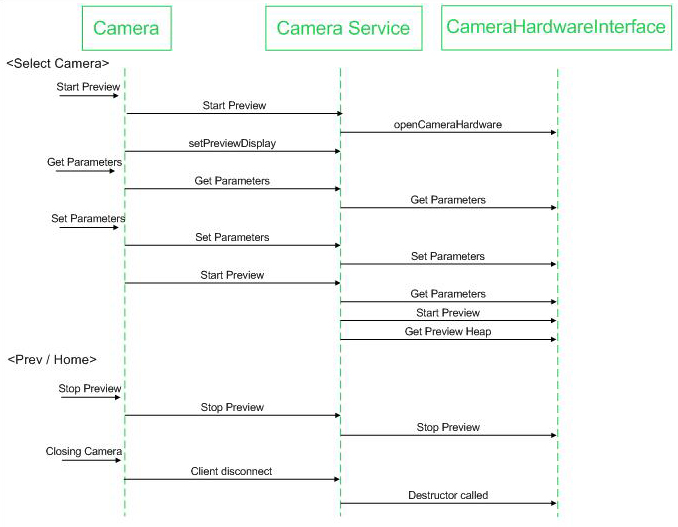
真正的preview 数据不会通过Binder IPC机制从service端复制到client端，但会通过回调函数与消息的机制将preview 数据的buffer地址传到client端，最终可在JAVA AP中操作处理这个preview数据。实现有3种方式：

1. Camera类通过Callback送给上层，由上层处理。
2. 在CameraHAL中，直接送给Overlay。
3. 在CameraService中，调用ISurface的postBuffer接口，送出数据。



在Android照相机/摄像机应用程序中，使用的是b和c这两种方法，具体由CameraService读取CameraHAL的useOverlay 接口来实现。

1. 系统运行



1. 打开内核设备文件。

CameraHardwareInterface.h中定义的 openCameraHardware()打开linux kernel中的camera driver的设备文件（如/dev/video0），创建初始化一些相关的类的实例。

1. 设置摄像头的工作参数。

CameraHardwareInterface.h中定义的 setParameters()函数，在这一步可以通过参数告诉camera HAL使用哪一个硬件摄像头，以及它工作的参数（size，format等），并在HAL层分配存储preview数据的buffers（如果buffers是在linux kernel中的camera driver中分配的，在这一步也会拿到这些buffers mmap后的地址指针）。

1. 设置显示目标。

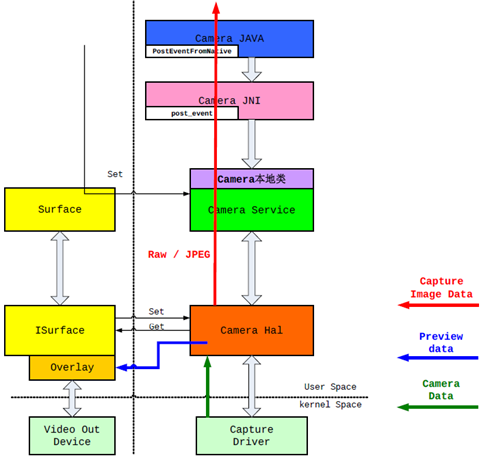
需在JAVA APP中创建一个surface然后传递到CameraService中。会调用到libcameraservice.so中的setPreviewDisplay(const sp<ISurface>& surface)函数中。在这里分两种情况考虑：一种是不使用overlay；一种是使用overlay显示。如果不使用overlay那设置显示目标最后就在libcameraservice.so中，不会进Camera HAL动态库。并将上一步拿到的preview数据buffers地址注册到surface中。 如果使用overlay那在libcameraservice.so中会通过传进来的Isurface创建Overlay类的实例，然后调用CameraHardwareInterface.h中定义的 setOverlay()设置到Camera HAL动态库中。

1. 开始preview工作。

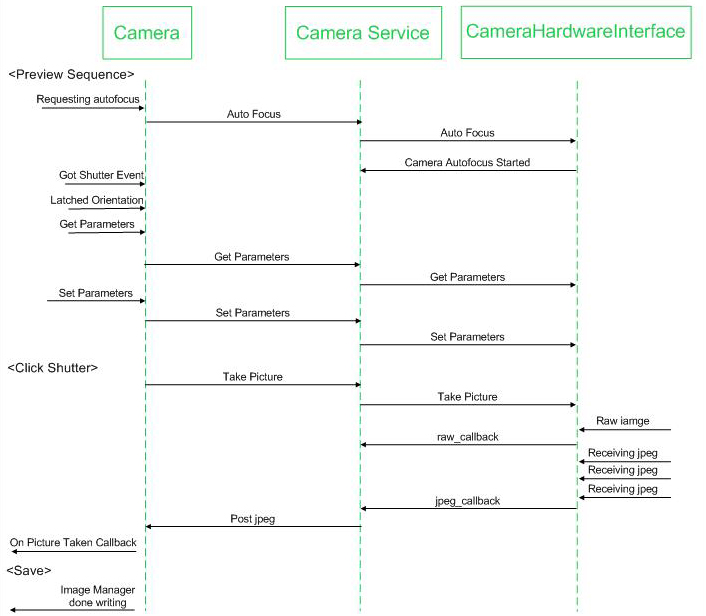
最终调用到CameraHardwareInterface.h中定义的 startPreview()函数。如果不使用overlay，Camera HAL得到linux kernel中的preview数据后回调通知到libcameraservice.so中。在libcameraservice.so中会使用上一步的surface进行显示。如果使用overlay，Camera HAL得到linux kernel中的preview数据后直接交给Overlay对象，然后有Overlay HAL去显示。

### 拍照(takePicture)

1. 数据流



1. 系统运行



在取景模式下，通过ClickShutterButton 接口，照相机从取景模式进入拍照模式。在拍照模式下， 应用程序通过发送focus 请求给硬件抽象层中的CameraHardware ，

CameraHardware 将完成focus 请求的具体操作，并通过notify 回调函数返回消息

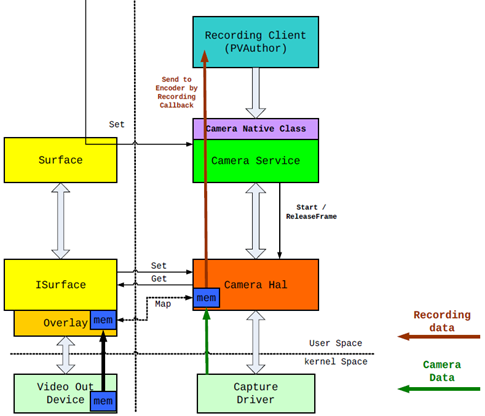
CAMERA\_MSG\_FOCUS。之后上层通过takepicture 接口启动硬件抽象层中的captureThread线程，线程启动之后将通过notify 回调函数向服务端CameraService 发送消息CAMERA\_MSG\_SHUTTER，此时进入真正的拍照模式。

接下来CameraHardware 将调用底层的GrabRawFrame 获取图像照片，首先将原始数据转化为与显示缓冲区一致的数据格式，通过回调函数notify 向服务端CameraService 发送消息CAMERA\_MSG\_RAW\_IMAGE，服务端收到消息后调用RawPicture，然后由SurfaceFlinger 完成照片的显示。

随后，底层将调用GrabJpegFrame返回Jpeg格式的照片，通过回调函数notify向服务端CameraService发送消息CAMERA\_MSG\_COMPRESSED\_IMAGE，服务端收到消息后调用handleCompressedPicture 将消息和照片数据内容都发送给客户端、JNI、上层应用程序，从而实现照片的存储。

### 录像(Recording)

1. 数据流



### 虚拟 camera

如果没有camera硬件，不实现真正的Camera HAL动态库，可以使用Fake Camera。

1. 源码位置：

frameworks/base/camera/libcameraservice/FakeCamera.cpp

frameworks/base/camera/libcameraservice/CameraHardwareStub.cpp

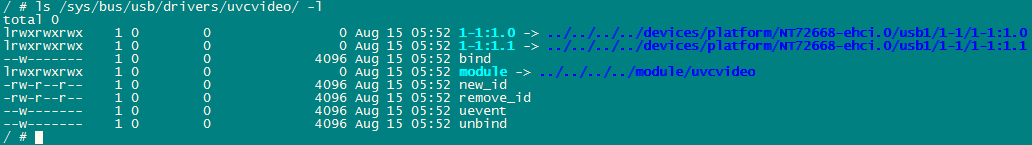
1. FakeCamera.cpp文件提供虚拟的preview数据。
2. CameraHardwareStub.cpp文件中实现了camera HAL的功能。当宏USE\_CAMERA\_STUB为true时可以使用这个虚拟的camera。



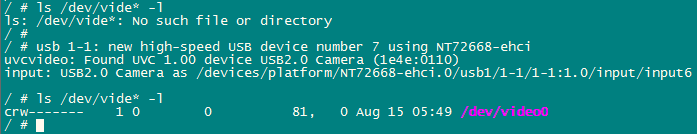
### usb camera

1. uvc 内核模块

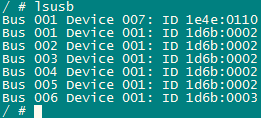


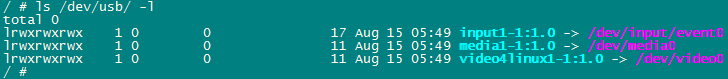


1. 插入USB摄像头

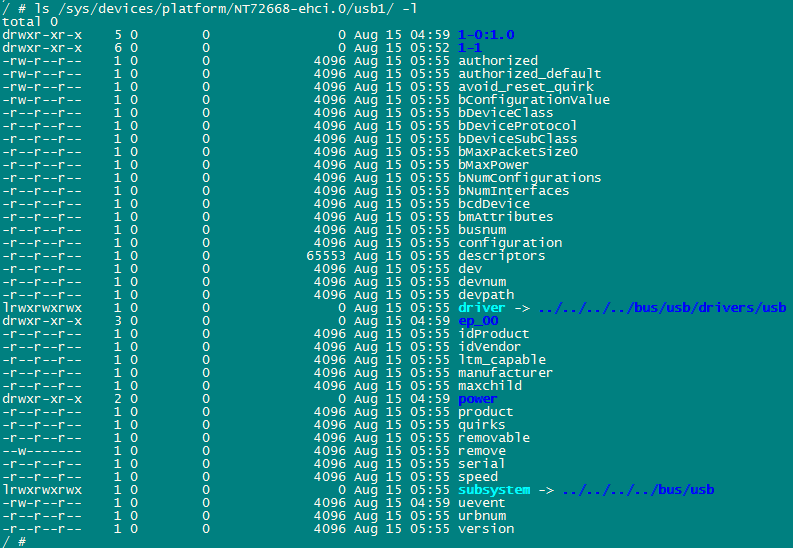


Vendor ID 为1e4e，Product ID 为0110.

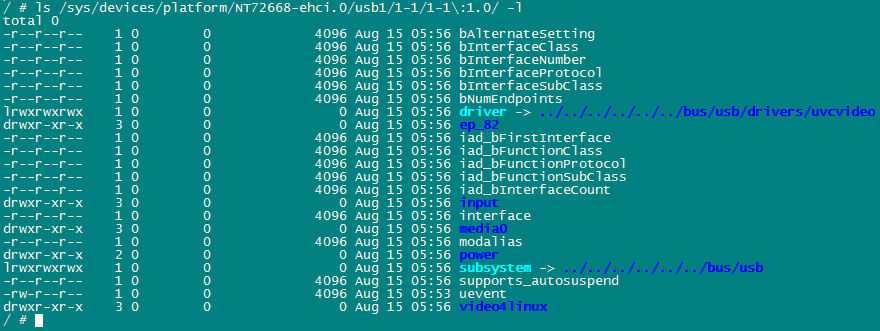


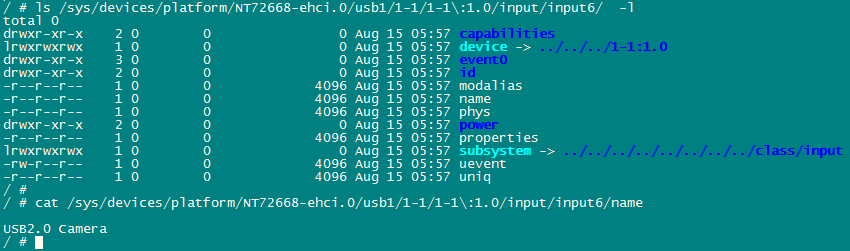


usb bus



usb camera





## Surface

### 功能

每个应用程序可能对应着一个或者多个图形界面，而每个界面就称之为一个surface ，或者说是window。

1. 信息记录

通过Surface记录应用程序界面的位置，大小，以及存储显示内容的buffer。

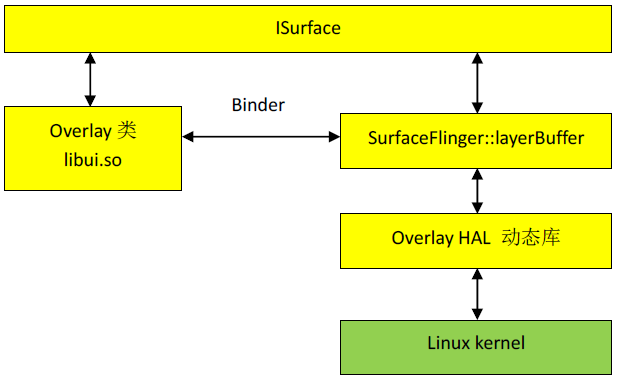
1. 图层合并
2. SurfaceFlinger(软件方式)

通过把各个surface 组合(compose/merge) 成一个main Surface ，最后将Main Surface 的内容发送给FB/V4l2 Output ，从而实现图形图层叠加，以及透明效果。

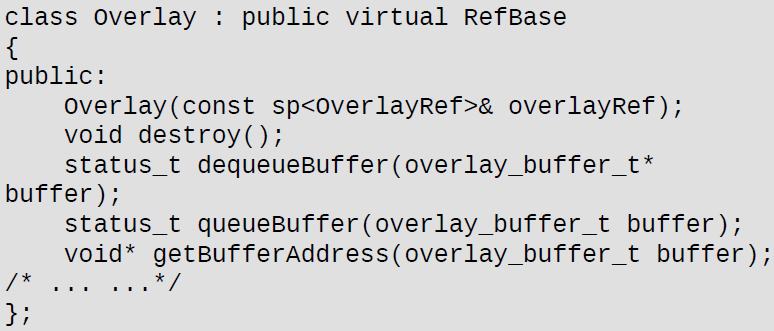
1. Overlay(硬件方式)

将硬件要求格式的数据和控制信息送到这个Overlay FrameBuffer，由硬件驱动来负责合并Overlay buffer和主屏buffer中的内容。

### 框架结构

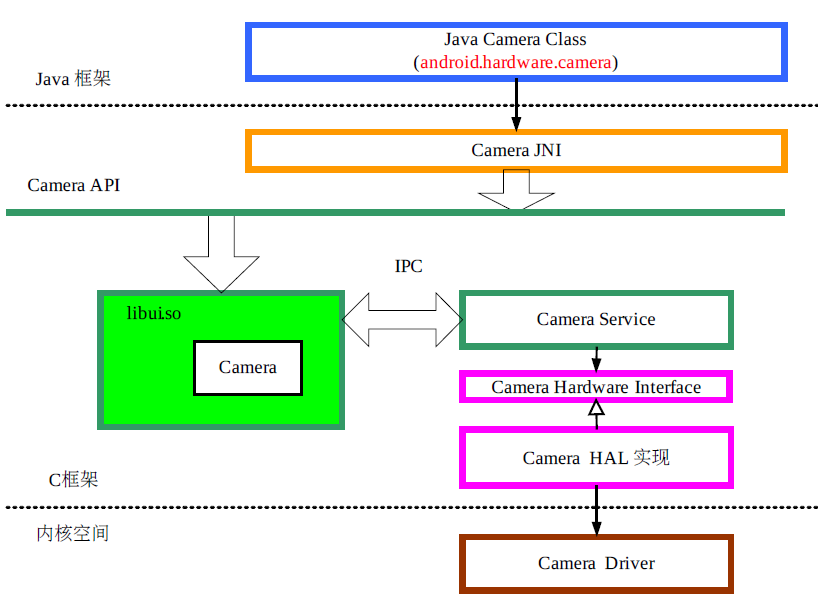
****

1. overlay没有 java 层的 code(即没有JNI调用)。一般在native代码中使用。
2. 源码位置：frameworks/base/libs/ui/ ，编译到libui.so中，各类实现文件如下。
3. Overlay，OverlayRef-->Overlay.cpp,Overlay.h：提供给外部程序调用的Overlay接口与API。通过binder与surfaceFlinger service实现通信，最终通过调用HAL实现具体功能。

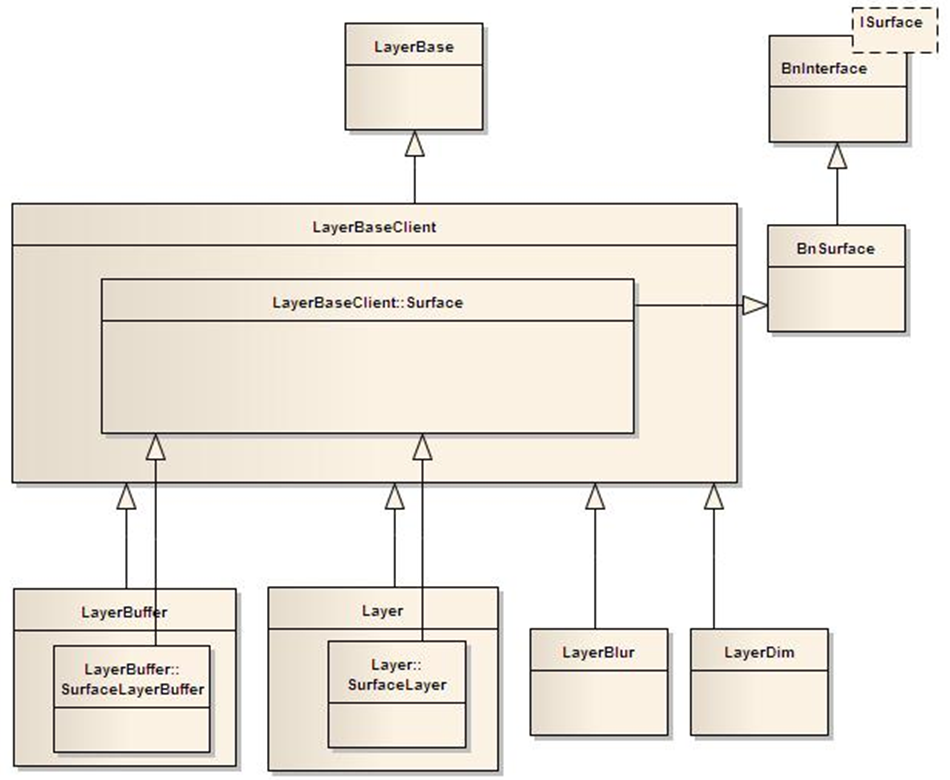


Overlay中的几个接口用于视频数据的输出，可以用队列，也可以直接使用地址。

1. IOverlay-->IOverlay.cpp, IOverlay.h：定义binder所需的类，其中LayerBuffer::OverlaySource::OverlayChannel继承自BnOverlay**。**
2. 与camera的关系



### 图层



1. Layer的分类

目前，android中有4中Layer类型，如上图所示。

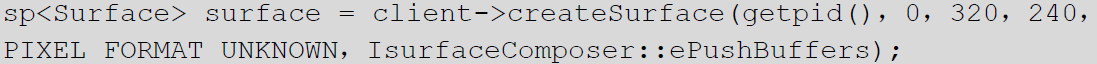
1. Layer, 普通的Layer，它为每个Client端请求的Surface创建显示Buffer。
2. LayerBuffer，这种Layer它并不会创建显示Buffer，它只是使用已有的Buffer作为显示Buffer，如Camera的preview；
3. LayerBlur，这种Layer也不会创建显示Buffer，它只是将通过这个Layer将原来FrameBuffer上的数据进行模糊处理；
4. LayerDim，这种Layer也不会创建显示Buffer，它只是将通过这个Layer将原来FrameBuffer上的数据进行暗淡处理；

### 使用方式

1. 创建surfaceflinger的客户端



1. 创建推模式surface



1. 获得surface接口



1. 获得overlay设备



这里会通过调用overlay hal层的createoverlay()打开对应的设备文件。

1. 创建overlay对象



1. 使用overlay API



address指针就是mmap后的overlay buffer 指针，只需将数据填充到这个 address 指针就可以看到画面了。

### Overlay

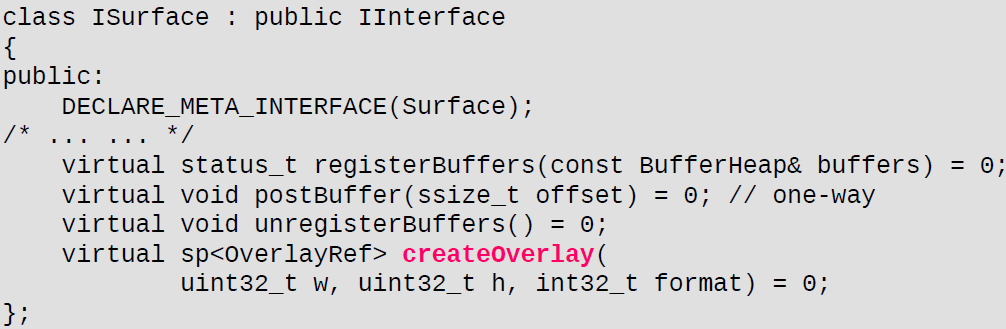
Overlay是Andriod的视频输出系统，其含义是叠加在主显示层上的另外一个显示层，这个叠加的显示层通常用在摄像头预览，视频播放等需要高帧率的地方，还有可能 UI 界面设计的需求，如 map 地图查看软件需两层显示信息。

1. Surface系统用于统一管理使用Overlay系统或者不使用Overlay系统的情况。所以Overlay 的service部分包含在SurfaceFlinger service中，主要的类LayerBuffer。

源码位置：frameworks/base/libs/surfaceflinger/

1. 使用Overlay系统需要通过ISurface接口，Overlay的使用与ISurface中的

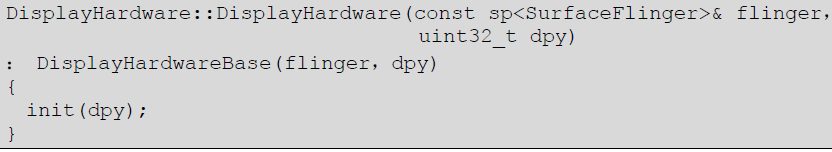
registerBuffers，postBuffer，unregisterBuffers几个接口是并立。



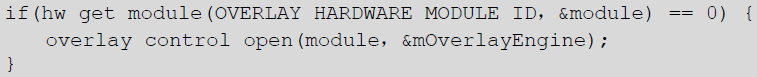
1. Overlay初始化
2. android 启动的时候会启动SurfaceFlinger service，SurfaceFlinger启动时会实例化一个 DisplayHardware：



1. DisplayHardware 构造函数调用函数init：



1. Init 函数中获得overlay的module参数，调用overlay\_control\_open获取控制设备结构mOverlayEngine。拥有了控制设备结构体就可以创建数据设备结构体 ，并具体控制使用 overlay 了：



### HAL

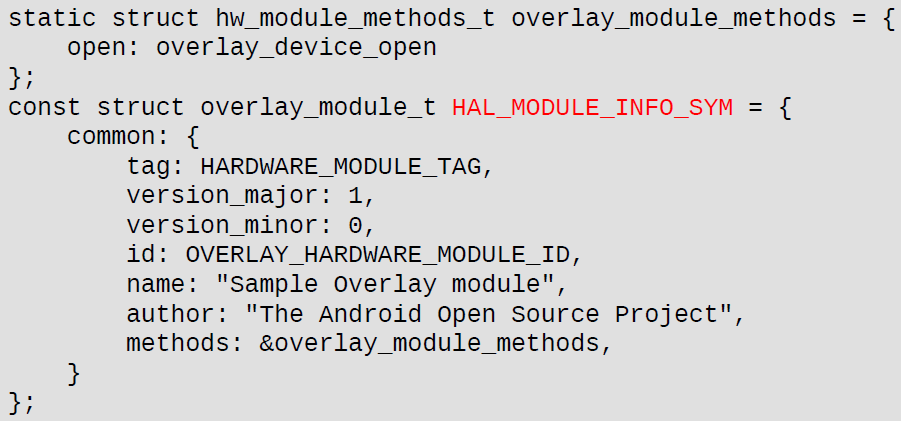
1. 接口定义

HAL接口定义位置：hardware/libhardware/include/hardware/overlay.h。主要定义了两个类：overlay\_control\_device\_t和overlay\_data\_device\_t，它们继承了hw\_device\_t和common，通过这两个类实现Overlay的硬件抽象层。

1. 实现

实现一个Overlay HAL使用的是Android硬件模块的标准方法，通过类overlay\_module\_t来完成。

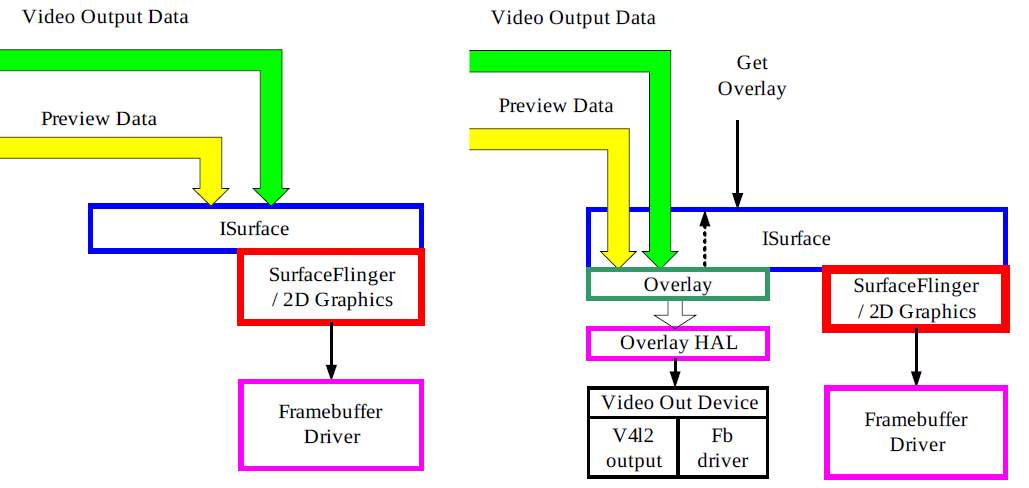
android中HAL实现示例：hardware/libhardware/modules/overlay/overlay.cpp。



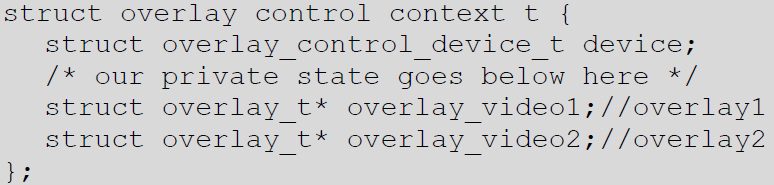
1. HAL 层生成的动态库在SurfaceFlinger中采用显式的动态方式打开(hw\_get\_module -> dlopen)，所以这个库文件必须放在文件系统的 system/lib/hw/下。
2. 驱动类型

Overlay硬件抽象层需要基于一个视频显示的驱动来实现。Overlay的硬件抽象层通常基于两类驱动：

1. framerbuffer驱动程序。基于framerbuffer驱动程序的实现，通常实现获得内存地址的接口即可。
2. Video for Linux 2中的视频输出驱动。基于v4l2的实现可以提供流方式的接口，获得更好的性能，其中又分成使用内核内存和使用用户空间内存两种方式。
3. Android系统不会主动使用Overlay，因此如果移植了Overlay的HAL。还需要增加使用Overlay的部分。不使用Overlay和使用Overlay的对比：



1. 多层 overlay
2. overlay hal 的 overlay\_control\_device\_t 中添加 overlay1 与 overlay2 的结构(假设需要同时支持 overlay1 与 overlay2)。其中每个overlay\_t代表一层overlay，每层ovelay有自己的handle。



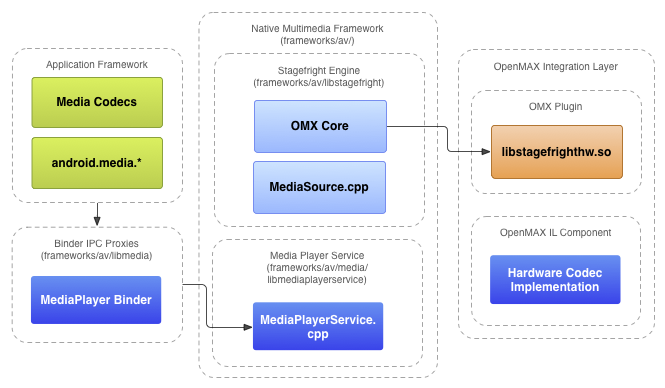
在构造OverlayRef之前需指明使用哪一层overlay：

****

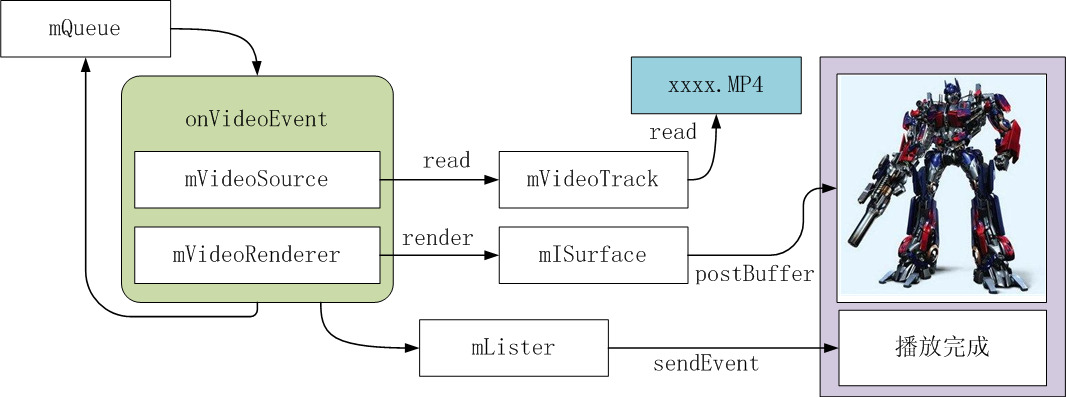
可以使用overlay\_control\_device\_t::setParameter()来指定Hal层具体实现。

1. 通过 Overlay object 来拿到 overlay1 与 overlay2 的 buffer 指针。

## media



### stagefright

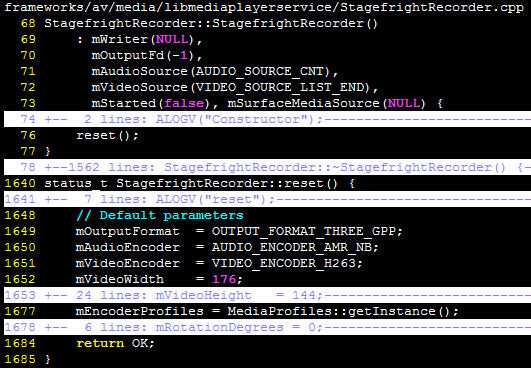


### meidaprofiles

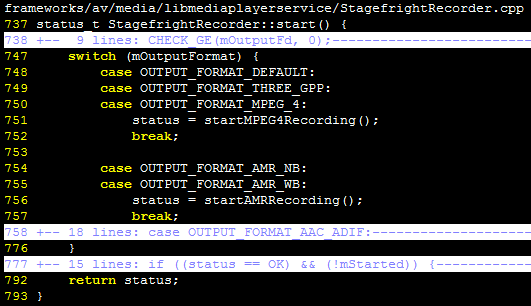
MediaProfiles类主要用于Recorder和Video Editor，包括设置视频的分辨率，Audio和Video的编解码参数等。

目前只有在StagefrightRecorder中用到了MediaProfiles，用于初始化录像参数。调用流程如下：

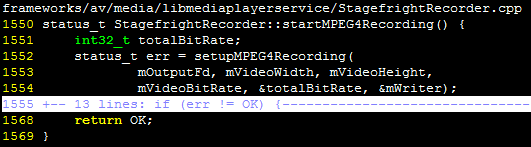
1. 获取配置
2. 构造MeidaProfile



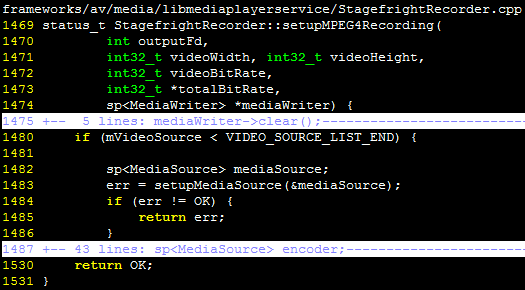
1. 应用配置
2. 启动Recorder



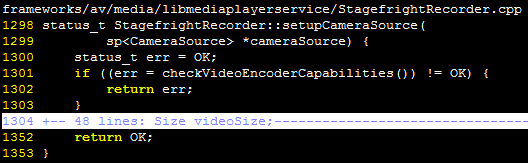
1. 配置Recorder



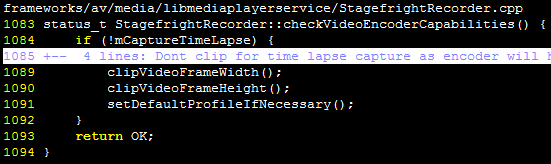
1. 设置媒体使用的源



1. 设置视频源



1. 使用Camera作为视频源



1. 读取配置

