

Unisoc Confidential For hiar

Android 10.0 编译系统变化介绍

文档版本
发布日期

V1.3
2020-09-15

版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

Unisoc Confidential For hiar

紫光展锐（上海）科技有限公司



前言

概述

本文主要总结了 Android 10 的编译系统变化。

读者对象


本文档适用于 Android 开发人员。

缩略语

缩略语	英文全名	中文解释
BSP	Board Support Package	板级支持包
GCC	GUN Compiler Collection	GUN 编译器套件

符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
 说明	用于突出重要/关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

修改记录

文档版本	发布日期	修改说明
V1.0	2019-09-30	初稿
V1.1	2020-12-16	适用产品信息增加 UIS8581E\SL8541E 平台
V1.2	2020-03-17	1. 文档名更新为《Android 10.0 编译系统变化介绍》 2. 结构调整、内容优化、格式更新等

文档版本	发布日期	修改说明
V1.3	2020-09-15	升级文档模板。

Unisoc Confidential For hiar

目 录

1 概述.....	1
2 Clang 使用	2
3 BSP 单编.....	3
3.1 简介	3
3.2 编译流程变化	4
3.3 BSP 单编实现原理	5
3.4 Android 单独编译	5
4 Makefile 规则	6
4.1 .PHONY 使用	6
4.2 Android.mk 中移除 BUILD_NUMBER	6
4.3 弃用 LOCAL_MODULE_TAGS := eng debug	7
4.4 限制 HOST TOOLS 的使用	7
4.5 禁用 export 和 unexport 关键字	7
4.6 弃用 USER	7
4.7 弃用 `*.c.arm` / `*.cpp.arm`	7
5 UNISOC 工具	8
5.1 get_build_var	8
5.2 编译耗时统计	8
5.3 umake 命令	9
6 参考文档.....	10

1 概述

Android 10.0 的 build 系统的变化主要有如下：

- Android 的编译全面从 GCC 切换为 clang
- Makefile 中命令采用白名单制，不在白名单的 shell 命令无法使用，包括 make。鉴于此项限制，u-boot、kernel、ko 等编译从 android 中分离出来进行单独编译，统一存放到 bsp 目录下。
- Makefile 规则更加严格

Unisoc Confidential For hiar

2 Clang 使用

Clang 相对于 GCC 调试功能更加强大，而且 License 更加宽松，适合商业开发。因此从 Android 10.0 开始，Android 的 C/C++ 编译全面切换为 clang。

- Android.bp 组织的 module，都使用 clang 编译，不需要像下面这样单独的设置：

```
cc_library_shared {
    name: "libhwc2onladapter",
    vendor: true,

    clang: true,
    cflags: [
        "-Wall",
        "-Werror",
        "-Wno-user-defined-warnings",
    ],
}
```

如果设置 clang:false 的话，编译会报错，相关逻辑控制如下：

```
if c.Properties.Clang != nil && *c.Properties.Clang == false {
    ctx.PropertyErrorf("clang", "false (GCC) is no longer supported")
}
```

- Android.mk 组织的 module，默认也是采用 clang 编译，相关控制逻辑如下：

在 build/core/binary.mk 下定义使用的编译器

```
my_cxx := $(LOCAL_CXX)
ifeq ($(strip $(my_cxx)),)
    my_cxx := $(my_cxx_wrapper) $(CLANG_CXX)
endif
```

如果在 Android.mk 文件中定义 LOCAL_CXX，相关模块的编译器会替换掉 clang，采用自定义的编译器进行编译。

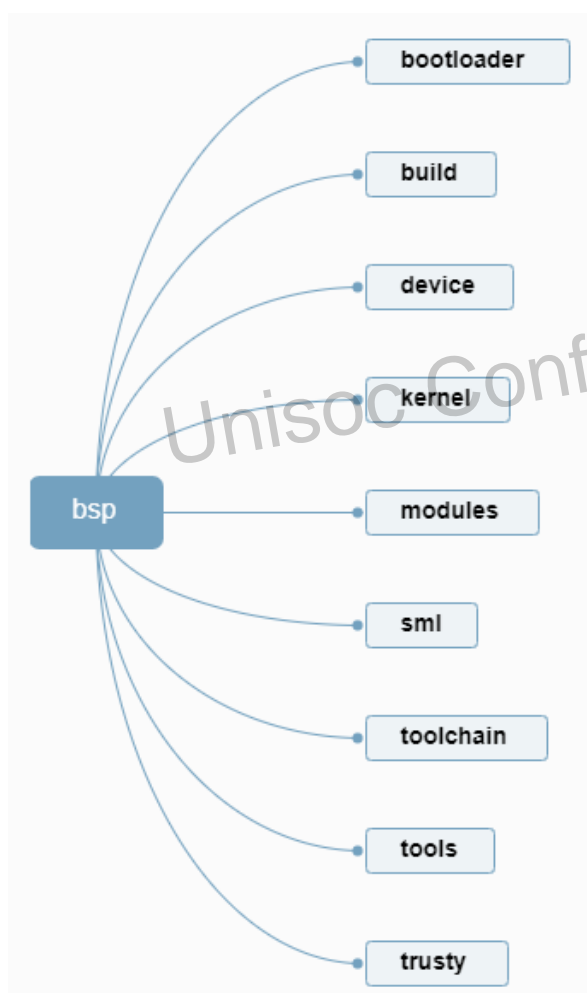
3

BSP 单编

3.1 简介

Android 10.0 开始，makefile 中 make 命令不允许使用，因此之前使用 make 命令进行编译的模块将无法编译。因此从 Android 10.0 开始，bootloader、kernel、driver 等编译放到 bsp 目录下。bsp 目录的结构如图 3-1 所示。

图3-1 BSP 目录结构



在 bsp 目录下的各个子目录主要用途如下：

- Bootloader: 存放 u-boot 和 chipram 源代码
- build: 存放 Kernel 单编的相关脚本
- device: board 配置

- kernel: kernel 源代码
- modules: kernel driver
- sml: arm-trusted-firmware
- toolchain: gcc 工具
- tools: 制作镜像等相关工具
- trusty: trusty 相关模块

3.2 编译流程变化

经过单编改造后，Android 编译流程有少许变化，主要变化如下：

- 全编，在输入 make 后(无其他后缀参数)，先编译 bsp 然后再进行 Android 代码编译

```
make
|--- bsp/build/make_for_android.sh $@
|--- _wrap_build $(get_make_command "$@" ) "$@"
```

- 在输入 make bootimage、recoveryimage、vendorimage、superimage、systemimage 时，需要先编译 bsp 中的依赖，具体如下：

```
make bootimage
|-- make kernel (bsp build part)
make recoveryimage
|-- make kernel (bsp build part)
|-- make dtbo (bsp build part)
make vendorimage
|-- make headers (bsp build part)
make superimage
|-- make all (bsp build part)
make systemimage
|-- make kernel (bsp build part)
|-- make chipram (bsp build part)
|-- make bootloader (bsp build part)
```

- 如果 make 后跟的参数不在下面列表中，bsp 不参与相关编译命令。

```
BSP_MODULES_LIST="
chipram
bootloader
sml
trusty
bootimage
dtboimage
recoveryimage
systemimage
sockoimage
odmkoimage
superimage
vendorimage
vbmataimage
"
```

3.3 BSP 单编实现原理

1. 添加 bsp 单编命令

在运行 build/envsetup.sh 时，函数 source_vendorsetup 运行，主要功能是查找 device vendor 和 product 目录下的'vendorsetup.sh'脚本（vendor/sprd/feature_configs/vendorsetup.sh 和/vendor/sprd/external/tools-build/vendorsetup.sh），如果脚本存在的话运行，代码如下：

```
function source_vendorsetup() {  
    ...  
    for dir in device vendor product; do  
        for f in $(test -d $dir && \  
            find -L $dir -maxdepth 4 -name 'vendorsetup.sh' 2>/dev/null | sort); do  
  
            if [[ -z "$sallowed" || "$sallowed_files" =~ $f ]]; then  
                echo "including $f"; . "$f"  
            else  
                echo "ignoring $f, not in $sallowed"  
            fi  
        done  
    done  
}
```

2. 在 vendor/sprd/feature_configs/vendorsetup.sh 中添加了 make 函数来协调 bsp 和 Android 的编译。

```
function aosp_make()  
{  
    bash $PWD/bsp/build/make_for_android.sh $@  
    if [ $? -ne 0 ]; then  
        return 1  
    fi  
  
    _wrap_build $(get_make_command "$@" ) "$@"  
}
```

bsp/build/make_for_android.sh 脚本的主要作用是根据 make 命令后面的参数决定是否要编译 bsp 里面的目标。

3. 在/vendor/sprd/external/tools-build/vendorsetup.sh 中主要是导出 kheader umk 等命令，主要是对 bsp 下相关模块的单编命令。

3.4 Android 单独编译

由于 bsp 单编的改造，make 命令已经被重写，造成有些目标在编译时必须首先要编译 bsp，在 bsp 未修改的场景下，每次编译都要编译 bsp 造成不必要的编译时间开销。对于想单独编译 Android 时，建议采用 m 命令。

\$m

4 Makefile 规则

本章节主要是总结 Android 10.0 升级后，对于日常使用比较多的场景变化，如果需要详细的修改可以参考 Android 源代码下的 build/make/Changes.md 文件。

4.1 .PHONY 使用

为了改善 Android 增量编译的速度和可靠性，google 规范了 .PHONY 规则的一些用法。

.PHONY 标识的目标通常作为实际目标的 shortcuts，以便为目标提供一个更加友好的名字，但是，这同时也意味着该目标始终被编译系统认为是 dirty 的，从而每次编译时都进行重新构建。对于 aliases 或者单次用户请求操作来说，这不是什么大问题，但如果一个真实的编译步骤依赖于 .PHONY 目标，对于一次小型的编译来说，这可能变得非常昂贵。

- .PHONY 标识的目标，其名称不得包含 “/”，例如：

```
test: foo/bar foo/baz
foo/bar: .KATI_IMPLICIT_OUTPUTS := foo/baz
foo/bar:
    @echo "END"
.PHONY: test foo/bar
```

Makefile:4: *** PHONY target "foo/bar" looks like a real file (contains a "/")

- 目标是一个实际存在的文件，但位于 out 目录之外。Google 要求编译系统的所有输出都在 out 目录中，否则，`m clean` 将无法彻底清除到之前编译的中间文件，进而可能对之后的编译造成不可预知的影响。

...mk:42: warning: writing to readonly directory: "kernel-modules"

- out/foo 看起来像一个真实的文件，因此，它不可以依赖于其它伪目标，例如：

```
.PHONY: test
test: out/foo
out/foo: bar
```

Makefile:4: *** real file "out/foo" depends on PHONY target "bar"

4.2 Android.mk 中移除 BUILD_NUMBER

Android.mk 文件不得使用 BUILD_NUMBER，因为每次 BUILD_NUMBER 发生变化时都需要重新读取。如果可以的话，最好直接去除，确实需要使用的話，可用 BUILD_NUMBER_FROM_FILE 替代。例如：

```
$(LOCAL_BUILT_MODULE):
    mytool --build_number $(BUILD_NUMBER_FROM_FILE) -o $@
```

BUILD_NUMBER_FROM_FILE 的影响范围限制在子 shell 内。

4.3 弃用 LOCAL_MODULE_TAGS := eng debug

LOCAL_MODULE_TAGS 的值 eng 和 debug 正在废弃中。他们允许模块自行决定是否安装在 eng/userdebug 类型的编译中。这与产品指定安装哪些模块的设定产生了冲突，使得精简产品配置变得困难。

使用 PRODUCT_PACKAGES_ENG 以及 PRODUCT_PACKAGES_DEBUG 可以达到同样的目的。

4.4 限制 HOST TOOLS 的使用

编译系统开始限制 host 操作系统中各种工具在编译过程中的使用。这将有助于在不同的机器上编译出同样的目标文件，进而尽可能消除隐藏的问题。首先，将系统默认 PATH 替换为指定的目录列表，任何不在这些目录列表中的工具在使用时将会导致错误，除非配置为允许使用（随着时间的推移，这将会越来越严格）。

默认配置位于 build/soong/ui/build/paths/config.go，其中包含了大多数编译过程中所使用的到的所有通用工具。

4.5 禁用 export 和 unexport 关键字

Android 10.0 编译系统中，export 和 unexport 已被禁用，根据使用场景这将触发错误或者警告。具体来说，在编译的早期阶段，即，产品配置以及读取 BoardConfig.mk 时，将会弹出警告（将来会变成错误）。而在解析 Android.mk 以及后续处理时，这将会弹出错误。

4.6 弃用 USER

随着 Android 编译逐步沙盒化，USER 很快将意味着“nobody”。在大多数情况下，没必要知道谁在执行编译，如果需要，可以使用 make 变量 BUILD_USERNAME。类似的，hostname 工具将返回在 Android 编译时将返回同样的值，真正的值可以通过 BUILD_HOSTNAME 获取。

4.7 弃用 `*.c.arm` / `*.cpp.arm`

以前在 Android.mk 文件中可以通过为原文件添加 .arm 后缀来指定 LOCAL_ARM_MODE。Soong 不支持这种不常见的行为，因而优先考虑使用 arm 而不是 thumb 来编译整个库。目前，Android.mk 文件中也进行类似要求。

5 UNISOC 工具

紫光展锐添加了一些小工具以便于定位 Android 编译中的问题，从而提高编译的效率。

5.1 get_build_var

get_build_var 作用是获取 mk 文件中变量的值，使用格式：get_build_var VARIABLE_NAME

get_build_var 定义在 envsetup.sh 中，执行完 source build/envsetup.sh，lunch 对应工程后，就可以使用了。例如：

```
$ get_build_var BOARD_VNDK_VERSION
```

最后输出结果，值为“current”，在 device/sprd/sharkl3/common/DeviceCommon.mk 定义。

```
build/make/core/dumpvar.mk:28: warning: var resolved:BOARD_VNDK_VERSION~build/make/core/config.mk
device/sprd/sharkl3/common/DeviceCommon.mk~current
current
```

说明

对于 get_build_var 得到的目录有不准现象，获取的 path 仅供参考。

5.2 编译耗时统计

编译耗时统计功能可以统计 bsp 编译，Android 编译及 image 签名的耗时并打印，其中会细化 Android 编译耗时，包括 regen，soong build，kati build 等的耗时。

在 make 执行完成后，最后会输出相关统计。

```
=====BUILD TIME ANALYSIS=====
[ninja regen reason]
1: status: out/build-s9863a1h10_Natv-cleanspec.ninja is missing, regenerating...
2: status: out/build-s9863a1h10_Natv.ninja is missing, regenerating...
3: status: out/build-s9863a1h10_Natv-package.ninja is missing, regenerating...
[total build time 1:53:49 (hh:mm:ss)]
bsp build time: 0:14:41(12.90%)
android build time: 1:34:21(82.90%)
  [android build]regen: 0:02:28(2.61%)
  [android build]real build: 1:31:53(97.39%)
sign images time: 0:04:47(4.20%)
[android build time details]
startup: 45.38ms(0.00%)
find modules: 155.28ms(0.00%)
path: 433.40ms(0.01%)
dumpvars: 1,157.31ms(0.02%)
[soong]blueprint bootstrap: 383.54ms(0.01%)
```

```
[soong]environment check: 0.07ms(0.00%)
[soong]minibp: 1,250.23ms(0.02%)
[soong]bpglob: 391.39ms(0.01%)
[soong]minibootstrap: 245.92ms(0.00%)
[soong]bootstrap: 38,944.60ms(0.57%)
soong: 41,215.99ms(0.60%)
kati cleanspec: 2,411.65ms(0.04%)
kati build: 101,916.23ms(1.49%)
kati package: 404.01ms(0.01%)
ninja: 5,512,976.12ms(80.73%)
[build command]
build/soong/soong_ui.bash --make-mode KALLSYMS_EXTRA_PASS=1 -j32
=====BUILD TIME ANALYSIS=====
```

5.3 umake 命令

在只修改源码，未修改.mk 或.bp 文件时（大多数编译也是此类），使用 **umake** 命令来替代 **make** 命令可以提高编译的速率。

umake 使用方式：**umake module_name**，如 **umake SystemUI**

Unisoc Confidential For hiar

6

参考文档

1. Android Bootcamp 2019 - Platform Build System
2. Android Bootcamp 2018 - Soong Build System

说明

以上文档从谷歌获取。

Unisoc Confidential For hiar