openwrt 系统的编译框架分析

第一章 openwrt 概述

1.1 openwrt 简介

OpenWrt 是一个比较完善的嵌入式 Linux 开发平台，最初版本是由 Linsys 基于 Linux 内核开发，并搭载在其旗下的一款路由器上。OpenWrt 在增加软件包方面使用极其方便，编译框架的优点在于模块解耦，方便配置，易于扩展。QSDK、SLP 均是基于 OpenWRT，学习 OpenWRT 的编译框架，定制我司的 Linux 产品开发平台。

1.2 openwrt 目录结构

1.2.1 原版 openwrt 目录结构

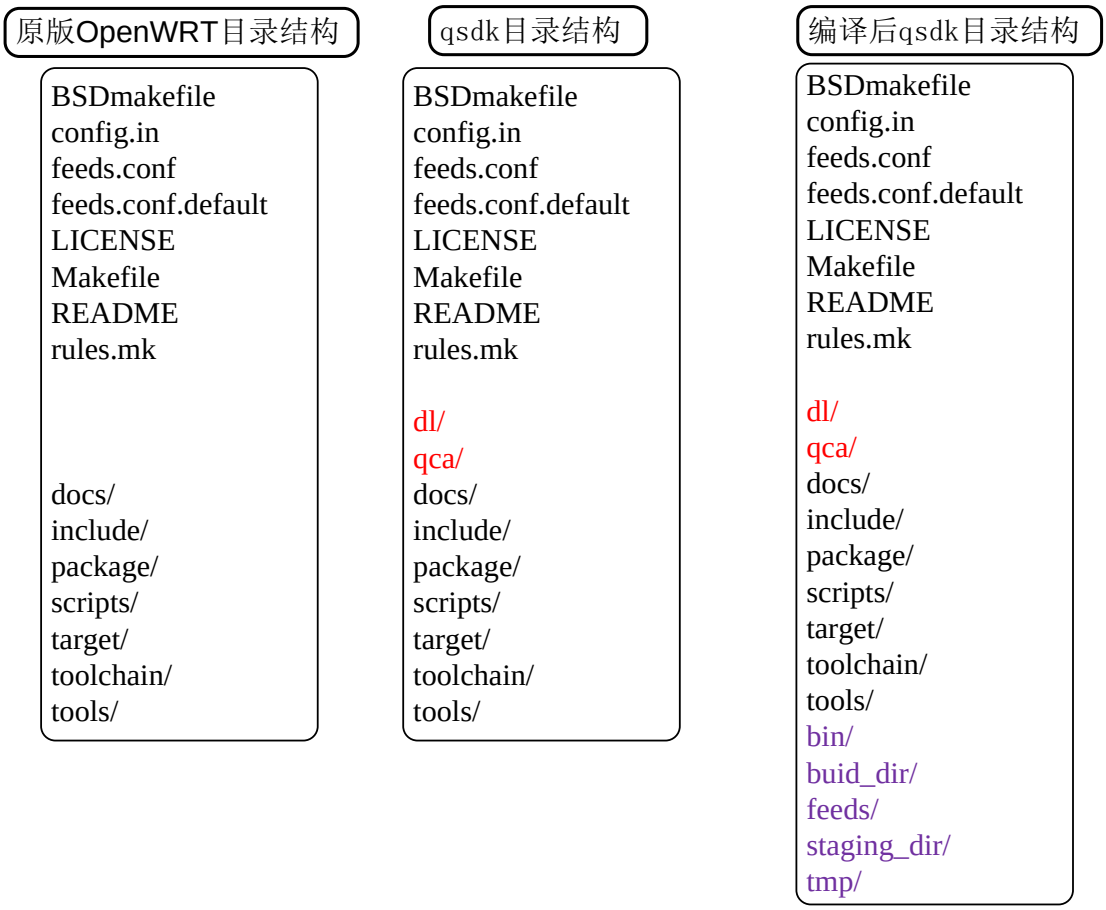


图 1-1 原版 openwrt 和 QSDK 目录结构

如图 1-1 所示，最左边为原版 openwrt 的目录结构，仅包含编译所需的基本框架。

1.2.2 QSDK 目录结构

如图 1-1 所示，中间部分为编译之前的 QSDK 目录结构，在原版 openwrt 的

基础上增加目录 dl 和 qca，添加了芯片支持、驱动代码、指定的 profile 和一些修改的 patch。图 1-1 右边部分为编译之后的 QSDK 目录结构，图中紫色部分为编译过程中增加的目录。相关目录的功能描述见 1.2.4。

1.2.3 SLP 目录结构

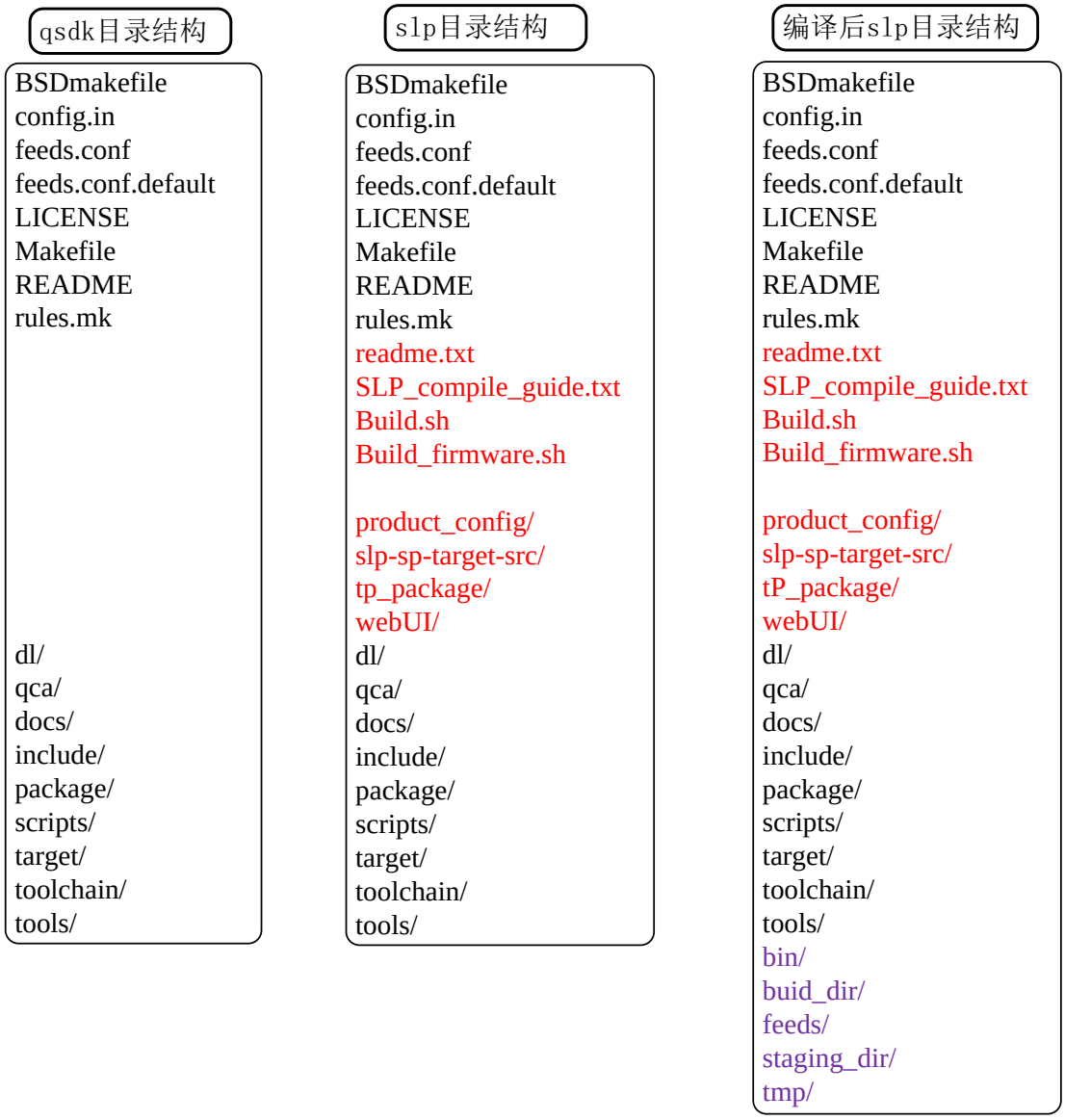


图 1-2 QSDK 和 SLP 目录结构

SLP 是根据产品线开发需要，在 QSDK 的基础上进行改进版本。图 1-2 中间部分为编译前的 SLP 目录结构，SLP 在 QSDK 的基础上增加 product_config、slp-sp-target-src、tp_package。product_config 放置产品型号的配置信息，tp_package 目录下放置定制的 package 包，slp-sp-target-src 目录下放置 mtk 和 qca 芯片 wifi 驱动等。图 1-2 中右部分为编译之后的目录结构，其中，紫色部分为编译过程中增加的目录。

1.2.4 重要目录介绍

表 1-1 openwrt 系统重要目录

Openwrt 系统的基础目录	
Makefile	控制项目编译过程，根据配置文件，指明依赖关系，指定编译流程。
rules.mk	定义重要的全局变量
docs	说明文档
include	编译过程中需要的基础 makefile 文件，包括 host, kernel, package 等编译规则，debug, quilt, download 等基础规则
scripts	构建过程中用到的脚本/工具
tools	包含构建过程中需要在 Host 机器上使用到的工具程序，每个工具包含 Makefile 和 patch(可选)，用于编译相应的工具。
toolchain	工具链，包含编译 kernel-headers, uclibc, binutils, gcc, gdb 等工具链的包，每个包含 Makefile、patch(可选)和其他 file
package	各种软件包
target	各种 CPU 平台，定义不同平台 firmware 和 kernel 的编译过程
QSDK、SLP 增加的目录	
dl	存放下载好的源码包(省去编译时自动下载、解压、patch)
qca	qca 针对特定平台提供的 profile 文件；qca 添加的 patch 文件。
product_config	产品型号的配置信息
tp_package	定制的 package 包
slp-sp-target-src	放置 mtk 和 qca 芯片 wifi 驱动等
编译过程中增加的目录	
feeds	从源上同步下来的 package
build_dir	在此目录中进行构建，所有软件包在 build_dir 下进行编译
staging_dir	安装 host 工具及 toolchain
tmp	存放目录扫描得到的缓存信息，主要扫描 package 和 target
bin	存放构建好的固件(xxx.bin)及 ipk (xxx.ipk)

1.3 特别说明

本文档的分析基于 QSDK，内核版本 LINUX_VERSION=3.3.8，体系架构为 ARCH=mips，板子型号为 BOARD= ar71xx。为便于阅读，文档中很多全局变量使用了具体化的参数。比如内核的编译路径(LINUX_DIR)，在 Makefile 文件中定义为$(KERNEL_BUILD_DIR)/linux-$(LINUX_VERSION)，文档一般使用具体化的内容：(LINUX_DIR)= build_dir/linux-ar71xx_generic/linux-3.3.8。当体系架构不同、板子型号不同，相应的(LINUX_DIR)会不同，请读者注意。

在没有特别说明的前提下，文档的文件路径省去 QSDK 根目录(TOPDIR)，只写出从 QSDK 根目录开始的相对目录。

如：(LINUX_DIR)= (TOPDIR)/build_dir/linux-ar71xx_generic/linux-3.3.8；通用写成：(LINUX_DIR)= build_dir/linux-ar71xx_generic/linux-3.3.8

第二章 openwrt 编译系统的主体框架

本章主要论述主 Makefile 的框架结构，分析主 Makefile 及其包含文件在编译目标定义、编译流程执行等过程中的作用。总结顶层输入带不同目标的编译命令时，系统的命令解析与编译执行的流程。

2.1 主 Makefile 的架构

Makefile 用于控制项目编译过程，根据配置文件，指明依赖关系，指定编译流程。openwrt 系统的主 Makefile 包括三个部分：前导部分、首次执行部分、再次执行部分。前导部分主要是定义全局变量；首次执行部分完成编译的基础准备工作；再次执行部分完成目标编译。

```
# 顶层 Makefile
```

前导部分

```
world:

ifneq ($(OPENWRT_BUILD),1)
  override OPENWRT_BUILD=1
  export OPENWRT_BUILD
  include $(TOPDIR)/include/toplevel.mk
else
prepare: .config $(tools/stamp-install) $(toolchain/stamp-install)

world: prepare $(target/stamp-compile) $(package/stamp-cleanup)\
        $(package/stamp-compile) $(package/stamp-install)\
        $(package/stamp-rootfs-prepare) $(target/stamp-install)
    $(_SINGLE)$(SUBMAKE) -r package/index

.PHONY: prepare world
endif
```

首次执行部分

再次执行部分

图 2-1 Makefile 的主体架构

说明：

1、上图中，只是提取主 Makefile 的部分信息，include 的.mk 文件、clean 编译目标等均未全部展示出来。

2、 为从整体展示 openwrt 的编译架构，2.2 节介绍输入 make 命令，没有带入其他目标的编译执行过程。

3、 2.3 和 2.4 节将分别介绍主 Makefile 首次执行部分和再次执行部分，从主体架构、目标关系等层面进行阐释。

4、 2.5 节将对 Makefile 的文件包含关系和目标依赖关系进行归纳总结

2.2 编译执行的总体流程

本节的主要目的是概述单次整体编译的执行过程，作为对 2.1 节的功能性解释，从整体展示 openwrt 系统的编译过程。

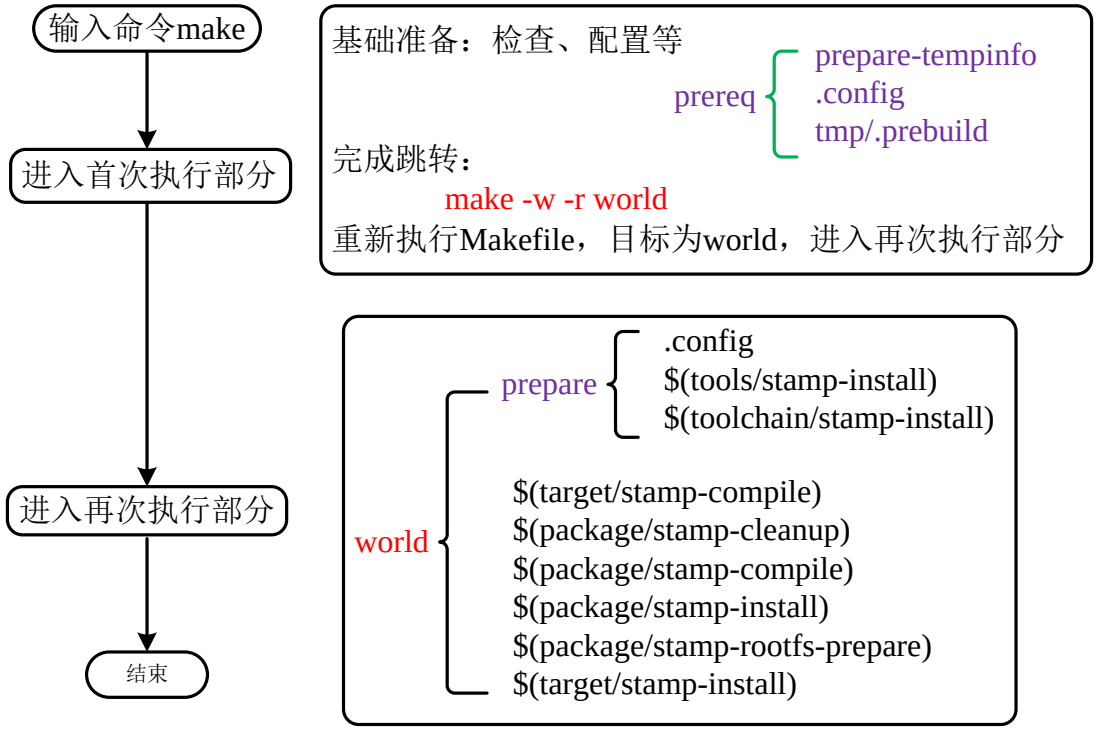


图 2-2 编译执行的总体流程

说明：

1、此编译执行的总体流程图是以生成固件为目标的执行过程，输入命令为 make V=s/1/99(可选项)，输入命令没有带入目标，以 world 为默认目标。

2、当输入命令带入目标时，如 make menuconfig、make clean，执行过程不完全一致，后面进行详细说明。

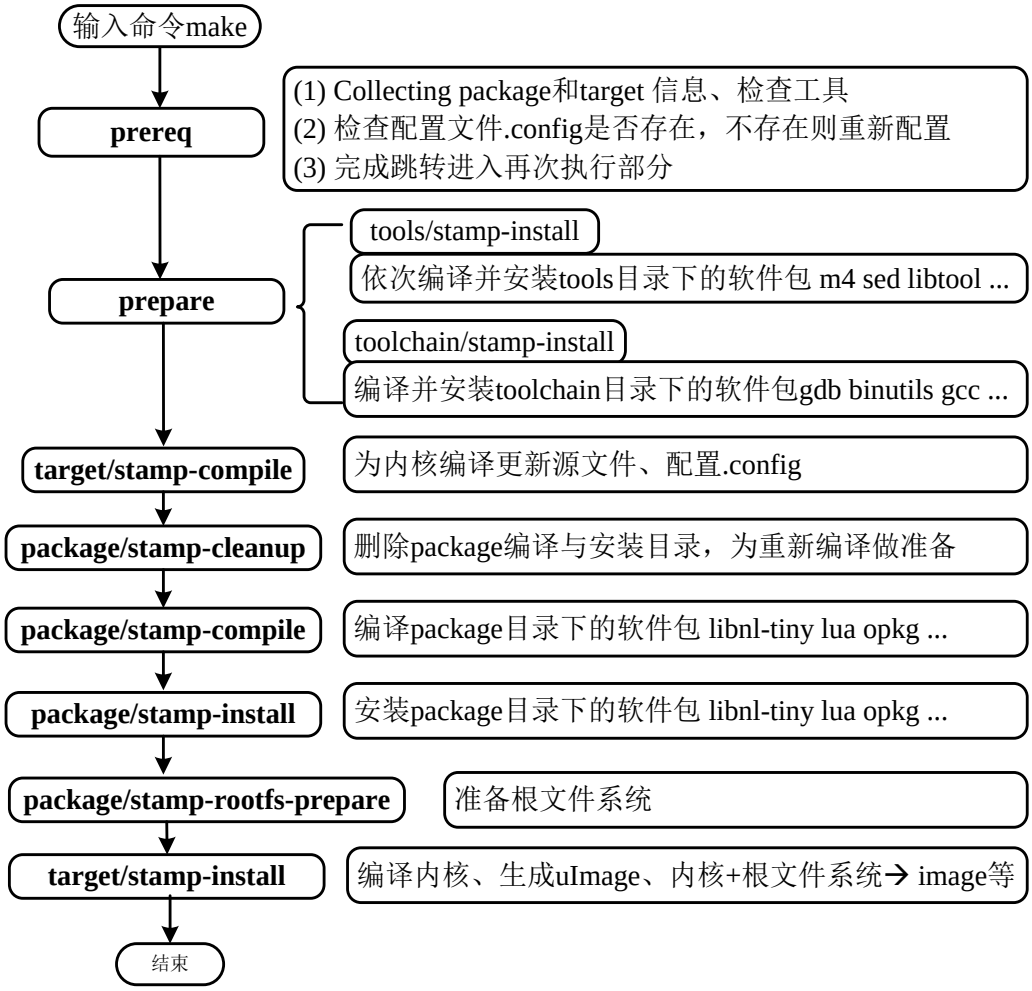


图 2-3 完整编译的流程与功能简述

说明：

1、图 2-3 所示的编译过程，输入命令为 make，或者 make V=s/1/99(控制打印信息)，不带其他编译目标，以 world 为默认目标

2、图左边所示为 openwrt 系统首次完整编译的流程，右边是不同阶段的目标的功能简述，是以生成固件为目标的执行过程。

3、非首次编译时，tools 和 toolchain 软件包不会重复编译。

2.3 首次执行部分

```
# 顶层 Makefile 首次执行部分

  empty:=
  space:= $(empty) $(empty)
  _SINGLE=export MAKEFLAGS=$(space);

  override OPENWRT_BUILD=1
  export OPENWRT_BUILD
  GREP_OPTIONS=
  export GREP_OPTIONS
  include $(TOPDIR)/include/debug.mk
  include $(TOPDIR)/include/depends.mk
  include $(TOPDIR)/include/toplevel.mk
```

控制下次执行Makefile时进入再次执行部分

include files

```
# include/toplevel.mk

%::
    $(PREP_MK) $(NO_TRACE_MAKE) -r -s prereq
#   = OPENWRT_BUILD= QUIET=0 make V=s -r -s prereq
    $(SUBMAKE) -r $@
#   = umask 022; make -w -r world

  prereq::                  tmp/.prereq-build:
  prepare-tmpinfo:          .config:              config:
  scripts/config/conf:      scripts/config/mconf:
  defconfig:                menuconfig:
  kernel_menuconfig:        kernel_nconfig:
  printdb:                  download:
  clean dirclean:           distclean:
  help:                     docs docs/compile:
```

默认编译目标

执行语句

其他目标

图 2-3 首次执行部分的主体架构

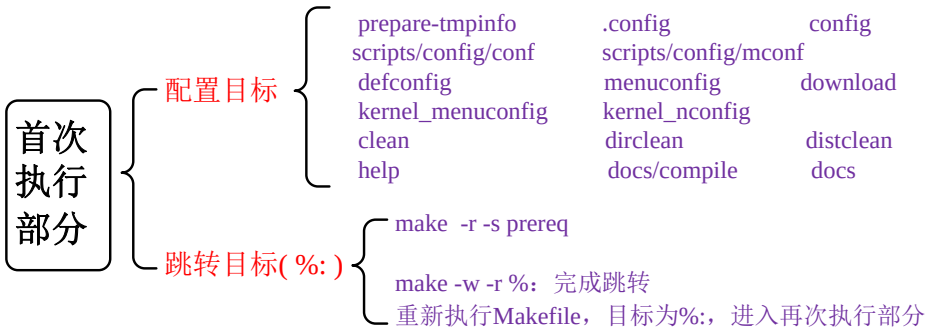


图 2-4 首次执行部分的目标

说明：

1、 首次执行部分的目标定义在文件 include/toplevel.mk 中。

2 、输入命令不带目标时，"%:"代表默认目标 world。默认编译目标(%:)的最后一条执行语句，重新执行 Makefile 文件，目标为 world，由于 OPENWRT_BUILD=1，会进入再次执行部分。

3、 当输入命令带目标时，首先在配置目标集合里面查找，如果可以找到，就编译相关

目标。如输入目标为：make menuconfig，会找到相应目标进行编译。

4、当输入命令带目标，且目标在首次执行部分没有找到，"%:"代表目标值，会再次执行 Makefile，进入再次执行部分，搜寻目标。

5 、输入命令带目标的编译执行流程见第 4 章详述

6、目标为 world 时，首次执行部分的目标关系图见图 2-5

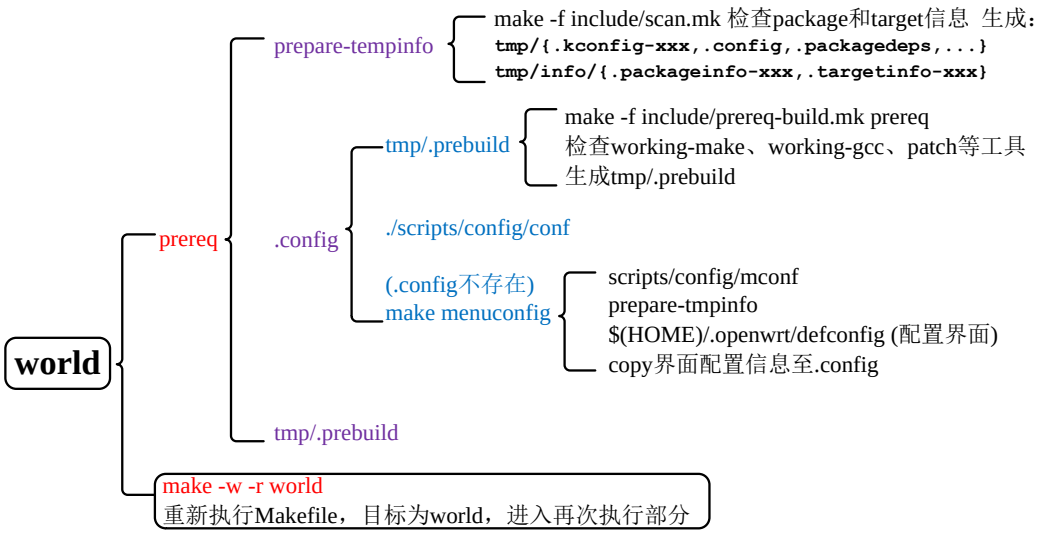


图 2-5 目标为 world 时首次执行部分的目标关系图

说明：

1、首次执行部分主要完成基础准备(prereq)和编译跳转。

2、基础准备主要编译目标为 prepare-tempinfo、.config、tmp/.prebuild。prepare-tempinfo 检查 package 和 target 信息，检查结果记录在 tmp 目录下。tmp/.prebuild 检查 patch、svn、working-gcc 等编译工具。

3、开发者在配置界面（make menuconfig），选择的编译目标，如 linux，sdk，imagebuilder，toolchain，也可以选择是否编译某一个 package 包， openwrt configuration 信息保存在文件.config 内，当此文件缺失时，执行 make menuconfig，.config 保存配置信息。

4、编译跳转执行 make world，重新执行 Makefile 文件，目标为 world，进入再次执行部分。

2.4 再次执行部分

```
# 顶层 Makefile 再次执行部分
include rules.mk
include $(INCLUDE_DIR)/depends.mk
include $(INCLUDE_DIR)/subdir.mk
include target/Makefile
include package/Makefile
include tools/Makefile
include toolchain/Makefile

$(toolchain/stamp-install): $(tools/stamp-install)
$(target/stamp-compile): $(toolchain/stamp-install)
$(package/stamp-cleanup): $(target/stamp-compile)
$(package/stamp-compile): $(target/stamp-compile)
$(package/stamp-install): $(package/stamp-compile)
$(package/stamp-rootfs-prepare): $(package/stamp-install)
$(target/stamp-install): $(package/stamp-compile)

world: prepare $(target/stamp-compile) $(package/stamp-cleanup) \
       $(package/stamp-compile) $(package/stamp-install) \
       $(package/stamp-rootfs-prepare) $(target/stamp-install) FORCE
   $(_SINGLE)$(SUBMAKE) -r package/index

dirclean:
prereq: $(target/stamp-prereq) tmp/.prereq_packages
prepare: .config $(tools/stamp-install) $(toolchain/stamp-install)
package/symlinks:
package/symlinks-install:
package/symlinks-clean:
.PHONY: clean dirclean prereq prepare world package/symlinks \
        package/symlinks-install package/symlinks-clean
```

include files

定义动态目标和子目录编译规则

动态目标的依赖关系

默认目标

其他目标

图 2-6 再次执行部分的主体架构

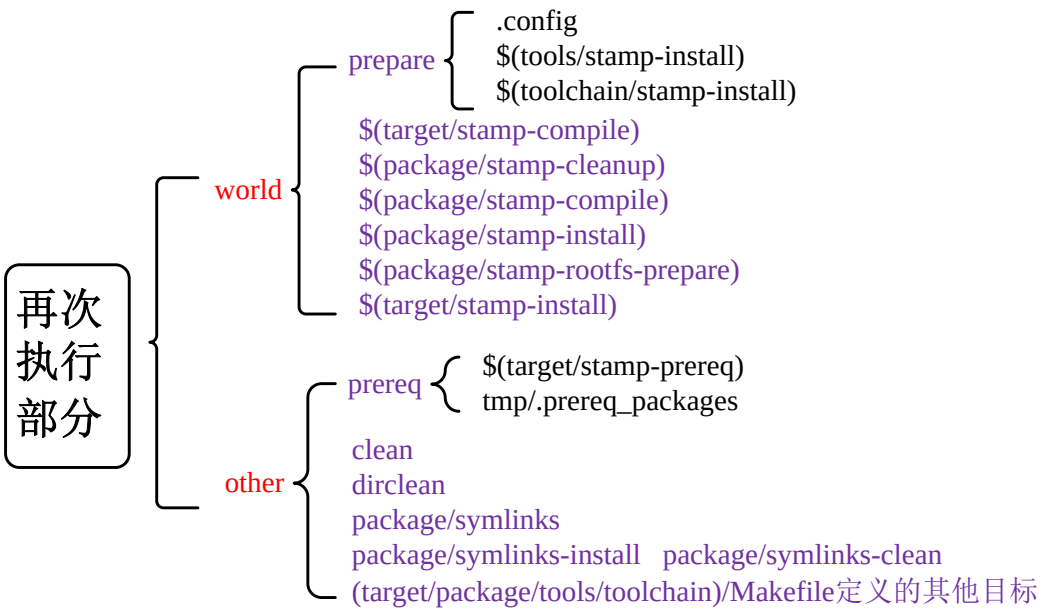


图 2-7 再次执行部分的目标关系图

说明：

1、 主 Makefile 的再次执行部分，完成了绝大部分的编译目标的定义。

2、输入命令只有 make，没有带其他目标时，跳转进入再次执行部分时，以 world 为默

认目标，依次编译图 1-6 的上半部分，诸如$(tools/stamp-install)等动态生成目标及其编译规则，由如下四个文件完成：target/Makefile、package/Makefile、tools/Makefile、toolchain/Makefile

3、 target/Makefile、package/Makefile、tools/Makefile、toolchain/Makefile 不仅定义图 1-6 上半部分的动态目标，还会定义 target、package、tools、toolchain 目录下编译包的编译目标及其规则。例如，tools 目录下有包 m4，tools/Makefile 会定义子目录编译目标：tools/m4/prereq、tools/m4/prepare、tools/m4/compile、tools/m4/install、tools/m4/clean 等，还会定义这些子目录编译目标的编译规则。子目录编译目标的定义与编译规则将在第三章详尽描述。

2.5 主 Makefile 总结

主 Makefile 是控制 openwrt 系统编译实现过程最核心的文件，是系统内所有编译目标的入口。图 2-8 中描述了 Makefile 的文件包含关系和主目标依赖关系。

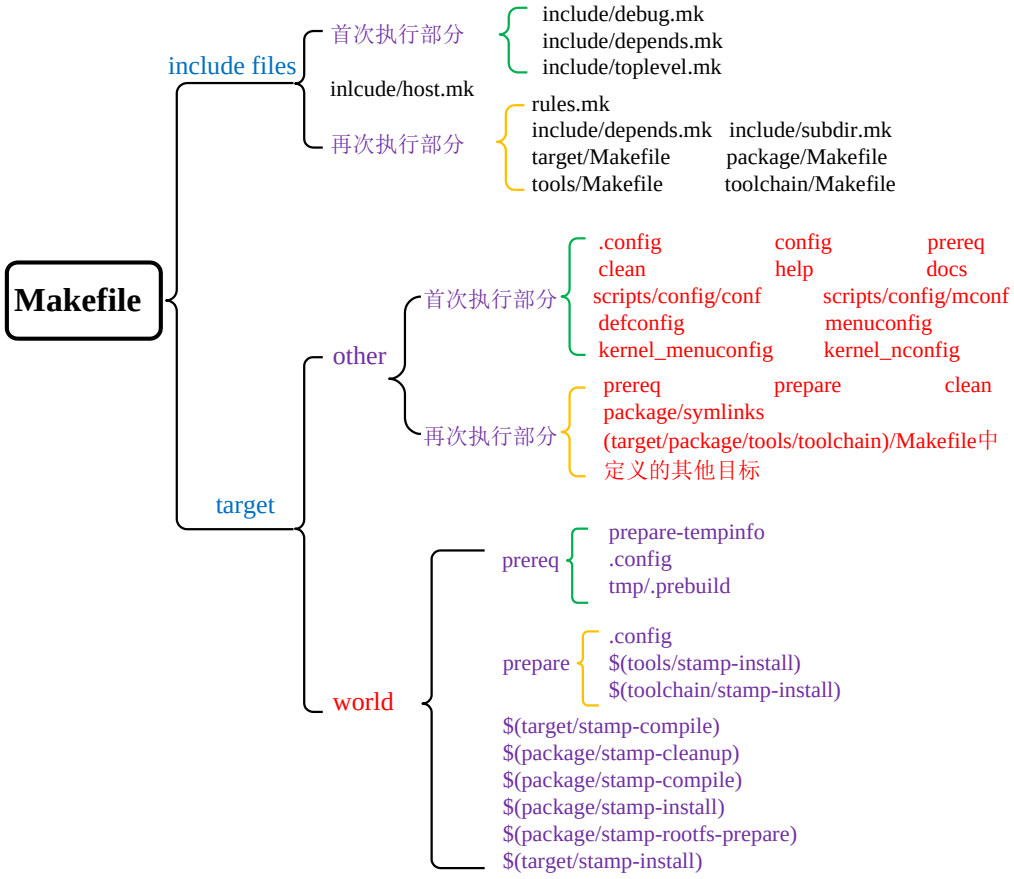


图 2-8 Makefile 的文件和目标

说明：

1、主 Makefile 主要职能是 include files 和定义目标依赖关系，部分目标不是在主 Makefile 里面定义，而是在主 Makefile 引入的文件中定义。include/toplevel.mk 定义了首次执行部分的目标，menuconfig 和跳转目标(%:)等等。tools/Makefile 定义了动态生成目标

$(tools/stamp-install)和 tools 目录下其他包的编译目标(如:tools/m4/install)。(target/package/toolchain)/Makefile 的功能类似。

2、红色字体标识目标，包括默认目标 world、首次执行部分目标(如:menuconfig)和再次执行部分目标(如:package/symlinks)。

3、上图中绿色括弧内为首次执行部分包含的文件和定义的目标，橙色括弧内为再次执行部分包含的文件及定义的目标。

4、不同的输入目标不同的编译执行流程，输入目标之后的编译流程大致如图 1-9 所示。

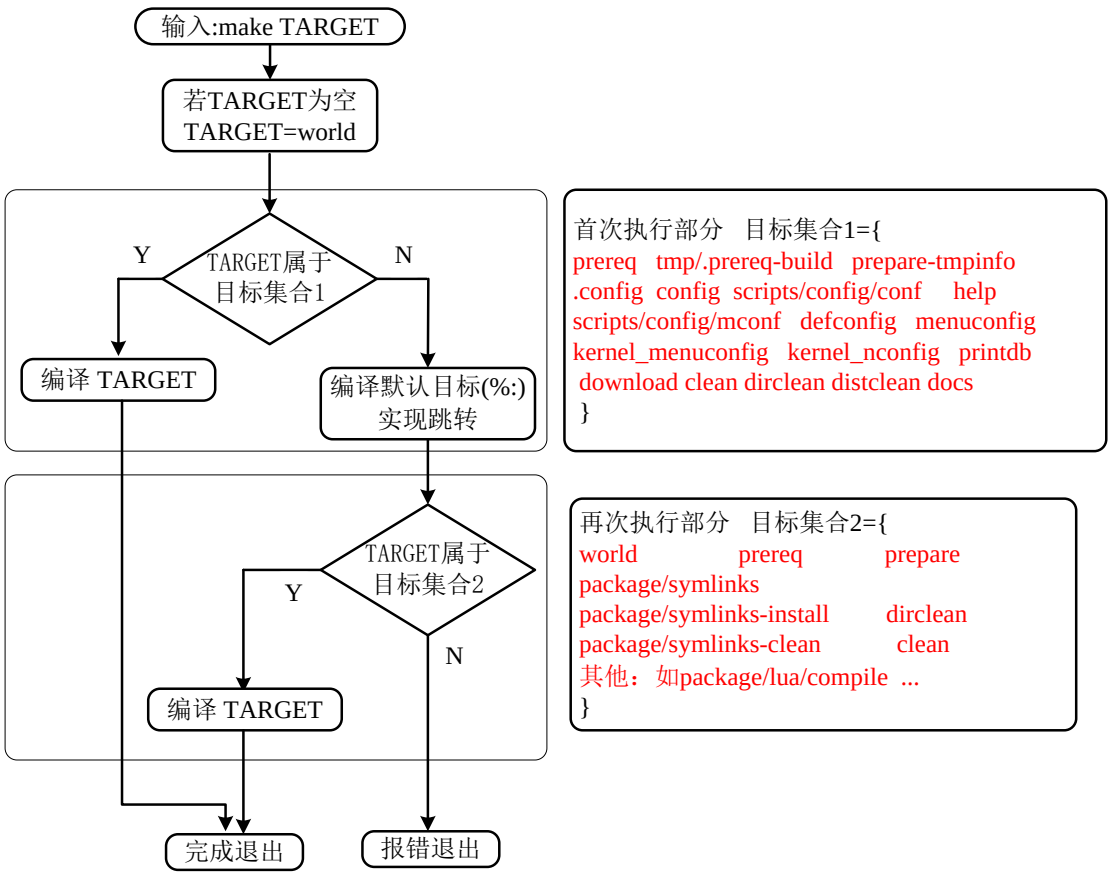


图 2-9 make 命令的解析流程

说明：

1、主 Makefile 实现了系统所有编译目标的定义，当输入命令：make TARGET，如果 TARGET 在图 2-9 所述的目标集合 1 中，则直接编译，完成之后直接退出，不进入再次执行部分。

2、如果 TARGET 不在图 2-9 所述的目标集合 1 中，先执行默认编译目标(%:)，完成跳转，第二次执行 Makefile，进入 Makefile 的再次执行部分。

3、如果 TARGET 在图 2-9 右图所述的目标集合 2 中，则编译执行 TARGET，如果 TARGET 在目标集合 1/2 均没有定义，则系统会退出并报错“make[1]: No rule to make target ` TARGET '. Stop.”

第三章 动态目标与子目录编译规则

本章首先分析动态目标生成函数 stampfile 和子目录编译目标生成函数 subdir。在此基础上，总结 tools/Makefile、toolchain/Makefile、package/Makefile、target/Makefile 调用函数 stampfile 和 subdir，实现动态目标定义、子目录编译目标及其编译规则定义。进而说明主 Makefile 包含 tools/Makefile、toolchain/Makefile、package/Makefile、target/Makefile，在动态目标生成和子目录切换编译过程中的控制机理。

3.1 函数 stampfile 和 subdir 详解

3.1.1 动态目标生成函数 stampfile

stampfile (in include/subdir.mk, # Parameters: <subdir> <name> <target> <depends> <config options> <stampfile location>)，这个函数参数比较多，根据不同的参数构建不同的编译目标，加速 make 过程，方便检查时间戳。常用在顶层 makefile 中，如 package/Makefile，tools/Makefile。

```
# include/subdir.mk 函数stampfile

# Parameters: <subdir> <name> <target> <depends> <config options> <stampfile location>
# Parameters:   $(1)     $(2)    $(3)      $(4)          $(5)               $(6)
define stampfile
  $(1)/stamp-$(3):=$(if $(6),$(6),$(STAGING_DIR))/stamp/.$(2)_$(3)$(if $(5),_$(call confvar,$(5)))
  $$($(1)/stamp-$(3)): $(TMP_DIR)/.build $(4)
    @+$(SCRIPT_DIR)/timestamp.pl -n $$($(1)/stamp-$(3)) $(1) $(4) || \
        $(MAKE) $(if $(QUIET),--no-print-directory) $$($(1)/flags-$(3)) $(1)/$(3)
    @mkdir -p $$$$(dirname $$($(1)/stamp-$(3)))
    @touch $$($(1)/stamp-$(3))
  $$(if $(call debug,$(1),v),,.SILENT: $$($(1)/stamp-$(3)))

  .PRECIOUS: $$($(1)/stamp-$(3)) # work around a make bug

  $(1)//clean:=$(1)/stamp-$(3)/clean
  $(1)/stamp-$(3)/clean: FORCE
    rm -f $$($(1)/stamp-$(3))
endef
```

P1 P2 P3

图 3-1 函数 stampfile

说明：

函数 stampfile 的核心作用是定义动态目标$(1)/stamp-$(3)，函数功能的详尽说明图 3-2，函数的使用实例见图 3-3。

P1的功能

为 $(1)/stamp-$(3)赋值

P2的功能

定义编译目标、依赖关系、执行语句
编译目标: $$($(1)/Stamp-$(3))
依赖：$(TMP_DIR)/.build $(4)
执行：(1)：make $$($(1)/flag-$(3)) $(1)/$(3) (2): touch 目标文件

P3的功能

定义其他目标：$(1)//clean $(1)/stamp-$(3)/clean

图 3-2 函数 stampfile 的功能描述

例：$(eval $(call stampfile,$(curdir),tools,install,,CONFIG_CCACHE CONFIG_powerpc CONFIG_GCC_VERSION_4_5 CONFIG_GCC_USE_GRAPHITE CONFIG_TARGET_orion_generic))
$(1)=$(curdir)=tools $(2)=tools $(3)=install $(4)= $(6)=
$(5)=CONFIG_CCACHE CONFIG_powerpc CONFIG_GCC_VERSION_4_5 CONFIG_GCC_USE_GRAPHITE CONFIG_TARGET_orion_generic

$(1)/stamp-$(3)=系统路径/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.tools_install_nnnnn

```
$$(tools/stamp-install): $(TMP_DIR)/.build
	@+$(SCRIPT_DIR)/timestamp.pl -n $$(tools/stamp-install) tools || \
		$(MAKE) $(if $(QUIET),--no-print-directory) $$(tools/flags-install) tools/install
	@mkdir -p $$$$(dirname $$(tools/stamp-install))
	@touch $$(tools/stamp-install)
```

效果：定义了动态目标：$$(tools/stamp-install) 编译时会比较时间戳，确定是否重新:
make $$(tools/flags-install) tools/install

图 3-4 函数 stampfile 的使用实例

3.1.2 子目录编译目标生成函数 subdir

subdir（in include/subdir.mk, # Parameters: <subdir>），针对子目录生成标准的 make 目标，以目录名(subdir)作为参数调用此函数。主要用在 package、tools、toolchain、target 的 Makefile 中。

```
# include/subdir.mk 函数subdir的重要语句摘录

SUBTARGETS:=clean download prepare compile install update \
            refresh prereq dist distcheck configure

define subdir
 $(foreach bd,$($(1)/builddirs),                                    P1
   $(foreach target,$(SUBTARGETS),
     $(1)/$(bd)/$(target):
       $(if $(BUILD_LOG),@mkdir -p $(BUILD_LOG_DIR)/$(1)/$(bd))
       $$(SUBMAKE) -C $(1)/$(bd) $(target) BUILD_VARIANT="")

 $(foreach target,$(SUBTARGETS),$(call subtarget,$(1),$(target)))   P2
endef
```

图 3-4 subdir 的重要语句摘录

说明：

1、图 3-4 中变量 SUBTARGETS 在 subdir.mk 中定义，包括所有的 make 常见目标。

2、图 3-4 中只摘录了函数 subdir 的部分重要语句，详细信息请阅读 include/subdir.mk 文件。

3、函数 subdir 的功能主要分为两部分，如图中 P1 和 P2 所示。P1、P2 的功能描述见图 3-5，函数的使用实例见图 3-6。

P1的功能

为$(1)目录下所有的子目录($(bd))，定义：编译目标 + 执行语句
编译目标($(1)/$(bd)/$(target))
编译执行语句(make -C $(1)/$(bd) $(target))
进入目录$(1)/$(bd) 执行$(1)/$(bd)/Makefile文件 目标为$(target)

P2的功能

为$(1)/$(target)定义依赖关系：
$(1)/$(target): $($(1)/) $(1)/$(bd)/$(target)
说明：此处的$(bd)不是所有的子目录，而是系统确定编译的子目录，在$(1)/Makefile中配置。编译子包(子目录)的依据如下：(优先级)
$(1)/builddirs-$(target) > $(1)/builddirs-default > $(1)/builddirs

图 3-5 函数 subdir 的功能描述

例：$(eval $(call subdir,tools))
tools/builddirs为tools目录下的所有子目录,包括m4、mtools、squashfs4 …
tools/builddirs-default为需要编译的子目录，包括m4，不包括mtools
P1:
tools/m4/clean (执行: make -C tools/m4 clean) tools/m4/download tools/m4/prepare …
tools/mtools/clean tools/mtools/download tools/mtools/prepare …
tools/squashfs4/clean tools/squashfs4/download tools/squashfs4/prepare …
…
P2:
tools/clean : tools/ tools/m4/clean tools/squashfs4/clean … （没有tools/mtools/clean ）
tools/prepare : tools/ tools/m4/prepare tools/squashfs4/prepare …
tools/install : tools/ tools/m4/install tools/squashfs4/install …
…
效果：make tools/install时，会编译tools/m4/install等，不会编译tools/mtools/install
编译单个软件包均可以执行，如make tools/m4/install make tools/mtools/install

图 3-6 函数 subdir 的使用实例

3.2 tools/Makefile

tools/Makefile 在主 Makefile 中被引用，此文件主要功能是 tools 目录下的子包定义目标及其执行规则，定义动态目标$(tools/stamp-install)，以及编译动态目标所需要的准备工作。

```
# tools/Makefile  主体语句摘录
$(curdir)/builddirs := $(tools-y) $(tools-dep) $(tools-)
$(curdir)/builddirs-default := $(tools-y)                          P1

# preparatory work
$(STAGING_DIR)/.prepared: $(TMP_DIR)/.build
$(STAGING_DIR_HOST)/.prepared: $(TMP_DIR)/.build                   P2

define PrepareCommand
$(STAGING_DIR_HOST)/bin/$(1): $(STAGING_DIR)/.prepared
    ...
endef
$(STAGING_DIR_HOST)/bin/stat: $(STAGING_DIR)/.prepared
$(eval $(call PrepareCommand,find,gfind find))
$(eval $(call PrepareCommand,md5sum,gmd5sum md5sum $(SCRIPT_DIR)/md5sum))   P3
$(eval $(call PrepareCommand,cp,gcp cp))
$(eval $(call PrepareCommand,seq,gseq seq))
$(eval $(call PrepareCommand,python,python2 python))
$(curdir)/cmddeps = $(patsubst %,$(STAGING_DIR_HOST)/bin/%,find md5sum cp stat seq python)

$(curdir)//prepare = $(STAGING_DIR)/.prepared $(STAGING_DIR_HOST)/.prepared $($(curdir)/cmddeps)
$(curdir)//compile = $(STAGING_DIR)/.prepared $(STAGING_DIR_HOST)/.prepared $($(curdir)/cmddeps)

# prerequisites for the individual targets
$(curdir)/ := .config prereq
$(curdir)//install = $(1)/compile

$(eval $(call stampfile,$(curdir),tools,install,,CONFIG_CCACHE CONFIG_powerpc CO   GCC_VERSION_4_5
$(eval $(call subdir,$(curdir)))                                   P4
```

图 3-7 tools/Makefile 的主体内容

说明：

1、图 3-7 展示的为 tools/Makefile 的部分内容，详细内容见文件 tools/Makefile。

2、P1 部分的 tools/builddirs 记录扫描 tools 目录下的子目录，tools/builddirs-default 保存需要编译 tools/install 时需要编的子包(子目录)的名称。

3、P2 和 P3 部分，均为 tools//prepare 和 tools//compile 是基础准备工作。P2 主要作用是 mkdir 编译目录，touch 相应的.prepared 文件；P3（tools/cmddeps）完成的工作是在 staging_dir/host/bin 目录下放置 find、md5sun、cp、stat、seq、python 文件。

4、P4 分别调用函数 stampfile 和 subdir，stampfile 生成目标$(tools/stamp-install)及其编译规则，详细说明请参见图 3-4 函数 stampfile 的使用实例。subdir 为 tools 目录下的子目录定义编译目标及其子目录切换执行的规则，详细说明见图 3-6 函数 subdir 的使用实例。

stampfile: 定义动态目标 + 执行语句

$$(tools/stamp-install): $(TMP_DIR)/.build　　(目标+依赖)
　　make tools/install　　(主要执行语句)

subdir: 子目录编译目标及规则 + 依赖关系

子目录编译目标及规则：

　　tools/m4/clean (执行: make -C tools/m4 clean) tools/m4/download tools/m4/prepare …
　　tools/mtools/clean tools/mtools/download tools/mtools/prepare …
　　…

依赖关系：

　　tools/clean : tools/ tools/m4/clean tools/squashfs4/clean …
　　tools/prepare : tools/ tools/m4/prepare tools/squashfs4/prepare …
　　tools/install : tools/ tools/m4/install tools/squashfs4/Install …
　　…

图 3-8 tools/Makefile 中调用函数(subdir+stampfile)实现的功能

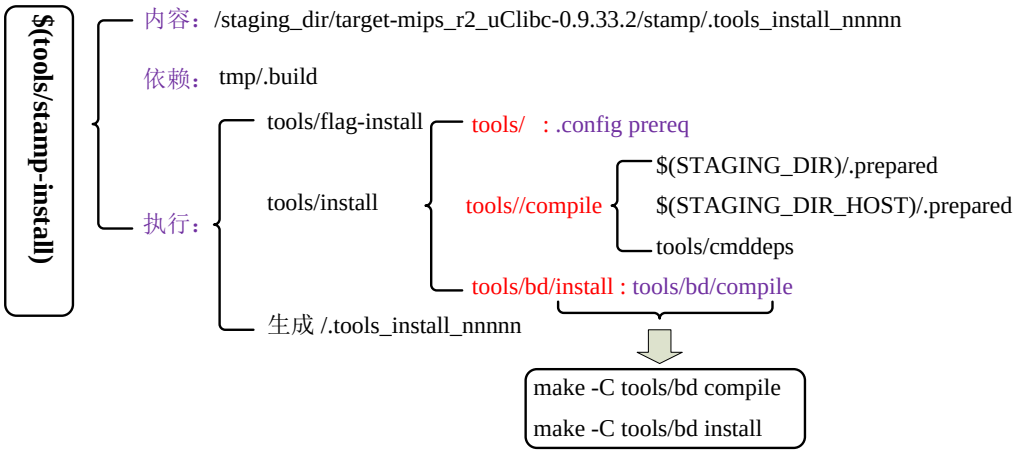


图 3-9 $(tools/stamp-install)的目标关系图

说明：

1、编译动态目标$(tools/stamp-install)的核心是：make tools/install，系统会根据 tools/install 和子目录软件包的依赖关系以及软件包之间的依赖关系，依次编译 tools 目录下的软件包。

2、图中 tools/bd/install 中的 bd 是 tools 目录下要编译的软件包的代称，bd 不是实际的软件包名字。make tools/install 时，bd 依次的值为 m4、sed、libtool …。

3、make tools/install 不会编译 tools 目录下的所有软件包，只编译 tools/builddirs -default 里面的软件包。相关说明见图图 3-6 函数 subdir 的使用实例。

4、tools 目录下单个软件包编译过程的分析，见第四章 4.2 节 tools 软件包的编译。

3.3 toolchain/Makefile

toolchain/Makefile 和 tools/Makefile 的结构大体一致，功能基本相同。Makefile

为 toolchain 目录下的所有子包定义编译目标(如:toolchain/gdb/install)及其执行规则，定义动态目标$(toolchain/stamp-install)，以及编译动态目标所需要的准备工作。toolchain/Makefile 主体功能的描述见图 3-10，$(toolchain/stamp-install)的目标关系见图 3-11。

1、定义toolchain/builddirs，toolchain/builddirs-compile， toolchain/builddirs-install
用于确定make toolchain/install时需要编译的子目录软件包

2、定义软件包编译目标的依赖关系，用于确定软件包编译次序

3、调用函数**stampfile和subdir**

$(eval $(call **stampfile,toolchain,toolchain,install**,$(TOOLCHAIN_DIR)/stamp/.gcc-initial_installed , , $(TOOLCHAIN_DIR))
stampfile: 定义动态目标$$(toolchain/stamp-install) **+ 执行语句(**make toolchain/install)

$(eval $(call **subdir**,**toolchain**))
subdir: 子目录编译目标及规则 + 依赖关系
子目录(软件包)编译目标及规则：
toolchain/gdb/clean (执行: make -C toolchain/m4 clean) toolchain/gdb/prepare …
toolchain/binutils/clean toolchain/binutils/prepare toolchain/binutils/compile …
…
依赖关系：
toolchain/clean : toolchain/ toolchain/gdb/clean toolchain/binutils/clean …
toolchain/prepare : toolchain/ toolchain/gdb/prepare toolchain/binutils/prepare …
toolchain/install : toolchain/ toolchain/gdb/install toolchain/binutils/install …
…

图 3-10 toolchain/Makefile 主体功能

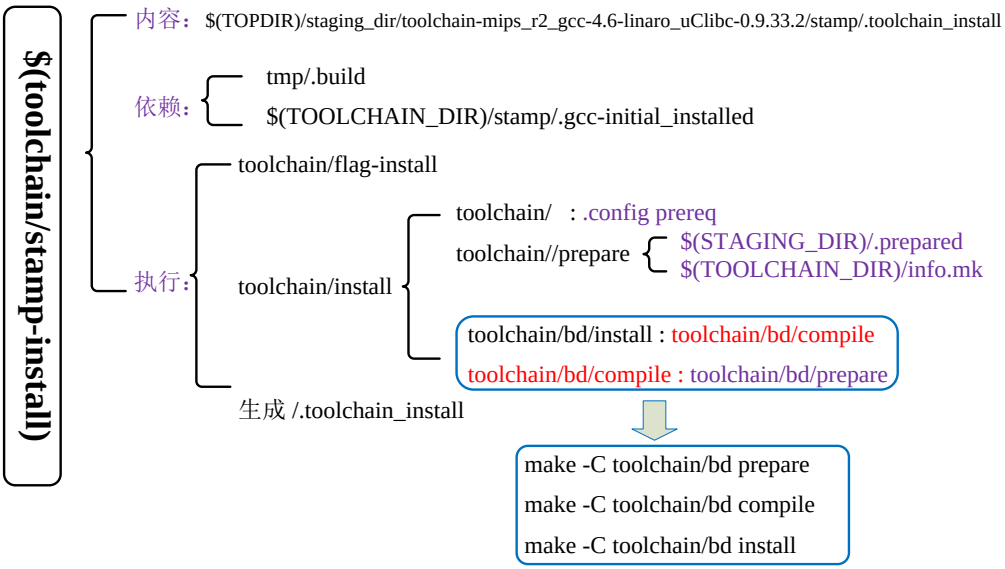


图 3-11 $(toolchain/stamp-install)的目标关系图

说明：

1、编译动态目标$(toolchain/stamp-install)的核心是：make toolchain/install，系统会根据 toolchain/install 和子目录软件包的依赖关系，以及软件包之间的依赖关系，依次编译 toolchain

目录下的软件包。

2、图中 toolchain/bd/install 中的 bd 是 toolchain 目录下要编译的软件包的代称，bd 不是实际的软件包名字。make toolchain/install 时，bd 依次的值为 gdb、binutils、gcc/minimal、kernel-headers、uClibc/headers、gcc/initial、uClibc、gcc/final、uClibc/utils。

3、make toolchain/install 不会编译 toolchain 目录下的所有软件包，只编译 toolchain /builddirs -install 里面的软件包。相关说明见图图 3-6 函数 subdir 的使用实例。

4、toolchain 目录下单个软件包编译过程的分析，见第四章 4.3 节 toolchain 软件包的编译。

3.4 package/Makefile

1、定义：package/builddirs，package/builddirs-default，package/builddirs-install，package/builddirs-prereq
用于确定动态目标$(package/stamp-<TARGET>)需要编译的子目录软件包

2、定义函数和编译目标
函数： push_git_build_stats mklibs
目标：package/install: package/cleanup: package/rootfs-prepare: package/index：

3、调用函数**stampfile和subdir**

$(eval $(call stampfile,package,package,prereq,.config))
$(eval $(call stampfile,package,package,cleanup,$(TMP_DIR)/.build))
$(eval $(call stampfile,package,package,compile,$(TMP_DIR)/.build))
$(eval $(call stampfile,package,package,install,$(TMP_DIR)/.build))
$(eval $(call stampfile,package,package,rootfs-prepare,$(TMP_DIR)/.build))
生成目标及其编译规则: (package/stamp-prereq), (package/stamp-cleanup), (package/stamp-compile), (package/stamp-install), (package/stamp-rootfs-prepare)

$(eval $(call subdir,$(curdir)))
subdir: 子目录编译目标及规则 + 依赖关系
子目录(软件包)编译目标及规则：
package/lua/clean (执行: make -C package/lua clean) package/lua/prepare …
package/busybox/clean package/busybox/prepare package/busybox/compile …
…
依赖关系：
package/clean : package/ package/lua/clean package/busybox/clean …
package/prepare : package/ package/lua/prepare package/busybox/prepare …
package/install : package/ package/lua/install package/busybox/install …
…

图 3-12 package/Makefile 的功能描述

说明：

1、$(eval $(call subdir,package))会为 package 目录下(包括链接的其他文件目录)所有软件包定义编译规则，执行语句均为：make -C package/<包名> TARGET。这条语句会打开 package/<包名>/Makefile 文件，编译目标为 TARGET。当 TARGET 在 package/<包名>/Makefile 中未定义时，会报错退出。

2、package/Makefile定义package/builddirs，package/builddirs-default，package/builddirs-install，package/builddirs-prereq等，用于确立动态目标

$(package/stamp-<TARGET>)与软件包的依赖关系。

3、编译动态目标(如: $(package/stamp-compile))时，只会编译存在依赖关系的软件包。

4、开发者在 make menuconfig 配置界面进行的操作，会体现在上述几个 builddirs 中。

5、package/Makefile 定义的动态(stamp)目标有五个，五个动态目标的关系整理如下表所示。

表 3-1 package 动态目标

<table>
<tr><td colspan="4">package/stamp-prereq</td></tr>
<tr><td>内容</td><td colspan="3">$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.package_prereq</td></tr>
<tr><td>依赖</td><td colspan="3">.config</td></tr>
<tr><td rowspan="2">执行</td><td colspan="3">make package/prereq : package/bd/prereq （bd 属于 package/builddirs-prereq）</td></tr>
<tr><td colspan="3">touch $(package/stamp-prereq)</td></tr>
<tr><td colspan="4">package/stamp_cleanup</td></tr>
<tr><td>内容</td><td colspan="3">$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.package_cleanup</td></tr>
<tr><td>依赖</td><td colspan="3">tmp/.build</td></tr>
<tr><td rowspan="2">执行</td><td>make package/cleanup</td><td colspan="2">rm -rf $(TARGET_DIR) $(STAGING_DIR_ROOT)</td></tr>
<tr><td colspan="3">touch $(package/stamp- cleanup)</td></tr>
<tr><td colspan="4">package/stamp-compile</td></tr>
<tr><td>内容</td><td colspan="3">$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.package_ compile</td></tr>
<tr><td>依赖</td><td colspan="3">tmp/.build</td></tr>
<tr><td rowspan="2">执行</td><td colspan="3">make package/ compile : package/bd/ compile（bd 属于 package/builddirs-default）</td></tr>
<tr><td colspan="3">touch $(package/stamp- compile)</td></tr>
<tr><td colspan="4">package/stamp-install</td></tr>
<tr><td>内容</td><td colspan="3">$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.package_install</td></tr>
<tr><td>依赖</td><td colspan="3">tmp/.build</td></tr>
<tr><td rowspan="4">执行</td><td colspan="3">make package /flags-install</td></tr>
<tr><td rowspan="2">make package/install</td><td colspan="2">package/cleanup</td></tr>
<tr><td colspan="2">package/bd/install （bd 属于 package/builddirs-install）</td></tr>
<tr><td colspan="3">touch $(package/stamp-install)</td></tr>
<tr><td colspan="4">package/stamp-rootfs-prepare</td></tr>
<tr><td>内容</td><td colspan="3">$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.package_rootfs-prepare</td></tr>
<tr><td>依赖</td><td colspan="3">tmp/.build</td></tr>
<tr><td rowspan="3">执行</td><td rowspan="2">make package/ rootfs-prepare</td><td colspan="2">make package/preconfig</td></tr>
<tr><td colspan="2">call mklibs</td></tr>
<tr><td colspan="3">touch $(package/stamp-prereq)</td></tr>
</table>

说明：

1、动态目标package/stamp-<TARGET>，一般会启动package包的编译:package/bd/<TARGET>，bd不是所有的子目录包，而是由package/Makefile中的builddirs决定。编译子目录软件包的依据如下：package/builddirs-$(target) > package/builddirs-default > package/builddirs。只有当前者未定义时，才以后者为依据，以此类推。

2、由于 include/subdir.mk 中的子目录目标(SUBTARGETS)没有定义 rootfs-prepare 和 cleanup， 所以 package/stamp-rootfs-prepare 不会编译 package/bd/rootfs-prepare，package/stamp-cleanup 不会编译 package/bd/cleanup。

3、package 目录下单个软件包编译过程的分析，见第四章 4.4 节 package 软件包的编译。

3.5 target/Makefile

```
# target/Makefile

curdir:=target

$(curdir)/builddirs:=linux sdk imagebuilder toolchain
$(curdir)/builddirs-default:=linux
$(curdir)/builddirs-install:=linux $(if $(CONFIG_SDK),sdk) \
    $(if $(CONFIG_IB),imagebuilder) $(if $(CONFIG_MAKE_TOOLCHAIN),toolchain)

$(curdir)/imagebuilder/install:=$(curdir)/linux/install

$(eval $(call stampfile,$(curdir),target,prereq,.config))
$(eval $(call stampfile,$(curdir),target,compile,$(TMP_DIR)/.build))
$(eval $(call stampfile,$(curdir),target,install,$(TMP_DIR)/.build))

$($(curdir)/stamp-install): $($(curdir)/stamp-compile)

$(eval $(call subdir,$(curdir)))
```

确定子目录编译范围

生成动态目标

定义子目录(linux sdk imagebuilder toolchain)的编译规则、配置动态目标与子目录编译的依赖关系

图 3-12 target/Makefile 的内容

target/Makefile 主体功能分为三个部分，分别为生成动态目标、定义子目录目标及其编译规、确立动态目标与子目录目标的依赖关系。动态目标与子目录目标的依赖关系主要由 builddirs 来确立。

target 目录下有 linux sdk imagebuilder toolchain 四个软件包，target/builddirs 的值为四个软件包的名字。$(eval $(call subdir,target))会为 target 目录下四个软件包定义目标及编译规则。

```
$(eval $(call subdir,target))
target/builddirs = linux sdk imagebuilder toolchain
目标：
        target/linux/clean          target/linux/compile          target/linux/install  …
        target/sdk/clean            target/sdk/compile            target/sdk/install  …
        target/imagebuilder/clean   target/imagebuilder/compile   target/imagebuilder/install  …
        target/toolchain/clean      target/toolchain/compile      target/toolchain/install  …
规则：
        make target/linux/clean   执行: make -C target/linux  clean
```

图 3-13 target 软件包的目标及编译规则

target/builddirs-default 表征默认编译的软件包，值为 linux。subdir 函数根据 target/builddirs-default 确立动态目标$(target/stamp-prereq)、$(target/stamp-compile)与子目录目标的依赖关系，target/prereq 只依赖于 target/linux/prereq，target/compile 只依赖于 target/linux/compile。

表 3-1 target 动态目标

target/stamp-prereq	
内容	$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.target_prereq
依赖	.config
执行	make target/prereq : target/linux/prereq
	touch $(target/stamp-prereq)
target/stamp-compile	
内容	$(TOPDIR)/staging_dir/target-mips_r2_uClibc-0.9.33.2/stamp/.target_compile
依赖	tmp/.build
执行	make target/compile : target/linux/compile
	touch $(target/stamp-compile)

target/builddirs-install 的值为需要执行安装的软件包的名字，默认有 linux。其他三种软件包是否进行安装，由系统配置决定。输入命令:make menuconfig 之后，会出现图 3-14 所示的页面。在红色方框内选择需要安装的软件包，保存退出，选择结果体现在 target/builddirs-install。

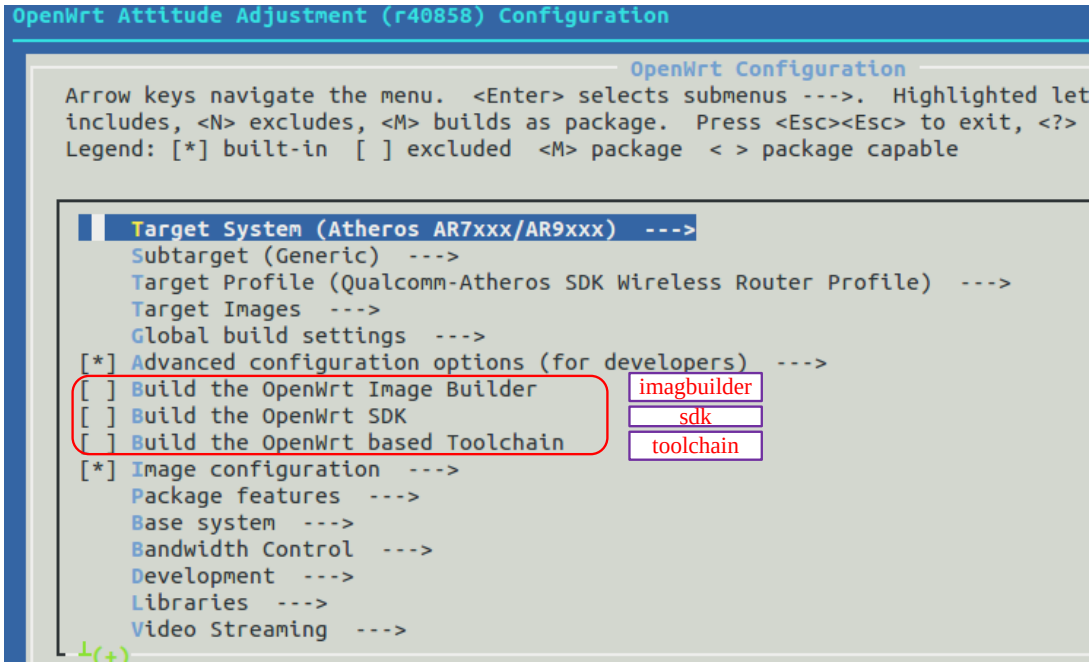


图 3-13 make menuconfig 的配置界面

subdir 函数根据 target/builddirs-install 确立动态目标$(target/stamp-install)与四

个子目录软件包目标的依赖关系。

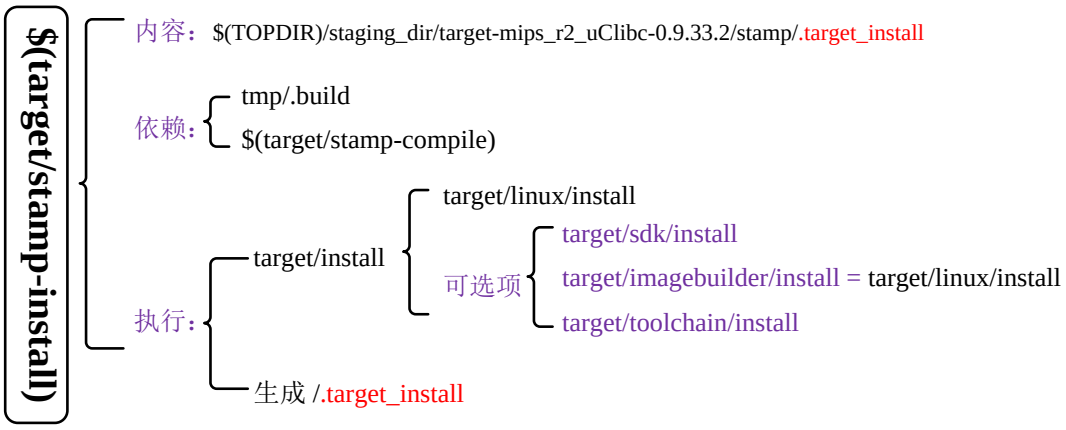


图 3-11 $(target/stamp-install)的目标关系图

说明：

1、target/Makefile 中有语句：target/imagebuilder/install:=target/linux/install，当编译 target/imagebuilder/install 时，会先编译 target/linux/install。

2、target/Makefile 中有语句：target/stamp-install: target/stamp-compile，make taget/install 之前必须先检查 target/compile 是否已经编译。检查的方法是查看是否存在.target_compile 文件。

3、四种软件包的功能、特点、编译执行过程的详尽分析见第四章 4.5 节。

第四章 软件包的编译规则分析

本章主要分析 tools、toolchain、package、target 目录下软件包的编译过程。分析内容主要包括：进入软件包的输入命令方式，软件包的 Makefile 结构，编译软件包相关的文件与函数关系，软件包的编译与安装流程，编译与安装路径等。

4.1 软件包的编译规则概述

openwrt 系统主要有四类软件包，分别放置在 tools、toolchain、package、target 四个文档目录下。不同类型的软件包的编译路径、目标文件、文件安装路径均不完全相同，软件包的编译路径、目标文件、文件安装路径等主要取决于软件包的 Makefile 文件（<类名>/<包名>/Makefile），本章 4.2 到 4.5 节将分类别进行阐述。但单个软件包的目标定义和执行编译的规则相同，四种类型软件包的目标定义和编译规则在主 Makefile 再次执行部分完成。

主 Makefile 再次执行部分分别引入四个文件 tools/Makefile、target/Makefile、toolchain/Makefile、package/Makefile。每个文件的最后一行调用函数 subdir，这个函数在上一章 3.1 节已经进行详尽描述。<类名>/Makefile 调用函数 subdir，为<类名>目录下的所有软件包定义了编译目标：<类名>/<包名>/TARGET，编译规则均为：make -C <类名>/<包名> TARGET，即进入软件包所在的目录，执行 Makefile 文件，目标为 TARGET。TARGET 包含常见的 clean、compile、install、prepare 等。进入某软件包进行编译的主要三种方式如下图所示：

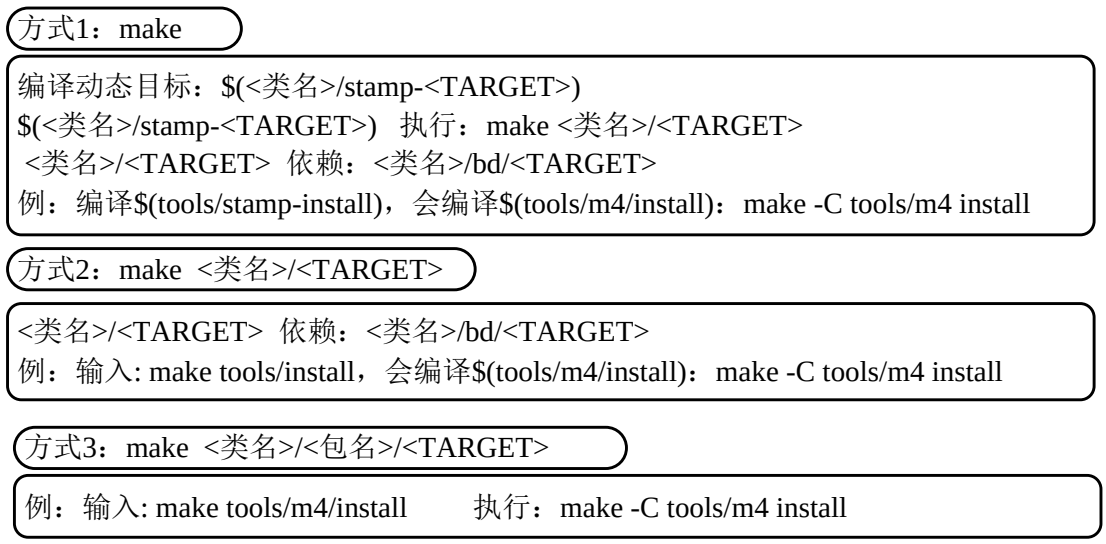


图 4-1 进入单个软件包执行编译的三种方式

说明：

进入单个软件包编译只能从 openwrt 系统的顶层目录$(TOP_DIR)下输入相应的 make 命

令，不能进入单个软件包目录然后输入编译命令。这是因为编译软件包需要的引入文件的路径为相对路径，均是相对于 openwrt 系统的顶层目录$(TOP_DIR)。如果进入单个软件包目录输入编译命令，会导致引入文件找不到，无法启动编译。

进入软件包目录之后，执行 Makefile 文件，编译目标为 TARGET，Makefile 文件里面会定义编译此软件包的目标，如 compile、install，但并不是所有的目标均有定义，有些目标如 configure、distcheck，可能没有定义。如果 TARGET 在 Makefile 中未定义，则报错提示并退出此次编译（<类名>/<包名>/TARGET）。当进入软件包的 TARGET 为 compile/install 时，编译三类软件包(target 除外)的一般流程如下：

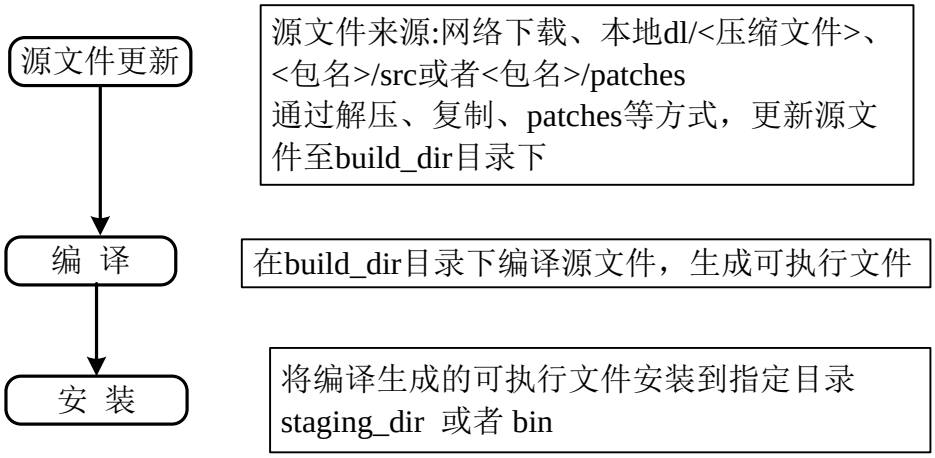


图 4-2 软件包编译的一般流程

4.2 tools 包的编译分析

4.2.1 Makefile 的基本规则

```
# tools/<PKG_NAME>/Makefile
include $(TOPDIR)/rules.mk
# 定义软件包信息:PKG_XX
PKG_NAME:=
PKG_VERSION:=
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_SOURCE_URL:=@GNU/$(PKG_NAME)
PKG_MD5SUM:=
include $(INCLUDE_DIR)/host-build.mk
# 配置编译工具相关信息
HOST_LDFLAGS += $(HOST_STATIC_LINKING)
HOST_CONFIGURE_VARS += CC="$(HOSTCC_NOCACHE)"
# 定义编译包信息
define Host/Clean
define Host/Prepare
define Host/Compile
define Host/Install
$(eval $(call HostBuild))
```

Makefile 基本规则

引入文件

定义软件包信息

配置编译工具(可选)

定义编译包信息

启动编译

图 4-3 tools 软件包的 Makefile 文件

tools 软件包的 Makefile 文件的大致内容和基本规则如上图所示。Makefile 中引入包编译必须的文件 rules.mk 和 host-build.mk，rules.mk 里面定义了很多全局变量，host-build.mk 定义了函数 HostBuild。定义好软件包基本信息和编译包信息之后，即可通过调用 HostBuild 函数来启动编译。HostBuild 函数的文件关系如下图所示。

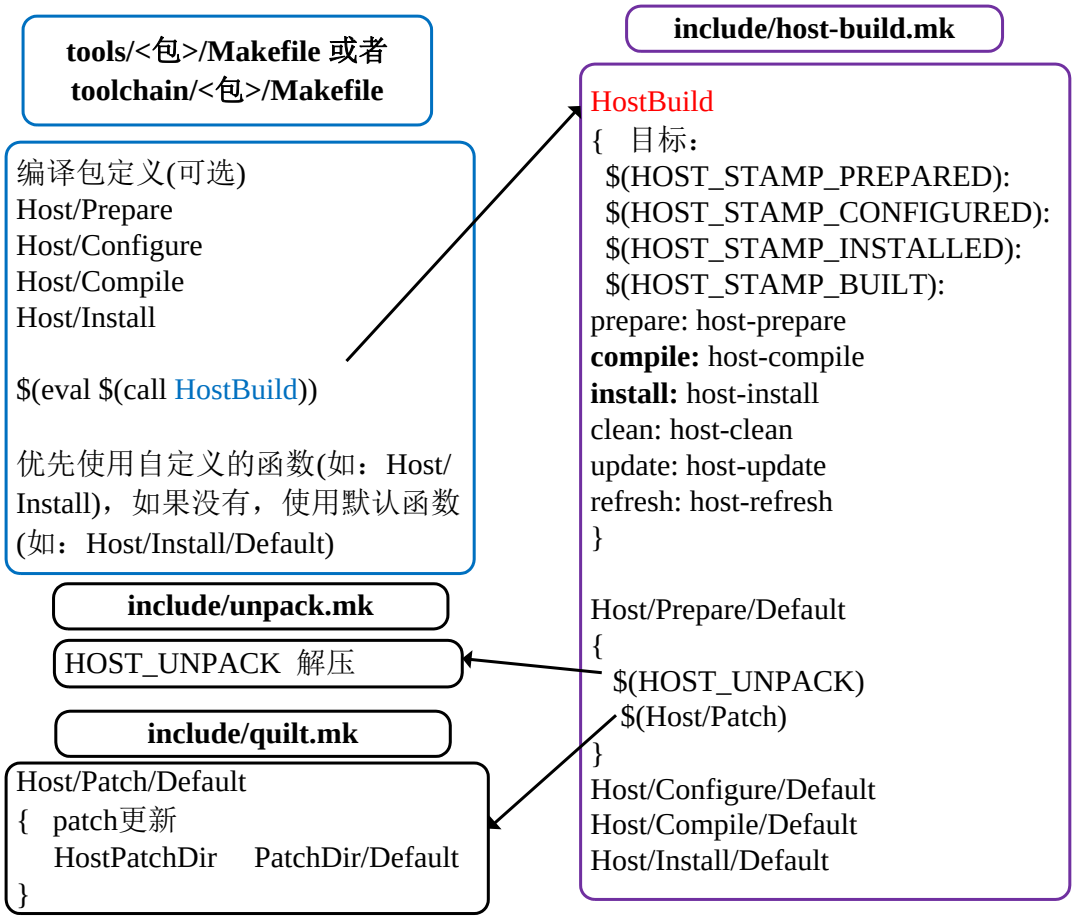


图 4-4 函数 HostBuild 的文件关系

4.2.2 编译执行流程

HostBuild 定义了大部分编译相关的目标，如 prepare、compile、install、clean 等，并且为目标规范了编译流程。函数 HostBuild 主要目标 compile 和 install 的目标依赖关系如下图所示。

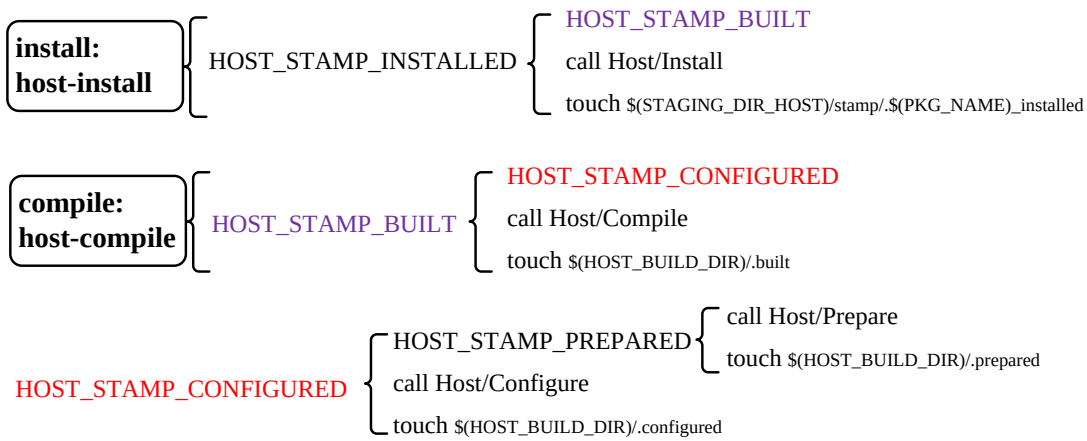


图 4-5 函数 HostBuild 核心编译目标的依赖关系

说明：

1、图 4-4 中，紫色和红色分别标识同一目标。

2、编译目标为 compile 时，根据依赖关系，从 Host/Prepare→ Host/Configure →Host/Compile。

3、编译目标为 install 时，执行次序为：Host/Prepare→ Host/Configure →Host/Compile→ Host/Install。

4、如果在软件包的 Makefile 没有定义编译包信息(宏/函数，如 Host/Prepare)，host-build.mk 会进行默认定义。

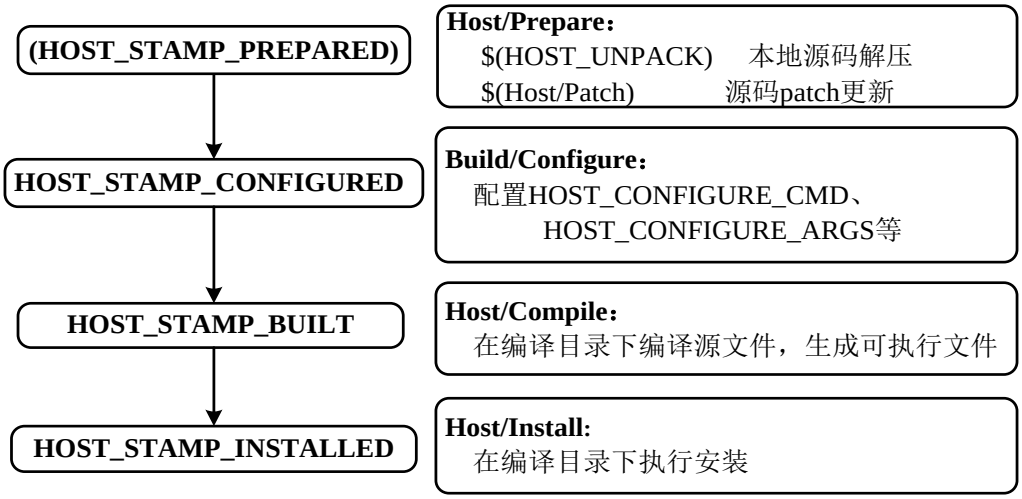


图 4-5 tools 软件包的编译执行流程

4.2.3 编译与安装路径

编译路径：**HOST_BUILD_DIR**
= $(TOPDIR)/build_dir/host/$(PKG_NAME)$(if $(PKG_VERSION),-$(PKG_VERSION))

安装路径：**STAGING_DIR_HOST** = $(TOPDIR)/staging_dir/host

示例：tools/m4

编译路径： $(TOPDIR)/build_dir/host/M4-1.4.16

安装路径：$(TOPDIR)/staging_dir/host

图 4-5 tools 软件包的编译与安装路径

4.3 toolchain 包的编译分析

4.3.1 Makefile 的基本规则

```
# toolchain/<PKG_NAME>/Makefile

include $(TOPDIR)/rules.mk

# 定义软件包信息:PKG_XX
PKG_NAME:=
PKG_VERSION:=
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_SOURCE_URL:=
PKG_MD5SUM:=

include $(INCLUDE_DIR)/toolchain-build.mk

# 定义编译包信息
define Host/Clean
define Host/Prepare
define Host/Compile
define Host/Install

$(eval $(call HostBuild))
```

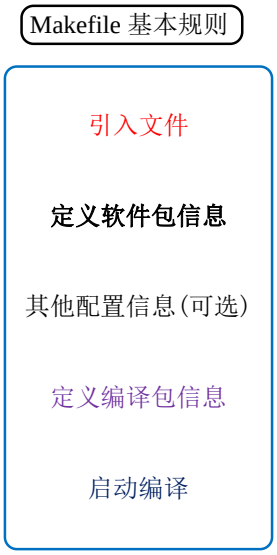


图 4-6 toolchain 软件包的 Makefile 文件

toolchain 和 tools 软件包的 Makefile 结构基本一致，差别在于 tools/<包>/Makefile 引入文件 host-build.mk，tools/<包>/Makefile 引入文件 toolchain-build.mk。toolchain-build.mk 引入文件 host-build.mk，两种软件包可以调用同一函数 HostBuild 执行编译。

toolchain 软件包 Makefile 中引入 toolchain-build.mk 而不是 host-build.mk，目的在于改变编译与安装路径。tools 软件包的编译与安装路径采用 rules.mk 中定义的默认路径，而 toolchain-build.mk 重新定义 STAGING_DIR_HOST 和 BUILD_DIR_HOST，使得 toolchain 软件包的编译和安装路径与 tools 软件包不同。

4.3.2 编译执行流程

toolchain 和 tools 软件包的 Makefile 结构基本一致，软件包基本信息和编译包信息格式一样，调用同一函数 HostBuild 执行编译。因此，toolchain 软件包的编译执行流程与 tools 软件包完全一致，toolchain 软件包的编译执行流程可参考本章 4.2.2 节的内容。

4.3.3 编译与安装路径

编译路径: **HOST_BUILD_DIR**

= $(TOPDIR)/build_dir/toolchain-$(ARCH)$(ARCH_SUFFIX)_gcc-$(GCCV)$(DIR_SUFFIX)/(PKG_NAME)(if $(PKG_VERSION),-$(PKG_VERSION))

安装路径：**STAGING_DIR_HOST**

= $(TOPDIR)/staging_dir/toolchain-$(ARCH)$(ARCH_SUFFIX)_gcc-$(GCCV)$(DIR_SUFFIX)

示例：toolchain/binutils

编译路径： $(TOPDIR)/build_dir/toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2/binutils

安装路径：$(TOPDIR)/staging_dir/toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2

图 4-7 toolchain 软件包的编译与安装路径

4.4 package 包的编译分析

package 软件包有用户态和内核态两种，以用户态的应用程序为主要类型。由于两者的结构与编译执行过程非常相似，本章首先分析用户态软件包的编译执行过程。本节最后部分会阐述内核态软件包的编译流程。

4.4.1 Makefile 的基本规则

```
# package/<PKG_NAME>/Makefile
include $(TOPDIR)/rules.mk
# 定义软件包信息:PKG_XX
PKG_NAME:=
PKG_VERSION:=
PKG_RELEASE:=
PKG_SOURCE:=
PKG_SOURCE_URL:=
HOST_PATCH_DIR :=
include $(INCLUDE_DIR)/package.mk
# 定义编译包信息
define Package/<pkgName>
define Package/<pkgName>/description
define Build/Prepare
define Build/Configure
define Build/Compile
define Package/<pkgName>/install
# 启动编译
$(eval $(call BuildPackage,<pkgName>))
```

Makefile 基本规则

引入文件

定义软件包信息

其他配置信息(可选)

定义编译包信息

启动编译

图 4-8 package 软件包的 Makefile

说明：

1、package/<包名>/Makefile 定义编译该软件包的基本信息，软件包信息定义中，

PKG_NAME 与软件包目录名称相同。当只有一个编译包时，编译包名字<pkgName>与包名相同。

2、一个软件包可以编译产生多个编译包，只需要分别定义编译包信息，然后分别调用编译启动函数 BuildPackage：

define package/ <pkgName1>、 package/ <pkgName1>/install、 …
$(eval $(call BuildPackage, <pkgName1>))，
define package/ <pkgName2>、 package/ <pkgName2>/install、…
$(eval $(call BuildPackage, <pkgName2>))

3、package 软件包的 Makefile 文件的大致大致内容和基本规则与 tools 软件包相似。区别在于：tools 软件包 Makefile 引入 host-build.mk，启动编译的函数为 HostBuild。package 软件包 Makefile 中引入 package.mk，启动编译的函数为 BuildPackage。

4、package 包编译时相关文件及函数的关系如下图所示。

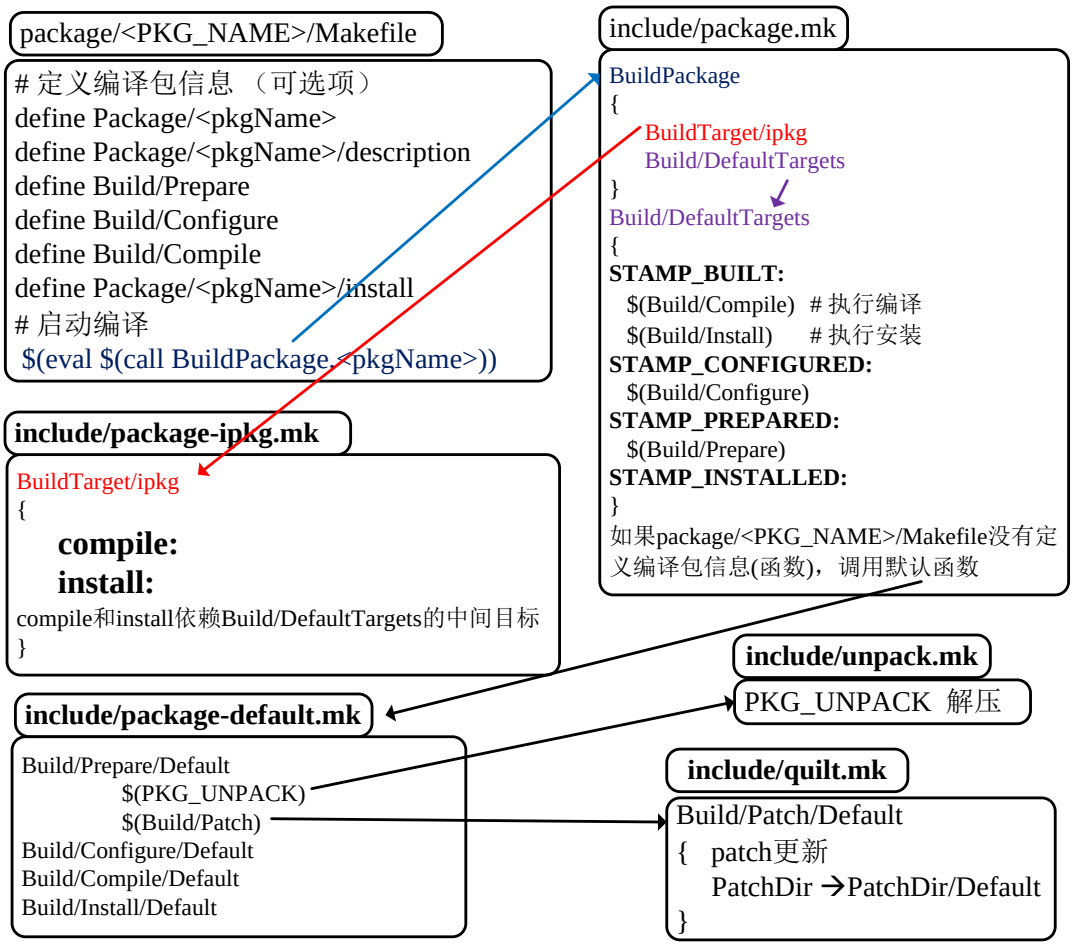


图 4-9 package 软件包编译时相关文件及其函数的关系

4.4.2 配置选项

编译软件包之前，需要配置该包的编译选项。方法如下：输入命令：make menuconfig，进入配置界面；找到需要编译的包<pkgName>，键入 N、Y、M。

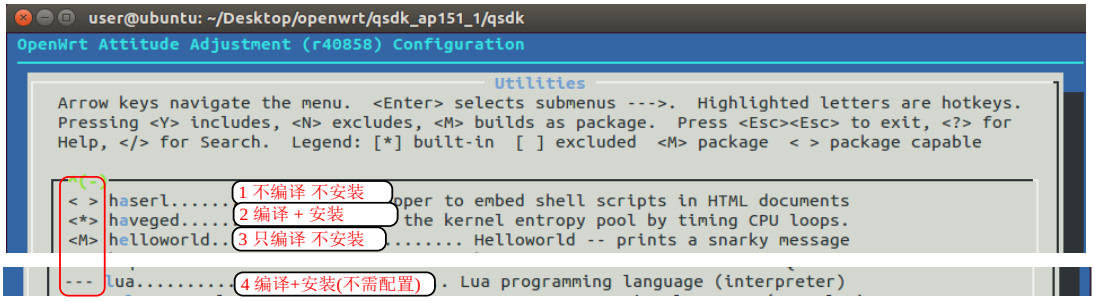


图 4-10 package 软件包启动编译的配置选项

说明：

1、上图所示的红色框内<>内的符号，代表软件包的配置选项。键值(Y/M/N)、<>内的内容、配置值(CONFIG_PACKAGE_<pkgName>)、compile 和 install 的有效执行，他们的对应关系如下表所示。

表 4-1 配置选项的对应关系

键 值	<>内容	CONFIG_PACKAGE_<pkgName>	compile	install
N/n	空白	空	no	no
M/m	M	m	yes	no
Y/y	*	y	yes	yes
	---	y	yes	yes

2、图中红色字体“编译”与“安装”分别对应 compile 和 install 命令的有效执行，前者编译源文件产生 ipk 文件，后者主要工作：Installing <pkgName> to root...并 Setting flags for package <pkgName> to ok。

3、图中第 1 个软件包，compile 或者 install，均会提示“WARNING: skipping <pkgName> -- package not selected”。第 2 个和第 4 个软件包，compile 命令和 install 均可以正常执行；第 3 个软件包，compile 命令有效执行，install 会提示“Nothing to be done for `install'.”。

4.4.3 编译与安装路径

编译路径：PKG_BUILD_DIR
默　认：build_dir/target-mips_r2_uClibc-0.9.33.2/<PKG_NAME>-<VERSION>
自定义：build_dir/linux-ar71xx_generic/<PKG_NAME>
特　例：package/kernel　PKG_BUILD_DIR = build_dir/linux-ar71xx_generic/packages

安装路径：　　均为可选项，根据软件包的需要进行安装
PKG_INSTALL_DIR　build_dir/target-mips_r2_uClibc-0.9.33.2/<PKG_NAME>-<version>/**ipkg-install**
IDIR_<pkgName>　build_dir/target-mips_r2_uClibc-0.9.33.2/<PKG_NAME>-<version>/**ipkg-ar71xx**/<pkgName>
STAGING_DIR_ROOT　staging_dir/target-mips_r2_uClibc-0.9.33.2/root-ar71xx
STAGING_DIR　　staging_dir/target-mips_r2_uClibc-0.9.33.2/

核心产物：bin/ar71xx/packages/<pkgName>-<version>_ar71xx.ipk

例：package/lua　pkgName = lua

PKG_BUILD_DIR　　build_dir/target-xxx/lua-5.1.4
PKG_INSTALL_DIR　build_dir/target-mips_r2_uClibc-0.9.33.2/lua-5.1.4/ipkg-install
IDIR_lua　　build_dir/target-mips_r2_uClibc-0.9.33.2/lua-5.1.4/ipkg-ar71xx/lua
STAGING_DIR_ROOT　staging_dir/target-mips_r2_uClibc-0.9.33.2/root-ar71xx
STAGING_DIR　　staging_dir/target-mips_r2_uClibc-0.9.33.2
bin/ar71xx/packages/lua_5.1.4-8_ar71xx.ipk

图 4-11 package 软件包的常规编译和安装路径

说明：

1、图中 PKG_BUILD_DIR 为 package 包的默认编译路径，如果在 package/<包>/Makefile 有定义，则以自定义为准，否则使用默认编译路径。

2、普通的 package 应用程序包，以默认编译路径为主。只有少数 package 包的编译路径在(KERNEL_BUILD_DIR)下，如 mtd：

PKG_BUILD_DIR = $(KERNEL_BUILD_DIR)/$(PKG_NAME)
= build_dir/linux-ar71xx_generic/mtd

4.4.4 编译执行流程

1 compile

图 2-12 package 软件包 compile 的执行流程

说明：

1、只有当 package/<包>/Makefile 中定义 Build/InstallDev，才编译 STAMP_INSTALLED。其他编译目标是必须项。

2、编译目标 STAMP_BUILD 时，如果 package/<包>/Makefile 没有定义 Build/Install，且 $(PKG_INSTALL)为空，调用此函数不进行任何操作。只有$(PKG_INSTALL)非空时，才调用函数 Build/Install/Default，执行的操作是：make -C (PKG_BUILD_DIR) install 安装的路径为(PKG_INSTALL_DIR)= (PKG_BUILD_DIR)/ipkg-install

3、图中所示 Build/Prepare 为$(PKG_UNPACK)非空时的默认操作函数，从 dl 目录下解压文件，并 patch 更新 package/包/patches/*.patch。如果源文件在 package/包/src 目录下，必须自定义 Build/Prepare，将 package/包/src/* --copy→ (PKG_BUILD_DIR)/*。

4、源文件不同的存储位置，对应不同的更新方式，更新后的源文件位置及其编译过程不完全一致。常见的有如下两种：

(1)源文件 → (PKG_BUILD_DIR)/* 进入编译目录即可进行编译

(2)源文件 → (PKG_BUILD_DIR)/src/* (PKG_BUILD_DIR)/Makefile 控制进入 src 目录

package/<包>/Makefile 中定义的编译包信息(build 函数：Build/Prepare、Build/Compile…)，以及(PKG_BUILD_DIR)/Makefile，配合起来实现对本包编译流程的完整控制。

5、每个中间目标执行的操作的详尽描述，如下图所示：

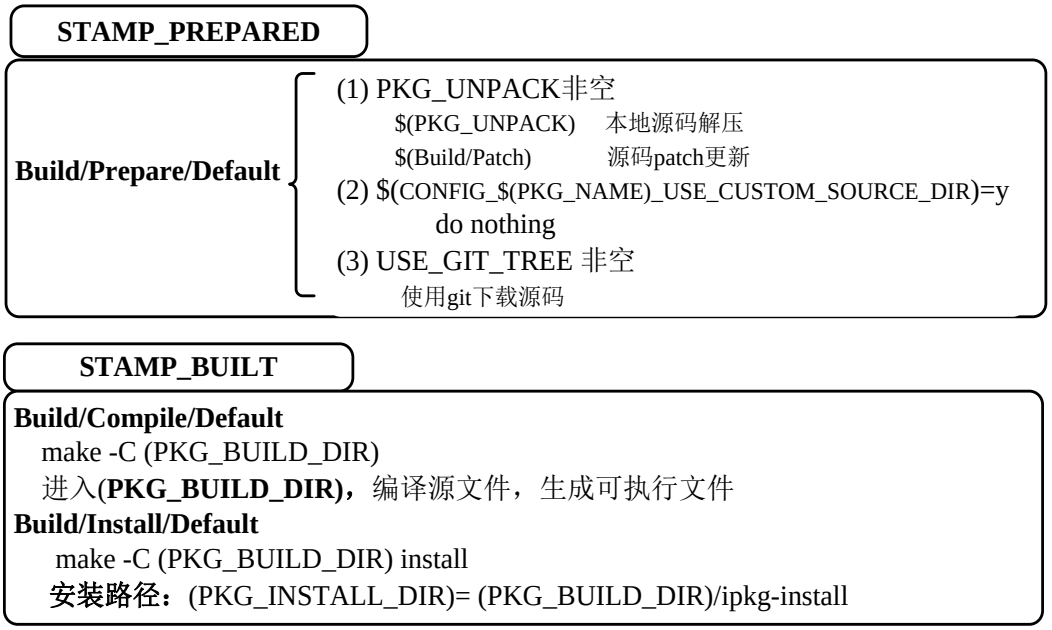


图 4-13 STAMP_PREPARED 和 STAMP_BUILD 的默认函数操作

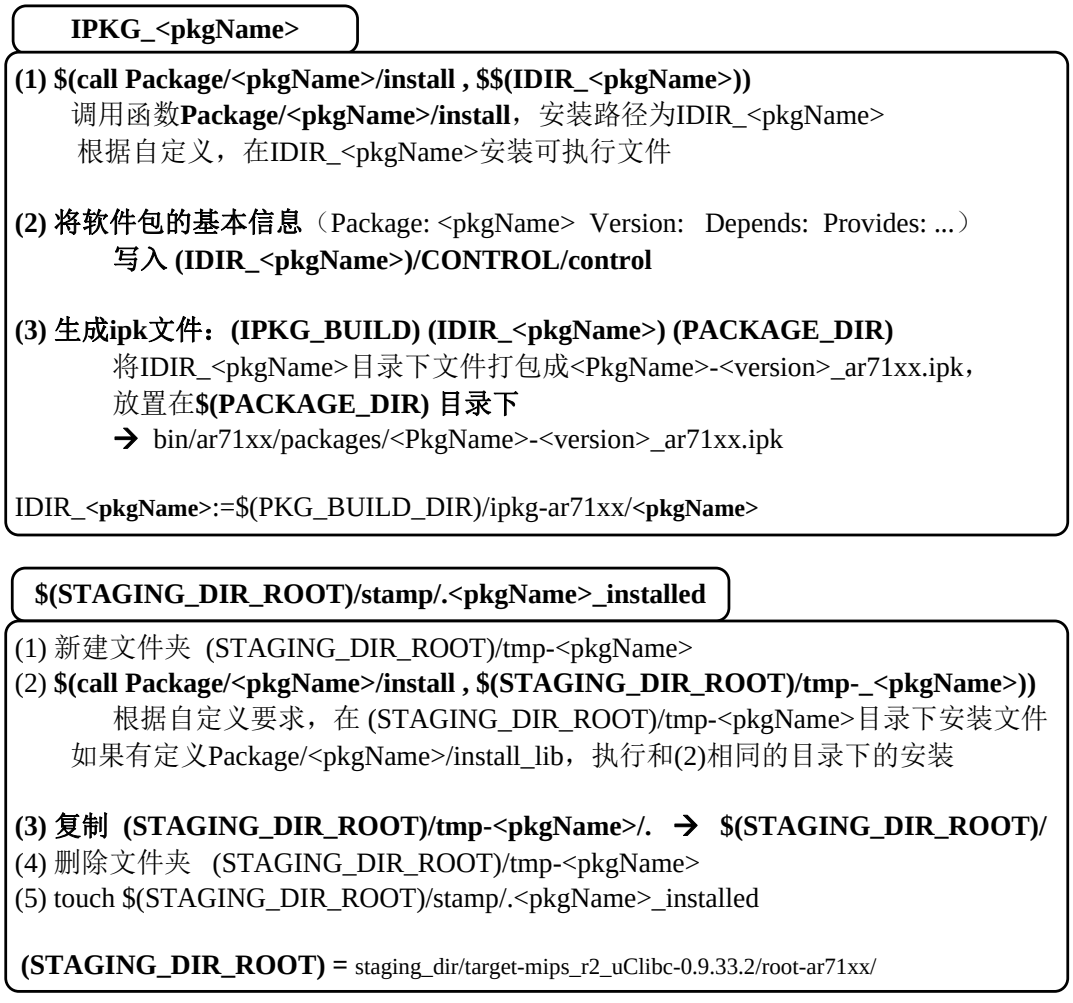


图 4-14 编译包<pkgName>的中间执行目标编译的详尽描述

2 install

如果软件包的配置选项为 Y，即 CONFIG_PACKAGE_<pkgName>=y，可以有效执行 install，install 的流程如下图所示。

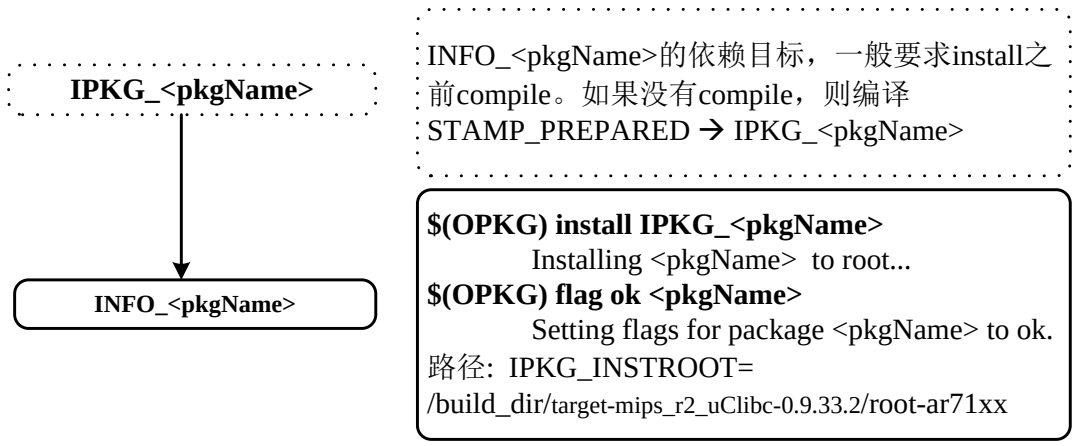


图 4-15 package 软件包 install 的执行流程

install 主要工作是在目录 IPKG_INSTROOT (build_dir/*/root-ar71xx)，安装 compile 生成的 ipk 文件。安装过程主是有如下两步：

$(OPKG) install IPKG_<pkgName>

Installing <pkgName> (20150814-1) to root...

$(OPKG) flag ok <pkgName>

Setting flags for package <pkgName> to ok。

安装完成之后，该软件包的安装效果如下图所示。

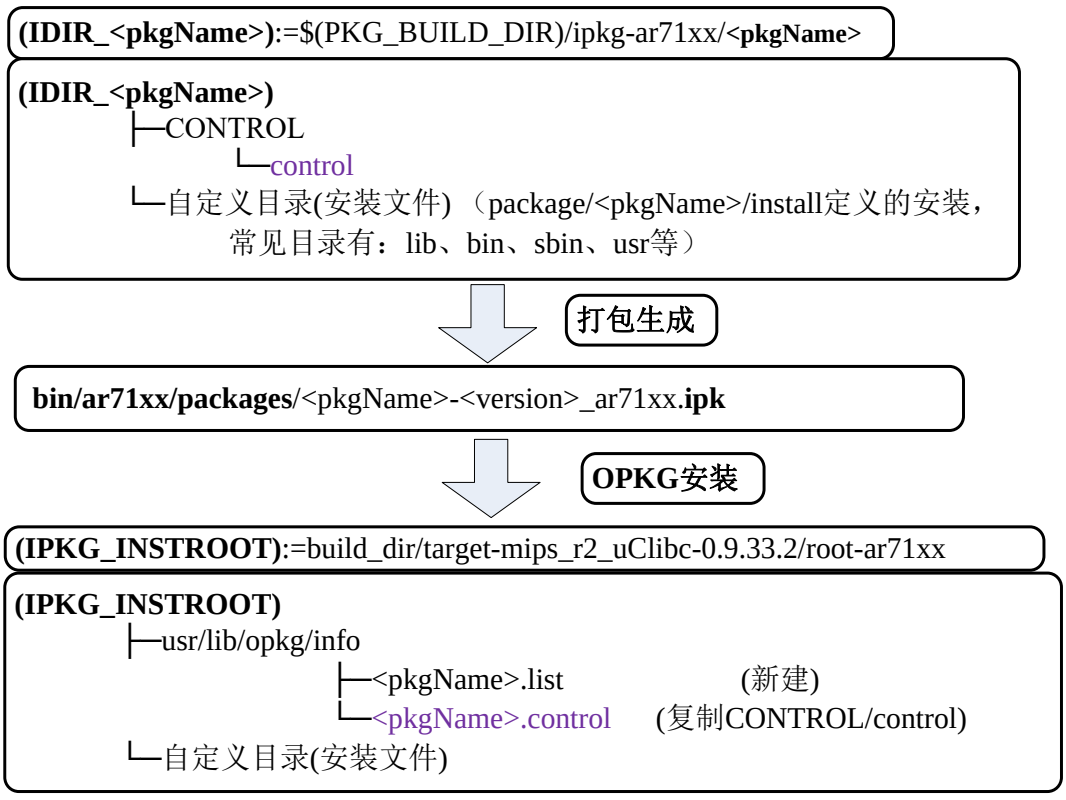


图 4-16 package 软件包生成 ipk 文件和安装的效果

3 编译流程补充说明

如果一个软件包，package/<PKG_NAME>/Makefile 内定义多个编译包信息，分别调用编译启动函数 BuildPackage，可以编译产生多个编译包。

package/<pkgName1>、package/<pkgName1>/install、 …

$(eval $(call BuildPackage, <pkgName1>))，

package/<pkgName2>、package/<pkgName2>/install、…

$(eval $(call BuildPackage, <pkgName2>))

compile 和 install 的过程中，中间目标 STAMP_PREPARED、STAMP_CONFIGURED、STAMP_BUILT、STAMP_INSTALLED 只编译一次，相对应的函数：Build/Prepare、Build/Configure、Build/Compile、Build/Install、Build/installDev 只调用一次。而与 pkgName 相关的中间目标 IPKG_<pkgName>、$(STAGING_DIR_ROOT)/stamp/.<pkgName>_installed、INFO_<pkgName>会多次编译。

综合来看，软件包(PKG_NAME)的源文件只编译一次，可以生成多个可执行文件；根据每个编译包(pkgName)的要求，生成不同的 ipk 文件，独立安装。

4.4.5 内核模块的编译分析

package 软件包有内核态软件包和用户态软件包两种，开发者开发的内核部分可以直接加入 Linux 的 Kernel 程序，也可以生成内核模块以便需要时装入内核。OpenWRT 系统一般希望开发者生成内核模块，在 Linux 启动后自动装载或手工使用 insmod 命令装载。

1 内核模块 Makefile

内核模块软件包的 Makefile 文件和应用程序的 Makefile 结构几乎一致，在下图中一起展示，直观展示相同部分与差异内容。

用户态 应用程序包

```
# package/<PKG_NAME>/Makefile

include $(TOPDIR)/rules.mk

# 定义软件包信息:PKG_XX
PKG_NAME:=
PKG_VERSION:=
PKG_RELEASE:=
PKG_SOURCE:=
PKG_SOURCE_URL:=
HOST_PATCH_DIR :=

include $(INCLUDE_DIR)/package.mk

# 定义编译包信息
define Package/<pkgName>
define Package/<pkgName>/description
define Build/Prepare
define Build/Configure
define Build/Compile
define Package/<pkgName>/install

# 启动编译
$(eval $(call BuildPackage,<pkgName>))
```

内核态 内核模块包

```
# package/<PKG_NAME>/Makefile

include $(TOPDIR)/rules.mk

include $(INCLUDE_DIR)/kernel.mk

# 定义软件包信息:PKG_XX
PKG_NAME:=
PKG_VERSION:=
PKG_RELEASE:=
PKG_BUILD_DIR:=$(KERNEL_BUILD_DIR)/$(PKG_NAME)

include $(INCLUDE_DIR)/package.mk

# 定义编译包信息
define KernelPackage/<pkgName>
define KernelPackage/<pkgName>/description
define Build/Prepare
define Build/Configure
define Build/Compile
define KernelPackage/<pkgName>/install

# 启动编译
$(eval $(call KernelPackage,<pkgName>))
```

图 4-17 应用程序和内核模块软件包 Makefile 的对比

图中红色标识部分是应用程序和内核模块软件包 Makefile 文件的主要差别，差别主要体现在：

(1) 内核模块 Makefile 中必须引入文件 include/kernel.mk，因为内核模块启动编译的函数 KernelPackage 定义在 include/kernel.mk 中。

(2) 应用程序启动 package 编译的函数是 BuildPackage，而内核模块启动编译的函数是 KernelPackage。

2 文件与函数关系

内核模块软件包启动编译的函数 KernelPackage 在 include/kernel.mk 中定义，KernelPackage 会调用其他函数一起完成编译任务。函数 KernelPackage 调用的最重要函数是 BuildPackage，这个调用会使得内核模块的编译过程和应用程序的编译过程高度相同。内核模块编译相关的文件与函数关系总结如下图所示。

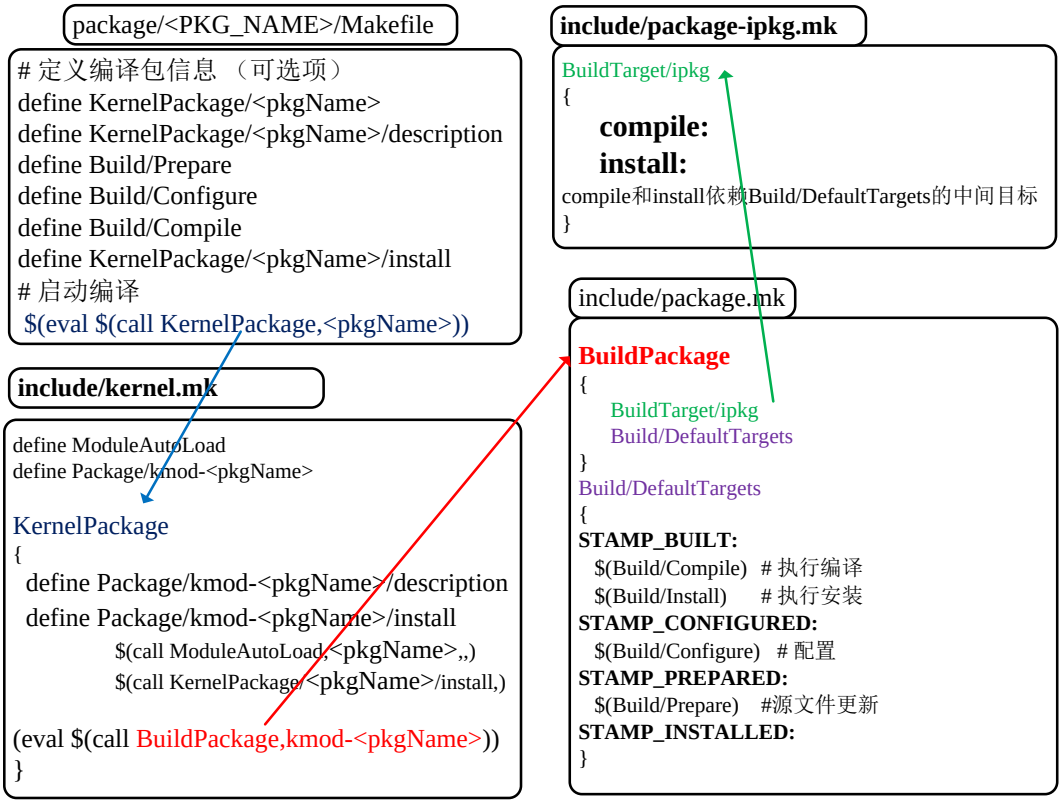


图 4-18 内核模块编译的文件与函数关系

说明：

1、内核模块编译的文件与函数关系与图 2-2 所示的应用程序的文件与函数关系非常相似，唯一不同是增加了文件 include/kernel.mk。

2、图 2-13 右下角中间编译目标 STAMP_BUILT 等，调用 build 函数(Build/Compile 等)，优先执行 package/包/Makefile 中定义的函数。如果未定义，则使用 include/package-default.mk 中定义的默认函数(Build/Compile/Default 等)。

3、kernel.mk 在内核模块编译过程中的作用可以总结如下：

include/kernel.mk

1 配置变量、编译/安装路径、编译选项
2 定义函数
（1） KernelPackage/Defaults KernelPackage/depends ModuleAutoLoad(自动装载)
（2） **KernelPackage**
改装package/<pkgName>/Makefile定义的编译包信息(函数KernelPackage/<pkgName>等)
KernelPackage/<pkgName> → Package/**kmod**-<pkgName>
KernelPackage/<pkgName>/description → Package/**kmod**-<pkgName>/description
KernelPackage/<pkgName>/configure → Package/**kmod**-<pkgName>/configure
KernelPackage/<pkgName>/install → Package/**kmod**-<pkgName>/install
调用BuildPackage函数，与应用程序包一样的编译流程
$$(eval $$(call **BuildPackage,kmod-<pkgName>**))

图 4-19 include/kernel.mk 在内核模块编译过程中的作用总结

3 compile 流程

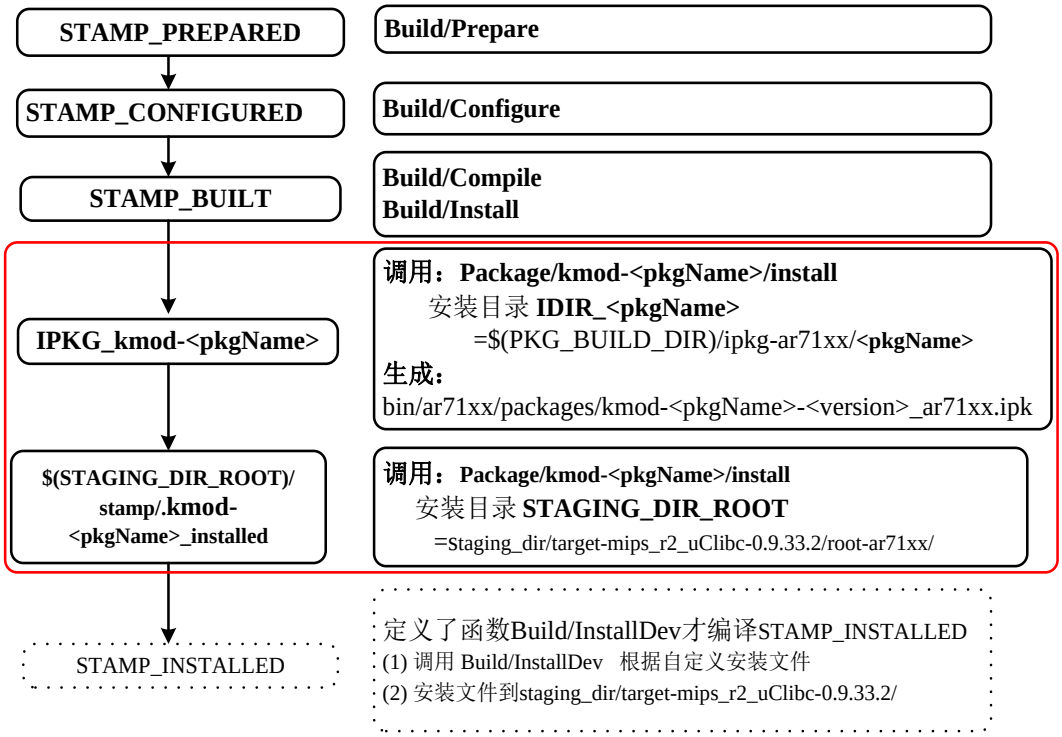


图 4-20 package 内核模块包 compile 的编译流程

图 4-21 内核模块包 compile 的部分中间目标的详细描述

说明：

1、package 内核模块包 compile 的流程与 package 应用程序的 compile 流程几乎一致，

以 STAMP 开头的中间目标(如 STAMP_BUILT)的执行内容相同。详尽说明参考本章 2.2.3 节的图 2-7 和 2-8。

2、图 2-13 展示的是图 2-12 中红色方框内的执行内容，与图 2-9 所示应用程序包执行内容几乎一致，外表不同的地方是<pkgName>的名字前面多了头部 kmod，最主要的差别是调用函数 Package/*/install 不一样，简述如下:

应用程序：调用 package/包/Makefile 中定义的 Package/<pkgName>/install

内核模块：调用 include/kernel.mk 中定义的 Package/kmod-<pkgName>/install

Package/kmod-<pkgName>/install 调用：

(1) include/kernel.mk 中的函数 ModuleAutoLoad

(2) package/包/Makefile 中定义的函数 KernelPackage/<pkgName>/install

3、函数 Package/kmod-<pkgName>/install 的功能描述如下图所示。

Package/kmod-<pkgName>/install

调用方法：$(**call Package/kmod-<pkgName>/install**, $(1))
参　　数: $(1)=安装路径

1 安装文件：$(FILES)
$(FILES)在package/包/Makefile中的函数KernelPackage/<pkgName>中定义
for mod in $(FILES)
if mod 存在
安装：$(1)/$(MODULES_SUBDIR)/mod = $(1)/lib/modules/3.3.8/mod
elseif mod 在文件(LINUX_DIR)/modules.builtin可以找到
提示已经安装
else 提示mod is missing
done

2 调用函数
（1） $(call **ModuleAutoLoad, <pkgName>, $$(1),** $(AUTOLOAD) **)**
只有当package/包/Makefile中的函数KernelPackage/<pkgName>中定义了
AUTOLOAD:=$(call AutoLoad,优先级,<pkgName>,boot值)
在安装目录$$(1)下，放置文件如下：
$$(1)
├─etc
├─modules.d/优先级-<pkgName>
├─modules-boot.d/优先级-<pkgName> （**boot值=1**）
└─CONTROL/postinst （**boot值=1**）

（2） $(call **KernelPackage/<pkgName>/install**, **$$(1)**)
在$(1)目录下，根据自定义**KernelPackage/<pkgName>/install的要求**安装文件

图 4-22 Package/kmod-<pkgName>/install 的功能描述

4 install 流程

如果软件包的配置选项为 Y，即 CONFIG_PACKAGE_kmod-<pkgName>=y，可以有效执行 install，install 的执行流程与安装效果如图 2-16 和 2-17 所示。

图4-23 内核模块install的执行流程

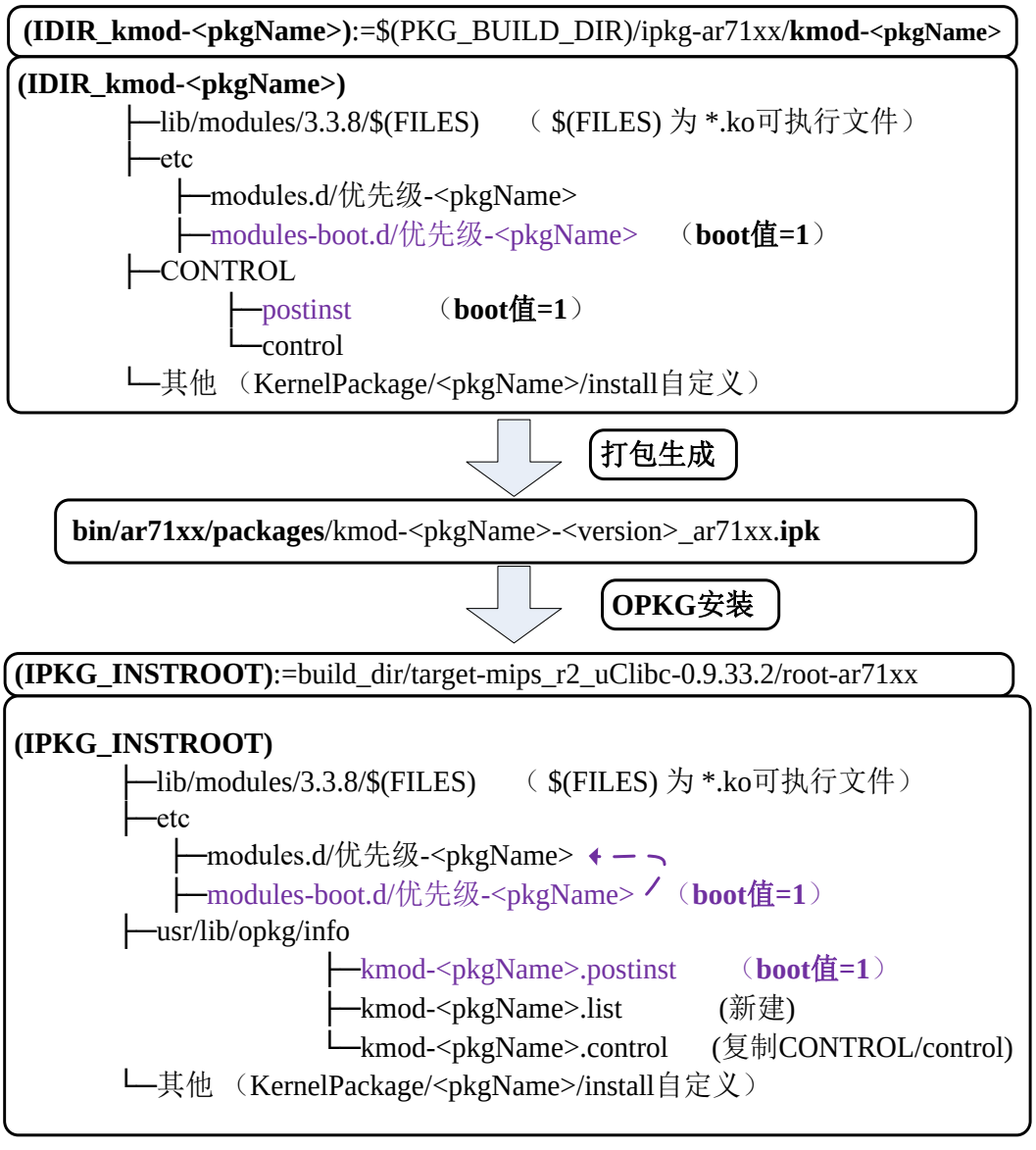


图4-24 内核模块生成ipk文件和安装的效果

4.4.6 package/kernel 的编译分析

package/kernel 包是内核模块编译的最重要入口，因为 package/kernel/Makefile 在定义 kernel 编译包的基础上，还有两条引入 mk 文件的语句，分别引入./modules 目录下的 mk 文件和 target/linux/*/modules.mk。前者定义通用内核模块编译包信息，后者定义平台内核模块编译包信息。package/kernel/Makefile 的主体内容如下图所示。

```
include $(TOPDIR)/rules.mk
include $(INCLUDE_DIR)/kernel.mk

PKG_NAME:=kernel
PKG_BUILD_DIR:=$(KERNEL_BUILD_DIR)/packages

include $(INCLUDE_DIR)/package.mk

CONFIG_PACKAGE kernel=y
# kernel包为必选项

define Package/kernel
define Package/kernel/install
$(eval $(if $(DUMP),,$(call BuildPackage,kernel)))

include $(sort $(wildcard ./modules/*.mk))
# 引入 package/kernel/modules/*.mk (通用内核模块)

-include $(TOPDIR)/target/linux/*/modules.mk
# 引入 target/linux/<BORAD>/modules.mk (平台内核模块)
# 如: target/linux/ar71xx/modules.mk
#     target/linux/x86/modules.mk  ...
```

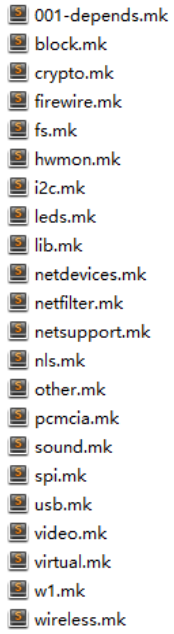


图4-25 package/kernel/Makefile的主体内容

内核模块编译包信息的结构如下图所示，左边为通用平台内核模块，右边为专用平台内核模块。

```
# package/kernel/modules/usb.mk

define KernelPackage/usb-storage
  SUBMENU:=USB Support
  TITLE:=USB Storage support
  DEPENDS:=+!TARGET_etrax:kmod-usb-core +kmod-scsi-core
  KCONFIG:=CONFIG_USB_STORAGE
  FILES:=$(LINUX_DIR)/drivers/usb/storage/usb-storage.ko
  AUTOLOAD:=$(call AutoLoad,60,usb-storage,1)
endef

define KernelPackage/usb-storage/description
 Kernel support for USB Mass Storage devices
endef

$(eval $(call KernelPackage,usb-storage))
```

```
# target/linux/ar71xx/modules.mk

define KernelPackage/leds-wndr3700-usb
  SUBMENU:=$(LEDS_MENU)
  TITLE:=WNDR3700 USB LED support
  DEPENDS:=@TARGET_ar71xx
  KCONFIG:=CONFIG_LEDS_WNDR3700_USB
  FILES:=$(LINUX_DIR)/drivers/leds/leds-wndr3700-usb.ko
  AUTOLOAD:=$(call AutoLoad,60,leds-wndr3700-usb)
endef

define KernelPackage/leds-wndr3700-usb/description
 Kernel module for the USB LED on the NETGEAR WNDR3700 board.
endef

$(eval $(call KernelPackage,leds-wndr3700-usb))
```

图4-26 内核模块编译包

两者编译包的结构基本相同，唯一的差别是专用平台的 DEPENDS 必须包含 TARGET_<平台>。界面配置时，Target System 选择某平台，该平台的内核模块才会出现在配置界面上供选择。内核模块的默认一级菜单为 Kernel modules，因此配置路径通常为：Kernel modules → <SUBMENU>。KernelPackage/<pkgName> 在配置界面的选择项名字为 kmod-<pkgName>，键入 y/m/n，结果记录在.config。

package/kernel/compile 首先以应用包的通用流程编译 kernel 包，生成 ipk 文件：kernel_3.3.8-1-<MD5>_ar71xx.ipk。然后编译被选中的内核模块，主要工作

是将 KernelPackage/<pkgName>中定义的 FILES:=<路径>/xxx.ko 文件，加上其他信息记录文件，打包成 kmod-<pkgName>_3.3.8_ar71xx.ipk。具体执行内容参考图 4-21 和图 4-22。

package/kernel/install 分别安装 kernel_3.3.8-1-xxx_ar71xx.ipk 和 kmod-<pkgName>_3.3.8_ar71xx.ipk，安装过程分别见图 4-15、4-15 和图 4-23、4-24。

4.4.7 uboot 编译

uboot 编译的选项配置界面如下图所示：

make menuconfig → Boot loader

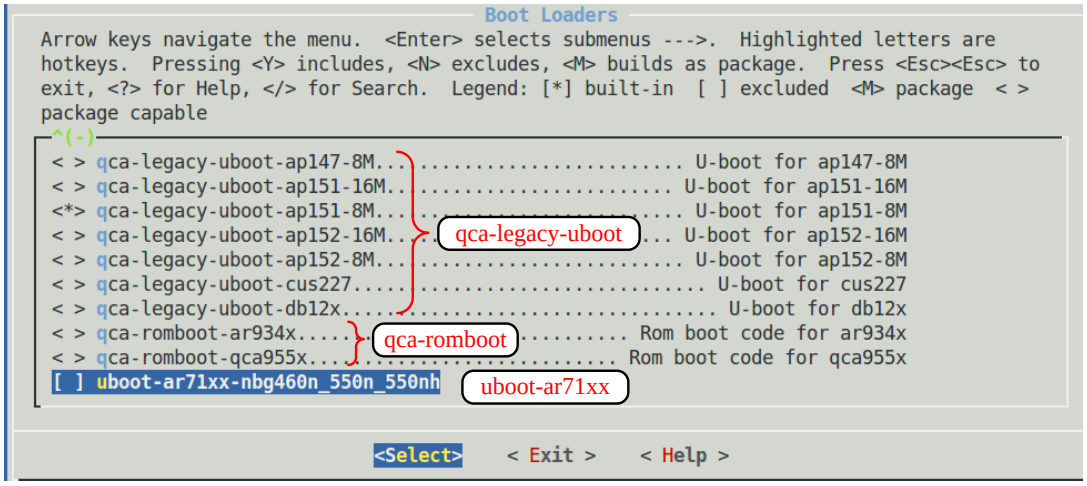


图 4-27 ar71xx uboot 编译的选项配置界面

QSDK 中，不同<TARGET>，Boot loader 界面显示可以配置的 uboot 选项不同，当<TARGET>=ar71xx 时，Boot loader 界面如上图所示。

qca-legacy-uboot 和 qca-romboot 包的 Makefile 中有如下语句：

include $(INCLUDE_DIR)/local-development.mk

表示源码放置在本地 qca 目录下，完整目录为

(TOPDIR)/qca/src/{ qca-legacy-uboot、qca-romboot }

编译 package/uboot-ar71xx 包，由于无法找到源码(dl 目录下没有)，没有编译成功。

在 Boot loader 界面配置并保存之后，可以启动编译 package/qca-romboot 软件包，单独编译的命令：

make package/qca-romboot/compile

编译之后的产物有：

build_dir/linux-ar71xx_generic/qca-romboot-g93eccd2/sco/{ rombootdrv.elf 和 rombootdrv.bin }

单独编译 package/qca-legacy-uboot 软件包的命令：

make package/qca-legacy-uboot/compile

编译之后的产物有：

build_dir/target-mips_.../qca-legacy-uboot/<ap>/{ uboot uboot.bin }
bin/ar71xx/openwrt-ar71xx-<ap>-qca-legacy-uboot.bin
或者 bin/ar71xx/openwrt-ar71xx-<ap>-qca-legacy-uboot.2fw

编译 package/qca-legacy-uboot 与普通 package 包的函数关系、目标依赖、编译流程完全一致。只是 package/qca-legacy-uboot/Makefile 中自定义了操作函数 Build/Configure 和 Build/Compile，编译过程中优先调用自定义操作函数。

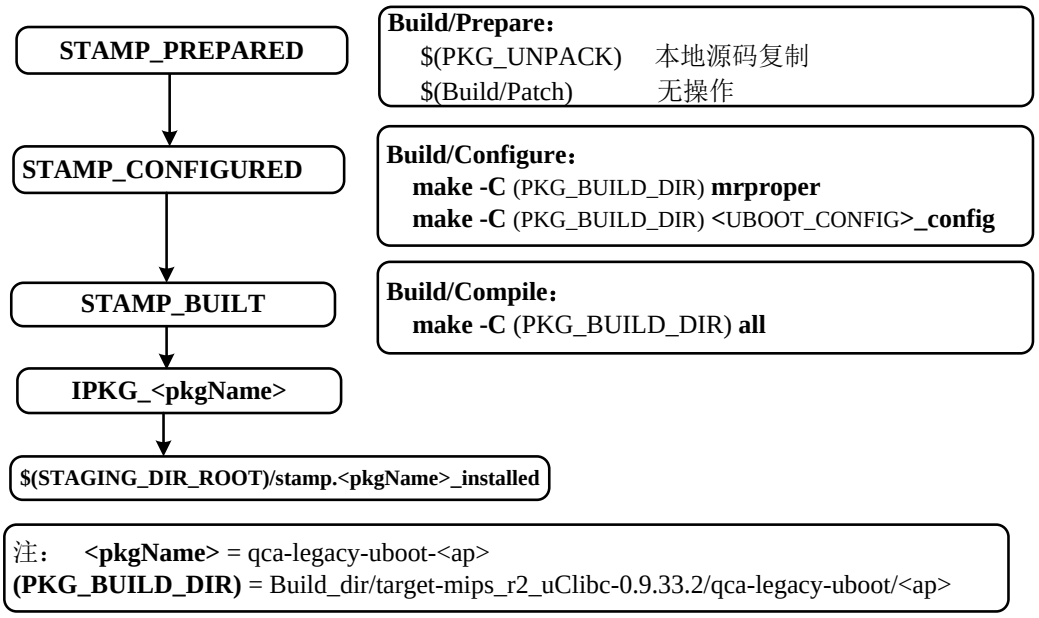


图 4-28 package/qca-legacy-uboot 的编译流程

IPKG_<pkgName>

(1) $(call Package/<pkgName>/install , $$(IDIR_<pkgName>))
调用函数**Package/<pkgName>/install，执行内容为:**
install -m0644 (PKG_BUILD_DIR)/u-boot.bin bin/ar71xx/openwrt-ar71xx-<ap>-qca-legacy-uboot.bin

(2) 将软件包的基本信息（Package: Version: Depends: ...）
写入 (IDIR_<pkgName>)/CONTROL/control

(3) 生成ipk文件: (IPKG_BUILD) (IDIR_<pkgName>) (PACKAGE_DIR)
将IDIR_<pkgName>目录下文件打包成.ipk，
放置在**$(PACKAGE_DIR) 目录下**
→ bin/ar71xx/packages/<pkgName>-<version>_ar71xx.ipk

IDIR_<**pkgName**>:=$(PKG_BUILD_DIR)/ipkg-ar71xx/<**pkgName**>

$(STAGING_DIR_ROOT)/stamp.<pkgName>_installed

(1) 新建文件夹 (STAGING_DIR_ROOT)/tmp-<pkgName>
(2) **$(call Package/<pkgName>/install , $(STAGING_DIR_ROOT)/tmp-<pkgName>))**
与**IPKG_<pkgName>调用Package/<pkgName>/install的执行内容完全一样**

(3) 复制 (STAGING_DIR_ROOT)/tmp-<pkgName>/. → $(STAGING_DIR_ROOT)/
(4) 删除文件夹 (STAGING_DIR_ROOT)/tmp-<pkgName>
(5) touch $(STAGING_DIR_ROOT)/stamp/.<pkgName>_installed

(STAGING_DIR_ROOT) = staging_dir/target-mips_r2_uClibc-0.9.33.2/root-ar71xx/

图 4-29 编译包<pkgName>的中间目标编译的详尽描述

说明：

1、Package/qca-legacy-uboot 有 nor 和 nand 两种不同的编译包类型，二者的调用方式如下：

$(eval $(call Package/qca-legacy-uboot/nor,ap135))

$(eval $(call Package/qca-legacy-uboot/nand,ap135-nand,qca955x))

Package/qca-legacy-uboot/nor 和 Package/qca-legacy-uboot/nand 唯一的区别在于 Package/qca-legacy-uboot-<ap>/install 的操作不完全相同。

```
define Package/qca-legacy-uboot/nor
  $(call Package/qca-legacy-uboot/common,$(1),$(2))

  define Package/qca-legacy-uboot-$(1)/install
    $(INSTALL_DATA) $(PKG_BUILD_DIR)/u-boot.bin $(BIN_DIR)/$(UBOOT_IMAGE)
  endef

  $$(eval $$(call BuildPackage,qca-legacy-uboot-$(1)))
endef

# $(INSTALL_DATA) = install -m0644

define Package/qca-legacy-uboot/nand
  $(call Package/qca-legacy-uboot/common,$(1),+qca-romboot-$(2))

  define Package/qca-legacy-uboot-$(1)/install
        UTILPATH=$(STAGING_DIR_HOST)/bin \
        $(STAGING_DIR_HOST)/bin/mk2stage-$(2) \
        -1 $(STAGING_DIR)/boot/openwrt-$(BOARD)-$(2)-rombootdrv.bin \
        -2 $(PKG_BUILD_DIR)/u-boot.bin \
        -o $(BIN_DIR)/$(patsubst %.bin,%.2fw,$(UBOOT_IMAGE))
  endef

  $$(eval $$(call BuildPackage,qca-legacy-uboot-$(1)))
endef

# UTILPATH= /staging_dir/host/bin    staging_dir/host/bin/mk2stage-qca955x
-1 staging_dir/target-mips_r2_uClibc-0.9.33.2/boot/openwrt-ar71xx-qca955x-rombootdrv.bin
-2 build_dir/target-mips_r2_uClibc-0.9.33.2/qca-legacy-uboot/ap135-nand/u-boot.bin
-o bin/ar71xx/openwrt-ar71xx-ap135-nand-qca-legacy-uboot.2fw
```

Package/qca-legacy-uboot-<ap>/install 的执行内容均为将构建目录下的 u-boot.bin 安装在不同的目录，只是安装方式、目录、名称存在差异。

编译 package/qca-legacy-uboot 包时，真正编译 uboot 的命令有三个：

make -C (PKG_BUILD_DIR) **mrproper**

make -C (PKG_BUILD_DIR) <UBOOT_CONFIG>**_config**

make -C (PKG_BUILD_DIR) **all**

(PKG_BUILD_DIR) =build_dir/target-mips /qca-legacy-uboot/<ap>。三个命令的执行内容总结如下：

(1) mrproper 和<UBOOT_CONFIG>_config

图 4-29 mrproper 和<UBOOT_CONFIG>_config 的执行内容总结

(1) all

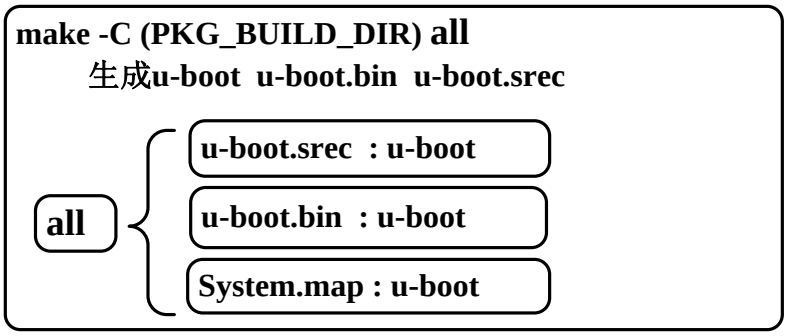


图 4-29 目标 all 的依赖关系

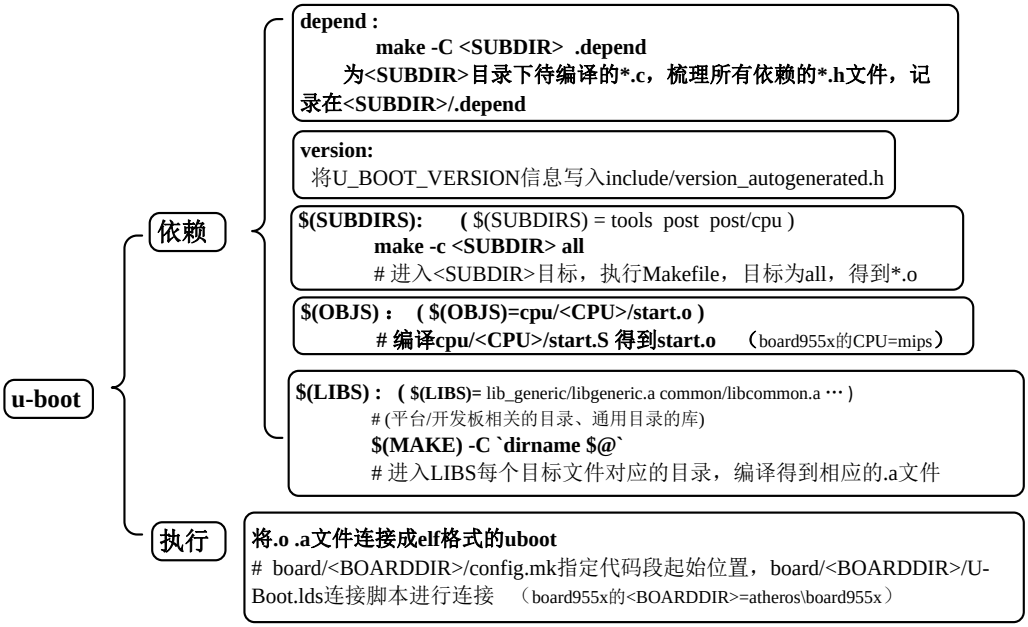


图 4-29 目标 u-boot 的依赖关系与执行内容

连接.o .a 文件成 uboot，只需要编译目标$(OBJS)和$(LIBS)生成的文件，目标$(SUBDIRS)生成的文件没有被利用？？？

u-boot

```
# 将*.o *.a文件连接成elf格式的uboot，连接规则(命令)如下：
UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
$(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) $(BOARD_EXTRA_OBJS) \
--start-group $(LIBS) --end-group $(PLATFORM_LIBS) -Map u-boot.map -o u-boot

$(OBJDUMP)=Mips-openwrt-linux-uclibc-objdump
$(LD)=Mips-openwrt-linux-uclibc-ld
$(LDFLAGS) =  -Bstatic -T /build_dir/target-mips_r2_uClibc-0.9.33.2/qca-legacy-uboot/<ap>/board/atheros/board956x/u-boot.lds -Ttext 0x9f000000
# $(LDFLAGS)确定连接方式
$(OBJS)= cpu/mips/start.o
$(LIBS)=lib_generic/libgeneric.a common/libcommon.a board/atheros/board956x/libboard956x.a cpu/mips/libmips.a lib_mips/libmips.a drivers/libdrivers.a net/libnet.a rtc/librtc.a
 $(PLATFORM_LIBS)= -L /staging_dir/toolchain-mips/lib/gcc/mips-openwrt-linux-uclibc/4.6.3 -lgcc
```

u-boot.srec

```
# 将elf格式的uboot，转换成S-Record格式：
$(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
# mips-openwrt-linux-uclibc-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec
```

u-boot.bin

```
# 将elf格式的uboot，转换成二进制格式：
$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
# mips-openwrt-linux-uclibc-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

图 4-29 不同格式的 u-boot 的生成方式

总结 make -C (PKG_BUILD_DIR) all（uboot 编译）的流程如下：

(1) 编译 cpu/<CPU>/start.S 得到 start.o，还可能编译 cpu/<CPU>/其他文件；

(2) 平台/开发板相关的目录、通用目录的目录，使用 Makefile，生成相应的

库(*.a)；

(3) 将(1)和(2)得到的.o .a文件，按着board/<BOARDDIR>/config.mk指定代码段起始位置，board/<BOARDDIR>/U-Boot.lds连接脚本进行连接，连接成elf格式的u-boot；

(4) 将 elf 格式的 u-boot 转换成二进制格式的 u-boot.bin 和 S-Record 格式的 u-boot.srec。

4.5 target 包的编译分析

target 目录下有 linux、imagebuilder、sdk、toolchain 四个软件包，启动单个软件包编译的方式主要有两种。

方式一：界面配置，在如图 3-13 所示的配置界面(make menuconfig)选择需要编译的 target 包，然后输入编译命令：make。正常编译情形下，选择编译的 target 包会被编译，并在 bin 目录生成相应的目标文件。

方式二：单独编译，只需要输入对应 target 包的编译命令即可，如:make target/sdk/install，make target/toolchain/install。这种方式成功的前提是编译此目标需要的文件已经存在，如果文件不完整，可能在编译过程报错退出，也可能不报错，但编译后的目标产物无法正常使用。

target 目录下的四个软件包，linux 包是必须编译的，是生成固件的核心。由于编译 target/linux 包涉及到的内容比较多，单独在本章 4.6 节阐述。4.5 节主要解释 toolchain、sdk、imagebuilder 包的编译分析，从目标关系、编译执行流程等方面进行分析。

4.5.1 target/toolchain

编译 target/toolchain 的作用是打包 toolchain 工具链。单独编译 target/toolchain 的命令为：make target/toolchain/compile 或者 make target/toolchain/install 或者 target/toolchain/all。编译的目标产物为 bin/$(BOARD)目录下的压缩文件，压缩名字为 OpenWrt-Toolchain-$(BOARD)-for-$(ARCH)$(ARCH_SUFFIX)-gcc-$(GCCV)$(DIR_SUFFIX).tar.bz2，简称$(TOOLCHAIN_NAME).tar.bz2。target/toolchain/Makefile 的目标依赖关系总结如下图所示。

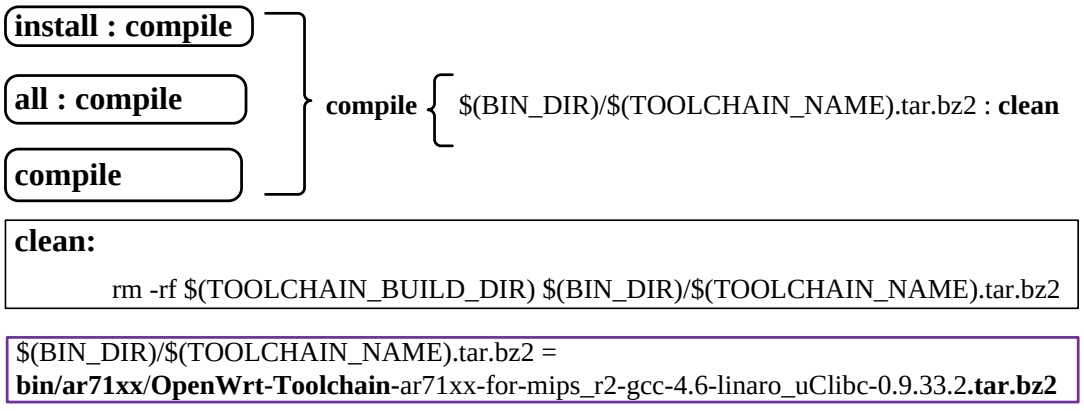


图 4-30 target/toolchain/Makefile 的目标关系

在 qsdk 系统中:make target/toolchain/compile，目标文件的名字如图所示。编译的主要工作是将 staging_dir/toolchain-ar71xx-for-mips_r2-gcc-4.6-linaro_uClibc-0.9.33.2 目录下的大部分安装文件和其他配置文件打包成压缩文件，放置在 bin/ar71xx 目录下，

大致执行流程表述如下图所示。

1、复制文件
将：staging_dir/ toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2中所有文件(除去ccache、Stamp等文件夹)
复制到build_dir/target-mips_r2_uClibc-0.9.33.2/OpenWrt-Toolchain-ar71xx-for-mips_r2-gcc-4.6-linaro_uClibc-0.9.33.2目录下

将：LICENSE target/toolchain/files/README.TOOLCHAIN target/toolchain/files/wrapper.sh
复制到build_dir/target-mips_r2_uClibc-0.9.33.2/OpenWrt-Toolchain-ar71xx-for-mips_r2-gcc-4.6-linaro_uClibc-0.9.33.2目录下

2、删除、修改、写入部分文件
删除: .git .svn CVS
修改: wrapper.sh、mips-openwrt-linux-uclibc-gcc等
写入： version.mk

3、压缩安装
将：build_dir/target-mips_r2_uClibc-0.9.33.2/OpenWrt-Toolchain-ar71xx-for-mips_r2-gcc-4.6-linaro_uClibc-0.9.33.2目录下所有文件压缩并放置：bin/ar71xx/**OpenWrt-Toolchain-ar71xx-for-mips_r2-gcc-4.6-linaro_uClibc-0.9.33.2.tar.bz2**

图 4-31 target/toolchain/compile 的编译执行流程

说明：

单独编译 target/toolchain 软件包，需要 toolchain 已经编译并安装完成，即文件夹 staging_dir/ toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2 已经存在且完整。若在 make target/toolchain/install 之前没有 make toolchain/install，会报错“build failed”，并退出。

4.5.2 target/sdk

单独编译 target/sdk 包的命令：make target/sdk/compile、make target/sdk/install、target/sdk/all。编译的目标产物为 bin/$(BOARD)目录下的压缩文件，压缩文件的名字为 OpenWrt-SDK-$(BOARD)-for-$(PKG_OS)-$(PKG_CPU)-gcc-$(GCCV)_$(LIBC)-$(LIBCV).tar.bz2，简称$(SDK_NAME).tar.bz2。target/sdk/Makefile 的目标依赖关系总结如下图所示。

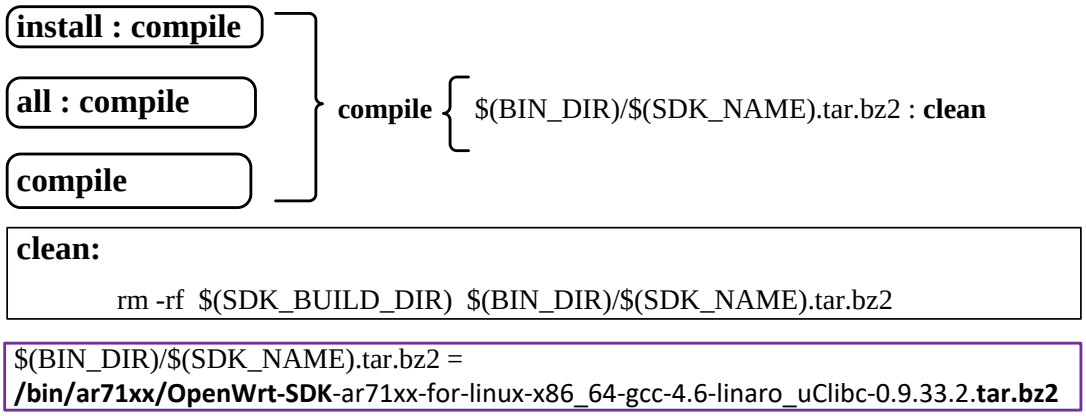


图 4-32 target/sdk/Makefile 的目标关系

在 qsdk 系统中 make target/sdk/compile，生成的目标文件全名如图 4-16 所示。编译的主要工作是将 openwrt 系统应用开发所必须的工具和配置文件打包成压缩文件，放置在 bin/ar71xx 目录下，大致执行流程表述如下图所示。

构建SDK的目录：简称:(SDK)
(SDK) = **build_dir/target-mips**_r2_uClibc-0.9.33.2/**OpenWrt-SDK-**ar71xx-for-linux-x86_64-gcc-4.6-linaro_uClibc-0.9.33.2
目标文件: OpenWrt-SDK-ar71xx-for-linux-x86_64-gcc-4.6-linaro_uClibc-0.9.33.2**.tar.bz2**
放置路径：bin/ar71xx

1、 新建文件夹 (SDK)/{ dl package target/linux }

2、复制文件/文件夹(黑体)
(1) **docs staging_dir/host staging_dir/target-mips_r2_uClibc-0.9.33.2 staging_dir/toolchain-mips_r2_gcc-4.6-linaro_uClibc-0.9.33.2 LICENSE rules.mk .config** target/sdk/files/{ **Config.in Makefile README.SDK** } → **(SDK)/**
(2) target/sdk/files/package/**rules.mk** + package/**Makefile** → (SDK)/package/ **{rules.mk、Makefile}**
(3) target/linux/{**ar71xx 、generic**} → (SDK)/target/linux/{**ar71xx 、generic**}

3、(SDK)目录下删除、写入文件
删除: .git .svn CVS
写入： feeds.conf.default include/version.mk

4、压缩安装
OpenWrt-SDK-ar71xx-for-linux-x86_64-gcc-4.6-linaro_uClibc-0.9.33.2 **压缩成目标文件并放置在bin/ar71xx/**

图 4-33 target/sdk/compile 的编译执行流程

说明：单独编译 target/sdk 软件包，需要 tools、toolchain、package、target/linux 已经编译并安装完毕，即文件夹 staging_dir/ {host、target-xx、toolchain-xx}已经存在且完整。若编译 target/sdk 之前，相关文件不完备，会报错“build failed”，并退出。

编译 target/sdk 的作用是打包 sdk 工具包，sdk 工具包可以理解为一个简化版 openwrt 系统。

图 4-34 qsdk 和 sdk 目录结构的对比

上图左半部分为 qsdk 目录结构，右半部分为 qsdk 编译生成的 sdk 包的目录结构。总体来看，sdk 的目录结构与 openwrt 基本一致，只是在原有系统的基础上进行裁剪。sdk 工具包的简化表现在如下三个方面：

1、没有 tools、toolchain 目录，tools 和 toolchain 的编译产物已经安装在 staging_dir 目录下，因此在 sdk 中，工具和工具链已经配置完备，省去工具编译的环节。

2、dl 为空目录，没有本地保存的源文件。package 目录下只有 Makefile 和 rules.mk 文件，不包含其他 package 包，开发者可以在 package 目录下新建 package 软件包，开发应用程序。

3、target 目录只有文件夹 linux，没有 sdk、imagebuilder、toolchain 子目录。而且 linux 目录下只有$(GENERIC_PLATFORM_DIR)和$(PLATFORM_DIR)。其他 CPU 平台的软件包全部省去。$(GENERIC_PLATFORM_DIR)为通用平台 generic，$(PLATFORM_DIR)为 CPU 平台的支持文件，公司目前使用的 QSDK/SLP 均为 ar71xx。

4、sdk 包为简化版的 openwrt 系统，sdk 包顶层 Makefile 的结构与 openwrt 系统顶层 Makefile 非常相近。因此，二者的编译框架几乎一致，区别在于：openwrt 系统需要编译 tools 和 toolchain，sdk 包的工具安装完毕，没有编译 tools 和 toolchain 的流程。

4.5.3 target/imagebuilder

单独编译 target/ imagebuilder 包的命令为：make target/imagebuilder/compile、make target/imagebuilder/install、target/imagebuilder/all。编译的目标产物为文件：OpenWrt-ImageBuilder-ar71xx_generic-for-linux-x86_64.tar.bz2，简称$(IB_NAME).tar.bz2。target/ imagebuilder/Makefile 文件中的目标依赖关系见下图。

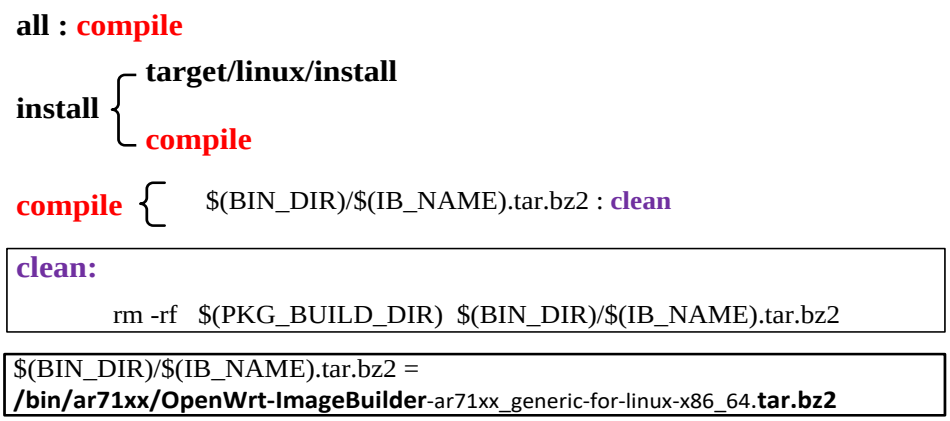


图 4-35 target/imagebuilder/Makefile 的目标关系

说明：

1、编译 target/imagebuilder 包时，如果命令为：make target/imagebuilder/install，根据依赖关系，会先编译 target/linux/install。

2、如果命令为 make target/imagebuilder/compile 或者 make target/imagebuilder/all，不会编译 target/linux/install，但默认要求之前已经编译 target/linux/install。如果编译 imagebuilder 之前没有编译过 target/linux/install，出现关键文件缺失，最后生成的压缩文件会因为缺少文件而无法实现其完整功能。

在 qsdk 系统中 make target/imagebuilder/install，生成的目标文件全名如上图所示，编译的大致执行流程表述如下图所示。

A make -C target/linux install （之前必须完成）

B make -C taget/imagebuilder install

构建ImageBuilder的目录（简称：ImageBuilder_DIR）
(ImageBuilder_DIR) = build_dir/target-mips_r2_uClibc-0.9.33.2/**OpenWrt-ImageBuilder**-ar71xx_generic-for-linux-x86_64
目标文件：OpenWrt-ImageBuilder-ar71xx_generic-for-linux-x86_64**.tar.bz2** **放置路径：bin/ar71xx**

1、(ImageBuilder_DIR) 新建文件夹
build_dir/linux-ar71xx_generic build_dir/linux-ar71xx_generic/linux-3.3.8 staging_dir/host target scripts

2、复制文件
(1) 文件/文件夹(黑体)
.config include scripts rules.mk target/imagebuilder/files/{ **Makefile、repositories.conf** }
tmp/**.targetinfo** tmp/**.packageinfo** → **(ImageBuilder_DIR)/**

(2) 编译package包的生成文件
bin/ar71xx/packages/{**所有文件**} → **(ImageBuilder_DIR)**/package/{**所有文件**}

(3) cpu平台文件
target/linux/{**所有文件**} → **(ImageBuilder_DIR)**/target/linux/{**所有文件**}

(4) 内核文件（已经编译）
build_dir/linux-ar71xx_generic/* → **(ImageBuilder_DIR)**/Build_dir/linux-ar71xx_generic/*

(5) 工具文件 staging_dir/host/**bin** → **(ImageBuilder_DIR)**/staging_dir/host/**bin**

3、(ImageBuilder_DIR) 删除、写入文件
删除: .git .svn target/linux/*/files target/linux/*/ patches{,-*}
写入： include/version.mk packages/repositories.conf

4、压缩安装
OpenWrt-ImageBuilder-ar71xx_generic-for-linux-x86_64**压缩成目标文件并放置在bin/ar71xx/**

图 4-36 target/imagebuilder/install 的编译执行过程

ImageBuilder目录结构

```
.conf
.packageinfo
.targetinfo
repositories.conf
Makefile
rules.mk

include/
scripts/
package/{ *.ipk Packages}
target
  └─linux
      └─{genric、ar71xx、adm5120...}
staging_dir
  └─host
      └─bin/{aclocal lua … }
build_dir
  └─linux-ar71xx_generic
       └ vmlinux vmlinux.bin.lzma
       └ vmlinux.elf  root.squashfs
       └ linux-3.3.8/{.config }
```

图 4-37 ImageBuilder 包的目录结构

ImageBuilder 压缩包的主要用途是构建 image，压缩包里面包含构建 image 所需要的应用包(xxx.ipk)、编译之后的内核文件、CPU 平台文件。解压此包，执行相应的命令，如 make image，就可以启动 image 的构建。

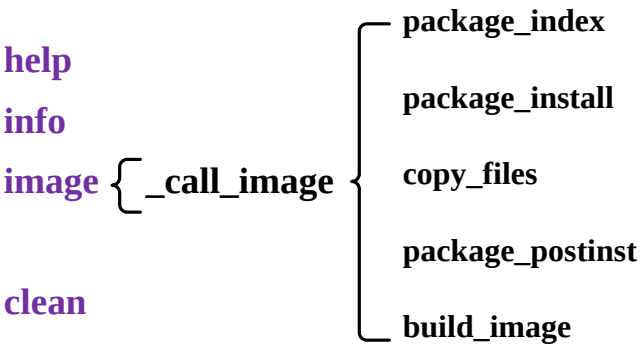


图 4-38 ImageBuilder 包的 Makefile 的目标关系

说明：image 的构建过程与 target/linux/ar71xx/image 包的编译类似，可参考本章 4.8 节

4.6 target/linux

如第三章 3.5 节所述，target/Makefile 定义三个动态目标:$(target/stamp-prereq)、$(target/stamp-compile)和$(target/stamp-install)。根据依赖关系，三个动态目标会分别编译 target/linux/prereq、target/linux/compile、target/linux/install。除此之外，还可以单独输入编译命令：make target/linux/<TARGET>，<TARGET>可以是 clean、install、prepare 等。有效的编译命令执行的操作均为：进入 target/linux 目录，编译文件 Makefile，目标为<TARGET>。

```
# target/linux/Makefile

prereq clean download prepare compile install menuconfig nconfig oldconfig update refresh: FORCE
	@+$(NO_TRACE_MAKE) -C $(BOARD) $@
#	make V=ss	-C ar71xx  <TARGET>
```

图 4-39 target/linux/Makefile 的编译目标

上图中所示的 target/linux/Makefile 定义的编译目标，执行的操作基本一致，均为：make -C <BORAD> <TARGET>。其中<BORAD>为 CPU 平台(芯片方案)，目前主要为 ar71xx，进入目录 ar71xx，执行 Makefile，目标为<TARGET>。

4.6.1 编译目标与文件关系

```
# target/linux/ar71xx/Makefile

include $(TOPDIR)/rules.mk
# 定义软件包信息
ARCH:=mips
BOARD:=ar71xx
BOARDNAME:=Atheros AR7xxx/AR9xxx
FEATURES:=squashfs jffs2 targz jffs2_nand
CFLAGS:=-Os -pipe -mips32r2 -mtune=74kc \
        -fno-caller-saves -freorder-blocks
SUBTARGETS:=generic nand
LINUX_VERSION:=3.3.8

include $(INCLUDE_DIR)/target.mk

# 配置信息
DEFAULT_PACKAGES += \
    kmod-leds-gpio kmod-gpio-button-hotplug swconfig \
    kmod-ledtrig-default-on kmod-ledtrig-timer \
    kmod-button-hotplug kmod-ath9k wpad-mini \
    kmod-wdt-ath79  kmod-ledtrig-netdev uboot-envtools
# 定义编译包信息
define Target/Description
    Build firmware images for Atheros AR7xxx/AR9xxx based boards.
endef

$(eval $(call BuildTarget))
```

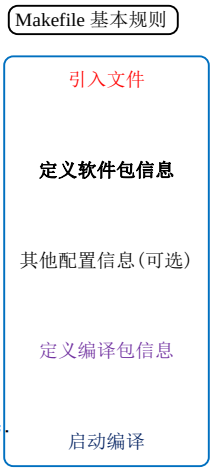


图 4-40 target/linux/ar71xx/Makefile

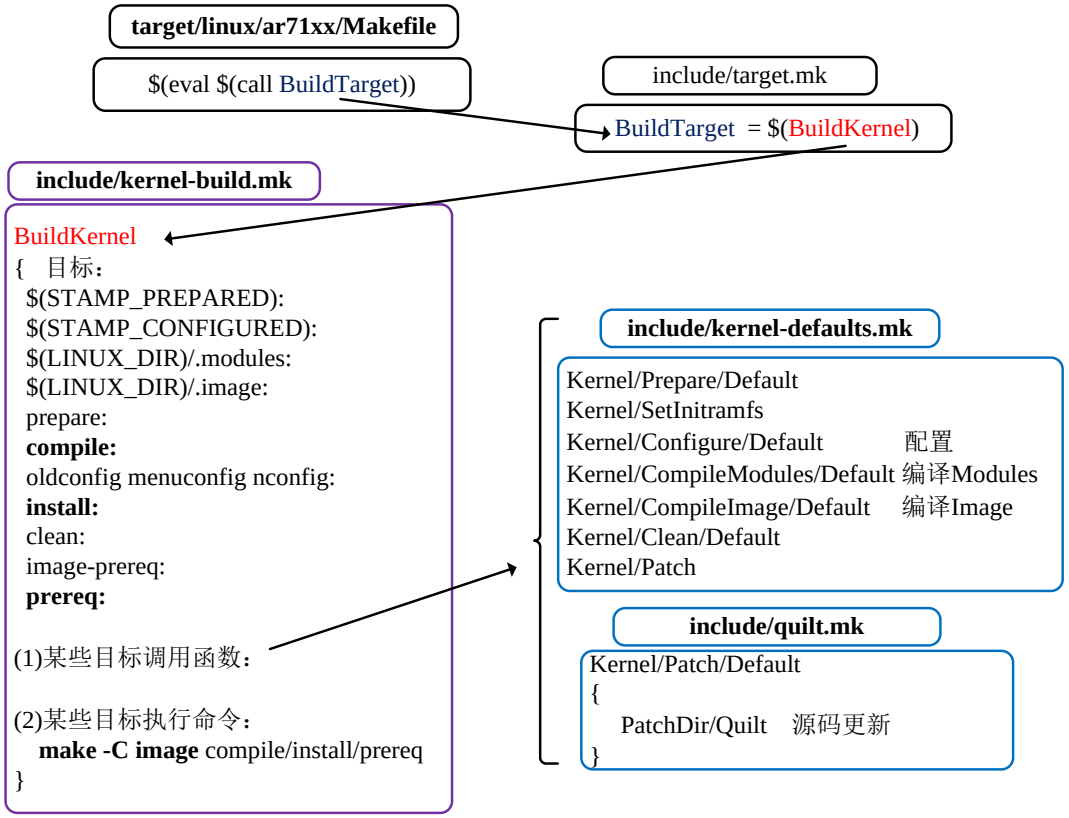
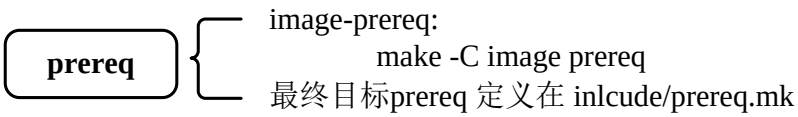


图 4-41 target/linux/ar71xx/Makefile 的编译目标及其文件关系

上图中的函数 BuildKernel 定义了编译 target/linux/ar71xx 包可能用到的目标：prereq、prepare、compile、install、clean 等。这些目标及其依赖目标（STAMP_CONFIGURED、$(LINUX_DIR)/.image），最终会调用 include/{quilt.mk、download.mk、kernel-defaults.mk}等文件中定义的函数，如：Kernel/Prepare/Default。

编译 target/linux/ar71xx 包最重要的两个目标是 compile 和 install，4.6.2 和 4.6.3 节将分别进行说明。除 compile 和 install 之外，其他编译目标的依赖关系和执行内容如下图所示。



clean:
rm -rf build_dir/linux-ar71xx_generic

oldconfig menuconfig nconfig: $(STAMP_PREPARED) $(STAMP_CHECKED)
rm -f $(STAMP_CONFIGURED)

图 4-42 target/linux/ar71xx 包的其他编译目标

4.6.2 target/linux/ar71xx 的编译

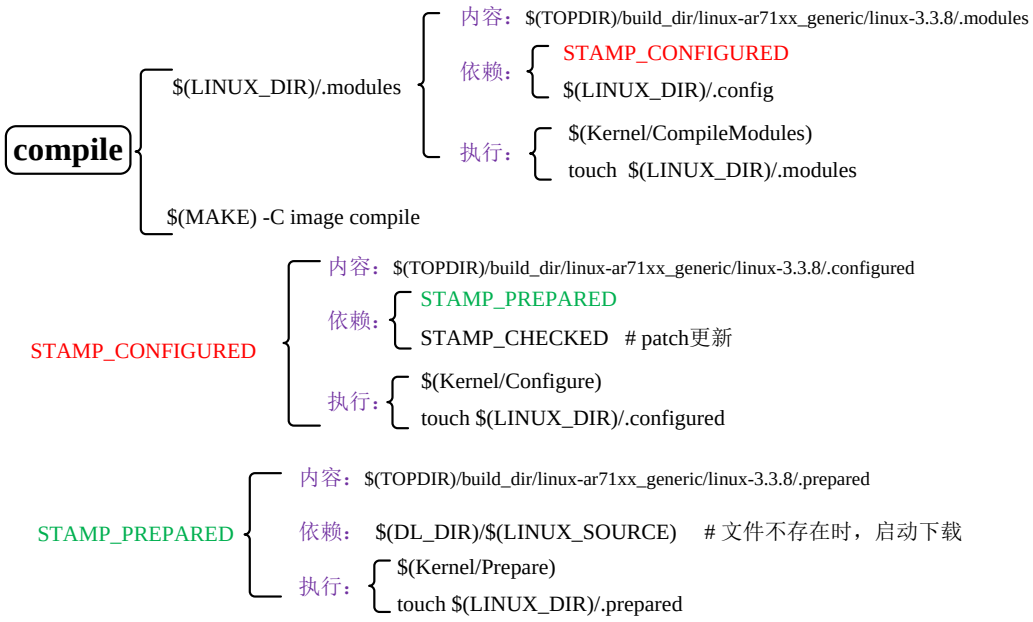


图 4-43 target/linux/ar71xx/compile 的目标依赖关系

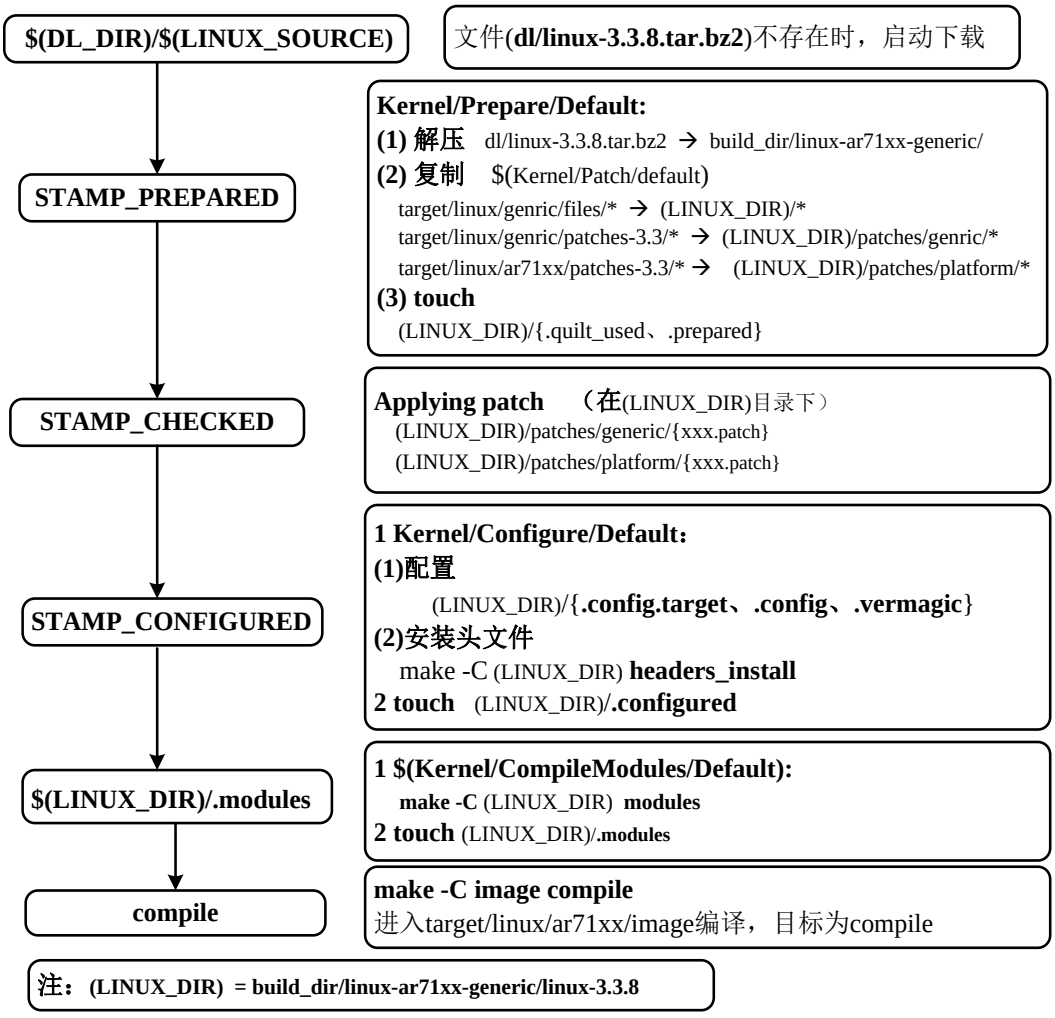


图 4-44 target/linux/ar71xx/compile 的编译执行流程

说明：

1、图 4-27 中，红色标识同一目标，浅蓝色文件为中间目标编译完成的标志文件，避免重复编译。

2、根据依赖关系，编译次序: Kernel/Prepare→ Kernel/Configure→ Kernel/CompileModules → make -C image compile。

3、根据 kernel-built.mk 对函数 Kernel/Prepare 的定义，最终会调用 kernel-defaults.mk 中定义的 Kernel/Prepare/Default。Kernel/Configure 和 Kernel/CompileModules 类似。

4、Kernel/Prepare/Default中调用include/quilt.mk中的函数Kernel/Patch/Default，将target/linux目录下与平台和版本相关的patch文件复制到编译目录(LINUX_DIR)/patches下面。区分generic平台和cpu平台:

target/linux/genric/patches-3.3/* → (LINUX_DIR)/patches/genric/*

target/linux/ar71xx/patches-3.3/* → (LINUX_DIR)/patches/platform/* 。

5、应用patch更新源文件，由目标STAMP_CHECKED（include/quilt.mk中定义）的执行语句来完成。

6、make -C (LINUX_DIR) <TARGET>和 make -C image compile 和分别见 4.6.4 和 4.7。

4.6.3 target/linux/ar71xx 的安装

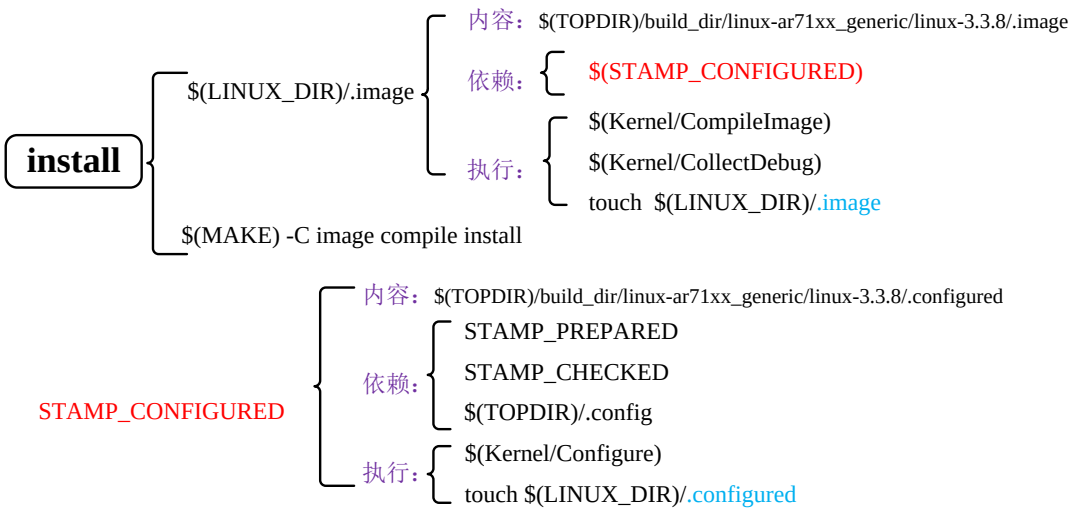


图 4-45 target/linux/ar71xx/install 的目标依赖关系

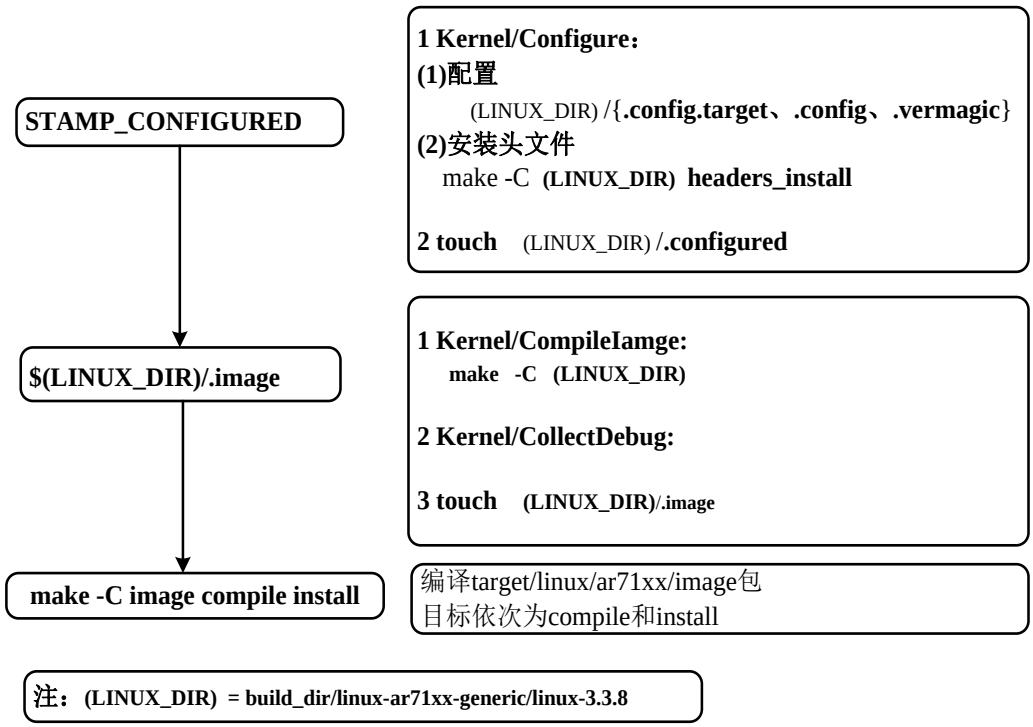


图 4-46 target/linux/ar71xx/install 的编译执行流程

说明：

1、图 4-40 中，红色标识同一目标，浅蓝色文件为中间目标编译完成的标志文件，避免重复编译。

2、根据依赖关系，编译次序: Kernel/Configure→ Kernel/CompileIamge→ Kernel/CollectDebug → make -C image compile install。

3、根据 kernel-built.mk 对函数 Kernel/Configure 的定义，最终会调用 kernel-defaults.mk

中定义的 Kernel/Configure/Default。Kernel/CompileIamge 也类似。

4、图 4-41 为 target/linux/ar71xx/install 的实际编译流程，一般在 target/linux/ar71xx/install 之前已经编译 target/linux/ar71xx/compile，理论上不会重复编译目标 STAMP_CONFIGURED 、STAMP_CHECKED 和 STAMP_PREPARED。实际试验的结果是：STAMP_PREPARED 和 STAMP_CHECKED 没有再次编译，不需要重新更新源文件。但是 STAMP_CONFIGURED 会重新编译，在调用函数 Kernel/Configure 时，没有再次安装头文件，即命令：make -C build_dir/linux-ar71xx_generic/linux-3.3.8 headers_install 没有再次执行。除此之外，STAMP_CONFIGURED 其他操作均重新执行一次。

5、目标$(LINUX_DIR)/.image 中会调用 Kernel/CollectDebug 函数，执行的操作如下：

(1) 复制文件

$(LINUX_DIR)/vmlinux → build_dir/linux-ar71xx_generic/debug/vmlinux

staging_dir/target-mips_r2_uClibc-0.9.33.2/root-ar71xx/lib/modules/3.3.8/{所有文件} → build_dir/linux-ar71xx_generic /debug/modules/{所有文件}

(2) 复制文件夹 debug

build_dir/linux-ar71xx_generic /debug → bin/ar71xx/debug

6、make -C (LINUX_DIR) <TARGET>和 make -C image compile install 和分别见 4.7 和 4.8。

4.7 内核编译

4.7.1 文件准备与编译命令

Linux 内核的编译路径为(LINUX_DIR) = build_dir/linux-ar71xx_generic/linux-3.3.8，Linux 内核的编译是相对独立的过程，openwrt 系统在 target/linux/ar71xx 包 compile 和 install 的过程中，会为内核编译更新源文件、配置.config、输入启动编译命令。在编译 target/linux/ar71xx 过程中，控制 Linux Kernel 编译的流程如下图所示。

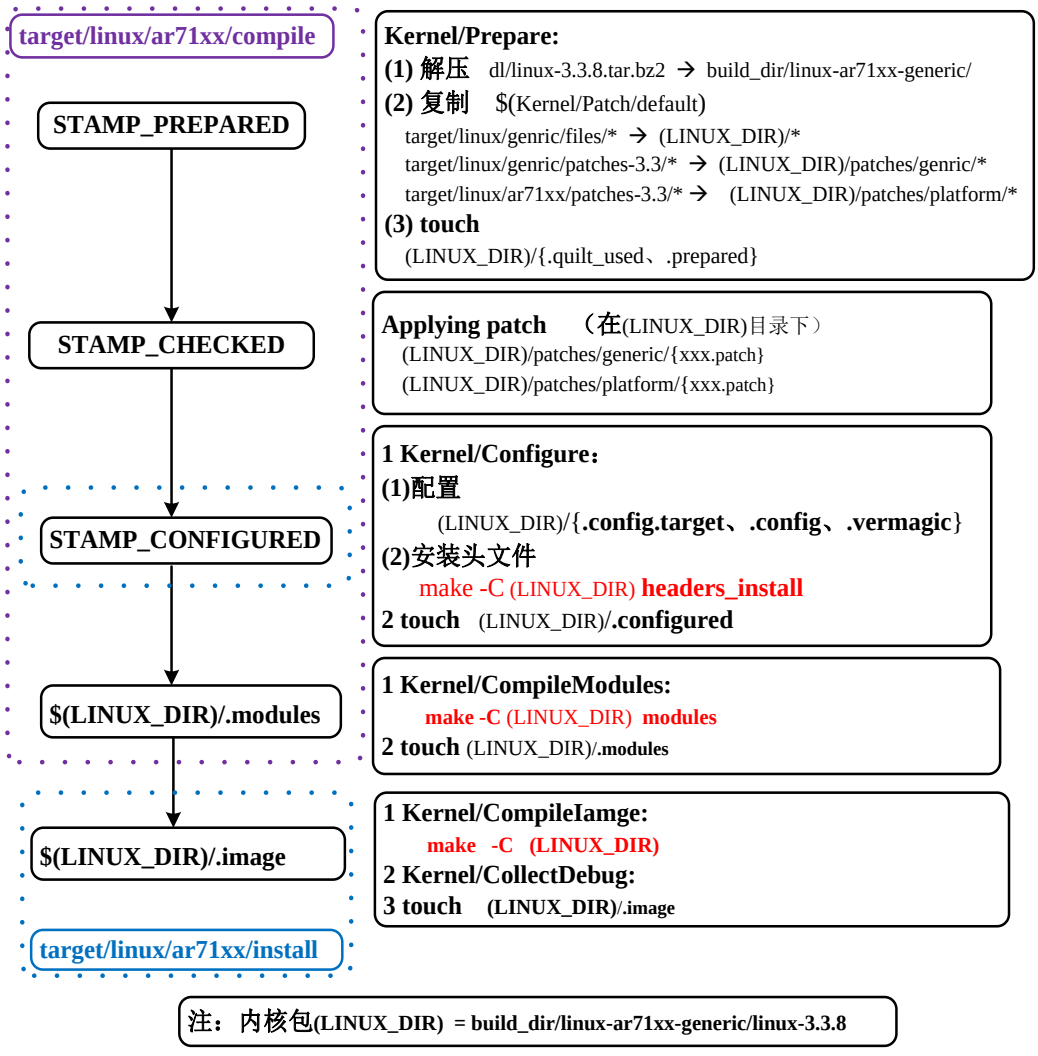


图 4-47 Linux Kernel 编译的控制机制

说明：

1、编译图的右半部分为中间目标对应的与内核编译相关的执行内容。

2、紫色虚线框内为 target/linux/ar71xx/compile 的中间目标，蓝色虚线框内为 target/linux/ar71xx/install 的中间目标。其中，STAMP_CONFIGURED 为 compile 和 install 重复编译目标。实验结果表明，安装头文件(make -C (LINUX_DIR) headers_install)只执行一次，其他操作会重复执行。

3、图中红色文字为启动内核编译的命令，均为切换进入(LINUX_DIR)，编译 Makefile，只是目标不同，分别是 headers_install、modules 和无目标。无目标其实是不指定生成的内核

镜像的名字，镜像名字为 vmlinux。由于内核编译均在(LINUX_DIR)进行，所以 4.7.2 至 4.7.3 小节的文件路径均为省略(LINUX_DIR)的相对路径。

4、编译 target/linux/ar71xx 过程中，控制 Linux Kernel 编译的作用总结如下图所示。

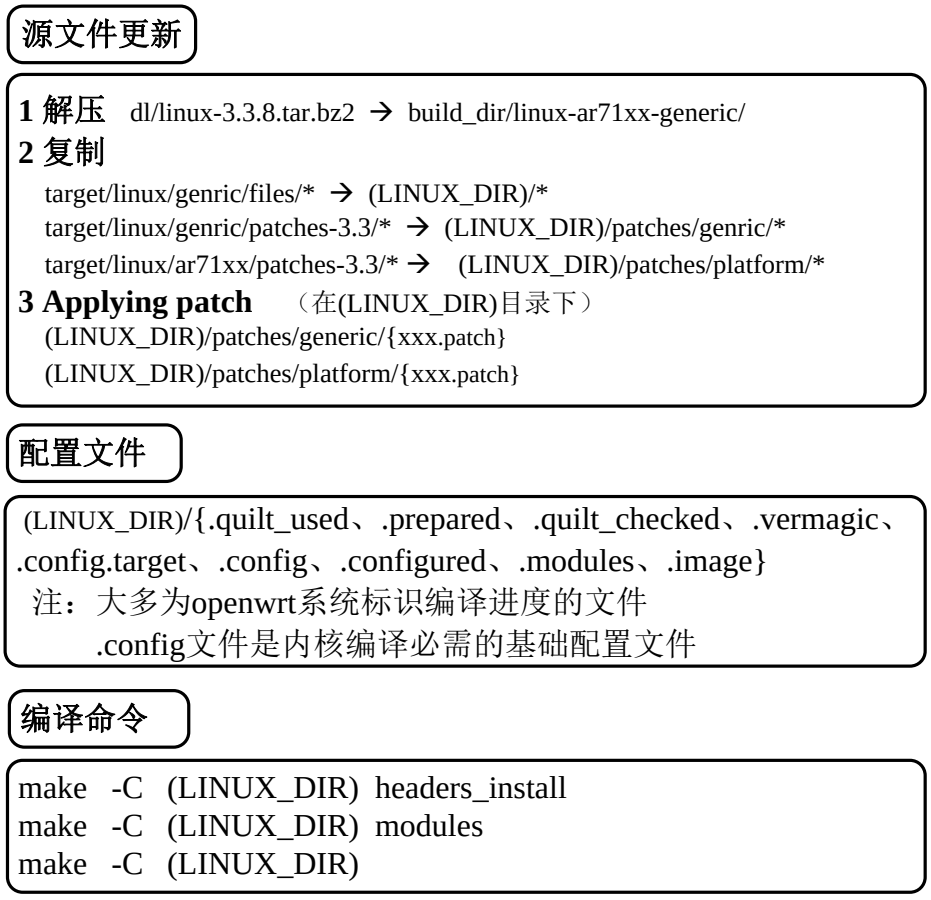


图 4-48 target/linux/ar71xx 启动内核编译的功能总结

4.7.2 headers_install

除非特别说明，否则默认本节的文件在(LINUX_DIR)目录下，文件路径为省略(LINUX_DIR)的相对路径。

(LINUX_DIR) = (TOPDIR)/build_dir/linux-ar71xx_generic/linux-3.3.8

示例：文件：scripts/Makefile.headersinst 路径为：(LINUX_DIR)/ scripts/Makefile.headersinst

(TOPDIR)/include/kernel-default.mk 中函数 Kernel/Configure/Default 关于导出内核头文件的命令(黑体)及其解释如下：

```
[ -d $(LINUX_DIR)/user_headers ] || $(MAKE) $(KERNEL_MAKEOPTS)
INSTALL_HDR_PATH=$(LINUX_DIR)/user_headers headers_install
# 如果$(LINUX_DIR)/user_headers 存在则不需要重复导出头文件
# 当(LINUX_DIR)/user_headers 不存在时，make headers_install 导出内核头文件
make -C build_dir/linux-ar71xx_generic/linux-3.3.8 CROSS_COMPILE="mips-openwrt-linux-uclibc-"
HOSTCFLAGS="-O2 -I staging_dir/host/include -Wall -Wmissing-prototypes -Wstrict-prototypes"
ARCH="mips" KBUILD_HAVE_NLS=no CONFIG_SHELL="/bin/bash" V=' '
```

```
CC="mips-openwrt-linux-uclibc-gcc" INSTALL_HDR_PATH= (LINUX_DIR)/user_headers headers_install
# 进入(LINUX_DIR)目录，执行 Makefile，目标为 headers_install。
```

从命令可以看到，导出内核头文件只需要执行一次，重复编译时不需要重复执行。headers_install 的目标依赖关系及其编译执行流程分别见下面两个图。

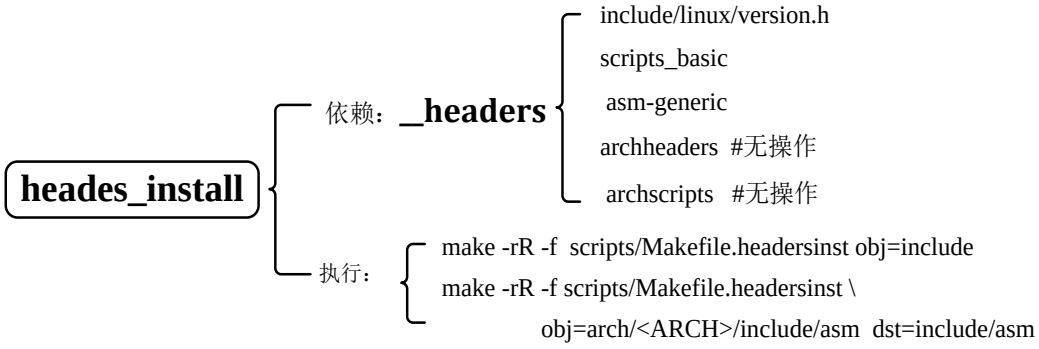


图 4-49 headers_install 的目标依赖关系

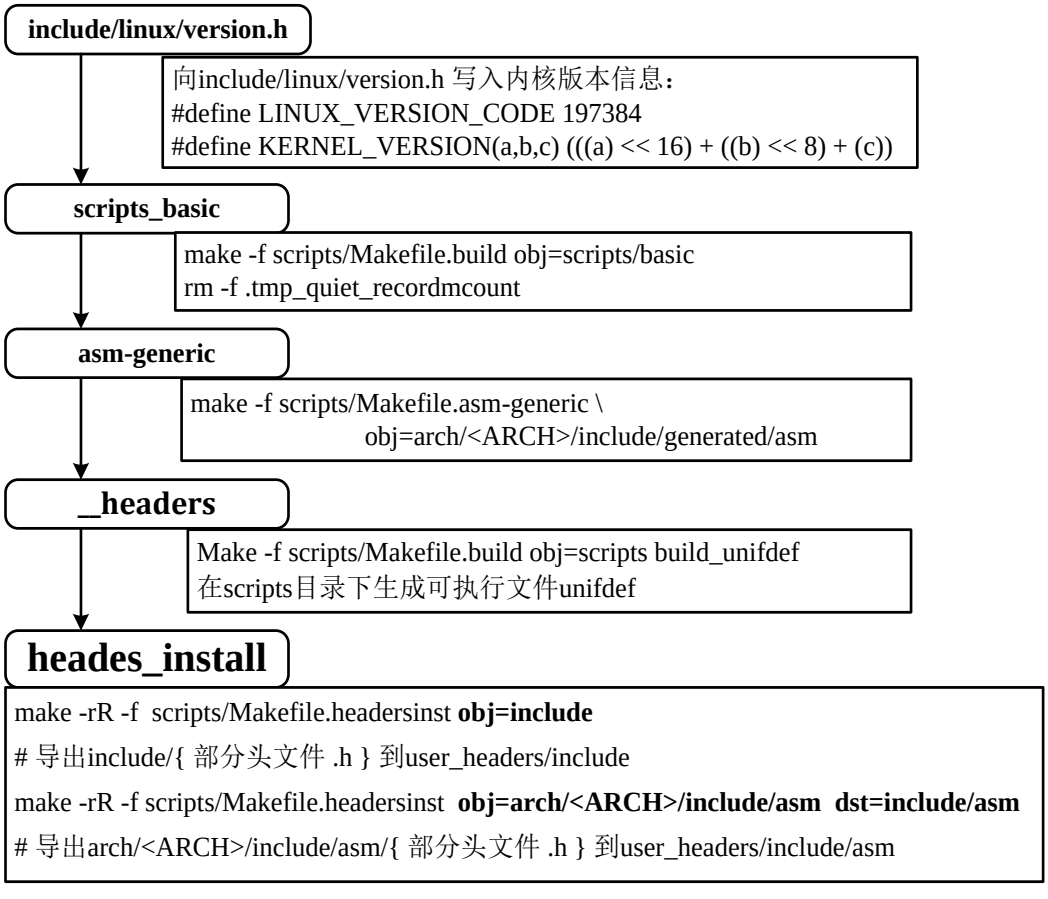


图 4-50 headers_install 的编译执行流程

4.7.3 modules

除非特别说明，否则默认本节的文件在(LINUX_DIR)目录下，文件路径为省略(LINUX_DIR)的相对路径。

(LINUX_DIR) = (TOPDIR)/build_dir/linux-ar71xx_generic/linux-3.3.8

(TOPDIR)/include/kernel-default.mk 中函数 Kernel/CompileModules/Default 的执行代码(黑体)及其注释如下：

```
+$(MAKE) $(KERNEL_MAKEOPTS) $(KERNEL_JFLAG) modules
# make -C (LINUX_DIR) CROSS_COMPILE="mips-openwrt-linux-uclibc-"
HOSTCFLAGS="-O2 -I staging_dir/host/include -Wall -Wmissing-prototypes
-Wstrict-prototypes" ARCH="mips" KBUILD_HAVE_NLS=no
CONFIG_SHELL="/bin/bash" V=" CC="mips-openwrt-linux-uclibc-gcc" -j1 modules
# 进入(LINUX_DIR)目录，执行 Makefile，目标为 modules。
```

make modules 的主要工作是编译内核模块，生成相应模块的可执行文件。modules 的目标依赖关系及其编译执行流程分别见下面两个图。

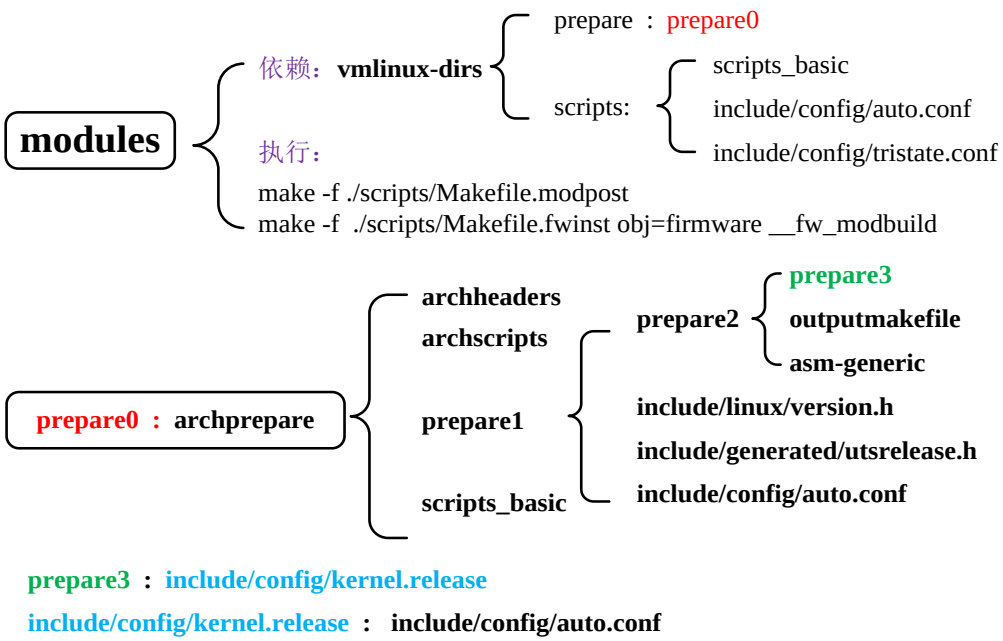


图 4-51 modules 的目标依赖关系

根据目标的依赖关系，依次编译 include/config/auto.conf → … → modules。目标 scripts_basic 和 asm-generic 和编译目标 headers_install 中的执行内容一致，下面就没有重复说明。下面将根据目标的编译次序分别进行简要说明，没有详细研究的部分，只列出执行命令。

(1) .config

编译内核模块以及内核镜像文件均依赖于(LINUX_DIR)/.config 中保存的配置选项。在 4.6.2 节 target 编译的中间目标 STAMP_CONFIGURED 执行函数 Kernel/Configure/Default，会生成(LINUX_DIR)/.config。文件的具体生成过程参考文档《openwrt 系统的 makefile 文件与函数总结》的 6.4.3 小节，下图的上半部分(红色框内) 简要展示(LINUX_DIR)/.config 的生成过程。

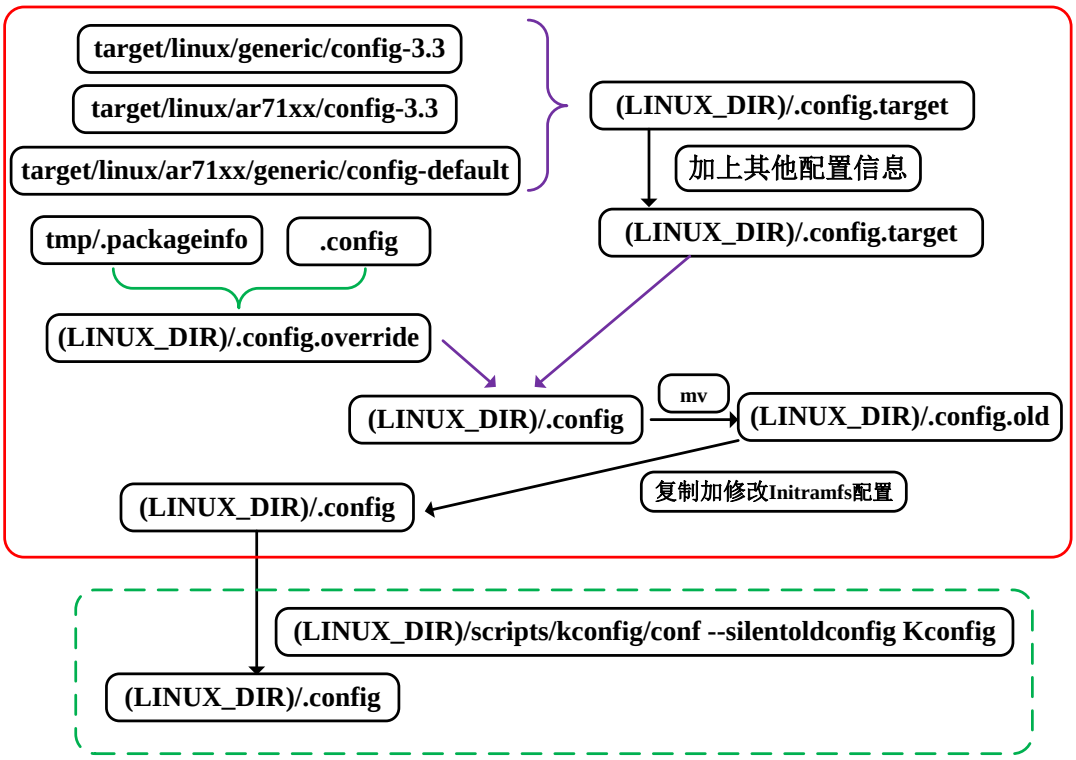


图 4-52 内核.config 的产生过程

(LINUX_DIR)/.config.target 主要保存 target 目标选项(平台、板子型号)的配置信息，(LINUX_DIR)/.config.override 保存内核 package 包(kmod-<pkgName>)的配置信息，将二者综合起来，写入(LINUX_DIR)/.config。在 make modules 的过程中，目标 include/config/auto.conf 会执行语句：

```
scripts/kconfig/conf --silentoldconfig Kconfig
```

按着 Kconfig 定义的内核配置文件格式，结合.config 的配置信息，重新改写.config。目标 include/config/auto.conf 的执行内容见下图。

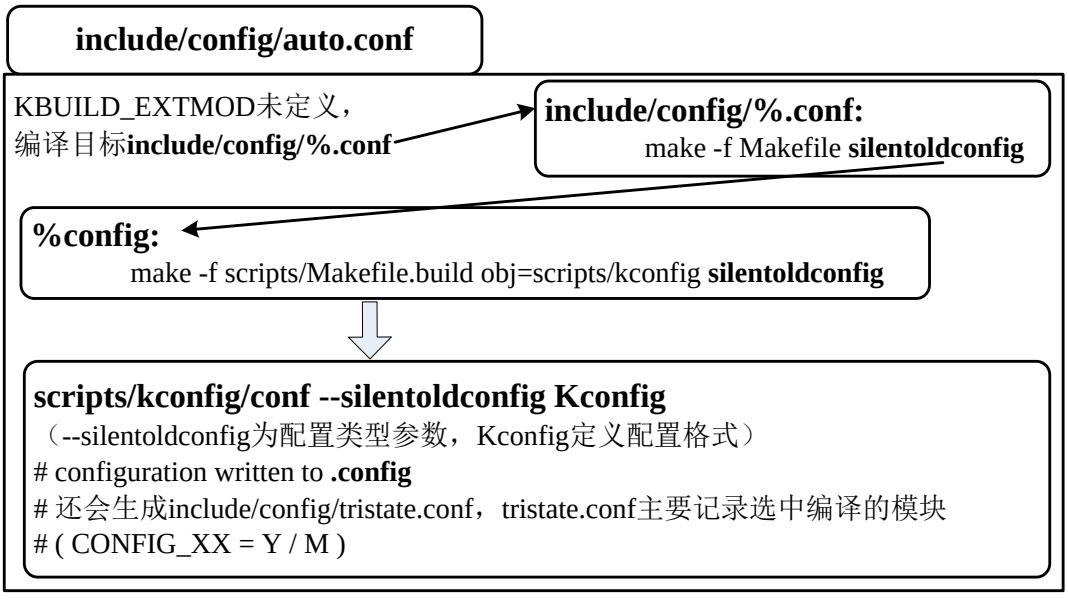


图 4-53 目标 include/config/auto.conf 的执行内容

(2) prepare

目标 prepare 及其依赖目标的执行内容概述如下图所示，只写出命令未做解释的执行语句未进行详尽分析。

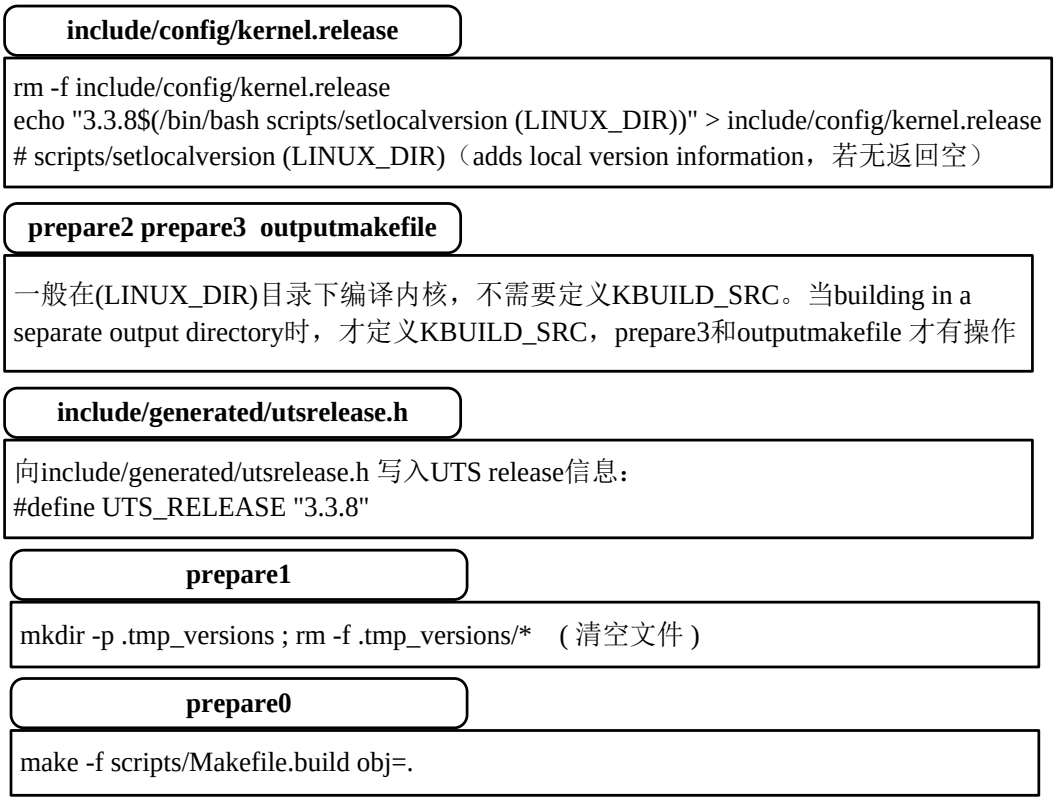


图 4-54 目标 prepare 及其依赖目标的执行内容

(3) scripts

目标 scripts 的执行命令就一条语句：

make -f scripts/Makefile.build obj=scripts

根据 scripts/Makefile 和配置结果，编译 scripts 及其子目录的相关模块，将生成的可执行文件记录在 scripts/modules.order 中

(4) vmlinux-dirs

目标 vmlinux-dirs 的命令语句及其执行内容如下图所示

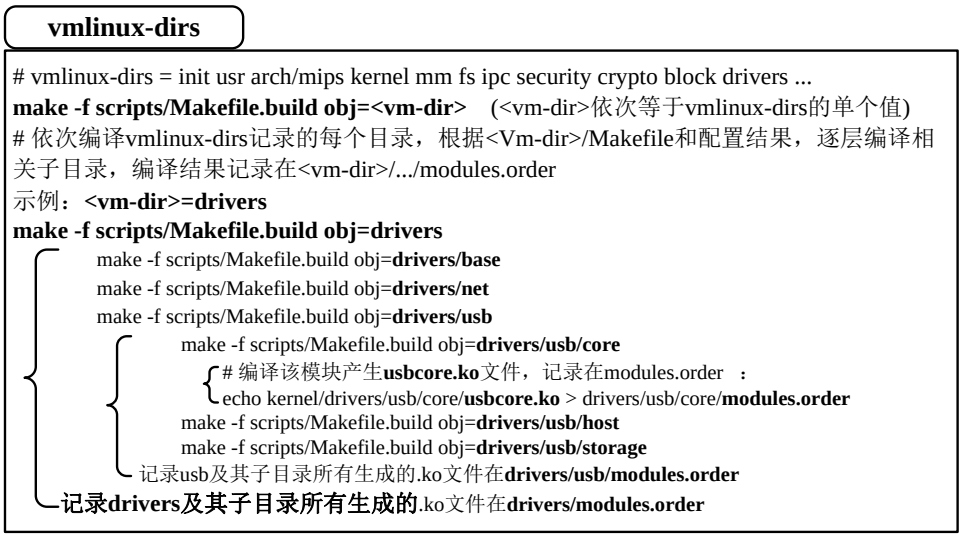


图 4-55 vmlinux-dirs 的编译解释

(5) %/modules.builtin 与 modules.builtin

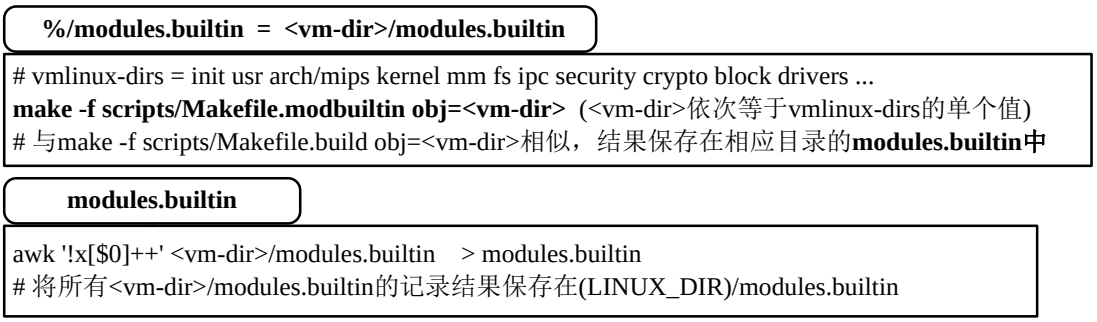


图 4-56 %/modules.builtin 与 modules.builtin 的执行命令

(6) modules

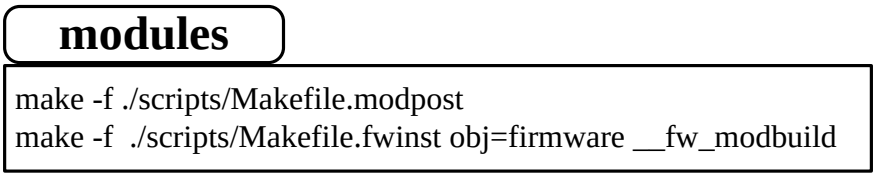


图 4-57 目标 modules 的执行命令

4.7.4 vmlinux

除非特别说明，否则默认本节的文件在(LINUX_DIR)目录下，文件路径为省略(LINUX_DIR)的相对路径。

(LINUX_DIR) = (TOPDIR)/build_dir/linux-ar71xx_generic/linux-3.3.8

(TOPDIR)/include/kernel-default.mk 中函数 Kernel/CompileImage/Default 的执行代码(黑体)及其注释如下：

```
+$(MAKE) $(KERNEL_MAKEOPTS) $(KERNEL_JFLAG) $(subst
",,$(KERNELNAME))
```

```
# make -C (LINUX_DIR) CROSS_COMPILE="mips-openwrt-linux-uclibc-"
HOSTCFLAGS="-O2 -I staging_dir/host/include -Wall -Wmissing-prototypes
-Wstrict-prototypes" ARCH="mips" KBUILD_HAVE_NLS=no
CONFIG_SHELL="/bin/bash" V=" CC="mips-openwrt-linux-uclibc-gcc" -j1
# 进入(LINUX_DIR)目录，执行 Makefile，无目标。
```

编译内核，最终的产物为(LINUX_DIR)/ vmlinux，vmlinux 为纯二进制可执行文件，因为体积比较大，不适合直接下载到开发板运行。一般需要通过压缩等方式，生成体积更小的内核镜像 zImage。

固件(zImage、uImage)可以在内核编译目录 (LINUX_DIR) 下通过相关的命令来完成，如 make bzImage ARCH=mips … 。将 vmlinux 进行压缩，压缩后加上解压代码，生成 zImage，zImage 加上 uboot 所需要的头，生成 uImage。。

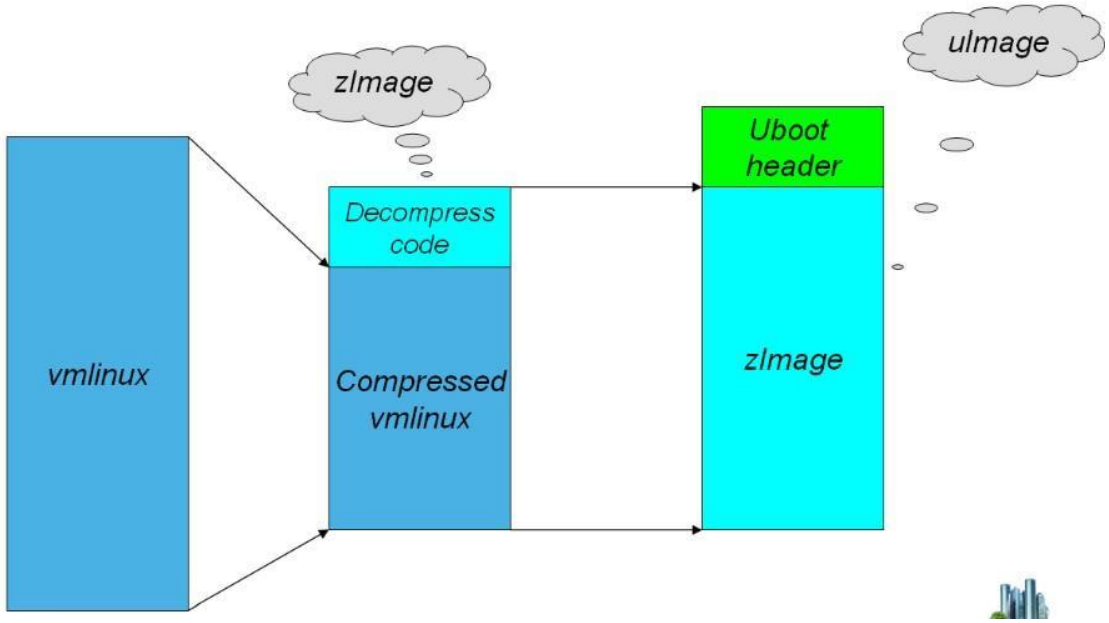


图 4-58 内核文件 uImage 的组成

openwrt 系统中，固件的生成分成两步，第一步编译内核生成 vmlinux，然后压缩成(KERNEL_BUILD_DIR)/{vmlinux 和 vmlinux.elf }。第二步，加上文件系统等，打包生成相应固件 image。image 的生成由 target/linux/ar71xx/image 包的编译过程中调用的函数 Image/BuildKernel 完成，见 4.8 节。

4.8 固件生成

4.8.1 文件函数及其目标关系

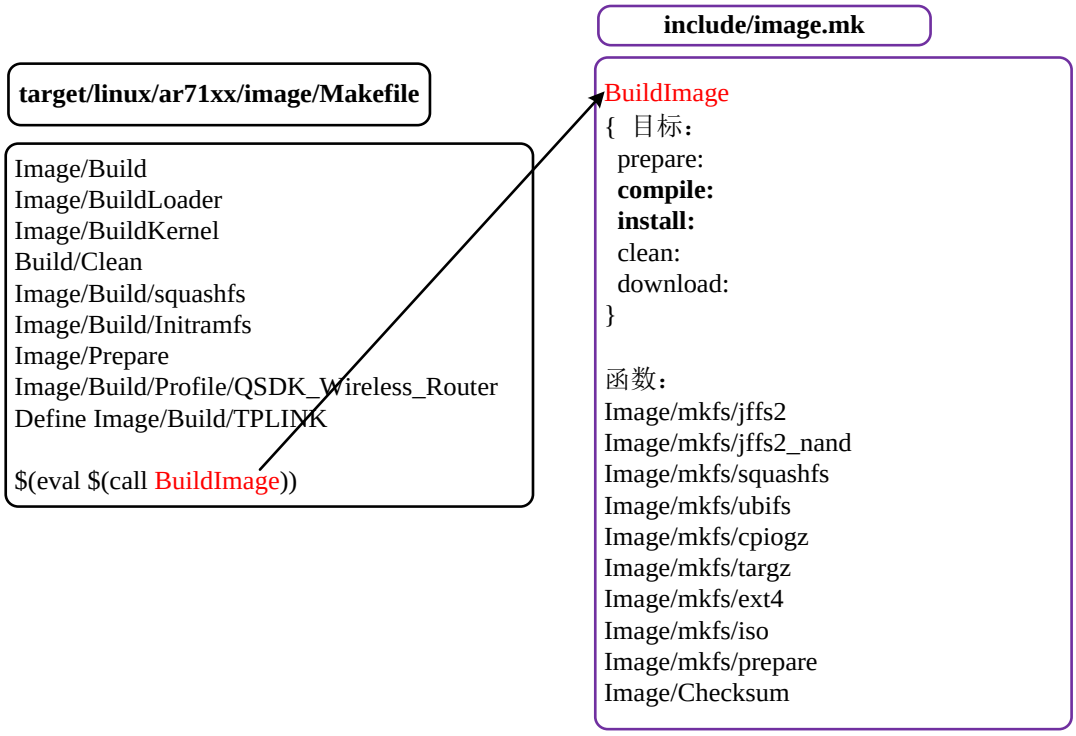


图 4-59 target/linux/ar71xx/image 编译的文件关系

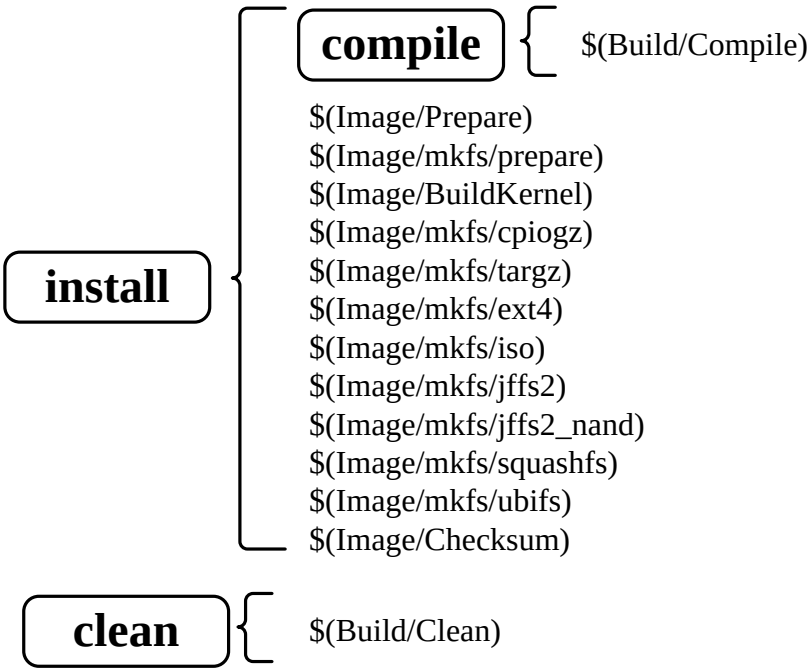


图 4-60 编译 target/linux/ar71xx/image 的目标与函数关系

4.8.2 配置 Target Image

编译 image 之前，需要配置 Target Image。选择不同的根文件系统，编译不

同的固件，配置文件系统的过程如下：(1) 顶层目录输入：make menuconfig；(2) 进入 target image 选择界面，出现如下界面：

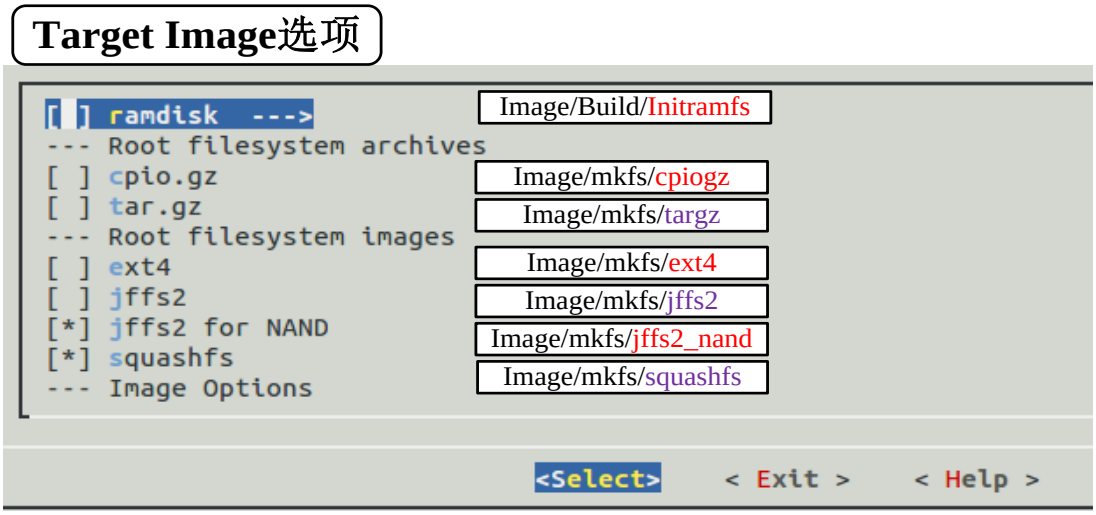


图 4-61 target image 的根文件系统选择界面

(3) 选择合适的文件系统之后，保存退出。需要注意的是 ramdisk 和 Root filesystem images 选项存在互斥，选择了 ramdisk，不能配置 Root filesystem images。

4.8.3 配置 Target Profile

编译 image 前，需配置 Target Profile。因为在编译过程中调用函数 Image/Build，Image/Build 会调用 Image/Build/Profile/$(PROFILE)，(PROFILE)的值即为 Target Profile 的选项。配置 Target Profile 的过程如下：

(1) 顶层目录输入：make menuconfig；

(2) 进入 target Profile 选择界面，出现如下界面：

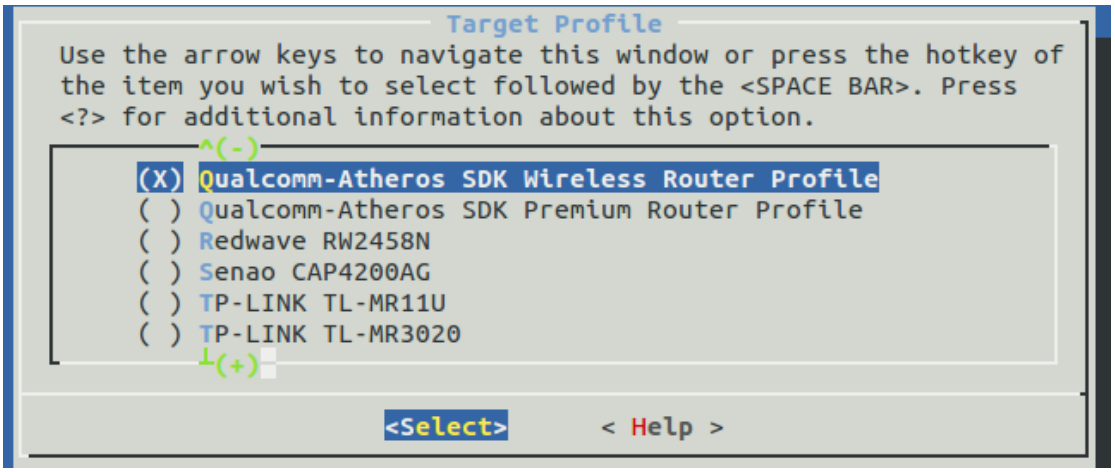


图 4-62 target Profile 的选择界面

Target Profile 选项可以有很多，但每次配置时，只能选择一个，不能同时选择多个。

4.8.4 编译执行过程

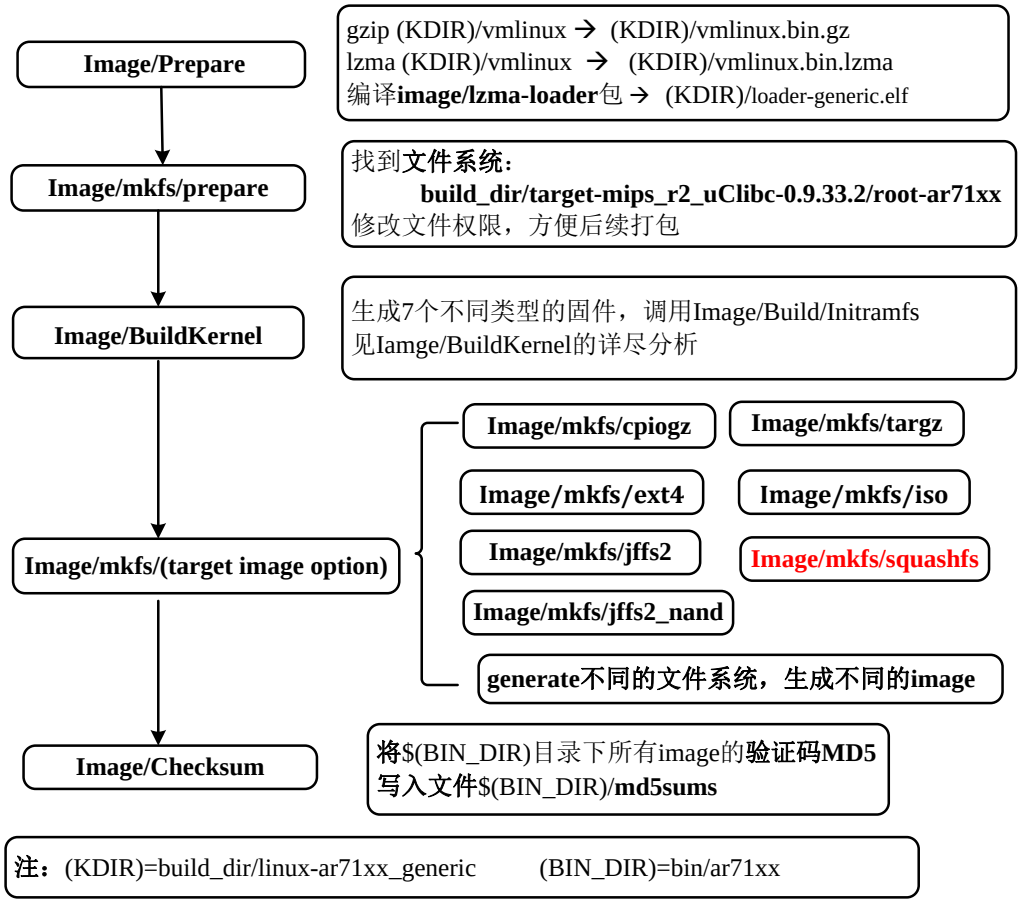


图 4-63 target/linux/ar71xx/image/install 的编译执行过程

说明：

1、Image/Prepare 过程中，编译 image/lzma_loader 包，生成(KDIR)/**loader-generic.elf**，执行过程如下图所示。

编译image/lzma-loader包 $(call image/BuildLoader,generic,elf)
编译目录：(DIR) = build_dir/linux-ar71xx_generic/lzma-loader

1 将target/linux/ar71xx/image/lzma-loader/src/{所有文件} 复制到: (DIR)/{所有文件}
2 切换到DIR，编译源文件产生 head.o loader.o cache.o board.o printf.o LzmaDecode.o
3 (KDIR)/vmlinux.bin.lzma → data.o
4 *.o → loader → loader.bin → loader2.0 → loader.elf
5 cp (DIR)/**loader.elf** → (KDIR)/**loader-generic.elf**

作用：压缩之后的内核文件vmlinux.bin.lzma + 解压代码 生成zImage

图 4-64 target/linux/ar71xx/image/lzma_loader 的编译执行过程

2、配置 target image 时，如果选择了 ramdisk，不能配置 Root filesystem images 选项， 且 Iamge/BuildKernel 会有效调用 Image/Build/Initramfs。如果未选中 ramdisk，Iamge/BuildKernel 调用 Image/Build/Initramfs 不会执行任何操作。

3、Iamge/BuildKernel 的详尽描述见 4.7.5 节。

4.8.5 Iamge/BuildKernel

```
# target/linux/ar71xx/image/Makefile
define Image/BuildKernel
    cp $(KDIR)/vmlinux.elf $(VMLINUX).elf                                    1
    cp $(KDIR)/vmlinux $(VMLINUX).bin                                        2
    dd if=$(KDIR)/vmlinux.bin.lzma of=$(VMLINUX).lzma bs=65536 conv=sync     3
    dd if=$(KDIR)/vmlinux.bin.gz of=$(VMLINUX).gz bs=65536 conv=sync         4
    $(call MkuImage,gzip,,$(KDIR)/vmlinux.bin.gz,$(UIMAGE)-gzip.bin)        5
    $(call MkuImage,lzma,,$(KDIR)/vmlinux.bin.lzma,$(UIMAGE)-lzma.bin)      6
    cp $(KDIR)/loader-generic.elf $(VMLINUX)-lzma.elf                        7
    -mkdir -p $(KDIR_TMP)
    $(call Image/Build/Initramfs)                                            8
endef
```

Image/BuildKernel

1: target image 未选中ramdisk

1	**cp**	$(KDIR)/vmlinux.elf	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux.elf**
2	**cp**	$(KDIR)/vmlinux	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux.bin**
3	**dd**	$(KDIR)/vmlinux.bin.lzma	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux.lzma**
4	**dd**	$(KDIR)/vmlinux.bin.gz	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux.gz**
5	**mkimage**	$(KDIR)/vmlinux.bin.gz	→ $(BIN_DIR)/openwrt-ar71xx-generic-**uImage-gzip.bin**
6	**mkimage**	$(KDIR)/vmlinux.bin.lzma	→ $(BIN_DIR)/openwrt-ar71xx-generic-**uImage-lzma.bin**
7	**cp**	$(KDIR)/loader-generic.elf	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux-lzma.elf**

8 调用：Image/Build/Initramfs　无效调用，不会生成更多固件

2: target image 选中ramdisk CONFIG_TARGET_ROOTFS_INITRAMFS = y

1	**cp**	$(KDIR)/vmlinux.elf	→ $(BIN_DIR)/**openwrt-ar71xx-generic-vmlinux-initramfs.elf**
2	**cp**	$(KDIR)/vmlinux	→ $(BIN_DIR)/**openwrt-ar71xx-generic-vmlinux-initramfs.bin**
3	**dd**	$(KDIR)/vmlinux.bin.lzma	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux-initramfs.lzma**
4	**dd**	$(KDIR)/vmlinux.bin.gz	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux-initramfs.gz**
5	**mkimage**	$(KDIR)/vmlinux.bin.gz	→ $(BIN_DIR)/**openwrt-ar71xx-generic-uImage-initramfs-gzip.bin**
6	**mkimage**	$(KDIR)/vmlinux.bin.lzma	→ $(BIN_DIR)/**openwrt-ar71xx-generic-uImage-initramfs-lzma.bin**
7	**cp**	$(KDIR)/loader-generic.elf	→ $(BIN_DIR)/openwrt-ar71xx-generic-**vmlinux-initramfs-lzma.elf**

8 调用：Image/Build/Initramfs　有效调用，生成：
$(BIN_DIR)/Openwrt-ar71xx-generic-<Target Profile 选项>-**initramfs-uImage.bin**

注：(KDIR)=build_dir/linux-ar71xx_generic　　(BIN_DIR)=bin/ar71xx

图 4-65 Iamge/BuildKernel 的详尽描述

说明：

1、Iamge/BuildKernel 执行语句 7：cp $(KDIR)/loader-generic.elf $(VMLINUX)-lzma.elf，loader-generic.elf 为 zImage(内核压缩文件+解压代码)，zImage 是固件 uImage 的核心部分。

2、配置 target image 时，选择 ramdisk，即 CONFIG_TARGET_ROOTFS_INITRAMFS=y。Iamge/BuildKernel 生成的内核固件的名称中多了 initramfs，此配置下，生成的固件大小比没有选中 ramdisk 时大几倍。所以一般情况下，不会选中 ramdisk。

3、Iamge/BuildKernel 用到的固件制作/文件拷贝的命令解释：

cp ：对文件进行操作

dd ：对块进行操作，常见语句：

dd if=vmlinux.bin.lzma of=$(VMLINUX).lzma bs=65536 conv=sync

if 代表输入文件，of 代表输出文件，bs 以 65536k 为单位读取/写入数据，conv=sync 表示如果输入数据不足 65536k，后面填充 NULL。

mkimage：uboot 提供的工具，主要是向 vmlinux 添加 64 字节头部，生成的 uImage 结构为：

头部信息（64Byte）	kernel image长度（4Byte）	data image长度（4Byte）	NULL（4Byte）	kernel image	data image

由于没有 data image 的数据部分，生成的 uImage 结构为：

头部信息（64Byte）	kernel image长度（4Byte）	NULL（4Byte）	kernel image

Iamge/BuildKernel 的作用是将内核文件 vmlinux 进行压缩，打包成内核固件。其示意图如下图所示。

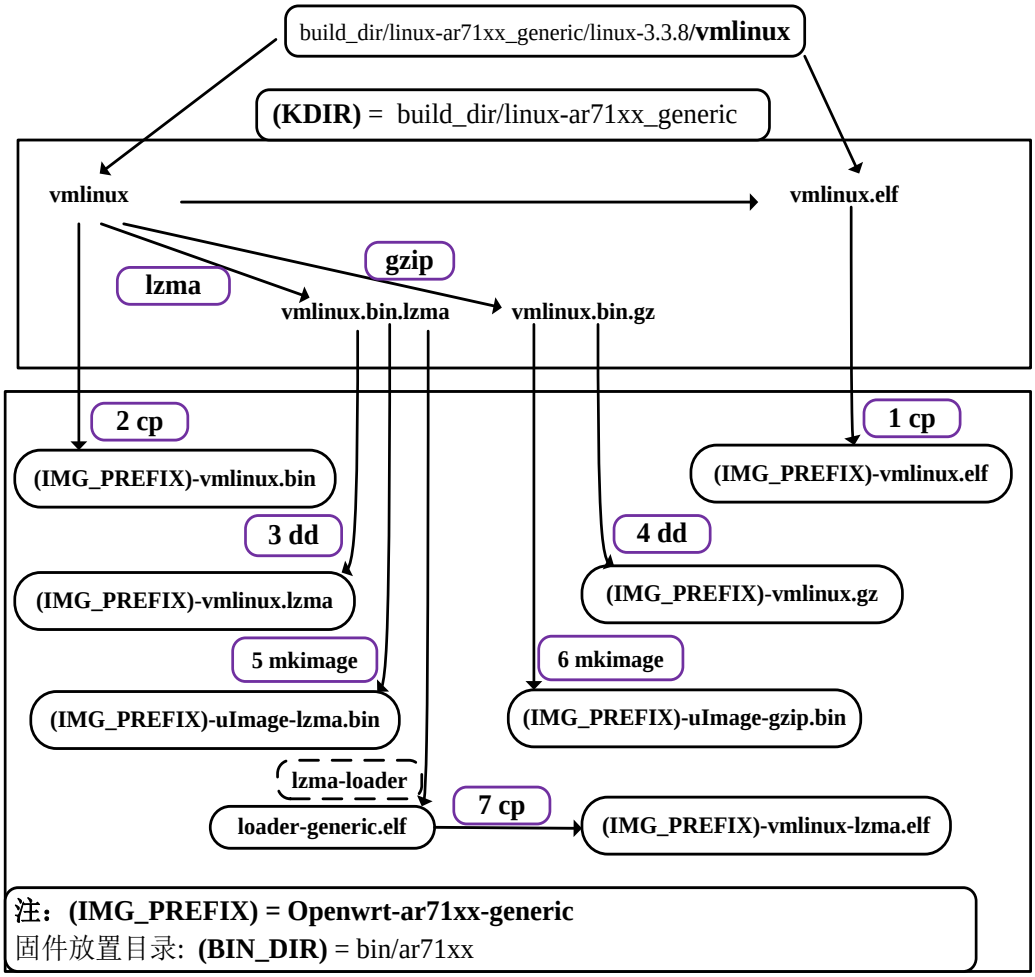


图 4-66 内核 vmlinux 的固件生成过程

说明：

不同的压缩方式可以得到不同的 uImage 文件：
openwrt-ar71xx-generic-uImage-gzip.bin 和
openwrt-ar71xx-generic-uImage-lzma.bin。

4.8.6 文件系统

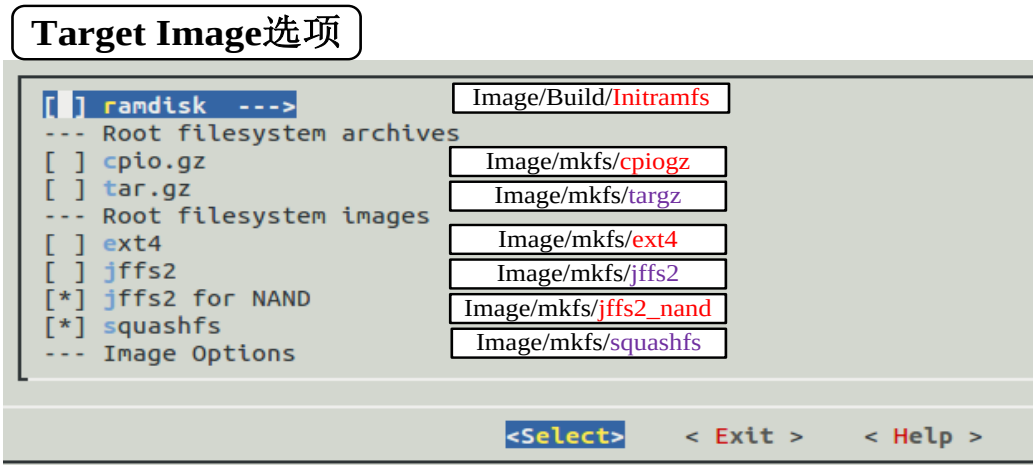


图4-67 Target Image的配置界面

配置Target Image时，ramdisk和Root filesystem images选项存在互斥，选择了ramdisk，不能配置Root filesystem images。Root filesystem archives为文件系统存档选项，将文件系统直接进行压缩。Root filesystem images用于生成包含内核文件和文件系统的固件。

(1) ramdisk

选择ramdisk，不会生成文件系统固件，只会根据Target Profile的配置，将内核文件压缩成uImage。

ramdisk示例：Target Profile选择QSDK_Wireless_Router

Iamge/Build/Initramfs会调用Image/Build/Profile/QSDK_Wireless_Router，Image/Build/Profile/QSDK_Wireless_Router调用Image/Build/Profile/AP135，参数为initramfs。执行过程简述如下：

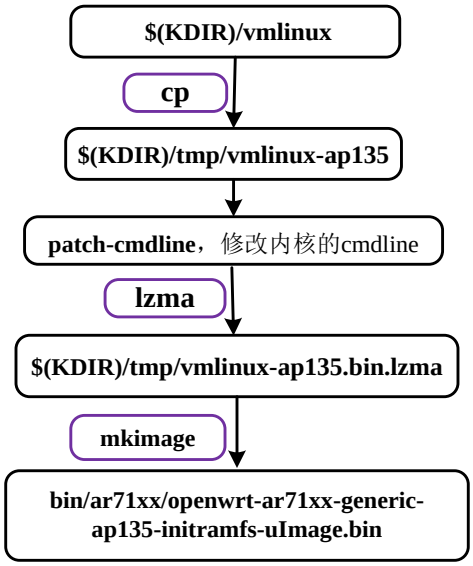


图4-68 Iamge/Build/Initramfs的执行过程

(2) Root filesystem archives

Root filesystem archives为独立可选部分，包含cpio.gz和tar.gz，分别对应Image/mkfs/cpiogz和Image/mkfs/targz。Image/mkfs/cpiogz和Image/mkfs/targz的主要操作是将文件系统直接进行压缩。

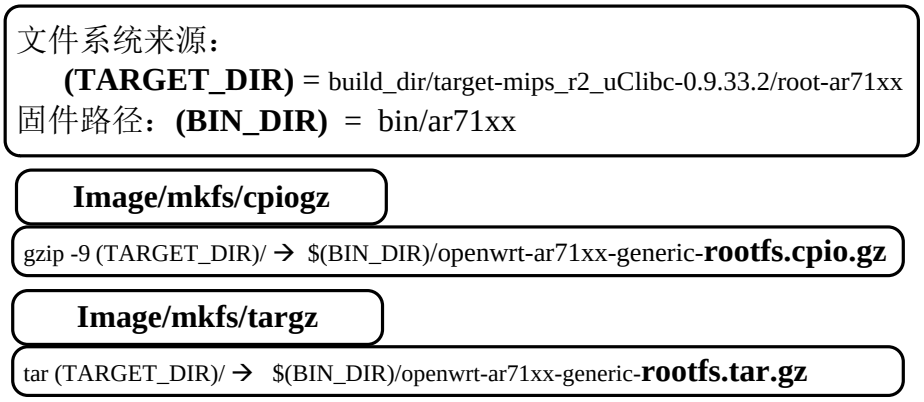


图4-69 Root filesystem archives选项对应的文件系统固件

(3) Root filesystem images

Root filesystem images与Root filesystem archives不同，他的可选项(如squashfs)对应的操作是：(1) 将文件系统(TARGET_DIR)制作成 (KDIR)/root.<选项>；(2) 制作文件系统固件；(3) 制作内核+文件系统的uImage。Root filesystem images选项的通用执行过程总结如下图所示。

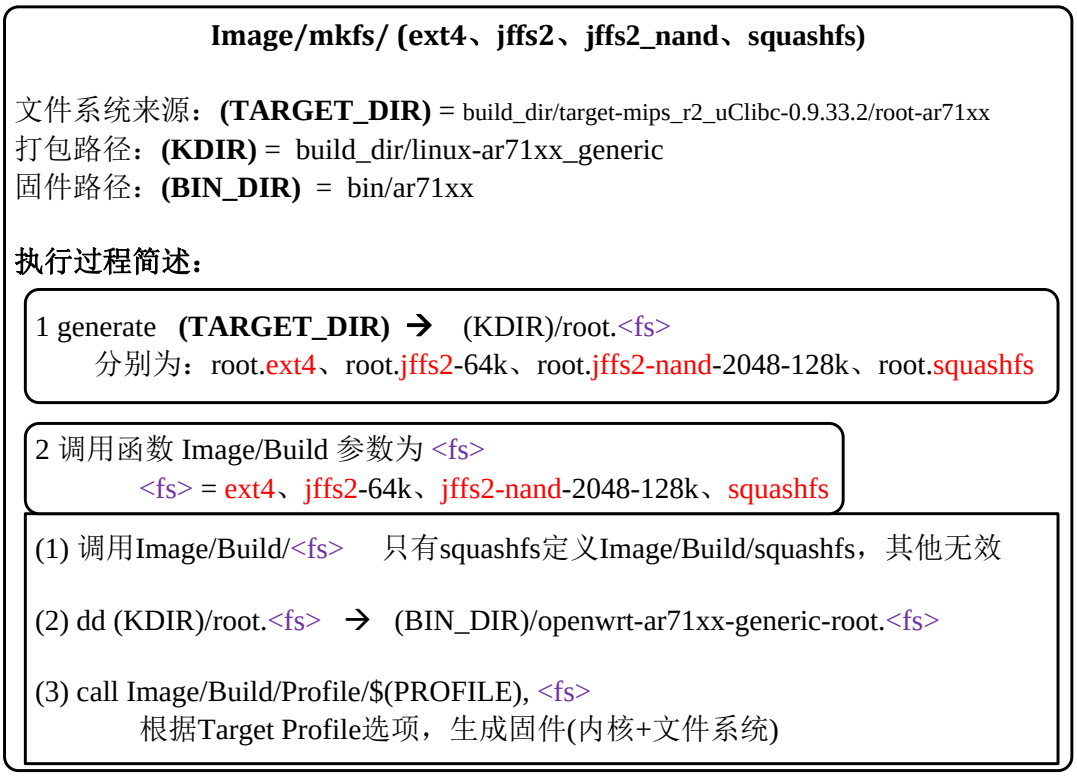


图 4-70 Root filesystem images 选项的通用执行过程

根据Target Profile配置，打包内核+文件系统制作uImage，将在4.7.6节进行阐述。选择不同的Root filesystem images选项，编译得到文件系统image的名称如下图所示。

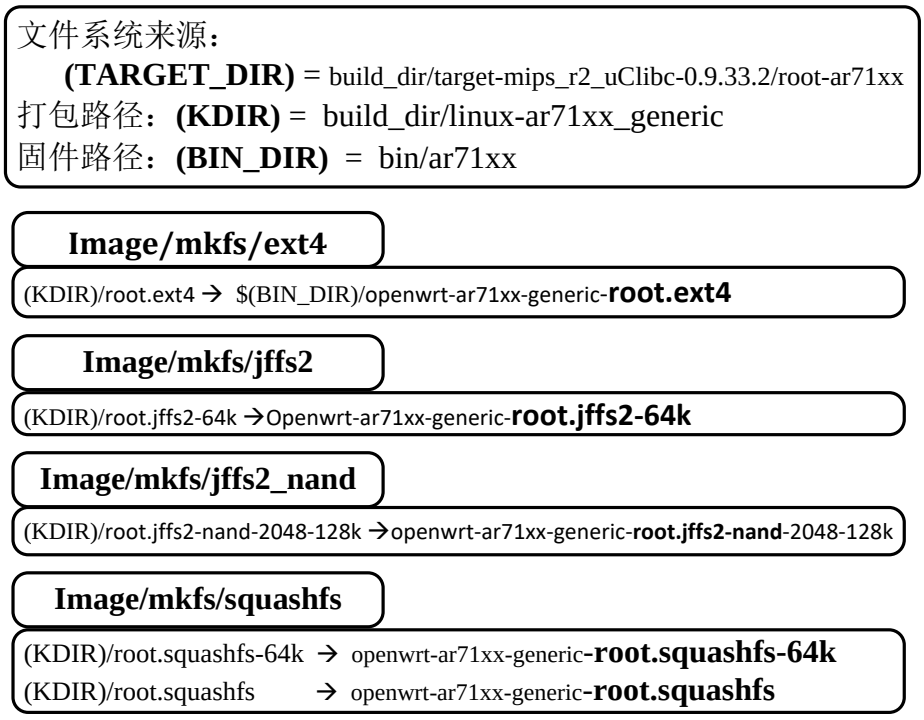


图 4-71 Root filesystem images 选项对应的文件系统固件

4.8.7 squashfs 示例分析

Target Iamge选择squashfs，Target Profile选择QSDK_Wireless_Router，即(PROFILE) = QSDK_Wireless_Router。Image/mkfs/squashfs调用的函数关系、函数对文件的操作内容、文件的流转过程分别见下面三个图。

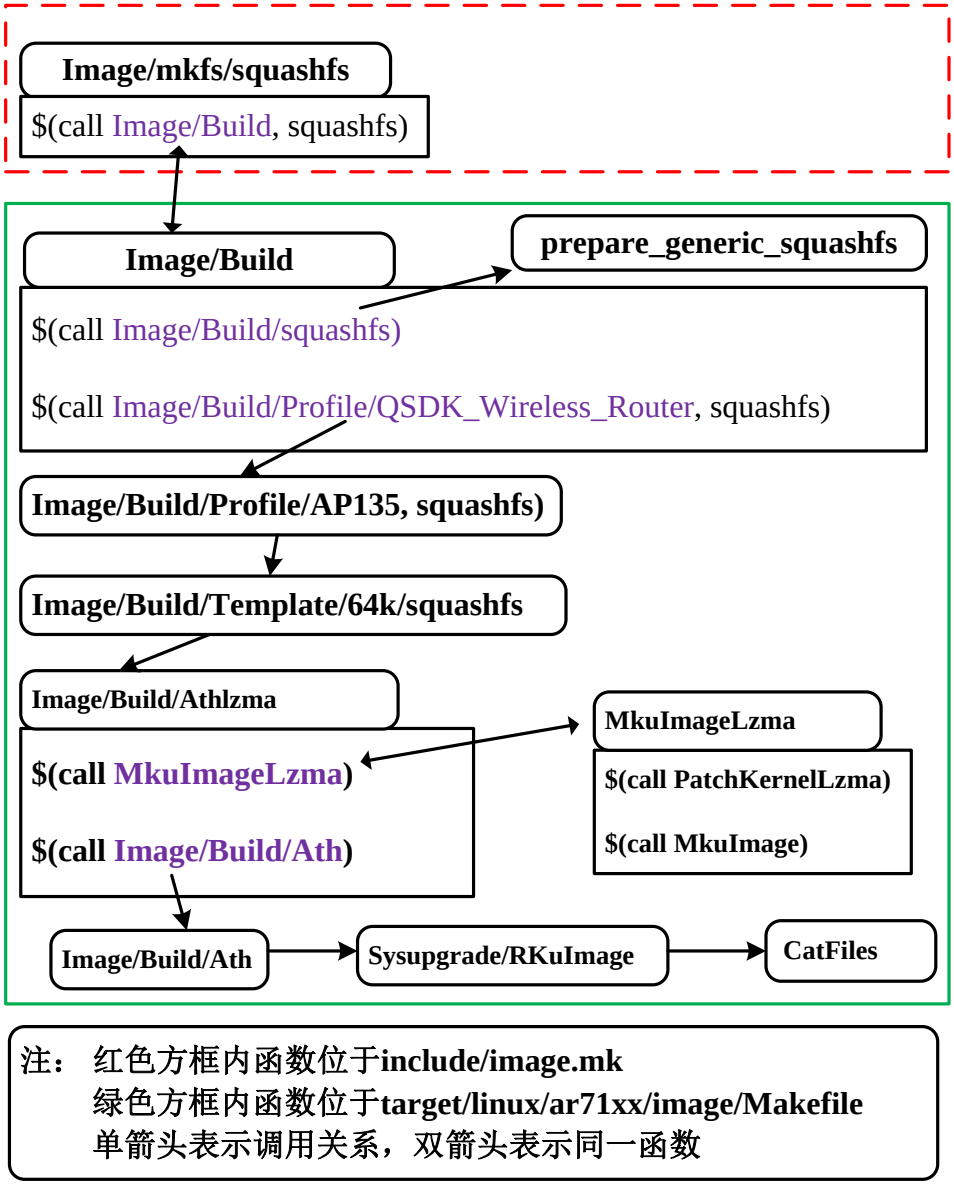


图 4-72 Image/mkfs/squashfs 调用的函数关系

Image/mkfs/squashfs

mksquashfs4 (TARGET_DIR) → (KDIR)/**root.squashfs**

Image/Build

生成文件系统固件：
dd (KDIR)/root.squashfs → (BIN_DIR)/(IMG_PREFIX)-**root.squashfs**

Image/Build/squashfs

1 **cp** $(KDIR)/root.squashfs → $(KDIR)/**root.squashfs-raw**
2 **cp** $(KDIR)/root.squashfs → $(KDIR)/**root.squashfs-64k**
3 **padjffs2 $(KDIR)/root.squashfs-64k 64** 填充jffs2分区 使最后一个块变成64k
4 **cp** $(KDIR)/root.squashfs-64k → (BIN_DIR) /(IMG_PREFIX)-**root.squashfs-64k**

prepare_generic_squashfs

padjffs2 $(KDIR)/root.squashfs 4 8 64 128 256 填充jffs2分区

PatchKernelLzma

1 **cp** (KDIR)/**vmlinux** → (KDIR)/tmp/**vmlinux-ap135**
2 **patch-cmdline** (KDIR)/tmp/vmlinux-ap135 # 修改内核的cmdline
3 **lzma** (KDIR)/tmp/vmlinux-ap135 → (KDIR)/tmp/**vmlinux-ap135.bin.lzma**

MkuImage,lzma

mkimage (KDIR)/tmp/vmlinux-ap135.bin.lzma → (KDIR)/tmp/**vmlinux-ap135.uImage**

CatFiles

生成文件系统+内核uImage的固件
dd (KDIR)/**root.squashfs-64k** + (KDIR)/tmp/**vmlinux-ap135.uImage**)
> (BIN_DIR)/openwrt-ar71xx-generic-**ap135-squashfs-sysupgrade.bin**

Image/Build/Ath

1 **dd** (KDIR)/tmp/vmlinux-ap135.uImage → (BIN_DIR)/(IMG_PREFIX)-**ap135-kernel.bin**
2 **dd** (KDIR)/root.squashfs-64k → (BIN_DIR)/(IMG_PREFIX)-ap135-**rootfs-squashfs.bin**

注：(IMG_PREFIX) = openwrt-ar71xx-generic
目录: **(BIN_DIR)** = bin/ar71xx **(KDIR)** = build_dir/linux-ar71xx_generic
(TARGET_DIR) = build_dir/target-mips_r2_uClibc-0.9.33.2/root-ar71xx

图 4-73 Image/mkfs/squashfs 相关函数对文件的操作内容

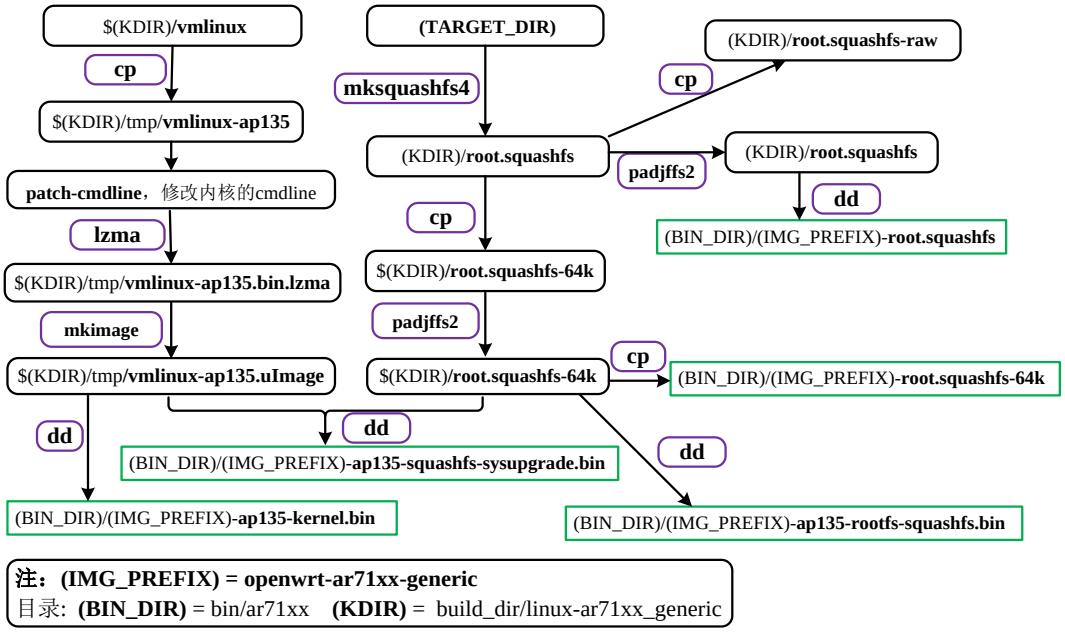


图 4-74 Image/mkfs/squashfs 的文件流转过程

说明：

1、图中紫色框内为操作命令，cp 为文件拷贝命令，dd 为文件块拷贝命令，mksquashfs4 为 squashfs 文件系统制作指令，lzma 为文件压缩命令。mkimage 为 uboot 提供的工具，在内核文件之前添加 64 字节头部，生成 uImage。

2、padjffs2 对文件系统填充 jffs2 分区，以正好填满一个 flash 块。当参数为 64 时，将在文件后面填充数据（每个字节都是 0xff），使得最后一个块变成 64k，然后再填充 4 字节的 jffs2 结束符(dead c0de)。

3、绿色框内的文件为生成的固件。

4、文件系统和内核镜像打包生成固件的命令为：

```
dd if= build_dir/linux-ar71xx_generic/root.squashfs-64k bs=14876672 conv=sync;
dd if= build_dir/linux-ar71xx_generic/tmp/vmlinux-ap135.uImage ) >
bin/ar71xx/openwrt-ar71xx-generic-ap135-squashfs-sysupgrade.bin
```

bs=14876672 表示以 14876672 字节读取/写入数据，conv=sync 表示如果输入数据不足 14876672 字节，后面填充 NULL (全 0)。

5、openwrt-ar71xx-generic-ap135-squashfs-sysupgrade.bin 的存储结构如下：

<table>
<tr><td colspan="2">14876672字节(14.18M)</td><td>kernel uImage (1.04M)</td></tr>
<tr><td>root.squashfs-64k</td><td>NULL</td><td>vmlinux-ap135.uImage</td></tr>
</table>