积小流

openwrt: Makefile 框架分析

本篇的主要目的是想通过分析Makefile，了解openwrt编译过程。着重关注以下几点：

1. openwrt目录结构
2. 主Makefile的解析过程，各子目录的目标生成。
3. kernel编译过程
4. firmware的生成过程
5. 软件包的编译过程

openwrt目录结构

官方源下载速度太度，我从github上clone了openwrt的代码仓库。

```
git clone https://github.com/openwrt-mirror/openwrt.git
```

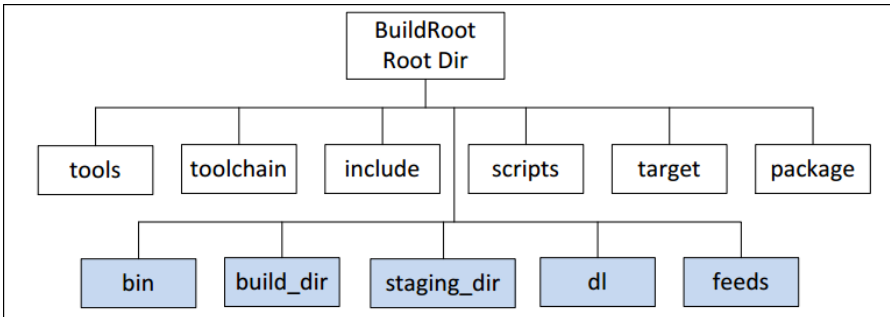

Figure 1. OpenWrt Buildroot Source Tree (The folders in the second lines are generated during compilation).

上图是openwrt目录结构，其中第一行是原始目录，第二行是编译过程中生成的目录。各目录的作用是：

- tools - 编译时需要一些工具， tools里包含了获取和编译这些工具的命令。里面是一些Makefile，有的可能还有patch。每个Makefile里都有一句 `$(eval $(call HostBuild))` ，表示编译这个工具是为了在主机上使用的。
- toolchain - 包含一些命令去获取kernel headers, C library, bin-utils, compiler, debugger
- target - 各平台在这个目录里定义了firmware和kernel的编译过程。
- package - 包含针对各个软件包的Makefile。openwrt定义了一套Makefile模板，各软件参照这个模板定义了自己的信息，如软件包的版本、下载地址、编译方式、安装地址等。
- include - openwrt的Makefile都存放在这里。
- scripts - 一些perl脚本，用于软件包管理。
- dl - 软件包下载后都放到这个目录里
- build_dir - 软件包都解压到build_dir/里，然后在此编译
- staging_dir - 最终安装目录。tools, toolchain被安装到这里，rootfs也会放到这里。
- feeds -

- bin - 编译完成之后，firmware和各ipk会放到此目录下。

OpenWrt Development Guide

main Makefile

openwrt根目录下的Makefile是执行make命令时的入口。从这里开始分析。

```
world:

ifndef ($(OPENWRT_BUILD),1)
  # 第一个逻辑
  ...
else
  # 第二个逻辑
  ...
endif
```

上面这段是主Makefile的结构，可以得知：

1. 执行make时，若无任何目标指定，则默认目标是world
2. 执行make时，无参数指定，则会进入第一个逻辑。如果执行命令 `make OPENWRT_BUILD=1` ，则直接进入第二个逻辑。

编译时一般直接使用 `make V=s -j5` 这样的命令，不会指定OPENWRT_BUILD变量

第一个逻辑

```
  override OPENWRT_BUILD=1
  export OPENWRT_BUILD
```

更改了OPENWRT_BUILD变量的值。这里起到的作用是下次执行make时，会进入到第二逻辑中。

toplevel.mk中的 %:: 解释world目标的规则。

```
prereq:: prepare-tmpinfo .config
	@+$(MAKE) -r -s tmp/.prereq-build $(PREP_MK)
	@+$(NO_TRACE_MAKE) -r -s $@

%::
	@+$(PREP_MK) $(NO_TRACE_MAKE) -r -s prereq
	@( \
	    cp .config tmp/.config; \
	    ./scripts/config/conf --defconfig=tmp/.config -w tmp/.config Config.in > /dev
	    if ./scripts/kconfig.pl '>' .config tmp/.config | grep -q CONFIG; then \
	        printf "$(_R)WARNING: your configuration is out of sync. Please run make
	    fi \
	)
	@+$(ULIMIT_FIX) $(SUBMAKE) -r $@
```

执行 `make V=s` 时，上面这段规则简化为：

```
prereq:: prepare-tmpinfo .config
	@make -r -s tmp/.prereq-build
	@make V=ss -r -s prereq

%::
	@make V=s -r -s prereq
	@make -w -r world
```

可见其中最终又执行了prereq和world目标，这两个目标都会进入到第二逻辑中。

第二逻辑

首先就引入了target, package, tools, toolchain这四个关键目录里的Makefile文件

```
  include target/Makefile
  include package/Makefile
  include tools/Makefile
  include toolchain/Makefile
```

这些子目录里的Makefile使用include/subdir.mk里定义的两个函数来动态生成规则，这两个函数是subdir和stampfile

stampfile

拿target/Makefile举例：

$(eval $(call stamp_file,(curdir),target,prereq,.config))

会生成规则：

```
target/stamp-prereq:=$(STAGING_DIR)/stamp/.target_prereq

$$(target/stamp-prereq): $(TMP_DIR)/.build .config
  @+$(SCRIPT_DIR)/timestamp.pl -n $$(target/stamp-prereq) target .config || \
      make $$(target/flags-prereq) target/prereq
  @mkdir -p $$$$(dirname $$(target/stamp-prereq))
  @touch $$(target/stamp-prereq)

$$(if $(call debug,target,v),,.SILENT: $$(target/stamp-prereq))

.PRECIOUS: $$(target/stamp-prereq) # work around a make bug

target//clean:=target/stamp-prereq/clean
target/stamp-prereq/clean: FORCE
  @rm -f $$(target/stamp-prereq)
```

所以可以简单的看作： $(eval $(call stamp_file,(curdir),target,prereq,.config)) 生成了目标 $(target/stamp-prereq)

- 对于target分别生成了：$(target/stamp - preq)$，(target/stamp-copile)， $(target/stamp-install)
- toolchain : $(toolchain/stamp-install)
- package : $(package/stamp - preq)$,(package/stamp-cleanup), $(package/stamp - compile)$,(package/stamp-install)
- tools : $(tools/stamp-install)

OpenWrt的主Makefile工作过程

subdir

subdir这个函数写了一大堆东西，看起来很复杂 。

$(call subdir, target) 会遍历下的子目录，执行 make -C 操作。这样就切入子目录中去了。

目录变量

几个重要的目录路径：

- KERNEL_BUILD_DIR

  build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/linux-3.14.18

- LINUX_DIR

  build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/linux-3.14.18

- KDIR

  build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a

- BIN_DIR

  bin/ramips
  Makefile中包含了rules.mk, target.mk等.mk文件，这些文件中定义了许多变量，有些是路径相关的，有些是软件相关的。这些变量在整个Makefile工程中经常被用到，

- TARGET_ROOTFS_DIR

  build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2

- BUILD_DIR

  build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2

- STAGING_DIR_HOST

2013年4月 (1)
2013年3月 (7)
2013年2月 (2)
2013年1月 (2)
2012年12月 (1)
2012年11月 (5)
2012年10月 (4)
2012年9月 (3)

read

Netfilter/iptables/OpenVPN guard

最新评论

1. Re:Android Studio 工程的 .gitignore

very goof

--fedorayang

2. Re:mac osx 下编译 OpenWrt

如果之前已经安装xcode呢Print or change the path to the active developer directory. This directorycontrols whi.....

--&token

3. Re:mac osx 下编译 OpenWrt

要把xcode安装到 /Volumes/OpenWrt?

--&token

4. Re:openwrt spi flash 分区适配过程

@白い目1. 没错。rootfs_data 是 jffs2 的。2. squashfs 分区和 jffs2 是独立的。如果想调整大小，需要在 dts 里调整。对于我这里的情况，只能在dts 里把 fir.....

--sammei

5. Re:openwrt spi flash 分区适配过程

据我之前查的资料，rootfs_data分区应该是可读写分区，采用jffs2格式，不知对不对？
如何将只读的squashfs分区的一部分重新分区成可读写的jffs2分区呢？

--白い目

阅读排行榜

1. openwrt: Makefile 框架分析 (15469)

2. 一款工作记录软件(5921)

3. 网络知识： 物理层PHY 和 网络层 MAC(5873)

4. openwrt hotplug(4926)

5. spi flash 操作(4829)

> staging_dir/toolchain-mipsel_24kec+dsp_gcc-4.8-linaro_uClibc-0.9.33.2

- TARGET_DIR

> build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/root-ramips

kernel 编译：

target/linux/ramips/Makefile: `$(eval $(call BuildTarget))`
target/linux/Makefile : `export TARGET_BUILD=1`
include/target.mk:

```
ifeq ($(TARGET_BUILD),1)
  include $(INCLUDE_DIR)/kernel-build.mk
  BuildTarget?=$(BuildKernel)
endif
```

BuildKernel是include/kernel-build.mk定义的一个多行变量，其中描述了如何编译内核, 主要关注其中install规则的依赖链：

```
  $(KERNEL_BUILD_DIR)/symtab.h: FORCE
    rm -f $(KERNEL_BUILD_DIR)/symtab.h
    touch $(KERNEL_BUILD_DIR)/symtab.h
    +$(MAKE) $(KERNEL_MAKEOPTS) vmlinux
    ...

  $(LINUX_DIR)/.image: $(STAMP_CONFIGURED) $(if $(CONFIG_STRIP_KERNEL_EXPORTS),$(KERN
    $(Kernel/CompileImage)
    $(Kernel/CollectDebug)
    touch $$@


  install: $(LINUX_DIR)/.image
    +$(MAKE) -C image compile install TARGET_BUILD=
```

```
1. 触发make vmlinux命令生成vmlinux： install --> $(LINUX_DIR)/.image --> $(KERNEL_BUILD_

2. 对vmlinux做objcopy, strip操作: $(LINUX_DIR)/.image --> $(Kernel/CompileImage) --> $

    $(KERNEL_CROSS)objcopy -O binary $(OBJCOPY_STRIP) -S $(LINUX_DIR)/vmlinux $(LINUX
        --> build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/vm

    $(KERNEL_CROSS)objcopy $(OBJCOPY_STRIP) -S $(LINUX_DIR)/vmlinux $(KERNEL_BUILD_DI
        --> build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/vm

    $(CP) $(LINUX_DIR)/vmlinux $(KERNEL_BUILD_DIR)/vmlinux.debug
        --> build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/vm
```

生成firmware

firmware由kernel和rootfs两个部分组成，要对两个部分先分别处理，然后再合并成一个.bin文件。先看一下这个流程。

"target/linux/ramips/image/Makefile" 文件中的最后一句： `$(eval $(call BuildImage))` ，将BuildImage展开在这里。BuildImage定义在 include/image.mk 文件中，其中定义了数个目标的规则。

```
define BuildImage

    compile: compile-targets FORCE
        **$(call Build/Compile)**

    install: compile install-targets FORCE
        ...
        $(call Image/BuildKernel) ## 处理vmlinux
        ...
        $(call Image/mkfs/squashfs) ## 生成squashfs，并与vmlinux合并成一个.bin文件
        ...

endef
```

处理vmlinux: Image/BuildKernel

target/linux/ramips/image/Makefile:

```
define Image/BuildKernel
    cp $(KDIR)/vmlinux.elf $(BIN_DIR)/$(VMLINUX).elf
    cp $(KDIR)/vmlinux $(BIN_DIR)/$(VMLINUX).bin
    $(call CompressLzma,$(KDIR)/vmlinux,$(KDIR)/vmlinux.bin.lzma)
    $(call MkImage,lzma,$(KDIR)/vmlinux.bin.lzma,$(KDIR)/uImage.lzma)
    cp $(KDIR)/uImage.lzma $(BIN_DIR)/$(UIMAGE).bin
ifneq ($(CONFIG_TARGET_ROOTFS_INITRAMFS),)
    cp $(KDIR)/vmlinux-initramfs.elf $(BIN_DIR)/$(VMLINUX)-initramfs.elf
    cp $(KDIR)/vmlinux-initramfs $(BIN_DIR)/$(VMLINUX)-initramfs.bin
    $(call CompressLzma,$(KDIR)/vmlinux-initramfs,$(KDIR)/vmlinux-initramfs.bin.lzma)
    $(call MkImage,lzma,$(KDIR)/vmlinux-initramfs.bin.lzma,$(KDIR)/uImage-initramfs.l
    cp $(KDIR)/uImage-initramfs.lzma $(BIN_DIR)/$(UIMAGE)-initramfs.bin
endif
    $(call Image/Build/Initramfs)
endef
```

lzma压缩内核

build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/ 目录中:

```
lzma e vmlinux -lc1 -lp2 -pb2 vmlinux.bin.lzma
```

MkImage

build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/ 目录中：

```
mkimage -A mips -O linux -T  kernel -C lzma -a 0x80000000 -e 0x80000000 -n "MIPS Open
```

copy

```
VMLINUX:=$(IMG_PREFIX)-vmlinux --> openwrt-ramips-mt7620a-vmlinux
UIMAGE:=$(IMG_PREFIX)-uImage --> openwrt-ramips-mt7620a-uImage
cp $(KDIR)/uImage.lzma $(BIN_DIR)/$(UIMAGE).bin
```

把uImage.lzma复制到bin/ramips/目录下：
cp $(KDIR)/**uImage.lzma** bin/ramips/**openwrt-ramips-mt7620a-uImage**

制作squashfs，生成.bin: $(call Image/mkfs/squashfs)

```
    define Image/mkfs/squashfs
        @mkdir -p $(TARGET_DIR)/overlay
        $(STAGING_DIR_HOST)/bin/mksquashfs4 $(TARGET_DIR) $(KDIR)/root.squashfs -nopa
        $(call Image/Build,squashfs)
endif
```

mkdir -p $(TARGET_DIR)/overlay

mkdir -p build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/root-ramips/overlay

mksquashfs4

```
$(STAGING_DIR_HOST)/bin/mksquashfs4 $(TARGET_DIR) $(KDIR)/root.squashfs -nopad -noapp
```

制作squashfs文件系统，生成root.squashfs:

```
mksquashfs4 root-ramips root.squashfs -nopad -noappend -root-owned -comp gzip -b 256k
```

$(call Image/Build,squashfs)

在 target/linux/ramips/image/Makefile 中：

```
define Image/Build
    $(call Image/Build/$(1))
    dd if=$(KDIR)/root.$(1) of=$(BIN_DIR)/$(IMG_PREFIX)-root.$(1) bs=128k conv=sync
    $(call Image/Build/Profile/$(PROFILE),$(1))
endef
```

- dd if=$(KDIR)/root.squashfsof=(BIN_DIR)/$(IMG_PREFIX)-root.squashfs bs=128k conv=sync

dd if=build_dir/target-mipsel_24kec+dsp_uClibc-0.9.33.2/linux-ramips_mt7620a/**root.squashfs** of=bin/ramips/**openwrt-ramips-mt7620-root.squashfs** bs=128k conv=sync

- $(call Image/Build/Profile/(PROFILE),squashfs)

target/linux/ramips/mt7620a/profiles/00-default.mk, 中调用 Profile 函数：

```
$(eval $(call Profile,Default))
```

include/target.mk 中定义了 Profile 函数， 其中令 PROFILE=Default

```
define Image/Build/Profile/Default
    $(call Image/Build/Profile/MT7620a,$(1))
    ...
endef
```

规则依赖序列如下：

```
$(call Image/Build/Profile/$(PROFILE),squashfs)
  --> $(call BuildFirmware/Default8M/squashfs,squashfs,mt7620a,MT7620a)
      --> $(call BuildFirmware/OF,squashfs,mt7620a,MT7620a,8060928)
          --> $(call MkImageLzmaDtb,mt7620a,MT7620a)
              --> $(call PatchKernelLzmaDtb,mt7620a,MT7620a)
              --> $(call MkImage,lzma,$(KDIR)/vmlinux-mt7620a.bin.lzma,$(KDIR)/vmlinu
      --> $(call MkImageSysupgrade/squashfs,squashfs,mt7620a,8060928)
```

其中的主要步骤：

- 复制： cp $(KDIR)/vmlinux(KDIR)/vmlinux-mt7620a
- 生成dtb文件： $(LINUX_DIR)/scripts/dtc/dtc -Odtb -o(KDIR)/MT7620a.dtb ../dts/MT7620a.dts
- 将内核与dtb文件合并：$(STAGING_DIR_HOST)/bin/patch-dtb(KDIR)/vmlinux-mt7620a $(KDIR)/MT7620a.dtb
- 使用lzma压缩：$(call CompressLzma,(KDIR)/vmlinux-mt7620a,$(KDIR)/vmlinux-mt7620a.bin.lzma)
- 将lzma压缩后的文件经过mkimage工具处理，即在头部添加uboot可识别的信息。

接下来就是合并生成firmware固件了：

MkImageSysupgrade/squashfs, squashfs, mt7620a,8060928

cat vmlinux-mt7620a.uImage root.squashfs > openwrt-ramips-mt7620-mt7620a-squashfs-sysupgrade.bin
--> 制作squashfs bin文档, 并确认它的大小 < 8060928 才是有效的，否则报错。

总结： 整个流程下来，其实最烦索的还是对内核生成文件vmlinux的操作，经过了objcopy, patch-dtb, lzma, mkimage 等过程生成一个uImage，再与mksquashfs工具制作的文件系统rootfs.squashfs合并。

分类: OpenWrt

sammei
关注 - 0
粉丝 - 11
+加关注

2 0

« 上一篇：openwrt mt7620 内存大小检测
» 下一篇：openwrt: sysupgrade

posted @ 2014-09-14 17:28 sammei 阅读(15469) 评论(0) 编辑 收藏