u-boot 的 Makefile 分析

U－BOOT 是一个 LINUX 下的工程，在编译之前必须已经安装对应体系结构的交叉编译环境，这里只针对 ARM，编译器系列软件为 arm-linux-*。

U－BOOT 的下载地址： http://sourceforge.net/projects/u-boot
我下载的是 1.1.6 版本，一开始在 FTP 上下载了一个次新版，结果编译失败。1.1.6 是没问题的。

u-boot 源码结构

解压就可以得到全部 u-boot 源程序。在顶层目录下有 18 个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为 3 类。

第 1 类目录与处理器体系结构或者开发板硬件直接相关；

第 2 类目录是一些通用的函数或者驱动程序；

第 3 类目录是 u-boot 的应用程序、工具或者文档。

u-boot 的源码顶层目录说明

目　录	特　性	解 释 说 明
board	平台依赖	存放电路板相关的目录文件， 例如：RPXlite(mpc8xx)、 smdk2410(arm920t)、 sc520_cdp(x86) 等目录
cpu	平台依赖	存放 CPU 相关的目录文件 例如：mpc8xx、ppc4xx、 arm720t、arm920t、 xscale、i386 等目录
lib_ppc	平台依赖	存放对 PowerPC 体系结构通用的文件， 主要用于实现 PowerPC 平台通用的函数
lib_arm	平台依赖	存放对 ARM 体系结构通用的文件， 主要用于实现 ARM 平台通用的函数

lib_i386	平台依赖	存放对 X86 体系结构通用的文件，主要用于实现 X86 平台通用的函数
include	通用	头文件和开发板配置文件，所有开发板的配置文件都在 configs 目录下
common	通用	通用的多功能函数实现
lib_generic	通用	通用库函数的实现
net	通用	存放网络的程序
fs	通用	存放文件系统的程序
post	通用	存放上电自检程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动
disk	通用	硬盘接口程序
rtc	通用	RTC 的驱动程序
dtt	通用	数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如 helloworld
tools	工具	存放制作 S-Record 或者 u-boot 格式的映像等工具，例如 mkimage
doc	文档	开发使用文档

u-boot 的源代码包含对几十种处理器、数百种开发板的支持。可是对于特定的开发板，配置编译过程只需要其中部分程序。这里具体以 S3C2410 & arm920t 处理器为例，具体分析 S3C2410 处理器和开发板所依赖的程序，以及 u-boot 的通用函数和工具。

编译

以 smdk_2410 板为例，编译的过程分两部：

```
# make smdk2410_config
# make
```

顶层 Makefile 分析

要了解一个 LINUX 工程的结构必须看懂 Makefile，尤其是顶层的，UNIX 什么东西都用文档去管理、配置。

以 smdk_2410 为例，顺序分析 Makefile 大致的流程及结构如下：

1) Makefile 中定义了源码及生成的目标文件存放的目录,目标文件存放目录 BUILD__DIR 可以通过 make O=dir 指定。如果没有指定，则设定为源码顶层目录。一般编译的时候不指定输出目录，则 BUILD__DIR 为空。其它目录变量定义如下：

#OBJTREE 和 LNDIR 为存放生成文件的目录，TOPDIR 与 SRCTREE 为源码所在目录

```
OBJTREE  := $(if $(BUILD_DIR),$(BUILD_DIR),$(CURDIR))
SRCTREE  := $(CURDIR)
TOPDIR  := $(SRCTREE)
LNDIR  := $(OBJTREE)
export TOPDIR SRCTREE OBJTREE
```

2）定义变量 MKCONFIG：这个变量指向一个脚本，即顶层目录的 mkconfig。

```
MKCONFIG := $(SRCTREE)/mkconfig
export MKCONFIG
```

在编译 U－BOOT 之前，先要执行

```
# make smdk2410_config
```

smdk2410_config 是 Makefile 的一个目标，定义如下：

```
smdk2410_config : unconfig
 @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0

 unconfig::
 @rm -f $(obj)include/config.h $(obj)include/config.mk \
  $(obj)board/*/config.tmp $(obj)board/*/*/config.tmp
```

显然，执行# make smdk2410_config 时，先执行 unconfig 目标，注意不指定输出目标时，obj，src 变量均为空，unconfig 下面的命令清理上一次执行 make *_config 时生成的头文件和 makefile 的包含文件。主要是 include/config.h 和 include/config.mk 文件。

然后才执行命令

```
@$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

MKCONFIG 是顶层目录下的 mkcofig 脚本文件，后面五个是传入的参数。

对于 smdk2410_config 而言，mkconfig 主要做三件事：

在 include 文件夹下建立相应的文件（夹）软连接，

#如果是 ARM 体系将执行以下操作：

```
#ln -s     asm-arm        asm

#ln -s   arch-s3c24x0     asm-arm/arch
#ln -s    proc-armv       asm-arm/proc
```

生成 Makefile 包含文件 include/config.mk，内容很简单，定义了四个变量：

```
ARCH    = arm
CPU     = arm920t
BOARD   = smdk2410
SOC     = s3c24x0
```

生成 include/config.h 头文件，只有一行：

```
/* Automatically generated - do not edit */
#include "config/smdk2410.h"
```

mkconfig 脚本文件的执行至此结束，继续分析 Makefile 剩下部分。

3）包含 include/config.mk，其实也就相当于在 Makefile 里定义了上面四个变量而已。

4) 指定交叉编译器前缀：

```
ifeq ($(ARCH),arm)#这里根据 ARCH 变量，指定编译器前缀。
CROSS_COMPILE = arm-linux-
endif
```

5)包含 config.mk:

#包含顶层目录下的 config.mk，这个文件里面主要定义了交叉编译器及选项和编译规则

```
# load other configuration
include $(TOPDIR)/config.mk
```

下面分析 config.mk 的内容：

@包含体系，开发板，CPU 特定的规则文件：

```
ifdef ARCH   #指定预编译体系结构选项
sinclude $(TOPDIR)/$(ARCH)_config.mk # include architecture dependend rules
endif
ifdef CPU #定义编译时对齐，浮点等选项
sinclude $(TOPDIR)/cpu/$(CPU)/config.mk # include   CPU specific rules
endif
ifdef SOC #没有这个文件
sinclude $(TOPDIR)/cpu/$(CPU)/$(SOC)/config.mk # include   SoC specific rules
endif

ifdef BOARD   #指定特定板子的镜像连接时的内存基地址，重要！
sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk # include board specific rules
endif
```

@定义交叉编译链工具

```
# Include the make variables (CC, etc...)
#
```

```
AS = $(CROSS_COMPILE)as
LD = $(CROSS_COMPILE)ld
CC = $(CROSS_COMPILE)gcc
CPP = $(CC) -E
AR = $(CROSS_COMPILE)ar
NM = $(CROSS_COMPILE)nm
STRIP = $(CROSS_COMPILE)strip
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump
RANLIB = $(CROSS_COMPILE)RANLIB
```

@定义 AR 选项 ARFLAGS，调试选项 DBGFLAGS，优化选项 OPTFLAGS

预处理选项 CPPFLAGS，C 编译器选项 CFLAGS，连接选项 LDFLAGS

```
LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE)
$(PLATFORM_LDFLAGS)   #指定了起始地址 TEXT_BASE
```

@指定编译规则：

```
$(obj)%.s: %.S
 $(CPP) $(AFLAGS) -o $@ $<
$(obj)%.o: %.S
 $(CC) $(AFLAGS) -c -o $@ $<
$(obj)%.o: %.c
 $(CC) $(CFLAGS) -c -o $@ $<
```

回到顶层 makefile 文件：

6）U-boot 需要的目标文件。

```
OBJS  = cpu/$(CPU)/start.o #  顺序很重要，start.o 必须放第一位
```

7）需要的库文件：

```
LIBS  = lib_generic/libgeneric.a
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
```

```
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a
\
 fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
LIBS += net/libnet.a
LIBS += disk/libdisk.a
LIBS += rtc/librtc.a
LIBS += dtt/libdtt.a
LIBS += drivers/libdrivers.a
LIBS += drivers/nand/libnand.a
LIBS += drivers/nand_legacy/libnand_legacy.a
LIBS += drivers/sk98lin/libsk98lin.a
LIBS += post/libpost.a post/cpu/libcpu.a
LIBS += common/libcommon.a
LIBS += $(BOARDLIBS)

LIBS := $(addprefix $(obj),$(LIBS))
.PHONY : $(LIBS)
```

根据上面的 include/config.mk 文件定义的 ARCH、CPU、BOARD、SOC 这些变量。硬件平台依赖的目录文件可以根据这些定义来确定。SMDK2410 平台相关目录及对应生成的库文件如下。

board/smdk2410/ ：库文件 board/smdk2410/libsmdk2410.a

cpu/arm920t/ ：库文件 cpu/arm920t/libarm920t.a

cpu/arm920t/s3c24x0/ ：库文件 cpu/arm920t/s3c24x0/libs3c24x0.a

lib_arm/ : 库文件 lib_arm/libarm.a

include/asm-arm/ :下面两个是头文件。

include/configs/smdk2410.h

8）最终生成的各种镜像文件：

```
ALL = $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map
$(U_BOOT_NAND)

all:  $(ALL)

$(obj)u-boot.hex: $(obj)u-boot
  $(OBJCOPY) ${OBJCFLAGS} -O ihex $< $@

$(obj)u-boot.srec: $(obj)u-boot
  $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@

$(obj)u-boot.bin: $(obj)u-boot
  $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
#这里生成的是 U-boot   的 ELF 文件镜像
$(obj)u-boot:  depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
  UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed  -n -e
''''''''''''''''''''''''''''s/.*\(__u_boot_cmd_.*\)/-u\1/p''''''''''''''''''''''''''''|sort|uniq`;\
  cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJS) \
   --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
   -Map u-boot.map -o u-boot
```

分析一下最关键的 **u-boot ELF** 文件镜像的生成：

@依赖目标 **depend** :生成各个子目录的.depend 文件，.depend 列出每个目标文件的依赖文件。生成方法，调用每个子目录的 make _depend。

```
depend dep:
  for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir _depend ; done
```

@依赖目标 **version**：生成版本信息到版本文件 VERSION_FILE 中。

```
version:
  @echo -n "#define U_BOOT_VERSION \"U-Boot " > $(VERSION_FILE); \
  echo -n "$(U_BOOT_VERSION)" >> $(VERSION_FILE); \
  echo -n $(shell $(CONFIG_SHELL) $(TOPDIR)/tools/setlocalversion \
    $(TOPDIR)) >> $(VERSION_FILE); \
  echo "\"" >> $(VERSION_FILE)
```

@伪目标 SUBDIRS: 执行 tools ,examples ,post,post\cpu 子目录下面的 make 文件。

```
SUBDIRS = tools \
   examples \
   post \
   post/cpu
.PHONY : $(SUBDIRS)

$(SUBDIRS):
  $(MAKE) -C $@ all
```

@依赖目标$(OBJS)，即 cpu/start.o

```
$(OBJS):
  $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))
```

@依赖目标$(LIBS)，这个目标太多，都是每个子目录的库文件*.a ，通过执行相应子目录下的 make 来完成：

```
$(LIBS):
  $(MAKE) -C $(dir $(subst $(obj),,$@))
```

@依赖目标$(LDSCRIPT)：

```
LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds
LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE)
$(PLATFORM_LDFLAGS)
```

对于 smdk2410,LDSCRIPT 即连接脚本文件是 board/smdk2410/u-boot.lds，定义了连接时各个目标文件是如何组织的。内容如下：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
```

```
{
 . = 0x00000000;

 . = ALIGN(4);
 .text    :/*.text 的基地址由 LDFLAGS 中-Ttext $(TEXT_BASE)指定*/
 {                    /*smdk2410 指定的基地址为 0x33f80000*/
   cpu/arm920t/start.o (.text)         /*start.o 为首*/
   *(.text)
 }

 . = ALIGN(4);
 .rodata : { *(.rodata) }

 . = ALIGN(4);
 .data : { *(.data) }

 . = ALIGN(4);
 .got : { *(.got) }

 . = .;
 __u_boot_cmd_start = .;
 .u_boot_cmd : { *(.u_boot_cmd) }
 __u_boot_cmd_end = .;

 . = ALIGN(4);
 __bss_start = .;
 .bss : { *(.bss) }
 _end = .;
}
```

@执行连接命令：

```
cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJS) \
  --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
  -Map u-boot.map -o u-boot
```

其实就是把 start.o 和各个子目录 makefile 生成的库文件按照 LDFLAGS 连接在一起，生成 ELF 文件 u-boot 和连接时内存分配图文件 u-boot.map。

9)对于各子目录的 makefile 文件，主要是生成*.o 文件然后执行 AR 生成对应的库文件。如 lib_generic 文件夹 Makefile：

```
LIB = $(obj)libgeneric.a

COBJS = bzlib.o bzlib_crctable.o bzlib_decompress.o \
   bzlib_randtable.o bzlib_huffman.o \
   crc32.o ctype.o display_options.o ldiv.o \
   string.o vsprintf.o zlib.o

SRCS  := $(COBJS:.o=.c)
OBJS := $(addprefix $(obj),$(COBJS))

$(LIB): $(obj).depend $(OBJS) #项层 Makefile 执行 make libgeneric.a
 $(AR) $(ARFLAGS) $@ $(OBJS)
```

整个 makefile 剩下的内容全部是各种不同的开发板的*_config:目标的定义了。

概括起来，工程的编译流程也就是通过执行执行一个 make *_config 传入 ARCH，CPU，BOARD，SOC 参数，mkconfig 根据参数将 include 头文件夹相应的头文件夹连接好，生成 config.h。然后执行 make 分别调用各子目录的 makefile 生成所有的 obj 文件和 obj 库文件*.a. 最后连接所有目标文件，生成镜像。不同格式的镜像都是调用相应工具由 elf 镜像直接或者间接生成的。