make 简介

make[1]是类 Unix 系统中常用的用于软件自动构建的工具，常被用于 C/C++编写的软件的编译，安装等工作。Wikipedia 对其的定义为：
使用 **make** 的一个关键原因是其能对代码的被修改时间进行跟踪，并只构建包含修改时间晚于上次编译时间的文件的规则。这将大大提高软件开发维护的效率。

> *"In software development,* **make** *is a utility for automatically building large applications. Files specifying instructions for make are called* **Makefile**s*.* **make** *is an expert system that tracks which files have changed since the last time the project was built and invokes the compiler on only those source code files and their dependencies."*

Apache **Ant** 与微软的 **cmake** 都可以完成与 **make** 类似的工作，集成开发环境（IDE）也可以自动化地完成软件的构建，但是由于 **make** 的高度可定制性和独特的时间相关性，仍然在当代的软件开发活动中被大量的使用。

makefile 简介

make 需要一个文件来指导其进行相应的构建动作，这个文件一般被称为 **makefile**。**makefile** 中记录了构建软件所需要的文件和它们之间的依赖关系；同时，**makefile** 中还可以定义构建规则和变量，他们可以在构建过程中起到相应的作用。
典型的 **makefile** 结构为 (1)：

```
# Comments use the hash symbol
target: dependencies
    command 1
    command 2
        .
        .
        .
    command n
```

注意，每个 `target` 下的每个 `command` 都需要以一个 TAB 开头，而不能以空格替代[2]。
一个简单的 **makefile** 如下：

[1] 本文中的 make 均指 GNU make，关于 GNU make 的详细信息可以访问 http://www.gnu.org/software/make。
[2] 这一约定在 GNU make 中有效。

```
helloworld: helloworld.o
    cc -o $@ $<

helloworld.o: helloworld.c
    cc -c -o $@ $<

.PHONY: clean

clean:
    rm -f helloworld helloworld.o
```

当在此 **makefile** 存在的目录下执行 **make** 时，**makefile** 中指定的第一条规则 **helloworld** 将会被执行。由于 **helloworld** 依赖于 **helloworld.o**，而 **helloworld.o** 又依赖于 **helloworld.c**，所以实际的构建过程中执行的第一条命令是：

```
cc -c -o helloworld.o helloworld.c
```

注意原规则中的$@将被替代为 target，$<将被替换为 dependencies 中的第一个文件名。这些特殊的符号被称为“自动变量” (2)。类似的还有：$%，$?，$^，$+，$*等，具体请参阅 **make** 手册。

当 **helloworld.o** 被满足之后，规则 **helloworld** 中的命令会被执行：

关键词.PHONY 定义其后续的规则不受修改时间的约束，即若该规则被调用，一定保证其执行。这个 **makefile** 中定义的 clean 规则即为一例。若执行 **make clean**，将删除 **helloworld.o**

```
cc -c -o helloworld.o helloworld.c
```

和 **helloworld** 文件。

makefile 描述

1 公共引用文件 include.mk

```
# Common includes in Makefile
#
# Copyright (C) 2007 Beihang University
# Written By Zhu Like ( zlike@cse.buaa.edu.cn )



CROSS_COMPILE := /usr/eldk/usr/bin/mips_4KC-
CC            := $(CROSS_COMPILE)gcc
CFLAGS        := -O -G 0 -mno-abicalls \
                 -fno-builtin -Wa,-xgot -Wall -fPIC
LD            := $(CROSS_COMPILE)ld
```

该文件定义了构建过程中所要用到的常用变量。CC 和 LD 分别为 C 编译器和链接器。CFLAGS

为传递给编译器的参数。请修改 CROSS_COMPILE 的路径为相应的交叉编译工具链路径。

2 主 makefile

变量定义部分

```
drivers_dir   := drivers
boot_dir      := boot
user_dir      := user
init_dir      := init
lib_dir       := lib
fs_dir        := fs
mm_dir        := mm
tools_dir     := tools
vmlinux_elf   := gxemul/vmlinux
link_script   := $(tools_dir)/scse0_3.lds

modules       := boot drivers init lib mm user fs
objects       := $(boot_dir)/start.o             \
                 $(init_dir)/main.o          \
                 $(init_dir)/init.o          \
                 $(init_dir)/code.o          \
                 $(drivers_dir)/gxconsole/console.o \
                 $(lib_dir)/*.o              \
                 $(user_dir)/*.x \
                 $(fs_dir)/*.x \
                 $(mm_dir)/*.o
```

这一部分定义了构建过程的所有模块名称（modules）、相应路径（模块名_dir），构建目标（vmlinux_elf），创建该目标所需要的文件（objects）和链接脚本（link_script）。

目标定义部分

```
.PHONY: clean

all: $(modules) vmlinux

vmlinux: $(modules)
    $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)

$(modules):
    $(MAKE) --directory=$@
```

对于 module 目标的构建采用了调用子 **makefile** 的方法，即在每个模块对应的目录中放置构建该模块的 **makefile**，在上层 **makefile** 中以$(MAKE) --directory=$@的方法调用来实现构建。

clean 目标为 PHONY 目标，实现中使用简单的 **bash** 脚本依次轮询每个模块 **makefile** 中的 clean 目标并执行。

```
clean:
    for d in $(modules);                    \
        do                                  \
            $(MAKE) --directory=$$d clean; \
        done;                               \
    rm -rf *.o *~ $(vmlinux_elf)
```

引用 include.mk

以下语句引用了 include.mk 中定义的变量

```
include include.mk
```

3 各模块 makefile

各模块的 **makefile** 相对较为简单。较为特殊的是目录 **fs** 下的 **makefile**，其中用到了两个预编译的程序 **bintoc** 和 **fsformat**：

```
%.b.c: %.b
    echo create $@
    echo bintoc $* $< > $@~
    ./bintoc $* $< > $@~ && mv -f $@~ $@
```

```
fs.img: $(FSIMGFILES)
    dd  if=/dev/zero  of=../gxemul/fs.img  bs=4096  count=1024
2>/dev/null
    ./fsformat ../gxemul/fs.img $(FSIMGFILES)
```

参考

1. **Wikipedia.** make (software). Wikipedia, the free encyclopedia. [联机] Wikipedia. http://en.wikipedia.org./wiki/Make_(software).

2. **MecklenburgRobert.** Managing Projects with GNU Make. 无出版地：O'Reilly Media, Inc., 2004.

Ld 简介

链接器把一个或多个输入文件合成一个输出文件.

输入文件: 目标文件或链接脚本文件.
输出文件: 目标文件或可执行文件.

> 目标文件(包括可执行文件)具有固定的格式, 在 UNIX 或 GNU/Linux 平台下, 一般为 ELF 格式. 若想了解更多, 可参考 UNIX/Linux 平台可执行文件格式分析 (http://www-128.ibm.com/developerworks/cn/linux/l-excutff/)

有时把输入文件内的 section 称为输入 section(input section), 把输出文件内的 section 称为输出 section(output sectin).

目标文件的每个 section 至少包含两个信息: 名字和大小. 大部分 section 还包含与它相关联的一块数据, 称为 section contents(section 内容). 一个 section 可被标记为"loadable(可加载的)"或"allocatable(可分配的)".

loadable section: 在输出文件运行时, 相应的 section 内容将被载入进程地址空间中.

allocatable section: 内容为空的 section 可被标记为"可分配的". 在输出文件运行时, 在进程地址空间中空出大小同 section 指定大小的部分. 某些情况下, 这块内存必须被置零.

如果一个 section 不是"可加载的"或"可分配的", 那么该 section 通常包含了调试信息. 可用 objdump -h 命令查看相关信息.

每个"可加载的"或"可分配的"输出 section 通常包含两个地址: VMA(virtual memory address 虚拟内存地址或程序地址空间地址)和 LMA(load memory address 加载内存地址或进程地址空间地址). 通常 VMA 和 LMA 是相同的.

> 在目标文件中, loadable 或 allocatable 的输出 section 有两种地址: VMA(virtual Memory Address)和 LMA(Load Memory Address). VMA 是执行输出文件时 section 所在的地址, 而 LMA 是加载输出文件时 section 所在的地址. 一般而言, 某 section 的 VMA == LMA. 但在嵌入式系统中, 经常存在加载地址和执行地址不同的情况: 比如将输出文件加载到开发板的 flash 中(由 LMA 指定), 而在运行时将位于 flash 中的输出文件复制到 SDRAM 中(由 VMA 指定).

可这样来理解 VMA 和 LMA, 假设:
(1) .data section 对应的 VMA 地址是 0x08050000, 该 section 内包含了 3 个 32 位全局变量, i、j 和 k, 分别为 1,2,3.
(2) .text section 内包含由"printf("j=%d ", j);"程序片段产生的代码.

连接时指定.data section 的 VMA 为 0x08050000, 产生的 printf 指令是将地址为 0x08050004 处的 4 字节内容作为一个整数打印出来。

如果.data section 的 LMA 为 0x08050000，显然结果是 j=2
如果.data section 的 LMA 为 0x08050004，显然结果是 j=1

还可这样理解 LMA:
.text section 内容的开始处包含如下两条指令(intel i386 指令是 10 字节，每行对应 5 字节):

```
jmp 0x08048285
movl $0x1,%eax
```

如果.text section 的 LMA 为 0x08048280, 那么在进程地址空间内 0x08048280 处为“jmp 0x08048285”指令, 0x08048285 处为 movl $0x1,%eax 指令. 假设某指令跳转到地址 0x08048280, 显然它的执行将导致%eax 寄存器被赋值为 1.

如果.text section 的 LMA 为 0x08048285, 那么在进程地址空间内 0x08048285 处为“jmp 0x08048285”指令, 0x0804828a 处为 movl $0x1,%eax 指令. 假设某指令跳转到地址 0x08048285, 显然它的执行又跳转到进程地址空间内 0x08048285 处, 造成死循环.

符号(symbol): 每个目标文件都有符号表(SYMBOL TABLE), 包含已定义的符号(对应全局变量和 static 变量和定义的函数的名字)和未定义符号(未定义的函数的名字和引用但没定义的符号)信息.

符号值: 每个符号对应一个地址, 即符号值(这与 c 程序内变量的值不一样, 某种情况下可以把它看成变量的地址). 可用 nm 命令查看它们.