## 第六章 Makefile与GCC

### 6.1 交叉编译器

#### 6.1.1 什么是交叉编译

简单地说，我们在PC机上编译程序时，这些程序是在PC机上运行的。我们想让一个程序在ARM板子上运行，怎么办？

ARM板性能越来越强，可以认为ARM板就相当于一台PC，当然可以在ARM板上安装开发工具，比如安装ARM版本的GCC，这样就可以在ARM板上编译程序，在ARM板上直接运行这个程序。

但是，有些ARM板性能弱，或者即使它的性能很强也强不过PC机，所以更多时候我们是在PC机上开发、编译程序，再把这个程序下载到ARM板上去运行。

这就引入一个问题：

1) 我们使用工具比如说gcc编译出的程序是给PC机用的，这程序里的指令是X86指令。

2)那么能否使用同一套工具给ARM板编译程序？

显示不行，因为X86的指令肯定不能在ARM板子上运行。所以我们需要使用另一套工具：交叉编译工具链。

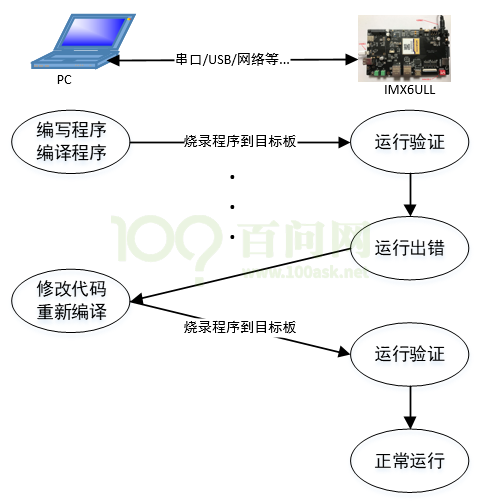
为何叫“交叉”？

首先，我们是在PC机上使用这套工具链来编译程序；

然后再把程序下载到ARM板运行；

如果程序不对，需要回到PC机修改程序、编译程序，再把程序下载到ARM板上运行、验证。如此重复。

在这个过程中，我们一会在PC上写程序、编译程序，一会在ARM板上运行、验证，中间来来回回不断重复，所以称之为“交叉”。对于所用的工具链，它是在PC机上给ARM板编译程序，称之为“交叉工具链”。



有很多种交叉工具链，举例如下：

1. Ubuntu平台：交叉工具链有arm-linux-gcc编译器、arm-linux-gnueabihf-编译器。

2. Windows 平台：利用ADS（ARM开发环境），使用armcc编译器。

3. Windows平台：利用cygwin环境，运行arm-elf-gcc编译器。

#### 6.1.2 验证实例

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序\6\_Makefile与GCC\001\_gcc\_01”目录下。

对于main.c，我们采用gcc编译得到可执行程序，把它放在目标板上看看是否能运行起来，代码如下：

01 #include <stdio.h>

02

03 int main()

04 {

05 printf("100ask\n");

06 return 0;

07 }

在虚拟机编译运行：

$ gcc main.c –o 100ask

$ ./100ask

100ask

$

在上面的运行结果，没有任问题，然后我们将这个可执行程序放到目标板上，如下：

$ chmod 777 100ask

$ ./100ask

./100ask: line 1: syntax error: unexpected “(”

$

报错无法运行。说明为X86平台制作的可执行文件，不能在其他架构平台上运行。交叉编译就是为了解决这个问题。

为了方便实验，我们在Ubuntu中使用gcc来做实验，如果想使用交叉编译，参考章节[《2.2 安装SDK、设置工具链》](#_安装SDK、设置工具链)，安装好工具链，设置好环境变量后，将所有的gcc替换为arm-linux-gnueabihf-gcc就可以完成交叉编译。

其中：

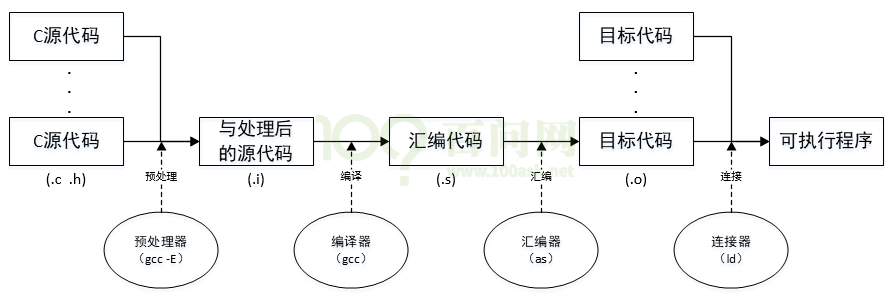
gcc是在x86电脑上运行，为x86机器编译程序。

arm-linux-gnueabihf-gcc也是x86电脑上运行，为RSIC（精简指令集）ARM架构芯片编译程序。

### 6.2 GCC常用选项及编译过程详解

#### 6.2.1 gcc编译过程详解

一个C/C++文件要经过预处理(preprocessing)、编译(compilation)、汇编(assembly)和连接(linking)等4步才能生成可执行文件，编译流程图如下。



预处理：

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。

编译：

对预处理后的源码进行词法和语法分析，生成目标系统的汇编代码文件，后缀名为“.s”。

汇编：

对汇编代码进行优化，生成目标代码文件，后缀名为“.o”。

链接：

解析目标代码中的外部引用，将多个目标代码文件连接为一个可执行文件。

编译器利用这4个步骤中的一个或多个来处理输入文件，源文件的后缀名表示源文件所用的语言，后缀名控制着编译器的缺省动作

|  |  |  |
| --- | --- | --- |
| 后缀名 | 语言种类 | 后期操作 |
| .c | C源程序 | 预处理、编译、汇编 |
| .C | C++源程序 | 预处理、编译、汇编 |
| .cc | C++源程序 | 预处理、编译、汇编 |
| .cxx | C++源程序 | 预处理、编译、汇编 |
| .m | Objective-C源程序 | 预处理、编译、汇编 |
| .i | 预处理后的C文件 | 编译、汇编 |
| .ii | 预处理后的C++文件 | 编译、汇编 |
| .s | 汇编语言源程序 | 汇编 |
| .S | 汇编语言源程序 | 预处理、汇编 |
| .h | 预处理器文件 | 通常不出现在命令行上 |

其他后缀名的文件被传递给连接器(linker)，通常包括：

.o：目标文件(Object file，OBJ文件)

.a：归档库文件(Archive file)

在编译过程中，除非使用了“-c”，“-S”或“-E”选项(或者编译错误阻止了完整的过程)，否则最后的步骤总是连接。在连接阶段中，所有对应于源程序的.o文件，“-l”选项指定的库文件，无法识别的文件名(包括指定的“.o”目标文件和“.a”库文件)按命令行中的顺序传递给连接器。

#### 6.2.2 gcc命令

gcc命令格式是：

gcc [选项] 文件列表

gcc命令用于实现c程序编译的全过程。文件列表参数指定了gcc的输入文件，选项用于定制gcc的行为。gcc根据选项的规则将输入文件编译生成适当的输出文件。

gcc的选项非常多，常用的选项，它们大致可以分为以下几类 。并且使用一个例子来描述这些选项。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_01”目录下。

代码为如下：

main.c:

01 #include <stdio.h>

02

03 #define HUNDRED 100

04

05 int main()

06 {

07 printf("%d ask\n",HUNDRED);

08 return 0;

09 }

过程控制选项

过程控制选项用于控制gcc的编译过程。无过程控制选项时，gcc将默认执行全部编译过程，产生可执行代码。常用的过程控制选项有：

（1）预处理选项(-E)

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。使用例子如下：

$ gcc -E main.c -o main.i

运行结果，生成main.i，main.i的内容（由于头文件展开内容过多，我将截取部分关键代码）：

extern int ftrylockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_)) ;

extern void funlockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_));

# 942 "/usr/include/stdio.h" 3 4

# 2 "main.c" 2

# 5 "main.c"

int main()

{

printf("%d ask\n",100);

return 0;

}

你会发现头文件被展开和printf函数中调用HUNDRED这个宏被展开。

（2）编译选项(-S)

编译就是把C/C++代码(比如上述的“.i”文件)“翻译”成汇编代码。使用例子如下：

$ gcc -S main.c -o main.s

运行结果，生成main.s，main.s的内容：

1 .file "main.c"

2 .text

3 .section .rodata

4 .LC0:

5 .string "%d ask\n"

6 .text

7 .globl main

8 .type main, @function

9 main:

10 .LFB0:

11 .cfi\_startproc

12 pushq %rbp

13 .cfi\_def\_cfa\_offset 16

14 .cfi\_offset 6, -16

15 movq %rsp, %rbp

16 .cfi\_def\_cfa\_register 6

17 movl $100, %esi

18 leaq .LC0(%rip), %rdi

19 movl $0, %eax

20 call printf@PLT

21 movl $0, %eax

22 popq %rbp

23 .cfi\_def\_cfa 7, 8

24 ret

25 .cfi\_endproc

26 .LFE0:

27 .size main, .-main

28 .ident "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"

29 .section .note.GNU-stack,"",@progbits

（3）汇编选项(-c)

汇编就是将上述的“.s”文件汇编代码翻译成符合一定格式的机器代码，在Linux系统上一般表现为ELF目标文件(OBJ文件)

$ gcc -c main.c -o main.o

运行结果，生成main.o(将源文件转为一定格式的机器代码)。

输出选项

输出选项用于指定gcc的输出特性等，常用的选项有：

（1）输出目标选项（-o *filename*）

-o选项指定生成文件的文件名为*filename*。使用例子如下

$ gcc main.c -o main

运行结果，生成可执行程序main，如下：

$ ls

main.c main

$ ./main

$ 100 ask

其中，如果无此选项时使用默认的文件名，各编译阶段有各自的默认文件名，可执行文件的默认名为a.out。使用例子如下：

$ gcc main.c

运行结果，生成可执行文件a.out，如下：

$ ls

a.out main.c

$ ./a.out

$ 100 ask

（2）输出所有警告选项（-Wall）

显示所有的警告信息，而不是只显示默认类型的警告。建议使用。我们把上面的main.c稍微修改一下。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_02”目录下。

main.c:

01 #include <stdio.h>

02

03 #define HUNDRED 100

04

05 int main()

06 {

07 int a = 0;

08 printf("%d ask\n",HUNDRED);

09 return 0;

10 }

编译不添加-Wall选项编译，没有任何警告信息，编译结果如下：

$ gcc main.c -o main.c

编译添加-Wall选项编译，现实所有警告信息，编译结果如下：

$ gcc main.c -Wall -o main.c

main.c: In function ‘main’:

main.c:7:6: warning: unused variable ‘a’ [-Wunused-variable]

int a=0;

^

头文件选项

头文件选项：-I*dirname*。

将*dirname*目录加入到头文件搜索目录列表中。当gcc在默认的路径中没有找到头文件时，就到本选项指定的目录中去找。

在上面的例子中创建一个inc目录，并在里面创建一个头文件test.h。

然后main.c里面增加#include“test.h”。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_03”目录下。

目录结构如下：

$ tree

.

├── inc

│   └── test.h

└── main.c

1 directory, 2 files

$

test.h代码如下:

01 #ifndef \_\_TEST\_H

02 #define \_\_TEST\_H

03 /\*

04 code

05 \*/

06 #endif

运行结果，这样就可以引用制定文件下的目录的头文件，如下：

$ gcc main.c -I inc -o main

如果不添加头文件选项，编译运行结果，如下：

$ gcc main.c -o main

main.c:2:18: fatal error: test.h: No such file or directory

compilation terminated.

会产生错误提示，无法找到test.h头文件。

链接库选项

（详细使用方法查看下一节：深入讲解GCC链接过程）

1. 添加库文件搜索目录（-L*dirname*）

将*dirname*目录加入到库文件的搜索目录列表中。

2. 加载库名选项（-l*name*）

加载名为lib*name*.a或lib*name*.so的函数库。例如：-lm表示链接名为libm.so的函数库。

3. 静态库选项（-static）

使用静态库。

注意：在命令行中，静态库必须放在目标文件之后。

比如：

gcc test.cpp -o test libexample.a -static

代码优化选项

gcc提供几种不同级别的代码优化方案，用“-O*level”*选项表示。level取值可以是0、1、2、3和s。

默认0级，即不进行优化。典型的优化选项：

（1）-O或-O1：基本优化，使代码执行的更快。

（2）-O2：产生尽可能小和快的代码。如无特殊要求，不建议使用O2以上的优化。

（3）-Os：生成最小的可执行文件，适合用于嵌入式软件。

调试选项及调试示例

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_02”目录下。

gcc支持数种调试选项：

-g：产生能被GDB调试器使用的调试信息。

举个例子，首先需要在编译时加上“-g”选项，操作步骤如下：

$ gcc main.c -g -o main

GDB调试示例:

（1）run命令

调试运行，使用run命令开始执行被调试的程序，run命令的格式：

run [运行参数]

$ gdb -q main <---进入调试程序

Reading symbols from output...done.

(gdb) run <---开始执行程序

Starting program: /home/100ask/makefile/

100 ask

[Inferior 1 (process 7425) exited normally]

(gdb)

（2）list命令

列出源代码，使用list命令来查看源程序以及行号信息，list命令的格式：

list [行号]

(gdb) list 1 <---列出第一行附近的源码，每次10行

#include <stdio.h>

#define HUNDRED 100

int main()

{

int a = 100;

printf("%d ask\n",HUNDRED);

return 0;

(gdb) <Enter> <---按Enter键，列出下10行源码

}

(gdb)

（3）设置断点

1）break命令，设置断点命令，break命令的格式： break <行号> | <函数名>

(gdb) break 7

Breakpoint 1 at 0x40052e: file main.c, line 7.

(gdb)

2）info break命令，查 看断点命令：

(gdb) info break

Num Type Disp Enb Address What

1 breakpoint keep y 0x000000000040052e in main at main.c:7

(gdb)

3）delete breakpoint命令，删除断点命令， delete breakpoint命令的格式： delete breakpoint <断点号>

(gdb) delete breakpoint 1

(gdb) info break

No breakpoints or watchpoints.

(gdb)

（4）跟踪运行结果

1）print命令，显示变量的值，print命令的格式：print[/格式] <表达式>

2）display命令，设置自动现实命令，display命令的格式： display <表达式>

3）step和 next命令，单步执行命令，step和next命令的格式：step <行号> 或 next <行号>

4）continue命令，继续执行命令。

(gdb) break 7

Breakpoint 1 at 0x40052e: file main.c, line 7.

(gdb) break 9

Breakpoint 2 at 0x400535: file main.c, line 9.

(gdb) run

Starting program:/home/100ask/makefile/

Breakpoint 1, main () at main.c:7

7 int a = 100;

(gdb) continue

Continuing.

Breakpoint 2, main () at main.c:9

9 printf("%d ask\n",HUNDRED);

(gdb) print a

$1 = 100

(gdb)

#### 6.2.3 编译错误警告

在写代码的时候，其实应该养成一个好的习惯就是任何的警告错误，我们都不要错过，

编译错误是必须要解决的，否则无法生成目标文件。但是即使有编译警告，也可以生成目标文件。所以编译警告往往会被人忽略。但是有时候，编译警告提示的信息你必须去修改，否则会影响程序运行。接下来我们来简单分析一下gcc的编译警告如何处理，例子如下：

main.c

01 #include <stdio.h>

02 #include "hander.h"

03

04 int main()

05 {

06 float a = 0.0;

07 int b = a;

08 char c = 'a'

09

10 printf("100ask: \n",a);

11

12 return 0;

13 }

上面文件中有三处错误：

第2行：包含了一个不存在的头文件。

第8行：语句后面没有加分号。

第10行：书写格式错误，变量a没有对应的输出格式。

我们对上面的文件进行编译，还记得上面讲的编译警告选项吗？在编译的时候加上它（-Wall），如下：

$ gcc main.c -Wall -o output

main.c: In function ‘main’:

main.c:2:20: fatal error: hander.h: No such file or directory

compilation terminated.

错误警告信息分析：在展开第二行的hander.h头文件的时候，产生编译错误，没有hander.h文件或者目录。接着我们把hander.h头文件去掉，再编译一次：

$ gcc -Wall main.c -o output

main.c: In function ‘main’:

main.c:10:2: error: expected ‘,’ or ‘;’ before ‘printf’

printf("100ask: \n",a);

^

main.c:8:7: warning: unused variable ‘c’ [-Wunused-variable]

char c = 'a'

^

main.c:7:6: warning: unused variable ‘b’ [-Wunused-variable]

int b = a;

^

错误警告信息分析：有一个错误和两个警告。一个错误是指第10行prntf之前缺少分号。两个警告是指第7行和第8行的变量没有使用。那么我继续解决错误信息和警告，将两个警告的变量删除和printf前添加分号，然后继续编译，如下：

$ gcc -Wall main.c -o output

main.c: In function ‘main’:

main.c:8:9: warning: too many arguments for format [-Wformat-extra-args]

printf("100ask: \n",a);

^

错误警告信息分析：还是有警告信息，该警告指的是printf中的格式参数太多，也就是没有添加变量a的输出格式，继续解决错误信息和警告，添加变量a的输出格式，然后继续编译，如下：

$ gcc -Wall main.c -o output

$ tree

.

├── main.c

└── output

最终编译成功，输出目标文件。

如果你忽略警告信息不去修改printf语句，程序无法输出你预期的结果。

### 6.3 深入讲解GCC链接过程

你会发现，可执行文件文件会比源代码大了。这是因为编译的最后一步是链接，它会解析代码中的外部应用，然后将你自己的OBJ文件，系统库的OBJ文件，库文件等等都链接起来。

我们用一个例子来说明上面描述。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_01”目录下。

执行如下命令：

$ gcc main.c -c

$ gcc -o output main.o

$ gcc -o output\_static main.o --static

$ ls -alh

drwxrwxr-x 2 tym tym 4.0K 2月 20 07:27 .

drwxrwxr-x 6 tym tym 4.0K 2月 20 07:25 ..

-rw-rw-r-- 1 tym tym 96 2月 20 07:25 main.c

-rw-rw-r-- 1 tym tym 1.5K 2月 20 07:26 main.o

-rwxrwxr-x 1 tym tym 8.5K 2月 20 07:27 output

-rwxrwxr-x 1 tym tym 892K 2月 20 07:27 output\_static

从上面的例子可以看出output\_static比output大很多。

#### 6.3.1 动态链接库和静态链接库使用例程

静态库和动态库，是根据链接时期的不同来划分。

静态库：在链接阶段，所用的库就被加进可执行程序里了。静态链接生成的可执行文件，已经内嵌了所有的库，可以独立运行。链接静态库从某种意义上来说是一种复制粘贴，被链接后库就直接嵌入可执行程序中了。如果有多个程序都用到这些库，并且都使用静态 接，那么系统里空间就有很大的浪费，而且一旦发现系统中有bug，就必须把所有程序都重新编译、重新链接，十分麻烦。静态库是不是一无是处了呢？不是的，如果代码在其他系统上运行，且没有相应的库时，解决办法就是使用静态库。而且由于动态库是在程序运行的时候被链接，因此动态库的运行速度比较慢。

动态库：在程序执行的时候，才把所需要的库跟程序链接在一起。多个程序可以合用一份动态库，节省存储空间。如果发现bug或者是要升级，只要用新的库把原来的替换掉就可以了。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_gcc\_04”目录下。

下面我们创建三个文件main.c，add.c，add.h，讲解静态库链接和动态库链接，如下：

main.c:

#include <stdio.h>

#include "add.h"

int main(int argc, char \*argv[])

{

printf("%d\n",add(10, 10));

printf("%d\n",add(20, 20));

return 0;

}

add.c:

#include "add.h"

int add(int a, int b)

{

return a + b;

}

add.h:

#ifndef \_\_ADD\_H

#define \_\_ADD\_H

int add(int a, int b);

#endif

静态库链接

静态库名字一般为“libxxx.a”。利用静态库编译生成的可执行文件比较大，因为整个函数库的所有数据都被整合进了可执行文件中。

**优点：**

1.不需要外部函数库支持。

2.加载速度快。

**缺点：**

1.静态库升级时，程序需要重新编译。

2.多个程序调用相同库，静态库会重复调入内存，造成内存的浪费。

静态库的制作，如下：

$ gcc add.c -o add.o -c

$ ar -rc libadd.a add.o

静态库的使用，例子如下：

$ gcc main.c -o output -ladd -L.

解析一下，“-L”用来表示库在哪里，上例中库在当前目录里(“.”表示当前目录)，“-l”用来表示库的名字(-ladd表示库libadd)。

运行结果：

$ ./output

20

40

动态库链接

动态库名字一般为“libxxx.so”，又称共享库。动态库在编译的时候没有被编译进可执行文件，所以可执行文件比较小。程序运行时，操作系统帮我们找到库，并跟程序链接起来。

**优点：**多个程序可以使用同一个动态库，节省内存。

**缺点：**加载速度慢。

动态库的制作，如下：

$ gcc -shared -fPIC lib.c -o libtest.so

$ sudo cp libtest.so /usr/lib/

动态库的使用，如下：

$ gcc main.c -L. -ltest -o output

链接时指定库在当前目录，但是运行时用的是/usr/lib或/lib目录下的库。

运行结果：

$ ./output

20

40

### 6.4 Makefile的引入及规则

#### 6.4.1 为什么需要Makefile?

在上一章节的例子中，我们都是在终端执行gcc命令来完成源文件的编译。感觉挺方便的，这是因为工程中的源文件只有一两个，在终端直接执行编译命令，确实快捷方便。

但是现在一些项目工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，如果仍然在终端输入这些命令来编译，那显然不切实际，开发效率极低。

我们需要一个工具来管理这些编译过程，这就是“make”。

make是一个应用程序，它根据Makefile来做事。Makefile负责管理整个编译流程：要编译哪些文件？怎么编译这些文件？怎么把它们链接成一个可执行程序。Makefile定义了一系列的规则来实现这些管理。

#### 6.4.2 Makefile的引入

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_Makefile\_01”目录下。

Makefile的引入是为了简化我们编译流程，提高我们的开发进度。下面我们用一个例子来说明Makefile如何简化我们的编译流程。我们创建一个工程内容分别main.c，sub.c，sub.h，add.c，add.h五个文件。sub.c负责计算两个数减法运算，add.c负责计算两个数加法运算，然后编译出可执行文件。其源文件内容如下：

main.c：

#include <stdio.h>

#include "add.h"

#include "sub.h"

int main()

{

printf("100 ask, add:%d\n", add(10, 10));

printf("100 ask, sub:%d\n", sub(20, 10));

return 0;

}

add.c：

#include "add.h"

int add(int a, int b)

{

return a + b;

}

add.h：

#ifndef \_\_ADD\_H

#define \_\_ADD\_H

int add(int a, int b);

#endif

sub.c：

#include "sub.h"

int sub(int a, int b)

{

return a - b;

}

sub.h：

#ifndef \_\_SUB\_H

#define \_\_SUB\_H

int sub(int a, int b);

#endif

我们使用gcc对上面工程进行编译及生成可执行程序，在终端输入如下命令，如下：

$ gcc main.c sub.c add.c -o ouput

$ ls

add.c add.h main.c output sub.c sub.h

$ ./output

100 ask, add:20

100 ask, sub:10

上面的命令是通过gcc编译器对 main.c sub.c add.c这三个文件进行编译，及生成可执行程序output，并执行可执行文件产生结果。

从上面的例子看起来比较简单，一条命令完成三个源程序的编译并产生结果，这是因为目前只有三个源文件。

如果有上千个源文件，即使你只修改了其中一个文件，你执行这样的命令时，它会把所有的源文件都编译一次。这样消耗的时间是非常恐怖的。

我们想实现：哪个文件被修改了，只编译这个被修改的文件即可，其它没有修改的文件就不需要再次重新编译了。可以使用下列命令，先单独编译文件，再最后链接，如下：

$ gcc -c main.c

$ gcc -c sub.c

$ gcc -c add.c

$ gcc main.o sub.o add.o -o output

我们将上面一条命令变成了四条，分别编译出源文件的目标文件，最后再将所有的目标文件链接成可执行文件。

当其中一个源文件的内容发生了变化，我们只需要单独编译它，然后跟其他文件重新链接成可知执行文件，不再需要重新编译其他文件。

假如我们修改了add.c文件，只需要重新编译生成add.c的目标文件，然后再将所有的.o文件链接成可执行文件，如下：

$ gcc -c add.c

$ gcc main.o sub.o add.o -o output

这样的方式虽然可以节省时间，但是仍然存在几个问题，如下：

1）如果源文件的数目很多，那么我们需要花费大量的时间，敲命令执行。

2）如果源文件的数目很多，然后修改了很多文件，后面发现忘记修改了什么。

3）如果头文件的内容修改，替换，更换目录，所有依赖于这个头文件的源文件全部需要重新编译。

这些问题我们不可能一个一个去找和排查，引入Makefile可以解决上述问题。

我们先扔出一个Makefile，如下：

Makefile：

output: main.o add.o sub.o

gcc -o output main.o add.o sub.o

main.o: main.c

gcc -c main.c

add.o: add.c

gcc -c add.c

sub.o: sub.c

gcc -c sub.c

clean:

rm \*.o output

Makefile编写好后只需要执行make命令，就可以自动帮助我们编译工程。注意，make命令必须要在Makefile的当前目录执行，如下：

$ ls

add.c add.h main.c Makefile sub.c sub.h

$ make

gcc -c main.c

gcc -c add.c

gcc -c sub.c

gcc -o output main.o add.o sub.o

$ ls

add.c add.h add.o main.c main.o Makefile output sub.c sub.h sub.o

通过make命令就可以生成相对应的目标文件.o和可执行文件。

如果我们再次使用make命令编译，如下，它说你的程序已经是最新的了，不需要再做什么事情：

$ make

make: 'output' is up to date.

我们可以修改一下add.c，然后再执行make，如下：

$ make

gcc -c add.c

gcc -o output main.o add.o sub.o

会发现，它重新编译了add.c，并且重新生成了可执行程序。

通过上述例子，Makefile将我们上面的三个问题都解决了，无需手工输入复杂的命令，只编译修改过的文件。在一个很庞大的工程中，只有第一次编译时间比较长，第二次编译时会大大缩短时间，节省了我们的开发周期。

下面我们来学习Makefile的知识。

#### 6.4.3 Makefile的规则

命名规则：

一般来说将Makefile文件取名为“Makefile”或“makefile”都可以，惯例是使用首字母大写的“Makefile”。也可以使用其他名字，比如makefile.linux，但你需要用“-f”参数指定，示例如下：

make -f makefile.linux

基本语法规则：

目标（target）：依赖（prerequisites）

[Tab]命令（command）

1）target：需要生成的目标文件

2）prerequisites：生成该target所依赖的一些文件

3）command：生成该目标需要执行的命令

三者的关系：target依赖于 prerequisites中的文件，其生成规则定义在command中。

举例，比如我们平时要编译一个文件：

$ gcc main.c -o main

换成Makefile的书写格式如下，，：

01 main:main.c

02 gcc main.c -o main

第1行表示main这个可执行程序依赖于main.c，第2行表示需要用“gcc main.c -o main”这个命令来生成main。

**注意：**Makefile文件里的命令必须要使用**Tab，**不能使用空格。

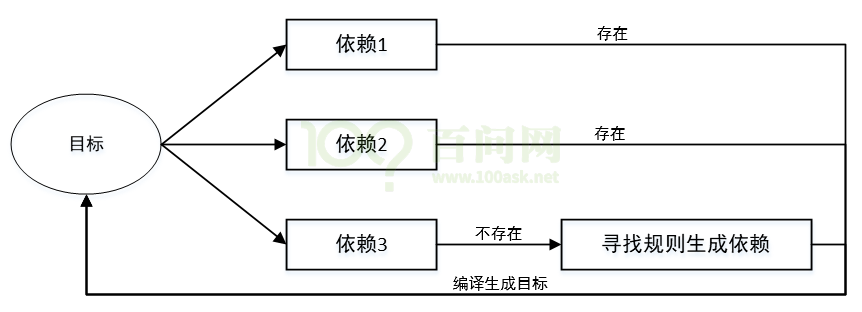
目标生成规则：

**目标生成：**

1）检查规则中的依赖文件是否存在。

2）若依赖文件不存在，则寻找是否有规则用来生成该依赖文件。

目标生成流程，如下：

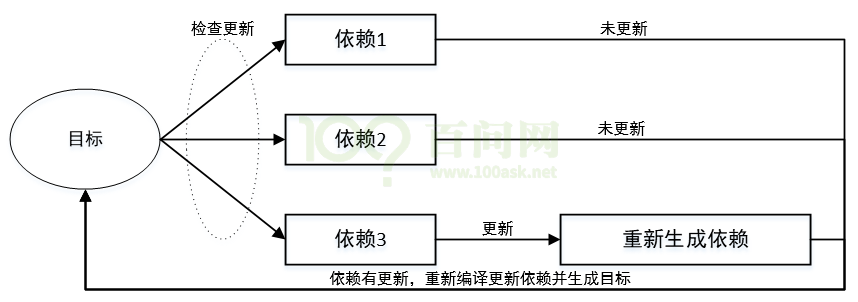


**目标更新：**

1）检查目标的所有依赖，任何一个依赖有更新时，就要重新生成目标。

2）目标文件比依赖文件更新时间晚，则需要更新。

目标更新流程，如下：



我们使用上面的例子，Makefile内容如下：

output: main.o add.o sub.o

gcc -o output main.o add.o sub.o

main.o: main.c

gcc -c main.c

add.o: add.c

gcc -c add.c

sub.o: sub.c

gcc -c sub.c

clean:

rm \*.o output

编译执行：

$ make

gcc -c main.c

gcc -c add.c

gcc -c sub.c

gcc -o output main.o add.o sub.o

make命令会检测寻找目标的依赖是否存在，不存在，则会寻找生成依赖的命令。

1). output依赖于“main.o add.o sub.o”，这三个文件都没有，分别去生成它们。怎么生成？继续寻找规则，发现main.o依赖于main.c，于是使用“gcc -c main.c”来生成；其他两个文件类似处理。

2). 现在output的依赖文件都有了，但是output文件还没有，所以使用“gcc -o output main.o add.o sub.o”来生成output文件。

当我们修改某一个文件时，比如之修改add.c文件，然后重新make，如下：

$ make

gcc -c add.c

gcc -o output main.o add.o sub.o

make命令还是根据Makefile来检查、执行：

1). output依赖于“main.o add.o sub.o”，这三个文件都有了，main.o依赖于main.c，main.c没改过，所以不需要重新生成main.o；但是add.o的依赖文件add.c被改过，需要重新生成add.o，使用命令“gcc -c add.c”；sub.o跟main.o类似，也不需要重新生成。

2). 现在output、它的三个依赖文件都有了，但是其中一个依赖文件add.o比output文件新，所以使用“gcc -o output main.o add.o sub.o”来生成output文件。

### 6.5 Makefile的语法

#### 6.5.1变量的定义及取值

Makefile也支持变量定义，变量的定义也让的我们的Makefile更加简化，可复用。

**变量的定义：**一般采用大写字母，赋值方式像C语言的赋值方式一样，如下：

DIR = ./100ask/

**变量取值：**使用括号将变量括起来再加美元符，如下：

FOO = $(DIR)

变量可让Makefile简化可复用，上面个的Makefile文件，内容如下：

output: main.o add.o sub.o

gcc -o output main.o add.o sub.o

main.o: main.c

gcc -c main.c

add.o: add.c

gcc -c add.c

sub.o: sub.c

gcc -c sub.c

clean:

rm \*.o output

我们可以将其优化，如下：

#Makefile变量定义

OBJ = main.o add.o sub.o

output: $(OBJ)

gcc -o output $(OBJ)

main.o: main.c

gcc -c main.c

add.o: add.c

gcc -c add.c

sub.o: sub.c

gcc -c sub.c

clean:

rm $(OBJ) output

我们分析一下上面简化过的Makefile，第一行是注释，Makefile的注释采用‘#’，而且不支持像C语言中的多行注释。第二行我们定义了变量OBJ，并赋值字符串”main.o，add.o，sub.o“。其中第三，四，十三行，使用这个变量。这样用到用一个字符串的地方直接调用这个变量，无需重复写一大段字符串。

Makefile除了使用‘=’进行赋值，还有其他赋值方式，比如‘:=’和‘?=’，接下来我们来对比一下这几种的区别：

赋值符‘=’

我们使用一个例子来说明赋值符‘=’的用法。Makefile内容如下：

01 PARA = 100

02 CURPARA = $(PARA)

03 PARA = ask

04

05 print:

06 @echo $(CURPARA)

分析代码：第一行定义变量PARA，并赋值为“100”，第二行定义变量CURPARA，并赋值引用变量PARA，此时CURPARA的值和PARA的值是一样的，第三行，将变量PARA的变量修改为“ask”。第六行输出CURPARA的值，echo前面增加@符号，代表只显示命令的结果，不显示命令本身。

通过命令“make print”执行Makefile，如下：

$ make print

ask

从结果上看，变量CURPARA的值并不是“100”，而是PARA的最后一次赋值。使用赋值符“=”设置的变量，它的值在运行时才能确定，这称为“延时变量”。

其实可以理解为在C语言中，定义一个指针变量指向一个变量的地址。如下：

01 int a = 10;

02 int \*b = &a;

03 a=20;

赋值符‘:=’

我们使用一个例子来说明赋值符‘:=’的用法。Makefile内容如下：

01 PARA = 100

02 CURPARA := $(PARA)

03 PARA = ask

04

05 print:

06 @echo $(CURPARA)

代码分析：我们见上面的Makefile的第二行的“=”替换成“:=”，重新编译，如下：

$ make print

100

$

从结果上看，变量CURPARA的值为“100”。使用“:=”设置的变量被称为“即时变量”，在赋值时就确定了它的值。

赋值符‘?=’

我们用两个Makefile来说明赋值符“?=”的用法。如下：

第一个Makefile：

PARA = 100

PARA ?= ask

print:

@echo $(PARA)

编译结果：

$ make print

100

$

第二个Makefile:

PARA ?= ask

print:

@echo $(PARA)

编译结果：

$ make print

ask

$

上面的例子说明，使用“?=”给变量设置值时，如果这个变量之前没有被设置过，那么“?=”才会起效果；如果曾经设置过这个变量，那么“?=”不会起效果。

赋值符‘+=’

Makefile 中的变量是字符串，有时候我们需要给前面已经定义好的变量添加一些字符串进去,此时就要使用到符号“+=”，比如如下：

01 OBJ = main.o add.o

02 OBJ += sub.o

这样的结果是OBJ的值为：”main.o，add.o，sub.o“。说明“+=”用作与变量的追加。

#### 6.5.2 系统自带变量

系统自定义了一些变量，通常都是大写，比如CC，PWD，CLFAG等等，有些有默认值，有些没有，比如以下几种，如下：

1）CPPFLAGS：预处理器需要的选项，如：-l

2）CFLAGS：编译的时候使用的参数，-Wall -g -c

3）LDFLAGS：链接库使用的选项，-L -l

其中：默认值可以被修改，比如CC默认值是cc，但可以修改为gcc：CC=gcc

使用的例子，如下：

01 OBJ = main.o add.o sub.o

02 output: $(OBJ)

03 gcc -o output $(OBJ)

04 main.o: main.c

05 gcc -c main.c

06 add.o: add.c

07 gcc -c add.c

08 sub.o: sub.c

09 gcc -c sub.c

10

11 clean:

12 rm $(OBJ) output

使用系统自带变量，如下：

01 CC = gcc

02 OBJ = main.o add.o sub.o

03 output: $(OBJ)

04 $(CC) -o output $(OBJ)

05 main.o: main.c

06 $(CC) -c main.c

07 add.o: add.c

08 $(CC) -c add.c

09 sub.o: sub.c

10 $(CC) -c sub.c

11

12 clean:

13 rm $(OBJ) output

在上面例子中，系统变量CC不改变默认值，也同样可以编译，修改的目的是为了明确使用gcc编译。

#### 6.5.3 自动变量

Makefile的语法提供一些自动变量，这些变量可以让我们更加快速地完成Makefile的编写，其中自动变量只能在规则中的命令使用，常用的自动变量如下：

1）$@：规则中的目标

2）$<：规则中的第一个依赖文件

3）$^：规则中的所有依赖文件

我们上面的例子继续完善，修改为采用自动变量的格式，如下:

01 CC = gcc

02 OBJ = main.o add.o sub.o

03 output: $(OBJ)

04 $(CC) -o $@ $^

05 main.o: main.c

06 $(CC) -c $<

07 add.o: add.c

08 $(CC) -c $<

09 sub.o: sub.c

10 $(CC) -c $<

11

12 clean:

13 rm $(OBJ) output

其中：第4行$^表示变量OBJ的值，即main.o add.o sub.o，第四，第六，第八行的$<分别表示main.c add.c sub.c。$@表示output。

#### 6.5.4 模式规则

模式规则实在目标及依赖中使用%来匹配对应的文件，我们依旧使用上面的例子，采用模式规则格式，如下：

01 CC = gcc

02 OBJ = main.o add.o sub.o

03 output: $(OBJ)

04 $(CC) -o $@ $^

05 %.o: %.c

06 $(CC) -c $<

07

08 clean:

09 rm $(OBJ) output

其中：第五行%.o: %.表示如下。

1.main.o由main.c生成

2.add.o 由 add.c生成

3.sub.o 由 sub.c生成

#### 6.5.5 伪目标

在前面的例子中，我们直接执行“make”命令，它的目的是去执行第1个规则，这跟执行“make output”的效果是一样的。在这里，“output”既是规则的目标，也是一个实际的文件。

而伪目标是什么呢？

对于以前的例子，先执行make命令，然后再执行“make clean”命令，如下：

$make

gcc -c main.c

gcc -c add.c

gcc -c sub.c

gcc -o output main.o add.o sub.o

$make clean

rm \*.o output

一切正常！接着我们做个手脚，在Makefile目录下创建一个clean的文件，然后依旧执行make和make clean，如下：

$touch clean

$make

gcc -c main.c

gcc -c add.c

gcc -c sub.c

gcc -o output main.o add.o sub.o

$make clean

make: 'clean' is up to date.

为什么“make clean”时命令没有被执行？

因为已经有名为clean的文件了，并且它的依赖是空的，执行规则的条件没满足。

伪目标就是为了解决这个问题，我们在clean前面增加“.PHONY:clean”，如下：

01 CC = gcc

02 OBJ = main.o add.o sub.o

03 output: $(OBJ)

04 $(CC) -o $@ $^

05 %.o: %.c

06 $(CC) -c $<

07

08 .PHONY:clean

09 clean:

10 rm $(OBJ) output

运行结果：

$make

gcc -c main.c

gcc -c add.c

gcc -c sub.c

gcc -o output main.o add.o sub.o

$make clean

rm \*.o output

当一个目标被声明为伪目标后，make在执行规则时就会默认它满足执行条件。这样就提高了make的执行效率，也不用担心由于目标和文件名重名了。

伪目标的两大好处：

1.避免只执行命令的目标和工作目录下的实际文件出现名字冲突。

2.提高执行Makefile时的效率

#### 6.5.6 Makefile函数

Makefile提供了大量的函数，函数调用的格式如下：

$(function arguments)

这里`function’是函数名，`arguments’是该函数的参数。参数和函数名之间是用空格或Tab隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_Makefile\_02”目录下。

我们经常使用的函数主要有两个(wildcard，patsubst)，先把它们单独拎出来讲讲。

创建一个文件夹src，在里下面创建两个文件，100.c，ask.c。如下：

.

├── Makefile

└── src

├── 100.c

└── ask.c

wildcard函数

用于查找指定目录下指定类型的文件，函数参数：目录+文件类型，如下：

$(wildcard 指定文件类型)

其中，指定文件类型，如果不写路径，则默认为当前目录查找，例子如下：

01 SRC = $(wildcard ./src/\*.c)

02

03 print:

04 @echo $(SRC)

执行命令make，结果如下：

$ make

./src/ask.c ./src/100.c

其中，这条规则表示：找到目录./src下所有后缀为.c的文件，并赋值给变量SRC。命令执行完，SRC变量的值：./src/ask.c ./src/100.c

patsubst函数

用于匹配替换。函数参数：原模式+目标模式+文件列表，如下:

$( patsubst 原模式, 目标模式, 文件列表)

其中，从文件列表中查找出符合原模式文件类型的文件，然后一一替换成目标模式。举例：将./src目录下的.c结尾的文件，替换成.o文件，并赋值给obj。如下：

SRC = $(wildcard ./src/\*.c)

OBJ = $(patsubst %.c, %.o, $(SRC))

print:

@echo $(OBJ)

执行命令make，结果如下：

$ make

./src/ask.o ./src/100.o

其中，这条规则表示：把变量中所有后缀为.c的文件替换为.o。 命令执行完，OBJ变量的值：./src/ask.o ./src/100.o

字符串替换和分析函数

（1）$(subst from,to,text)

在文本`text’中使用`to’替换每一处`from’。

比如：

$(subst ee,EE,feet on the street)

结果为‘fEEt on the strEEt’。

（2）$(patsubst pattern,replacement,text) ，前面讲过。

寻找`text’中符合格式`pattern’的字，用`replacement’替换它们。`pattern’和`replacement’中可以使用通配符。

比如：

$(patsubst %.c,%.o,x.c.c bar.c)

结果为：`x.c.o bar.o’。

（3）$(strip string)

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。

比如：

$(strip a b c )

结果为`a b c’。

（4）$(findstring find,in)

在字符串`in’中搜寻`find’，如果找到，则返回值是`find’，否则返回值为空。

比如：

$(findstring a,a b c)

$(findstring a,b c)

将分别产生值`a’和`’(空字符串)。

（5）$(filter pattern...,text)

返回在`text’中由空格隔开且匹配格式`pattern...’的字，去除不符合格式`pattern...’的字。

比如：

$(filter %.c %.s,foo.c bar.c baz.s ugh.h)

结果为`foo.c bar.c baz.s’。

（6）$(filter-out pattern...,text)

返回在`text’中由空格隔开且不匹配格式`pattern...’的字，去除符合格式`pattern...’的字。它是函数filter的反函数。

比如：

$(filter %.c %.s,foo.c bar.c baz.s ugh.h)

结果为`ugh.h’。

（7）$(sort list)

将‘list’中的字按字母顺序排序，并去掉重复的字。输出由单个空格隔开的字的列表。

比如：

$(sort foo bar lose)

返回值是‘bar foo lose’。

文件名函数

（1）$(dir names...)

抽取‘names...’中每一个文件名的路径部分，文件名的路径部分包括从文件名的首字符到最后一个斜杠(含斜杠)之前的一切字符。

比如：

$(dir src/foo.c hacks)

结果为‘src/ ./’。

（2）$(notdir names...)

抽取‘names...’中每一个文件名中除路径部分外一切字符（真正的文件名）。

比如：

$(notdir src/foo.c hacks)

结果为‘foo.c hacks’。

（3）$(suffix names...)

抽取‘names...’中每一个文件名的后缀。

比如：

$(suffix src/foo.c src-1.0/bar.c hacks)

结果为‘.c .c’。

（4）$(basename names...)

抽取‘names...’中每一个文件名中除后缀外一切字符。

比如：

$(basename src/foo.c src-1.0/bar hacks)

结果为‘src/foo src-1.0/bar hacks’。

（5）$(addsuffix suffix,names...)

参数‘names...’是一系列的文件名，文件名之间用空格隔开；suffix是一个后缀名。将suffix(后缀)的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用单个空格隔开。

比如：

$(addsuffix .c,foo bar)

结果为‘foo.c bar.c’。

（6）$(addprefix prefix,names...)

参数‘names’是一系列的文件名，文件名之间用空格隔开；prefix是一个前缀名。将preffix(前缀)的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。

比如：

$(addprefix src/,foo bar)

结果为‘src/foo src/bar’。

（7）$(wildcard pattern) ，前面讲过

参数‘pattern’是一个文件名格式，包含有通配符(通配符和shell中的用法一样)。函数wildcard的结果是一列和格式匹配的且真实存在的文件的名称，文件名之间用一个空格隔开。

比如若当前目录下有文件1.c、2.c、1.h、2.h，则：

c\_src := $(wildcard \*.c)

结果为‘1.c 2.c’。

其他函数

（1）$(foreach var,list,text)

前两个参数，‘var’和‘list’将首先扩展，注意最后一个参数‘text’此时不扩展；接着，‘list’扩展所得的每个字，都赋给‘var’变量；然后‘text’引用该变量进行扩展，因此‘text’每次扩展都不相同。

函数的结果是由空格隔开的‘text’ 在‘list’中多次扩展后，得到的新‘list’，就是说：‘text’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数foreach的返回值。

下面是一个简单的例子，将变量‘files’的值设置为 ‘dirs’中的所有目录下的所有文件的列表：

dirs := a b c d

files := $(foreach dir,$(dirs),$(wildcard $(dir)/\*))

这里‘text’是‘$(wildcard $(dir)/\*)’，它的扩展过程如下：

⑤ 第一个赋给变量dir的值是`a’，扩展结果为‘$(wildcard a/\*)’；

⑥ 第二个赋给变量dir的值是`b’，扩展结果为‘$(wildcard b/\*)’；

⑦ 第三个赋给变量dir的值是`c’，扩展结果为‘$(wildcard c/\*)’；

⑧ 如此继续扩展。

这个例子和下面的例有共同的结果：

files := $(wildcard a/\* b/\* c/\* d/\*)

（2）$(if condition,then-part[,else-part])

首先把第一个参数‘condition’的前导空格、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件‘condition’为‘真’；如果扩展为空字符串，则条件‘condition’为‘假’。

如果条件‘condition’为‘真’,那么计算第二个参数‘then-part’的值，并将该值作为整个函数if的值。

如果条件‘condition’为‘假’,并且第三个参数存在，则计算第三个参数‘else-part’的值，并将该值作为整个函数if的值；如果第三个参数不存在，函数if将什么也不计算，返回空值。

注意：仅能计算‘then-part’和‘else-part’二者之一，不能同时计算。这样有可能产生副作用（例如函数shell的调用）。

（3）$(origin variable)

变量‘variable’是一个查询变量的名称，不是对该变量的引用。所以，不能采用‘$’和圆括号的格式书写该变量，当然，如果需要使用非常量的文件名，可以在文件名中使用变量引用。

函数origin的结果是一个字符串，该字符串变量是这样定义的：

‘undefined' ：如果变量‘variable’从没有定义；

‘default' ：变量‘variable’是缺省定义；

‘environment' ：变量‘variable’作为环境变量定义，选项‘-e’没有打开；

‘environment override' ：变量‘variable’作为环境变量定义，选项‘-e’已打开；

‘file' ：变量‘variable’在Makefile中定义；

‘command line' ：变量‘variable’在命令行中定义；

‘override' ：变量‘variable’在Makefile中用override指令定义；

‘automatic' ：变量‘variable’是自动变量

（4）$(shell command arguments)

函数shell是make与外部环境的通讯工具。函数shell的执行结果和在控制台上执行‘command arguments’的结果相似。不过如果‘command arguments’的结果含有换行符（和回车符），则在函数shell的返回结果中将把它们处理为单个空格，若返回结果最后是换行符（和回车符）则被去掉。

比如当前目录下有文件1.c、2.c、1.h、2.h，则：

c\_src := $(shell ls \*.c)

结果为‘1.c 2.c’。

### 6.6 Makefile实例

**代码**：GIT下载后在“10\_裸机开发/01\_100ASK\_IMX6ULL裸机程序/6\_Makefile与GCC/001\_Makefile\_03”目录下。

在上面的例子中，我们都是把头文件，源文件放在同一个文件里面，这样不好用于维护，所以我们将其分类，把它变得更加规范一下，把所有的头文件放在文件夹：inc，把所有的源文件放在文件夹：src。

代码目录如下：

$ tree

.

├── inc

│   ├── add.h

│   └── sub.h

├── Makefile

└── src

├── add.c

├── main.c

└── sub.c

其中Makefile的内容如下：

01 SOURCE = $(wildcard ./src/\*.c)

02 OBJECT = $(patsubst %.c, %.o, $(SOURCE))

03

04 INCLUEDS = -I ./inc

05

06 TARGET = 100ask

07 CC = gcc

08 CFLAGS = -Wall -g

09

10 $(TARGET): $(OBJECT)

11 @mkdir -p output/

12 $(CC) $^ $(CFLAGES) -o output/$(TARGET)

13

14 %.o: %.c

15 $(CC) $(INCLUEDS) $(CFLAGES) -c $< -o $@

16

17 .PHONY:clean

18 clean:

19 @rm -rf $(OBJECT) output/

分析：

行1：获取当前目录下src所有.c文件，并赋值给变量SOURCE。

行2：将./src目录下的.c结尾的文件，替换成.o文件，并赋值给变量OBJECT。

行4：通过-I选项指明头文件的目录，并赋值给变量INCLUDES。

行6：最终目标文件的名字100ask，赋值给TARGET。

行7：替换CC的默认之cc，改为gcc。

行8：将显示所有的警告信息选项和gdb调试选项赋值给变量CFLAGS。

行11：创建目录output，并且不再终端现实该条命令。

行12：编译生成可执行程序100ask，并将可执行程序生成到output目录

行15：将源文件生成对应的目标文件。

行17：伪目标，避免当前目录有同名的clean文件。

行19：用与执行命令make clean时执行的命令，删除编译过程生成的文件。

最后编译的结果，如下:

$ make

gcc -I ./inc -c src/main.c -o src/main.o

gcc -I ./inc -c src/add.c -o src/add.o

gcc -I ./inc -c src/sub.c -o src/sub.o

gcc src/main.o src/add.o src/sub.o -o output/100ask

$tree

.

├── inc

│   ├── add.h

│   └── sub.h

├── Makefile

├── output

│   └── 100ask

└── src

├── add.c

├── add.o

├── main.c

├── main.o

├── sub.c

└── sub.o

上面的Makefile文件算是比较完善了，不过项目开发中，代码需要不断的迭代，那么必须要有东西来记录它的变化，所以还需要对最终的可执行文件添加版本号，如下：

01 VERSION = 1.0.0

02 SOURCE = $(wildcard ./src/\*.c)

03 OBJECT = $(patsubst %.c, %.o, $(SOURCE))

04

05 INCLUEDS = -I ./inc

06

07 TARGET = 100ask

08 CC = gcc

09 CFLAGS = -Wall -g

10

11 $(TARGET): $(OBJECT)

12 @mkdir -p output/

13 $(CC) $^ $(CFLAGES) -o output/$(TARGET)\_$(VERSION)

14

15 %.o: %.c

16 $(CC) $(INCLUEDS) $(CFLAGES) -c $< -o $@

17

18 .PHONY:clean

19 clean:

20 @rm -rf $(OBJECT) output/

分析：

行1：将版本号赋值给变量VERSION。

行13：生成可执行文件的后缀添加版本号。

编译结果：

$ tree

.

├── inc

│   ├── add.h

│   └── sub.h

├── Makefile

├── output

│   └── 100ask\_1.0.0

└── src

├── add.c

├── add.o

├── main.c

├── main.o

├── sub.c

└── sub.o