写在前面的话

本节视频来自2018年为2440的裸机加强版视频，非常适合本教程。
要注意几点：

- 以前源码目录是`003_Makefile`，现在目录改为`04_2018_Makefile`
  - GIT仓库里：`01_all_series_quickstart\04_嵌入式Linux应用开发基础知识\source\04_2018_Makefile`
- 本节视频配套的文档，就是本文档，位于：
  - GIT仓库里：`01_all_series_quickstart\04_嵌入式Linux应用开发基础知识\doc_pic\04.2018_Makefile`

Makefile的语法

本节我们只是简单的讲解Makefile的语法，如果想比较深入

学习Makefile的话可以：

- a. 百度搜 "gnu make 于凤昌"。
- b. 查看官方文档: http://www.gnu.org/software/make/manual/

a. 通配符

假如一个目标文件所依赖的依赖文件很多，那样岂不是我们要写很多规则，这显然是不合乎常理的

我们可以使用通配符，来解决这些问题。

我们对上节程序进行修改代码如下：

```
test: a.o b.o
    gcc -o test $^

%.o : %.c
    gcc -c -o $@ $<
```

%.o：表示所用的.o文件

%.c：表示所有的.c文件

$\@：表示目标

$<：表示第1个依赖文件

$\^：表示所有依赖文件

我们来在该目录下增加一个 c.c 文件，代码如下：

```
#include <stdio.h>

void func_c()
{
    printf("This is C\n");
}
```

然后在main函数中调用修改Makefile，修改后的代码如下：

```
test: a.o b.o c.o
    gcc -o test $^

%.o : %.c
    gcc -c -o $@ $<
```

执行：

```
make
```

结果：

```
gcc -c -o a.o a.c
gcc -c -o b.o b.c
gcc -c -o c.o c.c
gcc -o test a.o b.o c.o
```

运行：

```
./test
```

结果：

```
This is B
This is C
```

b. 假想目标: .PHONY

1.我们想清除文件，我们在Makefile的结尾添加如下代码就可以了：

```
clean:
    rm *.o test
```

*1）执行 make ： 生成第一个可执行文件。
*2）执行 make clean : 清除所有文件，即执行： rm *.o test。

make后面可以带上目标名，也可以不带，如果不带目标名的话它就想生成第一个规则里面的第一个目标。

2.使用Makefile

执行：**make [目标]** 也可以不跟目标名，若无目标默认第一个目标。我们直接执行make的时候，会在makefile里面找到第一个目标然后执行下面的指令生成第一个目标。当我们执行 make clean 的时候，就会在 Makefile 里面找到 clean 这个目标，然后执行里面的命令，这个写法有些问题，原因是我们的目录里面没有 clean 这个文件，这个规则执行的条件成立，他就会执行下面的命令来删除文件。

如果：该目录下面有名为clean文件怎么办呢?

我们在该目录下创建一个名为 “clean” 的文件，然后重新执行：make然后make clean，结果(会有下面的提示：):

```
make: \`clean' is up to date.
```

它根本没有执行我们的删除操作，这是为什么呢?

我们之前说，一个规则能过执行的条件:

*1)目标文件不存在
*2)依赖文件比目标新

现在我们的目录里面有名为“clean”的文件，目标文件是有的，并且没有

依赖文件，没有办法判断依赖文件的时间。这种写法会导致：有同名的"clean"文件时，就没有办法执行make clean操作。解决办法：我们需要把目标定义为假象目标，用关键子PHONY

```
.PHONY: clean //把clean定义为假象目标。他就不会判断名为“clean”的文件是否存在，
```

然后在Makfile结尾添加.PHONY: clean语句，重新执行：make clean，就会执行删除操作。

C. 变量

在makefile中有两种变量:

1), 简单变量(即使变量):

A := xxx # A的值即刻确定，在定义时即确定

对于即使变量使用 “:=” 表示，它的值在定义的时候已经被确定了

2）延时变量

B = xxx # B的值使用到时才确定

对于延时变量使用“=”表示。它只有在使用到的时候才确定，在定义/等于时并没有

确定下来。

想使用变量的时候使用“$”来引用，如果不想看到命令是，可以在命令的前面加上"\@"符号，就不会显示命令本身。当我们执行make命令的时候，make这个指令本身，会把整个Makefile读进去，进行全部分析，然后解析里面的变量。常用的变量的定义如下:

```
:= # 即时变量
= # 延时变量
?= # 延时变量，如果是第1次定义才起效，如果在前面该变量已定义则忽略这句
\+= # 附加，它是即时变量还是延时变量取决于前面的定义
?=: 如果这个变量在前面已经被定义了，这句话就会不会起效果，
```

实例:

```
A := $(C)
B = $(C)
C = abc

#D = 100ask
D ?= weidongshan

all:
    @echo A = $(A)
    @echo B = $(B)
    @echo D = $(D)

C += 123
```

执行：

```
make
```

结果：

```
A =
B = abc 123
D = weidongshan
```

分析：

1) A := $(C)：

A为即使变量，在定义时即确定，由于刚开始C的值为空，所以A的值也为空。

2) B = $(C)：
B为延时变量，只有使用到时它的值才确定，当执行make时，会解析Makefile里面的所用变量，所以先解析C= abc,然后解析C += 123，此时，C = abc 123，当执行：\@echo B = $(B) B的值为 abc 123。

3) D ?= weidongshan：

D变量在前面没有定义，所以D的值为weidongshan，如果在前面添加D = 100ask，最后D的值为100ask。

我们还可以通过命令行存入变量的值 例如：

执行：make D=123456 里面的 D ?= weidongshan 这句话就不起作用了。

结果：

```
A =
B = abc 123
D = 123456
```