工程管理器 make

当我们需要编译一个比较大的项目时，编译命令会变得越来越复杂，需要编译的文件越来越多。其次就是项目中并不是每一次编译都需要把所有文件都重新编译，比如没有被修改过的文件则不需要重新编译。工程管理器就帮助我们来优化这两个问题。

MakeFile就类似于make工程管理的工作的脚本。用来告诉工程管理器如何正确的编译我们的程序。

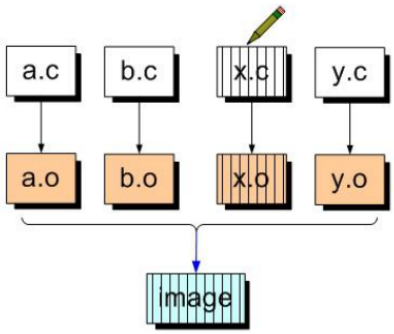


图1-38 修改了 image 所依赖的其中一个文件

依赖于目标的关系：

在MakeFile中依赖于目标是相互的，并不是绝对

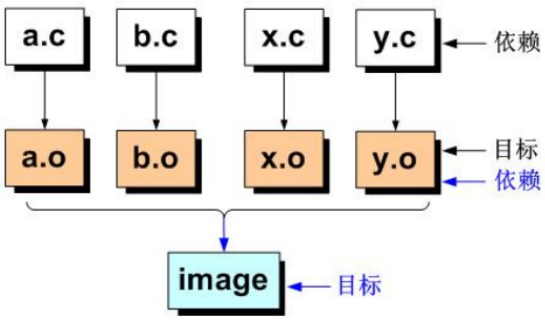


图 1-39 Makefile 眼中的目标和依赖

比如 a.c 是生成a.o的一个依赖文件， 对于a.o 则是a.c的目标，a.o 又是image的依赖。

在我们使用make 进行编译的时候，工程管理器则会根据依赖于目标的关系来检查它们之间时间戳关系，如果依赖有给你更新那么目标文件则需要执行。

安装make

```
sudo apt install make
```

第一个MakeFile

语法：

```
目标:依赖
    命令
```

注意：

目标必须存在

依赖可以没有

命令前面必须是一个制表符"TAB"

Makefile 文件的命名一般是 Makefile没有后缀也没有前缀，M

一下两行则称为一套规则:

当我们输入make 的时候：

1. 工程管理器，先会默认在当前路径下寻找一个名为Makefile的文件
2. 确认Makefile文件中的目标（最终）
3. 检查最终目标是否已经存在
    a. 如果存在则检查规则中是否存在依赖
        i. 如果规则中没有依赖，则不运行规则
        ii. 如果有依赖则检查依赖是否真实存在
            1. 有存在检查时间戳判断是否执行规则
            2. 不存在则之直接报错
    b. 如果不存在，则直接运行规则

注意：

如果规则中没有写依赖，则无论如何该规则该规则都会执行
如果目标已经存在，然后也没有写依赖则不执行该规则

```
$make
make: 'Even' is up to date.
```

如果目标文件已经存在，规则中有写依赖并且依赖文件比目标文件更新，则规则会被执行
注意制表符表示后面紧接着的是一个Shell 命令

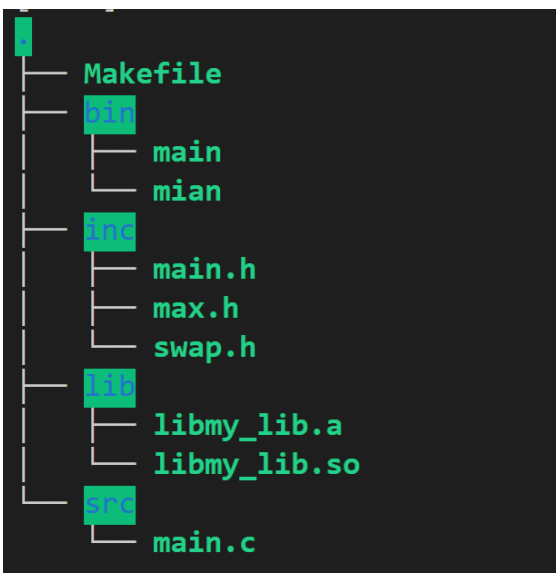
目标与依赖的互相嵌套（类似函数调用）

```
Even:Jacy
    @echo "Hello Makefile"

Jacy:ChuiHua
    @echo "Hello Even"

ChuiHua:
    @echo "Hello Jacy"

```

执行：

```
make   // 工程管理器把第一个目标当成最终目标 Even
make Jacy  // 告诉工程管理器 Jacy是我们需要的最终目标
```

如何使用刚才所学的去编译程序：



对于以上工程：

目标： ./bin/main

依赖： ./src/*.c

```
./bin/main:./src/*.c
    gcc  ./src/*.c  -o ./bin/main -I./inc -L./lib -lmy_lib
```

变量：

在Makefile 中变量属于弱类型，在Makefile中变量就是一个名字（像是C语言中的宏），代表一个文本字符串（变量的值），在Makefile的目标、依赖、命令中引用一个变量的地方。

在Makefile中变量的特征有以下几点：

1. 变量和函数的展开（除规则的命令行以外），是在make读取Makefile文件时进行的，这里的变量包括了使用“=”定义和使用指示符“define”定义的变量。

2. 变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表和所有我们能够想到的事物。

3. 变量名不能包括“:”、“#”、“=”、前置空白和尾空白的任何字符串。需要注意的是，尽管在GNU make中没有对变量的命名有其它的限制，但定义一个包含除字母、数字和下划线以外的变量的做法也是不可取的，因为除字母、数字和下划线以外的其它字符可能会在以后的make版本中被赋予特殊含义，并且这样命名的变量对于一些Shell来说不能作为环境变量使用。

4. 变量名是大小写敏感的。变量“foo”、“Foo”和“FOO”指的是三个不同的变量。Makefile传统做法是变量名是全采用大写的方式。推荐的做法是在对于内部定义的一般变量（例如：目标文件列表objects）使用小写方式，而对于一些参数列表（例如：编译选项CFLAGS）采用大写方式，这并不是要求的。但需要强调一点：对于一个工程，所Makefile中的变量命名应保持一种风格，否则会显得你是一个蹩脚的开发者（就像代码的变量命名风格一样），随时有被鄙视的危险。

5. 另外有一些变量名只包含了一个或者很少的几个特殊的字符（符号）。称它们为自动化变量。像“<”、“@”、“?”、“*”、“@D”、“%F”、“^D”等等，后面会详述之。

6. 变量的引用跟Shell脚本类似，使用美元符号和圆括号，比如有个变量叫A，那么对他的引用则是$(A)，有个自动化变量叫@，则对他的引用是$(@)，有个系统变量是CC则对其引用的格式是$(CC)。对于前面两个变量而言，他们都是单字符变量，因此对他们引用的括号可以省略，写成$A和$@。

1.自定义变量

顾名思义就是用户自己定义的变量

```
A = apple    # 定义并赋值变量
B = I love China
C = $(A) tree   #  $() 则是对某一个变量进行引用

Even:
    @echo $(A)
    @echo $(B)
    @echo $(C)
```

通过自定义变量来修改的Makefile 第二版本：

```
TAG=./bin/main
SRC=./src/*.c
CC=gcc
O=-o
CONFIG=-I./inc -L./lib -lmy_lib


$(TAG):$(SRC)
    $(CC)  $(SRC)  $(O) $(TAG) $(CONFIG)


clean:
    rm ./bin/*
```

2.系统预定义变量

此时的 CC 就不是 gcc 而是交叉工具链 arm-none-linux-gnueabi-gcc 了，很方便。常用的系统预定义变量，请看下表：

变量名	含义	备注
AR	函数库打包程序，可创建静态库.a文档。默认是“ar”。	无
AS	汇编程序。默认是“as”。	无
CC	C编译程序。默认是“cc”。	无
CXX	C++编译程序。默认是“g++”。	无
CPP	C程序的预处理器。默认是“$(CC) –E”。	无
RM	删除命令。默认是“rm –f”。	无
ARFLAGS	执行AR命令的命令行参数。默认值是“rv”。	无
ASFLAGS	汇编器AS的命令行参数（明确指定“.s”或“.S”文件时）。	无
CFLAGS	执行CC编译器的命令行参数（编译.c源文件的选项）。	无
CXXFLAGS	执行g++编译器的命令行参数（编译.cc源文件的选项）。	无

表 1-7 Makefile 预定义变量

3.自动化变量

<、@、?、#等等，这些特殊的变量之所以称为自动化变量，是因为他们的值会“自动地”发生变化

有关自动化变量的详细情况，见下表：

变量名	含义	备注
@	代表其所在规则的目标的完整名称	
%	代表其所在规则的静态库文件的一个成员名	
<	代表其所在规则的依赖列表的第一个文件的完整名称	
?	代表所有时间戳比目标文件新的依赖文件列表，用空格隔开	
^	代表其所在规则的依赖列表	同一文件不可重复
+	代表其所在规则的依赖列表	同一文件可重复，主要用在程序链接时，库的交叉引用场合。
*	在模式规则和静态模式规则中，代表茎	茎是目标模式中“%”所代表的部分（当文件名中存在目录时，茎也包含目录（斜杠之前）部分。

表1-8 Makefile自动化变量

变量名	含义	备注
@D	代表目标文件的目录部分（去掉目录部分的最后一个斜杠）	如果“$@”是“dir/foo.o”，那么“$(@D)”的值为“dir”。如果“$@”不存在斜杠，其值就是“.”（当前目录）。注意它和函数“dir”的区别
@F	目标文件的完整文件名中除目录以外的部分（实际文件名）	如果“$@”为“dir/foo.o”，那么“$(@F)”只就是“foo.o”。“$(@F)”等价于函数“$(notdir $@)”
*D	代表目标茎中的目录部分	
*F	代表目标茎中的文件名部分	
%D	当以如“archive(member)”形式静态库为目标时，表示库文件成员“member”名中的目录部分	仅对“archive(member)”形式的规则目标有效
%F	当以如“archive(member)”形式静态库为目标时，表示库文件成员“member”名中的文件名部分	仅对“archive(member)”形式的规则目标有效
<D	代表规则中第一个依赖文件的目录部分	
<F	代表规则中第一个依赖文件的文件名部分	
^D	代表所有依赖文件的目录部分	同一文件不可重复
^F	代表所有依赖文件的文件名部分	同一文件不可重复
+D	代表所有依赖文件的目录部分	同一文件可重复
+F	代表所有依赖文件的文件名部分	同一文件可重复
?D	代表被更新的依赖文件的目录部分。	
?F	代表被更新的依赖文件的文件名部分。	

表 1-9 Makefile 自动化变量的变种

Makefile 中定义的变量有以下几种不同的方式：

1，递归定义方式：

```
A = I love $(B)    # 在第一行使用到变量B但是还没有定义，以此管理器进行全文搜索找到B并引用
B = China
```

2，直接定义方式：

```
B = China
A := I love $(B)
```

此处，定义 A 时用的是所谓的“直接”定义方式，说白了就是如果其定义里出现有对其他变量的引用的话，只会其前面的语句进行搜寻（不包含自己所在的那一行），而不是搜寻整个文件，因此，如果此处将变量 A 和变量 B 的定义交换一个位置：

```
A := I love $(B)  # A在B之前引用B 则为空
```

```
B = China
```

则 A 的值将不包含 China，因此在定义 A 时 B 的值为空。

3，条件定义方式：

有时我们需要先判断一个变量是否已经定义了，如果已经定义了则不作操作，如果没有定义再来定义它的值，这时最方便的方法就是采用所谓的条件定义方式：

```
A = apple
A ?= I love China
```

此处对 A 进行了两次定义，其中第二次是条件定义，其含义是：如果 A 在此之前没有定义，则定义为“I love China”，否则维持原有的值。

4，多行命令定义方式：

```
define commands
    echo “thank you!”
    echo “you are welcome.”
endef
```

此处定义了一个包含多行命令的变量commands，我们利用它的这个特点实现一个完整命令包的定义。注意其语法格式：以define开头，以endef结束，所要定义的变量名必须在指示符“define”的同一行之后，指示符define所在行的下一行开始一直到“end”所在行的上一行之间的若干行，是变量的值。这种方式定义的所谓命令包，可以理解为编程语言中的函数。

Makefile中的变量还有以下几种操作方式：

1，追加变量的值，例如：

```
A = apple
A += tree
```

这样，变量A的值就是apple tree。

2，修改变量的值，例如：

```
A = srt.c string.c tcl.c
B = $(A:%.c=%.o)
```

第三个版本：

```
TAG=./bin/main
SRC=./src/Input.c ./src/main.c ./src/Oper.c ./src/Output.c
OBJ=$(SRC:%.c=%.o)
CC=gcc
O=-o
CONFIG=-I./inc


$(TAG):$(OBJ)
    $(CC)  $(^)  $(O) $(@) $(CONFIG)


%.o:%.c
    $(CC) $< -o $(@) $(CONFIG) -c

clean:
    $(RM) ./bin/*  ./src/*.o
```

函数

```
$(subst FROM,TO,TEXT)
```

功能：
将字符串 TEXT 中的字符 FROM 替换为 TO。
返回：
替换之后的新字符串。
范例：

```
A = $(subst pp,PP,apple tree)
```

替换之后变量 A 的值是”aPPle tree“

```
$(wildcard PATTERN)
```

功能：
获取匹配模式为 PATTERN 的文件名。
返回：
匹配模式为 PATTERN 的文件名。
范例：

```
A = $(wildcard *.c)
```

假设当前路径下有两个.c 文件 a.c 和 b.c，则处理后 A 的值为：” a.c b.c” 。

override
override一个变量，例如：

```
override CFLAGS += -Wall
```

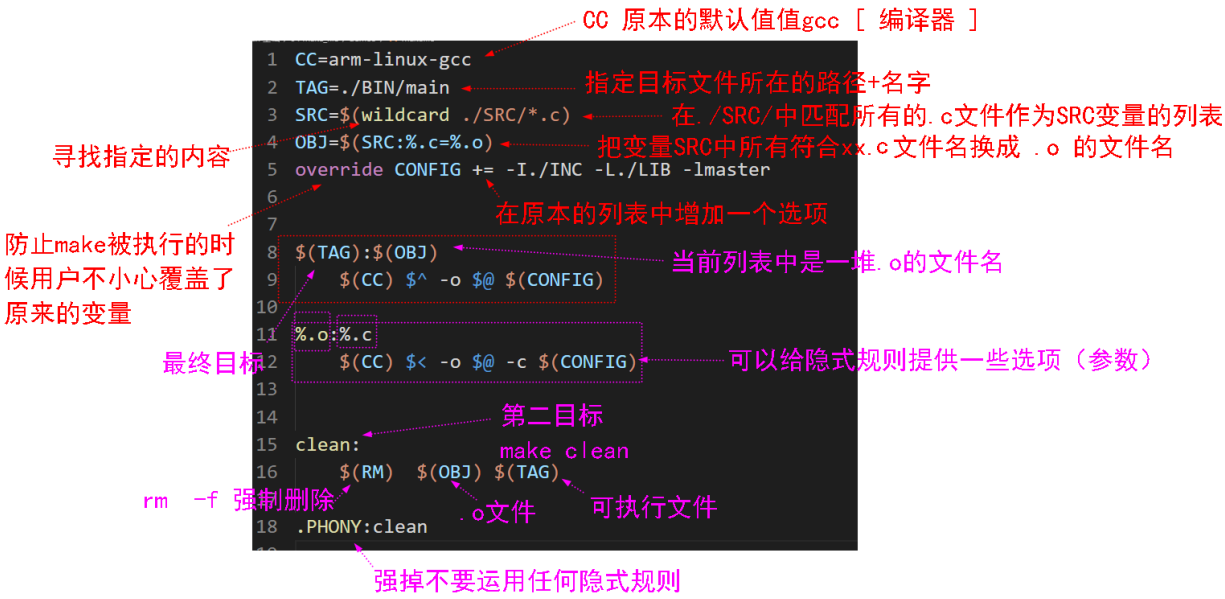
.PHONY
.PHONY 来明确地告诉 Makefile,不要对 clean 运用任何隐式规则,不能运用隐式规则的目标被称为伪目标.

```
.PHONY:clean
用来修饰 clean 清空的工作不会被误以为是一个目标来执行
```

第四版本（通用版本）

```
TAG=./bin/main
SRC= $(wildcard src/*.c)
OBJ=$(SRC:%.c=%.o)
CC=gcc
override CONFIG += -I./inc

$(TAG):$(OBJ)
    $(CC)  $(^)  -o  $(@) $(CONFIG)

%.o:%.c
    $(CC) $< -o $(@) $(CONFIG) -c

```

```
clean:
    $(RM) ./bin/*  ./src/*.o

.PHONY:clean
```



练习：

一个一个去试一下依赖于目标的关系。