makefile

make工具的使用
makefile规则的概念
makefile变量的使用
makefile函数的使用
makefile自动生成依赖

1 makefile及make简介

1.1 makefile简介

```
target:prerequisites
<tab> command1
<tab> command2
.....
<tab> commandN

arget：规则的目标
prerequisites：规则的依赖列表
command：规则的命令

特别注意：每行命令都必须以tab键开始！
```

1.2 make命令工作机理

1.简单粗暴，不带任何参数，直接执行make：

```
$ make
```

2.指定makefile文件：

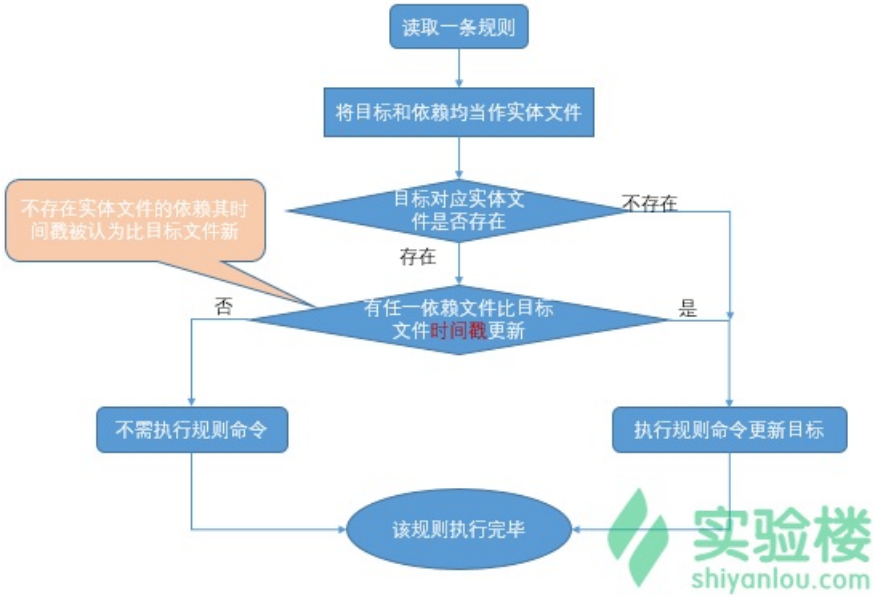
```
$ make -f <makefile_name>
```

3.指定makefile 目标：

```
$ make <target>
```

4.到指定目录下执行make：

```
$ make -C <subdir> <target>
```

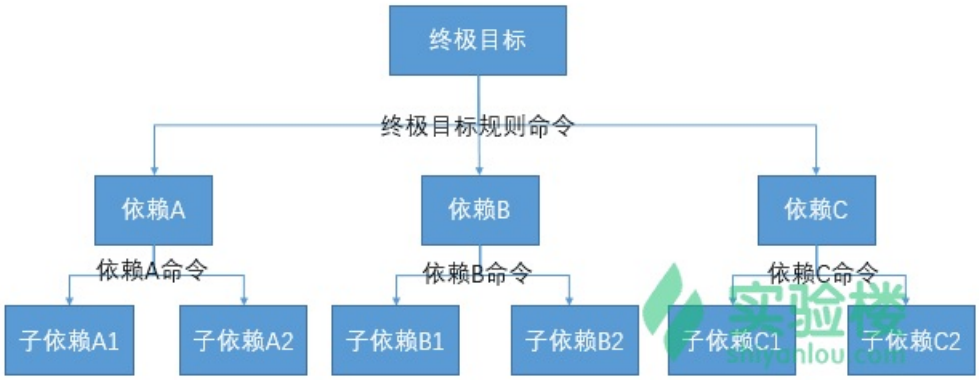


make解析makefile的流程如下：

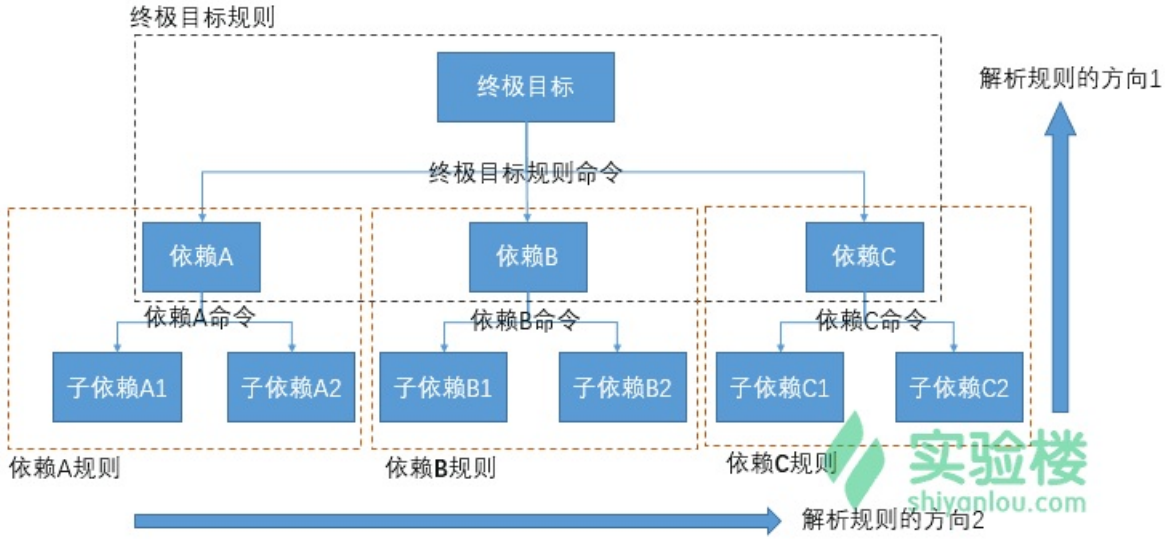
假设有mekefile内容如下：

```
终极目标：依赖A  依赖B 依赖C
    终极目标命令

依赖A：子依赖A1 子依赖A2
    依赖A命令

依赖B：子依赖B1 子依赖B2
    依赖B命令

依赖C：子依赖C1 子依赖C2
    依赖C命令
```

过程一，以终极目标为树根，解析出整颗依赖树：

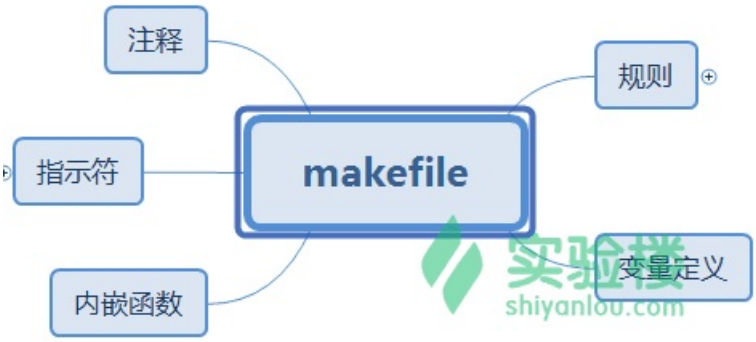


过程二，对整颗依赖树以从底到上，从左到右的顺序，解析执行每一条规则：



2 makefile语法工具箱

一般情况下一个完整的makefile所包含的语法模块



2.1 使用变量让makefile更灵活

变量定义的基本格式：

```
变量名 赋值符 变量值

**递归展开式**
=
?= 此变量在之前没有赋值的情况下才会对这个变量进行赋值
+= 对于一个已经存在定义的变量进行追加赋值

**直接展开式**
:=
```

递归式变量的求值时机在于**变量被引用时**;
直接展开式的求值时机在于**变量被定义时。**

2.2 变量的引用

```
$ {变量名}
$ （ 变量名 ）
$ 单字符变量名 ，变量名仅包含一个字符，如$@ 、$^等
```

2.3 特殊的变量-自动化变量

```
$@ -- 代表规则中的目标文件名

$< -- 代表规则的第一个依赖的文件名
```

$^ -- 代表规则中所有依赖文件的列表，文件名用空格分割

eg:

```
all: first second third
    echo "\$$@ = $@"
    echo "$$< = $<"
    echo "$$^ = $^"
first second third:
```

2.4 变量的替换引用

格式为$(VAR:A=B)或者${VAR:A=B}，意思是，将变量“VAR”所表示的值中所有字符串“A”结尾的字符替换为“B”的字。“结尾”的含义是空格之前（变量值的多个字以空格分开）。而对于变量其它部分的“A”字符不进行替换

eg:

```
sources := a.c b.c c.c o.d
objects := $(sources:.c=.o)
all:
    echo "objects = $(objects)"
```

3 更加深入的认识makefile的规则

3.1 多目标规则与多规则目标

假设我们有以下makefile：

```
all: target1 target2
    echo "This is a rule for $@"

target1: dep
    echo "This is a rule for $@"

target2: dep
    echo "This is a rule for $@"
```

利用多目标规则，可以将makefile改写成：

```
dep:
#多目标规则
all: target1 target2
    echo "This is a rule for $@"

# 利用多目标规则合并 target1 和target2的规则
target1 target2: dep
    echo "This is a rule for $@"

dep:
```

多规则目标
Makefile中，一个目标可以同时出现在多条规则中。这种情况下，此目标文件的所有依赖文件将会被合并成此目标一个依赖文件列表，其中任何一个依赖文件比目标更新（比较目标文件和依赖文件的时间戳）时，make将会执行特定的命令来重建这个目标。对于一个多规则的目标，重建此目标的命令只能出现在一个规则中（可以是多条命令）。如果多个规则同时给出重建此目标的命令，make将使用最后一个规则的命令，同时提示错误信息。

3.2 静态模式规则

静态模式规则，可以理解为一种特殊的多目标规则，它仅要求多条规则具有相同的命令，而依赖可以不完全一样。

静态模式规则，其基本语法：

```
TARGETS ...: TARGET-PATTERN: PREREQ-PATTERNS ...
    COMMANDS
    ...
```

其大致意思就是，用TARGET-PATTERN: PREREQ-PATTERNS …描述的模式，从TARGETS …取值来形成一条条规则，所有规则的命令都用COMMANDS。

TARGETS …代表具有相同模式的规则的目标列表，在我们的项目中就是main.o和complicated.o，我们可以直接引用我们先前定义的objects变量。

TARGET-PATTERN: PREREQ-PATTERNS …部分定义了，如何为目标列表中的目标，生成依赖；TARGET-PATTERN称为目标模式，PREREQ-PATTERNS称为依赖模式；目标模式和依赖模式中，一般需要包含模式字符%。

目标模式的作用就是从目标列表中的目标匹配过滤出需要的值，目标模式中的字符%表示在匹配过滤的过程中不做过滤的部分，目标模式中的其他字符表示要与目标列表中的目标精确匹配，例如，目标模式%.o， 表示从目标列表的目标中匹配所有已.o结尾的目标，然后过滤掉匹配目标的.o部分， 因此目标main.o经过目标模式%.o匹配过滤后，得到的输出就是main。

依赖模式的作用就是表示要如何生成依赖文件。具体的生成过程，就是使用目标模式过滤出来的值，替换依赖模式字符%所表示的位置。因此，如果依赖模式为%.c， 则使用上述例子过滤出来的main来替换字符%， 最终得到依赖文件main.c

3.3 伪目标

定义一个为目标的基本语法：

```
.PHONY: <伪目标>
```

3.4 命令

3.4.1 关闭命令回显有以下几种方式：

i. 每个需要关闭回显的命令行前加上”@”字符，上述例子关闭回显：

```
all:
    @echo "Hello world!"
```

ii. 执行make时带上参数-s或–slient禁止所有执行命令的显示

iii. 在Makefile中使用没有依赖的特殊目标.SILENT也可以禁止所有命令的回显

3.4.2 命令的执行

在Makefile中书写在同一行中的多个命令属于一个完整的shell命令行，书写在独立行的一条命令是一个独立的shell命令行。

```
target1:
    @echo "target1"
    @cd ~
    @pwd

target2:
    @echo "target2"
    @cd ~; pwd
```

4 内嵌函数

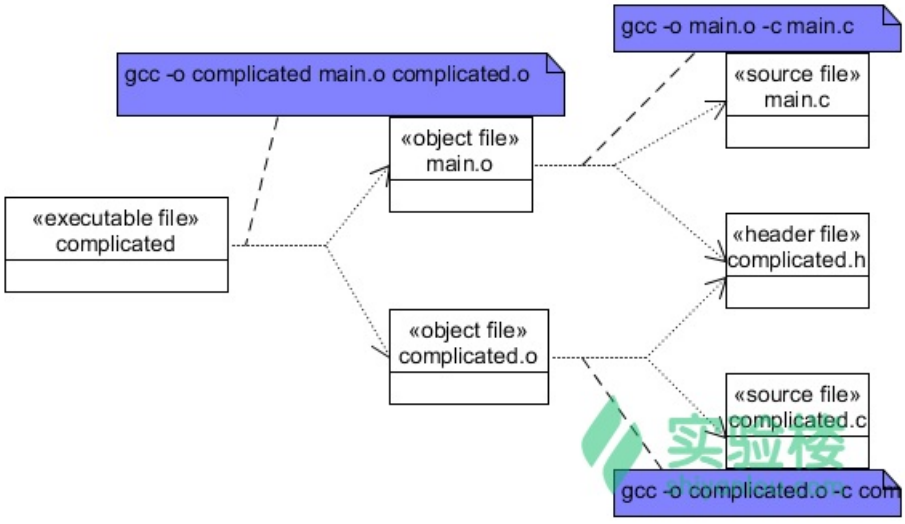
函数调用方式1：

```
$(FUNCTION ARGUMENTS)
```

函数调用方式2：

```
${FUNCTION ARGUMENTS}
```

3.5 自动生成依赖关系



gcc为我们提供了这样的功能，帮助我们分析一个文件对其他文件的依赖关系列表。当我们在执行gcc时带上-MM选项时，gcc工具就会为我们列出指定文件对其他文件的依赖关系列表。直接来看例子：

makefile支持使用sinclude关键字将指定文件导入到当前的makefile当中，它的作用与C语言的#inlucde预处理命令是一样的。使用方式sinclude 。因此，我们可以将gcc对于源文件的依赖关系分析输出到某个文件(我们可以称为依赖描述文件，一般命名为与源文件同名但以.d结尾的文件)当中，然后再将依赖描述文件导入到makefile中。

```
# 使用变量的引用替换，定义依赖描述文件列表
deps := $(sources:.c=.d)

# 导入依赖描述文件列表
sinclude $(deps)
```

当我们使用sinclude关键字向当前makefile导入文件时，如果所导入的文件不存在，make会试图去执行可以生产导入文件的规则去生产被导入的文件，然后再执行导入。因此我们可以使用静态模式规则，让make在执行时，去调用gcc生成依赖关系文件，我们可以这么写：

```
$(deps):%.d:%.c
    gcc -MM $< > $@
```

因此，我们complicated项目的最终makefile可以这么写：

```
# 描述：complicated 项目 makefile文件
# 版本：v1.5
# 修改记录：
# 1. 为complicated项目makefile添加注释
# 2. 使用变量改进我们complicated项目的makefile
# 3. 使用静态模式规则，简化makefile
# 4. 使用伪目标，加上clean规则
# 5. 引进wildcard函数，自动扫描当前目录下的源文件
# 6. 加入自动规则依赖

# 定义可执行文件变量
executbale := complicated
# wildcard函数扫描源文件，定义列表变量
sources := $(wildcard *.c)
# 使用变量的引用替换，定义object文件列表
objects := $(sources:.c=.o)
# 使用变量的引用替换，定义依赖描述文件列表
deps := $(sources:.c=.d)

# 定义编译命令变量
CC := gcc
RM := rm -rf

# 终极目标规则，生成complicated可执行文件
$(executbale): $(objects)
#  使用自动化变量改造我们的编译命令
    $(CC) -o $@ $^

# 子规则, main.o和complicated.o的生成规则，使用静态模式规则
$(objects):%.o:%.c
    $(CC) -o $@  -c $<

# clean规则
.PHONY: clean
clean:
    $(RM) $(executbale) $(objects) $(deps)

# 自动规则依赖
sinclude $(deps)

$(deps):%.d:%.c
    $(CC) -MM $< > $@
```