Makefile/makefile介绍

■ 一个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，Makefile 文件定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 Makefile 文件就像一个 Shell 脚本一样，也可以执行操作系统的命令。
■ Makefile 带来的好处就是“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 Makefile 文件中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如 Delphi 的 make，Visual C++ 的 nmake，Linux 下 GNU 的 make。

Makefile的文件命名和规则

■ 文件命名
makefile 或者 Makefile
■ Makefile 规则

- 一个 Makefile 文件中可以有一个或者多个规则

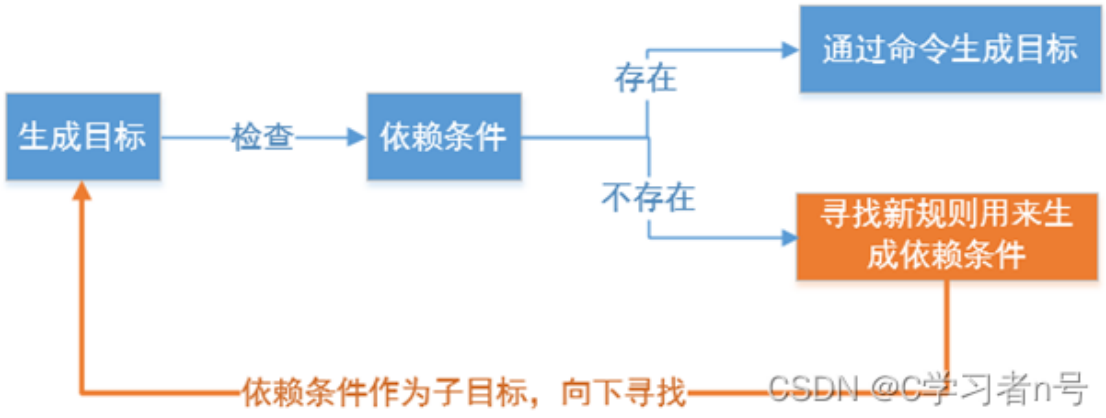
> 目标 ...: 依赖 ...
> (Tab空格)命令（Shell 命令）
> * 目标：最终要生成的文件（伪目标除外）
> * 依赖：生成目标所需要的文件或是目标
> * 命令：通过执行命令对依赖操作生成目标（命令前必须 Tab 缩进）

- Makefile 中的其它规则一般都是为第一条规则服务的。如果第一条规则中有的依赖文件在当前目录中找不到，那么就会查找下面的其他规则是否能够生成这个没有的依赖文件，如果生成了，那么就再找第一条规则中的其他依赖文件。以此类推。
  所以，除了第一条规则会执行外，如果没有被用到，其他规则是不会执行的。

工作原理

1、命令在执行之前，需要先检查规则中的依赖是否存在。

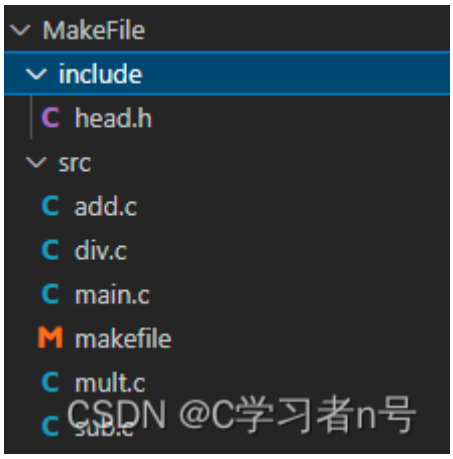
- 如果存在，执行命令；
- 如果不存在，则向下检查其他的规则，检查有没有一个规则是用来生成这个依赖的，如果找到了，则执行该规则中的命令。



2、检测更新，在执行规则中的命令时，会比较目标和依赖文件的时间

- 如果依赖的时间比目标的时间晚，则需要重新生成目标
- 如果依赖的时间比目标的时间早，目标不需要更新，对应规则中的命令不需要被执行。

两种基本格式的makefile




```
#1、最简单的makefile
app:add.c sub.c mult.c div.c main.c
    gcc -o app add.c sub.c mult.c div.c main.c -I../include
```

这种方式的makefile效率低，修改其中一个文件，那么所有的文件都会重新编译

```
#2、修改格式。更新更好，但是当.c文件过多，那么就很难写
app:add.o sub.o mult.o div.o main.o
    gcc -o app add.o sub.o mult.o div.o main.o -I../include

main.o:main.c
    gcc -c main.c -o main.o -I../include

add.o:add.c
    gcc -c add.c -o add.o -I../include

sub.o:sub.c
    gcc -c sub.c -o sub.o -I../include

mult.o:mult.c
    gcc -c mult.c -o mult.o -I../include

div.o:div.c
    gcc -c div.c -o div.o -I../include
```

执行步骤

第二种格式中的第一条规则是主要执行的规则，一开始没有编译生成的.o文件，那么搜索add.o时就会去其他依赖找到是否有规则的目标文件是add.o，如果有这个规则，那么就执行对应的命令，再返回第一条规则，查找其他依赖是否存在，依此规则处理其他的。

更新步骤

如果第一条规则中的目标文件已经存在了，那么就检查规则中的目标是否需要更新，也就是要检查他所对应的依赖，然后再判断这些依赖文件是否需要更新。比如搜索到第一条规则的add.o，此时要判断add.o是否需要更新，那么就检测到了第二条规则，判断add.o和add.c两者的时间，如果add.o要比add.c早，也就说明add.c被修改过了，那么就要重新生成这个add.o。以此判断第一条依赖的其他依赖文件，最后再决定执行命名与否。

所以，在第二中格式的情况下，不需要全部更新所有的文件，只需要更新修改过的即可。

makefile文件的变量与模式匹配

1. 自定义变量
   变量名=变量值　(var=hello)
2. 预定义变量
   AR : 归档维护程序的名称，默认值为 ar
   CC : C 编译器的名称，默认值为 gcc
   CXX : C++ 编译器的名称，默认值为 g++
   CPPFLAGS : C预处理的选项 -I
   CFLAGS:　C编译器的选项 -Wall -g -c
   LDFLAGS :　链接器选项 -L　-l

   $@ : 目标的完整名称
   $< : 第一个依赖文件的名称
   $^ : 所有的依赖文件
3. 获取变量的值
   $(变量名)
4. 模式匹配
   %.o:%.c
   %: 通配符，匹配一个字符串
   两个%匹配的是同一个字符串

通过变量和模式匹配可以将上面的makefile进行简化，更加通用

```
#定义变量
src=add.o sub.o mult.o div.o main.o
target=app
CC=gcc
CPPFLAGS=-I../include

$(target):$(src)
    $(CC) $(src) -o $(target)

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

Makefile文件的函数

1. $(wildcard PATTERN...)

- 功能：获取指定目录下指定类型的文件列表
- 参数：PATTERN 指的是某个或多个目录下的对应的某种类型的文件，如果有多个目录，一般使用空格间隔
- 返回：得到的若干个文件的文件列表，文件名之间使用空格间隔
- 举例：
  $(wildcard *.c ./sub/*.c)
  返回值格式: a.c b.c c.c d.c e.c f.c

2.$(patsubst <pattern>,<replacement>,<text>)

- 功能：查找<text>中的单词(单词以“空格”、“Tab”或“回车”“换行”分隔)是否符合模式，如果匹配的话，则以<replacement>替换。
- <pattern>可以包括通配符`%`，表示任意长度的字串。如果<replacement>中也包含`%`，那么，<replacement>中的这个`%`将是<pattern>中的那个%所代表的字串。(可以用`\`来转义，以`\%`来表示真实含义的`%`字符)
- 返回：函数返回被替换过后的字符串

- 示例:
  $(patsubst %.c, %.o, x.c bar.c)
  返回值格式: x.o bar.o

根据这两个函数对上面的第三个格式进行进一步的修改

```
src=$(wildcard ./*.c)
objs=$(patsubst *.c, *.o, $(src))
target=app
CC=gcc
CPPFLAGS=-I../include

$(target):$(objs)
    $(CC) $(objs) -o $(target) $(CPPFLAGS)

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

```
src=$(wildcard ./*.c)
objs=$(patsubst *.c, *.o, $(src))
target=app
CC=gcc
CPPFLAGS=-I../include

$(target):$(objs)
    $(CC) -o $(target) $(objs) $(CPPFLAGS)

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

.PHONY:clean
clean:
    rm -f *.o
```