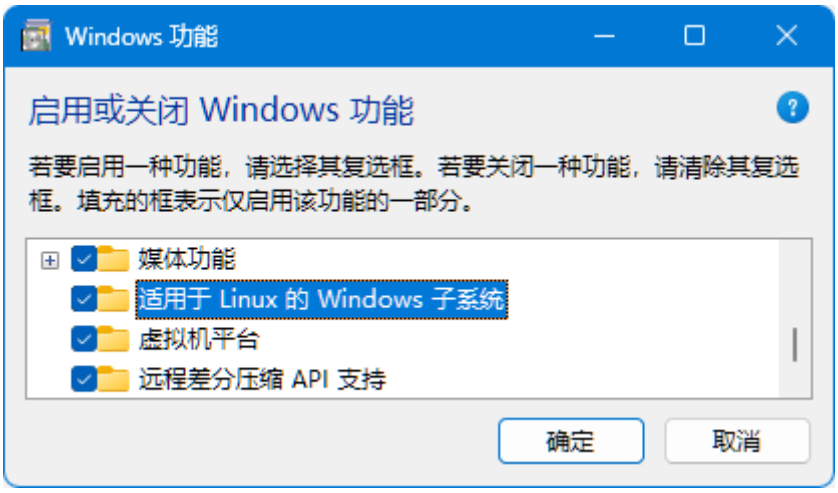
廖雪峰makefile

安装make

安装`make`时，因为`make`只能在Unix/Linux下运行，所以，如果使用Windows系统，我们要先想办法在Windows下跑一个Linux。

方法一：安装VirtualBox，然后下载Linux发行版安装盘，推荐Ubuntu 22.04，这样就可以在虚拟机中运行Linux。

方法二：对于Windows 10/11，可以首先安装WSL（Windows Subsystem for Linux）：



然后，在Windows应用商店，搜索Ubuntu 22.04，直接安装后运行，Windows会弹出PowerShell的窗口连接到Linux，在PowerShell中即可输入Linux命令，和SSH连接类似。

以Ubuntu为例，在Linux命令行下，用`apt`命令安装`make`以及GCC工具链：

```
$ sudo apt install build-essential
```

对于macOS系统，因为它的内核是BSD（一种Unix），所以也能直接跑`make`，推荐安装Homebrew，然后通过Homebrew安装`make`以及GCC工具链：

```
$ brew install make gcc
```

安装完成后，可以输入`make -v`验证：

```
cyb@DESKTOP-UOIAA21:~/graduateWork/STATool/edacontest-ssta$ make -v
GNU Make 4.3
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

输入`gcc --version`验证GCC工具链：

```
cyb@DESKTOP-UOIAA21:~/graduateWork/STATool/edacontest-ssta$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

这样，我们就成功地安装了`make`，以及GCC工具链。

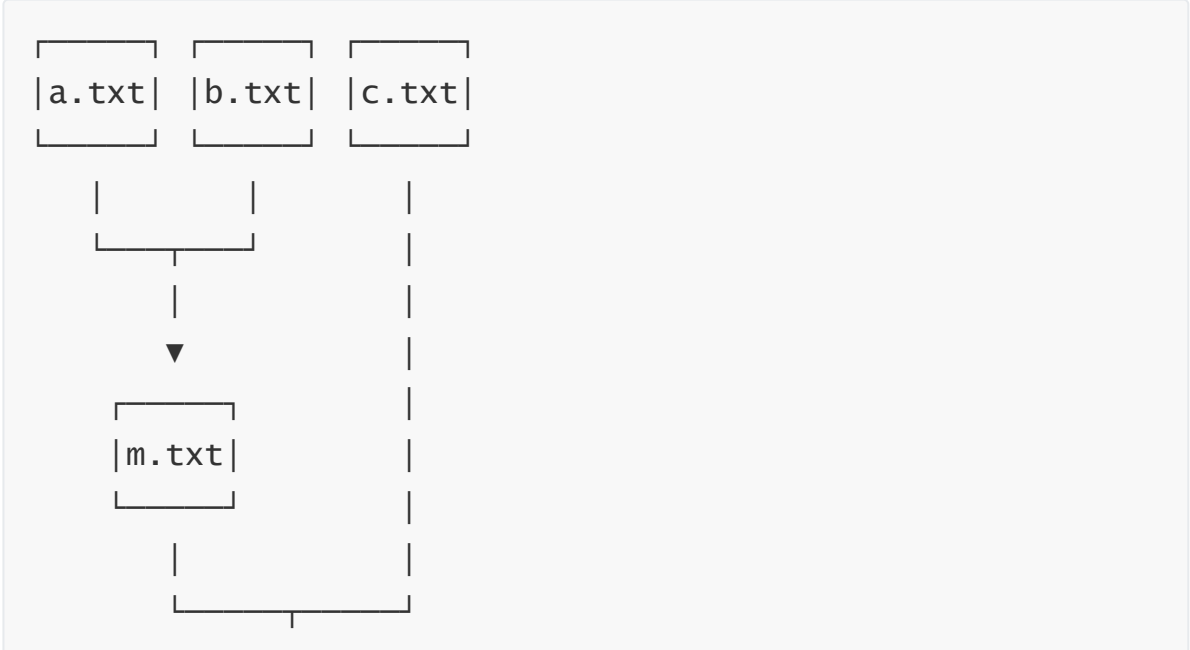
Makefile基础

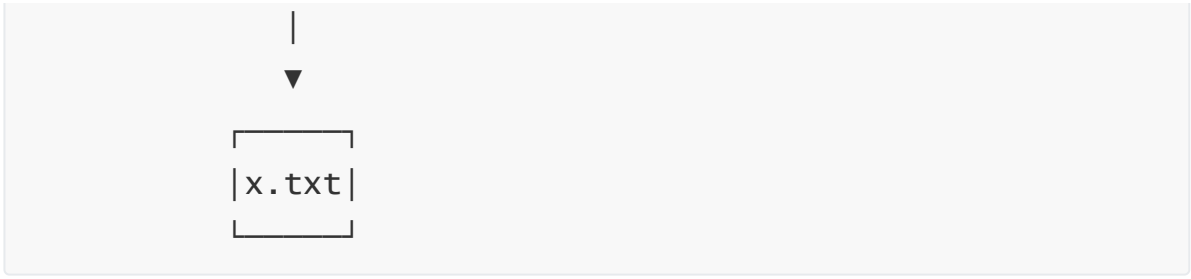
在Linux环境下，当我们输入`make`命令时，它就在当前目录查找一个名为`Makefile`的文件，然后，根据这个文件定义的规则，自动化地执行任意命令，包括编译命令。

`Makefile`这个单词，顾名思义，就是指如何生成文件。

我们举个例子：在当前目录下，有3个文本文件：`a.txt`，`b.txt`和`c.txt`。

现在，我们要合并`a.txt`与`b.txt`，生成中间文件`m.txt`，再用中间文件`m.txt`与`c.txt`合并，生成最终的目标文件`x.txt`，整个逻辑如下图所示：





根据上述逻辑，我们来编写`Makefile`。

规则

`Makefile`由若干条规则（Rule）构成，每一条规则指出一个目标文件（Target），若干依赖文件（prerequisites），以及生成目标文件的命令。

例如，要生成`m.txt`，依赖`a.txt`与`b.txt`，规则如下：

```
# 目标文件: 依赖文件1 依赖文件2
m.txt: a.txt b.txt
    cat a.txt b.txt > m.txt
```

注意：Makefile的规则中，命令必须以Tab开头，不能是空格。

类似的，我们写出生成`x.txt`的规则如下：

```
x.txt: m.txt c.txt
    cat m.txt c.txt > x.txt
```

由于`make`执行时，默认执行第一条规则，所以，我们把规则`x.txt`放到前面。完整的`Makefile`如下：

```
x.txt: m.txt c.txt
    cat m.txt c.txt > x.txt

m.txt: a.txt b.txt
    cat a.txt b.txt > m.txt
```

在当前目录创建`a.txt`、`b.txt`和`c.txt`，输入一些内容，执行`make`：

```
$ make
cat a.txt b.txt > m.txt
cat m.txt c.txt > x.txt
```

```
cyb@DESKTOP-UOIAA21:~/test-makefile$ make
cat a.txt b.txt > m.txt
cat m.txt c.txt > x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ ls
a.txt  b.txt  c.txt  m.txt  makefile  x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ cat x.txt
a.txt
b.txt
c.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ cat m.txt
a.txt
b.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$
```

`make`默认执行第一条规则，也就是创建`x.txt`，但是由于`x.txt`依赖的文件`m.txt`不存在（另一个依赖`c.txt`已存在），故需要先执行规则`m.txt`创建出`m.txt`文件，再执行规则`x.txt`。执行完成后，当前目录下生成了两个文件`m.txt`和`x.txt`。

可见，`Makefile`定义了一系列规则，每个规则在满足依赖文件的前提下执行命令，就能创建出一个目标文件，这就是英文Make file的意思。

把默认执行的规则放第一条，其他规则的顺序是无关紧要的，因为`make`执行时自动判断依赖。

此外，`make`会打印出执行的每一条命令，便于我们观察执行顺序以便调试。

如果我们再次运行`make`，输出如下：

```
$ make
make: `x.txt' is up to date.
```

```
cyb@DESKTOP-UOIAA21:~/test-makefile$ make
make: 'x.txt' is up to date.
```

`make`检测到`x.txt`已经是最新版本，无需再次执行，因为`x.txt`的创建时间晚于它依赖的`m.txt`和`c.txt`的最后修改时间。

make使用文件的创建和修改时间来判断是否应该更新一个目标文件。

修改`c.txt`后，运行`make`，会触发`x.txt`的更新：

```
$ make
cat m.txt c.txt > x.txt
```

但并不会触发`m.txt`的更新，原因是`m.txt`的依赖`a.txt`与`b.txt`并未更新，所以，`make`只会根据`Makefile`去执行那些必要的规则，并不会把所有规则都无脑执行一遍。

在编译大型程序时，全量编译往往需要几十分钟甚至几个小时。全量编译完成后，如果仅修改了几个文件，再全部重新编译完全没有必要，用`Makefile`实现增量编译就十分节省时间。

当然，是否能正确地实现增量更新，取决于我们的规则写得对不对，`make`本身并不会检查规则逻辑是否正确。

伪目标

因为`m.txt`与`x.txt`都是自动生成的文件，所以，可以安全地删除。

删除时，我们也不希望手动删除，而是编写一个`clean`规则来删除它们：

```
clean:
    rm -f m.txt
    rm -f x.txt
```

`clean`规则与我们前面编写的规则有所不同，它没有依赖文件，因此，要执行`clean`，必须用命令`make clean`：

```
$ make clean
rm -f m.txt
rm -f x.txt
```

然而，在执行`clean`时，我们并没有创建一个名为`clean`的文件，所以，因为目标文件`clean`不存在，每次运行`make clean`，都会执行这个规则的命令。

如果我们手动创建一个`clean`的文件，这个`clean`规则就不会执行了！

```
cyb@DESKTOP-UOIAA21:~/test-makefile$ make
cat a.txt b.txt > m.txt
cat m.txt c.txt > x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ ls
a.txt  b.txt  c.txt  clean  m.txt  makefile  x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ make clean
make: 'clean' is up to date.
cyb@DESKTOP-UOIAA21:~/test-makefile$
```

如果我们希望`make`把`clean`不要视为文件，可以添加一个标识：

```
.PHONY: clean
clean:
    rm -f m.txt
    rm -f x.txt
```

此时，`clean`就不被视为一个文件，而是伪目标（Phony Target）。

大型项目通常会提供`clean`、`install`这些约定俗成的伪目标名称，方便用户快速执行特定任务。

一般来说，并不需要用`.PHONY`标识`clean`等约定俗成的伪目标名称，除非有人故意搞破坏，手动创建名字叫`clean`的文件。

```
cyb@DESKTOP-UOIAA21:~/test-makefile$ make
cat a.txt b.txt > m.txt
cat m.txt c.txt > x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ ls
a.txt  b.txt  c.txt  clean  m.txt  makefile  x.txt
cyb@DESKTOP-UOIAA21:~/test-makefile$ make clean
make: 'clean' is up to date.
cyb@DESKTOP-UOIAA21:~/test-makefile$
```

执行多条命令

一个规则可以有多条命令，例如：

```
cd:
    pwd
    cd ..
    pwd
```

执行`cd`规则：

```
$ make cd
pwd
/home/ubuntu/makefile-tutorial/v1
cd ..
pwd
/home/ubuntu/makefile-tutorial/v1
```

观察输出，发现`cd ..`命令执行后，并未改变当前目录，**两次输出的`pwd`是一样的，这是因为`make`针对每条命令，都会创建一个独立的Shell环境，类似`cd ..`这样的命令，并不会影响当前目录。**

解决办法是把多条命令以`;`分隔，写到一行：

```
cd_ok:
    pwd; cd ..; pwd;
```

再执行`cd_ok`目标就得到了预期结果：

```
$ make cd_ok
pwd; cd ..; pwd
/home/ubuntu/makefile-tutorial/v1
/home/ubuntu/makefile-tutorial
```

可以使用`\`把一行语句拆成多行，便于浏览：

```
cd_ok:
    pwd; \
    cd ..; \
    pwd
```

另一种执行多条命令的语法是用`&&`，它的好处是当某条命令失败时，后续命令不会继续执行：

```
cd_ok:
    cd .. && pwd
```

控制打印

默认情况下，`make`会打印出它执行的每一条命令。如果我们不想打印某一条命令，可以在命令前加上`@`，表示不打印命令（但是仍然会执行）：

```
no_output:
    @echo 'not display'
    echo 'will display'
```

执行结果如下：

```
$ make no_output
not display
echo 'will display'
will display
```

注意命令`echo 'not display'`本身没有打印，但命令仍然会执行，并且执行的结果仍然正常打印。

控制错误

`make`在执行命令时，会检查每一条命令的返回值，如果返回错误（非0值），就会中断执行。

例如，不使用`-f`删除一个不存在的文件会报错：

```
has_error:
    rm zzz.txt
    echo 'ok'
```

执行上述目标，输出如下：

```
$ make has_error
rm zzz.txt
rm: zzz.txt: No such file or directory
make: *** [has_error] Error 1
```

由于命令`rm zzz.txt`报错，导致后面的命令`echo 'ok'`并不会执行，`make`打印出错误，然后退出。

有些时候，我们想忽略错误，继续执行后续命令，可以在需要忽略错误的命令前加上`-`：

```
ignore_error:
    -rm zzz.txt
    echo 'ok'
```

执行上述目标，输出如下：

```
$ make ignore_error
rm zzz.txt
rm: zzz.txt: No such file or directory
make: [ignore_error] Error 1 (ignored)
echo 'ok'
ok
```

`make`检测到`rm zzz.txt`报错，并打印错误，但显示`(ignored)`，然后继续执行后续命令。

对于执行可能出错，但不影响逻辑的命令，可以用`-`忽略。

```
cyb@DESKTOP-UOIAA21:~/test-makefile/test3$ make -f makefile2
rm 1.txt
rm: cannot remove '1.txt': No such file or directory
make: *** [makefile2:2: ignore_error] Error 1
cyb@DESKTOP-UOIAA21:~/test-makefile/test3$ vim makefile2
cyb@DESKTOP-UOIAA21:~/test-makefile/test3$ make -f makefile2
rm 1.txt
rm: cannot remove '1.txt': No such file or directory
make: [makefile2:2: ignore_error] Error 1 (ignored)
echo "ok"
ok
cyb@DESKTOP-UOIAA21:~/test-makefile/test3$
```

编译C程序

C程序的编译通常分两步：

1. 将每个`.c`文件编译为`.o`文件；
2. 将所有`.o`文件链接为最终的可执行文件。

我们假设如下的一个C项目，包含`hello.c`、`hello.h`和`main.c`。

`hello.c`内容如下：

```
#include <stdio.h>

int hello()
{
    printf("hello, world!\n");
    return 0;
}
```

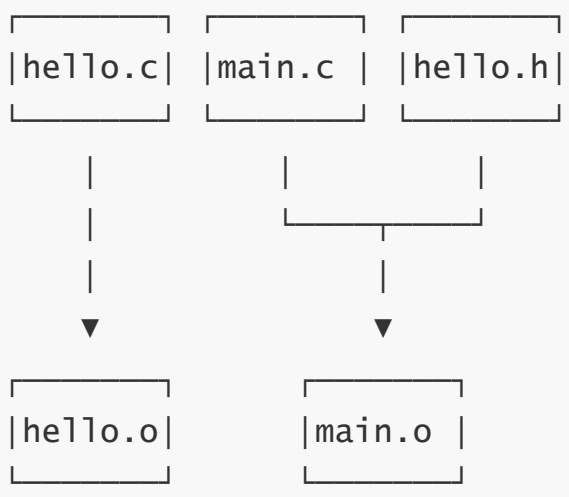
`hello.h`内容如下：

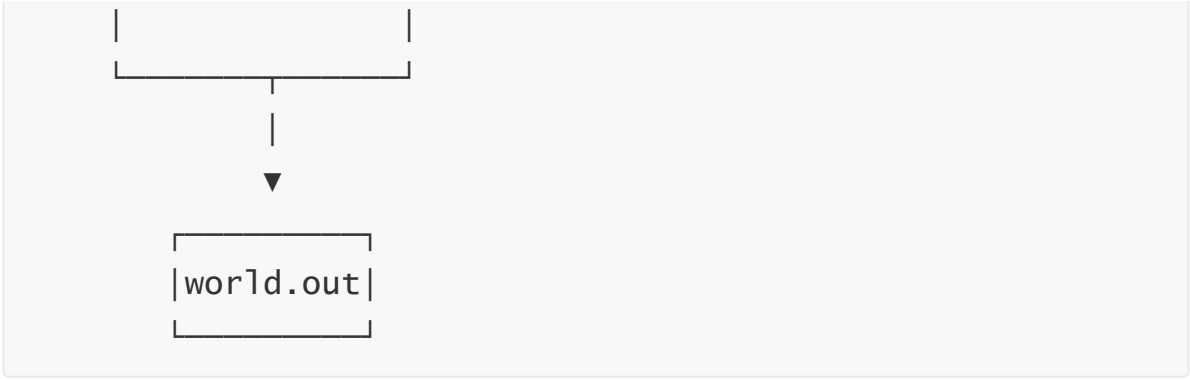
```
int hello();
```

`main.c`内容如下：

```
#include <stdio.h>
#include "hello.h"

int main()
{
    printf("start...\n");
    hello();
    printf("exit.\n");
    return 0;
}
```

注意到`main.c`引用了头文件`hello.h`。我们很容易梳理出需要生成的文件，逻辑如下：





假定最终生成的可执行文件是`world.out`，中间步骤还需要生成`hello.o`和`main.o`两个文件。根据上述依赖关系，我们可以很容易地写出`Makefile`如下：

```
# 生成可执行文件:
world.out: hello.o main.o
    cc -o world.out hello.o main.o

# 编译 hello.c:
hello.o: hello.c
    cc -c hello.c

# 编译 main.c:
main.o: main.c hello.h
    cc -c main.c

clean:i
    rm -f *.o world.out
```

执行`make`，输出如下：

```
$ make
cc -c hello.c
cc -c main.c
cc -o world.out hello.o main.o
```

在当前目录下可以看到`hello.o`、`main.o`以及最终的可执行程序`world.out`。执行`world.out`：

```
$ ./world.out
start...
hello, world!
exit.
```

与我们预期相符。

修改`hello.c`，把输出改为`"hello, bob!\n"`，再执行`make`，观察输出：

```
$ make
cc -c hello.c
cc -o world.out hello.o main.o
```

仅重新编译了`hello.c`，并未编译`main.c`。由于`hello.o`已更新，所以，仍然要重新生成`world.out`。执行`world.out`：

```
$ ./world.out
start...
hello, bob!
exit.
```

与我们预期相符。

修改`hello.h`：

```
// int 变为 void:
void hello();
```

以及`hello.c`，再次执行`make`：

```
$ make
cc -c hello.c
cc -c main.c
cc -o world.out hello.o main.o
```

会触发`main.c`的编译，因为`main.c`依赖`hello.h`。

执行`make clean`会删除所有的`.o`文件，以及可执行文件`world.out`，再次执行`make`就会强制全量编译：

```
$ make clean && make
rm -f *.o world.out
cc -c hello.c
cc -c main.c
cc -o world.out hello.o main.o
```

这个简单的`Makefile`使我们能自动化编译C程序，十分方便。

不过，随着越来越多的`.c`文件被添加进来，如何高效维护`Makefile`的规则？我们后面继续讲解。

gcc编译过程
-E：预处理、只关联主函数cpp文件，若有其他cpp文件会报错

```
g++: fatal error: cannot specify -o with -c, -S or -E with
multiple files
```

-S：编译、只关联预处理文件产生的 .ii文件，若有其他.ii文件会报错

-c：汇编成目标代码（.o）只关联编译产生的 .s文件，若有其他.s文件会报错

链接：多文件编译时，需要main cpp文件中头文件对应的cpp文件(一般也将该文件编译成.o文件)，若不包含会产生头文件中函数找不到的错误

使用隐式规则

们仍然以上一节的C项目为例，当我们添加越来越多的`.c`文件时，就需要编写越来越多的规则来生成`.o`文件。

实际上，有的同学可能发现了，即使我们把`.o`的规则删掉，也能正常编译：

```
# 只保留生成 world.out 的规则:
world.out: hello.o main.o
    cc -o world.out hello.o main.o

clean:
    rm -f *.o world.out
```

执行`make`，输出如下：

```
$ make
cc     -c -o hello.o hello.c
cc     -c -o main.o main.c
cc -o world.out hello.o main.o
```

```
cyb@DESKTOP-UOIAA21:~/test-makefile/test5$ make
cc     -c -o hello.o hello.c
cc     -c -o main.o main.c
gcc hello.o main.o -o world.out
cyb@DESKTOP-UOIAA21:~/test-makefile/test5$ ls
hello.c  hello.h  hello.o  main.c  main.o  makefile  world.out
cyb@DESKTOP-UOIAA21:~/test-makefile/test5$ ./world.out
start ...
hello world
exit ...
你好，世界
```

我们没有定义`hello.o`和`main.o`的规则，为什么`make`也能正常创建这两个文件?

因为`make`最初就是为了编译C程序而设计的，为了免去重复创建编译`.o`文件的规则，`make`内置了隐式规则（Implicit Rule），即遇到一个`xyz.o`时，如果没有找到对应的规则，就自动应用一个隐式规则：

```
xyz.o: xyz.c
    cc -c -o xyz.o xyz.c
```

`make`针对C、C++、ASM、Fortran等程序内置了一系列隐式规则，可以参考官方手册查看。

对于C程序来说，使用隐式规则有一个潜在问题，那就是无法跟踪`.h`文件的修改。如果我们修改了`hello.h`的定义，由于隐式规则`main.o: main.c`并不会跟踪`hello.h`的修改，导致`main.c`不会被重新编译，这个问题我们放到后面解决。

小结

针对C、C++、ASM、Fortran等程序，`make`内置了一系列隐式规则，使用隐式规则可减少大量重复的通用编译规则。

使用变量

当我们在`Makefile`中重复写很多文件名时，一来容易写错，二来如果要改名，要全部替换，费时费力。

编程语言使用变量来解决反复引用的问题，类似的，在`Makefile`中，也可以使用变量来解决重复问题。

以上一节的`Makefile`为例：

```
world.out: hello.o main.o
    cc -o world.out hello.o main.o

clean:
    rm -f *.o world.out
```

编译的最终文件`world.out`重复出现了3次，因此，完全可以定义一个变量来替换它：

```
TARGET = world.out

$(TARGET): hello.o main.o
    cc -o $(TARGET) hello.o main.o

clean:
    rm -f *.o $(TARGET)
```

变量定义用`变量名 = 值`或者`变量名 := 值`，通常变量名全大写。引用变量用`$(变量名)`，非常简单。

注意到`hello.o main.o`这个“列表”也重复了，我们也可以用变量来替换：

```
OBJS = hello.o main.o
TARGET = world.out

$(TARGET): $(OBJS)
    cc -o $(TARGET) $(OBJS)

clean:
    rm -f *.o $(TARGET)
```

如果有一种方式能让`make`自动生成`hello.o main.o`这个“列表”，就更好了。注意到每个`.o`文件是由对应的`.c`文件编译产生的，因此，可以让`make`先获取`.c`文件列表，再替换，得到`.o`文件列表：

```
# $(wildcard *.c) 列出当前目录下的所有 .c 文件: hello.c main.c
# 用函数 patsubst 进行模式替换得到: hello.o main.o
OBJS = $(patsubst %.c,%.o,$(wildcard *.c))
TARGET = world.out

$(TARGET): $(OBJS)
    cc -o $(TARGET) $(OBJS)

clean:
    rm -f *.o $(TARGET)
```

这样，我们每添加一个`.c`文件，不需要修改`Makefile`，变量`OBJS`会自动更新。

思考：为什么我们不能直接定义`OBJS = $(wildcard *.o)`让`make`列出所有`.o`文件？

内置变量

我们还可以用变量`$(CC)`替换命令`cc`：

```
$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS)
```

没有定义变量`CC`也可以引用它，因为它是`make`的内置变量（Builtin Variables），表示C编译器的名字，默认值是`cc`，我们也可以修改它，例如使用交叉编译时，指定编译器：

```
CC = riscv64-linux-gnu-gcc
...
```

自动变量

在`Makefile`中，经常可以看到`$@`、`$<`这样的变量，这种变量称为自动变量（Automatic Variable），它们在一个规则中自动指向某个值。

例如，`$@`表示目标文件，`$^`表示所有依赖文件，$< 表示第一个依赖文件，因此，我们可以这么写：

```
world.out: hello.o main.o
    cc -o $@ $^
```

在没有歧义时可以写`$@`，也可以写`$(@)`，有歧义时必须用括号，例如`$(@D)`。

为了更好地调试，我们还可以把变量打印出来：

```
world.out: hello.o main.o
    @echo '$$@ = $@' # 变量 $@ 表示target
    @echo '$$< = $<' # 变量 $< 表示第一个依赖项
    @echo '$$^ = $^' # 变量 $^ 表示所有依赖项
    cc -o $@ $^
```

执行结果输出如下：

```
$@ = world.out
$< = hello.o
$^ = hello.o main.o
cc -o world.out hello.o main.o
```

小结

使用变量可以让`Makefile`更加容易维护。

使用模式规则

前面我们讲了使用隐式规则可以让`make`在必要时自动创建`.o`文件的规则，但`make`的隐式规则的命令是固定的，对于`xyz.o: xyz.c`，它实际上是：

```
    $(CC) $(CFLAGS) -c -o $@ $<
```

能修改的只有变量`$(CC)`和`$(CFLAGS)`。如果要执行多条命令，使用隐式规则就不行了。

这时，我们可以自定义模式规则，它允许`make`匹配模式规则，如果匹配上了，就自动创建一条模式规则。

我们修改上一节的`Makefile`如下：

```
OBJS = $(patsubst %.c,%.o,$(wildcard *.c))
TARGET = world.out

$(TARGET): $(OBJS)
    cc -o $(TARGET) $(OBJS)

# 模式匹配规则：当make需要目标 xyz.o 时，自动生成一条 xyz.o:
xyz.c 规则:
%.o: %.c
    @echo 'compiling $<...'
    cc -c -o $@ $<

clean:
    rm -f *.o $(TARGET)
```

当`make`执行`world.out: hello.o main.o`时，发现没有`hello.o`文件，于是需要查找以`hello.o`为目标的规则，结果匹配到模式规则`%.o: %.c`，于是`make`自动根据模式规则为我们动态创建了如下规则：

```
hello.o: hello.c
    @echo 'compiling $<...'
    cc -c -o $@ $<
```

查找`main.o`也是类似的匹配过程，于是我们执行`make`，就可以用模式规则完成编译：

```
$ make
compiling hello.c...
cc -c -o hello.o hello.c
compiling main.c...
cc -c -o main.o main.c
cc -o world.out hello.o main.o
```

模式规则的命令完全由我们自己定义，因此，它比隐式规则更灵活。

但是，模式规则仍然没有解决修改`hello.h`头文件不会触发`main.c`重新编译的问题，这个依赖问题我们继续放到后面解决。

最后注意，模式规则是按需生成，如果我们在当前目录创建一个`zzz.o`文件，因为`make`并不会在执行过程中用到它，所以并不会自动生成`zzz.o: zzz.c`这个规则。

小结

使用模式规则可以灵活地按需动态创建规则，它比隐式规则更灵活。

自动生成依赖

前面我们讲了隐式规则和模式规则，这两种规则都可以解决自动把`.c`文件编译成`.o`文件，但都无法解决`.c`文件依赖`.h`文件的问题。

因为一个`.c`文件依赖哪个`.h`文件必须要分析文件内容才能确定，没有一个简单的文件名映射规则。

但是，要识别出`.c`文件的头文件依赖，可以用GCC提供的`-MM`参数：

```
$ cc -MM main.c
main.o: main.c hello.h
```

上述输出告诉我们，编译`main.o`依赖`main.c`和`hello.h`两个文件。

因此，我们可以利用GCC的这个功能，对每个`.c`文件都生成一个依赖项，通常我们把它保存到`.d`文件中，再用`include`引入到`Makefile`，就相当于自动化完成了每个`.c`文件的精准依赖。

我们改写上一节的`Makefile`如下：

```
# 列出所有 .c 文件:
SRCS = $(wildcard *.c)

# 根据SRCS生成 .o 文件列表:
OBJS = $(SRCS:.c=.o)

# 根据SRCS生成 .d 文件列表:
```

```
DEPS = $(SRCS:.c=.d)

TARGET = world.out

# 默认目标:
$(TARGET): $(OBJS)
    $(CC) -o $@ $^

# xyz.d 的规则由 xyz.c 生成:
%.d: %.c
    rm -f $@; \
    $(CC) -MM $< >$@.tmp; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.tmp > $@; \
    rm -f $@.tmp

# 模式规则:
%.o: %.c
    $(CC) -c -o $@ $<

clean:
    rm -rf *.o *.d $(TARGET)

# 引入所有 .d 文件:
include $(DEPS)
```

变量`$(SRCS)`通过扫描目录可以确定为`hello.c main.c`，因此，变量`$(OBJS)`赋值为`hello.o main.o`，变量`$(DEPS)`赋值为`hello.d main.d`。

通过`include $(DEPS)`我们引入`hello.d`和`main.d`文件，但是这两个文件一开始并不存在，不过，`make`通过模式规则匹配到`%.d: %.c`，这就给了我们一个机会，在这个模式规则内部，用`cc -MM`命令外加`sed`把`.d`文件创建出来。

运行`make`，首次输出如下：

```
$ make
Makefile:31: hello.d: No such file or directory
Makefile:31: main.d: No such file or directory
rm -f main.d; \
```

```
        cc -MM main.c >main.d.tmp; \
        sed 's,\(main\)\.o[ :]*,\1.o main.d : ,g' <
main.d.tmp > main.d; \
        rm -f main.d.tmp
rm -f hello.d; \
        cc -MM hello.c >hello.d.tmp; \
        sed 's,\(hello\)\.o[ :]*,\1.o hello.d : ,g' <
hello.d.tmp > hello.d; \
        rm -f hello.d.tmp
cc -c -o hello.o hello.c
cc -c -o main.o main.c
cc -o world.out hello.o main.o
```

`make`会提示找不到`hello.d`和`main.d`，不过随后自动创建出`hello.d`和`main.d`。`hello.d`内容如下：

```
hello.o hello.d : hello.c
```

上述规则有两个目标文件，实际上相当于如下两条规则：

```
hello.o : hello.c
hello.d : hello.c
```

`main.d`内容如下：

```
main.o main.d : main.c hello.h
```

因此，`main.o`依赖于`main.c`和`hello.h`，这个依赖关系就和我们手动指定的一致。

改动`hello.h`，再次运行`make`，可以触发`main.c`的编译：

```
$ make
rm -f main.d; \
        cc -MM main.c >main.d.tmp; \
        sed 's,\(main\)\.o[ :]*,\1.o main.d : ,g' <
main.d.tmp > main.d; \
        rm -f main.d.tmp
cc -c -o main.o main.c
cc -o world.out hello.o main.o
```

在实际项目中，对每个`.c`文件都可以生成一个对应的`.d`文件表示依赖关系，再通过`include`引入到`Makefile`，同时又能让`make`自动更新`.d`文件，有点蛋生鸡和鸡生蛋的关系，不过，这种机制能正常工作，除了`.d`文件不存在时会打印错误，有强迫症的同学肯定感觉不满意，这个问题我们后面解决。

小结

利用GCC生成`.d`文件，再用`include`引入`Makefile`，可解决一个`.c`文件应该如何正确触发编译的问题。

完善Makefile

上一节我们解决了自动生成依赖的问题，这一节我们对项目目录进行整理，把所有源码放入`src`目录，所有编译生成的文件放入`build`目录：

```
<project>
├── Makefile
├── build
└── src
    ├── hello.c
    ├── hello.h
    └── main.c
```

整理`Makefile`，内容如下：

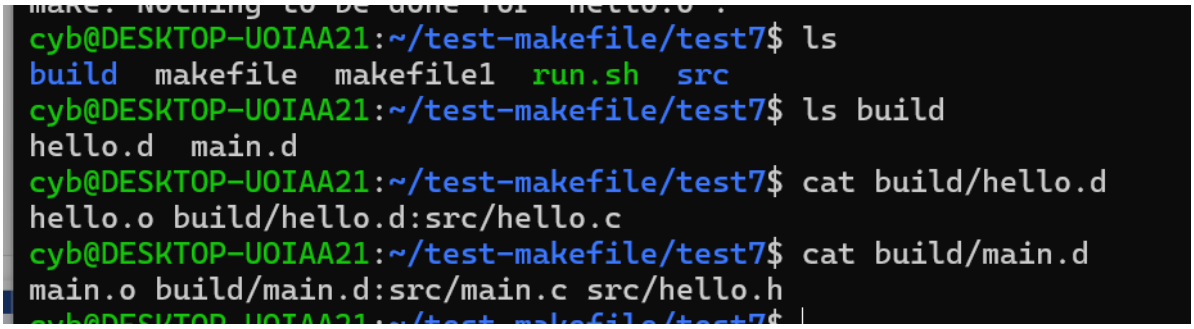
```
SRC_DIR = ./src
BUILD_DIR = ./build
TARGET = $(BUILD_DIR)/world.out

CC = cc
CFLAGS = -Wall

# ./src/*.c
SRCS = $(shell find $(SRC_DIR) -name '*.c')
# ./src/*.c => ./build/*.o
OBJS = $(patsubst $(SRC_DIR)/%.c,$(BUILD_DIR)/%.o,$(SRCS))
# ./src/*.c => ./build/*.d
DEPS = $(patsubst $(SRC_DIR)/%.c,$(BUILD_DIR)/%.d,$(SRCS))
```

```
# 默认目标:
all: $(TARGET)

# build/xyz.d 的规则由 src/xyz.c 生成:
$(BUILD_DIR)/%.d: $(SRC_DIR)/%.c
    @mkdir -p $(dir $@); \
    rm -f $@; \
    $(CC) -MM $< >$@.tmp; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.tmp > $@; \
    rm -f $@.tmp

# build/xyz.o 的规则由 src/xyz.c 生成:
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
    @mkdir -p $(dir $@)
    $(CC) $(CFLAGS) -c -o $@ $<

# 链接:
$(TARGET): $(OBJS)
    @echo "buiding $@..."
    @mkdir -p $(dir $@)
    $(CC) -o $(TARGET) $(OBJS)

# 清理 build 目录:
clean:
    @echo "clean..."
    rm -rf $(BUILD_DIR)

# 引入所有 .d 文件:
-include $(DEPS)
```

这个`Makefile`定义了源码目录`SRC_DIR`、生成目录`BUILD_DIR`，以及其他变量，同时用`-include`消除了`.d`文件不存在的错误。执行`make`，输出如下:

```
$ make
cc -Wall -c -o build/hello.o src/hello.c
cc -Wall -c -o build/main.o src/main.c
buiding build/world.out...
cc -o ./build/world.out ./build/hello.o ./build/main.o
```

可以说基本满足编译需求，收工！



Linux mkdir 命令

Linux 命令大全

Linux mkdir（英文全拼：make directory）命令用于创建目录。

语法

```
mkdir [-p] dirName
```

参数说明：

- -p 确保目录名称存在，不存在的就建一个。

实例

在工作目录下，建立一个名为 runoob 的子目录：

```
mkdir runoob
```

在工作目录下的 runoob2 目录中，建立一个名为 test 的子目录。

若 runoob2 目录原本不存在，则建立一个。（注：本例若不加 -p 参数，且原本 runoob2 目录不存在，则产生错误。）

```
mkdir -p runoob2/test
```

小结

除了基础的用法外，`Makefile`还支持条件判断，环境变量，嵌套执行，变量展开等各种功能，需要用到时可以查询官方手册。

Linux sed命令

Linux sed 命令是利用脚本来处理文本文件。

sed 可依照脚本的指令来处理、编辑文本文件。

Sed 主要用来自动编辑一个或多个文件、简化对文件的反复操作、编写转换程序等。

语法

```
sed [-hnV][-e<script>][-f<script文件>][文本文件]
```

参数说明：

- -e或--expression= 以选项中指定的script来处理输入的文本文件。
- -f<script文件>或--file=<script文件> 以选项中指定的script文件来处理输入的文本文件。
- -h或--help 显示帮助。
- -n或--quiet或--silent 仅显示script处理后的结果。
- -V或--version 显示版本信息。

动作说明：

- a ： 新增， a 的后面可以接字串，而这些字串会在新的一行出现(目前的下一行)～
- c ： 取代， c 的后面可以接字串，这些字串可以取代 n1,n2 之间的行！
- d ： 删除，因为是删除啊，所以 d 后面通常不接任何东东；**[n1,n2]**
- i ： 插入， i 的后面可以接字串，而这些字串会在新的一行出现(目前的上一行)；
- p ： 打印，亦即将某个选择的数据印出。通常 p 会与参数 sed -n 一起运行～
- s ： 取代，可以直接进行取代的工作哩！通常这个 s 的动作可以搭配正则表达式！例如 1,20s/old/new/g 就是啦！

实例

我们先创建一个 **testfile** 文件，内容如下：

```
$ cat testfile #查看testfile 中的内容
HELLO LINUX!
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
Google
Taobao
Runoob
Tesetfile
Wiki
```

在 **testfile** 文件的第四行后添加一行，并将结果输出到标准输出，在命令行提示符下输入如下命令：

```
sed -e 4a\newLine testfile
```

使用 **sed** 命令后，输出结果如下：

```
$ sed -e 4a\newLine testfile
HELLO LINUX!
Linux is a free unix-type opterating system.
This is a linux testfile!
Linux test
newLine
Google
Taobao
Runoob
Tesetfile
Wiki
```

以行为单位的新增/删除

将 **testfile** 的内容列出并且列印行号，同时，请将第 2~5 行删除！

```
$ nl testfile | sed '2,5d'
     1  HELLO LINUX!
     6  Taobao
     7  Runoob
     8  Tesetfile
     9  Wiki
```

sed 的动作为 **2,5d**，那个 **d** 是删除的意思，因为删除了 2-5 行，所以显示的数据就没有 2-5 行了， 另外，原本应该是要下达 sed -e 才对，但没有 -e 也是可以的，同时也要注意的是， sed 后面接的动作，请务必以 **'...'** 两个单引号括住喔!

只要删除第 2 行:

```
$ nl testfile | sed '2d'
     1  HELLO LINUX!
     3  This is a linux testfile!
     4  Linux test
     5  Google
     6  Taobao
     7  Runoob
     8  Tesetfile
     9  Wiki
```

要删除第 3 到最后一行:

```
$ nl testfile | sed '3,$d'
     1  HELLO LINUX!
     2  Linux is a free unix-type opterating system.
```

在第二行后(即加在第三行) 加上**drink tea?** 字样:

```
$ nl testfile | sed '2a drink tea'
     1  HELLO LINUX!
     2  Linux is a free unix-type opterating system.
drink tea
     3  This is a linux testfile!
     4  Linux test
     5  Google
     6  Taobao
     7  Runoob
     8  Tesetfile
     9  Wiki
```

如果是要在第二行前，命令如下：

```
$ nl testfile | sed '2i drink tea'
     1  HELLO LINUX!
drink tea
     2  Linux is a free unix-type opterating system.
     3  This is a linux testfile!
     4  Linux test
     5  Google
     6  Taobao
     7  Runoob
     8  Tesetfile
     9  Wiki
```

如果是要增加两行以上，在第二行后面加入两行字，例如 **Drink tea or ..... 与 drink beer?**

```
$ nl testfile | sed '2a Drink tea or ......\
drink beer ?'

1  HELLO LINUX!
     2  Linux is a free unix-type opterating system.
Drink tea or ......
drink beer ?
     3  This is a linux testfile!
     4  Linux test
     5  Google
     6  Taobao
```

```
     7  Runoob
     8  Tesetfile
     9  Wiki
```

每一行之间都必须要以反斜杠 '\' **来进行新行标记。上面的例子中，我们可以发现在第一行的最后面就有** '\'存在。

以行为单位的替换与显示

将第 **2-5** 行的内容取代成为 **No 2-5 number** 呢?

```
$ nl testfile | sed '2,5c No 2-5 number'
     1  HELLO LINUX!
No 2-5 number
     6  Taobao
     7  Runoob
     8  Tesetfile
     9  Wiki
```

透过这个方法我们就能够将数据整行取代了。

仅列出 testfile 文件内的第 5-7 行:

```
$ nl testfile | sed -n '5,7p'
     5  Google
     6  Taobao
     7  Runoob
```

可以透过这个 sed 的以行为单位的显示功能， 就能够将某一个文件内的某些行号选择出来显示。

数据的搜寻并显示

搜索 testfile 有 **oo** 关键字的行:

```
$ nl testfile | sed -n '/oo/p'
     5  Google
     7  Runoob
```

如果 root 找到，除了输出所有行，还会输出匹配行。

数据的搜寻并删除

删除 testfile 所有包含 **oo** 的行，其他行输出

```
$ nl testfile | sed  '/oo/d'
     1  HELLO LINUX!
     2  Linux is a free unix-type opterating system.
     3  This is a linux testfile!
     4  Linux test
     6  Taobao
     8  Tesetfile
     9  Wiki
```

数据的搜寻并执行命令

搜索 testfile，找到 **oo** 对应的行，执行后面花括号中的一组命令，每个命令之间用分号分隔，这里把 **oo** 替换为 **kk**，再输出这行：

```
$ nl testfile | sed -n '/oo/{s/oo/kk/;p;q}'
     5  Gkkgle
```

最后的 **q** 是退出。

数据的查找与替换

除了整行的处理模式之外， sed 还可以用行为单位进行部分数据的查找与替换<。

sed 的查找与替换的与 **vi** 命令类似，语法格式如下：

```
sed 's/要被取代的字串/新的字串/g'
```

将 testfile 文件中每行第一次出现的 oo 用字符串 kk 替换，然后将该文件内容输出到标准输出:

```
sed -e 's/oo/kk/' testfile
```

g 标识符表示全局查找替换，使 sed 对文件中所有符合的字符串都被替换，修改后内容会到标准输出，不会修改原文件：

```
sed -e 's/oo/kk/g' testfile
```

选项 **i** 使 sed 修改文件:

```
sed -i 's/oo/kk/g' testfile
```

批量操作当前目录下以 **test** 开头的文件:

```
sed -i 's/oo/kk/g' ./test*
```

接下来我们使用 /sbin/ifconfig 查询 IP:

```
$ /sbin/ifconfig eth0
eth0 Link encap:Ethernet HWaddr 00:90:CC:A6:34:84
inet addr:192.168.1.100 Bcast:192.168.1.255
Mask:255.255.255.0
inet6 addr: fe80::290:ccff:fea6:3484/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
.....(以下省略).....
```

本机的 ip 是 192.168.1.100。

将 IP 前面的部分予以删除:

```
$ /sbin/ifconfig eth0 | grep 'inet addr' | sed
's/^.*addr://g'
192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0
```

接下来则是删除后续的部分，即：**192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0**。

将 IP 后面的部分予以删除:

```
$ /sbin/ifconfig eth0 | grep 'inet addr' | sed
's/^.*addr://g' | sed 's/Bcast.*$//g'
192.168.1.100
```

特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的 `runoo*b` 中的 `*`，简单的说就是表示任何字符串的意思。如果要查找字符串中的 `*` 符号，则需要对 `*` 进行转义，即在其前加一个 `\`，`runo\*ob` 匹配字符串 **runo*ob**。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符 `\` 放在它们前面。下表列出了正则表达式中的特殊字符：

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，当该符号在方括号表达式中使用时，表示不接受该方括号表达式中的字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
|	指明两项之间的一个选择。要匹配 |，请使用 \\|。

限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 `*` 或 `+` 或 `?` 或 `{n}` 或 `{n,}` 或 `{n,m}` 共6种。

正则表达式的限定符有：

字符	描述	实例
*	匹配前面的子表达式零次或多次。例如，`zo*` 能匹配 "z" 以及 "zoo"。`*` 等价于 `{0,}`。	尝试一下 »
+	匹配前面的子表达式一次或多次。例如，`zo+` 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。`+` 等价于 `{1,}`。	尝试一下 »
?	匹配前面的子表达式零次或一次。例如，`do(es)?` 可以匹配 "do" 、 "does"、 "doxy" 中的 "do" 和 "does"。`?` 等价于 `{0,1}`。 	尝试一下 »
{n}	n 是一个非负整数。匹配确定的 n 次。例如，`o{2}` 不能匹配 "Bob" 中的 o，但是能匹配 "food" 中的两个 o。	尝试一下 »
{n,}	n 是一个非负整数。至少匹配n 次。例如，`o{2,}` 不能匹配 "Bob" 中的 o，但能匹配 "foooood" 中的所有 o。`o{1,}` 等价于 `o+`。`o{0,}` 则等价于 `o*`。	尝试一下 »
{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，`o{1,3}` 将匹配 "fooooood" 中的前三个 o。`o{0,1}` 等价于 `o?`。请注意在逗号和两个数之间不能有空格。	尝试一下 »

以下正则表达式匹配一个正整数，**[1-9]**设置第一个数字不是 0，`[0-9]*` 表示任意多个数字：

```
/[1-9][0-9]*/
```

多点编辑

一条 sed 命令，删除 testfile 第三行到末尾的数据，并把 HELLO 替换为 RUNOOB :

```
$ nl testfile | sed -e '3,$d' -e 's/HELLO/RUNOOB/'
     1  RUNOOB LINUX!
     2  Linux is a free unix-type opterating system.
```

-e 表示多点编辑，第一个编辑命令删除 testfile 第三行到末尾的数据，第二条命令搜索 HELLO 替换为 RUNOOB。

直接修改文件内容(危险动作)

sed 可以直接修改文件的内容，不必使用管道命令或数据流重导向！ 不过，由于这个动作会直接修改到原始的文件，所以请你千万不要随便拿系统配置来测试！ 我们还是使用文件 regular_express.txt 文件来测试看看吧!

regular_express.txt 文件内容如下：

```
$ cat regular_express.txt
runoob.
google.
taobao.
facebook.
zhihu-
weibo-
```

利用 sed 将 regular_express.txt 内每一行结尾若为 . 则换成 !

```
$ sed -i 's/\.$/\!/g' regular_express.txt
$ cat regular_express.txt
runoob!
google!
taobao!
facebook!
zhihu-
weibo-
```

:q:q

利用 sed 直接在 regular_express.txt 最后一行加入 **# This is a test**:

```
$ sed -i '$a # This is a test' regular_express.txt
$ cat regular_express.txt
runoob!
google!
taobao!
facebook!
zhihu-
weibo-
# This is a test
```

由於 $ 代表的是最后一行，而 a 的动作是新增，因此该文件最后新增 **# This is a test**!

sed 的 **-i** 选项可以直接修改文件内容，这功能非常有帮助！举例来说，如果你有一个 100 万行的文件，你要在第 100 行加某些文字，此时使用 vim 可能会疯掉！因为文件太大了！那怎办？就利用 sed 啊！透过 sed 直接修改/取代的功能，你甚至不需要使用 vim 去修订！

追加行的说明：

```
sed -e 4a\newline testfile
```

a 动作是在匹配的行之后追加字符串，追加的字符串中可以包含换行符（实现追加多行的情况）。

追加一行的话前后都不需要添加换行符 **\n**，只有追加多行时在行与行之间才需要添加换行符(最后一行最后也无需添加，添加的话会多出一个空行)。

man sed 信息：

```
Append text, which has each embedded newline preceded by a
backslash.
```

例如：

4 行之后添加一行：

```
sed -e '4 a newline' testfile
```

4 行之后追加 2 行:

```
sed -e '4 a newline\nnewline2' testfile
```

4 行之后追加 3 行(2 行文字和 1 行空行)

```
sed -e '4 a newline\nnewline2\n' testfile
```

4 行之后追加 1 行空行:

```
#错误：sed -e '4 a \n' testfile
sed -e '4 a \ ' testfile 实际上
```

实际上是插入了一个含有一个空格的行，插入一个完全为空的空行没有找到方法（不过应该没有这个需求吧，都要插入行了插入空行干嘛呢？）

添加空行:

```
# 可以添加一个完全为空的空行
sed '4 a \\'

# 可以添加两个完全为空的空行
sed '4 a \\n'
```

Makefile中的$@、$^、$<、$?、$%、$+、$*

$@ 表示目标文件

$^ 表示所有的依赖文件

$< 表示第一个依赖文件

$? 表示比目标还要新的依赖文件列表

\%”就是“bar.o”，“$@”就是“foo.a”。如果目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

$+ 这个变量很像“$^”，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$* 这个变量表示目标模式中“%”及其之前的部分。如果目标是“dir/a.foo.b”，并且目标的模式是“a.%.b”，那么，“$”的值就是“dir/a.foo”。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么“$”也就不能被推导出，但是，如果目标文件的后缀是make所识别的，那么“$”就是除了后缀的那一部分。例如：如果目标是“foo.c”，因为“.c”是make所能识别的后缀名，所以，“$”的值就是“foo”。这个特性是GNU make的，很有可能不兼容于其它版本的make，所以，你应该尽量避免使用“$*”，除非是在隐含规则或是静态模式中。如果目标中的后缀是make所不能识别的，那么“$”就是空值。

自动处理头文件的依赖关系

现在我们的Makefile写成这样：

```
all: main

main: main.o stack.o maze.o
    gcc $^ -o $@

main.o: main.h stack.h maze.h
stack.o: stack.h main.h
maze.o: maze.h main.h

clean:
    -rm main *.o

.PHONY: clean
```

按照惯例，用`all`做缺省目标。现在还有一点比较麻烦，在写`main.o`、`stack.o`和`maze.o`这三个目标的规则时要查看源代码，找出它们依赖于哪些头文件，这很容易出错，一是因为有的头文件包含在另一个头文件中，在写规则时很容易遗漏，二是如果以后修改源代码改变了依赖关系，很可能忘记修改Makefile的规则。为了解决这个问题，可以用`gcc`的`-M`选项自动生成目标文件和源文件的依赖关系：

```
$ gcc -M main.c
main.o: main.c /usr/include/stdio.h
/usr/include/features.h \
  /usr/include/sys/cdefs.h /usr/include/bits/wordsize.h \
  /usr/include/gnu/stubs.h /usr/include/gnu/stubs-32.h \
  /usr/lib/gcc/i486-linux-gnu/4.3.2/include/stddef.h \
  /usr/include/bits/types.h /usr/include/bits/typesizes.h
\
  /usr/include/libio.h /usr/include/_G_config.h
/usr/include/wchar.h \
  /usr/lib/gcc/i486-linux-gnu/4.3.2/include/stdarg.h \
  /usr/include/bits/stdio_lim.h
/usr/include/bits/sys_errlist.h main.h \
  stack.h maze.h
```

`-M`选项把`stdio.h`以及它所包含的系统头文件也找出来了，如果我们不需要输出系统头文件的依赖关系，可以用`-MM`选项：

```
$ gcc -MM *.c
main.o: main.c main.h stack.h maze.h
maze.o: maze.c maze.h main.h
stack.o: stack.c stack.h main.h
```

接下来的问题是怎么把这些规则包含到Makefile中，GNU `make`的官方手册建议这样写：

```
all: main

main: main.o stack.o maze.o
    gcc $^ -o $@

clean:
    -rm main *.o

.PHONY: clean

sources = main.c stack.c maze.c

include $(sources:.c=.d)
```

```
%.d: %.c
    set -e; rm -f $@; \
    $(CC) -MM $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

`sources`变量包含我们要编译的所有`.c`文件，`$(sources:.c=.d)`是一个变量替换语法，把`sources`变量中每一项的`.c`替换成`.d`，所以`include`这一句相当于：

```
include main.d stack.d maze.d
```

类似于C语言的`#include`指示，这里的`include`表示包含三个文件`main.d`、`stack.d`和`maze.d`，这三个文件也应该符合Makefile的语法。如果现在你的工作目录是干净的，只有`.c`文件、`.h`文件和`Makefile`，运行`make`的结果是：

```
$ make
Makefile:13: main.d: No such file or directory
Makefile:13: stack.d: No such file or directory
Makefile:13: maze.d: No such file or directory
set -e; rm -f maze.d; \
    cc -MM  maze.c > maze.d.$$; \
    sed 's,\(maze\)\.o[ :]*,\1.o maze.d : ,g' < maze.d.$$
> maze.d; \
    rm -f maze.d.$$
set -e; rm -f stack.d; \
    cc -MM  stack.c > stack.d.$$; \
    sed 's,\(stack\)\.o[ :]*,\1.o stack.d : ,g' <
stack.d.$$ > stack.d; \
    rm -f stack.d.$$
set -e; rm -f main.d; \
    cc -MM  main.c > main.d.$$; \
    sed 's,\(main\)\.o[ :]*,\1.o main.d : ,g' < main.d.$$
> main.d; \
    rm -f main.d.$$
cc    -c -o main.o main.c
```

```
cc     -c -o stack.o stack.c
cc     -c -o maze.o maze.c
gcc main.o stack.o maze.o -o main
```

一开始找不到`.d`文件，所以`make`会报警告。但是`make`会把`include`的文件名也当作目标来尝试更新，而这些目标适用模式规则`%.d: %c`，所以执行它的命令列表，比如生成`maze.d`的命令：

```
set -e; rm -f maze.d; \
    cc -MM  maze.c > maze.d.$$; \
    sed 's,\(maze\)\.o[ :]*,\1.o maze.d : ,g' < maze.d.$$
> maze.d; \
    rm -f maze.d.$$
```

注意，虽然在Makefile中这个命令写了四行，但其实是一条命令，`make`只创建一个Shell进程执行这条命令，这条命令分为5个子命令，用`;`号隔开，并且为了美观，用续行符`\`拆成四行来写。执行步骤为：

1. `set -e`命令设置当前Shell进程为这样的状态：如果它执行的任何一条命令的退出状态非零则立刻终止，不再执行后续命令。
2. 把原来的`maze.d`删掉。
3. 重新生成`maze.c`的依赖关系，保存成文件`maze.d.1234`（假设当前Shell进程的id是1234）。注意，在Makefile中`$`有特殊含义，如果要表示它的字面意思则需要写两个，所以$Makefile$中的四个传给Shell变成两个，两个在Shell中表示当前进程的id，一般用它给临时文件起名，以保证文件名唯一。
4. 这个`sed`命令比较复杂，就不细讲了，主要作用是查找替换。`maze.d.1234`的内容应该是`maze.o: maze.c maze.h main.h`，经过`sed`处理之后存为`maze.d`，其内容是`maze.o maze.d: maze.c maze.h main.h`。
5. 最后把临时文件`maze.d.1234`删掉。

不管是Makefile本身还是被它包含的文件，只要有一个文件在`make`过程中被更新了，`make`就会重新读取整个Makefile以及被它包含的所有文件，现在`main.d`、`stack.d`和`maze.d`都生成了，就可以正常包含进来了（假如这时还没有生成，`make`就要报错而不是报警告了），相当于在Makefile中添了三条规则：

```
main.o main.d: main.c main.h stack.h maze.h
maze.o maze.d: maze.c maze.h main.h
stack.o stack.d: stack.c stack.h main.h
```

如果我在`main.c`中加了一行`#include "foo.h"`，那么：

1、`main.c`的修改日期变了，根据规则`main.o main.d: main.c main.h stack.h maze.h`要重新生成`main.o`和`main.d`。生成`main.o`的规则有两条：

```
main.o: main.c main.h stack.h maze.h
%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

第一条是把规则`main.o main.d: main.c main.h stack.h maze.h`拆开写得到的，第二条是隐含规则，因此执行`cc`命令重新编译`main.o`。生成`main.d`的规则也有两条：

```
main.d: main.c main.h stack.h maze.h
%.d: %.c
    set -e; rm -f $@; \
    $(CC) -MM $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

因此`main.d`的内容被更新为`main.o main.d: main.c main.h stack.h maze.h foo.h`。

2、由于`main.d`被Makefile包含，`main.d`被更新又导致`make`重新读取整个Makefile，把新的`main.d`包含进来，于是新的依赖关系生效了。

由于GNU Make中文手册触发深入理解sed

http://mp.blog.csdn.net/postedit?ref=toolbar

最近由于要分析Uboot的代码。

于是乎，再一次开始复习《GNU Make中文手册》（）

第一次看这本手册是在快一年前的事情了，当时是啥都不懂。一头雾水。

这次细细品味的时候，发现收获颇多。建议初学者去多看看。

今天看到《4.14 自动产生依赖》的时候，一段代码在一次让我郁闷了。同样的地方，同样的不理解。

今天偶就要好好揭开这个惑!

代码如下:

```
1 %.d: %.c

2         $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \

3         sed 's,$*\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;
\

4         rm -f $@.
```

其实这里主要是为每个C文件建立一个同名的后缀为.d。该文件的作用是使用gcc的-M属性来自动生成.o文件的头文件依赖关系。

对于第3行， 我知道sed的s命令是一个替换命令。但是里面的用到了太多高深的匹配规则了。 sed命令果真如传闻中的那么强大，对于现在的我来说还真的很陌生。不管咋样， 要把它解决。

首先，我们先要知道sed是什么概念。

sed是一个非交互式的流编辑器。所谓非交互式，是指使用sed只能在命令行下输入编辑命令来编辑文本，然后在屏幕上查看输出；而流编辑器是指sed每次只从文件（或输入）读入一行，然后对该行进行指定的处理，并将结果输出到屏幕，接着读入下一行。

为了简化的阐述，下面将静态模式用一个特例代替---main.c 。 通过第2行，针对main.c编译器生成了如下的依赖关系:

main.o:main.c defs.h

而通过第三行将会被替换成main.o:main.d:main.c defs.h, 并且把这个依赖关系输出到文件main.d中。

OK，大致知道了它的意思，接下在，就细细的分析第三行命令的整个执行过程，如下：

1：将($@.

)的临时文件中的字符串信息(main.o:main.c defs.h)通过 “<” 输送到sed命令中.

2：sed中的s符号告诉sed命令，这次要做一个替换的任务。s符号的格式为：[address[,address]] s/pattern-to-find/replacement-pattern/[g p w n]。 下面来匹配上面的示例：

[address[,address]]：是指要处理的行的范围，在这次的操作中采用的是默认值。

pattern-to-find等价于**.o[:]*

replacement-pattern等价于\1.o $@ ：

3：Makefile使用%=main进行替换后，命令变成了sed 's,main����.o[:]*,\1.o main.d ： ，g' < main.pid > main.d ;

接下来就比较好分析了，主要是正则表达式的知识了。 pattern-to-find使用到了4个正则表示式的知识点。

first, main����为创建一个字符标签，给后边的replacement-pattern使用。如\1.o，展开后就是main.o

second, . 在正则表达式中‘.’作用是匹配一个字符。所以需要使用转义元字符‘\’来转义。

third, [:] 匹配一组字符里的任意字符 。

forth, *匹配0个或多个前一字符

4： 通过sed的正则表达式，输入的main.o:main.c defs.h被替换成了main.o main.d : main.c defs.h。

这里还有个有趣的东西，平时我们对命令s符号使用‘/’作为参数分割符，其实‘/’只是一种默认的习惯罢了。你也可以使用','来作为分割符号，只要前后统一就OK。这里就是使用了','来作为分割符。

这里我们使用了make的隐含规则来编译.c的源文件。对变量的赋值也用到了一个特殊的符号（:=）。

1、wildcard : 扩展通配符

2、notdir： 去除路径

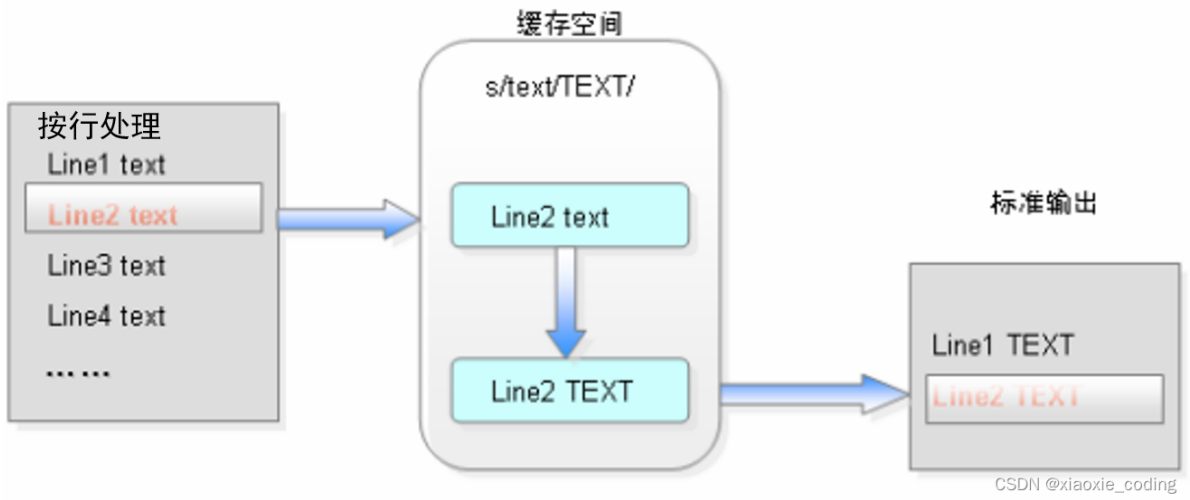
3、patsubst： 替换通配符

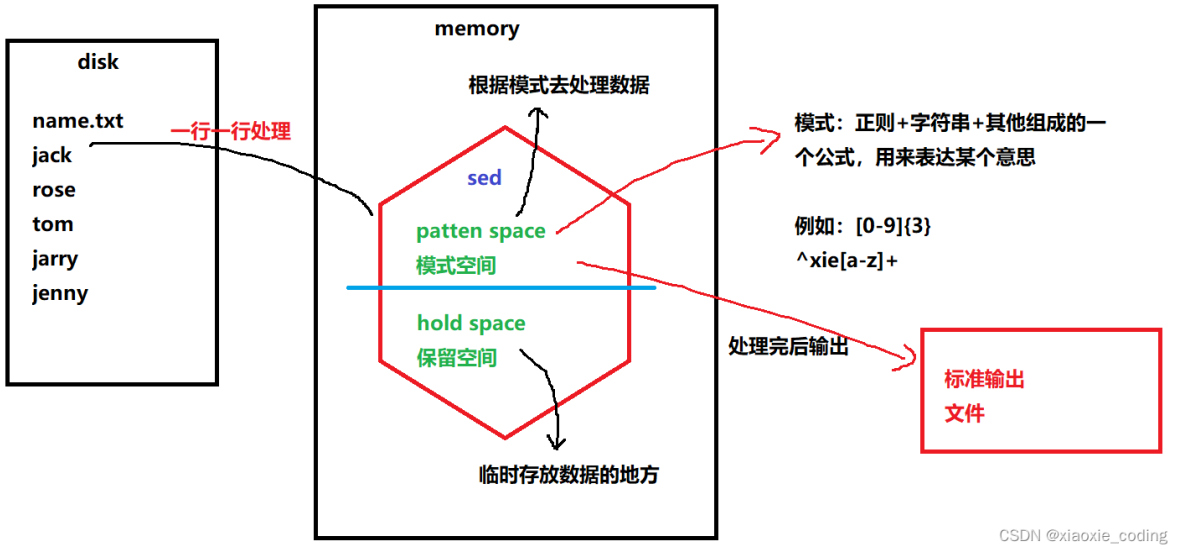
Linux文本处理三剑客之———sed详解

Linux文本处理三剑客之———sed详解——sed看这一篇就够啦~PS:文末有练习，来练练手吧_linux sed-CSDN博客

sed --> 文本替换的命令，是支持正则表达式的非交互式流编辑器(stream editor)

sed - stream editor for filtering and transforming text 可以用来过滤和转换文本





模式空间：可以想成加工车间，根据某种模式将数据进行处理。

保持空间：可以想成仓库，我们在进行数据处理的时候，临时存放数据的地方

PS：正常情况下，如果不显示使用某些高级命令，保持空间不会使用到！

模式空间处理完一行数据，马上就是清空里面的内容，接着，再重复执行刚才的动作，文件中的新的一行被读入，直到文件处理完毕。

sed语法命令格式

- sed [选项] sed编辑命令 输入文件
- shell 命令 | sed [选项] sed编辑命令
- sed [选项] -f sed脚本文件 输入文件

sed常用选项

- -n：只显示匹配处理的行（否则会输出所有）
- -e：执行多个编辑命令时（一般用；代替）
- -i：直接在文件中进行修改，而不是输出到屏幕
- -r：支持扩展正则表达式
- -f：从脚本文件中读取内容并执行（文件中的编辑命令每行一个,不用;隔开）
- sed的常用编辑命令
- p：打印匹配行 print
- d：删除指定行 delete
- a：在匹配行后面追加 append
- i：在匹配行前面插入 insert
- c：整行替换
- r：将文件的内容读入 read
- w：将文本写入文件 write
- s：字符串替换（匹配正则表达式）substitution
- =：输出行号

sed查找方式

- 根据行号
- 根据模式 (正则表达式=字符+特殊符号)
- 根据字符串

sed的p命令示例

根据行号：sed -n '行号1，行号2p' 输入文件

```
# 输出第1行
sed -n '1p' /etc/passwd

# 输出一到五行
sed -n '1,5p' /etc/passwd

# 输出最后一行
sed -n '$p' /etc/passwd

# 输出第四行及其后面五行
sed -n '4,+5p' /etc/passwd

# 只显示1~3行，4到最后一行都不显示
sed -n '4,$!p' /etc/passwd

# 不连续输出，只输出1，3，5行
sed -n '1p;3p;5p' /etc/passwd

# 可以设置步长值，输出单数行，步长为2
cat -n /etc/passwd|sed -n '1~2p'
```

根据模式：sed -n '/模式/p' 输入文件

```
# 输出有root的行
sed -n '/root/p' /etc/passwd

# 以#或者$开头的行不显示
cat /etc/ssh/sshd_config |sed -rn '/^#|^$/!p'

# 显示以/结尾的行，需要转义
df -Th| sed -n '/\/$/p'
```