Makefile教程

入门Linux开发，从零开始编写Makefile！

廖雪峰

2025-06-16

https://liaoxuefeng.com/books/makefile/

目录

简介

原文链接

Linux的 `make` 程序用来自动化编译大型源码，很多时候，我们在Linux下编译安装软件，只需要敲一个 `make` 就可以全自动完成，非常方便。

`make` 能自动化完成这些工作，是因为项目提供了一个 `Makefile` 文件，它负责告诉 `make` ，应该如何编译和链接程序。

`Makefile` 相当于Java项目的 `pom.xml` ，Node工程的 `package.json` ，Rust项目的 `Cargo.toml` ，不同之处在于， `make` 虽然最初是针对C语言开发，但它实际上并不限定C语言，而是可以应用到任意项目，甚至不是编程语言。此外， `make` 主要用于Unix/Linux环境的自动化开发，掌握 `Makefile` 的写法，可以更好地在Linux环境下做开发，也可以为后续开发Linux内核做好准备。

在本教程中，我们将由浅入深，一步一步学习如何编写 `Makefile` ，完全针对零基础小白，只需要提前掌握如何使用Linux命令。

评论

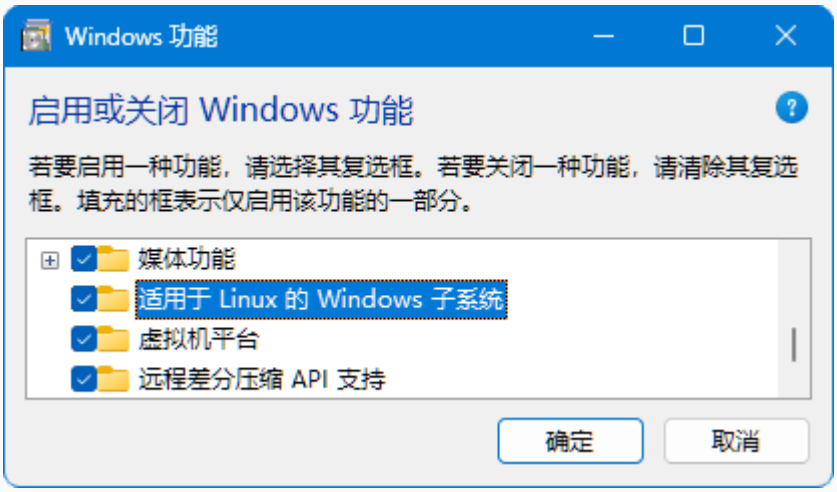
安装make

原文链接

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：

```
$ make -v
GNU Make 4.3
```

```
...
```

输入 `gcc --version` 验证GCC工具链：

```
$ gcc --version
gcc (Ubuntu ...) 11.4.0
...
```

这样，我们就成功地安装了 `make` ，以及GCC工具链。

评论

Makefile基础

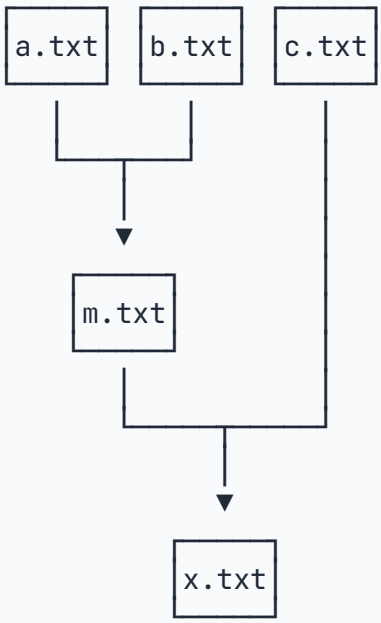
原文链接

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
```

一条规则的格式为 `目标文件：依赖文件1 依赖文件2 ...` ，紧接着，以Tab开头的是命令，用来生成目标文件。上述规则使用 `cat` 命令合并了 `a.txt` 与 `b.txt` ，并写入到 `m.txt` 。用什么方式生成目标文件 `make` 并不关心，因为命令完全是我们自己写的，可以是编译命令，也可以是 `cp` 、 `mv` 等任何命令。

以 `#` 开头的是注释，会被 `make` 命令忽略。

> ⚠ 注意
>
> 在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
```

`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.
```

`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。

> 提示
>
> 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！

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。

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。

参考源码

可以从GitHub下载源码。

GitHub

小结

编写 `Makefile` 就是编写一系列规则，用来告诉 `make` 如何执行这些规则，最终生成我们期望的目标文件。

查看官方手册：

- 编写规则
- 执行命令
- 伪目标

评论

编译C程序

原文链接

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：

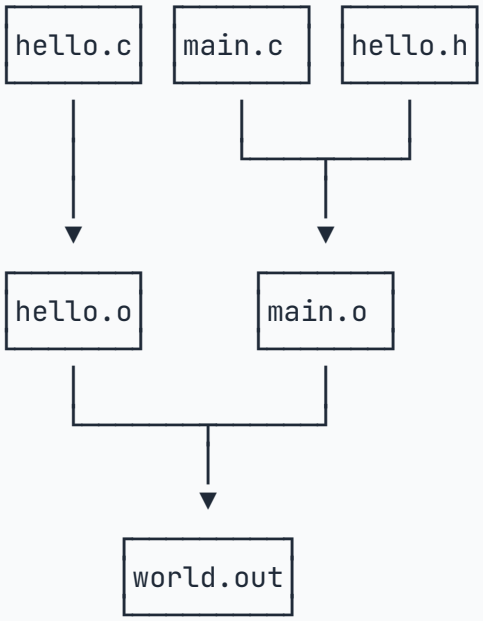
```
int hello();
```

`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:
    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。

参考源码

可以从GitHub下载源码。

GitHub

小结

在 `Makefile` 正确定义规则后，我们就能用 `make` 自动化编译C程序。

评论

使用隐式规则

原文链接

我们仍然以上一节的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
```

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。

参考源码

可以从GitHub下载源码。

GitHub

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。

查看官方手册：

- 使用隐式规则

评论

使用变量

原文链接

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（Variable）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，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
```

参考源码

可以从GitHub下载源码。

GitHub

小结

使用变量可以让 `Makefile` 更加容易维护。

查看官方手册：

- 如何使用变量
- 自动变量

评论

使用模式规则

原文链接

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（Pattern Rules），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。

参考源码

可以从GitHub下载源码。

GitHub

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。

查看官方手册：

- 模式规则

评论

自动生成依赖

原文链接

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。

参考源码

可以从GitHub下载源码。

GitHub

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。

查看官方手册：

- 自动生成依赖

评论

完善Makefile

原文链接

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[ :]*,$(BUILD_DIR)/\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!

参考源码

可以从GitHub下载源码。

GitHub

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。

评论

Makefile教程

入门Linux开发，从零开始编写Makefile！

Author: 廖雪峰

Version: 2025-06-16

Website: https://liaoxuefeng.com/books/makefile/