Makefile 教程

零基础入门系列教程

作者: 魏明择

2025 年版

https://weimingze.com

目录

序

前言

第一章、Makefile 入门

1. 安装 Makefile
2. 最简 Makefile
3. 规则
4. 多规则 Makefile
5. 伪目标

第二章、变量和模式

1. 变量
2. 自动变量
3. 隐含规则
4. 通配符
5. 模式规则
6. 静态模式规则

第三章、函数

1. 函数
    1.1 wildcard 函数
    1.2 patsubst 函数
2. 预置函数

第四章、高级用法

1. 命令回显
2. vpath 指令和 VPATH 变量
3. 条件判断
4. 修改默认 Shell
5. 使用 Shell 变量
6. 命令的执行

附录

Makefile 实战

序

前言

Makefile 是用于自动化编译和构建项目的工具。Makefile 主要用于 C/C++文件的编译场景中。它将构建项目的**规则**写入一个或多个名为 `makefile` 或 `Makefile` 的文件中，然后通过 `make` 命令就可以调用这些规则来构建项目。

Makefile 是 UNIX/Linux 系统中用于自动化构建项目或编译和安装 开源项目的常用工具。

我们有如下的 C 语言的源代码程序，现在需要将其编译链接成为一个应用程序 `myapp`。

源码文件如下

```
myapp/
├── file1.c
├── file2.c
└── main.c
```

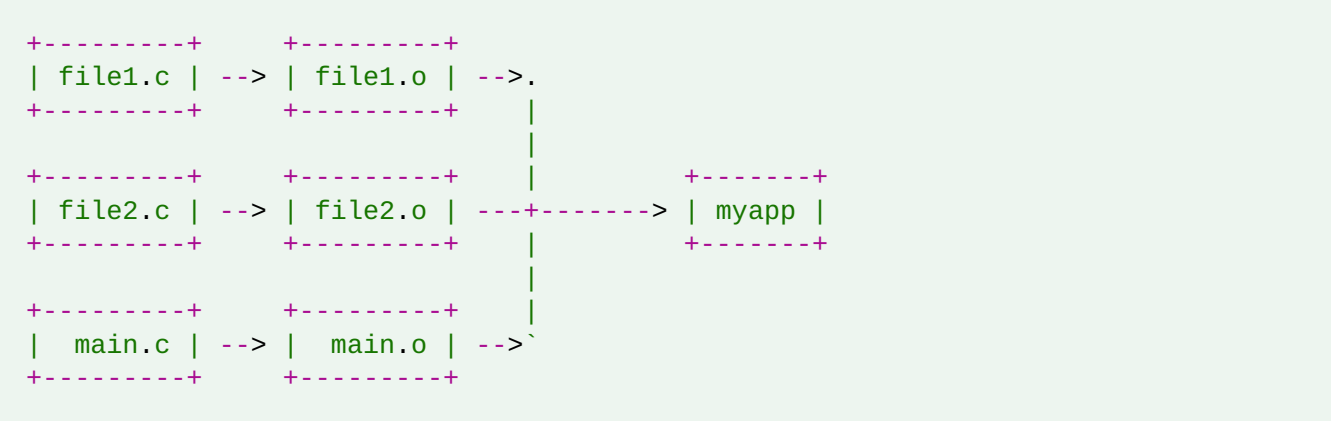
我们手动编译上述程序需要将 `file1.c`、`file2.c` 和 `main.c` 生成目标文件 `file1.o`、`file2.o` 和 `main.o`，然后再将 三个 `.o` 文件链接成一个应用程序 `myapp`

如下步骤:

```
weimingze@mzstudio:~/myapp$ ls
file1.c  file1.h  file2.c  file2.h  main.c
weimingze@mzstudio:~/myapp$ gcc -c file1.c
weimingze@mzstudio:~/myapp$ gcc -c file2.c
weimingze@mzstudio:~/myapp$ gcc -c main.c
weimingze@mzstudio:~/myapp$ gcc -o myapp main.o file1.c file2.c
weimingze@mzstudio:~/myapp$ ls
file1.c  file1.h  file1.o  file2.c  file2.h  file2.o  main.c  main.o  myapp
```

> 注：`weimingze@mzstudio:~/myapp$` 是我的终端的命令行提示符。
>
> 如果你的环境中没有 `gcc` 这个命令，请用如下命令安装。
>
> 1、在 Ubuntu Linux 下可以使用 `sudo apt install gcc` 安装。
>
> 2、在 CentOS Linux 下可以使用 `sudo yum install gcc` 安装。
>
> 3、在 Mac OS 下可以使用 `brew install gcc` 安装。

上述文件编译的过程是这样的



可见手动编译和链接这些文件比较麻烦。如果文件超过几百个，那么编译的工作量将是巨大的。解决这个问题的办法是使用 Makefile。

Makefile 常用于如下场景:

1. C/C++ 语言编译和链接。
2. Go 语言管理和构建。
3. Qt 应用程序的构建。
4. Linux 内核编译。

本课程将用最简单的方式，逐步深入的学习 Makefile 的用法。

第一章、Makefile 入门

1. 安装 Makefile

通常在 Mac OS 或 Linux 系统下使用 `Makefile` 是通过 `make` 命令来完成的。因此在你学习和使用 `Makefile` 前要确保你的电脑能够正常执行 `make` 命令。

需要注意的是 `make` 是基于终端的命令，在安装和使用时要通过命令行终端来进行。

Ubuntu Linux 下安装 Makefile

```
sudo apt install make
```

CentOS 下安装 Makefile

```
sudo yum install make
```

Mac OS 下安装 Makefile

安装 Xcode 命令行工具

```
xcode-select --install
```

验证 Makefile 是否成功安装

在 Mac OS 或 Linux 系统下打开终端，输入 `make -v` 命令查看 `make` 命令的版本信息，如果能够显示 `make` 的版本信息，则说明你已经安装了 `make` 命令。如果提示 `make` 命令没有找到，则说明你需要按上述方法安装 `make` 命令。

示例

Ubuntu Linux 成功安装的输出。

```
weimingze@mzstudio:~$ make -v
GNU Make 4.3
为 x86_64-pc-linux-gnu 编译
Copyright (C) 1988-2020 Free Software Foundation, Inc.
许可证：GPLv3+：GNU 通用公共许可证第 3 版或更新版本<http://gnu.org/licenses/gpl.html>。
```

```
本软件是自由软件：您可以自由修改和重新发布它。
在法律允许的范围内没有其他保证。
```

Mac OS 成功安装的输出。

```
weimz@mzstudio ~ % make -v
GNU Make 3.81
Copyright (C) 2006  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.

This program built for i386-apple-darwin11.3.0
```

实验：

在你的电脑上安装 `Makefile` 的执行命令 `make`，然后验证是否安装成功。

2. 最简 Makefile

Makefile 是由的核心组成是名为 **规则** 的结构，一个 Makefile 可以有一个或多个 **规则** 组成。

本节课的目标是学会 `Makefile` 的基本语法和使用方法，以及运用规则。

下面我们编写一个 `Makefile` 文件。内容如下:

```
hello :
    echo "Hello World!"
```

> 命令 `echo "Hello World!"` 是在控制台终端打印 `Hello World!` 一行文字。

说明:

- 上述 Makefile 文件的功能是运行 `make` 命令时，如果当前文件夹下不存在 `hello` 这个文件，则执行 `echo "Hello World!"` 命令，在终端上打印一行 `Hello World!`
- 需要注意 `echo "Hello World!"` 命令前面是一个制表符（`<tab>` 键对应的字符），而不是多个空格。你也可以点击这个 `Makefile` 下载此文件。

在当前的 `Makefile` 文件所在的文件夹内执行 `make` 命令，看到执行结果如下:


```
weimingze@mzstudio:~$ make
echo "Hello World!"
Hello World!
```

说明:

- 上述执行结果中 `echo "Hello World!"` 是命令执行的回显。`Hello World!` 是该命令执行的结果。

可见 规则 `hello :` 对应的 命令 `echo "Hello World!"` 正确的执行了。

make 的查找规则

在执行 `make` 命令时，标准的 `make` 命令查找本地文件的顺序是 `makefile`、 `Makefile` 文件。这两个文件名任选其一就可以了。

本课程所有示例都使用首字母大写的 `Makefile` 作为文件名。

注释的语法

在 Makefile 中可以注释，注释中的代码不会参与 Makefile 的执行。

Makefile 的注释是以英文的井号（#）开头，直至行尾。

> 这种注释的语法和 `Shell`、`Python` 的注释语法相同。

如:

```
# 这是一个 makefile 的最简示例的注释
hello :
    echo "Hello World!"
```

实验:

在上述 `Makefile` 所在的文件夹内使用 `touch hello` 命令，创建文件 `hello`，然后再执行 `make` 命令，查看规则命令的执行情况。

3. 规则

要学会使用 Makefile 首先你要了解**规则**和定义**规则**。本节课我们来学习规则定义的语法和用法。

我们以编译 C 语言的应用程序为例。首先我们使用编辑器编写一个 C 语言源文件 `hello.c`

内容如下：

文件: `hello.c`

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello World!\n");
    return 0;
}
```

下面是在我的电脑上手动进行编译成可执行程序 `hello` 并进行运行的方法。

```
weimingze@mzstudio:~$ ls
hello.c
weimingze@mzstudio:~$ gcc -o hello hello.c
weimingze@mzstudio:~$ ls
hello  hello.c
weimingze@mzstudio:~$ ./hello
Hello World!
```

> 注:
>
> `ls` 命令是列出当前文件夹下所有的文件。
>
> `gcc -o hello hello.c` 是编译 `hello.c` 并输出为 `hello` 可执行程序。
>
> `./hello` 是运行 编译后生成的 `hello` 程序文件。

可见，上述 `gcc -o hello hello.c` 的目标是 使用 `hello.c` 生成 `hello`。

下面我们把上述编译规则写入 `Makefile` 文件，然后使用 `make` 命令进行编译。

先来看 **规则语法：**

```
目标文件1 [目标文件2]: [依赖文件1 依赖文件2 ...]
<制表符>命令1
<制表符>命令2
...
```

或者

```
目标文件1 [目标文件2]: [依赖文件1 依赖文件2 ...] ; 命令1
<制表符>命令2
...
```

说明：

1. `[]` 表示其中的内容可以省略。
2. 冒号（`:`） 是**目标文件列表**和**依赖文件列表**的分隔符。
3. **目标文件**是要生成的文件，可以是一个或多个（大多只有一个），用空格分隔开，如：`hello`。
   - 当一个规则含有多个目标时，会为每个目标单独执行相应的命令，各个目标共享依赖。自动变量 `$@` 表示当前处理的目标（后面会讲）
4. **依赖文件**通常是为生成目标文件所需要的文件，可以有零个、一个或多个文件，用空格分隔。如：`hello.c`。
5. **命令** 通常是用来生成目标文件所需要的命令，可以写多条命令，每一条要写在一行内。如：`gcc -o hello hello.c`。
6. 命令前必须使用制表符（`<tab>` 键打出来的符号），不能是空格。
7. 一个 Makefile 文件中的规则语法可以有多个。如果在使用 `make` 命令时不指定目标文件，则默认执行第一个规则。

示例

文件：`Makefile`

```
hello : hello.c
    gcc -o hello hello.c
    echo "compile complete!"
```

make 命令的用法

```
make [目标文件]
```

Makefile 规则命令的执行顺序

1. 先检查**目标文件**，如果目标文件不存在，则执行此规则对应的命令。
2. 如果目标文件存在，则对比目标文件和依赖文件的修改时间，如果目标文件的修改时间只要比其中一个依赖文件的修改时间早（目标文件比较旧），则执行此规则对应的命令。
3. 否则放弃执行规则对应的命令。

`make` 两次执行的结果如下:

```
weimingze@mzstudio:~$ ls
hello.c  Makefile
weimingze@mzstudio:~$ make
gcc -o hello hello.c
echo "compile complete!"
compile complete!
weimingze@mzstudio:~$ ls
hello  hello.c  Makefile
weimingze@mzstudio:~$ ls -l hello*
-rwxrwxr-x 1 weimingze weimingze 15960 12月 18 21:11 hello
-rw-rw-r-- 1 weimingze weimingze   101 12月 18 19:29 hello.c
weimingze@mzstudio:~$ make
make: “hello”已是最新。
```

从执行结果可知。Makefile 中只有这一条规则，第一次执行 `make` 命令时，因为 不存在文件 `hello`，所以规则对应的命令都执行了，并生成了 `hello`。此时文件 `hello` 的修改时间是 `12月 18 21:11` 比文件 `hello.c`的修改时间 `12月 18 19:29` 新，因此第二次在执行 `make` 命令时，对应的规则命令不会执行。

尝试重新编辑 `hello.c` 并保存，此时再运行 `make` 命令，则此规则对应的命令又会重新执行。原因是重新保存 `hello.c` 文件的同时会更新此文件的修改时间为当前时间。

实验:

完成上述示例程序的编写，尝试使用 touch 命令，修改文件 `hello.c` 和 `hello` 的时间，然后再使用 `make` 命令进行编译，查看规则命令的执行情况。

4. 多规则 Makefile

本节课我们讲解多规则 Makefile 的编写。

上节课使用 `gcc -o hello hello.c` 命令将 `hello .c` 编译成 `hello` 文件看似编译过程是这样的:

```
+---------+                    +-------+
| hello.c | ------------------> | hello |
+---------+                    +-------+
```

而实际编辑过程中会生成目标文件 `hello.o`，也就是说命令 `gcc -o hello hello.c` 实际是由 `gcc -o hello.o -c hello.c` 和 `gcc -o hello hello.o` 两条命令组合而成。

实际编译过程如下图所示:

```
+---------+     +---------+     +-------+
| hello.c | --> | hello.o | --> | hello |
+---------+     +---------+     +-------+
```

我们改写 `Makefile` 文件如下:

```
hello : hello.o
    gcc -o hello hello.o

hello.o : hello.c
    gcc -o hello.o -c hello.c
```

上述 Makefile 文件中有两个规则，其中规则 `hello : hello.o` 通过 `hello.o` 生成 `hello`；而规则 `hello.o : hello.c` 则是通过 `hello.c` 生成 `hello.o`。在执行 `make` 命令时，实际执行的是第一个规则。但第一个规则发现没有 `hello.o` 这个文件，则 `make` 会再以 `hello.o` 作为目标来执行第二个规则 `hello.o : hello.c`。

`Makefile` 为了生成目标，通常会形成了一个依赖树。这个树中的任何一个文件更新。都会执行对应的规则来生成新的目标。而对于不需要更新的目标则不会执行对应的**规则**。

执行结果如下:

```
weimingze@mzstudio:~$ ls
Makefile    hello.c
weimingze@mzstudio:~$ make
gcc -o hello.o -c hello.c
gcc -o hello hello.o
weimingze@mzstudio:~$ ls
Makefile    hello       hello.c     hello.o
weimingze@mzstudio:~$ make
make: “hello”已是最新。
```

可见 执行 make 后，会生成 `hello.o` 和 `hello` 这两个文件，两个规则都执行了。

在 使用 Makefile 时，并不是所有的规则都一定会执行。而只有需要更新的目标对应的规则才会执行。如 现在我们使用 `rm hello` 命令删除最终的目标文件 `hello` 而保留 `hello.o`。再次 `make` 则只会调用 规则 `hello : hello.o`，而不会执行规则 `hello.o : hello.c`。这样可以避免没有必要的更新以节省 `make` 执行的时间。如:

```
weimingze@mzstudio:~$ rm hello
weimingze@mzstudio:~$ ls
Makefile  hello.c  hello.o
weimingze@mzstudio:~$ make
gcc -o hello hello.o
```

```
weimingze@mzstudio:~$ ls
Makefile  hello  hello.c  hello.o
```

生成指定的目标

默认 `make` 命令会 执行 `Makefile` 中的第一个规则。我们也可以使用 `make 目标文件` 的方式来指定执行的第一个规则，如上述程序中我们使用 `rm` 命令删除 `hello` 和 `hello.o` 两个文件，然后只使用 `make hello.o` 命令来生成 `hello.o` 而 `hello` 则不会生成。

执行结果如下:

```
weimingze@mzstudio:~$ rm hello hello.o
weimingze@mzstudio:~$ ls
Makefile  hello.c
weimingze@mzstudio:~$ make hello.o
gcc -o hello.o -c hello.c
weimingze@mzstudio:~$ ls
Makefile  hello.c  hello.o
```

通过上述执行结果。可见使用 `make 目标文件` 的方式可以将 `Makefile` 中的任意规则当成**第一执行规则**。

实验:

将上述 `Makefile` 的两个规则上下颠倒，改写如下：

```
hello.o : hello.c
    gcc -o hello.o -c hello.c

hello : hello.o
    gcc -o hello hello.o
```

使用 `make` 命令执行的结果是怎样的？如何才能够生成最终的可执行文件 `hello` 呢？尝试使用命令来验证你的想法。

5. 伪目标

伪目标 是 `Makefile` 中不对应文件的目标。

在编写 `Makefile` 时，通常为了达成某种目标来编写一些规则。但此规则并不是要生成这些目标，而是需要执行这个规则中的命令。我们把这种目标称为**伪目标**。

示例：

我们改写 `Makefile` 文件如下:

```
all: hello  # 编译最终目标

hello : hello.o
    gcc -o hello hello.o

hello.o : hello.c
    gcc -o hello.o -c hello.c

install:  # 安装程序
    cp hello /usr/local/bin/

clean:  # 清除目标文件
    rm hello.o

distclean:  # 恢复成源码状态
    rm hello.o hello
```

在上述 Makefile 文件中，我们编写了目标为 `all`、`install`、`clean` 和 `distclean` 的规则，但我们并不是要生成这些文件，而是使用 make 命令后跟参数就可以执行相应的命令。

上述 `Makefile` 的执行规则如下：

1. `make` 或 `make all` 生成目标文件和可执行程序。
2. `make install` 将生成的目标安装在指定位置。
3. `make clean` 删除目标文件。
4. `make distclean` 函数目标文件和最终可执行程序，恢复成源码状态。

但上述程序有个问题，就是如果当前目标已经存在，如存在名为 `clean` 的文件，则 `make clean` 则不会执行规则中的语句。为此我们要使用 `.PHONY` 声明来告诉 `make` 命令，这些目标不需要找对应文件，直接执行即可，这些目标是**伪目标**。

语法:

```
.PHONY: 伪目标1 伪目标2 伪目标3
```

完整示例如下:

```
# 声明如下目标是伪目标
.PHONY: all install clean distclean

all: hello  # 编译最终目标
```

```
hello : hello.o
    gcc -o hello hello.o

hello.o : hello.c
    gcc -o hello.o -c hello.c

install:  # 安装程序
    cp hello /usr/local/bin/

clean:  # 清除目标文件
    rm hello.o

distclean:  # 恢复成源码状态
    rm hello.o hello
```

编译结果如下

```
weimingze@mzstudio:~$ ls
Makefile    hello.c
weimingze@mzstudio:~$ make
gcc -o hello.o -c hello.c
gcc -o hello hello.o
weimingze@mzstudio:~$ ls
Makefile    hello       hello.c     hello.o
weimingze@mzstudio:~$ make distclean
rm hello.o hello
weimingze@mzstudio:~$ ls
Makefile    hello.c
weimingze@mzstudio:~$ make all
gcc -o hello.o -c hello.c
gcc -o hello hello.o
weimingze@mzstudio:~$ ls
Makefile    hello       hello.c     hello.o
weimingze@mzstudio:~$ make clean
rm hello.o
weimingze@mzstudio:~$ ls
Makefile    hello       hello.c
```

用户标准目标

在使用 Makefile 时，为了统一伪目标的用法，Makefile 规定了 **用户标准目标**，这些目标如下表所示

标准目标	说明
`all`	编译整个程序。这应该是缺省的目标。
`install`	安装编译后的可执行程序到系统中。
`uninstall`	卸载安装，`install` 的反向操作。
`clean`	删除当前目录下创建目标程序时的所有中间文件。
`distclean`	删除所有中间文件和最终程序，恢复源码状态。
`check`	执行自我检查。

实验：

在上述示例中，在本地创建一个名为 `clean` 的文件，然后执行 `make clean` 规则 `clean:` 中的命令是否执行。

第二章、变量和模式

1. 变量

同其他编程语言一样，`Makefile` 支持变量的语法。使用变量可以代替一段字符串，在需要的时候可以使用引用变量的方式来将变量的值（字符串）替换到相应的位置。

使用变量可以让 `Makefile` 更易于阅读和修改。

在了解变量之前，我们先来了解 `make` 命令执行的两个阶段。第一阶段读取 Makefile（或 makefile）文件，解析其内置变量及值，构造目标和依赖等。第二阶段是根据目标执行规则。

变量的四种定义方法

```
# 创建立即变量
立即变量名 := 值（立即计算）
# 创建延时变量
延时变量名 = 值（延时计算）
# 如果没有定义变量时使用创建延时变量
延时变量名 ?= 值（延时计算）
# 在原有变量上追加值
延时变量名或立即变量名 += 值
```

说明：

1. 变量创建需要在 Makefile 内部书写，不能写在规则中。
2. 立即变量会在创建变量是直接计算值并交给变量。
3. 延时变量在创建时不会计算值，只有在使用时才会动态的计算其值（即延时计算）。
4. `?=` 是在没有此变量时，使用该值创建延时变量，如果变量已经存在则不执行赋值操作。
5. `+=` 是在已有变量的基础上追加值，如果变量是立即变量则立即计算该值，如果变量是延时变量则延时计算该值。

示例：

简化上节课的 Makefile 文件如下：

```
.PHONY: all clean

all: hello  # 编译最终目标

hello : hello.o
```

```
    gcc -o hello hello.o

hello.o : hello.c
    gcc -o hello.o -c hello.c

clean:  # 清除目标文件
    rm hello.o
```

现在我们依旧可以使用 `hello.c` 来生成 `hello` 这个可执行文件。

现在有个需求，就是将生成的可执行程序由 `hello` 改成 `myapp`，将目标文件名由 `hello.o` 改为 `temp.o`。每次名称的修改我们都要修改多处，并容易出错。如果以后还会有改动，那我们最好使用变量。

我们将 `hello` 用立即变量 `TARGET` 代替。`hello.o` 用立即变量 `OBJECT` 代替。

示例：

改写后的 `Makefile` 文件如下:

```
.PHONY: all clean

# 创建立即变量 TARGET、OBJECT 并立即取值
TARGET := hello
OBJECT := hello.o


all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECT)
    gcc -o $(TARGET) $(OBJECT)

$(OBJECT) : hello.c
    gcc -o $(OBJECT) -c hello.c

clean:  # 清除目标文件
    rm $(OBJECT)
```

其中 `TARGET := hello` 和 `OBJECT := hello.o` 是创建两个立即变量。后续使用 `$(TARGET)` 和 `$(OBJECT)` 是变量引用。

变量引用

变量引用 是取其变量的值，将其放入到引用的地方。

变量引用的语法

```
$(变量名)
# 或
${变量名}
```

说明:

- 使用 `$()` 和 `${}` 的两种语法没有实质的区别。两者可以替换。

上述使用变量的 `Makefile` 和没有使用变量的 `Makefile` 效果相同。但如果将生成的可执行文件改名为 `myapp` 则只需要修改变量 `TARGET` 的值即可，如：`TARGET := myapp`

立即变量和延迟变量的区别:

1. 立即变量会在 赋值 `:=` 或 `+=` 时直接计算变量值并改变变量，即在 make 执行的第一阶段计算值。
2. 延时变量会在 赋值 `=` 、`+=` 或 `?=` 时只赋值表达式，在变量引用时才计算变量值，即在 make 执行的第二阶段计算值。

立即变量示例

```
# 创建MYVAR1 变量
MYVAR1 := aaa
# 取 MYVAR1的值作为 MYVAR2的值
MYVAR2 := $(MYVAR1)
# 修改MYVAR1变量的值
MYVAR1 := bbb

all:
    echo "MYVAR1: $(MYVAR1)"
    echo "MYVAR2: $(MYVAR2)"
```

执行结果如下:

```
weimingze@mzstudio:~$  make
echo "MYVAR1: bbb"
MYVAR1: bbb
echo "MYVAR2: aaa"
MYVAR2: aaa
```

延时变量示例

```
# 创建MYVAR1 变量
MYVAR1 := aaa
# 取 MYVAR1的值作为 MYVAR2的值
```

```
MYVAR2 = $(MYVAR1)
# 修改MYVAR1变量的值
MYVAR1 := bbb

all:
    echo "MYVAR1: $(MYVAR1)"
    echo "MYVAR2: $(MYVAR2)"
```

执行结果如下:

```
weimingze@mzstudio:~$  make
echo "MYVAR1: bbb"
MYVAR1: bbb
echo "MYVAR2: bbb"
MYVAR2: bbb
```

上述两个示例的差别在于 `MYVAR2 := $(MYVAR1)` 和 `MYVAR2 = $(MYVAR1)`，但从结果上却有所不同。`MYVAR2 := $(MYVAR1)` 是立即算出 `MYVAR2` 的值。而 `MYVAR2 = $(MYVAR1)` 则是在 `$(MYVAR2)` 变量引用时才计算其值。

实验

尝试创建立即变量和延时变量，再尝试使用 `?=` 和 `+=` 修改变量的值，推断 Makefile 的执行过程。

2. 自动变量

在 Makefile 的每个规则的命令中都可以使用 **自动变量** 来代替目标文件、依赖文件列表、单个的依赖文件等特殊的变量。这些变量称之为**自动变量**。

自动变量可以 Makefile 更加简洁，通用。

Makefile 的**自动变量**如下表所示：

自动变量	说明
$@	目标文件名。
$<	第一个依赖文件名。
$^	所有依赖文件名（去掉重复项）。
$+	所有依赖文件名，保持原依赖列表（包含重复项）。
$?	比目标文件新的所有的依赖文件名。
$*	通配符匹配的部分。
$%	档案成员名 。

示例：

现在有如下的 C 语言的源代码程序，我们需要将其编译成一个应用程序 `myapp`。

```
myapp/
├── file1.c
├── file1.h
├── file2.c
├── file2.h
└── main.c
```

点击下载上述文件压缩包 myapp.zip

文件内容如下：

文件: file1.h

```h
// filename: file1.h
#ifndef __FILE1_H
#define __FILE1_H

void myfunc1(void);

#endif // __FILE1_H
```

文件: file1.c

```c
// filename: file1.c
#include <stdio.h>
#include "file1.h"

void myfunc1(void) {
```

```
    printf("库函数 myfunc1 被调用\n");
}
```

文件: file2.h

```
// filename: file2.h
#ifndef __FILE2_H
#define __FILE2_H

void myfunc2(void);

#endif // __FILE2_H
```

文件: file2.c

```
// filename: file2.c
#include <stdio.h>
#include "file2.h"

void myfunc2(void) {
    printf("库函数 myfunc2 被调用\n");
}
```

文件: main.c

```
// filename: main.c
#include "file1.h"
#include "file2.h"

int main(int argc, char * argv[]) {
    // 调用合作方的库函数
    myfunc1();
    myfunc2();

    return 0;
}
```

上述文件编译的过程如下:

```
+---------+     +---------+
| file1.h | --> | file1.o | -->.
+---------+     +---------+    |
                               |
+---------+     +---------+    |         +-------+
| file2.c | --> | file2.o | ---+------> | myapp |
+---------+     +---------+    |         +-------+
                               |
+---------+     +---------+    |
```

```
|  main.c | --> |  main.o | -->`
+---------+     +---------+
```

编写 Makefile 如下:

```
.PHONY: all clean

SOURCES := file1.c file2.c main.c
HEADERS := file1.h file2.h
OBJECTS := file1.o file2.o main.o
TARGET := myapp

all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

main.o : main.c $(HEADERS)
    gcc -o $@ -c $<

file1.o : file1.c $(HEADERS)
    gcc -o $@ -c $<

file2.o : file2.c $(HEADERS)
    gcc -o $@ -c $<

clean:  # 清除目标文件
    rm $(OBJECTS)
```

编译和运行结果如下:

```
weimingze@mzstudio:~/myapp$ make
gcc -o file1.o -c file1.c
gcc -o file2.o -c file2.c
gcc -o main.o -c main.c
gcc -o myapp file1.o file2.o main.o
weimingze@mzstudio:~/myapp$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

在上述 `Makefile` 的规则中，我们使用 `$@` 代替目标，`$<` 代替依赖的第一个文件，`$^` 代替依赖列表中的所有文件。

实验:

使用上述示例程序中的五个文件，修改 Makefile 如下，运行此 Makefile，查看打印结果是什么？为什么？

```
all: main.c file1.c file2.c file1.h file2.h main.c
    echo $@
    echo $<
    echo $^
    echo $+
```

3. 隐含规则

隐含规则 是 `make` 程序内置的规则，他会在 make 命令的第一阶段加载，并可以根据隐含规则来生成对应的目标文件。

默认的**隐含规则**会在没有 `.o` 目标文件时，会尝试使用 `cc` 编译器将同名的 `.c` 文件生成同名的 `.o` 文件。如果不存在同名的 `.c` 文件也会尝试使用 `yacc` 词法分析工具将同名的 `.y` 文件生成 同名的 `.c` 文件。如果也没有 `.y` 文件，则会尝试使用 `g++` 编译器将同名的 `.cc` 或 `.cpp` 编译成对应的 `.o` 文件。这样的规则称为**隐含规则**。类似的**隐含规则**还有很多。

在上节课的示例中我们定义了**显式规则**如下：

```
main.o : main.c $(HEADERS)
    gcc -o $@ -c $<

file1.o : file1.c $(HEADERS)
    gcc -o $@ -c $<

file2.o : file2.c $(HEADERS)
    gcc -o $@ -c $<
```

在执行 `make` 时，是使用 `gcc` 编译器 对 `mail.c`、`file1.c`、`file2.c` 编译生成对应的 `.o` 文件。编译命令如下:

```
gcc -o file1.o -c file1.c
gcc -o file2.o -c file2.c
gcc -o main.o -c main.c
```

我们删除 Makefile 中的上述显式规则，你会发现在执行 `make` 命令时依旧可以编译成功。

Makefile 文件内容如下:

```
.PHONY: all clean

SOURCES := file1.c file2.c main.c
HEADERS := file1.h file2.h
OBJECTS := file1.o file2.o main.o
```

```
TARGET := myapp

all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

clean:  # 清除目标文件
    rm $(OBJECTS)
```

运行 `make` 结果如下:

```
weimingze@mzstudio:~/myapp$ make
cc    -c -o file1.o file1.c
cc    -c -o file2.o file2.c
cc    -c -o main.o main.c
gcc -o myapp file1.o file2.o main.o
weimingze@mzstudio:~/myapp$ ./myapp
库函数 myfunc1 被调用
库函数 myfunc2 被调用
```

上述程序中，实际是使用的是隐含规则 来生成 `mail.o`、`file1.o` 和 `file2.o` 这三个文件，他们使用的命令是 `$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $^`。

其中 `CC`、`CPPFLAGS` 和 `CFLAGS` 都是隐含规则的变量，Makefile 允许我们修改这些变量来更换编译器和编译选项。

隐含规则的变量

变量	说明
AR	档案管理程序，缺省为：ar
AS	汇编编译程序，缺省为：as
CC	C语言编译程序，缺省为：cc
CXX	C++编译程序，缺省为：g++
CO	从RCS文件中解压缩抽取文件程序，缺省为：co
CPP	带有标准输出的C语言预处理程序，缺省为：$(CC) -E
FC	Fortran 以及 Ratfor 语言的编译和预处理程序，缺省为：f77
GET	从SCCS文件中解压缩抽取文件程序，缺省为：get
LEX	将 Lex 语言转变为 C 或 Ratfor程序的程序，缺省为：lex
PC	Pascal 程序编译程序，缺省为：pc
YACC	将 Yacc语言转变为 C程序的程序，缺省为：yacc
MAKEINFO	将Texinfo 源文件转换为信息文件的程序，缺省为：makeinfo
TEX	从TeX源产生TeX DVI文件的程序，缺省为：tex
TEXI2DVI	从Texinfo源产生TeX DVI 文件的程序，缺省为：texi2dvi
WEAVE	将Web翻译成TeX的程序，缺省为：weave
CWEAVE	将CWeb翻译成TeX的程序，缺省为：cweave
TANGLE	将Web翻译成 Pascal的程序，缺省为：tangle
RM	删除文件的命令，缺省为：rm -f
	以下为 编译时的额外标志，如果不重新定义，缺省为空
ARFLAGS	用于档案管理程序的标志，缺省为：rv
ASFLAGS	用于汇编编译器的额外标志 (当具体调用‘.s'或‘.S'文件时)。
CFLAGS	用于C编译器的额外标志。
CXXFLAGS	用于C++编译器的额外标志。
COFLAGS	用于RCS co程序的额外标志。
CPPFLAGS	用于C预处理以及使用它的程序的额外标志 (C和 Fortran 编译器)。
FFLAGS	用于Fortran编译器的额外标志。

GFLAGS	用于SCCS get程序的额外标志。
LDFLAGS	用于调用linker('ld')的编译器的额外标志。
LFLAGS	用于Lex的额外标志。
PFLAGS	用于Pascal编译器的额外标志。
RFLAGS	用于处理Ratfor程序的Fortran编译器的额外标志。
YFLAGS	用于Yacc的额外标志。

在上述示例中，我们将编译器改为 `gcc` ，将编译选项 添加一个 `-g -Wall` 参数，改写后代码如下:

改写后的 Makefile 文件如下:

```
.PHONY: all clean

SOURCES := file1.c file2.c main.c
HEADERS := file1.h file2.h
OBJECTS := file1.o file2.o main.o
TARGET := myapp

# 修改隐含规则的变量，更改编译命令
CC = gcc
CFLAGS = -g -Wall

all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^


clean:  # 清除目标文件
    rm $(OBJECTS)
```

使用 `make` 编译结果如下:

```
weimingze@mzstudio:~/myapp$ make
gcc -g -Wall   -c -o file1.o file1.c
gcc -g -Wall   -c -o file2.o file2.c
gcc -g -Wall   -c -o main.o main.c
gcc -o myapp file1.o file2.o main.o
```

从上述 `make` 命令的执行结果可以看出。我们通过修改 `CC` 等变量的值将 C 语言的编译器改成了 `gcc`，而不再使用默认的编译器 `cc`。

可用的隐含规则

1. `.c` 文件生成同名 `.o` 文件。
2. `.cc`、`.C` 或 `.cpp` 文件生成同名 `.o` 文件。
3. `.s` 文件生成同名 `.o` 文件。
4. `.f` 文件生成同名 `.o` 文件。
5. `.o` 文件生成同名无后缀文件。
6. `.i` 文件生成同名 `.o` 文件。
7. `.y` 文件生成同名 `.c` 文件。
8. `.l` 文件生成同名 `.c` 文件。
9. `.o` 文件生成同名 `lib`开头，`.a`结尾的静态库文件。
10. `.dvi` 文件生成同名 `.ps` 文件。

实验：

尝试使用 隐含规则编译多个 C++ 文件。将多个 C++ 文件编译成一个可执行文件。如果你没有 C++ 文件，你可以尝试将 `file1.c`、`file2.c` 和 `mail.c` 的文件后缀由 `.c` 改为 `.cpp` 然后尝试运行 `make` 编译。

4. 通配符

通配符 是用于匹配任意非空字符串的符号。

在 Makefile 中有两种用于匹配任意字符串的通配符 `*` 和 `%`。

星号（*）通配符

星号通配符通常用于在文件系统中搜索匹配的文件名。

星号通配符只能用于两种场景中：

1. 规则中的命令部分
2. 用于 `wildcard` 函数中(后面才讲)。

在规则的命令部分中使用 星号通配符示例：

```
clean:
    rm *.o
```

百分号（%）通配符

百分号通配符可用于四种场景中：

1. 静态模式规则
2. 模式规则
3. 字符串替换
4. vpath 指令

5. 模式规则

模式规则是在规则中使用百分号（%）通配符 来代替显示规则的一种方式，他可以制作显式规则的模版。让多个规则总结为一个规则的方法。

我们之前定义多个显式规则，使用显式规则可以自定义每一个规则执行的命令，当然文件比较多时，规则也就相应的增多，代码量会变大。

如下：

```
main.o : main.c $(HEADERS)
    gcc -o $@ -c $<

file1.o : file1.c $(HEADERS)
    gcc -o $@ -c $<

file2.o : file2.c $(HEADERS)
    gcc -o $@ -c $<
```

如果文件比较多的时候，我们可以使用隐含规则，但隐含规则只能匹配默认的几种规则，且不能够自定义命令的格式。此时使用**模式规则**可以解决上述问题。

模式规则的语法：

```
目标文件模式 ： 依赖文件模式
<制表符> 命令1
<制表符> 命令2
...
```

说明：

- 目标文件模式 和 依赖文件模式 需要使用百分号通配符来表示文件的模式和匹配样式。

例如，上述模式可以改写成

```
%.o : %.c $(HEADERS)
    gcc -o $@ -c $<
    echo "compiling $@, % is $* ..."
```

此模式规则中 `%.o` 表示以任何 `.o` 结尾的目标文件。`%.c` 是和目标同名且以 `.c` 结尾的文件。此时的 `%` 匹配非空的文件名，且文件名一定相同。即 `%.o` 匹配 `file1.o` 时，`%.c` 一定匹配 `file1.c`。

在模式匹配中，使用自动变量 `$*` 可以获取通配符 `%` 匹配的字符串。

改写后的 Makefile 如下:

```
.PHONY: all clean

SOURCES := file1.c file2.c main.c
HEADERS := file1.h file2.h
OBJECTS := file1.o file2.o main.o
TARGET := myapp

# 修改隐含规则的变量，更改编译命令
CC = gcc
CFLAGS = -g -Wall

all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

# 自定义模式规则
%.o : %.c $(HEADERS)
    gcc -o $@ -c $<
    echo "compiling $@, % is $* ..."

clean:  # 清除目标文件
    rm $(OBJECTS)
```

make 的执行结果如下:

```
weimz@anonymous myapp % make
gcc -o file1.o -c file1.c
echo "compiling file1.o, % is file1 ..."
compiling file1.o, % is file1 ...
gcc -o file2.o -c file2.c
echo "compiling file2.o, % is file2 ..."
compiling file2.o, % is file2 ...
gcc -o main.o -c main.c
echo "compiling main.o, % is main ..."
compiling main.o, % is main ...
gcc -o myapp file1.o file2.o main.o
```

从上述执行结果可以看出，使用模式规则可以代替隐含规则并能够自定义任何通用规则，使用 `$*` 可以获取每次匹配的字符串。

实验：

尝试在自己的 Makefile 中使用模式规则。

6. 静态模式规则

静态模式规则 是一种应用于指定目标的模式规则。

模式规则是通用的规则，静态模式规则你可以认为他是为某些文件制定的专用的模式规则。

语法：

```
目标文件列表 ： 目标文件模式 ： 依赖文件模式
<制表符> 命令1
<制表符> 命令2
...
```

示例:

使用静态模式规则，在编写 `main.o` 时使用一种模式规则。在编译 `file1.o` 和 `file2.o` 时使用另外一种模式规则。

改写后的 Makefile，如下:

```
.PHONY: all clean

SOURCES := file1.c file2.c main.c
HEADERS := file1.h file2.h
OBJECTS := file1.o file2.o main.o
TARGET := myapp

# 修改隐含规则的变量，更改编译命令
CC = gcc
CFLAGS = -g -Wall

all: $(TARGET)  # 编译最终目标

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

# 自定义静态模式规则1
main.o : %.o : %.c
    gcc -o $@ -c $<
    echo "pattern rule 1..."
```

```
# 自定义静态模式规则2
file1.o file2.o : %.o : %.c $(HEADERS)
    gcc -c $< -I.
    echo "pattern rule 2..."

clean:  # 清除目标文件
    rm $(OBJECTS)
```

多目标规则

上述规则 `file1.o file2.o : %.o : %.c $(HEADERS)` 是多目标规则，每个目标会单独执行相应命令，自动变量 `$@` 置代表当前正在处理的目标。

执行 make 命令后则结果如下：

```
weimz@anonymous myapp % make
gcc -c file1.c -I.
echo "pattern rule 2..."
pattern rule 2...
gcc -c file2.c -I.
echo "pattern rule 2..."
pattern rule 2...
gcc -o main.o -c main.c
echo "pattern rule 1..."
pattern rule 1...
gcc -o myapp file1.o file2.o main.o
```

运行结果可以看出，在编译 `file1.o` 和 `file2.o` 时使用的是规则 `file1.o file2.o : %.o : %.c $(HEADERS)`，而编译 `main.o` 时使用的是规则 `main.o : %.o : %.c`。

实验：

尝试使用静态模式规则。

第三章、函数

1. 函数

函数是指实现了某些功能的指令或命令，通常一个函数有一个名称，这个名称代表这个指令或命令的序列。

Makefile 允许编写者自己定义函数和调用函数。Makefile 中已经定义了很多预置的函数供使用。

Makefile 函数调用的语法:

```
$(函数名  实际调用参数1,实际调用参数2,...)
# 或者
${函数名  实际调用参数1,实际调用参数2,...}
```

说明:

1. 语法中 `${}` 和 `$()` 的调用方式没有区别。在嵌套调用时可以使用不同的括号来区分调用的语法边界。
2. **函数名**和**实际调用参数**间要添加至少一个空格用于区分语法边界。
3. 如果**实际调用参数**有两个或两个以上，则需要使用英文的逗号（`,`）来进行分隔。
4. 函数的**实际调用参数**由函数定义者规定，给出多余的实际调用参数会被忽略。

1.1 wildcard 函数

wildcard 函数 用于使用 星号（`*`）通配符搜索文件名，并返回找到的文件路径的列表。

调用格式:

```
$(wildcard pattern…)
```

说明:

- pattern... 通常是含有星号（`*`）通配符的表达式。如:
  - `$(wildcard *.c)` 是搜索本地所有 `.c` 为后缀的文件并返回路径。
  - `$(wildcard *.c *.h)` 是搜索本地所有 `.c` 和 `.h` 为后缀的文件并返回路径。

示例：

改写之前的示例。让程序能搜索到所有的 `.c` 结尾的文件。让其作为变量 `SOURCES` 的值，改写后的 Makefile 如下：

```
.PHONY: all clean

# 调用 wildcard 函数动态获取所有的 .c 文件路径
SOURCES := $(wildcard *.c)
HEADERS := ${wildcard *.h}  # 获取所有的 .h 文件路径
OBJECTS := file1.o file2.o main.o
TARGET := myapp

CC = gcc
CFLAGS = -g -Wall

all: $(TARGET)

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

%.o : %.c $(HEADERS)
    gcc -o $@ -c $<

clean:
    rm $(OBJECTS)
```

使用 make 编译结果如下:

```
weimingze@mzstudio:~/myapp$ make
gcc -o file1.o -c file1.c
gcc -o file2.o -c file2.c
gcc -o main.o -c main.c
gcc -o myapp file1.o file2.o main.o
```

从运行结果可见使用 `SOURCES := $(wildcard *.c)` 代替前的 `SOURCES := file1.c file2.c main.c` 的效果是一样的。但使用函数时，我们可以动态的增加 `.c` 文件而不需要重新编写 Makefile 文件。

实验：

尝试使用 `wildcard` 函数修改自己的 Makefile 文件。

1.2 patsubst 函数

patsubst 函数是一个字符串替换相关的函数。它可以根据由百分号（`%`）通配符组成模式来对目标字符串进行替换。

调用格式:

```
$(patsubst pattern,replacement,text)
```

说明:

- pattern 通常是含有百分号（%）通配符的字符串。
- replacement 是要替换的文字，内部可以使用 百分号（%）通配符来代替匹配的内容。
- text 是要匹配的字符串。
- 此函数返回替换后的字符串

如:

```
OBJECTS := $(patsubst %.c,%.o,file1.c file2.c main.c)
```

替换后的 `OBJECTS` 变量的值为 `file1.o file2.o main.o`

示例:

使用 patsubst 函数，根据变量 `SOURCES` 中的 `.c` 文件组成的列表动态生成 `.o` 文件的列表，并存储到变量 `OBJECTS` 中。

改写后的 Makefile 如下

```
.PHONY: all clean

SOURCES := $(wildcard *.c)
HEADERS := ${wildcard *.h}
# 使用 SOURCE 中的值生成目标文件
OBJECTS := $(patsubst %.c,%.o,${SOURCES})
TARGET := myapp

CC = gcc
CFLAGS = -g -Wall

all: $(TARGET)

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

%.o : %.c $(HEADERS)
    gcc -o $@ -c $<

clean:
    rm $(OBJECTS)
```

使用 make 编译结果如下:

```
weimingze@mzstudio:~/myapp$ make
gcc -o file1.o -c file1.c
gcc -o file2.o -c file2.c
gcc -o main.o -c main.c
gcc -o myapp file1.o file2.o main.o
```

从结果上看 `make` 执行编译的结果和之前没有任何区别，但此时的 Makefile 更加通用。在项目文件夹中新增 `.c` 和 `.h` 文件时，几乎可以不用改写 Makefile 文件。

字符替换的简洁写法:

上述 Makefile 中，`$(patsubst %.c,%.o,${SOURCES})` 这个字符串替换函数有一种简洁的写法 `$(SOURCES:%.c=%.o)`，即:

```
$(text:pattern=replacement)
```

等同于

```
$(patsubst pattern,replacement,${text})
```

> 参数 `pattern` 和 `replacement` 要使用通配符则只能使用百分号通配符（`%`）。

实验:

将上述示例中的 `OBJECTS := $(patsubst %.c,%.o,${SOURCES})` 改写为 `OBJECTS := $(SOURCES:%.c=%.o)`，然后运行 `make` 命令查看运行结果有何不同?

2. 预置函数

以下将以表格的形式列出 Makefile 中 的一些内置函数。

函数	说明	示例
`$(subst from,to,text)`	字符串替换	`$(subst BC,bc,aBCd)` 返回 `abcd`
`$(patsubst pattern,replacement,text)`	模式字符串替换	`$(patsubst %.c, %.o,a.c b.c)` 返回 `a.o b.o`
`$(strip string)`	去掉字符串两侧的空白字符	`$(strip a b c )` 返回 `a b c`
`$(findstring find,text)`	查找字符串，找到返回 find 找不到返回空内容	`$(findstring cde,abcdef)` 返回 `cde`
`$(filter pattern…,text)`	筛选符合模式的字符串	`echo $(filter %o,hello world)` 返回 `hello`
`$(filter-out pattern…,text)`	筛选不符合模式的字符串	`$(filter-out %o,hello world)` 返回 `world`
`$(word n,text)`	返回 text 中的第几个单词	`$(word 2,hello world)` 返回 `world`
`$(shell command)`	执行Shell 命令，返回 命令的标准输出	`$(shell pwd` 返回当前的工作路径
`$(foreach var,list,text)`	对 list 中的每个单词进行迭代，然后替换成 test。	`$(foreach wrd,who are you,$(wrd)!)` 返回 `who! are! you!`

以下预置函数将只给出函数和说明。

函数	说明
`$(sort list)`	按字典序对列表中的单词进行排序，并移除重复项。
`$(words text)`	统计文本中的单词数量。
`$(wordlist s,e,text)`	返回文本中从第 s 个到第 e 个单词的列表。
`$(firstword names…)`	提取 names 中的第一个单词。
`$(lastword names…)`	提取 names 中的最后一个单词。
`$(dir names…)`	提取每个文件名的目录部分。
`$(notdir names…)`	提取每个文件名的非目录部分。
`$(suffix names…)`	提取每个文件名的后缀（最后一个 '.' 及其后的字符）。
`$(basename names…)`	提取每个文件名的基本名称（不含后缀）。
`$(addsuffix suffix,names…)`	将后缀 suffix 追加到 names 中的每个单词后。
`$(addprefix prefix,names…)`	将前缀 prefix 添加到 names 中的每个单词前。
`$(join list1,list2)`	连接两个并行列表中的单词。
`$(wildcard pattern…)`	查找与 shell 文件名模式（非 '%' 模式）匹配的文件名。
`$(realpath names…)`	对于 names 中的每个文件名，扩展为不含任何 .、.. 或符号链接的绝对路径名。
`$(abspath names…)`	对于 names 中的每个文件名，扩展为不含任何 . 或 .. 成分的绝对路径名，但保留符号链接。

`$(error text…)`	当此函数被求值时，make 会生成一条包含消息 text 的致命错误。
`$(warning text…)`	当此函数被求值时，make 会生成一条包含消息 text 的警告。
`$(origin variable)`	返回一个字符串，描述 make 变量 variable 是如何定义的。
`$(flavor variable)`	返回一个字符串，描述 make 变量 variable 的类型。
`$(let var [var ...],words,text)`	将 vars 绑定到 words 中的单词，然后对 text 进行求值。
`$(if condition,then-part[,else-part])`	对条件 condition 进行求值；如果非空，则替换为 then-part 的扩展结果，否则替换为 else-part 的扩展结果。
`$(or condition1[,condition2[,condition3…]])`	依次对每个条件 conditionN 进行求值；替换为第一个非空的扩展结果。如果所有扩展结果都为空，则替换为空字符串。
`$(and condition1[,condition2[,condition3…]])`	依次对每个条件 conditionN 进行求值；如果任何扩展结果为空字符串，则替换为空字符串。如果所有扩展结果均为非空字符串，则替换为最后一个条件的扩展结果。
`$(intcmp lhs,rhs[,lt-part[,eq-part[,gt-part]]])`	对 lhs 和 rhs 进行数值比较；根据左侧小于、等于或大于右侧的情况，分别替换为 lt-part、eq-part 或 gt-part 的扩展结果。
`$(call var,param,…)`	对变量 var 进行求值，并将任何对 $(1)、$(2) 等的引用替换为第一个、第二个等 param 值。

`$(eval text)`	对 text 进行求值，然后将结果作为 makefile 命令读取。扩展为空字符串。
`$(file op filename,text)`	扩展参数，然后以模式 op 打开文件 filename 并将 text 写入该文件。
`$(value var)`	求值为变量 var 的内容，且不对其进行任何扩展。

第四章、高级用法

1. 命令回显

在 执行 make 命令时，Makefile 内规则的命令总是会显示在控制台终端中，如:

```
all:
    echo "Hello Laowei!"
    echo "finished!"
```

执行 Make 的结果如下：

```
weimingze@mzstudio:~$ make
echo "Hello Laowei!"
Hello Laowei!
echo "finished!"
finished!
```

其中 `echo "Hello Laowei!"` 和 `echo "finished!"` 都是命令的回显。如果不想在控制台终端中显示回显有两种方法：

1. 在需要在回显的 命令前加让一个 `@` 字符。
2. 如果所有命令都不显示回显，可以在 `make` 命令后加 `-s` 或 `--silent` 选项。

修改后的 Makefile 如下:

```
all:
    @echo "Hello Laowei!"
    echo "finished!"
```

执行 Make 的结果如下：

```
weimingze@mzstudio:~$ make
Hello Laowei!
echo "finished!"
finished!
weimingze@mzstudio:~$ make -s
Hello Laowei!
finished!
```

可见 `@echo "Hello Laowei!"` 命令则不会显示命令回显，使用 `make -s` 则取消所有命令回显。

2. vpath 指令和 VPATH 变量

本节课讲解两个内容：**vpath 指令**和 **VPATH 变量**。

vpath 指令是 Makefile 中的目录搜索机制，用于告诉 make 在哪些目录中查找源文件或依赖文件。 **VPATH 变量** 和 **vpath 指令**的作用基本相同，但 VPATH 只能全局统一设置，不能根据匹配模式进行精准控制。

为了说明上述内容，现在我们修改原来的程序文件的存放位置。我们分类将文件放入 `include`、`src` 和 `task` 三个文件夹中。

文件结构如下:

```
myapp2/
├── include
│   ├── file1.h
│   └── file2.h
├── src
│   └── main.c
└── task
│   ├── file1.c
│   └── file2.c
└── Makefile
```

> 点击此处可以下载 上述文件夹的压缩包 myapp2.zip

现在我们在 Makefile 文件中，内容如下

```
.PHONY: all clean

SOURCES := main.c file1.c file2.c
HEADERS := file1.h file2.h
OBJECTS := main.o file1.o file2.o
TARGET := myapp2

all: $(TARGET)

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

%.o : %.c $(HEADERS)
    gcc -o $@ -c $< -I include

clean:
    rm $(OBJECTS)
```

使用 `make` 命令编译，发送错误如下:

```
weimingze@mzstudio:~/myapp2$ make
make: *** 没有规则可制作目标“main.o”，由“myapp2” 需求。 停止。
```

上述 `make` 执行过程提示 `main.o` 没有找到。其原因是没有找到 `main.c` 来生成 `main.o` 因为 `main.c` 在文件夹 `src` 中。 `make` 命令只搜索当前文件夹，并不会搜索其更深层次的文件夹。要想让 make 能够搜索到其他文件夹有两种方法：

1. 使用 **vpath 指令**。
2. 使用 **VPATH 变量**。

1、vpath 指令

语法：

```
vpath 匹配模式 路径1:路径2:...
```

说明：

1. 匹配模式可以使用百分号（`%`）通配符进行模式匹配。
2. 路径可以有一个或多个，中间用冒号（`:`）分隔。
3. 当有多个路径时，按先后顺序进行搜索。
4. 如果没有给出任何路径，则取消当前匹配模式的 路径设置。

修改上述 Makefile 我们添加 vpath 指令如下:

```
# 指定 .c 文件在 src 和 task 文件夹中搜索
vpath %.c src:task
# 指定 .h 文件在 include 文件夹中搜索
vpath %.h include
```

修改后完整的 `Makefile` 如下:

```
.PHONY: all clean

# 指定 .c 文件在 src 和 task 文件夹中搜索
vpath %.c src:task
# 指定 .h 文件在 include 文件夹中搜索
vpath %.h include

SOURCES := main.c file1.c file2.c
HEADERS := file1.h file2.h
```

```
OBJECTS := main.o file1.o file2.o
TARGET := myapp2

all: $(TARGET)

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

%.o : %.c $(HEADERS)
    gcc -o $@ -c $< -I include

clean:
    rm $(OBJECTS)
```

> `gcc` 的 `-I <路径>` 是指定头文件的位置

执行 Make的结果如下:

```
weimingze@mzstudio:~/myapp2$ make
gcc -o main.o -c src/main.c -I include
gcc -o file1.o -c task/file1.c -I include
gcc -o file2.o -c task/file2.c -I include
gcc -o myapp2 main.o file1.o file2.o
weimingze@mzstudio:~/myapp2$ ls
Makefile  file2.o  main.o  src
file1.o  include  myapp2  task
```

由执行结果可以看出 `gcc -o main.o -c src/main.c -I include` 命令中 `main.c` 文件自动添加了路径 `src/`。

2、VPATH 变量

VPATH 变量是指定全局搜索路径，与 **vpath 指令**不同，他不能针对模式匹配的方式进行搜索路径的设置。

用法：

```
VPATH = 路径1:路径2:...
```

如，我们将上述示例中的 `src`、`task` 和 `task` 设置为VPATH的值。如下:

```
# 指定`src`、`task` 和 `task` 为全局搜索路径
VPATH = src:task:include
```

修改后的完整 Makefile 文件 如下:

```
.PHONY: all clean

# 指定`src`、`task` 和 `task` 为全局搜索路径
VPATH = src:task:include

SOURCES := main.c file1.c file2.c
HEADERS := file1.h file2.h
OBJECTS := main.o file1.o file2.o
TARGET := myapp2

all: $(TARGET)

$(TARGET) : $(OBJECTS)
    gcc -o $@ $^

%.o : %.c $(HEADERS)
    gcc -o $@ -c $< -I include

clean:
    rm $(OBJECTS)
```

执行 Make的结果如下:

```
weimingze@mzstudio:~/myapp2$ make
gcc -o main.o -c src/main.c -I include
gcc -o file1.o -c task/file1.c -I include
gcc -o file2.o -c task/file2.c -I include
gcc -o myapp2 main.o file1.o file2.o
weimingze@mzstudio:~/myapp2$ ls
Makefile  file2.o  main.o  src
file1.o  include  myapp2  task
```

可见使用 VPATH 和 vpath 实现了同样的效果。

实验:

分别使用 **vpath 指令**和 **VPATH 变量**来实现搜索路径的设置。

3. 条件判断

在 Makefile 中可以使用条件判断的方式来执行某个命令或某些变量创建或变量赋值等操作。

Makefile 条件判断的语法

条件判断有三种语法:

1、只有一个执行部分的语法

```
条件指示
# ... 条件指示为真时（成立时）执行的部分
endif
```

2、有两个执行部分二选一的语法

```
条件指示
# ... 条件指示为真时（成立时）执行的部分
else
# ... 条件指示为假时（不成立时）执行的部分
endif
```

3、有三个执行部分三选一的语法

```
条件指示1
# ... 条件指示1为真时（成立时）执行的部分
else 条件指示2
# ... 条件指示2为真时（成立时）执行的部分
else
# ... 条件指示1和条件指示2都为假时执行的部分
endif
```

条件指示时判断时，根据真（成立）和假（不成立）的值决定下面的部分是否执行。

条件指示也有三种:

1. 根据两个值是否相等进行条件判断
2. 根据是否已经定义某个变量进行条件判断
3. 根据是否没有定义某个变量进行条件判断

1、 根据两个值是否相等进行判断的语法：

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

其中 arg1 和 arg2 都是字符串值，两个字符串的值相等返回**真**，否则返回**假**。

2、 根据是否已经定义某个变量进行判断的语法：

```
ifdef 变量名
```

如果变量已经定义则返回**真**，否则返回**假**。

3、 根据是否没有定义某个变量进行判断的语法：

```
ifndef 变量名
```

如果变量没有定义则返回**真**，否则返回**假**。

示例：

1、根据 OS 变量的值来决定如何执行相应的命令

```
# 修改成当前的操作系统
OS :=Windows
MEMORY :=

all:
ifeq ($(OS),Windows)
    @echo "OS is Windows"
else ifeq ($(OS), MacOS)
    @echo "OS is MacOS"
else
    @echo "OS is other"
endif
ifeq ($(MEMORY),)  #判断变量的值为空
    @echo "MEMORY is not set"
endif
```

2、根据是否定义变量执行相应的命令

```
A := aaaa

all:
ifdef A
    @echo "A is exist, value is $(A)"
else
    @echo "A is not exist"
endif
ifndef B
    @echo "B is not exist"
else
```

```
    @echo "B is exist, value is $(B)"
endif
```

4. 修改默认 Shell

执行 Makefile 时，默认执行命令的 Shell 是 `/bin/sh`，你可以通过修改变量 `SHELL` 来重新定义默认的 Shell。

示例:

```
# 修改默认的 Shell。
SHELL=/bin/bash

all:
    echo "Hello  from bash!"
```

5. 使用 Shell 变量

在 Makefile 使用 `$` 符号可以使用 Makefile 中的变量，使用 `$$` 可以使用 Shell 中的环境变量。

语法:

```
$${环境变量名}
# 或
$$环境变量名
```

示例:

```
MYVAR=我是 Makefile 的变量

all:
    @echo $(MYVAR)
    @echo $${HOME}
    @echo $$HOME
```

make 执行的结果

```
weimingze@mzstudio:~$ make
我是 Makefile 的变量
/home/weimingze
/home/weimingze
```

可见在我的系统中，`HOME` 环境变量对应的值是 `/home/weimingze`。

6. 命令的执行

在 Makefile 的规则中，每一行命令会在一个新的 Shell 中单独执行，这样不会相互影响。

如果你需要在一个 Shell 中执行多个命令。你需要将这些命名以分号或 `&&` 或 `||` 链接在一个命令行内。

示例:

```
all:
    cd ..
    # 上面的 cd .. 不会影响到下面 echo 命令的执行。
    # 因为 echo 在一个新的 Shell 内执行.
    echo `pwd`

    # 如下的 cd .. 命令会改变 echo  的运行路径，因为他们在一行内。
    cd ..;echo `pwd`

    # cd .. 会影响 `echo` 命令的运行路径。
    cd .. && echo `pwd`
```

make 执行的结果为

```
weimingze@mzstudio:~$ make
cd ..
# 上面的 cd .. 不会影响到下面 echo 命令的执行。
# 因为 echo 在一个新的 Shell 内执行.
echo `pwd`
/home/weimingze
# 如下的 cd .. 命令会改变 echo  的运行路径，因为他们在一行内。
cd ..;echo `pwd`
/home
# Same as above
cd .. && echo `pwd`
/home
```

附录

Makefile 实战

以下分享一个我编写的一个 Makefile。

此 Makefile 的功能是将对应的 `.c` 或 `.cpp` 的源代码编译成对应的可执行程序。如 `aaa.c` 会编译成 `aaa`；`bbb.cpp` 会编译成 `bbb`。这个 Makefile 可以帮助编写 C 和 C++ 程序的朋友快速编译自己的程序。

此 Makefile 可以用于 MacOS 和 Linux 系统中。

使用方法:

命令	说明
`make` 或 `make all`	编译当前文件夹下所有的 `.c` 和 `.cpp` 文件，并调用深层次文件夹内的 Makefile 进行编译
`make clean`	删除当前文件夹下所有的 `.c` 和 `.cpp` 编译后的文件，并调用深层次文件夹内的 Makefile 进行清理

点击当前位置下载此Makefile

```
#************************************
# 魏明择的官方网站
# https://weimingze.com
# 创建人   :  魏明择
# 创建日期:  2016年11月1日
# 魏明择版权所有，保留所有权利。
#************************************/

CC          := gcc
CFLAG       := -Wall -g -lpthread
CXX         := g++
CXXFLAG     := -Wall -g
CXXFLAG     += -std=c++11     # for C++ 11 支持
LIBS        := -lpthread

C_SRCS      := $(wildcard *.c)     #搜索所有的.c文件
CPP_SRCS    := $(wildcard *.cpp)   #搜索所有的.cpp文件
CTARGETS    := ${C_SRCS:%.c=%}      #把所有的.c文件转换成文件名的目标文件
CPPTARGETS  := ${CPP_SRCS:%.cpp=%} #把所有的.cpp文件转换成文件名的目标文件
```

```
MAKEFILS    := $(wildcard */[mM]akefile) #搜索下一层目录下所有的Makefile文件
SUBDIRS     := $(dir $(MAKEFILS))        #提取出所有的目录

#以下删除XCode编译器留下的.dSYM文件夹
XCODE_DSYM  := ${C_SRCS:%.c=%.dSYM}
XCODE_DSYM  += ${CPP_SRCS:%.cpp=%.dSYM}

all : $(CTARGETS) $(CPPTARGETS)
    @for MYDIR in $(SUBDIRS) ; do make -C $${MYDIR} $@ ; done

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

% : %.cpp
    $(CXX) -o $@ $< $(CXXFLAG) $(LIBS)

clean:
    @for MYDIR in $(SUBDIRS) ; do make -C $${MYDIR} $@ ; done
    rm -f $(CTARGETS) $(CPPTARGETS)
    rm -rf $(XCODE_DSYM)
```

缺点:

此 Makefile 只能对单个的源代码程序编译生成单独的应用程序。不能将多个 `.c` 和 `.cpp` 合并成一个 应用程序。

练习:

编写一个 Makefile，可以将此文件夹内的所有 `.c` 和 `.cpp` 文件编译并链接称为一个应用程序，应用程序名可以叫做 `myapp` 或者使用当前文件夹的名称作为应用程序名。