swpedia_keyword: [Makefile]

author: Sam [desen.li@wisewavetech.com]

Makefile Tutorial

1. Makefile 概述

Makefile 是管理项目自动化构建过程的重要工具，尤其适用于中大型、有多文件的工程项目。它描述了如何编译、链接、打包等构建相关操作，也能方便地固化和复用指令。

为什么要用Makefile?

- **自动化编译构建**：避免每次手动输入编译命令，简单维护项目。
- **固化命令，减少错误**：每次构建都用同一套严谨命令。
- **增量编译**：只重编修改过的代码，提升效率。
- **智能依赖管理**：自动关注依赖关系，避免错漏。
- **结构明晰**：构建流程一目了然，有助于项目交接或多人协作。
- **多任务与灵活扩展**：不仅能编译，还能扩展成执行测试、部署等任务。
- **跨平台与标准**：在主流 Unix/Linux/macOS/Windows 平台均有广泛支持和应用。

Makefile编译的内容

Makefile 文件用于描述整个工程的自动化构建规则。主要内容包括：

- 指定工程中哪些源文件需要被编译，以及具体的编译方法；
- 规定如何创建项目所需的库文件及其方法；
- 描述如何最终生成所需的可执行文件。

通过 Makefile，可以实现一条命令（通常为 make）自动完成所有这些过程，从而极大提高工程管理与编译的效率。

当使用 make 工具进行编译时，工程中以下几种文件在执行 make 时将会被编译（重新编译）：

- 所有的源文件没有被编译过，则对各个 C 源文件进行编译并进行链接，生成最后的可执行程序；
- 每一个在上次执行 make 之后修改过的 C 源代码文件在本次执行 make 时将会被重新编译；
- 头文件在上一次执行 make 之后被修改。则所有包含此头文件的 C 源文件在本次执行 make 时将会被重新编译。

后两种情况是 make 只将修改过的 C 源文件重新编译生成.o 文件，对于没有修改的文件不进 行任何工作。重新编译过程中，任何一个源文件的修改将产生新的对应的.o 文件，新的.o 文件将和 以前的已经存在、此次没有重新编译的.o 文件重新连接生成最后的可执行程序。

规则包含了目标和依赖的关系以及更新目标所要求的命令。

makefile 的常见组成

```
target: prerequisites
<TAB>command
```

组成部分	说明	示例
目标(Targets)	要生成的文件或操作名称	main.o, clean
依赖项(Prerequisites)	生成目标所需的文件/其他目标	main.c, utils.h
命令(Recipes / Commands)	生成目标的Shell命令	gcc -c main.c

一个Makefile会包含的元素

- **目标（Target）**：通常为要生成的文件或任务名。
- **依赖（Prerequisite）**：目标所依赖的文件或其他目标。
- **命令（Recipe/Command）**：为了更新目标需要执行的命令。
- **变量（Variables）**：用于简化和复用。
- **规则（Rules）**：上述三项的组合。
- **伪目标（Phony Target）**：不生成实际文件，仅为执行命令。
- **条件判断、包含指令**：提升灵活性和可维护性。
- 隐含规则

Make如何工作

默认的情况下，make会执行Makefile中的第一个规则，此规则也称之为“最终目标”。当不加任何参数执行Makefile的时候，就会执行该目标中的规则。

在Shell中输入 `make` 指令之后，make会自动读取该目录下的Makefile文件，并自动处理第一个目标里的规则。例如：

```
alpha beta gamma: common.c
    gcc $@ common.c -o $@
......

# 这是一个带有多个目标的规则，当在指令行执行 make 的时候，会默认且只执行 alpha 这个目标
# 此时 make 等价于 make alpha
# 配方用到了自动变量$@，$@会被替换为当前目标名
```

对于目标文件的更新，通常有几种情况：

1. 目标文件不存在，则执行规则创建目标；
2. 目标文件已存在，但是依赖的文件有更新，则根据规则重新链接生成

3. 目标文件已存在，依赖文件无更新，无变动

对于 `clean` 的目标在 GUN make文档中定义不能将其作为Makefile的第一个目标，因为Makefile的初衷是创建更新程序，如果只是为了clean，编写一个shell脚本也可以实现同样的功能。

Makefile的命名通常为“makefile”、“Makefile”、“GNUmakefile”（不推荐）。下达`make`指令的时候会自动查找工作目录下的上述三种命名的文件。如果makefile 文件的命名不是这三个任何一个时，需要通过 make 的`-f`或者`--file`选项来指定 make 读取的 makefile 文件。也可以通过 多个“-f”或者“--file”选项来指定多个需要读取的makefile 文件，多个 makefile 文件将会被 按照指定的顺序进行连接并被 make 解析执行。当通过`-f`或者`--file`指定 make 读取 makefile 的文件时，make 就不再自动查找这三个标准命名的 makefile 文件。

make命令执行后有三个退出码：

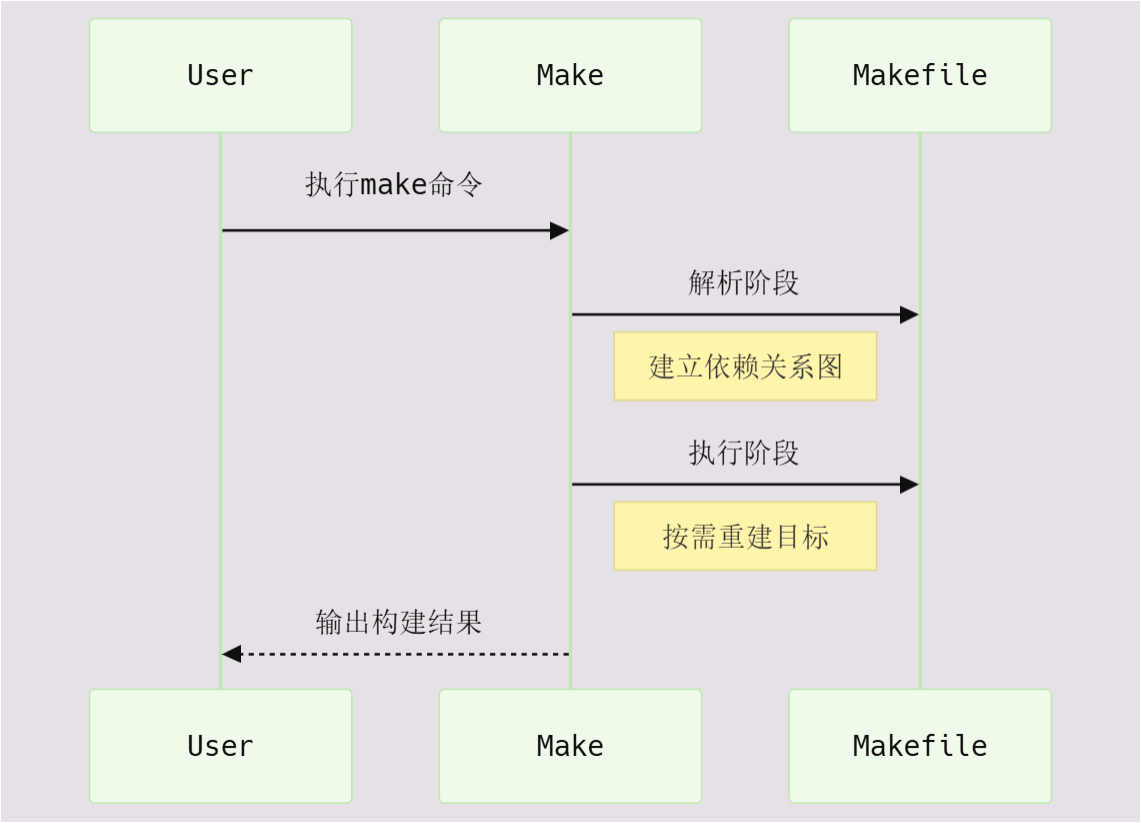
- 0 —— 表示成功执行。
- 1 —— 如果make运行时出现任何错误，其返回1。
- 2 —— 如果你使用了make的“-q”选项，并且make使得一些目标不需要更新，那么返回2。

makefile的执行过程：

阶段一：读取所有的makefile文件、以及内联的所有变量、、明确规则和隐含 规则，并建立所有目标和依赖之间的依赖关系结构链表。

阶段二：根据第一阶段已经建立的依赖关系结构链表决定哪些目标需要更新，并使用对应 的规则来重建这些目标。

1. **加载文件与变量**
   顺序读取 $MAKEFILES、当前目录下的 Makefile、被 `include` 包含的文件。
2. **自动重建 Makefile（如必要）**
   如果定义了生成 Makefile 的规则，会优先自动重建自身。
3. **构建依赖关系树**
   确定终极目标及依赖层级。
4. **判断哪些目标需更新**
   有依赖发生变化的目标需要重建。
5. **生成目标文件与最终执行文件**



增量编译机制总结：

- 只重编变化的源文件及其依赖。
- 任何头文件若变动，会影响所有包含它的源文件。

Makefile 的规则

命令是 shell 语句，默认 shell 为 `/bin/sh`。

典型例子：

```
foo.o : foo.c defs.h
    cc -c -g foo.c
```

这是一个经典的makefile例子，生成的目标是 `foo.o`，依赖的文件是 `foo.c defs.h`，源文件 `foo.c`，`-c` `-g` 分别表示“只编译成目标文件，不进行链接” 和 “在目标文件里加入调试信息”。

因此通常规则的样式为：

```
TARGETS : PREREQUISITES
    COMMAND
    ...

# 或者是

targets : prerequisites ; command
    command
    ...
```

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。
command是命令行，如果其不与“target: prerequisites”在一行，那么，必须以[Tab键]开头，如果在同一行，那么可以用分号做为分隔。
prerequisites也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。
如果命令太长，你可以使用反斜框（'\'）作为换行符。make对一行上有多少个字符没有限制。规则告诉make两件事，文件的依赖关系和如何成成目标文件。
一般来说，make会以UNIX的标准Shell，也就是`/bin/sh`来执行命令。

通配符

和bash shell类似，makefile中是可以使用通配符的，支持 `*`，`?`和`[...]` 这三个通配符，分别用于匹配任意长度的任意字符、匹配任意单个字符、匹配括号内任一单个字符

wildcard()

当尝试用通配符给变量赋值的时候，`objects = *.o` ，但此时 $objects 就是 *.o，而非所有.o文件，因此如果我们想让通配符在变量赋值时就展开，要使用 `wildcard`函数：`objects := $(wildcard *.o)`，此时就把所有的.o文件赋给了objects变量。

伪目标

一直有提到可以用 `make clean` （当存在clean目标）来清楚编译的结果，而实际上 `clean` 是一个伪目标，不同于直接用目标作为目标名。在这里，并不生成“clean”这个文件。

“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。所以通常要用 `.PHONY : clean` 来规避该风险。除此之外还有一些其他的内建特殊目标名

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“`make clean`”。

常用 all 伪目标把所有需要makefile执行的步骤涵盖，例如

```
.PHONY : all
all : prog1 prog2 prog3
prog1 : prog1.o utils.o
	cc -o prog1 prog1.o utils.o

prog2 : prog2.o
	cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
	cc -o prog3 prog3.o sort.o utils.o
```

常用的伪目标：

- `all` - 编译所有目标
- `clean` - 删除由 make 创建的文件
- `install` - 安装已编译好的程序（拷贝执行文件到指定位置）
- `print` - 列出改变过的源文件
- `tar` - 将源程序打包备份为 tar 文件
- `dist` - 创建压缩文件（通常将 tar 文件压缩为 Z 或 gz 文件）
- `TAGS` - 更新所有目标，以备完整重编译
- `check` / `test` - 测试 makefile 流程

多目标

Makefile的规则中的目标可以不止一个，也支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。但是，多个目标的生成规则的执行命令是同一个，可能会引起不必要的麻烦，使用自动化变量`$@`，这个变量表示着目前规则中所有的目标的集合。例如：

```
bigoutput littleoutput : text.g
    generate text.g -$(subst output,,$@) > $@

# 上述规则等价于：

 bigoutput : text.g
           generate text.g -big > bigoutput
 littleoutput : text.g
           generate text.g -little > littleoutput
```

静态模式

静态模式的规则是：规则存在多个目标，并且不同的目标可以根据目标文件的名字 来自动构造出依赖文件。静态模式规则比多目标规则更通用，它不需要多个目标具有相同的依赖。 但是静态模式规则中的依赖文件必须是相类似的而不是完全相同的。

适用的场景：

- 每个目标文件的依赖、命令有共同模式
- 需要很明确地指定目标

基本用法：

```
targets ... : target-pattern : prerequisites ...
    commands
```

- **targets ...**：具体要生成的目标文件列表
- **target-pattern**：带模式的目标，比如 `%.o`
- **prerequisites ...**：依赖的文件，比如 `%.c`
- **commands**：命令

例子：

假设要把 `foo.c bar.c baz.c` 分别生成 `foo.o bar.o baz.o`，最直接写法可能是：

```
foo.o: foo.c
    gcc -c foo.c

bar.o: bar.c
    gcc -c bar.c

baz.o: baz.c
    gcc -c baz.c
```

但是用了静态模式，写法可以更改为：

```
objs = foo.o bar.o baz.o

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

- `foo.o`、`bar.o`、`baz.o`就是**targets**
- `%.o`是**目标模式**
- `%.c`是**依赖模式**
- `$<`表示第一个依赖，即`foo.c`、`bar.c`等
- `$@`表示目标，即`foo.o`、`bar.o`等

自动生成依赖

在大型C/C++项目中，每个 `.c` 源文件常常会 `#include` 若干头文件（如 `.h`），而这些头文件在更新后需要让对应的 `.o` 文件重新编译。如果手动为每个 `.o` 写出所有相关 `.h` 文件的依赖关系（如 `main.o: main.c defs.h`），不仅容易出错，还很难维护。

使用场景：如果一个编译规则：`main.o : main.c defs.h config.h`，只要任何一个被修改，就要重新生成 `main.o`，如果需要增删头文件，makefile里就要手动更新。

“自动生成依赖性” 就是让编译器帮你自动找出每个源文件实际依赖的头文件，并以规则形式输出，从而让 Makefile 自动感知哪些 `.o` 文件要重新编译，无需人工维护依赖列表。

方法：比如 gcc 提供了 `-M` 或 `-MM` 参数

```
gcc -MM main.c

# 等价于

main.o: main.c defs.h config.h
```

常用示例：

1. 让编译器为每个 `.c` 文件生成一个 `.d` 文件
2. 在 Makefile 里自动“包含”所有 `.d` 文件；
3. 如何自动生成 `.d`?:

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

这样只用维护 `.c` 文件和主 Makefile，就能自动追踪依赖变化。

Makefile 的命令

每条规则中的命令和操作系统Shell的命令行是一致的。make会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以Tab键开头的，那么make会认为其是一个空命令。

在UNIX下可能会使用不同的Shell，但是make的命令默认是被“/bin/sh”——UNIX的标准Shell解释执行的。除非你特别指定一个其它的Shell。

显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，则不会输出在命令行中，例如：

```
echo Compiling xxx module

@echo Compiling xxx module

# or
rm -rf $(TARGET_FILE)

@rm -rf $(TARGET_FILE)
```

> **Note**
>
> 如果make执行时，带入make参数 `-n` 或 `--just-print`，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。
>
> 而make参数 `-s` 或 `--slient` 则是全面禁止命令的显示。
>
> 其他有关 makefile的参数总结 请参考 附录1：Makefile 参数总结

命令执行及出错

make的规则是一条一条的执行命令，如果我们希望的是下一条指令在上一条指令基础上执行，可以使用同一行加 `;` 的方式：

```
exec:
    cd /home/hchen
    pwd

# 此时 cd 并不有效，pwd仍然打印当前工作目录
# 正确用法：
exec:
    cd /home/hchen；pwd
```

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。但是我们可以选择忽略该错误，继续执行后续命令：

```
targets：
    -mkdir wapis_example/
```

自定义命令包

可以理解为makefile中的函数定义，用 `define` ... `endef` 语句把这些命令封装成一个“命令变量”(命令包）！

基本格式

```
define name_demo
command1
command2
command3
endef

# 使用
foo.c: foo.y
    $(name_demo)
```

实际例子

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef

foo.c: foo.y
    $(run-yacc)

# 上述效果等价于
foo.c: foo.y
    yacc foo.y
    mv y.tab.c foo.c
```

自动化变量:

- `$@` : 代表规则的“目标”(如 `foo.c`)
- `$^` : 代表规则的“所有依赖”(如 `foo.y`)
- `$<` : 代表规则的“第一个依赖”(如 `foo.y`)

Makefile中的变量

定义

```
# 定义方法
变量名 = 值

# Example
CC = gcc
CFLAGS = -Wall -g
OBJ = main.o util.o foo.o
```

特殊用法：

- `=`（递归变量）：右边内容在**使用时**再展开（可以引用未定义或后定义的变量）
- `:=`（简单变量）：右边内容在**定义时**立即展开
- `+=`（追加变量值）
- `?=`（条件赋值）：仅在变量未被定义时赋值

变量名：

- Makefile变量为大小写敏感
- 常用大写表示系统级/重要变量，小写或驼峰为自定义变量
- 避免与 Make 或 Shell 关键字重名，如 `SHELL`、`PATH` 等
- Makefile 默认变量是全局可见

引用

标准用法是`$(变量名)` 或 `${变量名}`也可写成 `$变量名`，但容易出错，推荐加括号。

```
all : $(OBJ)
    $(CC) $(CFLAGS) -o myapp $(OBJ)
```

高级操作

- 变量的字符串操作

  ```
  # Make 支持字符串替换
  SRC = foo.c bar.c
  OBJ = $(SRC:.c=.o)                    # 替换后缀 .c 为 .o

  # 常见用法

  # 替换后缀：
  $(VAR:%.c=%.o)
  #替换文本：
  $(VAR:old=new)
  #提取文件名：
  $(notdir $(VAR))
  #拼接路径：
  $(addprefix path/,$(VAR))
  ```

- 嵌套变量引用

  ```
  x = y
  y = z
  a := $($(x))  # a 的值为 z
  ```

- 动态变量名

```
first_second = Hello
a = first
b = second
all = $($a_$b)  # all 的值为 Hello
```

- 追加变量值

```
# 方法1
objects = main.o foo.o bar.o utils.o
objects += another.o  # objects 变为 main.o foo.o bar.o utils.o another.o

# 方法2
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

override 指示符

`override`用于强制覆盖命令行或环境变量传入的值，确保Makefile中的定义生效。用法：

```
override <variable> = <value>
override <variable> := <value>
override <variable> += <more text>

# 示例
override CFLAGS = -g  # 强制将 CFLAGS 设置为 -g
```

局部变量

有时某些目标只是提供给某一个目标，这个时候可以为某个目标单独定义一个变量：

```
CFLAGS = -O2  # 全局变量

prog : CFLAGS = -g
prog : prog.o foo.o bar.o
    $(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c
    $(CC) $(CFLAGS) prog.c
```

常见的变量以及赋值

1. **CFLAGS**

   用于指定C编译器的选项。这些选项控制编译器的行为，比如生成调试信息、优化代码或启用警告。

   常见用法：

   - `-g`：生成调试信息，方便调试。
   - `-O2`：优化代码，提高性能。
   - `-Wall`：启用所有警告，帮助发现潜在问题。

   在编译时，这些选项会通过`$(CC) $(CFLAGS)`传递给编译器（如`gcc`）。

2. **INC_FLAGS**

   用于指定头文件的包含路径，特别是当头文件不在默认路径中时

   常见用法：

   - 通过`-I`选项告诉编译器在哪里查找头文件。

   在编译时，`$(INC_FLAGS)`会被加入，例如`$(CC) $(CFLAGS) $(INC_FLAGS) -c main.c`。

3. **LIB_NAME**

   通常表示要生成的库文件的名称，常用于构建静态库（`.a`）或动态库（`.so`）。

4. **OBJECTS**

   列出目标文件（`.o`文件），这些文件是由源文件编译生成的中间文件。

   在链接时，`$(OBJECTS)`会被用作输入，例如`$(CC) -o program $(OBJECTS)`。

5. **SRC_FILES**

   列出源文件（通常是`.c`文件），这些文件需要被编译成目标文件。

   在编译时，`$(SRC_FILES)`会被编译，例如`$(CC) $(CFLAGS) -c $(SRC_FILES)`。

6. **SYSTEM_DIR**

   指定路径，通常与上述变量联合使用

示例：

```
CC = gcc                # 编译器
CFLAGS = -g -Wall -O2   # 编译选项
LIB_NAME = libmylib.a   # 库文件名
OBJECTS = main.o utils.o # 目标文件
SRC_FILES = main.c utils.c # 源文件
INC_FLAGS = -I./include # 头文件路径

all: $(LIB_NAME)

$(LIB_NAME): $(OBJECTS)
    ar rcs $@ $(OBJECTS)  # 打包成库

$(OBJECTS): $(SRC_FILES)
    $(CC) $(CFLAGS) $(INC_FLAGS) -c $(SRC_FILES)  # 编译源文件
```

```
clean:
    rm -f $(OBJECTS) $(LIB_NAME)  # 清理
```

Makefile 中的条件判断

Makefile中的条件判断类似于编程语言中的`if`语句，允许根据特定条件执行不同的定义或命令。它们在以下场景中尤为重要：

- 根据调试模式调整编译参数。
- 根据操作系统执行不同命令。
- 检查变量是否存在或值是否匹配。

Makefile支持的条件指令包括基础指令：`ifdef`、`ifndef`、`ifeq`、`ifneq`，以及GNU make中才有的`if`。

1. `ifdef` 和 `ifndef`
    - `ifdef VARIABLE`：如果变量`VARIABLE`已定义，则执行后续内容。
    - `ifndef VARIABLE`：如果变量`VARIABLE`未定义，则执行后续内容。
2. `ifeq` 和 `ifneq`
    - `ifeq (ARG1, ARG2)`：如果`ARG1`等于`ARG2`，执行后续内容。
    - `ifneq (ARG1, ARG2)`：如果`ARG1`不等于`ARG2`，执行后续内容。
    - 必须注意，此函数空格敏感，括号内不要加多余的空格
3. `if`
    - 据条件表达式的结果（真或假）执行内容。

示例

```
# 如果定义了DEBUG，启用调试；否则，优化代码。
ifdef DEBUG
    CFLAGS += -g -DDEBUG
else
    CFLAGS += -O2
endif

program: main.o
    $(CC) $(CFLAGS) -o program main.o
```

Makefile 函数

Makefile中的函数是一种强大的工具，它们能够帮助我们处理文本、文件名、条件判断等各种任务，大大增强了Makefile的灵活性和功能性。函数使得我们能够在构建过程中进行复杂的变量转换、文件操作和逻辑判断。函数在Makefile中扮演着类似编程语言中子程序的角色，但语法和使用方式有所不同。

与大多数编程语言不同，Makefile中的函数调用通常使用以下格式：

```
$(function_name parameter1,parameter2,...)

# of

${function_name parameter1,parameter2,...}

# 函数参数用逗号分隔，注意逗号前后不要有多余空格
# Makefile中的函数不需要用引号包围参数
# 注意变量是立即扩展(:=)还是延迟扩展(=)
```

分类

为了便于理解，可以将Makefile中的函数分为以下几类：

- 字符串操作函数
- 文件名操作函数
- 列表操作函数
- 控制函数

字符串操作函数

- `subst` - 字符串替换

  语法：`$(subst from,to,text)`

  功能：将`text`中所有的`from`替换为`to`。

  ```
  SOURCE_FILES = main.c utils.c helper.c
  OBJECT_FILES = $(subst .c,.o,$(SOURCE_FILES))
  # 结果：OBJECT_FILES = main.o utils.o helper.o
  ```

- `patsubst` - 模式替换

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

  功能：寻找`text`中符合`pattern`的单词，并将其替换为`replacement`。`pattern`可以包含通配符`%`，表示任意长度的字符串。

  ```
  SOURCES = foo.c bar.c baz.s ugh.h
  OBJS = $(patsubst %.c,%.o,$(SOURCES))
  # 结果：OBJS = foo.o bar.o baz.s ugh.h
  ```

- `strip` - 去除空格

  语法：`$(strip string)`

  功能：去除`string`开头和结尾的空格，并将多个连续空格缩减为一个。

  ```
  VAR = "   hello     world   "
  TRIMMED = $(strip $(VAR))
  # 结果：TRIMMED = "hello world"
  ```

- `findstring` - 查找子字符串

  语法：`$(findstring find,text)`

  功能：在`text`中查找`find`，如果找到则返回`find`，否则返回空字符串。

  ```
  HAS_MAIN = $(findstring main.c,$(SOURCES))
  # 如果SOURCES包含main.c，HAS_MAIN为"main.c"，否则为空
  ```

- `filter` 和 `filter-out` - 过滤单词

  语法：

  `$(filter pattern...,text)`

  `$(filter-out pattern...,text)`

  功能：

  `filter`：保留`text`中符合`pattern`的单词

  `filter-out`：去除`text`中符合`pattern`的单词

  ```
  SOURCES = foo.c bar.c baz.s ugh.h
  C_SRCS = $(filter %.c,$(SOURCES))
  # 结果：C_SRCS = foo.c bar.c

  NON_C_SRCS = $(filter-out %.c,$(SOURCES))
  # 结果：NON_C_SRCS = baz.s ugh.h
  ```

文件名操作函数

- `dir` - 提取目录部分

  语法：`$(dir names...)`

  功能：提取文件名中的目录部分。

  ```
  PATHS = src/main.c include/header.h
  DIRS = $(dir $(PATHS))
  # 结果：DIRS = src/ include/
  ```

- `notdir` - 提取非目录部分

  语法：`$(notdir names...)`

功能：提取文件名中的非目录部分。

```
PATHS = src/main.c include/header.h
FILES = $(notdir $(PATHS))
# 结果：FILES = main.c header.h
```

- `suffix` - 提取后缀

  语法：`$(suffix names...)`

  功能：提取文件名中的后缀。

  ```
  FILES = main.c helper.h
  SUFFIXES = $(suffix $(FILES))
  # 结果：SUFFIXES = .c .h
  ```

- `basename` - 不含后缀的文件名

  语法：`$(basename names...)`

  功能：提取不含后缀的文件名。

  ```
  FILES = main.c helper.h
  BASENAMES = $(basename $(FILES))
  # 结果：BASENAMES = main helper
  ```

列表操作函数

- `word` - 取特定位置的单词

  语法：`$(word n,text)`

  功能：返回`text`中第`n`个单词。

  ```
  FILES = main.c helper.c utils.c
  SECOND_FILE = $(word 2,$(FILES))
  # 结果：SECOND_FILE = helper.c
  ```

- `wordlist` - 取单词范围

  语法：`$(wordlist start,end,text)`

  功能：返回`text`中从第`start`到第`end`的单词。

  ```
  FILES = main.c helper.c utils.c test.c
  MIDDLE = $(wordlist 2,3,$(FILES))
  # 结果：MIDDLE = helper.c utils.c
  ```

- `sort` - 排序单词

  语法：`$(sort list)`

  功能：将`list`中的单词按字母顺序排序，并去除重复项。

```
FILES = main.c helper.c main.c utils.c
SORTED = $(sort $(FILES))
# 结果：SORTED = helper.c main.c utils.c
```

控制函数

- `foreach` - 循环处理

  语法：`$(foreach var,list,text)`

  功能：对`list`中的每个单词，将其赋值给变量`var`，然后展开`text`，最后将所有展开结果连接起来。

```
DIRS = src include tests
CLEAN_DIRS = $(foreach dir,$(DIRS),$(dir)/*)
# 结果：CLEAN_DIRS = src/* include/* tests/*
```

- `call` - 自定义函数调用

  语法：`$(call function,param1,param2,...)`

  功能：调用自定义函数，并传递参数。

```
# 定义一个函数
define create_dir
    @mkdir -p $(1)
    @echo "Directory $(1) created."
endef

# 调用函数
dirs_to_create = build output
all:
    $(foreach dir,$(dirs_to_create),$(call create_dir,$(dir)))
```

- `shell` - 执行shell命令

  语法：`$(shell command)`

  功能：执行shell命令，并返回命令的输出。

```
# 获取当前日期作为版本号
VERSION = $(shell date +%Y%m%d)
# 计算源文件数量
SRC_COUNT = $(shell ls -1 *.c | wc -l)
```

Makefile 隐含（示）规则

隐含规则（隐示规则）是 make 工具中一种非常强大的功能，它允许我们在不显式编写某些规则的情况下，让 make 自动为我们推导出如何构建目标的方法。这些规则是"隐含的"、"约定俗成的"，即使我们在 Makefile 中没有明确写出，make 也会按照这些预设的规则来执行相应的命令。

简单来说，隐含规则就像是一种"默认约定"，它让我们能够编写更加简洁的 Makefile，避免重复编写一些常见的构建规则，从而提高编写效率和可维护性。

隐含规则的优势

1. **简化 Makefile 编写**：无需为每个目标文件编写完整的构建规则
2. **提高可读性**：减少冗余代码，使 Makefile 更加简洁
3. **遵循惯例**：利用已有的开发习惯，降低学习门槛
4. **自动推导依赖**：make 能够根据目标文件自动找到合适的源文件

隐含规则的工作原理

当 make 需要构建一个目标文件，但 Makefile 中没有明确定义该目标的规则时，它会尝试使用隐含规则来构建该目标。具体步骤如下：

1. 检查目标文件的后缀（如 `.o`、`.exe` 等）
2. 在预定义的隐含规则库中查找能处理该后缀的规则
3. 根据目标文件名推导可能的源文件（如从 `file.o` 推导出 `file.c`）
4. 检查推导出的源文件是否存在
5. 如果源文件存在，应用相应的隐含规则来构建目标

示例

```
my_program: main.o utils.o
    gcc -o my_program main.o utils.o

# 注意：这里没有写如何生成 main.o 和 utils.o

# 但是隐式规则会帮助实现这行命令
# 这是一个隐式规则的“样子”，我们通常不需要写出来
%.o: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@
```

自定义隐含变量

在Makefile中，也可以利用make的 `-R` 或 `--no-builtin-variables` 参数来取消你所定义的变量对隐含规则的作用。例如，编译C程序的隐含规则的命令是 `$(CC) -c $(CFLAGS) $(CPPFLAGS)` 。Make默认的编译命令是 `cc` ，如果你把变量 `$(CC)` 重定义成 `gcc` ，把变量 `$(CFLAGS)` 重定义成 `-g` ，那么，隐含规则中的命令全部会以 `gcc -c -g $(CPPFLAGS)` 的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如 `CC` ；一种是参数相的关，如 `CFLAGS` 。下面是所有隐含规则中会用到的变量：

关于命令的变量

- `AR` : 函数库打包程序。默认命令是 `ar`
- `AS` : 汇编语言编译程序。默认命令是 `as`
- `CC` : C语言编译程序。默认命令是 `cc`
- `CXX` : C++语言编译程序。默认命令是 `g++`
- `CO` : 从 RCS文件中扩展文件程序。默认命令是 `co`
- `CPP` : C程序的预处理器（输出是标准输出设备）。默认命令是 `$(CC) -E`
- `FC` : Fortran 和 Ratfor 的编译器和预处理程序。默认命令是 `f77`
- `GET` : 从SCCS文件中扩展文件的程序。默认命令是 `get`
- `LEX` : Lex方法分析器程序（针对于C或Ratfor）。默认命令是 `lex`
- `PC` : Pascal语言编译程序。默认命令是 `pc`
- `YACC` : Yacc文法分析器（针对于C程序）。默认命令是 `yacc`
- `YACCR` : Yacc文法分析器（针对于Ratfor程序）。默认命令是 `yacc -r`
- `MAKEINFO` : 转换Texinfo源文件（.texi）到Info文件程序。默认命令是 `makeinfo`
- `TEX` : 从TeX源文件创建TeX DVI文件的程序。默认命令是 `tex`
- `TEXI2DVI` : 从Texinfo源文件创建军TeX DVI 文件的程序。默认命令是 `texi2dvi`
- `WEAVE` : 转换Web到TeX的程序。默认命令是 `weave`
- `CWEAVE` : 转换C Web 到 TeX的程序。默认命令是 `cweave`
- `TANGLE` : 转换Web到Pascal语言的程序。默认命令是 `tangle`
- `CTANGLE` : 转换C Web 到 C。默认命令是 `ctangle`
- `RM` : 删除文件命令。默认命令是 `rm -f`

关于命令参数的变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

- `ARFLAGS` : 函数库打包程序AR命令的参数。默认值是 `rv`
- `ASFLAGS` : 汇编语言编译器参数。（当明显地调用 `.s` 或 `.S` 文件时）
- `CFLAGS` : C语言编译器参数。
- `CXXFLAGS` : C++语言编译器参数。

- `COFLAGS`：RCS命令参数。
- `CPPFLAGS`：C预处理器参数。（ C 和 Fortran 编译器也会用到）。
- `FFLAGS`：Fortran语言编译器参数。
- `GFLAGS`：SCCS “get”程序参数。
- `LDFLAGS`：链接器参数。（如：`ld`）
- `LFLAGS`：Lex文法分析器参数。
- `PFLAGS`：Pascal语言编译器参数。
- `RFLAGS`：Ratfor 程序的Fortran 编译器参数。
- `YFLAGS`：Yacc文法分析器参数。

分类：后缀规则

历史上，隐式规则主要通过“**后缀规则 (Suffix Rules)** ”来定义。后缀规则基于文件的扩展名。

- 定义单个后缀规则：

  例如.c.o:，表示如何从一个 `.c` 文件生成一个 `.o` 文件

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

- `.SUFFIXES` 特殊目标：为了让 `make` 知道哪些后缀是“有意义的”，需要将这些后缀添加到 `.SUFFIXES` 特殊目标的依赖列表中。例如：

  ```
  .SUFFIXES: .o .c .cc
  ```

make有一个默认的后缀列表，包含了很多常用后缀。

然而，后缀规则有一些限制，例如它不能很好地处理没有后缀的文件，或者文件名中包含目录路径的情况。

分类：模式规则

因此，现代的 `make` 更推荐使用“**模式规则 (Pattern Rules)** ”，它更加灵活和强大。

- 模式规则的语法：`%.o: %.c`
  - `%` 是一个通配符，匹配任意非空字符串。
  - 这个规则表示：任何一个 `.o` 文件依赖于一个同名的 `.c` 文件。
  - 例如，目标 `main.o` 会匹配这个模式，`%` 匹配 `main`，于是它依赖于 `main.c`。

大部分 `make` 的内建隐式规则现在都是用模式规则定义的。 模式规则比后缀规则更通用，也更容易理解。虽然后缀规则仍然被支持以保证向后兼容性，但在编写新的 Makefile 时，如果需要自定义隐式规则，优先考虑模式规则。

隐式规则的控制方法

1. **覆盖隐式规则**
   如果为某个特定的目标写了一个显式的规则，那么 `make` 就不会再为这个目标去查找和使用隐式规则了。你的显式规则优先级更高。

   ```
   main.o: main.c
       echo "Compiling main.c with a custom command!"
       gcc -c main.c -o main.o

   # 对于 main.o，上面的显式规则会被使用，而不是内建的 .c.o 隐式规则。
   # 其他 .o 文件（如 utils.o，如果它依赖 utils.c）仍然会使用隐式规则。
   ```

2. **取消所有内建隐式规则**

   可以使用 `make` 的命令行选项 `-r` 或 `--no-builtin-rules` 来禁止所有内建的隐式规则。这样做之后，Makefile 中所有目标的生成都必须有显式规则，否则 `make` 会报错。这在某些情况下，你希望完全掌控所有构建步骤时可能有用。(注意：即使使用了 `-r`，如果 `.SUFFIXES` 列表不为空，某些基于后缀的隐式规则仍可能生效。彻底禁用通常还需要清空 `.SUFFIXES`，例如 `.SUFFIXES:`)。

3. **取消特定的内建隐式规则**

   如果你想禁用某个特定的隐式规则（例如，你不希望 `.c` 文件自动编译成 `.o` 文件），你可以为该模式定义一个空的规则。这会覆盖掉内建的模式规则。

   ```
   %.o: %.c
       # 空命令，这样就不会执行任何操作来通过 .c 生成 .o
       # 或者你也可以给出一个明确的 @echo "Implicit .c.o rule disabled"
   ```

4. **`make -p` 查看规则数据库**

   如果想知道 `make` 当前知道了哪些规则（包括所有内建的隐式规则、变量等），可以使用 `make -p` (或 `make --print-data-base`) 命令。输出会非常多，但对于理解 `make` 的内部状态非常有用。

5. **链式隐式规则**

   `make` 非常智能，它可以将多个隐式规则串联起来使用。例如，如果有一个 Yacc 文件 `parser.y`，`make` 可能知道如何从 `.y` 生成 `.c` (例如生成 `parser.c`)，然后又知道如何从 `.c` 生成 `.o` (生成 `parser.o`)。
   所以，如果你的规则是 `program: parser.o`，并且目录下有 `parser.y`，`make` 可能会自动：

   - `parser.y` -> `parser.c` (隐式规则，生成中间文件 `parser.c`)
   - `parser.c` -> `parser.o` (隐式规则)
     `make` 通常会自动删除这些由它自己生成的中间文件 (如 `parser.c`)，除非这些文件本身也是其他规则的目标或依赖。

Makefile的一些特殊使用方法

包含其他makefile文件

1. include

   Makefile中包含其他文件的关键字是 `include`，使用方法：`include FILENAMES`，这里的 `FILENAMES` 是shell支持的文件名并且也可以使用通配符。

   > **Note**
   >
   > include行不能用[Tab] 开始

2. 变量 `$MAKEFILES`

   可以在当前的环境定义一个环境变量 `$MAKEFILES`，make执行的时候会先读入此变量，变量可以涵盖多个Makefile文件（用空格隔开）。

   但是需要注意：

   - 用该方法引入的Makefile不会被当做make的“最终目标”，如果在工作目录下不存在可被找到的Makefile，则会报错；
   - 环境变量方法的使用即便有错误，make也不会提示，程序也会继续运行；
   - 该方法在make 执行的时候会先被读取，然后才会执行Makefile文件里的内容；而 `include` 的方法是可以随时打断程序运行转到要读取的Makefile。

每个执行的Makefile会被追加到环境变量 `$MAKEFILE_LIST` 中，想知道在执行了哪些Makefile可以打印该变量。

特殊使用场景

makefile不允许一个目录下或一次运行中存在相同目标的两个不同规则命令，但依赖可以。如果一个Makefile需要用到另一个Makefile 的变量和规则，可以使用 所有匹配模式，或者使用 `:` 追加命令。

文件的搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当make需要去找寻文件的依赖关系时，一般有两种方法：

1. 设定路径
2. 给一个路径给make，让make去寻找需要的依赖文件

Makefile文件中的特殊变量 `VPATH` 是用来实现方法2的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。例如： `VPATH = src:../headers`

此外除了设置VPATH变量，还可以使用 `vpath` 关键字，

```
# example
    vpath %.h ../headers

# Methods
    # 常规使用方法
```

```
    vpath < pattern> < directories>
    # 清楚搜索路径
    vpath < pattern>
    # 恢复默认
    vpath

# 说明
< pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符

# 示例
   vpath %.c foo
   vpath %   blish
   vpath %.c bar
```

嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

Example：

1. 假设目录为

```
project/
├── Makefile     # 总控 Makefile
├── subdir1/
|   └── Makefile # subdir1 的 Makefile
├── subdir2/
|   └── Makefile # subdir2 的 Makefile
├── ...
```

2. 在最上层Makefile 调用子目录的 make

```
subsystem:
    cd subdir1 && $(MAKE)
    cd subdir2 && $(MAKE)

# 或
subsystem:
    $(MAKE) -C subdir1
    $(MAKE) -C subdir2

# -C dir 相当于 cd dir && make，但是更安全、简洁。
```

   - `$(MAKE)` 是 GNU make 预定义变量，保证正确传递当前 make 的参数（如并行构建选项等），更安全可靠。

3. 变量的传递

从**上级 Makefile 到下级 Makefile**，只有 `SHELL` 和 `MAKEFLAGS` 始终传递。要传递自定义变量，需要用 `export`：

```
# 传递 MYFLAG
export MYFLAG = YES

# 一次性传递所有变量
export

# 不传递某个变量
unexport VAR
```

4. 变量的覆盖关系

   如果一个变量在下级 Makefile 里已经被定义，上级用 `export` 传下来的同名变量不会覆盖它，除非使用 `make -e`，`make -e` 表示“环境变量优先”。

常见问题及补充

补充

内建的特殊目标名

- **.PHONY**：声明伪目标，无论文件是否存在都执行命令。
- **.SUFFIXES**：指定用于后缀规则的后缀列表（老式用法）。
- **.DEFAULT**：给没有规则的目标指定默认命令。
- **.PRECIOUS**：目标不会因中断或无用而被自动删除，常保护中间文件。
- **.INTERMEDIATE**：依赖被作为中间文件，通常会在不需要时被自动删除。
- **.SECONDARY**：像中间文件一样处理，但永不自动删除。
- **.DELETE_ON_ERROR**：命令出错时删除相应目标。
- **.IGNORE**：忽略指定目标或所有目标的命令错误。
- **.SILENT**：指定目标或全局执行命令时不回显。
- **.EXPORT_ALL_VARIABLES**：自动传递所有变量到子 make 进程。
- **.NOTPARALLEL**：禁止当前 makefile 的目标并行构建。

自动变量

- `$@` ： 代表目标文件名
- `$^` ： 所有的依赖文件名（去重，空格分隔）
- `$<` ： 代表第一个依赖文件名
- `$?`： 代表比目标新的依赖文件
- `$%` : 仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是 `foo.a(bar.o)` ，那么，`$%` 就是 `bar.o` ， `$@` 就是 `foo.a` 。
- `$+` : 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/foo` 。

这几个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上 `D` 或 `F` 字样。这是GNU make中老版本的特性，在新版本中，我们使用函数 `dir` 或 `notdir` 就可以做到了。例如：

- `$(@D)`

  表示 `$@` 的目录部分（不以斜杠作为结尾），如果 `$@` 值是 `dir/foo.o` ，那么 `$(@D)` 就是 `dir` ，而如果 `$@` 中没有包含斜杠的话，其值就是 `.` （当前目录）。

- `$(@F)`

  表示 `$@` 的文件部分，如果 `$@` 值是 `dir/foo.o` ，那么 `$(@F)` 就是 `foo.o` ， `$(@F)` 相当于函数 `$(notdir $@)` 。

常见问题

1. 一个较长行可以使用反斜线（\）分解为多行，这样做可以使Makefile清晰、容易阅读。

   > **Note**
   >
   > 注反斜线之后不能有空格

2. 命令行的 [Tab]

   命令行必需以[Tab]键开 始，以和Makefile其他行区别。就是说所有的命令行必需以[Tab] 字符开始，但并不是所有的以 [Tab]键出现行都是命令行。但make程序会把出现在第一条规则之后的所有的以[Tab]字符开始的行都作为命令行来处理。

3. 书写规则建议的方式是：单目标，多依赖。就是说尽量要做到一个规则中只存在一个目标文件， 可有多个依赖文件。尽量避免多目标，单依赖的方式。这样后期维护也会非常方便，而且 Makefile 会更清晰、明了。
4. Makefile中可以使用 `-` 放在指令的开头，表示可以忽略该操作的错误，让make可以继续执行
5. 使用 `ifeq` `ifdef` 后忘记 `endif`
6.

附录1：Makefile 参数总结

参数之间可以根据不同需求组合使用，例如：`make -n -p`查看规则但不执行，或`make -C ~/project -j4`在指定目录下使用4个并行任务编译。

检查规则参数

- `-n` / `--just-print` / `--dry-run` / `--recon` - 只打印命令而不执行，用于调试
- `-t` / `--touch` - 更新目标文件时间戳但不实际编译
- `-q` / `--question` - 检查目标是否需要更新，不输出任何内容
- `-W <file>` / `--what-if=<file>` / `--assume-new=<file>` / `--new-file=<file>` - 假定指定文件已更新，可与`-n`配合使用观察影响

基本功能

- `-B` / `--always-make` - 强制重新编译所有目标
- `-C <dir>` / `--directory=<dir>` - 指定读取makefile的目录
- `-f <file>` / `--file=<file>` / `--makefile=<file>` - 指定执行的makefile文件
- `-j [<jobsnum>]` / `--jobs[=<jobsnum>]` - 指定同时运行的命令数量
- `-k` / `--keep-going` - 出错后继续执行其他非依赖任务

输出控制

- `--debug[=<options>]` - 输出调试信息，选项包括a(全部)、b(基本)、v(详细)、i(隐含规则)等
- `-d` - 等同于`--debug=a`
- `-p` / `--print-data-base` - 输出makefile所有数据，包括规则和变量
- `-s` / `--silent` / `--quiet` - 不显示命令执行时的输出
- `-v` / `--version` - 显示make版本信息
- `-w` / `--print-directory` - 显示执行目录信息，适合跟踪嵌套make调用

规则控制

- `-e` / `--environment-overrides` - 环境变量优先于makefile中的变量
- `-i` / `--ignore-errors` - 忽略所有执行错误
- `-I <dir>` / `--include-dir=<dir>` - 指定被包含makefile的搜索目录
- `-r` / `--no-builtin-rules` - 禁用内置隐含规则
- `-R` / `--no-builtin-variables` - 禁用内置变量
- `--warn-undefined-variables` - 对未定义变量发出警告

附录2：隐含规则

1. 编译C程序的隐含规则。

   `<n>.o` 的目标的依赖目标会自动推导为 `<n>.c` ，并且其生成命令是 `$(CC) -c $(CPPFLAGS) $(CFLAGS)`

2. 编译C++程序的隐含规则。

   `<n>.o` 的目标的依赖目标会自动推导为 `<n>.cc` 或 `<n>.cpp` 或是 `<n>.C` ，并且其生成命令是 `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` 。（建议使用 `.cc` 或 `.cpp` 作为C++源文件的后缀，而不是 `.C` ）

3. 编译Pascal程序的隐含规则。

   `<n>.o` 的目标的依赖目标会自动推导为 `<n>.p` ，并且其生成命令是 `$(PC) -c $(PFLAGS)` 。

4. 编译Fortran/Ratfor程序的隐含规则。

   `<n>.o` 的目标的依赖目标会自动推导为 `<n>.r` 或 `<n>.F` 或 `<n>.f` ，并且其生成命令是:

   - `.f` `$(FC) -c $(FFLAGS)`
   - `.F` `$(FC) -c $(FFLAGS) $(CPPFLAGS)`
   - `.r` `$(FC) -c $(FFLAGS) $(RFLAGS)`

5. 预处理Fortran/Ratfor程序的隐含规则。

   `<n>.f` 的目标的依赖目标会自动推导为 `<n>.r` 或 `<n>.F` 。这个规则只是转换Ratfor 或有预处理的Fortran程序到一个标准的Fortran程序。其使用的命令是：

   - `.F` `$(FC) -F $(CPPFLAGS) $(FFLAGS)`
   - `.r` `$(FC) -F $(FFLAGS) $(RFLAGS)`

6. 编译Modula-2程序的隐含规则。

   `<n>.sym` 的目标的依赖目标会自动推导为 `<n>.def` ，并且其生成命令是： `$(M2C) $(M2FLAGS) $(DEFFLAGS)` 。 `<n>.o` 的目标的依赖目标会自动推导为 `<n>.mod` ，并且其生成命令是： `$(M2C) $(M2FLAGS) $(MODFLAGS)` 。

7. 汇编和汇编预处理的隐含规则。

   `<n>.o` 的目标的依赖目标会自动推导为 `<n>.s` ，默认使用编译器 `as` ，并且其生成命令是： `$ (AS) $(ASFLAGS)` 。 `<n>.s` 的目标的依赖目标会自动推导为 `<n>.S` ，默认使用C预编译器 `cpp` ，并且其生成命令是： `$(CPP) $(CPPFLAGS)` 。

8. 链接Object文件的隐含规则。

   `<n>` 目标依赖于 `<n>.o` ，通过运行C的编译器来运行链接程序生成（一般是 `ld` ），其生成命令是： `$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)` 。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效。例如如下规则:

   ```
   x : y.o z.o
   ```

   并且 `x.c` 、 `y.c` 和 `z.c` 都存在时，隐含规则将执行如下命令:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

如果没有一个源文件（如上例中的x.c）和你的目标名字（如上例中的x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9. Yacc C程序时的隐含规则。

   `<n>.c` 的依赖文件被自动推导为 `n.y`（Yacc生成的文件），其生成命令是：`$(YACC) $(YFALGS)`。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10. Lex C程序时的隐含规则。

    `<n>.c` 的依赖文件被自动推导为 `n.l`（Lex生成的文件），其生成命令是：`$(LEX) $(LFALGS)`。（关于“Lex”的细节请查看相关资料）

11. Lex Ratfor程序时的隐含规则。

    `<n>.r` 的依赖文件被自动推导为 `n.l`（Lex生成的文件），其生成命令是：`$(LEX) $(LFALGS)`。

12. 从C程序、Yacc文件或Lex文件创建Lint库的隐含规则。

    `<n>.ln`（lint生成的文件）的依赖文件被自动推导为 `n.c`，其生成命令是：`$(LINT) $(LINTFALGS) $(CPPFLAGS) -i`。对于 `<n>.y` 和 `<n>.l` 也是同样的规则。

附录3：gcc编译器选项

1. 编译控制与输出类型

- `-c`
  - **作用**：只编译或汇编源文件，不进行链接。生成目标文件（通常是 `.o` 文件）。
  - **示例**：`gcc -c main.c` (生成 `main.o`)
- `-o <file>`
  - **作用**：指定输出文件的名称为 `<file>`。
  - **示例**：`gcc main.c -o my_program` (生成可执行文件 `my_program`)；`gcc -c main.c -o main_obj.o` (生成目标文件 `main_obj.o`)
- `-S`
  - **作用**：编译后停止，不进行汇编。生成汇编语言文件（通常是 `.s` 文件）。
  - **示例**：`gcc -S main.c` (生成 `main.s`)
- `-E`
  - **作用**：仅运行预处理器。预处理的结果（通常是展开了宏和头文件的 C/C++ 代码）会输出到标准输出。
  - **示例**：`gcc -E main.c > main.i` (将预处理后的内容保存到 `main.i`)

2. 警告选项

- `-Wall`
  - **作用**：开启大部分常用的警告。强烈推荐使用，有助于发现潜在的代码问题。
  - **示例**：`gcc -Wall main.c`
- `-Wextra` （或者旧版的 -W）
  - **作用**：开启一些 `-Wall` 未包含的额外警告。
  - **示例**：`gcc -Wall -Wextra main.c`
- `-Werror`
  - **作用**：将所有警告视为错误，导致编译在出现警告时失败。这有助于强制解决所有警告。
  - **示例**：`gcc -Wall -Werror main.c`
- `-pedantic`
  - **作用**：发出标准 ISO C/C++ 禁止的所有警告，并拒绝使用了禁止的扩展的程序。
- `-w`(小写w):
  - **作用**：禁止所有警告信息。**不推荐**在正常开发中使用。

3. 调试选项

- `-g`
  - **作用**：在编译生成的目标代码中包含调试信息（例如变量名、行号等）。这是使用调试器（如 GDB）的前提。
  - **示例**：`gcc -g main.c -o my_program`
- `-g<level>`: (例如-g0, -g1, -g2, -g3
  - **作用**：控制生成调试信息的详细程度。`-g` 通常等同于 `-g2`。`-g3` 包含更多信息，包括宏定义。`-g0` 表示不生成调试信息。

4. 优化选项

- `-O0`
  - **作用**：不进行优化。这是默认级别。编译速度最快，但生成的代码效率较低。
- `-O1` 或 `-o`
  - **作用**：进行基础级别的优化。
- `-O2`
  - **作用**：进行更高级别的优化。通常是发布版本推荐的优化级别，在编译时间和代码性能之间有较好的平衡。
- `-O3`
  - **作用**：进行最高级别的优化。可能会增加编译时间，有时甚至可能因为某些积极的优化导致代码体积变大或行为异常（尽管很少见）。
- `-Os`
  - **作用**：专门为减小生成代码的体积进行优化。
- `-Ofast`

- **作用**：启用所有 `-O3` 优化，并启用那些可能不完全符合标准的优化。

- `-march=<cpu-type>`
  - **作用**：为指定的 CPU 类型生成优化的代码，例如 `-march=native` 会为当前编译机器的 CPU 进行优化。
- `-mtune=<cpu-type>`
  - **作用**：为指定的 CPU 类型调整代码，但不使用该 CPU 特有的指令集。

5. 预处理器选项

- `-D<macro>`
  - **作用**：定义宏 `<macro>`，其值默认为 `1`。
  - **示例**：`gcc -DDEBUG main.c` (等同于在代码中 `#define DEBUG 1`)
- `-D<macro>=<value>`
  - **作用**：定义宏 `<macro>` 并将其值设为 `<value>`。
  - **示例**：`gcc -DVERSION="1.0"` main.c
- `-U<macro>`
  - **作用**：取消宏 `<macro>` 的定义。
- `-I<dir>`
  - **作用**：将目录 `<dir>` 添加到头文件搜索路径列表的开头。当代码中 `#include "header.h"` 或 `#include <header.h>` 时，编译器会在此目录中查找。
  - **示例**：`gcc -I../include main.c` (在 `../include` 目录下查找头文件)

6. 链接器选项 (通常通过 `gcc`/`g++` 命令传递给链接器)

- `-L<dir>`
  - **作用**：将目录 `<dir>` 添加到库文件搜索路径列表的开头。链接器会在这些目录中查找由 `-l` 选项指定的库。
  - **示例**：`gcc main.o -L../libs -lcustom` (在 `../libs` 目录下查找 `libcustom.so` 或 `libcustom.a`)
- `-l<library_name>`
  - **作用**：链接名为 `lib<library_name>.so` (共享库) 或 `lib<library_name>.a` (静态库) 的库文件。
  - **示例**：`gcc main.o -lm` (链接数学库 `libm.so`)；`gcc main.o -lpthread` (链接 POSIX 线程库)
- `-shared`
  - **作用**：生成一个共享库文件（例如 `.so` 文件）。
  - **示例**：`gcc -shared -fPIC mylib.c -o libmylib.so`
- `-static`
  - **作用**：强制使用静态链接，即使有共享库可用。
- `-Wl,<option>`
  - **作用**：将逗号后的 `<option>` 直接传递给链接器。
  - **示例**：`gcc main.o -Wl,-rpath=/opt/custom_libs` (设置运行时库搜索路径)

7. 语言标准选项

- `-std=<standard>`
  - **作用**：指定编译时遵循的 C 或 C++ 语言标准。
  - **C 示例**: `-std=c90`, `-std=c99`, `-std=c11`, `-std=c17`, `-std=c2x`
  - **C++ 示例**: `-std=c++98`, `-std=c++03`, `-std=c++11`, `-std=c++14`, `-std=c++17`, `-std=c++20`, `-std=c++23`
  - **GNU 扩展**: `-std=gnu99` (C99 + GNU 扩展), `-std=gnu++11` (C++11 + GNU 扩展)

8. 其他杂项

- `-fPIC` 或 `-fpic`
  - **作用**：生成位置无关代码 (Position Independent Code)。在编译共享库时通常需要此选项。
- `-pthread`
  - **作用**：为 POSIX 线程编程添加支持（通常会同时设置预处理器宏并链接 `libpthread`）。
- `--version`
  - **作用**：显示编译器的版本信息。
- `--help`
  - **作用**：显示编译器支持的选项摘要。
- `-v`
  - **作用**：显示编译器执行的详细步骤，包括调用的子程序和版本信息。

附录4：函数补充

1. 字符串操作函数

除了之前提到的`subst`、`patsubst`、`strip`、`findstring`、`filter`等，还有：

- `$(words text)` - 计算单词数量
- `$(firstword names...)` - 返回第一个单词
- `$(lastword names...)` - 返回最后一个单词

2. 文件名操作函数

除了之前提到的`dir`、`notdir`、`suffix`、`basename`等，还有：

- `$(join list1,list2)` - 将两个列表对应位置的单词连接起来
- `$(realpath names...)` - 返回绝对路径
- `$(abspath names...)` - 返回绝对路径，但不解析符号链接

3. 条件函数

- `$(if condition,then-part[,else-part])` - 条件判断
- `$(or condition1[,condition2[,condition3...]])` - 逻辑或
- `$(and condition1[,condition2[,condition3...]])` - 逻辑与

4. 循环函数

- `$(foreach var,list,text)` - 循环处理列表中的每个项目

5. 文件操作函数

- `$(wildcard pattern...)` - 获取匹配模式的文件列表
- `$(file op filename[,text])` - 文件读写操作

6. 控制函数

除了之前提到的`error`，还有：

- `$(warning text...)` - 输出警告信息，但不会中断make的执行过程
- `$(info text...)` - 输出信息
- `$(eval text)` - 在运行时评估文本作为makefile片段

7. 原生shell命令

- `$(shell command)` - 执行shell命令并返回输出

8. 值函数

- `$(value variable)` - 获取变量的文本值，而不是展开后的值
- `$(origin variable)` - 返回变量的来源（如命令行、环境变量等）
- `$(flavor variable)` - 返回变量的赋值方式（如简单赋值、递归赋值等）

9. 其他特殊函数

- `$(guile scheme-code)` - 执行Guile扩展中的Scheme代码（如果支持）