1. Linux编译过程
2. -E 预处理：把.h和.c展开形成一个文件，宏定义直接替换，头文件，库文件直接打开，最终生成一个.i文件。(gcc -E hello.c -o hello.i)
3. -S 汇编：.i生成一个汇编代码文件，.S (gcc -S hello.i -o hello.S)
4. -c 编译：.S生成.o .obj (gcc -c hello.S -o hello.o)
5. -o 链接：.o 链接成.exe(windows), .elf(linux) (gcc hello.o -o hello)
6. Makefile 目标和前置条件

（1）Target(目标)可以是文件，也可以是某个操作的名字，这称为“伪目标”。

clean:

rm \*.o

以上clean是一个操作，删除所有.o文件，但是当目录下有一个名为clean的文件时，make clean则不会执行。

为了避免这种情况，可以明确的声明clean为“伪目标”：

.PHONY: clean

clean:

rm \*.o temp

执行make命令后，它就不会去检查是否存在clean文件了。

1. 前置条件通常时一些文件名，只要前置文件不存在或者被更改，那么“目标”都会被更新重置。

result.txt: source.txt

cp source.txt result.txt

以上make能执行，需要前置文件sorce.txt存在，不然就得加一条规则来生成该目标：

source.txt:

echo "this is the source" > source.txt

以上source.txt相当于一个“伪目标”，只要该文件不存在，就执行规则。

1. makefile的执行顺序：

makefile文件的执行顺序是先从上至下执行Target规则之外的语句（包括变量赋值和函数的扩展），然后才执行相对应的目标及规则。具体执行顺序参考：

<https://blog.csdn.net/qq_35524916/article/details/77131555>

Makefile的执行顺序除了跟目标规则有关外，还以增加条件，如：

$(player) : $(libraries)

$(lib\_ui) : $(lib\_db) $(lib\_codec)

以上表示$(player)会依赖$(libraries)，$(lib\_ui) 会依赖$(lib\_db) $(lib\_codec)，也就是说要先生成$(libraries)，才可以生成$(player) 。

1. Command
2. 命令由一条或者多条shell命令组成，命令的结果一般是生成“目标”。每行命令必须有一个Tab键，而且每行命令在一个单独的shell中执行，属于不同的进程，它们之间没有继承的关系。

通常情况下，同一时刻只有一个命令在执行，下一个命令只有在当前命令结束之后才能够开始执行。不过可以通过 make 命令行选项 "-j" 或者 "--jobs" 来告诉 make 在同一时刻可以允许多条命令同时执行（不推荐使用）。

var-lost:

export foo=bar

echo "foo=[$$foo]"

以上执行“make var-lost”之后，取不到foo的值。

但是可以将它们写在一行，中间用“;”隔开；也可以在换行前加反斜杠转义，命令之间同样得加“;”；最后办法是加上.ONESHELL:命令：

Eg: var-kept:

export foo=bar; \

echo "foo=[$$foo]"

Eg: .ONESHELL:

var-kept:

export foo=bar;

echo "foo=[$$foo]"

（2）命令的回显

在执行make时，命令都会回显打印出来，可以在命令之前加上“@”，避免它回显。

Make -n 或者 make --just-print：执行时只显示要执行的命令，但是命令它不会去执行

all:

echo hello

echo world

输出：

Echo hello

Echo world

Make -s 或者 make --slient：则会禁止打印所有的命令，但是命令是会执行的。

（3）命令出错

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。

忽略命令的出错，我们可以在Makefile的命令行前加一个减号 - （在Tab键之后），标记为不管命令出不出错都认为是成功的。

clean:

-rm -f \*.o

还有一个全局的办法是，给make加上 -i 或是 --ignore-errors 参数，那么，Makefile中所有命令都会忽略错误。而如果一个规则是以 .IGNORE 作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的make的参数的是 -k 或是 --keep-going ，这个参数的意思是，如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其它规则。

1. 变量和赋值符

Makefile可以自定义变量：

txt = Hello World

test:

@echo $(txt)

以上：变量txt等于“Hello World”，调用时用$()或者${}。符号“@”表示执行时不打印“echo $(txt)”这条命令，不然它会有回声，即打印每一条命令。

**区别**：当调用shell变量时（命令行中定义的变量），需$$()来调用，那是因为执行make命令时，该命令会对美元符合进行转义。

**变量的赋值**：

简单赋值 ( := ) 编程语言中常规理解的赋值方式，只对当前语句的变量有效。

递归赋值 ( = ) 赋值语句可能影响多个变量，所有目标变量相关的其他变量都受影响。

条件赋值 ( ?= ) 如果变量未定义，则使用符号中的值定义变量。如果该变量已经赋值，则该赋值语句无效。

追加赋值 ( += ) 原变量用空格隔开的方式追加一个新值。

Note:

（1）简单赋值 ( := )：在当前的语句有效，指的是赋值当前变量的值，之后即使变量的值改变，也不会改变该赋值语句；

（2）递归赋值 ( = )：它则相反，之后的变量值变化，改赋值语句也会改变；

**内置变量：**

Make命令提供一系列内置变量，比如，$(CC) 指向当前使用的编译器（gcc或者其他），$(MAKE) 指向当前使用的Make工具。

**自动变量：**

1. $@

$@指代当前目标，就是Make命令当前构建的那个目标。

1. $<

$< 指代第一个前置条件。

1. $^

$^ 指代所有前置条件，之间以空格分隔。

1. $?

$? 指代比目标更新的所有前置条件，之间以空格分隔。

1. $\*

$\* 指代匹配符 % 匹配的部分， 比如% 匹配 f1.txt 中的f1 ，$\* 就表示 f1。

1. $(@D) 和 $(@F)

$(@D) 和 $(@F) 分别指向 $@ 的目录名和文件名。比如，$@是 src/input.c，那么$(@D) 的值为 src ，$(@F) 的值为 input.c。

1. $(<D) 和 $(<F)

$(<D)和$(<F)分别指向$<的目录名和文件名。

**系统预定义变量**：

除了自定义变量以外，还有一些系统自己定义好的变量，如CFLAGS，cc，MAKE等等。参考：<https://blog.csdn.net/WU9797/article/details/77454162>

关于CFLAGS,LDFLAGS,LIBS的一些描述：

<https://blog.csdn.net/hxlawf/article/details/94623786>

<https://www.cnblogs.com/taskiller/archive/2012/12/14/2817650.html>

**目标变量**：

之前定义的变量都是全局变量，谁都可以用。我们也可以为某个目标定义局部变量，它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

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

foo.o : foo.c

$(CC) $(CFLAGS) foo.c

bar.o : bar.c

$(CC) $(CFLAGS) bar.c

以上$(CFLAGS)为目标prog的局部变量。

**变量的高级用法**：

（1）变量的替换引用：

foo:=a.c b.c d.c

obj:=$(foo:.c=.o)

All:

@echo $(obj)

等价于

foo:=a.c b.c d.c

obj:=$(foo:%.c=%.o)

All:

@echo $(obj)

注意：括号中的变量使用的是变量名而不是变量名的引用，变量名的后面要使用冒号和参数选项分开，表达式中间不能使用空格。

1. 变量的嵌套使用

foo=bar

bar=test

var:=$($(foo))

All:

@echo $(var)

输出：test

**用override关键字定义变量**：

变量的赋值会覆盖之前变量的值，如（“=”“：=”“define”）。有时我们不希望后面的变量赋值影响之前变量的值，那么我们可以在变量之前加上override关键字，这样后面的变量赋值就不会影响之前的值了。如果要改变变量的值的话，同样还需要加override关键字重新赋值。

<https://blog.csdn.net/wh_19910525/article/details/41279909>

**define-endef的使用**:

它一般用于定义打包一些重用的命令，不用于定义变量，定义的变量是无效的。

<https://blog.csdn.net/seven_tree/article/details/106060613>

**Makefile 变量MAKEFILE\_LIST：**

<https://blog.csdn.net/qu1993/article/details/89020034>

**Makefile 变量MAKECMDGOALS：**

**<https://www.cnblogs.com/chenhuan001/p/6970686.html>**

1. 条件判断语句

ifeq ($(CC),gcc)

libs=$(libs\_for\_gcc)

else

libs=$(normal\_libs)

endif

以上语句判断编译器类别，根据类别选用不同的lib库。

条件判断关键字：ifeq和ifneq，ifdef和ifndef(判断变量的值是否为空)

Note:条件判断语句只能用于控制make实际执行的语句；但是，不能控制规则中命令的执行过程。也就是说不能写到控制规则当中。

1. for循环语句

用法：

LIST = one two three

all:

for i in $(LIST); do \

echo $$i; \

done

# 等同于

\all:

for i in one two three; do \

echo $$i;

done

Note：在makefile中使用for语句，其行结束要使用“；\”。而且for循环只能用于控制规则当中，不能出现在makefile语句中。

1. 目标文件搜索（VPATH和vpath）

VPATH 和 vpath 的区别：VPATH 是变量，更具体的说是环境变量，Makefile 中的一种特殊变量，使用时需要指定文件的路径；vpath 是关键字，按照模式搜索，也可以说成是选择搜索。搜索的时候不仅需要加上文件的路径，还需要加上相应限制的条件，它这里的路径是指相对路径。

**VPATH的使用**：

（1）VPATH:=src

表示在执行make的时候，会从src目录下搜索。首先搜索的是自己的当前目录，没有再去src目录搜索。

（2）VPATH:=src include或者VPATH:=src:include

多个路径之间要用空格或者冒号隔开，搜索的顺序为我们书写的顺序。

**vpath的使用**：

1. vpath PATTERN DIRECTORIES ( PATTERN：可以理解为要寻找的条件，DIRECTORIES：寻找的路径 )

vpath test.c src

表示在src目录下搜索test.c文件。或者多路径搜索：

vpath test.c src car

1. vpath PATTERN

vpath test.c

表示清除符合文件 test.c 的搜索目录。

1. vpath

表示清除所有已被设置的文件搜索路径。

1. Makefile 隐含规则

test:test.o

gcc -o test test.o

test.o:test.c

和

test:test.o

gcc -o test test.o

执行结果是一样的。这是隐含规则的作用，在某些时候，不需要建立目标规则及命令，它自己就自动化完成了。

注意：隐含规则只能省略重建中间目标文件的规则和命令，但是最终目标的规则和命令不能省略。

（1）隐含规则的具体的工作流程：make 执行过程中找到的隐含规则，提供了此目标的基本依赖关系。确定目标的依赖文件和重建目标需要使用的命令行。隐含规则所提供的依赖文件只是一个基本的（在C语言中，通常他们之间的对应关系是：test.o 对应的是 test.c 文件）。当需要增加这个文件的依赖文件的时候要在 Makefile 中使用没有命令行的规则给出。

test:test.o

gcc -o test test.o

test:test1.h

1. 其实在有些时候隐含规则的使用会出现问题。因为有一个 make 的“隐含规则库”。库中的每一条隐含规则都有相应的优先级顺序，优先级也就会越高，使用时也就会被优先使用。当我们不需要使用隐含规则时，一定要加上目标和规则。

下面是一些常用的隐含规则：

* 编译 C 程序
* 编译 C++ 程序
* 编译 Pascal 程序
* 编译 Fortran/Ratfor 程序
* 预处理 Fortran/Ratfor 程序
* 编译 Modula-2 程序
* 汇编和需要预处理的汇编程序
* 链接单一的 object 文件
* Yacc C 程序
* Lex C 程序时的隐含规则

1. Include 文件包含

当 make 读取到 "include" 关键字的时候，会暂停读取当前的 Makefile，而是去读 "include" 包含的文件，读取结束后再继读取当前的 Makefile 文件。"include" 使用的具体方式如下：

Include <filename>

filename是shell支持的文件名，比如.mk文件。

Include 就是把指向的.mk文件或是makefile文件搬到当前的makefile里面执行，包括执行的目录也是在当前目录。

注意："include" 关键字所在的行首可以包含一个或者是多个的空格（读取的时候空格会被自动的忽略），但是不能使用 Tab 开始，否则会把 "include" 当作式命令来处理。包含的多个文件之间要使用空格分隔开。使用 "include" 包含进来的 Makefile 文件中，如果存在函数或者是变量的引用，它们会在包含的 Makefile 中展开。

Include的文件怎么寻找？

如果使用 "include" 包含文件的时候，指定的文件路径（相对路径或者绝对路径）或者是为当前目录下没有这个文件，make 会根据文件名会在以下几个路径中去找，首先我们在执行 make 命令的时候可以加入选项 "-I" 或 "--include-dir" 后面添加上指定的路径，如果文件存在就会被使用，如果文件不存在将会在其他的几个路径中搜索："usr/gnu/include"、"usr/local/include" 和 "usr/include"。

Include 和 -include 区别：

Include <filename>:

make 在处理程序的时候，文件列表中的任意一个文件不存在的时候或者是没有规则去创建这个文件的时候，make 程序将会提示错误并保存退出。

-include <filename>:

当包含的文件不存在或者是没有规则去创建它的时候，make 将会继续执行程序，只有真正由于不能完成终极目标重建的时候我们的程序才会提示错误保存退出。

1. 目标和伪目标

伪目标不会去创建目标文件，它只会去执行目标下的命令，伪目标的存在可以帮我们找到对应的命令去执行。即使伪目标的规则命令创建了目标文件，或者当前目录下有和伪目标重名文件，规则依然会执行。

伪目标也可以有依赖，而且不管依赖是否更新，都会去执行规则。

而目标可以创建目标文件，也可以不创建目标文件。如果规则命令行里创建了目标文件，那么下一次执行make时，则不会去执行它下边的命令，除非它的依赖更新了；如果规则命令行里没有创建目标文件，那这个目标始终是最新的，每一次make，它都会去执行命令。

还有一种情况就是目标的依赖是伪目标，那么不管是目标的规则命令，还是伪目标的规则命令都会执行（一般情况不这样做）。

终极目标默认情况下是makefile文件里的第一条规则目标，如果第一条规则中的目标有很多个， 那么， 第一个目标会成为最终的目标（通常的做法是：all: target1 target2）。当然也可以通过make命令来指定终极目标，如：make clean。Makefile中以最终目标为收尾，也就是说只要最终目标和其依赖是最新的，它就不会再继续执行了。

**目标规则执行的条件**：

一般目标：目标不存在或者没有被创建或者目标的依赖被更新；前提是最终目标的依赖树中包含这个目标。

伪目标：不管目标和依赖是否最新，都会执行；前提是最终目标的依赖树中包含这个目标。

**使用伪目标的原因**：

1. 避免我们的 Makefile 中定义的只执行的命令的目标和工作目录下的实际文件出现名字冲突。即使伪目标和目录下的某个文件重名，make也会执行这个目标。
2. 提高效率。

**目标类型汇总**：

1. 强制目标

如果一个目标中没有命令或者是依赖，并且它的目标不是一个存在的文件名，在执行此规则时，目标总会被认为是最新的。就是说：这个规则一旦被执行，make 就认为它的目标已经被更新过。这样的目标在作为一个规则的依赖时，因为依赖总被认为更新过，因此作为依赖在的规则中定义的命令总会被执行。

clean:FORCE

rm $(OBJECTS)

FORCE:

这个例子中，目标 "FORCE" 符合上边的条件。它作为目标 "clean" 的依赖，在执行 make 的时候，总被认为更新过。因此 "clean" 所在的规则而在被执行其所定义的那个命令总会被执行。即使“clean”目标被创建或者已经存在，规则都会去执行。这样的一个目标通常我们将其命名为 "FORCE"。

1. 空目标文件

空目标文件只是用来记录上一次执行的此规则的命令的时间。在这样的规则中，命令部分都会使用 "touch" 在完成所有的命令之后来更新目标文件的时间戳，记录此规则命令的最后执行时间。

print:foot.c bar.c

lpr -p $?

touch print

以上“print”是被创建的空文件，用于标志，当它的依赖被更新时，就会触发该规则。

1. 多规则目标

Makefile 中，一个文件可以作为多个规则的目标。这种情况时，以这个文件为目标的规则的所有依赖文件将会被合并成此目标一个依赖文件列表，当其中的任何一个依赖文件比目标更新时，make 将会执行特定的命令来重建这个目标。

对于一个多规则的目标，重建这个目标的命令只能出现在一个规则中。如果多个规则同时给出重建此目标的命令，make 将使用最后一个规则中所定义的命令，同时提示错误信息。某些情况，需要对相同的目标使用不同的规则中所定义的命令，我们需要使用另一种方式——双冒号规则来实现。

objects=foo.o bar.o

foo.o:defs.h

bar.o:defs.h test.h

$(objects):config.h

1. 嵌套执行make

在一个大的工程文件中，不同的文件按照功能被划分到不同的模块中，也就说很多的源文件被放置在了不同的目录下。每个模块可能都会有自己的编译顺序和规则，如果在一个 Makefile 文件中描述所有模块的编译规则，就会很乱，执行时也会不方便，所以就需要在不同的模块中分别对它们的规则进行描述，也就是每一个模块都编写一个 Makefile 文件，这样不仅方便管理，而且可以迅速发现模块中的问题。这样我们只需要控制其他模块中的 Makefile 就可以实现总体的控制，这就是 make 的嵌套执行。

subsystem:

cd subdir && $(MAKE)

或者

subsystem：

$(MAKE) -C subdir

在 make 的嵌套执行中，我们需要了解一个变量 "CURDIR"，此变量代表 make 的工作目录。当使用 make 的选项 "-C" 的时候，命令就会进入指定的目录中，然后此变量就会被重新赋值。总之，如果在 Makefile 中没有对此变量进行显式的赋值操作，那么它就表示 make 的工作目录。我们也可以在 Makefile 中为这个变量赋一个新的值，当然重新赋值后这个变量将不再代表 make 的工作目录。

嵌套执行和include是两回事，include可以对变量和函数进行扩展，而嵌套执行不行。嵌套执行，工作目录改变了，而include则是将makefile文件搬过来用，工作目录始终是当前目录。

1. 变量的传递

变量的传递可以通过include关键字引入其他makefile文件的变量；还可以通过export关键字从上级makefile中传递给下一级makefile（注意不能从下级传给上级）。

**Export关键字使用**：

如果需要变量的传递，那么可以这样来使用：

export <variable>

如果不需要传递：

unexport <variable>

Makefile 中还有两个变量不管是不是使用关键字 "export" 声明，它们总会传递到下层的 Makefile 中。这两个变量分别是 SHELL 和 MAKEFLAGS，特别是 MAKEFLAGS 变量，包含了 make 的参数信息。如果执行总控 Makefile 时，make 命令带有参数或者在上层的 Makefile 中定义了这个变量，那么 MAKEFLAGS 变量的值将会是 make 命令传递的参数，并且会传递到下层的 Makefile 中，这是一个系统级别的环境变量。

make 命令中有几个参数选项并不传递，它们是:"-C"、"-f"、"-o"、"-h" 和 "-W"。如果我们不想传递 MAKEFLAGS 变量的值，在 Makefile 中可以这样来写：

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

1. makefile控制函数error,warning
2. $(error TEXT...)

函数说明如下：

**函数功能**：产生致命错误，并提示 "TEXT..." 信息给用户，并退出 make 的执行。需要说明的是："error" 函数是在函数展开时（函数被调用时）才提示信息并结束 make 进程。因此如果函数出现在命令中或者一个递归的变量定义（如：err=$(error TEXT...)）时，读取 Makefile 时不会出现错误。而只有包含 "error" 函数引用的命令被执行（在规则命令中引用），或者定义中引用此函数的递归变量被展开（如：err=$(error TEXT...) $(err)）时，才会提示知名信息 "TEXT..." 同时退出 make。

**返回值**：空

**函数说明**："error" 函数一般不出现在直接展开式的变量定义中，否则在 make 读取 Makefile 时将会提示致命错误。

ERROR1=1234

ifdef ERROR1

$(error error is $(ERROR1))

endif

或者

ERROR1=1234

err=$(error error is $(ERROR1))

ifdef ERROR1

$(err)

endif

还可以作为命令引用

ERROR1=1234

all:

$(error error is $(ERROR1))

1. $(warning TEXT...)

函数说明如下：

**函数功能**：函数 "warning" 类似于函数 "error" ，区别在于它不会导致致命错误（make不退出），而只是提示 "TEXT..."，make 的执行过程继续。

**返回值**：空

**函数说明**：用法和 "error" 类似，展开过程相同

1. 函数

1、abspath函数

用法：$(abspath \_names)

该函数主要用于将\_names中的各路径转换成绝对路径（包含目录和文件名），并将转换后的结果返回。

ROOT := $(abspath /usr/../lib)

all:

echo $(ROOT)

2、lastword函数

lastword 返回names中的最后一个字符串，names以空格进行分隔 $(lastword foo bar lose) 返回 lose。

1. call函数

$(call <expression>,<parm1>,<parm2>,...,<parmn>)

为<expression>的表达式或者命令集传递参数，而 <expression> 的返回值就是 call 函数的返回值。

makefile 中的$(1) $(2)用来表示call函数传过去的实参,

call函数原型: $(call variable,param,param,…)

call函数相当于c语言中函数调用,variable(函数名),param(实参1),param(实参2)...

由于调用的variable(函数名)函数的书写规范是不能带形参的,就用$(1),$(2)..表示第一个第二个形参。

define TestFun

$(1): gcc -o test $(2)

endef

$(call TestFun,test.o,test.c)