Makefile

Linux环境下的程序员如果不会使用GNU make来构建和管理自己的工程，应该不能算是一个合格的专业程序员，至少不能称得上是linux程序员。在Linux（unix）环境下使用GNU 的make工具能够比较容易的构建一个属于你自己的工程，整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。不过这需要我们投入一些时间去完成一个或者多个称之为Makefile文件的编写。此文件正是make正常工作的基础。

所要完成的Makefile文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建那些库文件以及如何创建这些库文件、如何最后产生我们想要得可执行文件。尽管看起来可能是很复杂的事情，但是为工程编写Makefile的好处是能够使用一行命令来完成“自动化编译”，一旦提供一个（通常对于一个工程来说会是多个）正确的Makefile。编译整个工程你所要做的唯一的一件事就是在shell 提示符下输入make命令。整个工程完全自动编译，极大提高了效率。

1 编译时机

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）：

1. 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；
2. 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；
3. 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。

2 Makefile语法

一个简单的Makefile描述规则组成：

```
TARGET... : PREREQUISITES...
    COMMAND
    ...
```

TARGET：规则的目标。通常是最后需要生成的文件名或者为了实现这个目的而必需的中间过程文件名。可以是.o文件、也可以是最后的可执行程序的文件名等。另外，目标也可以是一个make执行的动作的名称，如目标“clean”，我们称这样的目标是“伪目标”

PREREQUISITES：规则的依赖。生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。

COMMAND：规则的命令行。是规则所要执行的动作（任意的shell命令或者是可在shell下执行的程序）。它限定了make执行这条规则时所需要的动作。

一个规则可以有多个命令行，每一条命令占一行。注意：每一个命令行必须以[Tab]字符开始，[Tab]字符告诉make此行是一个命令行。make按照命令完成相应的动作。这也是书写Makefile中容易产生，而且比较隐蔽的错误。

3 makefile编写

- 最简单的makefile, makefile文件可以命名为`makefile`或`Makefile`

```
all:
    gcc main.c print.c -o helloworld
```

- 在目标上填上依赖的文件

```
all: main.c print.c
    gcc main.c print.c -o helloworld
```

- 将编译命令展开

```
all:print.o main.o
    gcc print.o main.o -o helloworld

print.o:print.c
    gcc -c print.c
main.o:main.c
    gcc -c main.c
```

4 变量

在这个规则中.o文件列表出现了两次；第一次：作为目标“all”的依赖文件列表出现，第二次：规则命令行中作为“gcc”的参数列表。这样做所带来的问题是：如果我们需要为目标“all”增加一个依赖文件，我们就需要在两个地方添加（依赖文件列表和规则的命令中）。添加时可能在“all”的依赖列表中加入了、但却忘记了给命令行中添加，或者相反。这就给后期的维护和修改带来了很多不方便，添加或修改时出现遗漏。

为了避免这个问题，在实际工作中大家都比较认同的方法是，使用一个变量，比如，叫objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示obj文件就行了。我们在makefile一开始就这样定义：

```
objs=print.o main.o
```

“objs”作为一个变量，它代表所有的.c文件的列表。在定义了此变量后，我们就可以在需要使用这些.o文件列表的地方使用“$(objects)”来表示它。

```
objs=print.o main.o
all:$(objs)
    gcc $(objs) -o helloworld

print.o:print.c
    gcc -c print.c
main.o:main.c
    gcc -c main.c
```

5清除工作目录过程文件

规则除了完成源代码编译之外，也可以完成其它任务。例如：前边提到的为了实现清除当前目录中编译过程中产生的临时文件（helloworld和那些.o文件）的规则：

```
clean :
    rm helloworld $(objs)
```

在实际应用时，我们把这个规则写成如下稍微复杂一些的样子。以防止出现始料未及的情况。

```
.PHONY : clean
clean :
    -rm hellworld $(objs)
```

这两个实现有两点不同：

1. 通过“.PHONY”特殊目标将“clean”目标声明为伪目标。避免当磁盘上存在一个名为“clean”文件时，在我们输入“make clean”时。clean规则没有依赖文件，无法进行时间对比，所以目标被认为是最新的而不去执行规则作定义的命令。

2. 在命令行之前使用“-”，意思是忽略命令“rm”的执行错误，继续执行后面的命令。

   这样的一个目标在Makefile中，不能将其作为终极目标（Makefile的第一个目标）。因为我们的初衷并不是当你在命令行上输入make以后执行删除动作。而是要创建或者更新程序。在我们上边的例子中。就是在输入make以后要需要对目标“all”进行创建或者重建。

7 @的使用

“@”不显示命令本身，只显示结果。如：“@echo helloworld”

- 不加@时

  ```
  echo:
      echo helloworld
  ```

```
where@ubuntu:~$ make echo
echo helloworld
helloworld
where@ubuntu:~$
```

- 加@时

```
echo:
    @echo helloworld
```

```
where@ubuntu:~$ make echo
helloworld
where@ubuntu:~$
```

6 自动推导

在使用make编译.c源文件时，编译.c源文件规则的命令可以不用明确给出。这是因为make本身存在一个默认的规则，能够自动完成对.c文件的编译并生成对应的.o文件。

它执行命令“cc -c”来编译.c源文件。在Makefile中我们只需要给出需要重建的目标文件名（一个.o文件），make会自动为这个.o文件寻找合适的依赖文件（对应的.c文件。对应是指：文件名除后缀外，其余都相同的两个文件），而且使用正确的命令来重建这个目标文件。

对于上边的例子，此默认规则就使用命令“cc -c main.c -o main.o”来创建文件“main.o”。对一个目标文件是“N.o”，倚赖文件是“N.c”的规则，完全可以省略其规则的命令行，而由make自身决定使用默认命令。此默认规则称为make的隐含规则。

通过`make -p > makehelp`，有一条默认执行命令。

```
#内建语法规则
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
#变量COMPILE.c

COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

%.o: %.c 代表通配所有的.c文件，并且指定其对应的目标.o,都执行内建命令

$(CC)：编译工具

$(CFLAGS)：编译时的参数，如-g,-Wall,-fPIC

$(CPPFLAGS)：预处理时的参数，如-I，-D, -shared

$(TARGET_ARCH)：代表cpu的构架，一般默认是x86.

这样，在书写Makefile时，我们就可以省略掉描述.c文件和.o依赖关系的规则。因此上边的例子就可以以更加简单的方式书写。

```
objs=print.o main.o
all:$(objs)
    gcc $(objs) -o helloworld
.PHONY : clean
clean :
    -rm hellworld $(objs)
```

我们也可以自己去写这条规则。$@表示目标，$^表示所有依赖，$<代表依赖列表中的第一个

```
objs=print.o main.o
all:$(objs)
    gcc $(objs) -o helloworld

%.o:%.c
    gcc -c $^ -o $@
.PHONY : clean
clean :
    -rm hellworld $(objs)
```

7 makefile函数

`wildcard`函数可以用来查找对应通配符的文件。如果想找所有的.c文件则可以这样写`$(wildcard *.c)`。找到.c文件后，我们希望能够自动生成.o文件列表。可以使用$(patsubst %.c, %.o, $(wildcard *.c))将所有的.c文件替换成.o。

```
objs=$(patsubst %.c, %.o, $(wildcard *.c))
all:$(objs)
    gcc $(objs) -o helloworld

%.o:%.c
    gcc -c $^ -o $@
.PHONY : clean
clean :
    -rm hellworld $(objs)
```

再把通用的目标用变量来替换

```
objs=$(patsubst %.c, %.o, $(wildcard *.c))
target=helloworld
$(target):$(objs)
    gcc $^ -o $@

%.o:%.c
    gcc -c $^ -o $@

.PHONY : clean
clean :
    -rm $(target) $(objs)
```

8 编译变量

```
objs=$(patsubst %.c, %.o, $(wildcard *.c))
target=helloworld
CC=gcc
CPPFLAGS=-Iinclude
CFLAGS=-g -Wall
LDFLAGS=-lmylib

$(target):$(objs)
    $(CC) $^ $(LDFLAGS) -o $@

%.o:%.c
    $(cc) -c $^ $(CPPFLAGS) $(CFLAGS)-o $@

.PHONY : clean
clean :
    -rm $(target) $(objs)
```

9 附加目标

distclean一般是用来彻底清除生成的过程文件和生成的配置文件。这个目标也是要自己写的。

比如命令:

```
distclean:
    rm -rf /usr/bin/app
    rm -rf /etc/app.config
install:
    cp  app /usr/bin/app
```

10 makefile调用makefile

make -C [目录]，指定进入到某个目录中，并且执行目录中的makefile

```
objs=$(patsubst %.c, %.o, $(wildcard *.c))
target=helloworld
CC=gcc
CPPFLAGS=-I. -DDEB
CFLAGS=-g -Wall
LDFLAGS=-lmy

$(target):$(objs)
	gcc $^ $(LDFLAGS) -o $@
	make -C ./mylib
%.o:%.c
	gcc -c  $^ $(CPPFLAGS) $(CFLAGS) -o $@

.PHONY : clean
clean :
	-rm -rf $(target) $(objs)
	-make clean -C ./mylib
```

11 获取当前路径

`$(shell [cmd])`用来执行shell命令，并且返回该命令的回显，所以可以通过`$(shell pwd)`获取当前的makefile的路径。

12 缺省makefile

当没有编写makefile文件的时候，直接执行`make`指令加某个目标，如果在当前目录中存在目标默认的依赖.c文件，则自动执行`gcc [目标].c -o [目标]`。

例如：当前路径有一个文件test.c，编写完代码后，直接执行

```
make test
```

如果编译通过，将在当前路径上生成一个test可执行程序。