Makefile 入门实验

假设我们有一个程序由 5 个文件组成，源代码如下：

```
/*main.c*/
#include "mytool1.h"
#include "mytool2.h"
int main()
{
	mytool1_print("hello mytool1!");
	mytool2_print("hello mytool2!");
	return 0;
}
```

```
/*mytool1.c*/
#include "mytool1.h"
#include <stdio.h>
void mytool1_print(char *print_str)
{
printf("This is mytool1 print : %s ",print_str);
}
/*
```

```
/*mytool1.h*/
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif
```

```
/*mytool2.c*/
#include "mytool2.h"
#include <stdio.h>
void mytool2_print(char *print_str)
{
printf("This is mytool2 print : %s ",print_str);
}
```

```
/*mytool2.h*/
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif
```

首先了解一下make 和Makefile。GNU make 是一个工程管理器，它可以管理较多的文
件。使用make 的最大好处就是实现了“自动化编译”。如果有一个上百个文件的代码构成
的项目，其中一个或者几个文件进行了修改，make 就能够自动识别更新了的文件代码，不
需要输入冗长的命令行就可以完成最后的编译工作。make 执行时，自动寻找Makefile
（makefile）文件，然后执行编译工作。所以我们需要编写Makefile 文件，这样可以提高实
际项目的工作效率。
在一个Makefile 中通常包含下面内容：
1、需要由make 工具创建的目标体（target），通常是目标文件或可执行文件。
2、要创建的目标体所依赖的文件（dependency_file）。
3、创建每个目标体时需要运行的命令（command）。
格式如下：
target：dependency_files
<TAB>command
target：规则的目标。通常是程序中间或者最后需要生成的文件名，可以是.o 文件、也可以
是最后的可执行程序的文件名。另外，目标也可以是一个make 执行的动作的名称，如目标
“clean”，这样的目标称为“伪目标”。
dependency_files：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make 程序所有执行的动作（任意的shell 命令或者可在shell
下执行的程序）。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。命令就是在任何一个目标的
依赖文件发生变化后重建目标的动作描述。一个目标可以没有依赖而只有动作（指定的命
令）。比如Makefile 中的目标“clean”，此目标没有依赖，只有命令。它所指定的命令用来
删除make 过程产生的中间文件（清理工作）。
在Makefile 中“规则”就是描述在什么情况下、如何重建规则的目标文件，通常规则中包
括了目标的依赖关系（目标的依赖文件）和重建目标的命令。make 执行重建目标的命令，
来创建或者重建规则的目标（此目标文件也可以是触发这个规则的上一个规则中

的依赖文
件）。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 中可以包含除规则以外的部分。一个最简单的Makefile 可能只包含规则描述。规则
在有些Makefile 中可能看起来非常复杂，但是无论规则的书写是多么的复杂，它都符合规
则的基本格式。

常规法写第一个Makefile

```
main:main.o mytool1.o mytool2.o
    gcc -o main main.o mytool1.o mytool2.o
main.o:main.c mytool1.h mytool2.h
    gcc -c main.c
mytool1.o:mytool1.c mytool1.h
    gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
    gcc -c mytool2.c
clean:
    rm -f *.o main
```

在shell 提示符下输入make，执行显示:

```
gcc -c main.c
gcc -c mytool1.c
gcc -c mytool2.c
gcc -o main main.o mytool1.o mytool2.o
```

执行结果如下：

```
[armlinux@lqm makefile-easy]$ ./main
This is mytool1 print : hello mytool1!
This is mytool2 print : hello mytool2!
```

这只是最为初级的Makefile，现在来对这个Makefile 进行改进。

改进一：使用变量

一般在书写Makefile 时，各部分变量引用的格式如下:

1. make 变量（Makefile 中定义的或者是make 的环境变量）的引用使用“$(VAR)”格式，无论“VAR”是单字符变量名还是多字符变量名。

2. 出现在规则命令行中shell 变量（一般为执行命令过程中的临时变量,它不属于Makefile 变量,而是一个shell 变量)引用使用shell 的“$tmp”格式。

3. 对出现在命令行中的make 变量同样使用“$(CMDVAR)” 格式来引用。

```
OBJ=main.o mytool1.o mytool2.o
main:$(OBJ)
    gcc -o main $(OBJ)
main.o:main.c mytool1.h mytool2.h
    gcc -c main.c
mytool1.o:mytool1.c mytool1.h
    gcc -c mytool1.c
mytool2.o:mytool2.c mytool2.h
    gcc -c mytool2.c
clean:
    rm -f main $(OBJ)
```

改进二：使用自动推导

让make 自动推导，只要make 看到一个.o 文件，它就会自动的把对应的.c 文件加到依赖文件中，并且gcc －c .c 也会被推导出来，所以Makefile 就简化了。

```
CC = gcc
OBJ = main.o mytool1.o mytool2.o
main: $(OBJ)
    $(CC) -o main $(OBJ)
main.o: mytool1.h mytool2.h
mytool1.o: mytool1.h
mytool2.o: mytool2.h
.PHONY: clean
clean:
    rm -f main $(OBJ)
```

改进三：自动变量（$^ $< $@）的应用

Makefile 有三个非常有用的变量，分别是$@、$^、$<。代表的意义分别是:

$@--目标文件，

$^--所有的依赖文件，

$<--第一个依赖文件。

```
CC = gcc

OBJ = main.o mytool1.o mytool2.o

main: $(OBJ)

$(CC) -o $@ $^

main.o: main.c mytool1.h mytool2.h

$(CC) -c $<

mytool1.o: mytool1.c mytool1.h

$(CC) -c $<

mytool2.o: mytool2.c mytool2.h

$(CC) -c $<

.PHONY: clean

clean:

rm -f main $(OBJ)
```

这些是最为初级的知识，现在至少可以减少编译时的工作量。

细节方面的东西还需要在以后的工作和学习中不断的总结，不断的深化理解。可以参考 GNU Make 手册，这里讲解的比较全面。