Makefile 学习

作为 Linux 下的 C/C++开发者，没接触过 makefile 一定说不过去，通常构建大型的 C/C++项目都离不开 makefile，也许你使用的是 cmake 或者其他类似的工具，但它们的本质都是类似的。

makefie 到底是什么？

在 Linux 下，对于下面这个简单的程序。

```
//main.c
#include <stdio.h>
#include <math.h>
int main()
{
    int a = 10;
    int b = 4;
    int c = pow(a,b);
    printf("10^4 = %d",c);
    return 0;
}
```

我们通常使用 gcc 就可以编译得到想要的程序了：

```
$ gcc -o main main.c -lm
```

对于单个文件的简单程序，一条命令就可以直接搞定了（编译+连接），但是如果是一个复杂的工程，可能有成千上万个文件，然后需要链接大量的动态或静态库。试想一下，你还想一条一条命令执行吗？**懒惰的基因是刻在程序员骨子里的。**因此你可能会想，那我写个脚本好了。嗯，听起来好多了。

文件多就多，你告诉我要编译哪里的文件，我遍历一下就好了，你再告诉我要链接哪些库，我一一帮你链接上就好了。

然而到这里又会想，既然编译链接都是这么类似的过程，能不能给它们写一些通用的规则，搞得这么复杂干嘛？然后按照规则去执行就好了。

而 makefile 就是这样的一个规则文件，make 是规则的解释执行者。可以类比 shell 脚本和 bash 解释程序的关系。

所以，makefile 并不仅仅用于编译链接，只不过它非常适合用于编译链接。

makefile 什么样？

它最重要的规则语法如下：

```
<target> : <prerequisites>
[tab]  <commands>
```

咋一看，就这么个玩意？但是什么意思？

- target 要生成的目标文件名称
- 要依赖的文件
- [tab] 对，就是 tab 键，初学者很容易忽略这个问题，请用 tab
- 要执行的指令

关键内容就这些，但是要细讲会有很多内容，仅举个简单的例子。假设要将前面的 main.c 复制名为 pow.c 的文件。
那么我们可以得到：

- target: pow.c 目标名称
- prerequisites：main.c，即得到 pow.c 需要有 main.c
- commands：cp main.c pow.c

因此我们得到我们的 makefile 文件内容如下：

```
pow.c:main.c
    cp main.c pow.c
clean:
    rm pow.c
```

假设当前目录下没有 main.c 文件，然后在当前目录下执行：

```
$ make pow.c
make: *** No rule to make target `main.c', needed by `pow.c'.  Stop.
```

我们发现会报错，因为你要依赖的文件找不到，而且也没有其他规则能够生成它。
现在把 main.c 放在当前目录下后继续执行：

```
$ make
cp main.c pow.c
```

看见没有，执行完 make 命令之后，我们的 pow.c 文件终于有了。
而执行下面的命令后：

```
$ make clean
rm pow.c
```

你就会发现 pow.c 被删除了。
如果当前目录有 clean 文件会发生什么？

```
$ make clean
make: `clean' is up to date.
```

至于原因，后面会讲到。
这里注意，如果你的 makefile 文件的文件名不是 makefile，那么就需要指定具体名字，例如假设前面的文件名为 test.txt：

```
$ make -f test.txt
```

以上例子介绍了 makefile 使用的基本流程，生成目标，清除目标。然而实际上这里面的门道还有很多，例如伪目标，自动推导，隐晦规则，变量定义。本文作为认识性的文章暂时不具体介绍。
总结来说就是，给规则，按照规则生成目标。

makefile 做了什么？

网上有很多教程介绍如何编写 makefile 的，很多也非常不错。不过本文换个角度来说。

既然我们要学 makefile，那么就需要知道构建 C/C++项目的时候，它应该做什么？然后再去学习如何编写 makefile。

实际上它主要做的事情也很清楚，那就是编译和链接。

- 将源代码文件编译成可重定位目标文件.o
- 设置编译器选项，例如是否开启优化，传递宏，打开警告等
- 链接，将静态库或动态库与目标文件链接

所以问题就变成了，如何利用 makefile 的语法规则快速的将成千上万的.c 编译成.o，并且正确链接到需要的库。

而如果用 makefile 应该怎么写才能得到我们的程序呢？为了帮助说明，我们把前面的编译命令拆分为两条：

```
$ gcc -g -Wall -c main.c -o main.o
$ gcc -o main main.o -lm
```

设置编译器

由于我们使用的是 gcc 编译器（套件），因此可以像下面这样写

```
CC=gcc
```

为了扩展性考虑，常常将编译器定义为某个变量，后面使用的时候就会方便很多。

设置编译选项

比如我们要设置-g 选项用来调试，设置-Wall 选项来输出更多警告信息。

```
CFLAGS=-g -Wall
```

设置链接库

我们这里只用到了 libm.so 库

```
LIBS=-lm
```

编译

我们的目标文件是 main.o 依赖 main.c，该规则应该是这样的：

```
OBJ=main.o
$(OBJ):main.c
    $(CC) $(CFLAGS) -c main.c -o $(OBJ)
```

这样就得到了我们的目标文件。

链接

接下来就需要将目标文件和库文件链接在一起了。

```
TARGET=main
$(target):main.o
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LIBS)
```

而为了使用 make clean，即通常用于清除这些中间文件，因此需要加一个伪目标 clean：

```
.PHONY:clean
clean:
    rm $(OBJ) $(TARGET)
```

伪目标的意思是，它不是一个真正的要生成的目标文件，.PHONY:clean 说明了 clean 是一个伪目标。在这种情况下，即使当前目录下有 clean 文件，它也仍然会执行后面的指令。

否则如果当前目录下有 clean 文件，将不会执行 rm 动作，而认为目标文件已经是最新的了。

完整内容：

```
CC=gcc
CFLAGS=-g -Wall
LIBS=-lm
OBJ=main.o
$(OBJ):main.c
    $(CC) $(CFLAGS) -c main.c -o $(OBJ)
TARGET=main
$(TARGET):main.o
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LIBS)
.PHONY:clean
clean:
    rm $(OBJ) $(TARGET)
```

可以看到，makefile 文件中有三个目标，分别是 main.o，main 和 clean，其中 clean 是一个伪目标。

注意，由于第一个目标是 main.o，因此你单单执行 make 的时候，它只是会生成 main.o 而已，如果你再执行一次会发现它提示你说 main.o 已经是最新的了：

```
$ make
gcc -g -Wall -c main.c -o main.o
$ make
make: `main.o' is up to date.
```

为了得到 main，我们执行：

```
$ make main
gcc -g -Wall -c main.c -o main.o
gcc -g -Wall -o main main.o -lm
$ ls
main   main.c   main.o   makefile
```

当然你也可以调整目标顺序。这里的目标文件 main 依赖的是 main.o，它开始会去找 main.o，发现这个文件也没有，就会看是不是有规则会生成 main.o，欸，你还别说，真有。main.o 又依赖 main.c，也有，最终按照规则就会先生成 main.o，然后生成 mian。

如果要清除这些目标文件，那么可以执行 make clean：

```
$ make clean
rm main.o main
$ ls
main.c   makefile
```

makefile 是什么东西

它是一个规则文件，里面按照某种语法写好了，然后使用 make 来解释执行，就像 shell 脚本要用 bash 解释运行一样。通常会用 makefile 来构建 C/C++项目。

构建 C/C++项目的 makefile 做了什么

makefile 主要做下面的事情（以 C 程序为例）

- 用变量保存各种设置项，例如编译选项，编译器，宏，包含的头文件等
- 把.c 编译成.o
- 把.o 与库进行链接
- 清除生成的文件
- 安装程序

其中最关键的事情就是编译链接，即想办法把.c 变成.o（可重定位目标文件）;.o+.so（动态库）+.a（静态库）变成可执行文件。

对于文本提到的例子，看起来实在有些笨拙，一条指令搞定，却要写这么多行的 makefile，但是它却指出了通常编写 makefile 的基本思路。

对于一个复杂的项目而言，makefile 还有很多东西可介绍，例如如何设置变量，如何交叉编译，如何多个目录编译，如何自动推导，如何分支选择等等。这些都是后话了。