附录A：Makefile简易教程

A.1 Makefile简介

在软件开发中，make通常被视为一种软件构建工具。该工具主要经由读取一种名为`makefile`或`Makefile`的文件来实现软件的自动化建构。它会通过一种被人们称之为“目标（target）”概念来检查相关文件之间的依赖关系，这种依赖关系的检查系统非常简单，主要通过对比文件的修改时间来实现。在大多数情况下，我们主要用它来编译源代码，生成结果代码，然后把结果代码连接起来生成可执行文件或者库文件。

A.1.1 优点与缺点

与大多数古老的UNIX工具一样，make也分别有着人数众多的拥护者和反对者。它在适应现代大型软件项目方面有着许许多多的问题。但是，依然有很多人坚定地认为（包括我）它能应付绝大多数常见的情况，而且使用非常的简单，功能强大，表达清楚。无论如何，make如今仍然被用来编译很多完整的操作系统，而且它的那些“更为现代”的替代品们在基本操作上与它没有太大差别。

当然，随着现代的集成开发环境（IDE）的诞生，特别是非UNIX的平台上，很多程序员不再手动管理源代码之间的依赖关系，甚至不用去管哪些文件是这个项目的一部分，而是把这些任务交给了他们的开发环境去做。类似的，很多现代的编程语言有自己专属的、能高效配置依赖关系的方法（譬如Ant）。

A.1.2 主要版本

make程序经历过各方多次的改写与重写，各方都依据自己的需要做了一些特定的改良。目前市面上主要流行有以下几种版本：

- **GNU make**：GNU make对make的标准功能（通过clean-room工程）进行了重新改写，并加入作者自认为值得加入的新功能，常和GNU编译系统一起被使用，是大多数GNU Linux默认安装的工具。
- **BSD make**：该版本是从Adam de Boor制作的版本上发展起来的。它在编译目标的时有并发计算的能力。主要应用于FreeBSD，NetBSD和OpenBSD这些系统。
- **Microsoft nmake**：该版本主要用于微软的Windows系统中，需要注意的是，微软的nmake与UNIX项目中的nmake是两种不同的东西，千万不要混淆。

A.2 从一个简单的例子入手

我们可以用K&R C [1] 中4.5那个例子来做个说明。在这个例子中，我们会看到一份主程序代码(`main.c`)、三份函数代码(`getop.c`、`stack.c`、`getch.c`)以及一个头文件(`calc.h`)。通常情况下，我们需要这样编译它：

```
gcc -o calc main.c getch.c getop.c stack.c
```

如果没有makefile，在开发+调试程序的过程中，我们就需要不断地重复输入上面这条编译命令，要不就是通过终端的历史功能不停地按上下键来寻找最近执行过的命令。这样做两个缺陷：

1. 一旦终端历史记录被丢失，我们就不得不从头开始；
2. 任何时候只要我们修改了其中一个文件，上述编译命令就会重新编译所有的文件，当文件足够多时这样的编译会非常耗时。

那么Makefile又能做什么呢？我们先来看一个最简单的makefile文件：

```
    calc: main.c getch.c getop.c stack.c
        gcc -o calc main.c getch.c getop.c stack.c
```

现在，你看到的就是一个最基本的Makefile语句，它主要分成了三个部分，第一行冒号之前的`calc`，我们称之为目标（target），被认为是这条语句所要处理的对象，具体到这里就是我们所要编译的这个程序`calc`。冒号后面的部分（`main.c`、`getch.c`、`getop.c`、`stack.c`），我们称之为依赖关系表，也就是编译`calc`所需要的文件，这些文件只要有一个发生了变化，就会触发该语句的第三部分，我们称其为命令部分，相信你也看得出这就是一条编译命令。现在我们只要将上面这两行语句写入一个名为`Makefile`或`makefile`的文件，然后在终端中输入`make`命令，就会看到它按照我们的设定去编译程序了。

> 请注意，在第二行的`gcc`命令之前必须要有一个tab缩进。语法规定Makefile中的任何命令之前都必须要有一个tab缩进，否则make就会报错。

接下来，让我们来解决一下效率方面的问题，先初步修改一下上面的代码：

```
    cc = gcc
    prom = calc
    src = main.c getch.c getop.c stack.c

    $(prom): $(src)
        $(cc) -o $(prom) $(src)
```

如你所见，我们在上述代码中定义了三个常量`cc`、`prom`以及`src`。它们分别告诉了make我们要使用的编译器、要编译的目标以及源文件。这样一来，今后我们要修改这三者中的任何一项，只需要修改常量的定义即可，而不用再去管后面的代码部分了。

> 请注意，很多教程将这里的`cc`、`prom`和`src`称之为变量，个人认为这是不妥当的，因为它们在整个文件的执行过程中并不是可更改的，作用也仅仅是字符串替换而已，非常类似于C语言中的宏定义。或者说，事实上它就是一个宏。

但我们现在依然还是没能解决当我们只修改一个文件时就要全部重新编译的问题。而且如果我们修改的是`calc.h`文件，make就无法察觉到变化了（所以有必要为头文件专门设置一个常量，并将其加入到依赖关系表中）。下面，我们来想一想如何解决这个问题。考虑到在标准的编译过程中，源文件往往是先被编译成目标文件，然后再由目标文件连接成可执行文件的。我们可以利用这一点来调整一下这些文件之间的依赖关系：

```
    cc = gcc
    prom = calc
    deps = calc.h
    obj = main.o getch.o getop.o stack.o

    $(prom): $(obj)
        $(cc) -o $(prom) $(obj)

    main.o: main.c $(deps)
        $(cc) -c main.c

    getch.o: getch.c $(deps)
        $(cc) -c getch.c

    getop.o: getop.c $(deps)
        $(cc) -c getop.c
```

```
17
18      stack.o: stack.c $(deps)
19          $(cc) -c stack.c
```

这样一来，上面的问题显然是解决了，但同时我们又让代码变得非常啰嗦，啰嗦往往伴随着低效率，是不祥之兆。经过再度观察，我们发现所有`.c`都会被编译成相同名称的`.o`文件。我们可以根据该特点再对其做进一步地简化：

```
1       cc = gcc
2       prom = calc
3       deps = calc.h
4       obj = main.o getch.o getop.o stack.o
5
6       $(prom): $(obj)
7           $(cc) -o $(prom) $(obj)
8
9       %.o: %.c $(deps)
10          $(cc) -c $< -o $@
```

在这里，我们用到了几个特殊的宏。首先是`%.o:%.c`，这是一个模式规则，表示所有的`.o`目标都依赖于与它同名的`.c`文件（当然还有`deps`中列出的头文件）。再来就是命令部分的`$<`和`$@`，其中`$<`代表的是依赖关系表中的第一项（如果我们想引用的是整个关系表，那么就应该使用`$^`），具体到我们这里就是`%.c`。而`$@`代表的是当前语句的目标，即`%.o`。这样一来，`make`命令就会自动将所有的`.c`源文件编译成同名的`.o`文件。不用我们一项一项去指定了。整个代码自然简洁了许多。

到目前为止，我们已经有了一个不错的makefile文件了，至少用它来维护这个小型工程是没有什么问题了。当然，如果要进一步增加上面这个项目的可扩展性，我们就会需要用到一些Makefile语法中的**伪目标**和**函数规则**了。例如，如果我们想增加自动清理编译结果的功能就可以为其定义一个带伪目标的规则；

```
1       cc = gcc
2       prom = calc
3       deps = calc.h
4       obj = main.o getch.o getop.o stack.o
5
6       $(prom): $(obj)
7           $(cc) -o $(prom) $(obj)
8
9       %.o: %.c $(deps)
10          $(cc) -c $< -o $@
11
12      clean:
13          rm -rf $(obj) $(prom)
```

有了上面最后两行代码，当我们在终端中执行`make clean`命令时，它就会去删除该工程生成的所有编译文件。

另外，如果我们需要往工程中添加一个`.c`或`.h`，可能同时就要再手动为obj常量再添加第一个`.o`文件，如果这列表很长，代码会非常难看，为此，我们需要用到Makefile中的函数，这里我们演示两个：

```
1       cc = gcc
2       prom = calc
3       deps = $(shell find ./ -name "*.h")
4       src = $(shell find ./ -name "*.c")
```

```
5     obj = $(src:%.c=%.o)
6
7     $(prom): $(obj)
8         $(cc) -o $(prom) $(obj)
9
10    %.o: %.c $(deps)
11        $(cc) -c $< -o $@
12
13    clean:
14        rm -rf $(obj) $(prom)
```

其中，shell函数主要用于执行shell命令，具体到这里就是找出当前目录下所有的`.c`和`.h`文件。而`$(src:%.c=%.o)`则是一个字符替换函数，它会将`src`所有的`.c`字串替换成`.o`，实际上就等于列出了所有`.c`文件要编译的结果。有了这两个设定，无论我们今后在该工程加入多少`.c`和`.h`文件，`Makefile`文件都能自动将其纳入到工程中来。

A.3 请学习更多资料

到这里，我们就基本上将日常会用到的Makefile写法介绍了一遍。如果你想了解更多关于makefile和make的知识，请参考GNU Make Manual。[2]

---

1. 注释：即C语言的经典教程*The C Programming Language*，业界通常以其两位作者的首字母为缩写，将其简称为K&R C。↩

2. 注释：下载链接：http://www.cs.utexas.edu/~cannata/cs345/GNU%20Make%20Manual.pdf↩