第六章 Makefile写法

第六章：Makefile写法

目标:

本章使学员掌握Linux操作系统下Makefile的写法：

- ☑ 了解Makefile的语法及规则
- ☑ 掌握利用Makefile文件进行编译

学时：3 学时

教学方法：讲授ppt＋上机操作+实例演示

6.1 make工具介绍

- 一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。

- make需要得到两方面的信息:

    1. 是关于可执行文件和各程序模块间的相互关系

    2. 二是文件的修改日期

6.2 Makefile的概念

Neuedu

Makefile产生的背景

- 一个软件项目通常包含多个源码文件,每个源代码的编译和可执行文件的链接都要书写大量的命令.
  - 如Linux 下要大量调用gcc来处理
- 如果用IDE开发环境,这编译和链接一般由IDE自动完成.但绝大部分Linux和开源项目并不使用IDE,而是使用gcc之类命令行工具来编译
  - Linux内核.
- 在一项目里, 代码通常都有引用的关系.因此需要指定谁先编译,谁后编译,甚至是更复杂的功能操作.
- Makefile就是为解决上述一系列问题而创造的.可以把Makefile 理解成是一种由make程序进行解释的一种特殊脚本.
- Linux 几乎所有项目都是通过Makefile方式编译的,如MySQL,Apache和操作系统本身,因此Linux下开发必须掌握Makefile的编写和使用

6.2 Makefile的概念

Neuedu

Makefile与Shell脚本的异同

- 相同点:
  - 都是文本文件格式的脚本.
  - 都可以执行Shell命令
  - 都可以定义变量,和条件控制语句.(使用格式上有差别)
- 不同点:
  - 解释器不同,Shell脚本是由对应Shell程序解释.而Makefile是由make程序解释
  - 格式不一样,Shell脚本以命令行为基本单位,而Makefile以规则为基本单位
  - Shell脚本只要有执行权限即可直接执行,Makefile必须要用make来显式调用才行,本身不需执行权限

6.2 Makefile的概念

Neuedu

Makefile相对Shell脚本的优点

- 在开发领域,Makefile还是有相当优势
  - Makefile具有自动推导,判断源码依赖关系的功能
  - Makefile有可以使用隐含规则来简化makefile的编写
    但这样会使makefile可读性下降

6.3 make的调用

- 最常用的make调用形式.就是直接执行make
  - 它会自动查找当前目录下的名称为Makefile或makefile的文件,并自动从第一个target开始执行
- 象其它GNU工具一样,make有一些命令参数,以便应用在一些特殊场所
- 如果makefile脚本名称不是缺省名称,则需要用-f参数来通知make
  - make –f hello.mk
  - 表示执行名称叫hello.mk makefile脚本
- 如果需要make不去查找当前目录,而是查找另外一个目录下的makefile,则使用-C 参数
  - make -C /home/hxy
  - 表示去找查找 /home/hxy下的makefile

6.3 make的调用

Neuedu

只编译特定部分

- 一个项目可能不同版本或不同部分需要编译,在makefile中以target表示不同的编译部分
- make可以通用在命令行直接写target名称,用于一个或多个target进行编译.
  - make install
  - make clean
  - make target1 target2

6.4 Makefile的格式

Makefile由一组依赖关系和规则构成.
每个依赖关系由一个目标和他所依赖的源文件组成.
目标就是即将要创建的文件.
规则描述了怎样从被依赖的文件创建出目标文件.

6.4.1 规则(rule)概念

- 一个Makefile可以看作是一系列规则的组合,一个规则也称为一个目标(target)
- 规则的格式
  - 目标名称是需要创建的结果的一个称呼.可以取任意标识名.
  - 依赖对象,表示创建这个目标之前,必须预先创建的其它目标,这里的对象可以是另一个规则的名称,也可是基本的文件名称
  - 命令列表表示为了创建这一个目标,需要执行哪些Shell命令.可以是一行或多行Shell命令.**每一行命令行的行首必须是一个跳格字符(即tab),**
    - 注意行首空格是无效,否则执行makefile报错
    - 如果命令行过长,可用\分行,分行后的新行,无需使用tab打头
- 整个规则的可做如下解读."为了创建这个目标,必须先创建依赖对象(或是依赖的对象必须存在),然后再调用命令行列表进行创建"

目标名称：[依赖对象]
<tab>命令列表

6.4.1 规则(rule)概念

- 在Makefile里,把源代码编译成目标代码(.o文件)一般是一个规则
- 把所有中间文件(.o文件)链接在一起也是一个规则

```
#要想生成hello.o目标,必须先有hello.c,
#然后调用命令行gcc编译生成hello.o
#依赖对象hello.c在这里可以省略
hello.o : hello.c
        gcc –c hello.c –o hello.o
```

```
#要想生成执行程序hello,必须先执行规则hello.o,
#然后调用命令行gcc链接生成hello
hello : hello.o
        gcc hello.o –o hello
```

6.4.1 规则(rule)概念

- 跟Shell不同,在Makefile里规则的前后顺序不太重要.
- 实际的调用顺序取决目标之间的依赖关系.
- 因此make 采用逆推的方式来判断和执行目标

6.4.2 伪目标

- 一般的目标最终结果都是生成一个文件,但有一些目标可以不生成结果文件.只是为了调用命令或依赖对象.这称为伪目标(Phony target)
- make内置一些常用的伪目标.
  - all 编译所有目标.
  - clean 清除项目生成的中间文件和最终成文件,如何清除需要开发者自行编写.
  - install 项目如何安装, .具体动作要开发者自行编写
  - uninstall 项目如何卸载.具体动作要开发者自行编写

6.4.3 Makefile基本结构

Neuedu

- makefile中一般包含如下内容
  - 需要由 make 工具创建的项目，通常是目标文件和可执行文件。通常使用“目标（target）”一词来表示要创建的项目。
  - 要创建的目标依赖于哪些文件。
  - 创建每个目标时需要运行的命令,**每个命令之前必须有tab打头**
  - 通常都包含一些固定的伪目标,如all,install,clean用作缺省编译,安装和清除文件
  - #打头表示注释行

Makefile实例

例如，对于之前的hello程序，其Makefile可书写如下：

```
1 #Makefile for exp1
2 hello: hello.c
3         gcc -o hello hello.c
```

目标文件为hello，其依赖于hello.c，其生成指令为gcc -o hello hello.c。在终端下执行make，会自动生成可执行文件hello。

```
root@neusoft-vm:~/exp1# make
gcc -o hello hello.c
root@neusoft-vm:~/exp1# ls
hello  hello.c  Makefile
root@neusoft-vm:~/exp1# ./hello
Hello World!
```

Makefile实例

```c
//main.c
#include "main.h"
int main()
{
    add();
    del();
    modify();
    return 0;
}
```

```c
//add.c
#include <stdio.h>
void add()
{
    printf("add\n");
}
```

```c
//del.c
#include <stdio.h>
void del()
{
    printf("del\n");
}
```

```c
//main.h
void add();
void del();
void modify();
```

```c
//modify.c
#include <stdio.h>
void modify()
{
    printf("modify\n");
}
```

- 其 Makefile 可书写如下：

```
1 #Makefile for exp2
2 main: main.o add.o del.o modify.o
3         gcc -o main main.o add.o del.o modify.o
4 main.o: main.c
5         gcc -c main.c
6 add.o: add.c
7         gcc -c add.c
8 del.o: del.c
9         gcc -c del.c
10 modify.o: modify.c
11         gcc -c modify.c
```

6.5 Makefile扩展用法

- 在复杂项目里,为了简化makefile的书写.往往会采用扩展写法.这样大大方便开发者的编写
- 这些方法包括
  - 变量
  - 隐含规则
  - Makefile的引用
  - Makefile的函数

6.5.1 Makefile变量

- 如果你要以相同的编译选项同时编译十几个C源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义，可避免这种乏味的工作：

```
# Define macros for name of compiler
CC = gcc

# Define a macr o for the CC flags
CCFLAGS = -D_DEBUG -g -m486

# A rule for building a object file
test.o: test.c test.h
    $(CC) -c $(CCFLAGS) test.c
```

- 在上面的例子中，CC和CCFLAGS 就是make 的变量。GNU make通常称之为变量，而其他UNIX的make工具称之为宏，实际是同一个东西。在makefile中引用变量的值时，只需变量名之前添加$符号，如上面的$(CC)和$(CCFLAGS)。

6.5.1 makefile变量

- Makefile变量定义跟Shell变量定义刚好相反!
  - Shell变量定义时,=两边不能有空格.
  - Makefile变量定义,=两边一定要有空格
- Makefile的变量定义要独立于规则之外,
  - 是有定义先后顺序的要求
  - 一般要放在所有规则前面进行定义

6.5.2 GNU make的主要预定义变量

Neuedu

- GNU make 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。
- 除这些变量外，GNU make 还将所有的环境变量作为自己的预定义变量。

预定义变量	含义
$<	第一个依赖文件的名称。
$@	目标的完整名称。
$^	所有的依赖文件，以空格分开，不包含重复的依赖文件。

例如，如果目标名称为 mytarget.so，则 $@ 为 mytarget.so

6.5.2 GNU make的主要预定义变量

- AR　　归档维护程序的名称，默认值为 ar。
- ARFLAGS　　归档维护程序的选项。
- AS　　汇编程序的名称，默认值为 as。
- ASFLAGS　　汇编程序的选项。
- CC　　C 编译器的名称，默认值为 cc。
- CCFLAGS　　C 编译器的选项。

6.5.3 自动化变量使用

```
1 #Makefile for exp2
2 main: main.o add.o del.o modify.o
3         gcc -o main main.o add.o del.o modify.o
4 main.o: main.c
5         gcc -c main.c
6 add.o: add.c
7         gcc -c add.c
8 del.o: del.c
9         gcc -c del.c
10 modify.o: modify.c
11         gcc -c modify.c
```

简化为

```
1 #Makefile2 for exp2
2 main: main.o add.o del.o modify.o
3         gcc -o $@ $^
4 main.o: main.c
5         gcc -c $<
6 add.o: add.c
7         gcc -c $<
8 del.o: del.c
9         gcc -c $<
10 modify.o: modify.c
11         gcc -c $<
```

进一步简化

```
1 #Makefile3 for exp2
2 main: main.o add.o del.o modify.o
3         gcc -o $@ $^
4 .c.o:
5         gcc -c $<
```

6.6 隐含规则

- GNU make 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。
- GNU make 支持两种类型的隐含规则：

1：后缀规则（Suffix Rule）。
后缀规则定义了将一个具有某个后缀的文件（例如.c文件）转换为具有另外一种后缀的文件（例如.o文件）的方法。每个后缀规 则以两个成对出现的后缀名定义，例如，将.c文件转换为.o文件的后缀规则可定义为：

```
.c.o:
    $(CC) $(CCFLAGS) -c -o $@ $<
```

6.6 隐含规则

2：模式规则（pattern rules）。

- 这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。 在目标名称的前面多了一个%号，同时可用来定义目标和依赖 文件之间的关系，例如下面的模式规则定义了如何将任意一个X.c文件转换为X.o文件：

```
%.o:%.c
    $(CC) $(CCFLAGS) -c -o $@ $<
```

练习：

- 完成上面例子的Makefile
- 要求：
  - 增加all、clean、install伪目标
  - 使用变量增加Makefile文件的可移植性

6.7 Makefile目标编译

Neuedu

- 如果不指定目标（target），make会默认第一个target

- 规范的makefile文件都有以下常见的几个目标：
  make all - 编译所有目标
  make clean - 在编译结束后删除 .o文件
  make install - 编译结束后将最终的可执行文件安装到系统的某一个位置

```
#makefile for example
example: example.o add.o modify.o delete.o
        $(CC) -o $@ $^
.c.o:
        $(CC) -c $<
all: example
clean: all
        rm -f *.o
install: clean
        cp example /usr/local/bin
```

课堂练习

- 把gcc的学生测试程序用makefile来构造
- 自行libtu.*的测试程序用makefile来构造