

说说**Makefile**那些事儿

## 说说Makefile那些事儿

[扬说]透过现象看本质

工作至今，一直对Makefile半知半解。突然某天幡然醒悟，觉得此举极为不妥，只得洗心革面从头学来，以前许多不明觉厉之处顿时茅塞顿开，想想好记性不如烂笔头，便来说说Makefile那些事儿。

### • Makefile到底是个啥玩意儿

Makefile就是一文本文件。

-----

\$ file Makefile

Makefile: ASCII make commands text

-----

一般来说，我们平时所称的Makefile是指make命令以及Makefile文件，Makefile文件中记录着各种规则，make命令通过分析Makefile执行规则中的操作。

看看百度百科的定义：

make是一个命令工具，它解释Makefile 中的指令（应该说是规则）。在Makefile文件中描述了整个工程所有文件的编译顺序、编译规则。Makefile 有自己的书写格式、关键字、函数。像C 语言有自己的格式、关键字和函数一样。而且在Makefile 中可以使用系统shell所提供的任何命令来完成想要的工作。Makefile（在其它的系统上可能是另外的文件名）在绝大多数的IDE 开发环境中都在使用，已经成为一种工程的编译方法。

再看看官方文档的定义：

You need a file called a makefile to tell make what to do. Most often, the makefile tells make how to compile and link a program.

### • Makefile的由来

任何一种技能或知识都是源于某种社会需求，那为什么要用Makefile呢？

当项目源文件很少的时候，我们也许还可以手动使用gcc命令来进行编译，但是当项目发展到一个庞大的规模时，再手动敲gcc命令去编译就变得不可能的事情。所以呢，在这样的历史背景下，就出现了某个大牛（斯图亚特·费尔德曼），在某年（1977年）在某地（贝尔实验室）制作了这样一个软件，它的名字就叫做make。

用一句话来说明为啥用Makefile：为了实现自动化（当然大多数场景都是用在自动化编译中）。

另外在编译过程中，为了节省时间，希望仅编译修改过的文件，这也是Makefile在设计时一个重要的设计观点。

### • Makefile的组成部分

Makefile包含五个东西：显示规则，隐式规则，变量定义，文件指示，注释。具体含义还是直接引用网上的版本吧|||

- 1、显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
  - 2、隐式规则。由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。
  - 3、变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
  - 4、文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。
  - 5、注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“/ #”。
- 最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

这里说说规则(Rules):

```
target ... : prerequisites ...  
    command  
    ...  
    ...
```

一条规则由三部分组成，目标 (target)、先决条件 (prerequisites)、命令 (commands)。

target也就是一个目标文件，可以是Object File，也可以是执行文件。还可以是一个标签 (Label)。

prerequisites就是，要生成那个target所需要的文件或是目标。

command也就是make需要执行的命令。（任意的Shell命令）

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在command中。

说白了就是说，prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。

这就是Makefile的规则。也就是Makefile中最核心的内容。

## • Makefile的核心思想

四个字，依赖关系

## • Makefile之执行过程

1. 依次读取变量“MAKEFILES”定义的makefile文件列表
2. 读取工作目录下的makefile文件（根据命名的查找顺序“GNUmakefile”，“makefile”，“Makefile”，首先找到那个就读取那个）
3. 依次读取工作目录makefile文件中使指示符“include”包含的文件
4. 查找重建所有已读取的makefile文件的规则（如果存在一个目标是当前读取的某一个makefile文件，则执行此规则重建此makefile文件，完成后从第一步开始重新执行）
5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支
6. 根据“终极目标”以及其他目标的依赖关系建立依赖关系链表
7. 执行除“终极目标”以外的所有的目标的规则（规则中如果依赖文件中任一个文件的时间戳比目标文件新，则使用规则所定义的命令重建目标文件）

8. 执行“终极目标”所在的规则

• Makefile之模式规则

模式规则其实也是普通规则，但它使用了如%这样的通配符。如下面的例子：

```
%.o : %.c

$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

此规则描述了一个.o文件如何由对应的.c文件创建。规则的命令行中使用了自动化变量“\$<”和“\$@"，其中自动化变量“\$<”代表规则的依赖，“\$@"代表规则的目标。此规则在执行时，命令行中的自动化变量将根据实际的目标和依赖文件取对应值。

• Makefile之隐式规则

如果发现某变量在shell和makefile中未找不到其定义，那么恭喜你，你极大可能遇到隐式规则了。当然隐式规则中的变量只是隐式规则的一部分。

1. 隐式规则中的变量

隐式规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是参数相关的，如“CFLAGS”。

1) 与命令相关的变量

变量	含义
AR	函数库打开包程序。默认命令是“ar”
AS	汇编语言编译程序。默认命令是“as”
CC	C语言编译程序。默认命令是“cc”
CXX	C++语言编译程序。默认命令是“g++”
CO	从RCS文件中扩展文件程序。默认命令是“co”
CPP	C程序的预处理器（输出是标准输出设备）。默认命令是“\$(CC) -E”
FC	Fortran和Ratfor的编译器和预处理程序。默认命令是“f77”
GET	从SCCS文件扩展文件的程序。默认命令是“get”
LEX	Lex方法分析器程序（针对于C或Ratfor）。默认命令是“lex”
PC	Pascal语言编译程序。默认命令是“pc”
YACC	Yacc文法分析器（针对C程序）。默认命令是“yacc”
YACCR	Yacc文法分析器（针对Ratfor程序）。默认命令是“yacc -r”
MAKEINFO	转换Texinfo源文件(.texi)到info文件程序。默认命令是“makeinfo”
TEX	从TeX源文件创建TeX DVI文件的程序。默认命令是“tex”
WEAVE	转化Web到TeX的程序。默认命令是“weave”

TEXI2DVI	从Texinfo源文件创建TeX DVI文件的程序。默认命令是 “texi2dvi”
CWEAVE	转化C Web到TeX的程序。默认命令是 “cweave”
TANGLE	转换Web到Pascal语言的程序，默认命令是“ tangle ”
CTANGLE	转换C Web到C。默认命令是“ ctangle ”
RM	删除文件命令。默认命令是“ rm -f ”

## 2) 与参数相关的变量

变量	含义
ARFLAGS	函数库打包程序AR命令的参数。默认值是 “rv”
ASFLAGS	汇编语言编译参数（当明显地调用“ .s” 或“ .S” 文件时）
CFLAGS	C语言编译器参数
CXXFLAGS	C++语言编译器参数
COFLAGS	RCS命令参数
CPPFLAGS	C预处理器参数（C和Fortran编译器也会用到）
FFLAGS	Fortran语言编译器参数
GFLAGS	SCCS “ get ”程序参数
LDFLAGS	连接器参数（如 “ld” ）
LFLAGS	Lex文法分析器参数
PFLAGS	Pascal语法编译器参数
RFLAGS	Ratfor程序的Fortran编译器参数
YFLAGS	Yacc文法分析器参数

## 2. 使用模式规则

可以使用模式规则定义一个隐式规则。和一般规则类似，只是在模式规则中，目标的定义需要有 “%” 字符。 “%” 定义对文件名的匹配，表示任意长度的非空字符串。在依赖目标中同样可以使用 “%” ，只是依赖目标中 “%” 的取值，取决于其目标。

模式规则中 “%” 的展开和变量与函数的展开是有区别的， “%” 的展开发生在变量和函数的展开之后。变量和函数的展开发生在make载入Makefile时，而 “%” 的展开则发生在运行时。

### 1) 模式规则举例

模式规则中，至少在规则的目标中要包含 “%” 符号。

```
%o : %c ; <command .....>
```

其含义是，字指出了从所有的.c文件生成相应的.o文件的规则。如果要生成的目标是“ a.o b.o” ，那么

%c” 就是“ a.c b.c” 。

```
%o : %c
```

`$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@`

表示把所有的.c文件都编译成.o文件。

其中，“\$@" 表示所有目标的集合，“\$<” 表示所有依赖目标的集合（在模式定义规则的情形下）。

## 2) 自动化变量

自动化变量只应出现在规则的命令中。

变量	含义
<code>\$@</code>	表示规则中的所有目标文件的集合。在模式规则中如果有多个目标，“\$@" 就是匹配于目标中模式定义的集合
<code>%</code>	仅当目标是函数库文件时，表示规则中的目标成员名，如果目标不是函数库文件（UNIX下是 .a，Windows是 .lib），其值为空。
<code>\$&lt;</code>	依赖目标中的第一个目标名字，如果依赖目标是以模式（即“%”）定义的，则“\$<” 是符合模式的一系列的文件集
<code>\$?</code>	所有比目标新的依赖目标的集合，以空格分隔
<code>^</code>	所有依赖目标的集合，以空格分隔。如如果在依赖目标中有多个重复的，则自动去除重复的依赖目标，只保留一份
<code>+</code>	同“^”，也是所有依赖目标的集合，只是它不去除重复的依赖目标。
<code>*</code>	目标模式中“%” 及其之前的部分
<code>\$(@D)</code>	“\$@" 的目录部分（不以斜杠作为结尾），如果“\$@" 中没有包含斜杠，其值为“.”（当前目录）
<code>\$(@F)</code>	“\$@" 的文件部分，相当于函数“\$(notdir \$@)”
<code>\$(*D)</code>	同“\$(@D)”，取文件的目录部分
<code>\$(*F)</code>	同“\$(@F)”，取文件部分，但不取后缀名
<code>\$(%D)</code>	函数包文件成员的目录部分
<code>\$(%F)</code>	函数包文件成员的文件名部分
<code>\$(&lt;D)</code>	依赖目标中的第一个目标的目录部分
<code>\$(&lt;F)</code>	依赖目标中的第一个目标的文件名部分
<code>\$(^D)</code>	所有依赖目标文件中目录部分（无相同的）
<code>\$(^F)</code>	所有依赖目标文件中文件名部分（无相同的）
<code>\$(+D)</code>	所有依赖目标文件中的目录部分（可以有相同的）
<code>\$(+F)</code>	所有依赖目标文件中的文件名部分（可以有相同的）
<code>\$(?D)</code>	所有被更新文件的目录部分
<code>\$(?F)</code>	所有被更新文件的文件名部分

## • Makefile那些稀奇古怪的符号

---

这些稀奇古怪的符号是前面隐式规则中出现过，单独拎出来是因为我们会经常用到它们。

这些符号也就是我们常说的自动变量：

`$@`：规则中的目标集

`$$`：规则中的所有先决条件

`$<`：表示规则中的第一个先决条件

再来说说`$VAR`和`$$VAR`的区别：

makefile文件中的规则绝大部分都是使用shell命令来实现的，这里就涉及到了变量的使用，包括**makefile中的变量**和**shell命令范畴内的变量**。在makefile的规则命令行中使用`$var`**就是在命令中引用makefile的变量**，这里仅仅是读取makefile的变量然后展开，将其值作为参数传给了一个shell命令；而`$$var`**是在访问一个shell命令内定义的变量**，而非makefile的变量。如果某规则有n个shell命令行构成，而相互之间没有用';'和'\''连接起来的话，就是相互之间没有关联的shell命令，相互之间也不能变量共享。

## • Makefile之伪目标

---

### 使用其原因一：避免和同名文件冲突

在现实中难免存在所定义的目标与所存在的目标是同名的，采用Makefile如何处理这种情况呢？Makefile中的假目标（phony target）可以解决这个问题。

假目标可以使用.PHONY关键字进行声明，对于假目标，可以想象，因为不依赖于某文件，make该目标的时候，其所在规则的命令都会被执行。

如果编写一个规则，并不产生目标文件，则其命令在每次make 该目标时都执行。

例如：

clean:

rm \*.o temp

因为"rm"命令并不产生"clean"文件，则每次执行"make clean"的时候，该命令都会执行。如果目录中出现了"clean"文件，则规则失效了：没有依赖文件，文件"clean"始终是最新的，命令永远不会执行；为避免这个问题，可使用".PHONY"指明该目标。如：

.PHONY : clean

这样执行"make clean"会无视"clean"文件存在与否。

已知phony 目标并非是由其它文件生成的实际文件，make 会跳过隐含规则搜索。这就是声明phony 目标会改善性能的原因，即使你并不担心实际文件存在与否。

完整的例子如下：

.PHONY : clean

clean :

rm \*.o temp

### 使用其原因二：提高执行make的效率

当一个目标被声明为伪目标后，make在执行此规则时不会试图去查找隐含规则来创建这个目标。这样也提高了make的执行效率，同时我们也不用担心由于目标和文件名重名而使我们的期望失败。

## • Makefile那些赋值的事儿

---

面试中经常被问到的问题：`[=]`和`[:=]`符号的区别。

=

可以先使用后定义，这就导致makefile在全部展开后才能决定变量的值。

有可能出现循环递归，无法暂开的问题。

: =

必须先定义然后再使用，在当前的位置就可以决定变量的值。

再补充两种符号？=、+=，如果熟悉C语言那对这两种符号理解会很容易。

? =

相当于选择疑问句，如果前面的变量没被赋值，那就做赋值操作

+=

相当于递加操作

看一个例子就明白了。新建一个Makefile，内容如下：



```
ifdef DEFINE_VRE

    VRE = "Hello World!"

else

endif

ifeq ($(OPT),define)

    VRE ?= "Hello World! First!"

endif

ifeq ($(OPT),add)

    VRE += "Kelly!"

endif

ifeq ($(OPT),recover)

    VRE := "Hello World! Again!"


endif

all:

    @echo $(VRE)
```



敲入以下make命令：



```
make DEFINE_VRE=true OPT=define 输出: Hello World!

make DEFINE_VRE=true OPT=add 输出: Hello World! Kelly!

make DEFINE_VRE=true OPT=recover 输出: Hello World! Again!

make DEFINE_VRE= OPT=define 输出: Hello World! First!
```

```
make DEFINE_VRE= OPT=add 输出: Kelly!  
make DEFINE_VRE= OPT=recover 输出: Hello World! Again!
```



---

posted @ 2014-12-12 16:43 子扬 阅读(11947) 评论(2) 编辑 收藏