联航精英训练营

UNIGRESS ELITE TRAINING CAMP

Linux开发基础-Makefile

GNU make和makefile

GNU make概述
Makefile 的基本结构
Makefile中的变量
GNU make 的主要预定义变量
Makefile的隐含规则
make命令行选项

GNU make概述

- 在大型的开发项目中，人们通常利用 make 工具来自动完成编译工作。这些工作包括：
  - 如果仅修改了某几个源文件，则只重新编译这几个源文件；
  - 如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。
  - 利用这种自动编译可大大简化开发工作，避免不必要的重新编译。
- 实际上，make 工具通过一个称为 makefile 的文件来完成并自动维护编译工作。makefile 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。 当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。
- 默认情况下，GNU make 工具在当前工作目录按如下顺序搜索 makefile：
  - GNUmakefile
  - makefile
  - Makefile

makefile举例

- 在UNIX中，习惯使用makefile 作为 makfile 文件。
- Linux程序员使用第三种文件名Makefile。因为第一个字母是大写，通常被列在一个目录的文件列表的最前面。
- 如果要使用其他文件作为 makefile，则可利用类 似下面的 make 命令选项指定 makefile 文件：

```
$ make -f Makefile.debug
```

- 例1：一个简单的makefile

```
prog:prog1.o prog2.o
        gcc prog1.o prog2.o -o prog
prog1.o:prog1.c lib.h
        gcc -c -I. -o prog1.o prog1.c
prog2.o:prog2.c
        gcc -c prog2.c
```

Makefile 的基本结构（1/2）

- Makefile是一个**文本形式的数据库文件**，其中包含一些规则来**告诉make处理哪些文件以及如何处理这些文件**。

- 规则主要是描述哪些文件是从哪些别的文件（称为dependency**依赖文件**）中产生的，以及**用什么命令**（command）来执行这个过程。

- 依靠这些信息，make会对磁盘上的文件进行检查，如果目标文件的生成或被改动时的时间（称为该文件时间戳）至少比它的一个依赖文件还旧的话，make就执行相应的命令，以更新目标文件。
- **目标文件不一定是最后的可执行文件**，可以是任何一个中间文件并可以作为其他目标文件的依赖文件。

Makefile 的基本结构（2/2）

- Makefile规则的一般形式如下：

  target：dependency dependency

  (tab)<command>

- 一个Makefile文件主要含有一系列的规则，每条规则包含以下内容。
  - 一个目标（target），即make最终需要创建的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如“clean”。
  - 一个或多个依赖文件（dependency）列表，通常是编译目标文件所需要的其他文件。
  - 一系列命令(command)，是make执行的动作，通常是把指定的相关文件编译成目标文件的编译命令，每个命令占一行，**且每个命令行的起始字符必须为TAB字符。**

- 除非特别指定，否则**make的工作目录就是当前目录。**target是需要创建的二进制文件或目标文件，dependency是在创建target时需要用到的一个或多个文件的列表，命令序列是创建target文件所需要执行的步骤，比如编译命令。

Makefile实例

```
# 以#开头的为注释行
test：prog.o code.o
        gcc –o test prog.o code.o
 prog.o：prog.c prog.h code.h
        gcc –c prog.c –o prog.o
 code.o：code.c code.h
        gcc –c code.c –o code.o
 clean：
        rm –f *.o
```

- 上面的Makefile文件中共定义了四个目标：test、prog.o、code.o和clean。
- 目标从每行的最左边开始写，后面跟一个冒号（：），如果有与这个目标有依赖性的其他目标或文件，把它们列在冒号后面，并以空格隔开。然后另起一行开始写实现这个目标的一组命令。
- 在Makefile中，可使用续行号（\）将一个单独的命令行延续成几行。但要注意在续行号（\）后面不能跟任何字符（包括空格和键）

Makefile实例

- 一般情况下，调用make命令可输入：
  - # make target
  - target是Makefile文件中定义的目标之一，如果省略target，make就将生成Makefile文件中定义的第一个目标。

- 对于上面Makefile的例子，单独的一个“make”命令等价于：
  - # make test
  - 因为test是Makefile文件中定义的第一个目标，make首先将其读入，然后从第一行开始执行，把第一个目标test作为它的最终目标，所有后面的目标的更新都会影响到test的更新。
  - 第一条规则说明只要文件test的时间戳比文件prog.o或code.o中的任何一个旧，下一行的编译命令将会被执行。

Make的工作过程

- 现在来看一下make做的工作：
  - 首先make按顺序读取makefile中的规则，
  - 然后检查该规则中的依赖文件与目标文件的时间戳哪个更新
    - 如果目标文件的时问戳比依赖文件还早，就按规则中定义的命令更新目标文件。
    - 如果该规则中的依赖文件又是其他规则中的目标文件，那么依照规则链不断执行这个过程，直到Makefile文件的结束，至少可以找到一个不是规则生成的最终依赖文件，获得此文件的时间戳
    - 然后从下到上依照规则链执行目标文件的时间戳比此文件时间戳旧的规则，直到最顶层的规则

- 通过以上的分析过程，可以看到make的优点，因为.o目标文件依赖.c源文件，源码文件里一个简单改变都会造成那个文件被重新编译，并根据规则链依次由下到上执行编译过程，直到最终的可执行文件被重新连接。
  - 例如，当改变一个头文件的时候，由于所有的依赖关系都在Makefile里，因此不再需要记住依赖此头文件的所有源码文件，make可以自动的重新编译所有那些因依赖这个头文件而改变了的源码文件，如果需要，再进行重新连接

Makefile中的变量

- Makefile里的变量就像一个环境变量。事实上，环境变量在make中也被解释成make的变量。这些变量对大小写敏感，一般使用大写字母。几乎可以从任何地方引用定义的变量，变量的主要作用如下：
  - 保存文件名列表。
  - 保存可执行命令名，如编译器。
  - 保存编译器的参数。

变量的定义和使用

- Makefile中的变量是用一个文本串在Makefile中定义的，这个文本串就是变量的值。只要在一行的开始写下这个变量的名字，后面跟一个“=”号，以及要设定这个变量的值即可定义变量，下面是定义变量的语法：

  VARNAME=string

- 使用时，把变量用括号括起来，并在前面加上$符号，就可以引用变量的值：

  ${VARNAME}

- make解释规则时，VARNAME在等式右端展开为定义它的字符串。
- 变量一般都在Makefile的头部定义。按照惯例，所有的Makefile变量都应该是大写。如果变量的值发生变化，就只需要在一个地方修改，从而简化了Makefile的维护。

Makefile变量举例

- 现在利用变量把前面的Makefile重写一遍：

```
OBJS=prog.o code.o
CC=gcc
 test：  ${ OBJS }
            ${ CC } –o test ${ OBJS }
 prog.o：  prog.c prog.h code.h
            ${ CC } –c prog.c –o prog.o
 code.o：  code.c code.h
            ${ CC } –c code.c –o code.o
.PHONY: clean
 clean：
            rm –f *.o
```

变量的类型

- 除用户自定义的变量外，make还允许使用
  - 环境变量
    - 使用环境变量的方法很简单，在make启动时，make读取系统当前已定义的环境变量，并且创建与之同名同值的变量，因此用户可以像在shell中一样在Makefile中方便的引用环境变量。
    - 需要注意的是，如果用户在Makefile中定义了同名的变量，用户自定义变量将覆盖同名的环境变量
  - 自动变量
  - 预定义变量

GNU make 的主要预定义变量

$* 不包含扩展名的目标文件名称。

$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。

$< 第一个依赖文件的名称。

$? 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。

$@ 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。

$% 如果目标是归档成员，则该变量表示目标的归档成员名称。例如，如果目标名称 为 mytarget.so(image.o)，则 $@ 为 mytarget.so，而 $% 为 image.o。

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。

GNU make 的主要预定义变量

CC	C 编译器的名称，默认值为 cc。
CFLAGS	C 编译器的选项。
CPP	C 预编译器的名称，默认值为 $(CC) -E。
CPPFLAGS	C 预编译的选项。 CXX C++ 编译器的名称，默认值为 g++。
CXXFLAGS	C++ 编译器的选项。

Makefile的隐含规则

- 在上面的例子中，几个产生目标文件的命令都是从“.c”的C语言源文件和相关文件通过编译产生“.o”目标文件，这也是一般的步骤。实际上，make可以使工作更加自动化，也就是说，make知道一些默认的动作，它有一些称作隐含规则的内置的规则，这些规则告诉make当用户没有完整地给出某些命令的时候，应该怎样执行。
- 例如，把生成prog.o和code.o的命令从规则中删除，make将会查找隐含规则，然后会找到并执行一个适当的命令。由于这些命令会使用一些变量，因此可以通过改变这些变量来定制make。象在前面的例子中所定义的那样，make使用变量CC来定义编译器，并且传递变量CFLAGS（编译器参数）、CPPFLAGS（C语言预处理器参数）、TARGET_ARCH（目标机器的结构定义）给编译器，然后加上参数-c，后面跟变量$<（第一个依赖文件名），然后是参数-o加变量$@（目标文件名）。
- 综上所述，一个C编译的具体命令将会是：

  $ {CC} $ {CFLAGS} $ {CPPFLAGS} $ {TARGET_ARCH} –c $< -o $@

隐含规则举例

- 在上面的例子中，利用隐含规则，可以简化为：

```
OBJS=prog.o code.o
CC=gcc
test：${ OBJS }
        ${ CC } –o $@ $^
 prog.o：prog.c prog.h code.h
 code.o：code.c code.h
 clean：
        rm –f *.o
```

make命令行选项

- 直接在 make 命令的后面键入目标名可建立指定的目标，如果直接运行 make，则建立第一个目标。还可以用 make -f mymakefile 这样的命令指定 make 使用特定的 makefile，而不是 默认的 GNUmakefile、makefile 或 Makefile。
- GNU make 命令还有一些其他选项，下面是 GNU make 命令的常用命令行选项命令行选项含义：
  - -C DIR　在读取 makefile 之前改变到指定的目录 DIR。
  - -f FILE　以指定的 FILE 文件作为 makefile。
  - -h　显示所有的 make 选项。
  - -i　忽略所有的命令执行错误。
  - -I DIR　当包含其他 makefile 文件时，可利用该选项指定搜索目录。
  - -n　只打印要执行的命令，但不执行这些命令。
  - -p　显示 make 变量数据库和隐含规则。
  - -s　在执行命令时不显示命令。
  - -w　在处理 makefile 之前和之后，显示工作目录。
  - -W FILE　假定文件 FILE 已经被修改。

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.