跟我一起寫**Makefile:MakeFile**介紹

出自**Ubuntu**中文

*導 * *航 * |概述 + **MakeFile**介紹 + 書寫規則 + 書寫命令 + 使用變量 + 使用條件判斷 | |使用函數 + make運行 + 隱含規則 + 使用make更新函數庫文件 + 後序 |

目錄

makefile 介紹

make命令執行時，需要一個 makefile 文件，以告訴make命令需要怎麼樣的去編譯和鏈接程序。

首先，我們用一個示例來說明makefile的書寫規則。

以便給大家一個感性認識。這個示例來源於gnu的make使用手冊，在這個示例中，我們的工程有8個c文件，和3個頭文件，我們要寫一個makefile來告訴make命令如何編譯和鏈接這幾個文件。我們的規則是：

1）如果這個工程沒有編譯過，那麼我們的所有c文件都要編譯並被鏈接。

2）如果這個工程的某幾個c文件被修改，那麼我們只編譯被修改的c文件，並鏈接目標程序。

3）如果這個工程的頭文件被改變了，那麼我們需要編譯引用了這幾個頭文件的c文件，並鏈接目標程序。

只要我們的makefile寫得夠好，所有的這一切，我們只用一個make命令就可以完成，make命令會自動智能地根據當前的文件修改的情況來確定哪些文件需要重編譯，從而自己編譯所需要的文件和鏈接目標程序。

makefile的規則

在講述這個makefile之前，還是讓我們先來粗略地看一看makefile的規則。

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

target可以是一個object file(目標文件)，也可以是一個執行文件，還可以是一個標籤（label）。對於標籤這種特性，在後續的“偽目標”章節中會有敘述。

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

command也就是make需要執行的命令。（任意的shell命令）

這是一個文件的依賴關係，也就是說，target這一個或多個的目標文件依賴於prerequisites中的文件，其生成規則定義在 command中。說白一點就是說，prerequisites中如果有一個以上的文件比target文件要新的話，command所定義的命令就會被執行。這就是makefile的規則。也就是makefile中最核心的內容。

說到底，makefile的東西就是這樣一點，好像我的這篇文檔也該結束了。呵呵。還不盡然，這是makefile的主線和核心，但要寫好一個makefile還不夠，我會以後面一點一點地結合我的工作經驗給你慢慢道來。內容還多着呢。：）

一個示例

正如前面所說的，如果一個工程有3個頭文件，和8個c文件，我們為了完成前面所述的那三個規則，我們的makefile應該是下面的這個樣子的。

```
edit  : main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o

main.o  : main.c defs.h
          cc -c main.c
kbd.o  : kbd.c defs.h command.h
          cc -c kbd.c
command.o  : command.c defs.h command.h
          cc -c command.c
display.o  : display.c defs.h buffer.h
          cc -c display.c
insert.o  : insert.c defs.h buffer.h
          cc -c insert.c
search.o  : search.c defs.h buffer.h
          cc -c search.c
files.o  : files.c defs.h buffer.h command.h
          cc -c files.c
utils.o  : utils.c defs.h
          cc -c utils.c
clean  :
          rm edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
```

反斜杠（\）是換行符的意思。這樣比較便於makefile的易讀。我們可以把這個內容保存在名字為“makefile”或“Makefile” 的文件中，然後在該目錄下直接輸入命令“make”就可以生成執行文件edit。如果要刪除執行文件和所有的中間目標文件，那麼，只要簡單地執行一下“make clean”就可以了。

在這個makefile中，目標文件（target）包含：執行文件edit和中間目標文件（*.o），依賴文件（prerequisites）就是冒號後面的那些 .c 文件和 .h文件。每一個 .o 文件都有一組依賴文件，而這些 .o 文件又是執行文件 edit 的依賴文件。依賴關係的實質上就是說明了目標文件是由哪些文件生成的，換言之，目標文件是哪些文件更新的。

在定義好依賴關係后，後續的那一行定義了如何生成目標文件的操作系統命令，一定要以一個tab鍵作為開頭。記住，make並不管命令是怎麼工作的，他只管執行所定義的命令。make會比較targets文件和prerequisites文件的修改日期，如果prerequisites文件的日期要比targets文件的日期要新，或者target不存在的話，那麼，make就會執行後續定義的命令。

這裏要說明一點的是，clean不是一個文件，它只不過是一個動作名字，有點像c語言中的lable一樣，其冒號后什麼也沒有，那麼，make就不會自動去找它的依賴性，也就不會自動執行其後所定義的命令。要執行其後的命令，就要在make命令后明顯得指出這個lable的名字。這樣的方法非常有用，我們可以在一個makefile中定義不用的編譯或是和編譯無關的命令，比如程序的打包，程序的備份，等等。

make是如何工作的

在默認的方式下，也就是我們只輸入make命令。那麼，

1. make會在當前目錄下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到，它會找文件中的第一個目標文件（target），在上面的例子中，他會找到“edit”這個文件，並把這個文件作為最終的目標文件。
3. 如果edit文件不存在，或是edit所依賴的後面的 .o 文件的文件修改時間要比edit這個文件新，那麼，他就會執行後面所定義的命令來生成edit這個文件。
4. 如果edit所依賴的.o文件也不存在，那麼make會在當前文件中找目標為.o文件的依賴性，如果找到則再根據那一個規則生成.o文件。（這有點像一個堆棧的過程）
5. 當然，你的C文件和H文件是存在的啦，於是make會生成 .o 文件，然後再用 .o 文件生成

make的終極任務，也就是執行文件edit了。

這就是整個make的依賴性，make會一層又一層地去找文件的依賴關係，直到最終編譯出第一個目標文件。在找尋的過程中，如果出現錯誤，比如最後被依賴的文件找不到，那麼make就會直接退出，並報錯，而對於所定義的命令的錯誤，或是編譯不成功，make根本不理。make只管文件的依賴性，即，如果在我找了依賴關係之後，冒號後面的文件還是不在，那麼對不起，我就不工作啦。

通過上述分析，我們知道，像clean這種，沒有被第一個目標文件直接或間接關聯，那麼它後面所定義的命令將不會被自動執行，不過，我們可以顯示要make執行。即命令——“make clean”，以此來清除所有的目標文件，以便重編譯。

於是在我們編程中，如果這個工程已被編譯過了，當我們修改了其中一個源文件，比如file.c，那麼根據我們的依賴性，我們的目標file.o會被重編譯（也就是在這個依性關係後面所定義的命令），於是file.o的文件也是最新的啦，於是file.o的文件修改時間要比edit要新，所以edit也會被重新鏈接了（詳見edit目標文件后定義的命令）。

而如果我們改變了“command.h”，那麼，kdb.o、command.o和files.o都會被重編譯，並且，edit會被重鏈接。

makefile中使用變量

在上面的例子中，先讓我們看看edit的規則：

```
edit  : main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
```

我們可以看到[.o]文件的字符串被重複了兩次，如果我們的工程需要加入一個新的[.o]文件，那麼我們需要在兩個地方加（應該是三個地方，還有一個地方在clean中）。當然，我們的makefile並不複雜，所以在兩個地方加也不累，但如果makefile變得複雜，那麼我們就有可能會忘掉一個需要加入的地方，而導致編譯失敗。所以，為了makefile的易維護，在makefile中我們可以使用變量。makefile的變量也就是一個字符串，理解成C語言中的宏可能會更好。

比如，我們聲明一個變量，叫objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什麼啦，只要能夠表示obj文件就行了。我們在makefile一開始就這樣定義：

```
objects = main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
```

於是，我們就可以很方便地在我們的makefile中以“$(objects)”的方式來使用這個變量了，於是我們的改良版makefile就變成下面這個樣子：

```
objects = main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o

edit  : $(objects)
          cc -o edit $(objects)
main.o  : main.c defs.h
          cc -c main.c
kbd.o  : kbd.c defs.h command.h
          cc -c kbd.c
command.o  : command.c defs.h command.h
          cc -c command.c
display.o  : display.c defs.h buffer.h
          cc -c display.c
insert.o  : insert.c defs.h buffer.h
          cc -c insert.c
search.o  : search.c defs.h buffer.h
          cc -c search.c
files.o  : files.c defs.h buffer.h command.h
          cc -c files.c
utils.o  : utils.c defs.h
          cc -c utils.c
clean  :
          rm edit $(objects)
```

於是如果有新的 .o 文件加入，我們只需簡單地修改一下 objects 變量就可以了。

關於變量更多的話題，我會在後續給你一一道來。

讓**make**自動推導

GNU的make很強大，它可以自動推導文件以及文件依賴關係後面的命令，於是我們就沒必要去在每一個[.o]文件后都寫上類似的命令，因為，我們的make會自動識別，並自己推導命令。

只要make看到一個[.o]文件，它就會自動的把[.c]文件加在依賴關係中，如果make找到一個whatever.o，那麼 whatever.c，就會是whatever.o的依賴文件。並且 cc -c whatever.c 也會被推導出來，於是，我們的makefile 再也不用寫得這麼複雜。我們的新makefile又出爐了。

```
objects = main.o kbd.o command.o display.o \
                  insert.o search.o files.o utils.o
 cc = gcc

edit  : $(objects)
          cc -o edit $(objects)

main.o  : defs.h
kbd.o  : defs.h command.h
command.o  : defs.h command.h
display.o  : defs.h buffer.h
insert.o  : defs.h buffer.h
search.o  : defs.h buffer.h
files.o  : defs.h buffer.h command.h
utils.o  : defs.h

.PHONY  : clean
clean  :
          rm edit $(objects)
```

這種方法，也就是make的“隱晦規則”。上面文件內容中，“.PHONY”表示，clean是個偽目標文件。

關於更為詳細的“隱晦規則”和“偽目標文件”，我會在後續給你一一道來。

另類風格的**makefile**

既然我們的make可以自動推導命令，那麼我看到那堆[.o]和[.h]的依賴就有點不爽，那麼多的

重複的[.h]，能不能把其收攏起來，好吧，沒有問題，這個對於make來說很容易，誰叫它提供了自動推導命令和文件的功能呢？來看看最新風格的makefile吧。

```
objects = main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

edit  : $(objects)
         cc -o edit $(objects)

$(objects)  : defs.h
kbd.o command.o files.o  : command.h
display.o insert.o search.o files.o  : buffer.h

.PHONY  : clean
clean  :
         rm edit $(objects)
```

這種風格，讓我們的makefile變得很簡單，但我們的文件依賴關係就顯得有點凌亂了。魚和熊掌不可兼得。還看你的喜好了。我是不喜歡這種風格的，一是文件的依賴關係看不清楚，二是如果文件一多，要加入幾個新的.o文件，那就理不清楚了。

清空目標文件的規則

每個Makefile中都應該寫一個清空目標文件（.o和執行文件）的規則，這不僅便於重編譯，也很利於保持文件的清潔。這是一個“修養”（呵呵，還記得我的《編程修養》嗎）。一般的風格都是：

```
clean:
         rm edit $(objects)
```

更為穩健的做法是：

```
.PHONY  : clean
clean  :
         -rm edit $(objects)
```

前面說過，.PHONY意思表示clean是一個“偽目標”，。而在rm命令前面加了一個小減號的意思就是，也許某些文件出現問題，但不要管，繼續做後面的事。當然，clean的規則不要放在文件的開頭，不然，這就會變成make的默認目標，相信誰也不願意這樣。不成文的規矩是——“clean從來都是放在文件的最後”。

上面就是一個makefile的概貌，也是makefile的基礎，下面還有很多makefile的相關細節，準備好了嗎？準備好了就來。

Makefile里有什麼？

Makefile里主要包含了五個東西：顯式規則、隱晦規則、變量定義、文件指示和註釋。

1. 顯式規則。顯式規則說明了，如何生成一個或多的的目標文件。這是由Makefile的書寫者明顯指出，要生成的文件，文件的依賴文件，生成的命令。
2. 隱晦規則。由於我們的make有自動推導的功能，所以隱晦的規則可以讓我們比較簡略地書寫Makefile，這是由make所支持的。
3. 變量的定義。在Makefile中我們要定義一系列的變量，變量一般都是字符串，這個有點像你C語言中的宏，當Makefile被執行時，其中的變量都會被擴展到相應的引用位置上。

4. 文件指示。其包括了三個部分，一個是在一個Makefile中引用另一個Makefile，就像C語言中的include一樣；另一個是指根據某些情況指定Makefile中的有效部分，就像C語言中的預編譯#if一樣；還有就是定義一個多行的命令。有關這一部分的內容，我會在後續的部分中講述。
5. 註釋。Makefile中只有行註釋，和UNIX的Shell腳本一樣，其註釋是用“#”字符，這個就像C/C++中的“//”一樣。如果你要在你的Makefile中使用“#”字符，可以用反斜框進行轉義，如：“\#”。

最後，還值得一提的是，在Makefile中的命令，必須要以[Tab]鍵開始。

Makefile的文件名

默認的情況下，make命令會在當前目錄下按順序找尋文件名為“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解釋這個文件。在這三個文件名中，最好使用“Makefile”這個文件名，因為，這個文件名第一個字符為大寫，這樣有一種顯目的感覺。最好不要用 “GNUmakefile”，這個文件是GNU的make識別的。有另外一些make只對全小寫的“makefile”文件名敏感，但是基本上來說，大多數的make都支持“makefile”和“Makefile”這兩種默認文件名。

當然，你可以使用別的文件名來書寫Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的Makefile，你可以使用make的“-f”和“--file”參數，如：make -f Make.Linux或make --file Make.AIX。

引用其它的**Makefile**

在Makefile使用include關鍵字可以把別的Makefile包含進來，這很像C語言的#include，被包含的文件會原模原樣的放在當前文件的包含位置。include的語法是：

```
include <filename>;
```

filename可以是當前操作系統Shell的文件模式（可以包含路徑和通配符）

在include前面可以有一些空字符，但是絕不能是[Tab]鍵開始。include和<filename>;可以用一個或多個空格隔開。舉個例子，你有這樣幾個Makefile：a.mk、b.mk、c.mk，還有一個文件叫foo.make，以及一個變量$(bar)，其包含了 e.mk和f.mk，那麼，下面的語句：

```
include foo.make *.mk $(bar)
```

等價于：

```
include foo.make a.mk b.mk c.mk e.mk f.mk
```

make命令開始時，會找尋include所指出的其它Makefile，並把其內容安置在當前的位置。就好像C/C++的#include指令一樣。如果文件都沒有指定絕對路徑或是相對路徑的話，make會在當前目錄下首先尋找，如果當前目錄下沒有找到，那麼，make還會在下面的幾個目錄下找：

1. 如果make執行時，有“-I”或“--include-dir”參數，那麼make就會在這個參數所指定的

目錄下去尋找。
2. 如果目錄<prefix>;/include（一般是：/usr/local/bin或/usr/include）存在的話，make也會去找。

如果有文件沒有找到的話，make會生成一條警告信息，但不會馬上出現致命錯誤。它會繼續載入其它的文件，一旦完成makefile的讀取， make會再重試這些沒有找到，或是不能讀取的文件，如果還是不行，make才會出現一條致命信息。如果你想讓make不理那些無法讀取的文件，而繼續執行，你可以在include前加一個減號“-”。如：

```
-include <filename>;
```

其表示，無論include過程中出現什麼錯誤，都不要報錯繼續執行。和其它版本make兼容的相關命令是sinclude，其作用和這一個是一樣的。

環境變量 MAKEFILES

如果你的當前環境中定義了環境變量MAKEFILES，那麼，make會把這個變量中的值做一個類似於include的動作。這個變量中的值是其它的Makefile，用空格分隔。只是，它和include不同的是，從這個環境變量中引入的Makefile的“目標”不會起作用，如果環境變量中定義的文件發現錯誤，make也會不理。

但是在這裏我還是建議不要使用這個環境變量，因為只要這個變量一被定義，那麼當你使用make時，所有的Makefile都會受到它的影響，這絕不是你想看到的。在這裏提這個事，只是為了告訴大家，也許有時候你的Makefile出現了怪事，那麼你可以看看當前環境中有沒有定義這個變量。

make的工作方式

GNU的make工作時的執行步驟入下：（想來其它的make也是類似）

1. 讀入所有的Makefile。
2. 讀入被include的其它Makefile。
3. 初始化文件中的變量。
4. 推導隱晦規則，並分析所有規則。
5. 為所有的目標文件創建依賴關係鏈。
6. 根據依賴關係，決定哪些目標要重新生成。
7. 執行生成命令。

1-5步為第一個階段，6-7為第二個階段。第一個階段中，如果定義的變量被使用了，那麼，make會把其展開在使用的位置。但make並不會完全馬上展開，make使用的是拖延戰術，如果變量出現在依賴關係的規則中，那麼僅當這條依賴被決定要使用了，變量才會在其內部展開。

當然，這個工作方式你不一定要清楚，但是知道這個方式你也會對make更為熟悉。有了這個基礎，後續部分也就容易看懂了。