linux下的C语言开发之Makefile编写

电子产品世界 2019-10-18

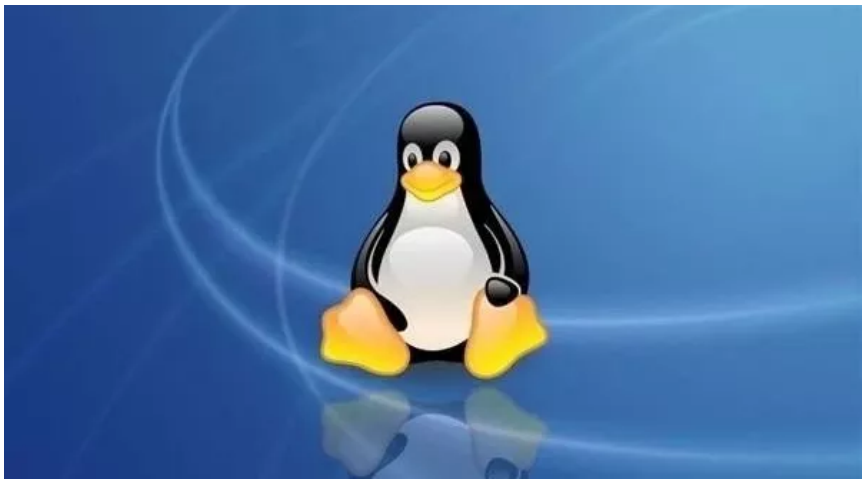
对于程序设计员来说，Makefile是我们绕不过去的一个坎。可能对于习惯Visual C++的用户来说，是否会编写Makefile无所谓。毕竟工具本身已经帮我们做好了全部的编译流程。但是在Linux上面，一切变得不一样了，没有人会为你做这一切。编代码要靠你，测试要靠你，最后自动化编译设计也要靠你自己。

Makefile介绍

首先，我们用一个示例来说明Makefile的书写规则，以便给大家一个感性认识。

我们的规则是：

a. 如果这个工程没有编译过，那么我们的所有C文件都要编译并被链接。
b. 如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。
c. 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。



只要我们的Makefile写得够好，所有的这一切，只用一个make命令就可以完成。make命令会自动、智能地根据当前文件修改的情况，来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

举例说明

代码示例

```
[cpp] view plaincopy
/* main.c */
#include “mytool1.h”
#include “mytool2.h”
int main(int argc，char **argv)
{
mytool1_print(“hello”);
```

```
mytool2_print(“hello”);
}
/* mytool1.h */
#ifndef _MYTOOL_1_H
#define _MYTOOL_1_H
void mytool1_print(char *print_str);
#endif
/* mytool1.c */
#include “mytool1.h”
void mytool1_print(char *print_str)
{
printf("This is mytool1 print %s ", print_str);
}
/* mytool2.h */
#ifndef _MYTOOL_2_H
#define _MYTOOL_2_H
void mytool2_print(char *print_str);
#endif
/* mytool2.c */
#include “mytool2.h”
void mytool2_print(char *print_str)
{
printf("This is mytool2 print %s ", print_str);
}
```

由于这个程序比较短，我们可以这样编译：

```
gcc -c main.c
gcc -c mytool1.c
gcc -c mytool2.c
gcc -o main main.o mytool1.o mytool2.o
```

这样的话我们也可以产生main程序，而且也不是很麻烦。

但是如果我们考虑一下如果有一天我们修改了其中的一个文件（比如说mytool1.c），那么我们难道还要重新输入上面的命令？

也许你会说，这个很容易解决啊，我写一个SHELL脚本，让它帮我去完成不就可以了。是的对于这个程序来说，是可以起到作用的。但是当我们把事情想的更复杂一点，如果我们的程序有几百个源程序的时候，难道也要编译器重新一个一个的去编译？

为此，聪明的程序员们想出了一个很好的工具来做这件事情，这就是make。我们只要执行以下make，就可以把上面的问题解决掉。在我们执行make之前，我们要先编写一个非常重要的文件——**Makefile**。

对于上面的那个程序来说，可能的一个Makefile文件是：

```
# 这是上面那个程序的Makefile文件
[plain] view plaincopy
main：main.o mytool1.o mytool2.o
gcc -o main main.o mytool1.o mytool2.o
main.o：main.c mytool1.h mytool2.h
gcc -c main.c
mytool1.o：mytool1.c mytool1.h
gcc -c mytool1.c
mytool2.o：mytool2.c mytool2.h
gcc -c mytool2.c
clean:
rm -rf *.o main
```

有了这个Makefile文件，不论我们什么时候修改了源程序当中的什么文件，只要执行make命令，我们的编译器都只会去编译和我们修改的文件有关的文件，其它的文件它连理都不想去理的。

那么，Makefile是如何编写的?

Makefile如何编写

在Makefile中“#”开始的行都是注释行。Makefile中最重要的是描述文件的依赖关系的说明。

一般的格式是：

```
target：components
TAB rule
```

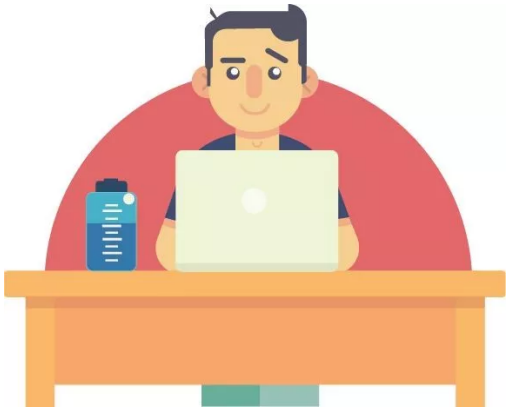
第一行表示的是依赖关系。第二行是规则。

例如上面的那个Makefile文件的前两行。

```
main：main.o mytool1.o mytool2.o
```

表示我们的目标(target)main的依赖对象(components)是main.o mytool1.omytool2.o。当倚赖的对象在目标修改后修改的话，就要去执行规则一行所指定的命令。就象我们的上面那个Makefile第二行所说的一样要执行 gcc-o main main.o mytool1.o mytool2.o 注意规则一行中的TAB表示那里是一个TAB键。

如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下“make clean”就可以了。



Makefile有三个非常有用的变量，分别是：“$@”“$^”“$<”。**它们代表的意义分别是：**

- “$@”代表目标文件。
- “$^”代表所有的依赖文件。
- “$<”代表第一个依赖文件。

如果我们使用上面三个变量，那么可以简化我们的Makefile文件为：

```
# 简化后的Makefile
[html] view plaincopy
main：main.o mytool1.o mytool2.o
gcc -o $@ $^
main.o：main.c mytool1.h mytool2.h
gcc -c $<
mytool1.o：mytool1.c mytool1.h
gcc -c $<
mytool2.o：mytool2.c mytool2.h
clean:
rm -rf *.o main
```

“gcc -c $<”经过简化后，我们的Makefile是简单了一点，不过人们有时候还想简单一点。

这里我们学习一个Makefile的缺省规则：

```
.c.o:
gcc -c $<
```

这个规则表示所有的 .o文件都是依赖与相应的.c文件的。例如mytool.o依赖于mytool.c，这样Makefile还可以变为：

```
# 这是再一次简化后的Makefile
main：main.o mytool1.o mytool2.o
gcc -o $@ $^
.c.o:
gcc -c $<
clean:
rm -rf *.o main
```

好了，我们的Makefile也讲得不多了。如果想知道更多关于Makefile的规则，可以查看相应的文档。

最后，总结下make执行过程：

a. make在当前目录下找 "Makefile"或"makefile"的文件。
b. 如果找到，则会找文件中第一个目标文件（target），如上个例子中的main。
c. 如果main命令的执行依赖后面命令执行所产生的文件，则先执行后面命令。
d. 当main命令需要的文件生成完毕，则执行main命令。