Linux环境下程序编译

曙光信息产业(北京)有限公司

提　纲

- 1. GCC编译
- 2. Make简介
- 3. 常用编译器

一个简单的例子—hello.c

- 用vi编写源文件:

```
#include <stdio.h>
void main()
{
    printf("hello world, I am \n");
}
```

- 用gcc编译

```
gcc hello.c
```

- 运行

```
./a.out
```

第二个例子

```
vim main.c
#include <stdio.h>
void main()
{
int sum=0,r;
for(i=1;i<=10;i++)
{
r=fx(i);
sum=sum+r;
}
printf("sum is %d \n",sum);
}
```

```
vim fx.c
#include <math.h>
#define N 4
int fx(int x)
{
int result;
result=pow(x,N);
}
```

```
gcc main.c fx.c
./a.out
```

```
gcc –o main main.c fx.c
./main
```

```
gcc –c main.c
gcc –c fx.c
gcc –o man main.o fx.o
```

GCC简介(1/2)

- gcc（GNU project C and C++ compiler）是GNU推出的功能强大、性能优越的C语言编译器，是GNU的代表作品之一。
- gcc编译器能将C、Fortran,C++语言源程序、汇程式化序和目标程序编译、连接成可执行文件，如果没有给出可执行文件的名字，gcc将生成一个名为a.out的文件。
- gcc最基本的用法是：

  gcc [options] [filenames]

  其中options就是编译器所需要的参数，filenames给出相关的文件名称

GCC简介(2/2)

- 在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。

- 而编译器则通过后缀来区别输入文件的类别
  - .c为后缀的文件，C语言源代码文件
  - .f为后缀的文件，Fortran77语言源代码文件
  - .f90为后缀的文件，Fortran90语言源代码文件
  - .C，.cc或.cxx 为后缀的文件，是C++源代码文件
  - .h为后缀的文件，是程序所包含的头文件
  - .o为后缀的文件，是编译后的目标文件，也是静态库文件
  - .so为后缀的文件，动态库文件
  - .a为后缀的文件，是由目标文件构成的静态链接库文件

GCC常用编译参数(1/3)

- -c：只编译，不连接成为可执行文件，编译器只是由输入的.c等源代码文件生成.o为后缀的目标文件，通常用于编译不包含主程序的子程序文件。
- -o output_filename：确定输出文件的名称为output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc就给出预设的可执行文件a.out。
- -g：产生符号调试工具(GNU的gdb)所必要的符号资讯，要使用gdb对源代码进行调试，我们就必须加入这个选项。
- -O：对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、连接的速度就相应地要慢一些。
- -O2：比-O更好的优化编译、连接，当然整个编译、连接过程会更慢。

GCC常用编译参数(2/3)

- -Idirname：将dirname所指出的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。C程序中的头文件包含两种情况：

  A)#include <stdio.h>

  B)#include “myinc.h”

  其中，A类使用尖括号(< >)，B类使用双引号(“ ”)。

对于A类，预处理程序cpp在系统预设包含文件目录(如/usr/include)中搜寻相应的文件，

对于B类，cpp在当前目录中搜寻头文件，这个选项的作用是告诉cpp，如果在当前目录中没有找到需要的文件，就到指定的dirname目录中去寻找。

在程序设计中，如果我们需要的这种包含文件分别分布在不同的目录中，就需要逐个使用-I选项给出搜索路径。

GCC常用编译参数(3/3)

- -Ldirname：将dirname所指出的目录加入到程序函数档案库文件的目录列表中，是在连接过程中使用的参数。在预设状态下，连接程序ld在系统的预设路径中(如/usr/lib)寻找所需要的档案库文件，这个选项告诉连接程序，首先到-L指定的目录中去寻找，然后到系统预设路径中寻找，如果函数库存放在多个目录下，就需要依次使用这个选项，给出相应的存放目录。

- -lname：在连接时，装载名字为“libname.a”的函数库，该函数库位于系统预设的目录或者由-L选项确定的目录下。例如，-lm表示连接名为“libm.a”的数学函数库,如果既有动态库，又有静态库，默认链接动态库。

上面我们简要介绍了gcc编译器最常用的功能和主要参数选项，更为详尽的资料可以参看Linux系统的联机帮助。

生成库文件

- 静态库
- ar -cr libmy.a my1.o my2.o my3.o
- 动态库
- gcc -shared libmy.so my1.o my2.o my3.o
- 查看可执行程序链接的动态库

GCC应用举例

1. gcc hello.c　　生成a.out
2. gcc –o hello hello.c　　生成hello
3. gcc –0 –o hello hello.c　　生成hello
4. gcc –02 –o hello hello.c　　生成hello
5. gcc –c hello.c　　生成hello.o
   gcc –o hello hello.o　　生成hello
6. gcc –c hello1.c　　生成hello1.o
   gcc –c hello2.c　　生成hello2.o
   gcc –o hello hello1.o hello2.o　　生成hello
7. gcc –o test test.o –lm –I/home/czn/include

Make简介

- 在开发大系统时，经常要将程序划分为许多模块。各个模块之间存在着各种各样的依赖关系，在Linux中通常使用 Makefile来管理。
  - 由于各个模块间不可避免存在关联，所以当一个模块改动后，其他模块也许会有所更新，当然对小系统来说，手工编译连接是没问题，但是如果是一个大系统，存在很多个模块，那么手工编译的方法就不适用了。
  - 为此，在Linux系统中，专门提供了一个make命令来自动维护目标文件。
  - 与手工编译和连接相比，make命令的优点在于他只更新修改过的文件，而对没修改的文件则置之不理，并且make命令不会漏掉一个需要更新的文件。

一个简单的例子

- 先举一个例子：a.c b.c两个程序

a.c

```
extern void p(char *);
main()
{
p("hello world");
}
```

b.c

```
void p(char *str)
{
  printf("%sn",str);
}
```

- Makefile

```
hello: a.c b.c
        gcc a.c b.c -o hello
```

←注意这里是一个Tab

- 执行make

```
gcc a.c b.c -o hello
```

产生一个叫hello的可执行程序

书写makefile文件

- Makefile时由规则来组成的,每一条规则都有三部分组成:目标(object),依赖(dependency)和命令(command).在上面的例子中， Makefile只有一条规则,其目标为hello,期依赖为a.c b.c,其命令为gcc a.c b.c -o hello.
- 依赖可以是另一条规则的目标,也可以是文件.每一条规则被这样处理.如目标是一个文件是:当它的依赖是文件时,如果依赖的时间比目标要新， 则运行规则所包含的命令来更新目标； 如果依赖是另一个目标则用同样的方法先来处理这个目标.如目标不是一个存在的文件时,则一定执行.

一个简单的makefile文件

- 例如：Makefile

```
hello: a.o b.o
    gcc a.o b.o -o hello
a.o: a.c
    gcc –c a.c
b.o: b.c
    gcc –c b.c
```

- 当运行make时，可以接一目标名(eg:make hello)作为参数，表示要处理改目标。如没有参数，则处理第一个目标。
- 对上述例子执行make，则是处理hello这个目标。
- hello依赖于文件目标a.o和b.o，则先去处理a.o,调用gcc –c a.c来更新a.o，之后更新b.o，最后调用gcc a.c b.o -o hello 来更新hello.

Make中的宏(macro)

- 在make中是用宏，要先定义，然后在makefile中引用。宏的定义格式为：

  宏名 = 宏的值　　(宏名一般习惯用大写字母)

  例：

```
CC = gcc
hello: a.o b.o
	$(CC) a.o b.o -o hello
a.o: a.c
	$(CC) –c a.c
b.o: b.c
	$(CC) –c b.c
```

系统定义的宏

- 还有一些设定好的内部变量，它们根据每一个规则内容定义。
  - $@　当前规则的目的文件名
  - $<　依靠列表中的第一个依靠文件
  - $^　整个依靠的列表（除掉了里面所有重复的文件名）。
  - $?　依赖中所有新于目标的
- 以用变量做许多其它的事情，特别是当你把它们和函数混合 使用的时候。如果需要更进一步的了解，请参考 GNU Make 手册。 ('man make', 'man makefile')

修改原先的makefile

```
CC       = gcc
CFLAGS   = -O2
OBJS     = a.o b.o
hello: $(OBJS)
   $(CC) $^ -o $@
a.o: a.c
   $(CC) $(CFLAGS) -c $<
b.o: b.c
   $(CC) $(CFLAGS) -c $<
clean:
   rm -f *.o hello
```

```
CC       = gcc
CFLAGS   = -O2
OBJS     = a.o b.o
hello: $(OBJS)
   $(CC) $^ -o $@
.c.o:
   $(CC) $(CFLAGS) -c $<
clean:
   rm -f *.o hello
```

隐含规则

- 请注意在上面的例子里，几个产生.o文件的命令都是一样的，都是从.c文件和相关文件里产生.o文件，这是一个标准的步骤。
- 其实make已经知道怎么做—它有一些叫做隐含规则的内置的规则，这些规则告诉它当你没有给出某些命令的时候，应该怎么办。
- 如果你把生成a.o和b.o的命令从它们的规则中删除，make将会查找它的隐含规则，然后会找到一个适当的命令。
- 它的命令会使用一些变量，因此你可以按照你的想法来设定它：它使用变量CC做为编译器，并且传递变量CFLAGS,CPPFLAGS,TARGET_ARCH，然后它加入‘-c’，后面跟变量$<，然后是‘-o’跟变量$@。一个C编译的具体命令将会是：

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

- 当然你可以按照你自己的需要来定义这些变量。

常用编译器

- GNU 编译器
- INTEL 编译器
- PGI 编译器

GNU 编译器

- 1 gcc C编译器
- 2 g++ C++编译器
- 3 g77   Fortran 77 编译器
- 4 gfortran Fortran 90 编译器

Intel 编译器

- 1 icc　C编译器
- 2 icpc　C++编译器
- 3 ifort　Fortran 77 编译器
- 4 ifort　Fortran 90 编译器

PGI 编译器

- 1 pgcc C编译器
- 2 pgCC C++编译器
- 3 pgf77 Fortran 77 编译器
- 4 pgf90 Fortran 90 编译器

谢谢！