Linux 之 GCC 经典入门教程

发表于: Linux, Tools, UNIX, 编程开发| 作者: 谋万世全局者
标签: GCC,Linux,入门教程,经典

准备工作

注意：本文可能会让你失望，如果你有下列疑问的话：为什么要在终端输命令啊？GCC 是什么东西，怎么在菜单中找不到？ GCC 不能有像 VC 那样的窗口吗？..... 那么你真正想要了解的可能是 anjuta，kdevelop，geany，code blocks，eclipse，netbeans 等 IDE 集成开发环境。即使在这种情况下，由于 GCC 是以上 IDE 的后台的编译器，本文仍值得你稍作了解。

如果你还没装编译环境或自己不确定装没装，不妨先执行

sudo apt-get install build-essential

如果你需要编译 Fortran 程序，那么还需要安装 gfortran(或 g77)

sudo apt-get install gfortran

编译简单的 C 程序

C 语言经典的入门例子是 **Hello World**，下面是一示例代码：

```
#include <stdio.h>
intmain(void)
{
printf(“Hello, world!n”);
return 0;
}
```

我们假定该代码存为文件‘hello.c’。要用 **gcc** 编译该文件，使用下面的命令：

$ gcc -g -Wall hello.c -o hello

该命令将文件‘hello.c’中的代码编译为机器码并存储在可执行文件 ‘hello’中。机器码的文件名是通过**-o** 选项指定的。该选项通常作为命令行中的最后一个参数。如果被省略，输出文件默认为 ‘a.out’。

注意到如果当前目录中与可执行文件重名的文件已经存在，它将被覆盖。

选项**-Wall** 开启编译器几乎所有常用的警告——**强烈建议你始终使用该选项**。编译器有很多其他的警告选项，但**-Wall** 是最常用的。默认情况下 GCC 不会产生任何警告信息。当编写 C 或 C++ 程序时编译器警告非常有助于检测程序存在的问题。**注意如果有用到 math.h 库等非 gcc 默认调用的标准库，请使用-lm 参数**

本例中，编译器使用了**-Wall** 选项而没产生任何警告，因为示例程序是完全合法的。

选项 “”-g”" 表示在生成的目标文件中带调试信息，调试信息可以在程序异常中止产生 core 后，帮助分析错误产生的源头，包括产生错误的文件名和行号等非常多有用的信息。

要运行该程序，输入可执行文件的路径如下：

```
$ ./hello
Hello, world!
```

这将可执行文件载入内存，并使 CPU 开始执行其包含的指令。路径**./**指代当前目录，因此**./hello** 载入并执行当前目录下的可执行文件 ‘hello’。

捕捉错误

如上所述，当用 C 或 C++ 编程时，编译器警告是非常重要的助手。为了说明这一点，下面的例子包含一个微妙的错误：为一个整数值错误地指定了一浮点数控制符 ‘%f’。

```
#include <stdio.h>

int main (void)
{
printf (“Two plus two is %fn”, 4);
return 0;
}
```

一眼看去该错误并不明显，但是它可被编译器捕捉到，只要启用了警告选项**-Wall**。

编译上面的程序‘bad.c’，将得到如下的消息：

```
$ gcc -Wall -o bad bad.c
main.c: 在函数‘main’中:
main.c:5: 警告：格式‘%f’需要类型‘double’，但实参 2 的类型为‘int’
```

这表明文件 ‘bad.c’第 6 行中的格式字符串用法不正确。GCC 的消息总是具有下面的格式文件名:行号:消息。编译器对错误与警告区别对待，前者将阻止编译，后者表明可能存在的问题但并不阻止程序编译。

本例中，对整数值来说，正确的格式控制符应该是**%d**。

如果不启用**-Wall**，程序表面看起来编译正常，但是会产生不正确的结果：

```
$ gccbad.c -o bad
$ ./bad
Two plus two is 0.000000
```

显而易见，开发程序时不检查警告是非常危险的。如果有函数使用不当，将可能导致程序崩溃或产生错误的结果。开启编译器警告选项**-Wall** 可捕捉 C 编程时的多数常见错误。

编译多个源文件

一个源程序可以分成几个文件。这样便于编辑与理解，尤其是程序非常大的时候。这也使各部分独立编译成为可能。

下面的例子中我们将程序 **Hello World** 分割成 3 个文件：‘hello.c’，‘hello_fn.c’和头文件‘hello.h’。这是主程序‘hello.c’：

```
#include “hello.h”
intmain(void)
{
hello (“world”);
return 0;
}
```

在先前例子的‘hello.c’中，我们调用的是库函数 **printf**，本例中我们用一个定义在文件‘hello_fn.c’中的函数 **hello** 取代它。

主程序中包含有头文件‘hello.h’，该头文件包含函数 **hello** 的声明。我们不需要在‘hello.c’文件中包含系统头文件‘stdio.h’来声明函数 **printf**，因为‘hello.c’没有直接调用 **printf**。

文件'hello.h'中的声明只用了一行就指定了函数 **hello** 的原型。

```
void hello (const char * name);
```

函数 **hello** 的定义在文件'hello_fn.c'中：

```
#include <stdio.h>
#include "hello.h"

void hello (const char * name)
{
printf ("Hello, %s!n", name);
}
```

语句**#include "FILE.h"**与**#include <FILE.h>**有所不同：前者在搜索系统头文件目录之前将先在当前目录中搜索文件'FILE.h'，后者只搜索系统头文件而不查看当前目录。

要用 **gcc** 编译以上源文件，使用下面的命令：

```
$ gcc -Wall hello.chello_fn.c -o newhello
```

本例中，我们使用选项**-o** 为可执行文件指定了一个不同的名字 **newhello**。注意到头文件'hello.h'并未在命令行中指定。源文件中的的**#include "hello.h"**指示符使得编译器自动将其包含到合适的位置。

要运行本程序，输入可执行文件的路径名：

```
$ ./newhello
Hello, world!
```

源程序各部分被编译为单一的可执行文件，它与我们先前的例子产生的结果相同。

简单的 Makefile 文件

为便于不熟悉 **make** 的读者理解，本节提供一个简单的用法示例。Make 凭借本身的优势，可在所有的 Unix 系统中被找到。要了解关于 Gnu make 的更多信息，请参考 Richard M. Stallman 和 Roland McGrath 编写的 **GNU Make** 手册。

Make 从**makefile**(默认是当前目录下的名为'Makefile'的文件)中读取项目的描述。makefile 指定了一系列目标（比如可执行文件）和依赖（比如对象文件和源文件）的编译规则，其格式如下：

目标：依赖
命令

对每一个目标，make 检查其对应的依赖文件修改时间来确定该目标是否需要利用对应的命令重新建立。注意到，makefile 中命令行必须以单个的 **TAB** 字符进行缩进，不能是空格。

GNU Make 包含许多默认的规则(参考隐含规则)来简化 makefile 的构建。比如说，它们指定'**.o**'文件可以通过编译'**.c**'文件得到，可执行文件可以通过将'**.o**'链接到一起获得。隐含规则通过被叫做 **make** 变量的东西所指定，比如 **CC**(C 语言编译器)和 **CFLAGS**(C 程序的编译选项)；在 makefile 文件中它们通过独占一行的**变量=值**的形式被设置。对 **C++**，其等价的变量是 **CXX** 和 **CXXFLAGS**，而变量 **CPPFLAGS** 则是编译预处理选项。

现在我们为上一节的项目写一个简单的 makefile 文件：

```
CC=gcc
CFLAGS=-Wall
hello: hello.ohello_fn.o
clean:
rm -f hello hello.ohello_fn.o
```

该文件可以这样来读：使用 C 语言编译器 **gcc**，和编译选项'-Wall',从对象文件'hello.o'和'hello_fn.o'生成目标可执行文件 **hello**（文件'hello.o'和'hello_fn.o'通过隐含规则分别由'hello.c'和'hello_fn.c'生成）。目标 **clean** 没有依赖文件，它只是简单地移除所有编译生成的文件。**rm** 命令的选项 '-f'(force) 抑制文件不存在时产生的错误消息。

另外，需要注意的是，如果包含 main 函数的 cpp 文件为 A.cpp, makefile 中最好把可执行文件名也写成 A。

要使用该 makefile 文件，输入 **make**。不加参数调用 make 时，makefile 文件中的第一个目标被建立，从而生成可执行文件'hello'：

```
$ make
gcc -Wall -c -o hello.ohello.c
```

```
gcc -Wall -c -o hello_fn.ohello_fn.c
gcchello.ohello_fn.o -o hello
$ ./hello
Hello, world!
```

一个源文件被修改要重新生成可执行文件，简单地再次输入 **make** 即可。通过检查目标文件和依赖文件的时间戳，程序 make 可识别哪些文件已经修改并依据对应的规则更新其对应的目标文件：

```
$ vim hello.c (打开编辑器修改一下文件)
$ make
gcc -Wall -c -o hello.ohello.c
gcchello.ohello_fn.o -o hello
$ ./hello
Hello, world!
```

最后，我们移除 make 生成的文件，输入 make clean：

```
$ make clean
rm -f hello hello.ohello_fn.o
```

一个专业的makefile文件通常包含用于安装(make install)和测试(make check)等额外的目标。

本文中涉及到的例子都足够简单以至于可以完全不需要 makefile，但是对任何大些的程序都使用 make 是很有必要的。

链接外部库

库是预编译的目标文件(object files)的集合，它们可被链接进程序。静态库以后缀为‘.a’的特殊的存档文件**(archive file)**存储。

标准系统库可在目录**/usr/lib** 与**/lib** 中找到。比如，在类 Unix 系统中 C 语言的数学库一般存储为文件**/usr/lib/libm.a**。该库中函数的原型声明在头文件**/usr/include/math.h** 中。C 标准库本身存储为**/usr/lib/libc.a**，它包含 ANSI/ISO C 标准指定的函数，比如‘printf’。对每一个 C 程序来说，libc.a 都默认被链接。

下面的是一个调用数学库 **libm.a** 中 **sin** 函数的的例子，创建文件 **calc.c**：

```
#include <math.h>
#include <stdio.h>

int main (void)
{
double x = 2.0;
double y = sin (x);
printf ("The value of sin(2.0) is %fn", y);
return 0;
}
```

尝试单独从该文件生成一个可执行文件将导致一个链接阶段的错误：

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function 'main':
/tmp/ccbR6Ojm.o(.text+0×19): undefined reference to 'sin'
```

函数 **sin**，未在本程序中定义也不在默认库'libc.a'中；除非被指定，编译器也不会链接'libm.a'。

为使编译器能将 **sin** 链接进主程序'calc.c'，我们需要提供数学库'libm.a'。一个容易想到但比较麻烦的做法是在命令行中显式地指定它：

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

函数库'libm.a'包含所有数学函数的目标文件，比如 **sin**,**cos**,**exp**,**log** 及 **sqrt**。链接器将搜索所有文件来找到包含 **sin** 的目标文件。

一旦包含 **sin** 的目标文件被找到，主程序就能被链接，一个完整的可执行文件就可生成了：

```
$ ./calc
The value of sin(2.0) is 0.909297
```

可执行文件包含主程序的机器码以及函数库'libm.a'中 **sin** 对应的机器码。

为避免在命令行中指定长长的路径，编译器为链接函数库提供了快捷的选项'-l'。例如，下面的命令

```
$ gcc -Wall calc.c -lm -o calc
```

与我们上面指定库全路径‘/usr/lib/libm.a’的命令等价。

一般来说，选项**-lNAME** 使链接器尝试链接系统库目录中的函数库文件 **lib**NAME**.a**。一个大型的程序通常要使用很多**-l** 选项来指定要链接的数学库，图形库，网络库等。

编译 C++与 Fortran

GCC 是 GNU 编译器集合（GNU Compiler Collection）的首字母缩写词。GNU 编译器集合包含 C，C++，Objective-C，Fortran，Java 和 Ada 的前端以及这些语言对应的库（libstdc++，libgcj，……）。

前面我们只涉及到 C 语言，那么如何用 gcc 编译其他语言呢？本节将简单介绍 C++ 和 Fortran 编译的例子。

首先我们尝试编译简单的 C++ 的经典程序 **Hello world**：

```
#include <iostream>
intmain(intargc,char *argv[])
{
std::cout<< “hello, world” <<std::endl;
return 0;
}
```

将文件保存为‘hello.cpp’，用 gcc 编译，结果如下：

```
$ gcc -Wall hello.cpp -o hello
/tmp/cch6oUy9.o: In function `__static_initialization_and_destruction_0(int,
int)’:
hello.cpp:(.text+0×23): undefined reference to `std::ios_base::Init::Init()’
/tmp/cch6oUy9.o: In function `__tcf_0′:
hello.cpp:(.text+0×6c): undefined reference to `std::ios_base::Init::~Init()’
/tmp/cch6oUy9.o: In function `main’:
hello.cpp:(.text+0×8e): undefined reference to `std::cout’
hello.cpp:(.text+0×93): undefined reference to `std::basic_ostream<char,
std::char_traits<char>>&std::operator<<<std::char_traits<char>>(std::basic_o
stream<char, std::char_traits<char>>&, char const*)’
/tmp/cch6oUy9.o:(.eh_frame+0×11): undefined reference to
`__gxx_personality_v0′
collect2: ld returned 1 exit status
```

出错了！！而且错误还很多，很难看懂，这可怎么办呢？在解释之前，我们先试试下面的命令：

$ gcc -Wall hello.cpp -o hello -lstdc++

噫，加上-lstdc++选项后，编译竟然通过了，而且没有任何警告。运行程序，结果如下：

$ ./hello
hello, world

通过上节，我们可以知道，-lstdc++ 选项用来通知链接器链接静态库 libstdc++.a。而从字面上可以看出，libstdc++.a 是 C++ 的标准库，这样一来，上面的问题我们就不难理解了——编译 C++ 程序，需要链接 C++ 的函数库 libstdc++.a。

编译 C 的时候我们不需要指定 C 的函数库，为什么 C++ 要指定呢？这是由于早期 gcc 是指 GNU 的 C 语言编译器（GNU C Compiler），随着 C++，Fortran 等语言的加入，gcc 的含义才变化成了 GNU 编译器集合（GNU Compiler Collection）。C 作为 gcc 的原生语言，故编译时不需额外的选项。

不过幸运的是，GCC 包含专门为 C++ 、Fortran 等语言的编译器前端。于是，上面的例子，我们可以直接用如下命令编译：

$ g++ -Wall hello.cpp -o hello

GCC 的 C++ 前端是 g++，而 Fortran 的情况则有点复杂：在 gcc-4.0 版本之前，Fortran 前端是 g77，而 gcc-4.0 之后的版本对应的 Fortran 前端则改为 gfortran。下面我们先写一个简单的 Fortran 示例程序：

C Fortran 示例程序
PROGRAM HELLOWORLD
WRITE(*,10)
10 FORMAT('hello, world')
END PROGRAM HELLOWORLD

将文件保存'hello.f'，用 GCC 的 Fortran 前端编译运行该文件

$ gfortran -Wall hello.f -o hello
$ ./hello
hello, world

我们已经知道，直接用 gcc 来编译 C++ 时，需要链接 C++ 标准库，那么用 gcc 编译 Fortran 时，命令该怎么写呢？

```
$ gcc -Wall hello.f -o helloworld -lgfortran -lgfortranbegin
```

注意：上面这条命令与 gfortran 前端是等价的（g77 与此稍有不同）。其中库文件 libgfortranbegin.a (通过命令行选项 -lgfortranbegin 被调用) 包含运行和终止一个 Fortran 程序所必须的开始和退出代码。库文件 libgfortran.a 包含 Fortran 底层的输入输出等所需要的运行函数。

对于 g77 来说，下面两条命令是等价的（注意到 g77 对应的 gcc 是 4.0 之前的版本）：

```
$ g77 -Wall hello.f -o hello
$ gcc-3.4 -Wall hello.f -o hello -lfrtbegin -lg2c
```

命令行中的两个库文件分别包含 Fortran 的开始和退出代码以及 Fortran 底层的运行函数。

本文翻译自 An Introduction to GCC 的部分章节（有改动）