Gcchowto

出自**Ubuntu**中文

本文翻譯自 *An Introduction to GCC* 的部分章節（有改動）。

目錄

準備工作

注意：本文可能會讓你失望，如果你有下列疑問的話：為什麼要在終端輸命令啊？ GCC 是什麼東西，怎麼在菜單中找不到？ GCC 不能有像 VC 那樣的窗口嗎？…… 那麼你真正想要了解的可能是 anjuta，kdevelop，geany，code blocks，eclipse，netbeans 等 IDE 集成開發環境。即使在這種情況下，由於 GCC 是以上 IDE 的後台的編譯器，本文仍值得你稍作了解。

如果你還沒裝編譯環境或自己不確定裝沒裝，不妨先執行

```
sudo apt-get install build-essential
```

如果你需要編譯 Fortran 程序，那麼還需要安裝 gfortran(或 g77)

```
sudo apt-get install gfortran
```

編譯簡單的 C 程序

C 語言經典的入門例子是 **Hello World**，下面是一示例代碼：

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

我們假定該代碼存為文件‘hello.c’。要用 **gcc** 編譯該文件，使用下面的命令：

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

該命令將文件‘hello.c’中的代碼編譯為機器碼並存儲在可執行文件 ‘hello’中。機器碼的文件名是通過 **-o** 選項指定的。該選項通常作為命令行中的最後一個參數。如果被省略，輸出文件默認為 ‘a.out’。

注意到如果當前目錄中與可執行文件重名的文件已經存在，它將被覆蓋。

選項 **-Wall** 開啟編譯器幾乎所有常用的警告——強烈建議你始終使用該選項。編譯器有很多其他的警告選項，但 **-Wall** 是最常用的。默認情況下GCC 不會產生任何警告信息。當編寫 C 或 C++ 程序時編譯器警告非常有助於檢測程序存在的問題。 注意如果有用到**math.h**庫等非**gcc**默認調用的標準庫，請使用**-lm**參數

本例中，編譯器使用了 *-Wall* 選項而沒產生任何警告，因為示例程序是完全合法的。

選項 ""-g"" 表示在生成的目標文件中帶調試信息，調試信息可以在程序異常中止產生core后，幫助分析錯誤產生的源頭，包括產生錯誤的文件名和行號等非常多有用的信息。

要運行該程序，輸入可執行文件的路徑如下：

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

這將可執行文件載入內存，並使 CPU 開始執行其包含的指令。 路徑 **./** 指代當前目錄，因此 **./hello** 載入並執行當前目錄下的可執行文件 ‘hello’。

點擊此處下載本節的操作視頻

捕捉錯誤

如上所述，當用 C 或 C++ 編程時，編譯器警告是非常重要的助手。為了說明這一點，下面的例子包含一個微妙的錯誤：為一個整數值錯誤地指定了一浮點數控制符‘%f’。

```
#include <stdio.h>

int main (void)
{
    printf ("Two plus two is  %f\n", 4);
    return 0;
}
```

一眼看去該錯誤並不明顯，但是它可被編譯器捕捉到，只要啟用了警告選項 **-Wall**。

編譯上面的程序‘bad.c’，將得到如下的消息：

```
$ 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’
```

這表明文件 ‘bad.c’第 6 行中的格式字符串用法不正確。GCC 的消息總是具有下面的格式 *文件名:行號:消息*。編譯器對錯誤與警告區別對待，前者將阻止編譯，後者表明可能存在的問題但並不阻止程序編譯。

本例中，對整數值來說，正確的格式控制符應該是 **%d**。

如果不啟用 **-Wall**，程序表面看起來編譯正常，但是會產生不正確的結果：

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

顯而易見，開發程序時不檢查警告是非常危險的。如果有函數使用不當，將可能導致程序崩潰或產生錯誤的結果。開啟編譯器警告選項 **-Wall** 可捕捉 C 編程時的多數常見錯誤。

編譯多個源文件

一個源程序可以分成幾個文件。這樣便於編輯與理解，尤其是程序非常大的時候。這也使各部分獨立編譯成為可能。

下面的例子中我們將程序 *Hello World* 分割成 3 個文件：‘hello.c’，‘hello_fn.c’和頭文件‘hello.h’。這是主程序‘hello.c’：

```
#include "hello.h"
int main(void)
{
    hello ("world");
    return 0;
}
```

在先前例子的‘hello.c’中，我們調用的是庫函數 **printf**，本例中我們用一個定義在文件‘hello_fn.c’中的函數 **hello** 取代它。

主程序中包含有頭文件‘hello.h’，該頭文件包含函數 **hello** 的聲明。我們不需要在‘hello.c’文件中包含系統頭文件‘stdio.h’來聲明函數 **printf**，因為‘hello.c’沒有直接調用 **printf**。

文件‘hello.h’中的聲明只用了一行就指定了函數 **hello** 的原型。

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

函數 **hello** 的定義在文件‘hello_fn.c’中：

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

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

語句 **#include "FILE.h"** 與 **#include <FILE.h>** 有所不同：前者在搜索系統頭文件目錄之前將先在當前目錄中搜索文件‘FILE.h’，後者只搜索系統頭文件而不查看當前目錄。

要用*gcc*編譯以上源文件，使用下面的命令：

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

本例中，我們使用選項 **-o** 為可執行文件指定了一個不同的名字 **newhello**。注意到頭文件‘hello.h’並未在命令行中指定。源文件中的的 **#include "hello.h"** 指示符使得編譯器自動將其包含到合適的位置。

要運行本程序，輸入可執行文件的路徑名：

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

源程序各部分被編譯為單一的可執行文件，它與我們先前的例子產生的結果相同。

點擊此處下載本節的操作視頻

簡單的 **Makefile** 文件

為便於不熟悉 **make** 的讀者理解，本節提供一個簡單的用法示例。Make 憑藉本身的優勢，可在所有的 Unix 系統中被找到。要了解關於Gnu make 的更多信息，請參考 Richard M. Stallman 和 Roland McGrath 編寫的 *GNU Make* 手冊。

Make 從 *makefile*(默認是當前目錄下的名為‘Makefile’的文件)中讀取項目的描述。makefile 指定了一系列*目標*（比如可執行文件）和*依賴*（比如對象文件和源文件）的編譯規則，其格式如下：

```
目标： 依赖
        命令
```

對每一個目標，make 檢查其對應的依賴文件修改時間來確定該目標是否需要利用對應的命令重新建立。注意到，makefile 中*命令*行必須以單個的 **TAB** 字符進行縮進，不能是空格。

GNU Make 包含許多默認的規則(參考*隱含*規則)來簡化 makefile 的構建。比如說，它們指定‘**.o**’文件可以通過編譯‘**.c**’文件得到，可執行文件可以通過將‘**.o**’鏈接到一起獲得。隱含規則通過被叫做*make變量*的東西所指定，比如 **CC**(C 語言編譯器)和 **CFLAGS**(C程序的編譯選項)；在makefile文件中它們通過獨佔一行的 變量=值 的形式被設置。對 **C++**，其等價的變量是**CXX**和**CXXFLAGS**，而變量**CPPFLAGS**則是編譯預處理選項。

現在我們為上一節的項目寫一個簡單的 makefile 文件：

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

該文件可以這樣來讀：使用 C 語言編譯器 **gcc**，和編譯選項‘-Wall’,從對象文件‘hello.o’和‘hello_fn.o’生成目標可執行文件 **hello**（文件‘hello.o’和‘hello_fn.o’通過隱含規則分別由‘hello.c’和‘hello_fn.c’生成）。目標**clean**沒有依賴文件，它只是簡單地移除所有編譯生成的文件。**rm**命令的選項 ‘-f’(force) 抑制文件不存在時產生的錯誤消息。

另外，需要注意的是，如果包含main函數的cpp文件為A.cpp, makefile中最好把可執行文件名也寫成 A。

要使用該 makefile 文件，輸入 **make**。不加參數調用make時，makefile文件中的第一個目標被建立，從而生成可執行文件‘hello’：

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

一個源文件被修改要重新生成可執行文件，簡單地再次輸入 **make** 即可。通過檢查目標文件和依賴文件的時間戳，程序 make 可識別哪些文件已經修改並依據對應的規則更新其對應的目標文件：

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

最後，我們移除 make 生成的文件，輸入 make clean：

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

一個專業的 makefile文件通常包含用於安裝(make install)和測試(make check)等額外的目標。

本文中涉及到的例子都足夠簡單以至於可以完全不需要makefile，但是對任何大些的程序都使用 make 是很有必要的。

鏈接外部庫

庫是預編譯的目標文件(object files)的集合，它們可被鏈接進程序。靜態庫以後綴為‘.a’的特殊的*存檔文件(archive file)*存儲。

標準系統庫可在目錄 **/usr/lib** 與 **/lib** 中找到。比如，在類 Unix 系統中 C 語言的數學庫一般存儲為文件 **/usr/lib/libm.a**。該庫中函數的原型聲明在頭文件 **/usr/include/math.h** 中。C 標準庫本身存儲為 **/usr/lib/libc.a**，它包含 ANSI/ISO C 標準指定的函數，比如‘printf’。對每一個 C 程序來說，libc.a 都默認被鏈接。

下面的是一個調用數學庫 **libm.a** 中 **sin** 函數的的例子，創建文件**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  %f\n", y);
    return 0;
}
```

嘗試單獨從該文件生成一個可執行文件將導致一個鏈接階段的錯誤：

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function 'main':
/tmp/ccbR6Ojm.o(.text+0x19): undefined reference to `sin'
```

函數 **sin**，未在本程序中定義也不在默認庫‘libc.a’中；除非被指定，編譯器也不會鏈接‘libm.a’。

為使編譯器能將 **sin** 鏈接進主程序‘calc.c’，我們需要提供數學庫‘libm.a’。一個容易想到但比較麻煩的做法是在命令行中顯式地指定它：

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

函數庫‘libm.a’包含所有數學函數的目標文件，比如**sin**,**cos**,**exp**,**log**及**sqrt**。鏈接器將搜索所有文件來找到包含 **sin** 的目標文件。

一旦包含 **sin** 的目標文件被找到，主程序就能被鏈接，一個完整的可執行文件就可生成了：

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

可執行文件包含主程序的機器碼以及函數庫‘libm.a’中 **sin** 對應的機器碼。

為避免在命令行中指定長長的路徑，編譯器為鏈接函數庫提供了快捷的選項‘-l’。例如，下面的命令

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

與我們上面指定庫全路徑‘/usr/lib/libm.a’的命令等價。

一般來說，選項 **-l***NAME*使鏈接器嘗試鏈接系統庫目錄中的函數庫文件 **lib***NAME*.**a**。一個大型的程序通常要使用很多 **-l** 選項來指定要鏈接的數學庫，圖形庫，網絡庫等。

編譯C++與Fortran

GCC 是 GNU 編譯器集合（GNU Compiler Collection）的首字母縮寫詞。GNU 編譯器集合包含 C，C++，Objective-C，Fortran，Java 和 Ada 的前端以及這些語言對應的庫（libstdc++，libgcj，……）。

前面我們只涉及到 C 語言，那麼如何用 gcc 編譯其他語言呢？本節將簡單介紹 C++ 和 Fortran 編譯的例子。

首先我們嘗試編譯簡單的 C++ 的經典程序 **Hello world**：

```
#include <iostream>
int main(int argc,char *argv[])
{
    std::cout << "hello, world" << std::endl;
    return 0;
}
```

將文件保存為‘hello.cpp’，用 gcc 編譯，結果如下：

```
$ gcc -Wall hello.cpp -o hello
/tmp/cch6oUy9.o: In function `__static_initialization_and_destruction_0(int, int)':
hello.cpp:(.text+0x23): undefined reference to `std::ios_base::Init::Init()'
/tmp/cch6oUy9.o: In function `__tcf_0':
hello.cpp:(.text+0x6c): undefined reference to `std::ios_base::Init::~Init()'
/tmp/cch6oUy9.o: In function `main':
hello.cpp:(.text+0x8e): undefined reference to `std::cout'
hello.cpp:(.text+0x93): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <
/tmp/cch6oUy9.o:(.eh_frame+0x11): undefined reference to `__gxx_personality_v0'
collect2: ld returned 1 exit status
```

出錯了！！而且錯誤還很多，很難看懂，這可怎麼辦呢？在解釋之前，我們先試試下面的命令：

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

噫，加上-lstdc++選項后，編譯竟然通過了，而且沒有任何警告。運行程序，結果如下：

```
$ ./hello
hello, world
```

通過上節，我們可以知道，-lstdc++ 選項用來通知鏈接器鏈接靜態庫 libstdc++.a。而從字面上

可以看出，libstdc++.a 是C++ 的標準庫，這樣一來，上面的問題我們就不難理解了——編譯 C++ 程序，需要鏈接 C++ 的函數庫 libstdc++.a。

編譯 C 的時候我們不需要指定 C 的函數庫，為什麼 C++ 要指定呢？這是由於早期 gcc 是指 GNU 的 C 語言編譯器（GNU C Compiler），隨着 C++，Fortran 等語言的加入，gcc的含義才變化成了 GNU 編譯器集合（GNU Compiler Collection）。C作為 gcc 的原生語言，故編譯時不需額外的選項。

不過幸運的是，GCC 包含專門為 C++ 、Fortran 等語言的編譯器前端。於是，上面的例子，我們可以直接用如下命令編譯：

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

GCC 的 C++ 前端是 g++，而 Fortran 的情況則有點複雜：在 gcc-4.0 版本之前，Fortran 前端是 g77，而gcc-4.0之後的版本對應的 Fortran 前端則改為 gfortran。下面我們先寫一個簡單的 Fortran 示例程序：

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

將文件保存‘hello.f’，用 GCC 的 Fortran 前端編譯運行該文件

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

我們已經知道，直接用 gcc 來編譯 C++ 時，需要鏈接 C++ 標準庫，那麼用 gcc 編譯 Fortran 時，命令該怎麼寫呢？

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

注意：上面這條命令與 gfortran 前端是等價的（g77 與此稍有不同）。其中庫文件 libgfortranbegin.a (通過命令行選項 -lgfortranbegin 被調用) 包含運行和終止一個 Fortran 程序所必須的開始和退出代碼。庫文件 libgfortran.a 包含 Fortran 底層的輸入輸出等所需要的運行函數。

對於 g77 來說，下面兩條命令是等價的（注意到 g77 對應的 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
```

命令行中的兩個庫文件分別包含 Fortran 的開始和退出代碼以及 Fortran 底層的運行函數。

其他參考

- **GCC**新手入門
- C編譯初步

- C/C++ IDE簡介
- 用GDB調試程序
- Gtk與Qt編譯環境安裝與配置
- 跟我一起寫Makefile
- C++編譯初步
- Fortran編譯初步
- C和C++混合編譯初步
- C和Fortran混合編譯初步