Linux vi/vim

所有的 Unix Like 系统都会内建 vi 文书编辑器，其他的文书编辑器则不一定会存在。

但是目前我们使用比较多的是 vim 编辑器。

vim 具有程序编辑的能力，可以主动的以字体颜色辨别语法的正确性，方便程序设计。

相关文章：[史上最全Vim快捷键键位图 — 入门到进阶]

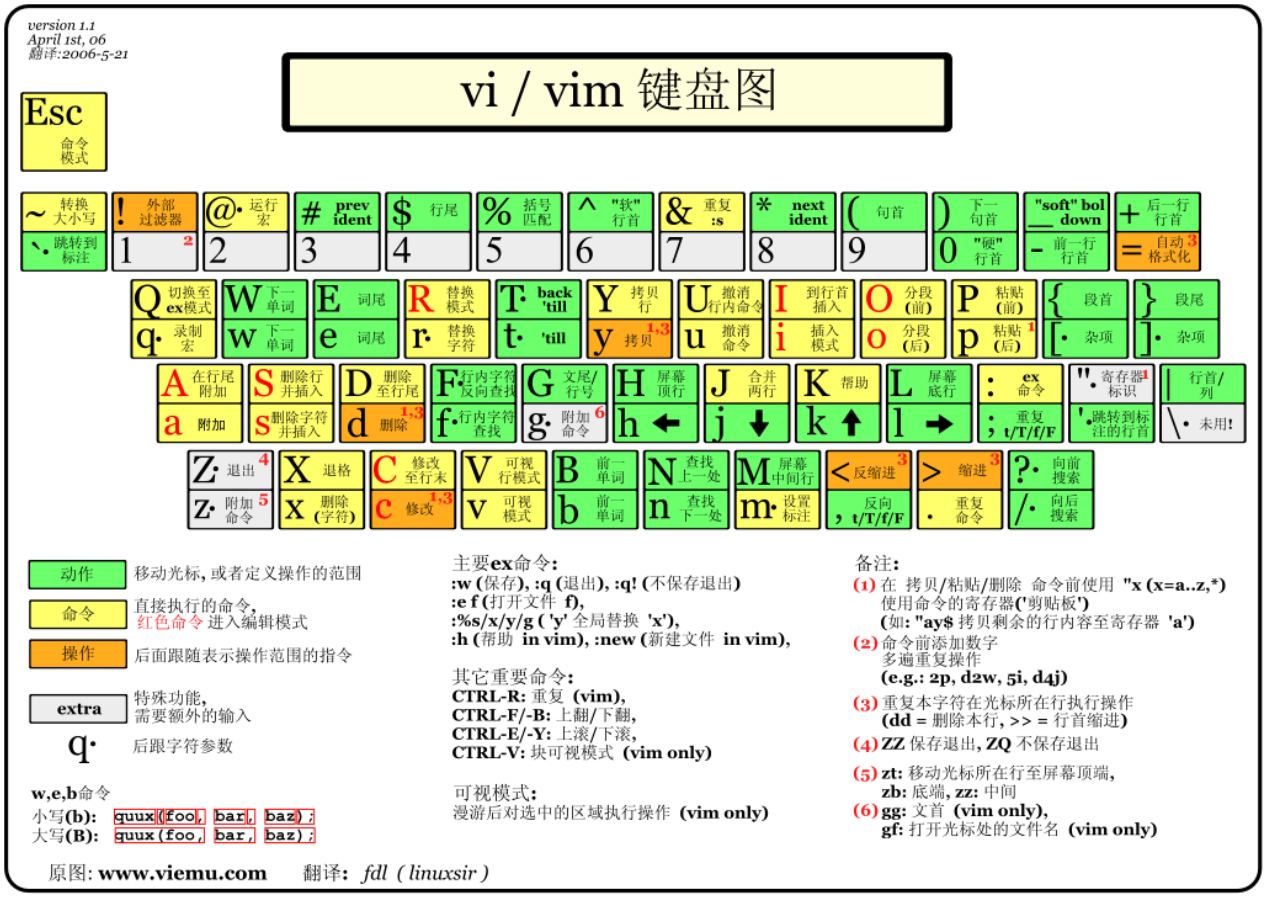
什么是 vim?

Vim是从 vi 发展出来的一个文本编辑器。代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

简单的来说， vi 是老式的字处理器，不过功能已经很齐全了，但是还是有可以进步的地方。 vim 则可以说是程序开发者的一项很好用的工具。

连 vim 的官方网站 (http://www.vim.org) 自己也说 vim 是一个程序开发工具而不是文字处理软件。

vim 键盘图：



vi/vim 的使用

基本上 vi/vim 共分为三种模式，分别是**命令模式（Command mode）**，**输入模式（Insert mode）**和**底线命令模式（Last line mode）**。 这三种模式的作用分别是：

命令模式：

用户刚刚启动 vi/vim，便进入了命令模式。

此状态下敲击键盘动作会被Vim识别为命令，而非输入字符。比如我们此时按下i，并不会输入一个字符，i被当作了一个命令。

以下是常用的几个命令：

- **i** 切换到输入模式，以输入字符。
- **x** 删除当前光标所在处的字符。
- **:** 切换到底线命令模式，以在最底一行输入命令。

若想要编辑文本：启动Vim，进入了命令模式，按下i，切换到输入模式。

命令模式只有一些最基本的命令，因此仍要依靠底线命令模式输入更多命令。

输入模式

在命令模式下按下i就进入了输入模式。

在输入模式中，可以使用以下按键：

- **字符按键以及Shift组合**，输入字符
- **ENTER**，回车键，换行
- **BACK SPACE**，退格键，删除光标前一个字符
- **DEL**，删除键，删除光标后一个字符
- **方向键**，在文本中移动光标
- **HOME**/**END**，移动光标到行首/行尾
- **Page Up**/**Page Down**，上/下翻页
- **Insert**，切换光标为输入/替换模式，光标将变成竖线/下划线
- **ESC**，退出输入模式，切换到命令模式

底线命令模式

在命令模式下按下:（英文冒号）就进入了底线命令模式。

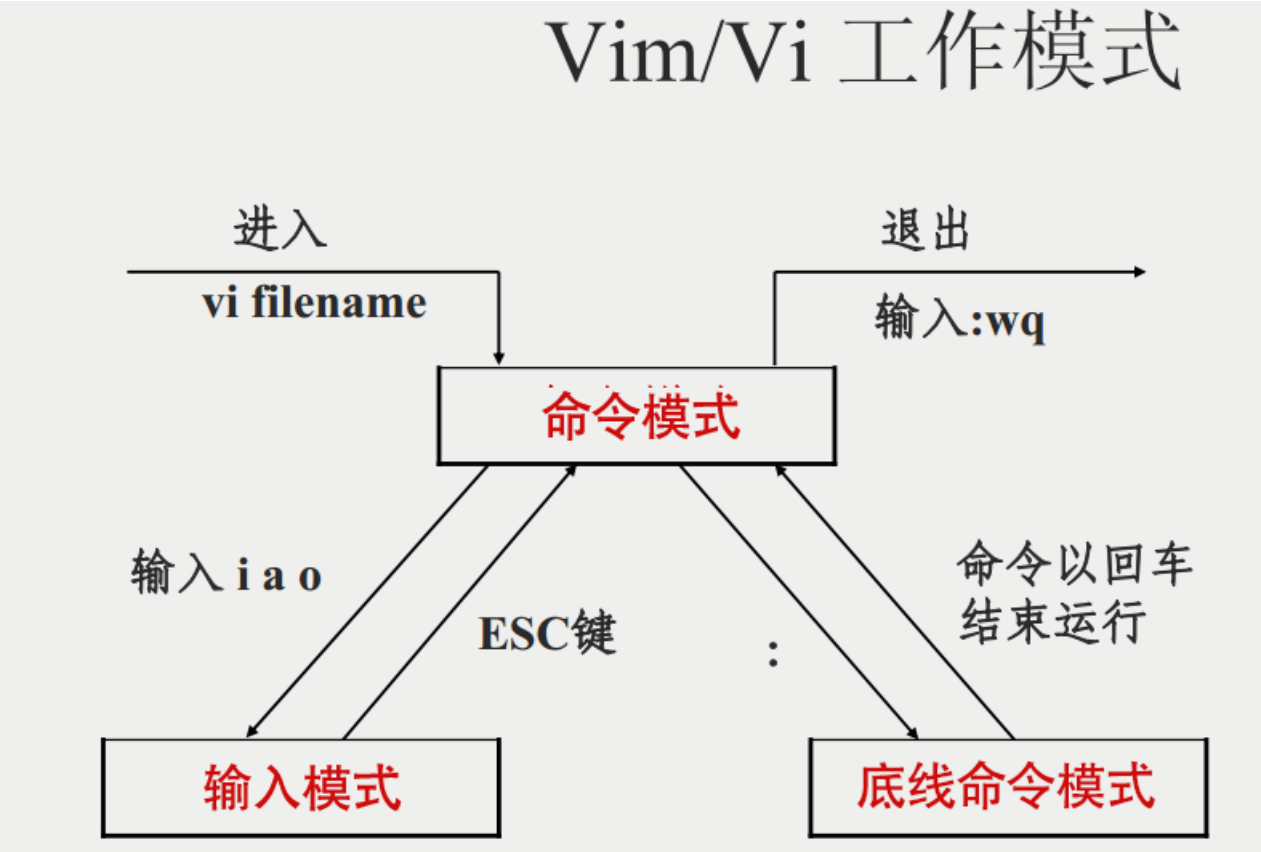
底线命令模式可以输入单个或多个字符的命令，可用的命令非常多。

在底线命令模式中，基本的命令有（已经省略了冒号）：

- q 退出程序
- w 保存文件

按ESC键可随时退出底线命令模式。

简单的说，我们可以将这三个模式想成底下的图标来表示：



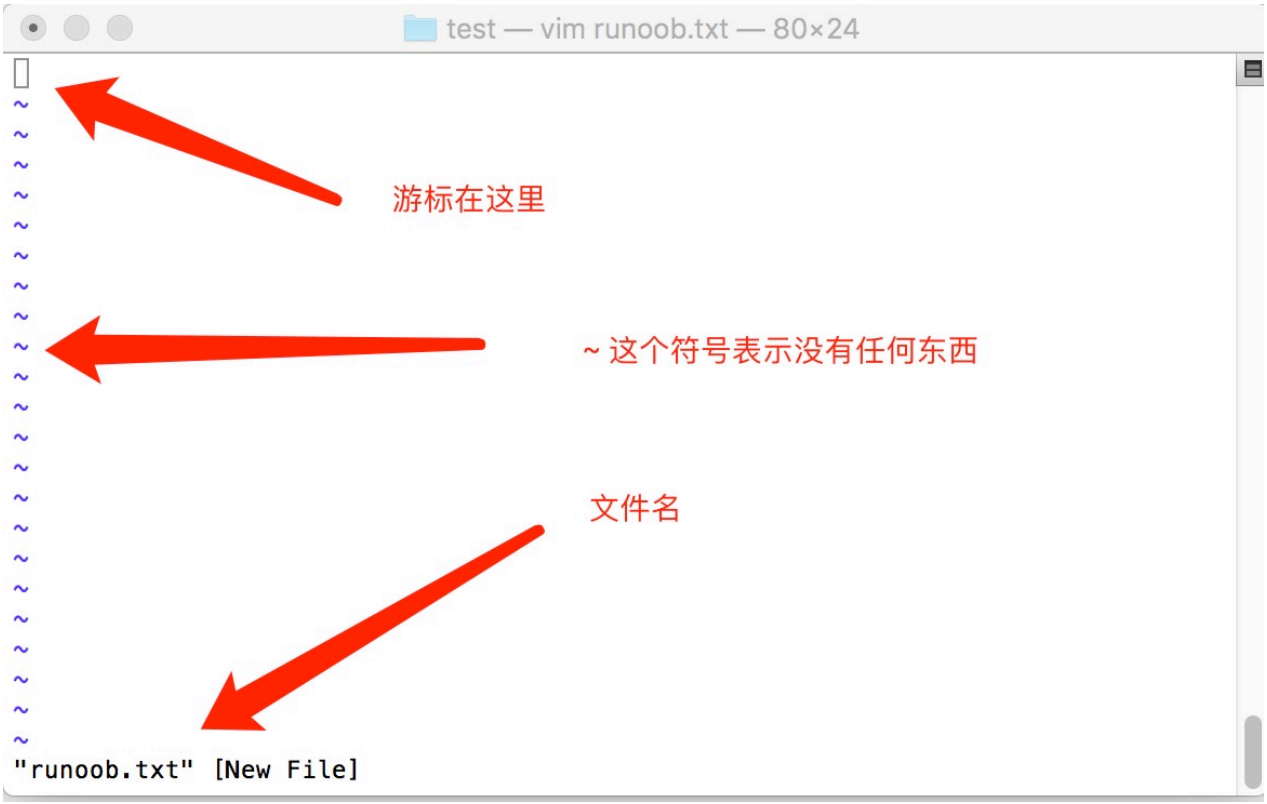
vi/vim 使用实例

使用 vi/vim 进入一般模式

如果你想要使用 vi 来建立一个名为 runoob.txt 的文件时，你可以这样做：

```
$ vim runoob.txt
```

直接输入 **vi 文件名** 就能够进入 vi 的一般模式了。请注意，记得 vi 后面一定要加文件名，不管该文件存在与否！



按下 i 进入输入模式(也称为编辑模式)，开始编辑文字

在一般模式之中，只要按下 i, o, a 等字符就可以进入输入模式了！

在编辑模式当中，你可以发现在左下角状态栏中会出现 –INSERT- 的字样，那就是可以输入任意字符的提示。

这个时候，键盘上除了 **Esc** 这个按键之外，其他的按键都可以视作为一般的输入按钮了，所以你可以进行任何的编辑。

按下 ESC 按钮回到一般模式

好了，假设我已经按照上面的样式给他编辑完毕了，那么应该要如何退出呢？是的！没错！就是给他按下 **Esc** 这个按钮即可！马上你就会发现画面左下角的 – INSERT – 不见了！

在一般模式中按下 :wq 储存后离开 vi

OK，我们要存档了，存盘并离开的指令很简单，输入 **:wq** 即可保存离开！

OK! 这样我们就成功创建了一个 runoob.txt 的文件。

---

vi/vim 按键说明

除了上面简易范例的 i, Esc, :wq 之外，其实 vim 还有非常多的按键可以使用。

第一部分：一般模式可用的光标移动、复制粘贴、搜索替换等

移动光标的方法	
h 或 向左箭头键(←)	光标向左移动一个字符
j 或 向下箭头键(↓)	光标向下移动一个字符
k 或 向上箭头键(↑)	光标向上移动一个字符
l 或 向右箭头键(→)	光标向右移动一个字符
如果你将右手放在键盘上的话，你会发现 hjkl 是排列在一起的，因此可以使用这四个	

按钮来移动光标。 如果想要进行多次移动的话，例如向下移动 30 行，可以使用 "30j" 或 "30↓" 的组合按键， 亦即加上想要进行的次数(数字)后，按下动作即可！	
[Ctrl] + [f]	屏幕『向下』移动一页，相当于 [Page Down]按键 (常用)
[Ctrl] + [b]	屏幕『向上』移动一页，相当于 [Page Up] 按键 (常用)
[Ctrl] + [d]	屏幕『向下』移动半页
[Ctrl] + [u]	屏幕『向上』移动半页
+	光标移动到非空格符的下一行
-	光标移动到非空格符的上一行
n	那个 n 表示『数字』，例如 20 。按下数字后再按空格键，光标会向右移动这一行的 n 个字符。例如 20 则光标会向后面移动 20 个字符距离。
0 或功能键[Home]	这是数字『 0 』： 移动到这一行的最前面字符处 (常用)
$ 或功能键[End]	移动到这一行的最后面字符处(常用)
H	光标移动到这个屏幕的最上方那一行的第一个字符
M	光标移动到这个屏幕的中央那一行的第一个字符
L	光标移动到这个屏幕的最下方那一行的第一个字符
G	移动到这个档案的最后一行(常用)
nG	n 为数字。移动到这个档案的第 n 行。例如 20G 则会移动到这个档案的第 20 行(可配合 :set nu)
gg	移动到这个档案的第一行，相当于 1G 啊！ (常用)
n	n 为数字。光标向下移动 n 行(常用)
搜索替换	
/word	向光标之下寻找一个名称为 word 的字符串。例如要在档案内搜寻 vbird 这个字符串，就输入 /vbird 即可！ (常用)
?word	向光标之上寻找一个字符串名称为 word 的字符串。

n	这个 n 是英文按键。代表重复前一个搜寻的动作。举例来说， 如果刚刚我们执行 /vbird 去向下搜寻 vbird 这个字符串，则按下 n 后，会向下继续搜寻下一个名称为 vbird 的字符串。如果是执行 ?vbird 的话，那么按下 n 则会向上继续搜寻名称为 vbird 的字符串！
N	这个 N 是英文按键。与 n 刚好相反，为『反向』进行前一个搜寻动作。 例如 /vbird 后，按下 N 则表示『向上』搜寻 vbird 。
使用 /word 配合 n 及 N 是非常有帮助的！可以让你重复的找到一些你搜寻的关键词！	
:n1,n2s/word1/word2/g	n1 与 n2 为数字。在第 n1 与 n2 行之间寻找 word1 这个字符串，并将该字符串取代为 word2 ！举例来说，在 100 到 200 行之间搜寻 vbird 并取代为 VBIRD 则：『:100,200s/vbird/VBIRD/g』。(常用)
:1,$s/word1/word2/g 或 **:%s/word1/word2/g**	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2 ！(常用)
:1,$s/word1/word2/gc 或 **:%s/word1/word2/gc**	从第一行到最后一行寻找 word1 字符串，并将该字符串取代为 word2 ！且在取代前显示提示字符给用户确认 (confirm) 是否需要取代！(常用)
删除、复制与贴上	
x, X	在一行字当中，x 为向后删除一个字符 (相当于 [del] 按键)， X 为向前删除一个字符(相当于 [backspace] 亦即是退格键) (常用)
nx	n 为数字，连续向后删除 n 个字符。举例来说，我要连续删除 10 个字符， 『10x』。
dd	删除游标所在的那一整行(常用)
ndd	n 为数字。删除光标所在的向下 n 行，例如 20dd 则是删除 20 行 (常用)
d1G	删除光标所在到第一行的所有数据
dG	删除光标所在到最后一行的所有数据
d$	删除游标所在处，到该行的最后一个字符
d0	那个是数字的 0 ，删除游标所在处，到该行的最前面一个字符
yy	复制游标所在的那一行(常用)

nyy	n 为数字。复制光标所在的向下 n 行，例如 20yy 则是复制 20 行(常用)
y1G	复制游标所在行到第一行的所有数据
yG	复制游标所在行到最后一行的所有数据
y0	复制光标所在的那个字符到该行行首的所有数据
y$	复制光标所在的那个字符到该行行尾的所有数据
p, P	p 为将已复制的数据在光标下一行贴上，P 则为贴在游标上一行！ 举例来说，我目前光标在第 20 行，且已经复制了 10 行数据。则按下 p 后， 那 10 行数据会贴在原本的 20 行之后，亦即由 21 行开始贴。但如果是按下 P 呢？ 那么原本的第 20 行会被推到变成 30 行。 (常用)
J	将光标所在行与下一行的数据结合成同一行
c	重复删除多个数据，例如向下删除 10 行，[10cj]
u	复原前一个动作。(常用)
[Ctrl]+r	重做上一个动作。(常用)
这个 u 与 [Ctrl]+r 是很常用的指令！一个是复原，另一个则是重做一次～ 利用这两个功能按键，你的编辑，嘿嘿！很快乐的啦！	
.	不要怀疑！这就是小数点！意思是重复前一个动作的意思。 如果你想要重复删除、重复贴上等等动作，按下小数点『.』就好了！ (常用)

第二部分：一般模式切换到编辑模式的可用的按钮说明

进入输入或取代的编辑模式	
i, I	进入输入模式(Insert mode)： i 为『从目前光标所在处输入』， I 为『在目前所在行的第一个非空格符处开始输入』。(常用)
a, A	进入输入模式(Insert mode)： a 为『从目前光标所在的下一个字符处开始输入』， A 为『从光标所在行的最后一个字符处开始输入』。(常用)
o, O	进入输入模式(Insert mode)： 这是英文字母 o 的大小写。o 为『在目前光标所在的下一行处输入新的一行』； O 为在目前光标所在处的上一行输入新的一行！(常用)
r, R	进入取代模式(Replace mode)： r 只会取代光标所在的那一个字符一次；R会一直取代光标所在的文字，直到按下 ESC 为止；(常用)
上面这些按键中，在 vi 画面的左下角处会出现『--INSERT--』或『--REPLACE--』的字样。 由名称就知道该动作了吧！！ 特别注意的是，我们上面也提过了，你想要在档案里面输入字符时， 一定要在左下角处看到 INSERT 或 REPLACE 才能输入喔！	
[Esc]	退出编辑模式，回到一般模式中(常用)

第三部分：一般模式切换到指令行模式的可用的按钮说明

指令行的储存、离开等指令	
:w	将编辑的数据写入硬盘档案中(常用)
:w!	若文件属性为『只读』时，强制写入该档案。不过，到底能不能写入， 还是跟你对该档案的档案权限有关啊！
:q	离开 vi (常用)
:q!	若曾修改过档案，又不想储存，使用 ! 为强制离开不储存档案。
注意一下啊，那个惊叹号 (!) 在 vi 当中，常常具有『强制』的意思～	
:wq	储存后离开，若为 :wq! 则为强制储存后离开 (常用)
ZZ	这是大写的 Z 喔！如果修改过，保存当前文件，然后退出！效果等同于(保存并退出)
ZQ	不保存，强制退出。效果等同于 **:q!**。
:w [filename]	将编辑的数据储存成另一个档案（类似另存新档）
:r [filename]	在编辑的数据中，读入另一个档案的数据。亦即将『filename』 这个档案内容加到游标所在行后面
:n1,n2 w [filename]	将 n1 到 n2 的内容储存成 filename 这个档案。
:! command	暂时离开 vi 到指令行模式下执行 command 的显示结果！例如『:! ls /home』即可在 vi 当中察看 /home 底下以 ls 输出的档案信息！
vim 环境的变更	
:set nu	显示行号，设定之后，会在每一行的前缀显示该行的行号
:set nonu	与 set nu 相反，为取消行号！

特别注意，在 vi/vim 中，数字是很有意义的！数字通常代表重复做几次的意思！ 也有可能是代表去到第几个什么什么的意思。

举例来说，要删除 50 行，则是用 『50dd』 对吧！ 数字加在动作之前，如我要向下移动 20 行呢？那就是『20j』或者是『20↓』即可。

GCC 编写 hello world

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>
int main(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’。

```c
#include <stdio.h>

int main (void)
{
printf (“Two plus two is %f\n”, 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：

```
$ gcc bad.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’：

```c
#include “hello.h”
int main(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.c hello_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.o hello_fn.o
clean:
rm -f hello hello.o hello_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.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.o hello.c
gcc hello.o hello_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.o hello_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 %f\n", 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+0x19): 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 **-l***NAME*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>
int main(int argc,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+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<< <std::char_traits<char> >
(std::basic_ostream<char, std::char_traits<char> >&, char const*)'
/tmp/cch6oUy9.o:(.eh_frame+0x11): 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）

来源：http://wiki.ubuntu.org.cn/