GCC 编程简介

for the GNU Compilers gcc and g++

作者：Brian Gough
译者：Walter Zhou
作序：Richard M. Stallman

序言

本序言由热心的 Richard M. Stallman 贡献，他是 GCC 的原始作者和 GNU 项目的奠基者。

本书是 GCC(GNU Compiler Collection，GNU 编译器集合)的入门教程，它将教会你怎样使用 GCC 这个编程工具。是的，GCC 是个编程工具，但它不仅止此。对计算机用户而言，它也是 20 年自由运动的一部分。

我们都想要好的软件，但对软件而言“好”意味着什么呢？简便的特征和可靠性是技术意义上的“好”，但这是不够的。好的软件必须在道德伦理上也是好的：它必须尊重用户的自由。

作为一个软件用户，你有权按你认为合适的方式运行它，有权研究源代码并按你认为合适的方式进行修改，有权发行软件的拷贝给其他人，有权公布修改版本以便贡献给建立中的自由软件社区。当一个程序用这种方式尊重你的自由时，我们称其为自由软件。在 GCC 出现以前，已经有了另外的 C，Fortran，Ada 等编译器，但它们都不是自由软件。你不能够自由地使用它们。我编写 GCC，以便我无需放弃我的自由就可以使用编译器。

仅仅有编译器对使用计算机系统而言是不够的，你需要整个操作系统。在 1983 年时，现代计算机的所有操作系统都不是自由的。为了有所补救，我于 1984 年开始开发 GNU 操作系统，一个属于自由软件的类 Unix 系统。开发 GCC 只是开发 GNU 项目的一部分。

到 90 年代早期，接近完成的 GNU 操作系统由于 Linux 内核的加入而完整了。Linux 是在 1992 年成为自由软件的。GNU/Linux 组合而成的操作系统使得达成这样一个目标成为可能：自由地使用计算机。但自由从来不会自动地保护你，我们需要工作来保卫它。自由软件运动需要你的支持。

Richard M. Stallman

2004 年 2 月

介绍

本书的目的是介绍GNU C和C++编译器（gcc和g++）的用法。阅读了本书后，你应该能学会怎样编译一个程序和怎样使用用于优化和调试的基本编译器选项。本书不会教你C或C++语言本身，因为这样的资料在许多地方都能找到（参见【进一步阅读】）。
熟悉其他系统但初次接触GNU编译器的有经验的程序员可以略过本章的前面部分【编译C程序】，【预处理】，【编译C++程序】。本章剩下的部分和其它章节应该能够为已经知道怎样使用其他编译器的程序员提供一个GCC特征的较好的概貌。

GCC 简史

GNU C 编译器（GCC）的原作者是 Richard Stallman，也是 GNU 项目的奠基者。
GNU 项目开始于 1984 年，为了提升自由和计算机用户与程序员间的合作，意图创建一个作为自由软件的完整的类 Unix 操作系统。每个类 Unix 操作系统都有 C 编译器，而当时还没有自由编译器存在，GNU 项目不得不白手起家开发一个。开发工作由来自个人和公司给予自由软件组织（Free Software Foundation）的捐献资助，该组织是一个非营利的机构，是为支持 GNU 项目的工作而建立的。
GCC 的第一版发行于 1987 年。作为以自由软件方式发行的第一个可移植的符合 ANSI C 标准的优化的编译器，这是一次重大突破。自此以后，GCC 就成为自由软件开发领域最重要的工具之一。
GCC 编译器的一个主要修订版是 1992 年的 2.0 系列版本，它加入了编译 C++的能力。1997 年，GCC 编译器的的一个实验性分支(EGCS)被传建，以改善优化和对 C++的支持。随着这些工作的进展，EGCS 被采纳为 GCC 开发版本的新主线，这些特征也在 2001 年的 GCC 3.0 发行版中被广泛采用。
随着时间的流逝，GCC被扩展以支持更多的语言，包括有Fortran，ADA，Java和Object-C。GCC这个缩略语现在是指“GNU Compiler Collection”[1]。它的开发由GCC指导委员会（GCC Steering Committe）管理。这是一个由业界的GCC用户社区，研究机构和学术界代表组成的组织。

GCC 主要特征

本节介绍一些 GCC 中的最重要的特性。
首先，GCC 是一个可移植的编译器----它能在今天绝大部分平台上运行，能为许多类型的 CPU 提供输出。除了在个人计算机方面用到的处理器外，它也支持微控制器，DSP，和 64 位 CPU。
GCC 不仅仅是个本地编译器----它还能够跨平台编译程序，即为不同于 GCC 本身所运行的系统生成可执行文件。这就可以为不能运行编译器的嵌入式系统编译软件。GCC 是用非常专注于可移植性的 C 语言写成的，它能对自身进行编译，所以它很容易被移植到新系统上。
GCC 有多种语言前端，用于解析不同的语言。对不同 CPU 架构不同语言的程序都能编译或跨平台编译。例如，为微控制器编译 ADA 程序，或者为超级计算机编译 C 程序。
GCC 是按模块化设计的，可以加入对新语言和新 CPU 架构的支持。给 GCC 加入新语言的前端

[1] 在以前是指“GNU C Compiler”(译者注)

就能使在新的架构上使用该语言，只要提供必需的运行期环境（比如库）。类似的，加入新架构的支持也使得所有语言同样支持该架构。
最后，也是最重要的，GCC 是自由软件，在 GNU General Public License(GNU GPL)保护下发行。这意味着你享有使用和修改 GCC 的自由，就像其他所有 GNU 软件一样。如果你需要支持一种新的 CPU，一种新语言，或一种新的特性，你可以自己添加它，或雇人来为你增强 GCC。如果 GCC 对你的工作很重要，你也可以雇人修补 bug。
此外，你有分享你增强的 GCC 功能的自由。这种自由的结果是你也能够利用因其它人开发而对 GCC 增强的功能。今天 GCC 提供的许多特性显示了这种自由是怎样使得协作以利于你和使用 GCC 的每一个人的。

C 和 C++编程

C 和 C++是两种允许直接访问计算机的内存的语言。历史上，它们被用于编写底层的系统软件和高性能的或控制资源很关键的应用程序。然而，需要特别当心以确保内存的正确访问，避免破坏其他数据结构。本书介绍的技巧有助于你在编译期间检测出潜在的错误，当然这种风险在使用象 C 或 C++这类语言时是不可避免的。
除了 C 和 C++，GNU 项目还提供了其他高级语言，比如象 GNU Common Lisp(gcl)，GNU Smalltalk(gst)，GNU Scheme 扩展语言（guile）和 GNU 的 Java 编译器（gcj)。这些语言不允许用户直接访问内存，避免了内存访问错误的可能性。对许多应用程序而言，它们是一种比 C 和 C++安全的选择。

本手册的排版约定

本书包含许多你可以通过键盘输入的例子。在终端上键入的命令象下面那样显示:

```
$ command
```

跟着的是输出。例如:

```
$ echo “hello world”
hello world
```

命令行上的第一个字符是终端提示符，不应当输入。美元符号“$”在本书中是标准提示符，而一些系统可能用其他符号。
如果例子中一行太长而不能放入单独一行上，则会折行并在下一行缩进，象这样:

```
$ echo "an example of a line which is too long to fit
    in this manual"
```

当在键盘上输入时，整行命令应当在一行上输入。
本书中用到的例子源代码文件可以从出版方的网站[2]下载，或者用任何文本编辑器手工输入，比如象标准的GNU编辑器，emacs。例子的编译命令用的是GNU C和C++的编译器gcc和g++，而cc指的是另外的编译器。例子程序应该可以在任何版本的GCC下编译。任何只有在最近的版本的GCC上才支持的命令行选项将在文中注明。
例子假设使用的是 GNU 操作系统----其他系统上的输出可能有稍许差异。例子中一些无关紧要的和冗长的与系统相关的输出信息（比如非常长的系统路径）会被删减。设置环境变量的命令用的是标准 GNU shell（bash）的语法，在任何版本的 Bourne shell 上应该也能工作。

[2] 见http://www.network-theory.co.uk/gcc/intro/

编译 C 程序

本章介绍了怎样用 gcc 来编译 C 程序。程序可以编译自单个源文件或多个源文件，还可以用到系统的库文件和头文件。
编译是指把一个是纯文本的源代码的“程序”，比如 C 或 C++这种编程语言，转变成机器码，即用于控制计算机的中央处理单元（CPU）的 1 和 0 的序列。这种机器码被存放在称为可执行文件的文件中，有时候也被称为二进制文件。

编译一个简单的 C 程序

Hello World 是 C 语言中的经典例子。下面是我们版本的该程序的源代码：

```
#include <stdio.h>

int
main (void)
{
   printf ("Hello, world!\n");
   return 0;
}
```

我们假设源代码被存放在“hello.c”的文件中。用 gcc 编译“hello.c”文件，可以用下面的命令：

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

这样就把在“hello.c”中的源代码编译成机器码并存储在可执行文件“hello”中。用“-o”选项可以指定存储机器码的输出文件，该选项通常是命令行上的最后一个参数。如果省略它，输出将被写到默认文件“a.out”中。
要注意的是，如果同名的可执行文件在当前目录下已经存在，则会被覆盖。
“-Wall”选项打开所有最常用到的编译警告----**推荐你总是使用该选项**！在后面的章节中我们将讨论许多另外的警告选项，但“-Wall”是最重要的。除非你打开使能这些选项，否则 GCC 输出任何警告。在用 C 和 C++编程时，编译器警告对检查错误是一种有用的帮助。
由于上面的例子程序是完全合法的，在这种情况下，即使带了“-Wall”选项编译器也不会输出任何警告。编译时不输出任何警告的源代码被称为“干净地编译”。
要运行该程序，象下面那样输入该可执行文件的路径名即可：

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

这会把可执行文件载入内存，CPU 开始执行包含其中的指令。路径中的./是指当前目录，所以./hello 载入并运行当前目录中的可执行文件“hello”。

在简单程序中找到错误

象上面提到的，使用 C 和 C++编程时，编译器警告是非常有用的帮助。为了演示这一点，下面的程序包含一个细微的错误：它不正确地调用 printf 函数，为一个整数指定的是浮点格式的“%f”：

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

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

这个错误在看第一眼时不太明显，但如果打开“-Wall”警告选项，就能被编译器检测到。
打开警告选项“-Wall”编译上面的程序“bad.c”会输出下面的信息：

```
$ gcc -Wall bad.c -o bad
bad.c: In function 'main':
bad.c:6: warning: double format, different
  type arg (arg 2)
```

这指示了在文件“bad.c”的第 6 行用到的格式化字符串不正确。GCC 输出的信息总是如file:line-number:message 这种形式。编译器区分报错信息和警告信息，不能成功编译的是报错信息，指示可能的错误的是警告（但并不停止程序的编译）。
在这里，正确的格式化指示符是“%d”（printf允许的格式化指示符可在任何一本讲解C的书上找到，比如GNU C库参考手册，见【进一步阅读】）。
如果不带警告选项“-Wall”编译，程序看上去干净地编译了，但输出不正确的结果：

```
$ gcc bad.c -o bad
$ ./bad
Two plus two is 2.585495 (incorrect output)
```

不正确的格式化指示符导致输出不对，因为 printf 函数传递的是整数而不是浮点数。整数和浮点数在内存中被用不同的格式存储，通常占有的字节数也不同，导致谬误的结果。上面显示的实际输出可能不一样，这依赖于特定的平台和环境。
无疑，开发程序而不检查编译器警告是很危险的。如果有函数没有被正确使用，那么它们会导致程序崩溃，或输出不正确的结果。打开编译器警告选项“-Wall”可以捕捉到许多在 C 编程中最常发生的错误。

编译多个源文件

一个程序可能被分成多个源文件，这样可以更容易编辑和理解，尤其是在大程序的情况下----它允许独立地编译各自的部分。
在下面的例子中，我们将把程序 Hello World 分成 3 个文件：“main.c”，“hello_fn.c”和头文件“hello.h”。这里是主程序“main.c”：

```
#include "hello.h"

int
main (void)
{
   hello ("world");
   return 0;
}
```

在前面的程序中对系统函数 printf 的调用被一个新的外部函数 hello 的调用取代，我们将把 hello 函数定义在单独的文件“hello_fn.c”中。

主程序也包括头文件“hello.h”，它将包含 hello 函数的声明。函数声明用于确保在函数调用和函数定义之间函数参数与返回值的类型能够正确匹配。我们不再需要在“main.c”文件中包括声明了 printf 函数的系统头文件“stdio.h”，因为“main.c”文件并没有直接调用 printf。

“hello.h”中的声明只是简单的一行，指定 hello 函数的原型：

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

hello 函数自身的定义被包含在“hello_fn.c”文件中：

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

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

该函数打印消息“Hello， *name*”，用函数参数作为这里 *name* 的值。

顺便说一下，#include “FILE.h”和“#include <FILE.h>”这两种 include 声明形式的含义是有差异的，前者是先在当前目录搜索“FILE.h”，然后再查看包含系统头文件的目录。#include <FILE.h>这种 include 声明是搜索系统目录的头文件，默认情况下不会在当前目录下查找头文件。

用下面的命令编译这些源码文件：

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

在这里，我们用“-o”选项来为可执行文件指定一个不同的输出文件“newhello”。注意，头文件“hello.h”不需要在命令行上的源文件名列表中指定。“hello.h”源码文件中的#include 指示符会指导编译器在合适的时候自动地包含它。

输入可执行文件的路径名来运行该程序：

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

程序中的所有部分已经被组合成单个的可执行文件，其象前面由单个源文件生成的可执行文件一样输出同样的结果。

独立地编译文件

如果整个程序代码被存储在单个源文件中，那么对某个函数的任何改变都需要改程序被重新编译以生成一个新的可执行文件。重新编译大的源码文件可能非常化时间。

当程序被存储在一个个单独的源文件中时，只有那些修改过源码的文件才需要重新编译。用这种方法，源文件被分开一个个编译，然后再链接在一起----分为两个步骤。在第一阶段，文件被编译但不生成可执行文件。生成的结果被称为对象文件（obj 文件），用 GCC 时有“.o”后缀名。在第二阶段，各个对象文件由一个被称为链接器的单独的程序合成在一起。链接器把所有的对象文件组合在一起生成单个的可执行文件。

对象文件包含的是机器码，其中任何对在其他文件中的函数（或变量）的内存地址的引用都留着没有被解析。这样就允许在互相之间不直接引用的情况下编译各个源代码文件。链接器在生成可执行文件时会填写这些还缺少的地址。

从源文件生成对象文件

命令行选项“-c”用于把源码文件编译成对象文件。例如，下面的命令将把源文件“main.c”编译成一个对象文件：

```
$ gcc -Wall -c main.c
```

这会生成一个包含 main 函数机器码的对象文件“main.o”。它包含一个对外部函数 hello 的引用，但在这个阶段该对象文件中的对应的内存地址留着没有被解析（它将在后面链接时被填写）。

编译源文件“hello_fn.c”中的 hello 函数的相应命令如下：

```
$ gcc -Wall -c hello_fn.c
```

这会生成对象文件“hello_fn.o”。

注意，在这里不需要用“-o”选项来指定输出文件的文件名。当用“-c”来编译时，编译器会自动生成与源文件同名，但用“.o”来代替原来的扩展名的对象文件。由于“main.c”和“hello_fn.c”中的#include 声明，hello.h 会自动被包括进来，所以在命令行上不需要指定该头文件。

从对象文件生成可执行文件

生成可执行文件的最后步骤是用 gcc 把各个对象文件链接在一起并缺失的外部函数的地址。要把对象文件链接在一起，只需把它们简单的列在命令行上即可：

```
$ gcc main.o hello_fn.o -o hello
```

这是几个很少需要用到“-Wall”警告选项的场合之一，因为每个源文件已经被成功地编译成对象文件了。一旦源文件被编译，链接是一个要么成功要么失败的明确的过程（只有在有引用不能解析的情况下才会链接失败）。

gcc使用链接器ld来施行链接，它是一个单独的程序。在GNU系统上用到的是GNU的链接器，即GNU ld。在另外的系统上可能也用GCC的GNU链接器，或者可能用它们自己的连接器。链接器将在后面讨论（见第 11 章【编译器怎么工作】）。通过运行连接器，gcc从对象文件生成可执行文件。

生成的可执行文件现在可以运行了：

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

它生成的输出与在前面章节中利用单个源文件编译的程序是一样的。

对象文件的链接次序

在类 Unix 系统上，传统上编译器和链接器搜索外部函数的次序是在命令行上指定的对象文件中从左到右的查找。这意味着包含函数定义的对象文件应当出现在调用这些函数的任何文件之后。由于是 main 调用 hello，在这种情况下，包含 hello 函数的文件“hello_fn.o”应该被放在“main.o”之后：

```
$ gcc main.o hello_fn.o -o hello (correct order)
```

如果次序搞反了，那么有些编译器或链接器会报错：

```
$ cc hello_fn.o main.o -o hello (incorrect order)
main.o: In function `main':
main.o(.text+0xf): undefined reference to `hello'
```

应为在“main.o”后面没有包含 hello 函数的对象文件。

当前绝大部分编译器和链接器会不管次序搜索所有的对象文件,但由于不是所有的编译器都这么做，最好遵守从左到右排序对象文件的惯例。
如果命令行上已经包括了所有必需的对象文件，但你还是碰到意料之外的未定义引用这种问题，那就因该想想这个问题。

重新编译和重新链接

为了展示源文件是怎样被独立地编译的，让我们来编辑主程序“main.c”，修改它以打印欢迎任何人的信息，而不是仅仅 world:

```
#include "hello.h"

int
main (void)
{
    hello ("everyone"); /* changed from "world" */
    return 0;
}
```

更新过的“main.c”文件现在能用下面的命令重新编译:

```
$ gcc -Wall -c main.c
```

这回生成一个新的对象文件“main.o”。由于“hello_fn.c”和及其相关的依赖文件，比如头文件，没有改变，所以没有必要为它生成新的对象文件。
新的对象文件可以和 hello 函数重新链接以生成新的可执行文件:

```
$ gcc main.o hello_fn.o -o hello
```

生成的可执行文件“hello”现在用的是新的 main 函数，输出下面的信息:

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

注意，只有“main.c”文件被重新编译，然后和含有hello函数的已有对象文件重新链接。相反，如果是“hello_fn.c”文件被修改，我们也只需要重新编译“hello_fn.c”以生成新的对象文件“hello_fn.o”，再和已有的“main.o”文件重新链接即可。[3]
通常，链接要快于编译----在一个有许多源文件的大型项目中，只重新编译那些被修改过的文件可以显著地节省时间。仅仅重编译项目中修改过的文件的过程可以用GNU Make来自动完成（见【进一步阅读】）。

与外部库文件链接

库是已经编译好并能被链接入程序的对象文件的集合。库中提供一些最常用的系统函数，比如象 C 的数学库中求平方根函数 sqrt。
库通常被存储在扩展名为“.a”的特殊归档文件中，被称为静态库。它们用一个单独的工具，GNU 归档器ar，从对象文件生成。在编译期，可以被链接器使用来解决对库中函数的引用。在后面我们会看到怎样用ar命令来生成库（见第 10 章【编译器相关工具】）。为了简化，本节只介绍静态库----使用共享库的运行期动态链接将在下一章介绍。

[3] 如果函数的原型也改变了，那需要修改和重编译所有用到它的其他源文件。

标准的系统库通常能在“/usr/lib”和“/lib”目录下找到[4]。比如，在类Unix系统上，C的数学库常被放在文件“/usr/lib/math.a”中，而该库中的相应的函数的原型声明在头文件“/usr/include/math.h”中。C标准库自身存放在“/usr/liblibc.a”中，包含ANSI/ISO C标准指定的各个函数，比如象“printf”----C库是每一个C程序默认都要被链接的库。

下面是调用数学库“libm.a”中的 sqrt 这个外部函数的一个例子：

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

int
main (void)
{
   double x = sqrt (2.0);
   printf ("The square root of 2.0 is %f\n", x);
   return 0;
}
```

试图只用该源文件就生成可执行文件会导致在链接阶段编译器包下面的错：

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR60jm.o: In function ‘main’:
/tmp/ccbR60jm.o(.text+0x19): undefined reference
to ‘sqrt’
```

问题是由于在没有外部数学库“libm.a”的情况下，对函数 sqrt 的引用被不能解决。函数 sqrt 并不定义在源程序中或默认的 C 库“libc.a”中，而且除非“libm.a”被显示指定，否则编译器不会链接该库文件。顺便说一下，在错误信息中提到的“/tmp/ccbR60jm.o”文件是由编译器为了执行链接而编译“calc.c”时生成的临时对象文件。

为了使得编译器能把 sqrt 函数链接到主程序“calc.c”，我们需要提供“libm.a”库。一个显然但麻烦的办法是在命令行上显示地指定该库文件：

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

“libm.a”库包含的是对象文件，里面是各种各样的数学函数，象 sin，cos，exp，log 和 sqrt。链接器搜索并能找到包含 sqrt 函数的对象文件。

一旦包含 sqrt 函数的对象文件被找到，主程序就可以被成功链接以生成可执行文件：

```
$ ./calc
The square root of 2.0 is 1.414214
```

该可执行文件包括 main 函数的机器码和从“libm.a”库中的相应对象文件中复制过来的 sqrt 函数的机器码。

为了避免在命令行上指定长路径名，编译器提供了短选项“-l”用于链接库文件。比如，下面的命令，

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

与上面原来在命令行上指定全路径的库名“/usr/lib/libm.a”是一样的。

通常，编译器选项“-l*NAME*”试图链接标准库目录下的文件名为“lib*NAME*.a”中的对象文件。另外的可以通过命令行和环境变量指定的目录会简短地讨论一下。在大型程序中通常会用到很多“-l”选项，来链接象数学库，图形库和网络库。

---

[4] 在 64 位和 32 位可执行文件都支持的系统上，64 位版本的库通常被存放在“/usr/lib64”和“/lib64”，而 32 位版本的则在“/usr/lib”和“/lib”。

库的链接次序

在命令行上的库的次序遵照象对象文件中的同样的惯例：它们被从左到右的搜索----包含函数定义的库应该出现在任何使用到该函数的源文件和对象文件之后。这包括用短选项“-l”指定的库，象下面命令所显示的：

```
$ gcc -Wall calc.c -lm -o calc (correct order)
```

如果次序搞反了（把“-lm”选项放到使用数学库中函数的文件前面），有些编译器会报错，

```
$ cc -Wall -lm calc.c -o calc (incorrect order)
main.o: In function `main':
main.o(.text+0xf): undefined reference to `sqrt'
```

这是因为在“calc.c”后面没有库或对象文件包含 sqrt 函数。“-lm”选项应当在文件“calc.c”后面。当用到多个库时，库之间也应当遵守同样的惯例。一个调用了定义在另外一个库中的函数的库应当被放在包含该函数代码的库的前面。

例如，程序“data.c”用到了 GNU Linear Programming 库“libglpk.a”，而该库又依次用到数学库“libm.a”，那么应当这么编译：

```
$ gcc -Wall data.c -lglpk -lm
```

因为“libglpk.a”中的对象文件用到了定义在“libm.a”中的函数。

对于对象文件，不管次序怎样，绝大部分编译器会搜索所有的库。然而，由于并不是所有的编译器都这么做，所以最好遵守从左到右对库排序的惯例。

使用库的头文件

使用库文件，为了得到函数参数和返回值正确类型的声明，必须包括入相应的头文件。如果没有函数声明，可能传递错误类型的函数参数，从而导致不对的结果。

下面的例子展示了另一个调用 C 库中函数的程序。在这里，函数 pow 用于计算 2 的立方：

```
#include <stdio.h>

int
main (void)
{
    double x = pow (2.0, 3.0);
    printf ("Two cubed is %f\n", x);
    return 0;
}
```

但该程序有个错误----缺少对“math.h”的#include 声明，这样头文件中的原型 double pow(double x, double y)不能被编译器看到。不带任何警告选项的编译该程序将生成一个输出错误结果的可执行文件：

```
$ gcc badpow.c -lm
$ ./a.out
Two cubed is 2.851120 (incorrect result, should be 8)
```

结果是错的，因为传递给pow调用的参数和返回值的类型都是错的。[5]该错在打开警告选项“-Wall”

[5] 上面实际的输出可能不一样，这依赖于特定平台和环境。

后能被检测到：

```
$ gcc -Wall badpow.c -lm
badpow.c: In function ‘main’:
badpow.c:6: warning: implicit declaration of
function ‘pow’
```

这个例子再次显示使用警告选项“-Wall”来检测很容易被忽略的严重问题的重要性。

编译选项

本章介绍 GCC 中的其他常用编译器选项。这些选项控制编译器的各种特征，如用于定位库和 include 文件的搜索路径，使用附加的警告和诊断，预处理宏和 C 语言的“方言”。

设置搜索路径

在上一章，我们看到怎样使用短选项“-lm”和头文件“math.h”把 C 数学库“libm.a”中的函数链接到程序中。

在编译用到库的程序时，常碰到的一个问题是报 include 的头文件有错误：

FILE.h: No such file or directory

如果头文件不在 GCC 用到的标准 include 文件路径中，就会报这样的错。对库而言，类似的问题如下：

/usr/bin/ld: cannot find library

如果链接时用到的库不在 GCC 用到的标准库目录中，就会报这样的错。

默认情况下，gcc 在下面目录中搜索头文件：

/usr/local/include/

/usr/include/

在下面目录中搜索库：

/usr/local/lib/

/usr/lib/

搜索头文件的目录列表常被称为 *include 路径*，而搜索库的目录列表被称为*搜索路径*或*链接路径*。在这些路径中的目录是按次序搜索的，在上面的两个列表中从第一个到最后一个[6]。例如，“/usr/local/include”中找到的头文件优先于“/usr/include”中的同名文件。类似的，“/usr/local/lib”中找到的库优先于“/usr/lib”中的同名库。

当有其他库被安装到另外的目录中，为了能按序找到这些库，需要扩展搜索路径。编译器选项“-I”和“-L”用于把新目录添加到各自的 include 路径和库搜索路径的头上。

搜索路径例子

下面的例子程序用到一个库，该库作为附加软件包可能已经被安装到系统中----GNU 数据管理库（GNU Database Management Library，GDBM）。GDBM 库在 DBM 文件中存储键-值对（key-value pair），DBM 文件是一种存储值（value），而用键（key，即任意序列的字符）来索引的文件。下面是一个例子程序，“dbmain.c”，它创建了一个 DBM 文件，包含“testkey”的键及对应的值“testvalue”：

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

[6] 默认的搜索路径也可以包括其他依赖于系统或指定位置的目录，和GCC在自身安装时的目录。例如，在64位平台上，附加的“lib64”目录也可以是默认被搜索的。

```
int
main (void)
{
    GDBM_FILE dbf;
    datum key = { "testkey", 7 }; /* key, length */
    datum value = { "testvalue", 9 }; /* value, length */

    printf ("Storing key-value pair... ");
    dbf = gdbm_open ("test", 0, GDBM_NEWDB, 0644, 0);
    gdbm_store (dbf, key, value, GDBM_INSERT);
    gdbm_close (dbf);
    printf ("done.\n");
    return 0;
}
```

该程序用到头文件“gdbm.h”和库“libgdbm.a”。如果库已经被安装到“/usr/local/lib”的默认位置，头文件也在“/usr/local/include”，那么该程序可以用下面的简单命令来编译：

```
$ gcc -Wall dbmain.c -lgdbm
```

这些目录都属于gcc默认的include路径和链接路径。

然而，如果GDBM被安装到不同位置，试图用同样命令编译该程序，会报下面的错：

```
$ gcc -Wall dbmain.c -lgdbm
dbmain.c:1: gdbm.h: No such file or directory
```

例如，如果1.8.3版本的GDBM软件包被安装在目录“/opt/gdbm-1.8.3”目录下，头文件的位置就会在

```
/opt/gdbm-1.8.3/include/gdbm.h
```

该目录不属于gcc默认include路径。用命令行选项“-I”添加相应目录到include路径中，这样该程序就能编译了，但还是链接失败：

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include dbmain.c -lgdbm
/usr/bin/ld: cannot find -lgdbm
collect2: ld returned 1 exit status
```

这是因为包含该库的目录还不在链接路径中。用下面的选项可以把库的路径添加到链接路径中去：

```
-L/opt/gdbm-1.8.3/lib/
```

下面的命令行可以使程序成功的编译和链接：

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
-L/opt/gdbm-1.8.3/lib dbmain.c -lgdbm
```

这就生成了链接到 GDBM 库的最终的可执行文件。在看怎样运行该可执行文件以前，让我们简略地看一下影响“-I”和“-L”选项的环境变量。

注意，你不应该在源代码中的#include 语句中放入头文件的绝对路径，因为这会让该程序不能在其他系统上编译。“-I”选项或下面就要介绍的 INCLUDE_PATH 变量用来设置头文件的 include 路径。

环境变量

通过 shell 中的环境可以控制头文件和库的搜索路径。可以在每次开始 shell 会话的相应登录文件

中，比如“.bash_profile”，自动地设置它们。

可以使用环境变量 C_INCLUDE_PATH（针对 C 的头文件）或 CPP_INCLUDE_PATH（针对 C++的头文件）把其他目录添加到 include 路径中。例如，当编译 C 程序时，下面的命令会把“/opt/gdbm-1.8.3/include”添加到 include 路径中：

```
$ C_INCLUDE_PATH=/opt/gdbm-1.8.3/include
$ export C_INCLUDE_PATH
```

该目录将在命令行上用选项“-I”指定的任何目录之后，但在标准默认目录“/usr/local/include”和“/usr/include”之前被搜索。Shell 命令 export 是必要的，以便 shell 以外的程序也能获得该环境变量，比如象编译器----对每一次 shell 会话的每一个变量来说，只需要设一次，也可以在相应的登录文件中设置。

类似的，使用环境变量 LIBRARY_PATH 可以把另外的目录添加到链接路径中去。例如，下面的命令会把“/opt/gdbm-1.8.3/lib”添加到链接路径中：

```
$ LIBRARY_PATH=/opt/gdbm-1.8.3/lib
$ export LIBRARY_PATH
```

该目录将在命令行上用选项“-L”指定的任何目录之后，但在标准默认目录“/usr/local/lib”和“/usr/lib”之前被搜索。

环境变量被设置好以后，上面的程序“dbmain.c”可以不用带“-I”和“-L”选项就能成功被编译，

```
$ gcc -Wall dbmain.c -lgdbm
```

因为现在默认路径包括了在环境变量 C_INCLUDE_PATH 和 LIBRARY_PATH 中指定的目录。

扩展搜索路径

遵循标准 Unix 搜索路径的规范，搜索目录可以在环境变量中用冒号分隔的列表形式一起指定：

```
DIR1:DIR2:DIR3:...
```

这些目录被依次从左到右搜索。单个点“.”可以用来指示当前目录。[7]

例如，下面的设置把安装软件包的当前目录，及在“/optgdbm-1.8.3”和“/net”目录下各自的“include”，“lib”目录放到默认的 include 和链接路径中去：

```
$ C_INCLUDE_PATH=.:/opt/gdbm-1.8.3/include:/net/include
$ LIBRARY_PATH=.:/opt/gdbm-1.8.3/lib:/net/lib
```

在命令行上可以重复使用“-I”和“-L”选项来指定多个搜索路径的目录。例如，下面的命令，

```
$ gcc -I. -I/opt/gdbm-1.8.3/include -I/net/include
-L. -L/opt/gdbm-1.8.3/lib -L/net/lib .....
```

等同于上面在环境变量中设置的。

当环境变量和命令行选项被同时使用时，编译器按下面的次序搜索目录：

1. 从左到右搜索由命令行“-I”和“-L”指定的目录
2. 由环境变量，比如 C_INCLUDE_PATH 和 LIBRARY_PATH，指定的目录
3. 默认的系统目录

在日常的使用情况中，通常用“-I”和“-L”选项把目录添加到搜索路径。

[7] 当前目录也可以用一个空的路径元素来指定。比如，*:DIR1:DIR2* 等同于 *.:DIR1:DIR2*。

共享库和静态库

虽然上面的例子程序被成功编译和链接，但生成的可执行文件要能被载入并运行，还缺少最后一步。

如果你试图直接启动该可执行文件，在绝大部分系统上将报下面的错：

```
$ ./a.out
./a.out: error while loading shared libraries:
libgdbm.so.3: cannot open shared object file:
No such file or directory
```

这是由于 GDBM 软件包提供的共享库的缘故。这种类型的库需要特殊对待----在可执行文件运行以前，它必须先从磁盘上被载入。

外部库通常用两种形式提供：静态库和共享库。静态库就是前面看到过的“.a”文件。当程序与一个静态库链接时，该程序用到的外部函数（在用到的静态库包含的对象文件中）的机器码被从库中复制到最终生成的可执行文件中。

处理共享库用的是一种更加高级的链接形式，它会使得可执行文件比较小。共享库使用“.so”后缀名，它代表*共享对象*（*shared object*）。

一个与共享库链接的可执行文件仅仅包含它用到的函数相关的一个表格，而不是外部函数所在的对象文件的整个机器码。在可执行文件开始运行以前，外部函数的机器码由操作系统从磁盘上的该共享库中复制到内存中----这个过程被称作*动态链接*（*dynamic linking*）。

因为一份库可以在多个程序间共享，所以动态链接使得可执行文件更小，也节省了磁盘空间。绝大部分操作系统提供了虚拟内存机制，该机制允许物理内存中的一份共享库被要用到该库的所有运行的程序共用，节省了内存和磁盘空间。

此外，共享库使得升级库而无需重新编译用到它的程序（只要库提供的接口不变就行）。

由于这些优点，如果可能，在绝大部分系统上 gcc 编译程序时默认链接到共享库。使用选项“-l*NAME*”的情况下，静态库“lib*NAME*”可以用于链接，但编译器首先会检查具有相同名字和“.so”为扩展名的共享库。

在上面的例子中，当编译器在链接路径中搜索“libgdbm”时，它在“/opt/gdbm-1.8.3/lib”目录中找到下面两个文件：

```
$ cd /opt/gdbm-1.8.3/lib
$ ls libgdbm.*
libgdbm.a libgdbm.so
```

结果是，“libgdbm.so”共享库优先于“libgdbm.a”静态库被使用。

然而，当启动可执行文件时，载入器为了把共享库载入内存，必须先找到它。默认情况下，载入器仅在一些预定义的系统目录中查找共享库，比如“/usr/local/lib”和“/usr/lib”。如果库不在这些目录中，那它必须被添加到载入路径（load path）中去。[8]

设置载入路径的最简单方法是通过环境变量 LD_LIBRARY_PATH。例如，下面的命令设置载入路径为“/opt/gdbm-1.8.3/lib”，以便载入器能够找到“libgdbm.so”：

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib
$ export LD_LIBRARY_PATH
$ ./a.out
Storing key-value pair... done.
```

[8] 注意，原则上包含有要用到的共享库的目录可以通过链接选项“-rpath”存储到可执行文件中去，但通常都不这样做。因为如果库被移走或该可执行文件被复制到另外的系统上，这会产生新的问题。

现在可执行文件成功运行，打印出消息并创建了一个被称为“test”的 DBM 文件，包含“testkey”和“testvalue”这对键-值对。

为了不要每次都输入，LD_LIBRARY_PATH 环境变量可以一次性的被设置到 shell 的相应登录文件中，象 GNU Bash shell 的“.bash_profile”文件。

多个共享库目录可以用冒号分隔列表 *DIR1:DIR2:DIR3:...:DIRN* 的形式放入载入路径中。例如下面的命令设置载入路径，用到了“/opt/gdbm-1.8.3”和“/opt/gtk-1.4”中的“lib”目录：

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
$ export LD_LIBRARY_PATH
```

如果载入路径包含已有项，可以用语法 LD_LIBRARY_PATH=*NEWDIRS*:$LD_LIBRARY_PATH 来扩展。例如，下面的命令象上面显示的那样把目录“/opt/gsl-1.5/lib”添加到载入路径中：

```
$ LD_LIBRARY_PATH=/opt/gsl-1.5/lib:$LD_LIBRARY_PATH
$ echo $LD_LIBRARY_PATH
/opt/gsl-1.5/lib:/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
```

系统管理员可以为所有用户设置 LD_LIBRARY_PATH 变量，只要把它添加到默认的登录脚本中，比如象“/etc/profile”。在 GNU 系统上，系统范围的路径也可以被定义在载入器配置文件“/etc/ld.so.conf”中。

相比较，使用“-static”选项可以迫使 gcc 静态链接，避免使用共享库：

```
$ gcc -Wall -static -I/opt/gdbm-1.8.3/include/
-L/opt/gdbm-1.8.3/lib/ dbmain.c -lgdbm
```

这就创建了一个与静态库“libgdbm.a”链接的可执行文件，它不需要设置环境变量 LD_LIBRARY_PATH 或把共享库存放在默认目录中就可以运行：

```
$ ./a.out
Storing key-value pair... done.
```

正像前面一样要注意的，通过在命令行上指定库的完整路径，直接与个别库文件链接也是可以的。例如，下面的命令将直接与静态库“libgdbm.a” 链接，

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
dbmain.c /opt/gdbm-1.8.3/lib/libgdbm.a
```

而下面的命令将与共享库文件“libgdbm.so”链接：

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
dbmain.c /opt/gdbm-1.8.3/lib/libgdbm.so
```

对于后者，要运行该可执行文件，还是需要设置该库的载入路径。

C 语言标准

默认情况下，gcc编译程序用的是C语言的GNU“方言”（一种C语言的GNU实现），被称为 *GNU C*。该实现集成了C语言官方ANSI/ISO标准和一些有用的GNU对C语言的扩展，比如内嵌函数和变长数组[9]。绝大部分符合ANSI/ISO的程序无需修改就能在GNU C下编译。

gcc提供了一些控制该种C的“方言”的选项，最常用的选项有“-ansi”和“-pedantic”。The specific dialects of the C language for each standard can also be selected with the ‘-std’ option。

[9] 这两者在标准中都是非法的----译者注。

ANSI/ISO

有时候，一个合法的 ANSI/ISO 程序可能并不兼容于 GNU C 中的扩展特性。为了处理这种情况，编译器选项“-ansi”禁止那些与 ANSI/ISO 标准冲突的 GNU 扩展特性。在使用 GNU C 库（glibc）的系统上，该选项也禁止了对 C 标准库的扩展。这样就允许那些基于 ANSI/ISO C 编写的程序在没有任何来自 GNU 扩展的情况下编译。

例如，下面是一个合法的 ANSI/ISO C 程序，它用到一个名为 asm 的变量：

```
#include <stdio.h>

int
main (void)
{
    const char asm[] = "6502";
    printf ("the string asm is '%s'\n", asm);
    return 0;
}
```

变量名 asm 在 ANSI/ISO 标准中是合法的，但 asm 是 GNU 扩展的关键字（它允许 C 函数中使用本地汇编指令），所以该程序在 GNU C 下不能编译。因此，它不能用作变量名，会产生编译错误：

```
$ gcc -Wall ansi.c
ansi.c: In function 'main':
ansi.c:6: parse error before 'asm'
ansi.c:7: parse error before 'asm'
```

相对应的，使用“-ansi”选项禁止 asm 这个扩展关键字后，上面的程序就能成功编译：

```
$ gcc -Wall -ansi ansi.c
$ ./a.out
the string asm is '6502'
```

为了便于参考，列出GNU C扩展特性定义的非标准关键字和宏asm，inline，typeof，unix和vax。更多细节可以在GCC参考手册“Using GCC”中找到（见【进一步阅读】）。

下面的例子展示了“-ansi”选项在使用GNU C库的系统（比如象GNU/Linux系统）上的效果。下面的程序打印pi的值，π = 3.14159...，预定义的M_PI来自头文件“math.h”：

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

int
main (void)
{
    printf("the value of pi is %f\n", M_PI);
    return 0;
}
```

常数 M_PI 并不是 ANSI/ISO C 标准库的一部分（它来自于 BSD 版本的 Unix）。在这种情况下，该程序不能用“-ansi”选项来编译：

```
$ gcc -Wall -ansi pi.c
pi.c: In function 'main':
```

```
pi.c:7: 'M_PI' undeclared (first use in this function)
pi.c:7: (Each undeclared identifier is reported only once
pi.c:7: for each function it appears in.)
```

该程序不使用“-ansi”选项就能编译。在这种情况下，对语言和库的扩展两者都被默认打开了：

```
$ gcc -Wall pi.c
$ ./a.out
the value of pi is 3.141593
```

编译程序时，只使用ANSI/ISO C标准，并且又用到GNU C库中的扩展特性是可以的。通过定义几个特殊的宏就可以做到，比如定义_GNU_SOURCE，它打开了对GNU C库中的扩展的支持：[10]

```
$ gcc -Wall -ansi -D_GNU_SOURCE pi.c
$ ./a.out
the value of pi is 3.141593
```

GNU C 库提供了一些这样的宏，用来控制对不同特性的支持（*参考特征测试宏*），比如 POSIX 扩展（_POSIX_C_SOURCE），BSD 扩展(_BSD_SOURCE), SVID 扩展(_SVID_SOURCE), XOPEN 扩展(_XOPEN_SOURCE)和 GNU 扩展(_GNU_SOURCE)。

_GNU_SOURCE宏打开所有的扩展，而POSIX扩展在这里如果与其他扩展有冲突，则优先于其他扩展。有关*特征测试宏*进一步信息可以参见GNU C库参考手册，见【进一步阅读】。

严格的 ANSI/ISO 标准

同时使用命令行选项“-pedantic”和“-ansi”会导致 gcc 拒绝所有的 GNU C 扩展，而不单单是那些不兼容于 ANSI/ISO 标准的。这有助于你写出遵循 ANSI/ISO 标准的可移植的程序。

下面是一个用到变长数组（一种 GNU C 扩展）的程序。数组 x[n]被声明成具有整数变量 n 指定的长度。

```
int
main (int argc, char *argv[])
{
    int i, n = argc;
    double x[n];

    for (i = 0; i < n; i++)
        x[i] = i;

    return 0;
}
```

由于支持变长数组并不会妨碍合法的 ANSI/ISO 程序的编译（这是一种向后兼容的扩展），该程序可以用“-ansi”编译：

```
$ gcc -Wall -ansi gnuarray.c
```

但是，如果用“-ansi -pedantic”来编译，就会报有关违反了 ANSI/ISO 标准的警告：

```
$ gcc -Wall -ansi -pedantic gnuarray.c
gnuarray.c: In function 'main':
gnuarray.c:5: warning: ISO C90 forbids variable-size
```

[10] 用来定义宏的“-D”选项将在下一章详细介绍。

```
array 'x'
```

注意，即使用“-ansi -pedantic”编译没有任何警告，也不能保证程序是严格遵循ANSI/ISO标准的。标准本身仅仅指定一套有限的应当产生诊断信息的circumstances，也就是“-ansi -pedantic”报的这些信息。

选择指定标准

可以用“-std”选项来控制 GCC 编译时采用的某个 C 语言标准。有下列 C 语言标准被支持:

‘-std=c89’ or ‘-std=iso9899:1990’
原来的 ANSI/ISO 语言标准（ANSI X3.159-1989，ISO/IEC 9899:1990）。GCC 把两份 ISO 技术勘误表中的修正集成到这两份标准中。

‘-std=iso9899:199409’
1994 年发布的 ISO C 语言标准的第一次修正版。该修正版主要涉及到国际化方面，比如为 C 库添加了多字节字符的支持。

‘-std=c99’ or ‘-std=iso9899:1999’
1999 年发布的修正过的 ISO C 语言标准（ISO/IEC 9899:1999）。

附带 GNU 扩展的 C 语言标准可以用选项“-std=gnu89”和“-std=gnu99”来选择。

-Wall 中的警告选项

象前面介绍的（见【编译一个简单的程序】），警告选项“-Wall”打开了针对许多常见错误的警告，所以最好编译时总使用该选项。它集成了很多的可以被单独指定的警告选项。下面是这些选项的概要:

‘-Wcomment’（包括在‘-Wall’内）
该选项对嵌套的注释发出警告。当包含注释的某段代码其后又被注释掉时就会产生嵌套的注释:

```
/* commented out
double x = 1.23 ; /* x-position */
*/
```

嵌套注释可能引起混乱 ---- 注释掉一段本身就包含注释的代码的安全方法是用预处理指示符 #if 0 ... #endif把它包起来:

```
/* commented out */
#if 0
double x = 1.23 ; /* x-position */
#endif
```

‘-Wformat’（包括在‘-Wall’内）
该选项警告象 printf 和 scanf 这种函数中格式化字符串的误用，即格式化字符串与对应的

函数参数的类型不一致。

‘-Wunused’（包括在‘-Wall’内）

该选项警告有没使用到的变量。当一个变量被声明但没有被使用时，可能是另一个变量意外地取代了它的位置。如果该变量确实不需要，那就从源代码中删除它。

‘-Wimplicit’（包括在‘-Wall’内）

该选项对任何没有声明就被使用的函数发出警告。没有声明就被使用函数的最常见原因是忘了包含该函数所在的头文件。

‘-Wreturn-type’（包括在‘-Wall’内）

该函数警告被定义的函数没有返回类型但并没有声明返回 void。它也能捕捉到返回值不是 void，但函数中返回空的 return 语句。

例如，下面的程序没有显式的返回值：

```
#include <stdio.h>

int
main (void)
{
    printf ("hello world\n");
    return;
}
```

上面代码中缺少返回值可能是程序员意外省略的结果----main 函数返回的值实际上是 printf 函数的返回值（即打印出的字符数）。为了避免暧昧不清，最好在 return 语句中返回显式的值，即或者是某个变量，或者是某个常数，比如象 return 0。

“-Wall”中包括有哪些警告选项在GCC参考手册“Using GCC”中都能找到（见【进一步阅读】）。“-Wall”中包含的选项有个普遍的特性，它们报告总是有问题的代码构造，或是很容易用明白无误的方法改写的错。这就是为什么它们如此有用----“-Wall”产生的任何警告可以被看作是有潜在严重问题的指示。

其他警告选项

GCC 还提供了许多其他警告选项，他们并没有包括在“-Wall”之内，但也很有用。典型的，这些选项对那些从技术上讲合法，但可能导致问题的代码产生警告。GCC 提供这些选项是基于程序员所犯常见错误的经验----它们没有被包括在“-Wall”是因为它们仅意味着可能有错误或是代码比较可疑。

由于对合法代码也可能报警，所以在编译时并不是总用到这些选项。周期性地使用这些选项并查看输出的结果，检查任何意外的输出，或仅在某些程序和文件中打开它们，可能更合适一点。

‘-W’

这是一个类似“-Wall”的通用选项，它对 a selection of 常见编程错误产生警告，比如象需要返回值但又没有返回值的函数（也被称作“falling off the end of the function body”），以及对有符号数与无符号数进行比较。例如，下面的函数测试一个无符号数是否是负数（当

然这是不可能的):

```
int
foo (unsigned int x)
{
if (x < 0)
    return 0; /* cannot occur */
else
    return 1;
}
```

用“-Wall”来编译该函数不会输出警告信息,

```
$ gcc -Wall -c w.c
```

但用“-W”编译时,报警了:

```
$ gcc -W -c w.c
w.c: In function ‘foo’:
w.c:4: warning: comparison of unsigned
expression < 0 is always false
```

实际上,“-W”和“-Wall”选项通常同时使用。

‘-Wconversion’

该选项警告可能引起意外结果的隐式类型转换。例如,象下面代码中的把一个负数赋值给一个无符号变量,

```
unsigned int x = -1;
```

在技术上,ANSI/ISO C 标准是允许这样做的(负整数被转换成一个正整数,因机器而不同),但这也可能是一个简单的编程错误。如果你需要实施这样一种转换,你可以用显式转换,比如象((unsigned int) -1),以避免来自该选项的警告。在 2 的补码的机器上,上面转换的结果是赋给一个无符号整数所能表示的最大值。

‘-Wshadow’

该选项用来警告这样一种情况,在定义过某个变量的代码范围内再次定义一个同名的变量。这被称作变量*遮蔽*,导致在变量出现的地方搞不清其对应哪个值。

下面的函数声明的一个局部变量 y,它遮蔽了该函数体内的另一个变量 y 的声明:

```
double
test (double x)
{
    double y = 1.0;
    {
        double y;
        y = x;
    }
    return y;
}
```

这是合法的 ANSI/ISO C 程序,它的返回值是 1。当看到 y = x 这一行时(尤其在大的复杂的函数中),对变量 y 的遮蔽可能使得它看上去返回值是 x(不正确)。

函数名也可能被遮蔽。例如,下面的程序试图定义一个会遮蔽标准函数 sin(x)的变量 sin。

```
double
sin_series (double x)
{
    /* series expansion for small x */
    double sin = x * (1.0 - x * x / 6.0);
    return sin;
}
```

‘-Wshadow’选项可以检测出这个错。

‘-Wcast-qual’

该选项警告对指针的转换操作移除了某种类型修饰符，比如象 const。例如，下面的函数丢弃了来自输入参数的 const 修饰符，从而允许覆盖指针所指向的空间：

```
void
f (const char * str)
{
    char * s = (char *)str;
    s[0] = '\0';
}
```

对 str 指向的原始内容的改写是违反它的 const 性质的。该选项将警告允许对 str 变量所指向的字符串做修改的不合适的类型转换。

‘-Wwrite-strings’

该选项隐含的使得定义在程序中的所以字符串常量都带有 const 修饰符，导致如果有代码试图覆写这些字符串，就会在编译时报警。修改一个字符串常量会导致什么结果，ANSI/ISO 标准并没有说明。在 GCC 中是使用可写的字符串常量是会报警的（译者注：程序员要把网硬上弓也行，但编译器会抱怨，以后甚至会禁止这样做）。

‘-Wtraditional’

该选项对那些在 ANSI/ISO 编译器下和在 ANSI 之前的“传统”编译器下编译方式不同的代码进行警告。当维护老的原有的软件时，对于该选项产生的警告，可能需要调查原始代码是接受传统编译呢还是接受 ANSI/ISO 标准编译。

上面的选项会生成诊断性的警告信息，但允许编译过程继续并生成对象文件或可执行文件。
对于大型程序，可能只要有警告信息产生，就停止编译。以便捕捉所有警告。“-Werror”选项通过把警告转变成错误，改变了编译器的默认行为，即只要有警告产生就停止编译。

预处理

本章介绍GNU C预处理器cpp，它是整个GCC的一部分。预处理器在源文件被编译以前展开其中的宏。它是由GCC在处理C或C++程序时被自动调用的。[11]

定义宏

下面的程序演示了 C 预处理器最常用的用法，它用预处理器条件命令#ifdef 来检查某个宏是否定义了。
当该宏被定义时，预处理器把直到#endif 命令的相应代码包括入源文件。在该例中，被检查的宏被称为 TEST，源代码中的条件部分是一行打印消息“Test mode”的 printf 声明：

```
#include <stdio.h>

int
main (void)
{
#ifdef TEST
   printf ("Test mode\n");
#endif
   printf ("Running...\n");
   return 0;
}
```

gcc 的“-D*NAME*”选项在命令行上定义预处理宏 *NAME*。如果上面的程序用带“-DTEST”命令行选项来编译，TEST 宏将被定义，生成的可执行文件会打印两行消息：

```
$ gcc -Wall -DTEST dtest.c
$ ./a.out
Test mode
Running...
```

如果同样的程序不带“-D”选项来编译，则在经过预处理后，“Test mode”消息将从源代码中略掉，最后的可执行文件不会包括该行代码：

```
$ gcc -Wall dtest.c
$ ./a.out
Running...
```

宏通常都是未定义的除非用“-D”选项在命令行上指定，或在源文件中（或库的头文件）用#define 定义。有些宏是由编译器自动定义的----这些宏会用到由双下划线（__）开始的保留的名字空间。预定义的宏的完整列表可以这样得到，对某个空文件运行带“-dM”选项的 GNU 预处理器 cpp：

```
$ cpp -dM /dev/null
#define __i386__ 1
#define __i386 1
```

[11] 在最近的GCC版本中，预处理器已经被集成进编译器，虽然单独的cpp命令也还提供。

```
#define i386 1
#define __unix 1
#define __unix__ 1
#define __ELF__ 1
#define unix 1
.......
```

注意，这个列表中包括了较少数目的由 gcc 定义的特定于系统的宏，它们并不是以双下划线开始。这些非标准宏可以用 gcc 的“-ansi”选项来禁止掉。

给宏赋值

除了定义宏，在 GCC 中你可以给宏赋值，这些值将被插入到源代码中宏所在的地方。下面的程序用到 NUM 这个宏，代表要被打印出的某个值：

```
#include <stdio.h>

int
main (void)
{
    printf("Value of NUM is %d\n", NUM);
    return 0;
}
```

注意，宏不会在字符串中被展开----预处理器只会代替字符串以外有 NUM 的地方。

“-D”命令行选项可以用来定义有值的宏，形式是“-D*NAME*=*VALUE*”这样。例如，在编译上面的程序时，下面的命令行把 NUM 定义为 100：

```
$ gcc -Wall -DNUM=100 dtestval.c
$ ./a.out
Value of NUM is 100
```

这个例子中用到的是定义成数字的宏，其实宏可以被定义成任何形式。无论宏被定义成什么，它只是被直接插入到源代码中该宏的名字出现的地方。例如，在下面的定义中，预处理器会把 NUM 出现的地方展开成 2+2：

```
$ gcc -Wall -DNUM="2+2" dtestval.c
$ ./a.out
Value of NUM is 4
```

在预处理器把 NUM 替换成 2+2 以后，就相当于编译下面的程序：

```
#include <stdio.h>

int
main (void)
{
    printf("Value of NUM is %d\n", 2+2);
    return 0;
}
```

注意，当宏是某个表达式的一部分时，用圆括号把宏括起来是个好主意。例如，下面的程序用圆括号来确保 10 × NUM 乘法的正确优先级：

```
#include <stdio.h>

int
main (void)
{
    printf ("Ten times NUM is %d\n", 10 * (NUM));
    return 0;
}
```

有了圆括号，当用上面同样的命令编译后，该程序输出意料中的结果：

```
$ gcc -Wall -DNUM="2+2" dtestmul10.c
$ ./a.out
Ten times NUM is 40
```

如果没有圆括号，该程序将会输出结果 22，因为展开后表达式的字面意义是 10 × 2 + 2 = 22，而不是你想要的 10 × (2 + 2) = 40。

当用“-D”来单独定义一个宏时，gcc 设置其为默认值 1。例如，用“-DNUM”选项编译原来的测试程序，生成的可执行文件产生下面的输出：

```
$ gcc -Wall -DNUM dtestval.c
$ ./a.out
Value of NUM is 1
```

利用命令行上的双引号，宏可以被定义成空值，象-NAME=“”。这样的宏还是会被如#ifdef 的条件命令看作已被定义，但该宏被展开成空。

宏可以包含引号，只要用到shell中的转义引号字符就行。例如，命令行选项-DMESSAGE=“\“Hello，World！ ”” 定义了一个MESSAGE宏，该宏会被扩展成字符串“Hello，World！”。关于shell中用到的不同类型的引用和转义的解释，请见【进一步阅读】中的“GNU Bash参考手册”。

预处理源文件

使用 gcc 的“-E”选项，你可以直接看到预处理器对源代码处理后的效果。例如，下面的文件定义并用到了宏 TEST：

```
#define TEST "Hello, World!"
const char str[] = TEST;
```

如果该文件是“test.c”，则预处理后的效果可以用下面的命令看到：

```
$ gcc -E test.c
# 1 "test.c"
const char str[] = "Hello, World!" ;
```

“-E”选项导致gcc只运行预处理器，显示展开后的输出后，没有编译预处理过的源代码就退出了。TEST宏在输出中已经被替换，生成了字符序列**const char str[] = "Hello, World!" ;**。预处理器也插入了一些行，以**# line-number "source-file"**的形式记录源文件和行数，有助于调试，允许编译器参考这些信息来报错。这些行并不会影响程序本身。

能够查看预处理过的源文件有助于检查include系统头文件的效果和查询系统函数的声明。下面的程序包括了头文件“stdio.h”，以便获得printf函数的声明：

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

```
int
main (void)
{
   printf ("Hello, world!\n");
   return 0;
}
```

通过对要预处理的文件使用 gcc －E，我们就能从被包括入的头文件中看到函数的声明：

```
$ gcc -E hello.c
```

在 GNU 系统上，输出可能象下面一样：

```
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3

extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;

extern int fprintf (FILE * __stream,
                    const char * __format, ...) ;
extern int printf (const char * __format, ...) ;

[ ... 另外的声明 ... ]

# 1 "hello.c" 2

int
main (void)
{
   printf ("Hello, world!\n");
   return 0;
}
```

被预处理过的系统头文件通常产生许多输出。它们可以被重定向到文件中，或者更方便的是使用 gcc 的“-save-temps”选项来保存：

```
$ gcc -c -save-temps hello.c
```

运行该命令以后，预处理过的输出将被存储在文件“hello.i”中。“-save-temps” 选项除了保存预处理过的“.i”文件外，还会保存“.s”的汇编文件和“.o”的对象文件。

带调试信息编译

通常，可执行文件并不包含原来程序中源代码的任何引用信息，比如象变量名或行号----可执行文件只是编译器生成的作为机器码的指令序列。如果程序崩溃了，由于没有简便的方法找到错误的原因，所以这对调试是不够的。

GCC提供了“-g”调试选项来在对象文件和可执行文件中存储另外的调试信息。这些调试信息可以使得在追踪错误时能从特定的机器码指令对应到源代码文件中的行。它也可以使得该程序能被象gdb之类的调试器追踪调试（更多信息请见【进一步阅读】的“Debugging with GDB:The GNU Source-Level Debugger”）。使用调试器可以在程序运行时检查变量的值。

调试器是通过把函数名和变量（和所有对它们的引用），以及它们相应的原代码的行号存储到对象文件和可执行文件的符号表中来工作的。

检查 core 文件

“-g”选项除了允许程序在调试器控制下运行以外，另一个有用的应用是找到程序崩溃的环境。当一个程序异常退出时，操作系统会写一个常规被称为“core”的文件，它包括了程序在崩溃刹那间的内存状态。结合由“-g”选项生成的符号表中的信息，程序员从 core 文件中可以查询到程序在哪一行停止了，和在那时候的变量的值。

这在软件开发期间和部署以后都是很有用的----它允许当程序在“某处”崩溃时能对问题展开调查。

下面是一个包含有非法内存访问错误的简单程序，我们用它来生成一个 core 文件：

```
int a (int *p);

int
main (void)
{
    int *p = 0; /* null pointer */
    return a (p);
}

int
a (int *p)
{
    int y = *p;
    return y;
}
```

上面的程序试图访问一个null指针，这是非法操作。在绝大部分系统上，将会导致程序崩溃。[12]

为了能找到崩溃的原因，我们需要带“-g”选项编译该程序：

```
$ gcc -Wall -g null.c
```

[12] 历史上，null指针对应到内存的 0 地址，这儿通常是被操作系统的内核所限制的，用户程序不能访问的。

注意，null 指针只会在运行期导致问题，“-Wall”选项不会输出任何警告。

在 x86 GNU/Linux 系统上运行该可执行文件将导致操作系统异常终止该程序：

```
$ ./a.out
Segmentation fault (core dumped)
```

只要显示了报错信息“core dumped”，操作系统就在当前目录下生成了一个名为“core”的文件。[13] 该core文件包含程序在被终结时用到的内存页面的完整备份。顺便说一下，术语“段违例（segmentation fault）”指的是程序试图访问不在分配给它的内存区域之列的受限制的内存“段”（segmentation）。

由于 core 文件可能很大并且可能快速地填满系统中的可用磁盘空间，一些系统被配置成在默认情况下不写 core 文件。在 GNU Bash shell 中，命令 ulimit －c 可以控制设定 core 文件的最大值。如果这个限定值是零，则不会生成 core 文件。输入下面的命令可以显示当前的限定值：

```
$ ulimit -c
0
```

如果结果是零，象上面显示的，用下面的命令可以扩大该值，以便允许写入任何大小的 core 文件：

```
$ ulimit -c unlimited
```

注意，这样设置只对当前 shell 有效。为了使设置的限定值在以后的会话中有效，该命令应被放置到适当的登录文件中，象 GNU Bash shell 的“.bash_profile”中。

GNU 调试器 gdb 可以用下面的命令载入 core 文件：

```
$ gdb EXECUTABLE-FILE CORE-FILE
```

注意，原来的可执行文件和 core 文件都要提供，以备调试----没有相应的可执行文件，只有 core 文件是不能调试的。在本例中，我们可以用下面的命令来载入可执行文件和 core 文件：

```
$ gdb a.out core
```

调试器马上打印出诊断信息和显示程序崩溃处的代码行（第 13 行）：

```
$ gdb a.out core
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080483ed in a (p=0x0) at null.c:13
13 int y = *p;
(gdb)
```

最后一行（gdb）是 GNU 调试器的提示符----它表示现在新的命令可以被输入了。

要调查程序崩溃的原因，我们用调试器的 print 命令来显示指针 p 的值：

```
(gdb) print p
$1 = (int *) 0x0
```

这表示 p 是个类型为“int *”的 null 指针，这样我们就知道了正是在该行的表达式*p 来解引用该 null 指针导致程序崩溃。

---

[13] 在一些系统上，比如象FreeBSD和Solaris，也可能利用sysctl或coreadm命令配置成把core文件写到指定目录，比如“/var/coredumps/”。

显示回溯堆栈

调试器也能够显示直到当前执行点的函数调用及其参数----这被称为堆栈回溯，用 backtrace 命令可以显示这一切:

```
(gdb) backtrace
#0 0x080483ed in a (p=0x0) at null.c:13
#1 0x080483d9 in main () at null.c:7
```

在这里，backtrace 命令显示第 13 行处引起的程序崩溃，在 main()中的第 7 行调用函数 a()，传递的参数是 p=0x0。程序员可以利用调试器命令 up 和 down 在堆栈的不同层间移动，以检查其间的变量。

gdb 中所有可用命令的完整介绍请见【进一步阅读】中的“Debugging with GDB:The GNU Source-Level Debugger”手册的介绍。

编译优化

GCC 是一个具有优化功能的编译器，它提供了广泛的选择，目的是提高生成的可执行文件的运行速度或减少它的大小。
优化是个复杂的过程。源码中每一行高级语言通常都可以对应到指令的许多组合，它们都能得到相应的结果。编译器必须考虑到这各种可能并选择其中之一。
通常，编译器为不同的处理器生成不同的代码，用的是不兼容的汇编语言和机器码。每种处理器也有自己的特性----有些 CPU 提供了大量的寄存器，可以存放计算的中间结果，而另外一些必须从内存存储和获取中间结果。编译器必须根据不同情况来生成不同的代码。
此外，不同的指令加上不同的指令次序，所花时间也都不同。GCC 会考虑所有这些因素，在优化编译时试图为一个给定系统产生运行最快的可执行文件。

源码级优化

GCC 用到的第一种优化形式发生在源代码一级，不需要任何机器指令的知识。有许多源码级的优化技巧----本节介绍常用的两种类型：公共子表达式消除和函数内嵌。

公共子表达式消除

一个容易理解的源码级优化方法是通过重用已经计算过的结果这种生成较少指令的方式来实现源文件中对某个表达式的计算。例如下面的赋值语句：

```
x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2))
```

可以被重写，用临时变量 t 来消除对第二个 sin（u/2）表达式的不需要的额外计算：

```
t = sin(u/2)
x = cos(v)*(1+t) + sin(w)*(1-t)
```

这种重写被称为公共子表达式消除（CSE），在优化被打开的情况下会自动实施[14]。公共子表达式消除是一种有力的优化手段，因为它同时提高了执行速度和减少了代码的大小。

函数内嵌

另一类型的被称为函数内嵌的源码级优化提升了被频繁调用函数的效率。
无论什么时候用到一个函数，CPU 都要花费一定的额外时间来实施这个调用：它必须把函数的参数存放到适合的寄存器或内存中，跳转到函数的起始地址处（如果需要的话，把被调用函数所在的虚拟内存页面调入物理内存或者是 CPU 的缓存中），开始执行函数中代码，当函数调用完成后，还要返回到原来的执行点。这些额外的工作被称为函数调用的开销。函数内嵌通过把对函数的调用替换为函数代码本身来消除这种开销（即众所周知的代码内嵌）。
绝大部分情况下，函数调用的开销在程序总运行时间中只占几乎可以忽略的一小部分。如果调用函数所需要的指令多于把该函数体嵌入后的指令，那这种优化肯定是值得的。C++中的简单的成

[14] 在进行公共子表达式消除优化期间编译器引入的临时变量只是被内部使用，不会影响到真正的变量。上展示的临时变量名“t”仅是示意性的。

员访问函数普遍是这种状况，它从嵌入（inline）中获益良多。此外，嵌入也有助于进一步优化，比如公共子表达式消除，把几个独立的函数合并成单个大函数。

下面的sq(x)函数是一个典型的得益于内嵌的例子。它求x^2的值，即参数x的平方：

```
double
sq (double x)
{
    return x * x;
}
```

该函数很小，所以调用开销与该函数本身执行单行乘法运算所花时间有得一比。如果该函数被用在循环中，比如像下面那样，那么函数调用开销将变得很值得关注：

```
for (i = 0; i < 1000000; i++)
{
    sum += sq (i + 0.5);
}
```

嵌入优化用函数体的代码来替换程序中的内层循环，象下面代码：

```
for (i = 0; i < 1000000; i++)
{
    double t = (i + 0.5); /* temporary variable */
    sum += t * t;
}
```

消除函数调用，实行乘法的内嵌会让循环以最高效率运行。

GCC使用一些启发式的方法来选择哪些函数要内嵌，比如函数要适当的小。“嵌入”这种优化方式只在单个对象文件基础上实施。关键字inline可以被用来显式要求某个指定函数在用到它的文件中应该尽可能被内嵌。GCC参考手册“Using GCC”对inline关键字，static和extern限定符与inline共用来显式控制内嵌函数的链接等方面有完全的介绍（见【进一步阅读】）。

速度-空间的折衷

有些优化形式，比如公共子表达式消除，能够同步地提高程序运行速度和缩减程序的大小，另外一些类型的优化在增加可执行文件大小的代价下生成运行更快的代码。在速度和内存之间的选择被称为速度-空间的折衷。速度-空间折衷后的优化可以使得可执行文件在运行稍微慢一点的代价下更小。

循环展开

速度-空间折衷优化的首个例子是循环展开。这种优化形式通过消除每次迭代都要检查的循环结束条件来提高循环的速度。例如，下面代码从 0 循环到 7，在每次迭代时都要测试 I<8 这个条件：

```
for (i = 0; i < 8; i++)
{
    y[i] = i;
}
```

到循环结束的时候，这个测试已经被执行了 9 次，运行时间的一大部分花在了检查条件上。写同样的代码却更加高效率的方法是展开循环，直接进行赋值操作：

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```

这种形式的代码不需要任何条件测试，以最快速度运行。由于每个赋值是独立的，这也允许编译器可以在支持并行处理的处理器上使用并行运算。循环展开是一种提升生成的可执行文件的运行速度但通常也增加它大小的优化（除非循环非常短，比如，只有一两次迭代)。

在循环的上边界未知,但提供的初始条件与结束条件被正确处理的情况下,循环展开也是可能的。例如，同样的循环但有任意的上边界，

```
for (i = 0; i < n; i++)
{
    y[i] = i;
}
```

可以被编译器重写成下面那样:

```
for (i = 0; i < (n % 2); i++)
{
    y[i] = i;
}
for ( ; i + 1 < n; i += 2) /* no initializer */
{
    y[i] = i;
    y[i+1] = i+1;
}
```

第一个循环处理 i = 0 开始的 n 为奇数的状况，第二个循环处理所有剩下的迭代。注意，第二个循环没有在 for 语句的第一个参数中用到初始值，因为它是接着第一个循环结束后运行的。第二个循环中的赋值操作可以并行化，总的条件测试次数所见为 1/2（大致上)。如果在循环内展开更多赋值操作，那速度会更快，但代价是代码大小会更大。

指令调度

最底层的优化是指令调度（scheduling)，即由编译器决定各条指令的最佳次序。绝大部分 CPU 允许在一条指令结束执行以前开始执行一条或多条新指令。许多 CPU 也支持流水线操作，即多条指令在同一 CPU 上并行地执行。

在指令调度优化打开的情况下，指令需要被重新安排，以便在适当的时间点，它们运行的结果对后面的指令是可获得的，以允许最大的并行执行。指令调度没有增加可执行文件的大小，但改善了它的运行速度。当然这需要额外的内存和编译处理的时间（由于指令调度的复杂性)。

优化级别

为了控制编译所花时间和编译器的内存用量，以及在生成的可执行文件的运行速度与文件大小之间折衷，GCC 提供了一系列的通用优化级别，数字从 1 到 3，也包括针对特定级别优化的各个选项。

命令行选项“-O*LEVEL*”用来选择哪一种优化级别，这里 *LEVEL* 是从 1 到 3 的数字。不同优化级别的效果在下面介绍：

“-O0”或没有“-O”选项（默认）

在该优化级别，GCC 不会实施任何优化，用近可能直接的方法来编译源代码。源代码中的每行代码被直接转换成可执行文件中的对应指令，没有任何调整。当调试一个程序时，这是使用的最佳选项。

“-O0”选项等同于不指定“-O”选项。

“-O1”或“-O”

这一级会打开那些不需要任何速度-空间折衷的最常见形式的优化。使用该选项后，生成的可执行文件应当比使用“-O0”更小也更快。更深度的优化，比如指令调度，在这一级别不会被用到。

带选项“-O1”编译所花时间可能常常少于带“-O0”编译，这是由于在简单优化后减少了需要处理的数据量。

“-O2”

该选项除了“-O1”用到的那些优化以外，打开进一步优化。这些另外的优化中包括指令调度。只有不需要速度-空间折衷的优化才被该级别用到，所以可执行文件在大小上不应该有增加。相比“-O1”，编译器将花更多时间来编译程序，并需要更多内存。对要部署的程序而言，该选项通常是最佳选择，因为在不增加可执行文件大小的情况下，它提供了最大的优化。它是各种 GNU 软件发行包的默认优化级别。

“-O3”

该选项除了用到低级别的“-O2”和“-O1”的所有优化以外，还打开更深度的优化，比如函数内嵌。“-O3”优化级别提升生成的可执行文件的速度，但也可能增加它的大小。在有些情况下，这些优化反而可能是不利的，实际上会使得程序运行减慢。

“-funroll-loops”

该选项打开循环展开，它独立于其他优化选项。它会增加可执行文件的大小。使用该选项是否会产生有益的结果，需要逐个结果来检验。

“-Os”

该选项选择缩减可执行文件大小的优化。它的目的是为内存和磁盘空间受限的系统生成尽可能小的可执行文件。在某些情况下，由于较好的缓冲利用率，较小的可执行文件也运行得更快。

记住，你要衡量在最高级别优化所获得的好处与为此付出的代价，这是重要的。优化的代价包括增加调试的复杂度，增加编译期所花的时间和所需要的内存。就绝大部分目的而言，调试时用“-O0”，开发和部署时用“-O2”就足够了。

例子

下面的程序用演示不同优化级别的效果：

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

```
double
powern (double d, unsigned n)
{
    double x = 1.0;
    unsigned j;

    for (j = 1; j <= n; j++)
        x *= d;

    return x;
}

int
main (void)
{
    double sum = 0.0;
    unsigned i;

    for (i = 1; i <= 100000000; i++)
    {
        sum += powern (i, i % 5);
    }

    printf ("sum = %g\n", sum);
    return 0;
}
```

主程序包含一个调用 powern 函数的循环。该函数通过累乘来计算某个浮点数的 n 次幂----该函数被选择是由于它既适合嵌入也适合循环展开。该程序的运行时间可以用 GNU Bash shell 中的 time 命令来计量。

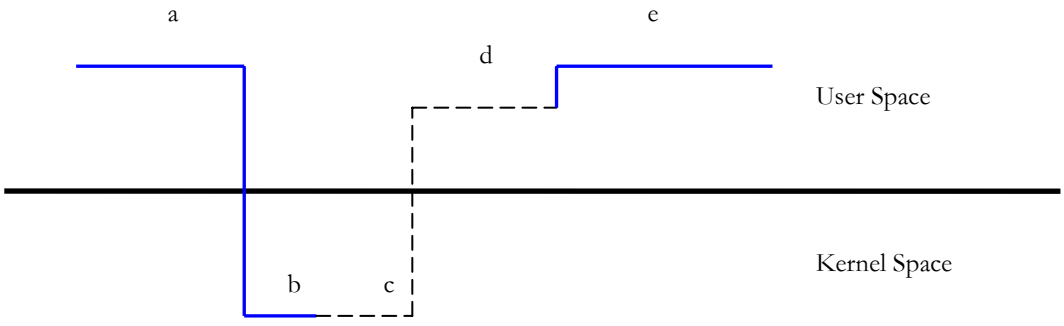
这里是一些上面程序运行的结果，该程序的编译环境是这样的：带有 16KB 一级缓存和 128KB 二级缓存的 566MHz 的 Intel Celeron CPU，在 GNU/Linux 系统下使用 GCC 3.3.1 版本。

```
$ gcc -Wall -O0 test.c -lm
$ time ./a.out
real 0m13.388s
user 0m13.370s
sys 0m0.010s
$ gcc -Wall -O1 test.c -lm
$ time ./a.out
real 0m10.030s
user 0m10.030s
sys 0m0.000s
$ gcc -Wall -O2 test.c -lm
$ time ./a.out
```

```
real 0m8.388s
user 0m8.380s
sys 0m0.000s
$ gcc -Wall -O3 test.c -lm
$ time ./a.out
real 0m6.742s
user 0m6.730s
sys 0m0.000s
$ gcc -Wall -O3 -funroll-loops test.c -lm
$ time ./a.out
real 0m5.412s
user 0m5.390s
sys 0m0.000s
```

上面比较了各个可执行文件的运行速度，在其输出中，“user”时间这一项给出的是进程运行所花的实际时间。另外两列，“real”和“sys”，记录了进程活了多长时间（包括其他进程使用的 CPU 的时间）和等待操作系统调用所花时间。虽然上面每种情况只运行了一次，但被执行多次的基准测试确证了这个结果。

译者注：
下图中蓝线表示我们关心的程序在运行，而虚线表示其他进程在运行。那么对测试程序 a.out 而言，“user”时间就是（a+e）时间段，“real”时间是指（a+b+c+d+e）时间段，而“sys”时间指 b 时间段。由于测试程序中主要的程序片段是运行 1 亿次的 powern ()，其完全在“user”时间段（由于没有调用系统调用，所以不会陷入内核，除非时间片耗尽而被调度出 CPU）。



从这个例子的结果中可看出，相对于用“-O0”编译的未优化代码而言，提高优化级别，从“-O1”到“-O2”到“-O3”，可以显著的提升速度。另外选项“-funroll-loops”可以进一步带来速度提升。从未优化代码到最高级优化之间，程序的速度提升几乎是两倍以上。
注意，对于类似这样的小程序的优化，要考虑到系统和编译器版本的不同而造成优化效果上的差

异。例如，在 Mobile 2.0 GHz Intel Pentium 4M 系统上，使用同样版本的 GCC 编译的结果是速度差相仿佛，就是用“-O2”编译后的性能略差与用“-O1”编译的。这说明了重要的一点：并不是在任何情况下，优化都会使程序更快。

优化和调试

在 GCC 下，可以组合使用优化与调试选项“-g”，而许多其他编译器不支持这样做。
当调试编译和优化一起使用时,优化器会对内部指令重排,使得在调试器中查看优化过的程序时，很难看清指令在干什么。比如，临时变量经常被消除，语句的次序也可能改变。
然而，当程序出人意料地崩溃时，有点调试信息总比什么都没有要好----所以推荐你在优化程序时加上“-g”选项，即为了开发，也为了部署。GNU 发行的软件包默认都打开了调试选项“-g”和优化选项“-O2”。

优化和编译警告

当打开优化后，GCC 会输出一些在不是优化编译时不出现的额外警告信息。
作为优化过程的一部分，编译器检查所有变量的使用和它们的初始值----这被称为*数据流分析*。它是其他优化策略的基础，比如象指令调度。*数据流分析*的一个副作用是编译器可以检测到是否使用了未初始化变量。
“-Wuninitialized”选项（已被包括进“-Wall”中）会对未初始变量的读操作产生警告。它只有在优化编译并打开了*数据流分析*后才工作。下面的函数中包含这样一个变量的例子：

```
int
sign (int x)
{
int s;

if (x > 0)
    s = 1;
else if (x < 0)
    s = -1;

return s;
}
```

该函数对绝大部分参数都能正常工作，但当 x 是 0 时就有 bug 了----在这种情况下，返回的变量 s 没有被初始化。
单单用“-Wall”选项来编译该程序，不会输出任何警告，因为在没有优化的情况下，不会进行数据流分析：

```
$ gcc -Wall -c uninit.c
```

该程序必须同时用“-Wall”和优化编译才能输出警告。实际上，需要优化级别“-O2”编译才能给出有用的警告信息：

```
$ gcc -Wall -O2 -c uninit.c
uninit.c: In function ‘sign’:
uninit.c:4: warning: ‘s’ might be used uninitialized
```

```
    in this function
```

这就正确地检测到变量 s 有可能没有被初始化就被使用到。

要注意的是，虽然通常 GCC 能够找到绝大部分未初始化变量，但它使用的启发式方法也会漏掉某些复杂情形或误报警高。后一种情形可能需要你用一种更简单的方式来重写相应的代码行，来消除误报的警告，同时也改善了源码的可读性。

编译 C++程序

本章介绍怎样用 GCC 来编译 C++程序，及特定于 C++的命令行选项。
GCC 提供的 GNU C++编译器是真正的 C++编译器----它直接把 C++源代码编译成汇编语言。市面上有一些 C++“编译器”其实只是翻译器，把 C++程序转换成 C，然后用现存的 C 编译器来编译生成 C 程序。一个真正的 C++编译器，象 GCC，能够对错误报告，调试和优化提供较好的支持。

编译一个简单的 C++程序

编译 C++程序的过程与编译 C 程序是一样的，但用到的是 g++，而不是 gcc。这两个编译器都是 GNU 编译器集合的一部分。
下面用 C++版本的 Hello World 程序来演示 g++的使用：

```
#include <iostream>

int
main ()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

该程序可以用下面的命令编译：

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

GCC 的 C++前端用到很多同 C 编译器 gcc 相同的选项。它也支持另外一些选项来控制 C++才有的语言特性，本章后面会介绍。注意，C++源代码文件合法的后缀名可以是“.cc”，“.cpp”，“.cxx”或是“.C”,但不能是 C 程序的后缀名“.c”。
生成的可执行文件同 C 版本的程序一样运行，只要输入文件名即可：

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

该可执行文件生成同其 C 版本一样的输出，只是调用的是 std::cout 而不是 C 的 printf 函数。在前面章节中用到的 gcc 命令的所有选项不需改变就可以应用到 g++上，比如编译，链接文件和库（当然用的是 g++，而不是 gcc）。在使用 g++时，一个很自然的不同点是“-ansi”选项的要求是兼容于 C++标准，而不是 C 标准。
要注意的是，C++对象文件必须用 g++来链接，以便与适当的 C++库链接。试图用 C 编译器 gcc 来链接 C++对象文件会导致由于找不到 C++标准库函数而报“未定义引用”的错：

```
$ g++ -Wall -c hello.cc
$ gcc hello.o (should use g++)
hello.o: In function `main':
hello.o(.text+0x1b): undefined reference to `std::cout'
.....
```

```
hello.o(.eh_frame+0x11):
undefined reference to `__gxx_personality_v0'
```

用 g++来链接同样的对象文件却会生成能正常工作的可执行文件，因为 g++能找到必要的 C++库。

```
$ g++ hello.o
$ ./a.out
Hello, world!
```

有时候让人迷惑的一点是，当 gcc 检测到是 C++的文件后缀名时，它确实能够编译 C++源代码，但不能链接自己生成的对象文件：

```
$ gcc -Wall -c hello.cc (succeeds, even for C++)
$ gcc hello.o
hello.o: In function `main':
hello.o(.text+0x1b): undefined reference to `std::cout'
```

为了避免这种问题，最好一致地对 C++程序使用 g++，而对 C 程序使用 gcc。

使用 C++标准库

C++标准库已经成为 GCC 的一部分以备程序员使用。下面的程序使用标准库中的 sring 类重新实现了 Hello World 程序：

```
#include <string>
#include <iostream>

using namespace std;

int
main ()
{
    string s1 = "Hello,";
    string s2 = "World!";
    cout << s1 + " " + s2 << endl;
    return 0;
}
```

该程序可用与上面相同的命令来编译和运行：

```
$ g++ -Wall hellostr.cc
$ ./a.out
Hello, World!
```

注意，根据 C++标准，C++库自身的头文件不再需要后缀名。标准库中的类都定义在 std 命名空间中，所以访问它们需要使用 using namespace std 的指示符，除非用到库函数的地方都明白的加上 std::前缀（象在前面章节那样）。

模板

模板为 C++类提供了支持泛型编程的能力。模板可以被看作是一种强有力的宏。当一个模板化

的类或函数被特定类型实例化时，比如象 float 或 int，对应的模板代码会被用该类型代码取代后编译。

使用 C++标准模板库

由 GCC 提供的 C++标准库“libstdc++”除了包含象排序等泛型算法外，还包含很多象列表和队列等的泛型容器类。这些类原本是标准模板库（STL）的一部分，是一个独立的包，但现在已经被包括进 C++标准库中。

下面的程序演示了模板库的用法，该例子用模板 list<string>创建了一个字符串列表：

```
#include <list>
#include <string>
#include <iostream>

using namespace std;

int
main ()
{
    list<string> list;
    list.push_back("Hello");
    list.push_back("World");
    cout << "List size = " << list.size() << endl;
    return 0;
}
```

使用标准库中的模板类不需要特殊的编译选项，编译该程序的命令行选项同前面是一样的：

```
$ g++ -Wall string.cc
$ ./a.out
List size = 2
```

注意，由 g++创建的使用 C++标准库的可执行文件会被链接到共享库“libstdc++”，该库作为 GCC 的一部分而被默认安装。它有几个版本----如果你要发布使用 C++标准库的可执行文件，你需要确保接受方安装了“libstdc++”的兼容版本，或者你索性用“-static”命令行选项来静态链接你的程序。

编写自己的模板

除了 C++标准库提供的模板类以外，你也可以定义自己的模板。g++中使用模板的推荐方法是遵循“包含编译模型（inclusion compilation model）”，即把模板的定义放到头文件中。GCC 的 C++标准库本身就使用该方法。用到模板的源文件通过“#include”指示符包括入模板所在的头文件。例如，下面的模板文件创建了一个简单的 Buffer<T>类，这是一个保存类型为 T 的对象的环形 Buffer。

```
#ifndef BUFFER_H
#define BUFFER_H

template <class T>
```

```
class Buffer
{
public:
    Buffer (unsigned int n);
    void insert (const T & x);
    T get (unsigned int k) const;
private:
    unsigned int i;
    unsigned int size;
    T *pT;
};

template <class T>
Buffer<T>::Buffer (unsigned int n)
{
    i = 0;
    size = n;
    pT = new T[n];
};

template <class T>
void
Buffer<T>::insert (const T & x)
{
    i = (i + 1) % size;
    pT[i] = x;
};

template <class T>
T
Buffer<T>::get (unsigned int k) const
{
    return pT[(i + (size - k)) % size];
};

#endif /* BUFFER_H */
```

该文件包含类的定义和成员函数的定义。该类仅用于演示目的，而不应当被看作是良好编程的范例。注意这里使用到的“include guards”，即测试 BUFFER_H 宏是否存在，以确保该头文件即使在同样的上下文中被包括入多次，但模板定义也仅被解析一次。

下面的程序使用 Buffer 模板类来创建一个大小为 10 的 Buffer，以存放浮点数 0.25 和 1.0:

```
#include <iostream>
#include "buffer.h"

using namespace std;
```

```
int
main ()
{
    Buffer<float> f(10);
    f.insert (0.25);
    f.insert (1.0 + f.get(0));
    cout << "stored value = " << f.get(0) << endl;
    return 0;
}
```

模板类和其成员函数的定义通过#include “buffer.h”在它们使用以前被包括入源文件中，该程序可以用下面的命令编译：

```
$ g++ -Wall tprog.cc
$ ./a.out
stored value = 1.25
```

在源文件中用到模板函数的地方，g++会编译来自头文件的对应函数并把编译过的函数代码放入相应的对象文件中。

如果程序中有好几处用到模板函数，那么它也将被存入对象文件多处，而由 GNU 链接器来确保只有一份拷贝被放入最终的可执行文件中。当碰到一个模板函数有多于一份拷贝时，其他编译器可能报“有多次被定义的符号”这样的链接错误----下面会介绍与这样的链接器一同工作的办法。

显式模板实例化

要实现 g++对模板编译的完全控制，需要用到“-fno-implicit-templates”选项来对代码中每一个模板出现的地方进行显式实例化。如果使用的是 GNU 链接器，那就不需要用到该方法。对于那些不能消除对象文件中的重复定义的模板函数的链接器的系统，这是“包含编译模型”的一种替代方案。

如果使用“-fno-implicit-templates”选项这种方案的话，模板函数并不在他们使用到地方被编译。取而代之的是，编译器搜寻模板被显式实例化的代码，template 关键字加上为模板指定类型，会迫使编译器编译该代码（译者注：即实例化该模板，如下面例子中的 template class Buffer<float>;）（这是 GNU 对 C++标准行为的扩展）。这种实例化被典型地放置在单独的源文件中，它被编译后生成一个包含所有程序中需要的模板函数的对象文件。这就确保了每个模板函数只出现在一个对象文件中，以便兼容于那些不能消除对象文件中的模板函数被重复定义的链接器。

例如，下面的文件“templates.cc”包含上面的程序“tprog.cc”用到的 Buffer<float>类的某个显式实例化：

```
#include "buffer.h"
template class Buffer<float>;
```

在显式实例化下，整个程序可以用下面的命令来编译链接：

```
$ g++ -Wall -fno-implicit-templates -c tprog.cc
$ g++ -Wall -fno-implicit-templates -c templates.cc
$ g++ tprog.o templates.o
$ ./a.out
stored value = 1.25
```

所有模板函数的代码包含在“ftemplates.o”文件中，使用“-fno-implicit-templates”选项编译 tprog.cc，

在“tprog.o”中没有模板函数的代码。
如果程序修改后用到了其他类型的模板，需要在“templates.cc”文件中添加对应的显式实例化代码。例如，下面的代码为 Buffer 对象添加了 double 和 int 两种类型的实例化：

```
#include "buffer.h"
template class Buffer<float>;
template class Buffer<double>;
template class Buffer<int>;
```

显式实例化的缺点是它需要知道程序用到了那些类型的模板。对于一个复杂的程序，预先知道这个是很困难的。任何缺少的实例化可以在链接时知道，通过链接器报告哪些函数没有被定义，然后把它们添加到显式实例化列表中。（译者注：确实够烦的，这种编译器还有人买？）
显式实例化也可以用来制作预编译的模板函数库,即通过创建包含所有需要的模板函数实例的对象文件（象上面的“templates.cc”文件）。例如，创建自上面的模板实例的对象文件包含了“float”，“double”和“int”类型的 Buffer 类的机器码，可以以库的形式发布。

export 关键字

在写本书时，GCC 还没有支持新的 C++ export 关键字（GCC 3.3.2）。
该关键字被建议用作分离模板的接口和其实现的方法，由于它增加了链接时处理的复杂度，所以也抵消了实际使用中的优点。
export 关键字还没有被广泛使用，绝大部分编译器也还没有支持它。前面介绍的“包含编译模型”作为使用模板最简单和可移植性最好的方式还是被推荐的。

平台相关编译选项[15]

GCC 为不同类型的 CPU 提供了一些与该 CPU 平台相关的编译选项。这些选项用于控制如硬件浮点模式，不同 CPU 的特殊指令的使用之类的特征。它们在命令行上用“-m”选项来指定，GCC 语言前端部分都支持，比如象 gcc 和 g++。
下面介绍在几个广泛使用的 CPU 平台上用到的一些选项。你可以在 GCC 参考手册“Using GCC”上找到所有平台相关的编译选项的完整列表。在新开发的处理器面市后，对该处理器的编译支持也会被加入 GCC 中，这样在本章介绍的一些选项可能在较老版本的 GCC 中不获支持。

Intel 和 AMD x86 的选项

GCC 平台相关选项可以充分利用被广泛采用的 Intel 和 AMD x86 家族 CPU（386，486，Pentium，等）的特有属性。
在这些平台上，GCC 默认生成的可执行文件兼容于所有的 x86 家族的处理器----全退回到 386 CPU 指令集上。然而为了获得较好的性能，GCC 也可以编译生成只能在特定处理器上运行的代码。例如，最近版本的 GCC 对象 Pentium 4 和 AMD Athlon 这类较新的处理器有了专门的支持。对 Pentium 4，你可以用下面的选项来编译：

```
$ gcc -Wall -march=pentium4 hello.c
```

对 Athlon 是这样的：

```
$ gcc -Wall -march=athlon hello
```

在 GCC 参考手册中有所有支持的 CPU 类型的完整列表。
编译时指定“-march=CPU”选项生成的代码运行会快一点，但将不能在 x86 家族中的其他处理器上运行。如果你计划让你发行的可执行文件能在 Intel 和 AMD 处理器上尽可能广泛的被应用，那你在编译时不应该带任何“-march”选项。
作为一种替代办法，“-mcpu=CPU”选项在速度与移植性方面提供了折衷方案----它生成的代码在代码调度方面是为特定 CPU 调优过的，但并不会用到 x86 家族中的其他 CPU 没有的指令。生成的代码能兼容于所有的 x86 CPU，但在由“-mcpu”指定的 CPU 上有速度优势。用“-mcpu”生成的可执行文件不能达到与象指定“-march”一样的性能，但在实际应用中可能更方便。
AMD 增强了 32 位 x86 的指令集，在它的 AMD64 处理器中实现了所谓 x86-64 的 64 位指令集。在 AMD64 系统上，GCC 默认生成 64 位代码，而“-m32”选项允许它生成 32 位代码。
AMD64 处理器在 64 位模式下运行时为程序员提供了几种不同的内存模型。默认的模型是小代码模型，在该模型下代码与数据最大可达 2GB。中等代码模型可以用“-mcmodel=medium”来指定，其允许无限的数据大小[16]。另外一种是大代码模型，其除了支持无限的数据大小外还支持无限的代码大小。由于中等代码模型在实际使用中已经足够，所以目前GCC没有实现大代码模型----在现实中超过 2GB大小的可执行文件没有碰到过。
GCC 为系统代码提供了一种特殊的内核代码模型“-mcmodel=kernel”，比如象 Linux 内核。需要注意的重要一点是对于 AMD64 CPU，默认情况下 GCC 会为临时数据在栈指针的下面分配 128

[15] 翻译这章，译者实在有点诚惶诚恐，因为到目前为止，译者本人只有Intel/AMD/UltraSPARC这 3 类CPU下的开发经验，对其他CPU是只耳闻过，无缘亲见。象DEC Alpha（现在是DEC都不存在了）10 年前就如雷贯耳，因为该型CPU曾被冠以世界上运行最快的CPU。
[16] 译者注：我想实际上不可能是无限的，只不过是非常大，达到 2 的 64 次方规模吧。

字节的空间，该区域也被称为“red-zone”，而在 Linux 内核中是不被支持的。所以在为 AMD64 CPU 编译 Linux 内核时需要加上“-mcmodel=kernel –mno-red-zone”选项。

DEC Alpha 的选项

DEC Alpha 处理器的默认设置是宁愿牺牲对 IEEE 数值运算特性的完全支持，也要最大提升浮点性能。

DEC Alpha 处理器在默认配置时是不支持“数学上的无穷”和“逐步下溢”的。产生这两种结果的操作会产生浮点异常（也就是所谓的陷阱，trap），除非操作系统捕捉并处理了该异常（通常会引起效率降低），否则会导致程序终止。IEEE 标准规定对于这些操作应该产生用 IEEE 数字格式代表的特殊值。

由于大部分程序不会产生“无限值”和“向下溢出”（译者注：因为这一般都是 bug），在绝大部分情况下，DEC Alpha 的默认行为是可以接受的（译者注：即不处理由此产生的异常）。对于需要这些特性的应用程序，GCC 提供了“-mieee”选项来全面支持 IEEE 所规定的数学上的“无穷”和“向下溢出”这两个概念。

下面的程序用中 1 处以 0 来展示了处理与不处理产生的异常这两种情况的区别：

```
#include <stdio.h>

int
main (void)
{
        double x = 1.0, y = 0.0;
        printf ("x/y = %g\n", x / y);
        return 0;
}
```

1/0 的结果在 IEEE 数字规范中是 inf（无穷）。如果该程序是为 Alpha 处理器并用默认设置编译的，那它将产生终止该程序的异常：

```
$ gcc -Wall alpha.c
$ ./a.out
Floating point exception (on an Alpha processor)
```

利用“-mieee”选项来全面支持 IEEE，1/0 的除法运算会正确地产生结果 inf，该程序也能成功地继续运行：

```
$ gcc -Wall -mieee alpha.c
$ ./a.out
x/y = inf
```

要注意的是，用“-ieee”选项编译的会产生浮点异常的程序运行会慢一点，因为异常是由软件处理的，而不是硬件。

SPARC 选项

在 SPARC 一类处理器上，“-mcpu=CPU”选项会使得编译器生成特定处理器的代码。这里的 CPU 有效的值是 v7，v8（SuperSPARC），Sparclite，Sparclet 和 v9（UltraSPARC）。除了处理器本身能支持向下兼容以外，用“-mcpu”选项生成的代码将不能在 SPARC 家族的其他处理器上运行。

对 64 位的 UltraSPARC 系统而言，选项“-m32”和“-m64”可以指导编译器是为 32 位还是 64 位环境生成代码。在“-m32”指定的 32 位环境中，int，long，和指针类型都是 32 位大小的，而“-m64”指定的 64 位环境中，int，long，和指针类型都是 64 位大小的。

POWER/PowerPC 选项

在使用 POWER/PowerPC 一类处理器的系统上，“-mcpu=CPU”用于为指定的 CPU 模型生成代码。这里 CPU 的可能值包括“power”，“power2”，“powerpc”，“powerpc64”和“common”，之外，还有其他更具体的型号。用“-mcpu=common”编译产生的代码能在任何该类处理器上运行。如果硬件有“Altivec vector processing”功能，则“-maltivec”选项可以用以产生实现支持该功能的指令。

Power/PowerPC 处理器包括一种组合“乘加”的指令 a * x + b，该指令为了加快速度同时执行两种运算----这被称为融合的乘与加，默认情况下，GCC 会用生成该指令。由于中间结果舍入可能造成采用融合的乘与加的结果与两个运算独立进行得到的结果不完全一致。在需要严格的符合 IEEE 计算的场合，这种组合指令的生成可以用“-mno-fused-madd”选项禁止。

在 AIX 系统上，“-mminimal-toc”选项可以减少 GCC 往可执行文件的全局内容表（global of contents，TOC）中的项数，避免因为全局内容表中项目太多而在链接阶段报告“TOC overflow”错误。“-mxl-call”选项使得编译自 GCC 的对象文件同编译自 IBM 的 XL 编译器的对象文件在链接上能兼容。对于使用 POSIX 线程的应用程序，即使该程序仅仅运行在单线程模式下，在编译时 AIX 系统也需要“-pthread”选项。

对多架构的支持

有些平台能够执行不止一个架构的代码，比如象AMD64，MIPS64，Sparc64 和PowerPC64 这类 64 位平台支持 32 位和 64 位代码的执行。类似的，ARM处理器支持普通ARM代码和一种被称为“Thumb”的更紧凑的代码。GCC都能为这些平台上的多种架构生成代码。默认情况下，GCC 编译器生成 64 对象文件，但在指定“-m32”选项后能生成对应架构的 32 位对象文件。[17]

要注意的是，要支持多架构还依赖于对应库文件是否可获得。在支持 64 位和 32 位可执行文件的 64 位平台上，64 位的库文件一般放在“lib64”目录下，而不是“lib”目录下，比如象“/usr/lib64”和“/lib64”。在另外的平台上，32 位库文件被放在默认的“lib”目录下。这样允许在同一系统上 32 位库和 64 位库有相同的文件名。另外的一些系统，象 IA64/Itanium，64 位库文件被放在“/usr/lib”和“/lib”目录下。GCC 知道这些路径并在编译 64 位或 32 位代码时使用正确的路径下的库文件。

[17] 在AIX上，是“-maix64”和“-maix32”选项。

疑难解答

GCC 提供了一些帮助和诊断用的编译选项，有助于解决编译过程中碰到的问题。本章介绍的所有编译选项，gcc 和 g++都支持。

help 命令行选项

要获得各个命令行选项的简短提示，GCC 提供了 help 选项。该选项可以显示 GCC 顶级命令行选项的概略：

```
$ gcc --help
```

要显示 GCC 的选项的完整列表和它们所关联的程序，比如象 GNU 链接器和 GNU 汇编器，需要在上面 help 选项的基础上加上 verbose（-v）选项：

```
$ gcc -v -help
```

该命令输出的完整的选项列表非常长，你可能要用**more**命令来一屏一屏看，或把输出重定向到文件中在看：

```
$ gcc -v -help 2>&1 | more
```

版本号

你可以用 version 选项来获知 gcc 的版本号：

```
$ gcc -version
gcc (GCC) 3.3.1
```

当调查编译问题的时候，这个版本号是蛮重要的，因为老的GCC版本可能缺少一些正被编译的程序使用的特性。版本号的形式是**主版本号.次版本号**或者**主版本号.次版本号.修正版本号**，这里的额外的第三个修正版本号（象上面所示）用在发行系列中的修正bug的发行版本。
关于版本的更详细信息可以用“-v”选项来看到：

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i686/3.3.1/specs
Configured with: ../configure --prefix=/usr
Thread model: posix
gcc version 3.3.1
```

这包括了编译器在自身被编译时的编译信息的设定和安装时的配置文件，“specs”。

详细的编译信息

“-v”编译选项可以显示用于编译和链接程序时 GCC 发出的确切的命令序列的详细信息。下面有个例子，其显示了编译 Hello World 程序时的详细信息：

```
$ gcc -v -Wall hello.c
Reading specs from /usr/lib/gcc-lib/i686/3.3.1/specs
```

```
Configured with: ../configure --prefix=/usr
Thread model: posix
gcc version 3.3.1
/usr/lib/gcc-lib/i686/3.3.1/cc1 -quiet -v -D__GNUC__=3
-D__GNUC_MINOR__=3 -D__GNUC_PATCHLEVEL__=1
hello.c -quiet -dumpbase hello.c -auxbase hello -Wall
-version -o /tmp/cceCee26.s
GNU C version 3.3.1 (i686-pc-linux-gnu)
compiled by GNU C version 3.3.1 (i686-pc-linux-gnu)
GGC heuristics: --param ggc-min-expand=51
--param ggc-min-heapsize=40036
ignoring nonexistent directory "/usr/i686/include"
#include "..." search starts here:
#include <...> search starts here:
/usr/local/include
/usr/include
/usr/lib/gcc-lib/i686/3.3.1/include
/usr/include
End of search list.
as -V -Qy -o /tmp/ccQynbTm.o /tmp/cceCee26.s
GNU assembler version 2.12.90.0.1 (i386-linux)
using BFD version 2.12.90.0.1 20020307 Debian/GNU
Linux
/usr/lib/gcc-lib/i686/3.3.1/collect2
--eh-frame-hdr -m elf_i386 -dynamic-linker
/lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o
/usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
-L/usr/lib/gcc-lib/i686/3.3.1
-L/usr/lib/gcc-lib/i686/3.3.1/../../.. /tmp/ccQynbTm.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i686/3.3.1/crtend.o
/usr/lib/crtn.o
```

当编译过程本身出现问题时，“-v”选项的输出可能是有用的。它显示了编译器搜索头文件和库文件的完整路径，已定义的预处理器的符号，和链接的对象文件与库文件。

编译器相关工具

本章介绍一些通常与 GCC 一起使用的工具，它们包括用于创建静态库的 GNU 归档工具 ar，分析程序性能的 gprof，分析代码覆盖率的 gcov。

用 GNU 归档工具 ar 创建静态库

GNU 归档工具 ar 用于把多个对象文件组合成归档文件，也被称作库。归档文件是一种把相关各个对象文件打包在一起发行的简便方法。

为了演示 GNU 归档工具的用法，我们创建包含两个函数 hello 和 bye 的小函数库。

第一个对象文件产生自前面见过的文件“hello_fn.c”中的 hello 函数的源代码：

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

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

第二个对象文件产生自源文件“bye_fn.c”，其含有新函数 bye：

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

void
bye (void)
{
    printf ("Goodbye!\n");
}
```

两个函数都用到了头文件“hello.h”，现在为函数 bye()添加原型：

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

用下面的命令可以把源代码编译成对象文件“hello_fn.o”和“bye_fn.o”：

```
$ gcc -Wall -c hello_fn.c
$ gcc -Wall -c bye_fn.c
```

这两个对象文件可以用下面的命令打包成一个静态库：

```
$ ar cr libhello.a hello_fn.o bye_fn.o
```

选项“cr”代表“create and replace”[18]。如果库文件不存在，则会创建。如果库文件已经存在，任何与它同名的原始文件将被命令行上指定的新文件取代。第一个参数“libhello.a”是库的文件名，剩下的参数是要被复制入该库的对象文件的文件名。

[18] 注意，ar的命令行选项不需要前缀“-”。

归档工具 ar 也提供了“内容列表”的选项“t”来列出已有库中的对象文件：

```
$ ar t libhello.a
hello_fn.o
bye_fn.o
```

注意，当发行一个库时，该库提供的公开的函数和变量相关的头文件应该是可获得的，以便最终用户能 include 该头文件以获得正确的函数原型。

我们现在能够写一个用到新建库中的函数的程序：

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

上面的例子文件可用下面的命令行编译，象在 2.6 节中介绍的，假设库“libhello.a”在当前目录：

```
$ gcc -Wall main.c libhello.a -o hello
```

主程序与库文件“libhello.a”中的对象文件链接，生成最后的可执行文件。

短形式的库链接选项“-l”也可以用于链接程序而无需显式指定库的全名：

```
$ gcc -Wall -L. main.c -lhello -o hello
```

“-L.”选项是需要的，其会把当前路径加入库搜索路径中。生成的可执行文件可以象通常那样运行：

```
$ ./hello
Hello, everyone!
Goodbye!
```

它显示了来自定义在库中的 hello 和 bye 函数的输出。

使用性能剖析器 gprof

GNU 性能剖析器 gprof 是衡量程序性能的有用工具----它记录每个函数调用的次数和每个函数每次调用所花的时间。函数在运行期所花费的大部分时间可以很容易地由 gprof 标识出来。加速程序运行的努力应该首先被放在那些在总体运行时间中“称王称霸”的函数。

我们用gprof来检查一个小的数值程序的性能，该程序计算一系列由数学上还未经证明的Collatz猜想产生的数字序列[19]（译者注：考拉兹猜想，又称为 3n＋1 猜想、角谷猜想、哈塞猜想、乌拉姆猜想或叙拉古猜想，是指对于每一个正整数，如果它是奇数，则对它乘 3 再加 1，如果它是偶数，则对它除以 2，如此循环，最终都能够得到 1。例如取一个数字 n = 6，根据上述数式，得出 6→3→10→5→16→8→4→2→1 。考拉兹猜想称，任何正整数，经过上述计算步骤后，最终都会得到 1 。也可以叫“奇偶归一猜想”。在 1930 年代，德国汉堡大学的学生考拉兹，曾经研究过这个猜想，因而得名。在 1960 年，日本人角谷静夫也研究过这个猜想。但这猜想到目前，仍没有任何进展。保罗·艾狄胥就曾称，数学上尚未为此类问题提供答案。他并称会替找出答案的人奖赏 500 元。目前已经有分布式计算在进行验证。到 2005 年 8 月 2 日，已验证正整数到 6 × 258 = 1,729,382,256,910,270,464，也仍未有找到例外的情况。但是这并不能够证明对于任何大小的

[19] 美国数学月刊，卷 92（1985），3-23

数，这猜想都能成立。有的数学家认为，该猜想任何程度的解决都是现代数学的一大进步，将开辟全新的领域。目前也有部分数学家和数学爱好者，在进行关于“负数的 3x＋1”、“5x＋1”、“7x＋1”等种种考拉兹猜想的变化形命题的研究)。Collatz猜想产生的数字序列由下面的规则定义：

$$x_{n+1} \leftarrow \begin{cases} x_n / 2 & \text{if } x_n \text{ is even} \\ 3x_n + 1 & \text{if } x_n \text{ is odd} \end{cases}$$

该数字序列从初始值x_0开始迭代，直到结果为 1 为止。根据该猜想，所有的数字序列最终都会终止与 1。下面的程序展示了当x_0增长时的最长的数字序列。“collatz.c”源码文件包含 3 个函数：main，nseq和step：

```
#include <stdio.h>

/* Computes the length of Collatz sequences */

unsigned int
step (unsigned int x)
{
    if (x % 2 == 0)
    {
        return (x / 2);
    }
    else
    {
        return (3 * x + 1);
    }
}

unsigned int
nseq (unsigned int x0)
{
    unsigned int i = 1, x;

    if (x0 == 1 || x0 == 0)
        return i;

    x = step (x0);

    while (x != 1 && x != 0)
    {
        x = step (x);
        i++;
    }

    return i;
}
```

```
int
main (void)
{
    unsigned int i, m = 0, im = 0;

    for (i = 1; i < 500000; i++)
    {
        unsigned int k = nseq (i);

        if (k > m)
        {
            m = k;
            im = i;
            printf ("sequence length = %u for %u\n", m, im);
        }
    }

    return 0;
}
```

为了剖析性能，该程序必须带“-pg”选项来编译和链接：

```
$ gcc -Wall -c -pg collatz.c
$ gcc -Wall -pg collatz.o
```

这就生成了一个会产生剖析数据的可执行文件，其包含有记录每个函数所花时间的额外指令。如果程序有多个源代码文件，则编译每个源文件时都要带上“-pg”选项，同样在把多个对象文件链接以生成最后的可执行文件时也要如法炮制（象上面展示的）。忘记在链接时带“-pg”选项是常见错误，在剖析性能时不会生成任何有用的信息。

可执行文件只有通过运行才能生成剖析数据：

```
$ ./a.out
(显示程序的正常输出)
```

运行生成的可测量的可执行文件后，剖析数据被悄无声息地记录到当前目录下的“gmon.out”文件中。以可执行文件名作为参数运行 gprof 就可以分析这些数据：

```
$ gprof a.out
Flat profile:
Each sample counts as 0.01 seconds.
% cumul. self self total
time seconds seconds calls us/call us/call name
68.59 2.14 2.14 62135400 0.03 0.03 step
31.09 3.11 0.97 499999 1.94 6.22 nseq
0.32 3.12 0.01 main
```

剖析数据的第一列显示该程序的绝大部分时间（几乎是 70%）花在了 step 函数上，nseq 函数花的时间占了 30%。因此，减少程序运行时间的努力应当专注于前者身上。相比较，main 函数所花的时间完全可以忽略（少于 1%）。

上面输出中的另外的列给出了各个函数被调用的总的次数和每个函数所花的时间的信息。

Additional output breaking down the run-time further is also produced by gprof but not shown here。完整的介绍，可以参考由 Jay Fenlason 和 Richard Stallman 撰写的手册“GNU gprof----The GNU Profiler”。

用 gcov 进行代码覆盖测试

GNU 代码覆盖测试工具 gcov 分析在程序运行期间每一行代码执行的次数。这就使找到没有被用到的代码区域或没有被测试到的代码成为可能。如果综合利用来自 gprof 的剖析性能方面的信息和来自代码覆盖工具的信息，测试的效率会大大加快，可以很快地把问题定位到该程序源代码中的特定行。

我们用下面的例子程序来展示 gcov 的功能。该程序从 1 循环到 9 用取模（%）操作符来测试它们能否被某些数整除。

```
#include <stdio.h>

int
main (void)
{
    int i;

    for (i = 1; i < 10; i++)
    {
        if (i % 3 == 0)
            printf ("%d is divisible by 3\n", i);
        if (i % 11 == 0)
            printf ("%d is divisible by 11\n", i);
    }

    return 0;
}
```

为了对该程序使用代码覆盖测试，编译时必须带如下选项：

```
$ gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

这就生成了一个可以用于测试代码覆盖的程序，其包含额外的指令用于记录程序执行到的每一行代码的次数。“-ftest-coverage”编译选项添加计数被执行到的行的次数，而“-fprofile-arcs”选项合并程序中每条分支中的用于计数的代码。分支中的用于计数的代码记录了在“if”语句和另外的条件下不同路径的执行次数。可执行文件只有在运行后才能生成代码覆盖测试数据：

```
$ ./a.out
3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

程序运行中采集到的数据被写入到当前目录下各个后缀名为“.bb”，“.bbg”和“.da”的文件中。利用 gov 命令跟上源码的文件名就可以给出分析这些数据的结果：

```
$ gcov cov.c
88.89% of 9 source lines executed in file cov.c
Creating cov.c.gcov
```

gcov 命令生成了一个原始源码文件的带注释信息的版本，其后缀名为“.gcov”，包含执行到的每一行代码的运行次数：

```
           #include <stdio.h>

           int
           main (void)
           {
        1      int i;

       10      for (i = 1; i < 10; i++)
               {
        9          if (i % 3 == 0)
        3              printf ("%d is divisible by 3\n", i);
        9          if (i % 11 == 0)
   ######              printf ("%d is divisible by 11\n", i);
        9       }

        1       return 0;
        1  }
```

输出文件的第一列就是每行的执行次数，没有执行到的行被用“######”标记上。命令“grep ‘######’ *.gcov”可以找到程序中没有执行到的部分。

编译器怎样工作

本章进一步详细介绍 GCC 怎样把源代码文件转变成可执行文件的。编译是一个多阶段的过程，涉及一些工具，包括 GNU 编译器本身（通过 gcc 或 g++的前端），GNU 汇编器 as，GNU 链接器 ld。编译过程中用到的整套工具被称为工具链。

编译过程概览

一次 GCC 的调用执行的命令序列包括下面几步：

- 预处理（对宏进行扩展）
- 编译（从源代码到汇编语言）
- 汇编（从汇编语言到机器码）
- 链接（生成最后的可执行文件）

用 Hello World 程序“hello.c”来检验整个编译过程的每一步：

```
#include <stdio.h>

int
main(void)
{
    printf(“Hello, world!\n”);
    return 0;
}
```

注意，并不需要用本章介绍的任何单独一条命令来编译程序。GCC 会自动且透明地执行所有命令，利用前面介绍的带-v 选项就能看到这个过程。本章的目的是帮助你理解编译器是怎样工作的。

虽然 Hello World 程序非常简单，但它用到了外部的头文件和库，所以经历了编译过程的所有主要步骤。

预处理器

编译过程的第一个步骤是利用预处理器来扩展宏和被包含的头文件。GCC 执行下面的命令来实施这个步骤：

```
$ cpp hello.c > hello.i
```

结果是 hello.i 文件，它包含所有宏被展开后的源代码。按惯例，C 程序预处理后的文件被冠以“.i”的后缀名，而 C++程序则是“.ii”后缀名。实际上，除非带有“-save-temp”选项，否则预处理后文件不会被存盘。

编译器

整个编译过程的下一步是把预处理过的源代码实际编译成特定处理器的汇编语言。命令行选项

“-S”指导 gcc 把预处理过的 C 源代码转变成汇编语言，但不生成对象文件（obj 文件）：

```
$ gcc –Wall –S hello.i
```

生成的汇编语言被存储到文件 hello.s 中。下面是生成的 Intel x86(i686)处理器上的 Hello World 汇编语言：

```
$ cat hello.s
.file "hello.c"
.section .rodata
.LC0:
.string "Hello, world!\n"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
movl $0, %eax
subl %eax, %esp
movl $.LC0, (%esp)
call printf
movl $0, %eax
leave
ret
.size main, .-main
.ident "GCC: (GNU) 3.3.1"
```

注意，上面的汇编语言包含对外部函数 printf 的调用。

汇编器

汇编器的目的是把汇编语言转变成机器码并生成对象文件（obj 文件）。如果在汇编源文件中有对外部函数的调用，汇编器会保留外部函数的地址处于未定义状态，有后面的链接器来填写。汇编器可以用下面的命令行来调用：

```
$ as hello.s –o hello.o
```

对 GCC 而言，输出文件由“-o”选项来指定。生成的“hello.o”文件包含 Hello World 程序的机器指令，和对 printf 的未定义引用。

链接器

整个编译过程的最后阶段是链接对象文件生成可执行文件。在实际操作中，一个可执行文件需要许多来自系统和 C 运行库(crt)的外部函数。Consequently，GCC 内部实际用到的链接命令是蛮复杂的。比如，链接 Hello World 程序的完整命令如下：

```
$ ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
```

```
/usr/lib/crti.o /usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
-L/usr/lib/gcc-lib/i686/3.3.1 hello.o -lgcc -lgcc_eh
-lc -lgcc -lgcc_eh /usr/lib/gcc-lib/i686/3.3.1/crtend.o
/usr/lib/crtn.o
```

幸运的是，你从来不需要直接打入上面的命令---整个链接过程由 gcc 以透明方式处理掉了，只需要调用下面的命令：

```
$ gcc hello.o
```

这会把对象文件“hello.o”与 C 标准库链接，生成可执行文件“a.out”：

```
$ ./a.out
Hello, world!
```

C++程序的对象文件可以用同样的方式被链接到 C++标准库，只是改成 g++命令。

检查被编译文件

本章介绍了一些检查可执行文件和对象文件内容的有用工具。

辨识文件

在源代码文件被编译成对象文件或可执行文件以后,编译时指定的编译选项就不再那么容易知道了。File 命令查看对象文件或可执行文件的内容来查看它的特性，比如象它是动态链接的还是静态链接的。

比如，下面是对一个典型的可执行文件应用 file 命令的结果:

```
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared
libs), not stripped
```

输出显示该可执行文件是动态链接的，是为 Intel 386 及其兼容处理器编译的。下面是输出的完整解释:

ELF
: 这是可执行文件的内部格式（ELF 代表“Executable and Linking Format”，另外的格式，象 COFF“Common object File Format”，被用于一些较老的操作系统（比如 MS-DOS））

32-bit
: 这是指字(word)的大小（在有些平台上可能是 64-bit 的）

LSB
: 该文件是为小尾端(LSB)平台编译的,象 Intel 和 AMD 的 x86 处理器(相应的大尾端(MSB)的处理器，如 Motorola 680x0)。一些处理器，如安腾(Itanium)和 MIPS 对 LSB 和 MSB 两者都支持)

Intel 80386
: 该可执行文件为什么处理器编译的。

Version 1(SYSV)
: 这是文件内部格式的版本。

Dynamically linked
: 可执行文件会用到共享库（statically linked 表示程序是静态链接的，比如用到“-static”选项）

Not stripped

可执行文件包含符号表（符号表可以用 strip 命令删除）。

File 命令也能用于对象文件，将给出相似的输出。Unix 系统的 POSIX 标准定义了 file 命令的行为。

检查符号表

正像前面讨论调试时介绍的，可执行文件和对象文件可以包含符号表（请参见第 5 章【带调试信息编译】）。符号表存储了函数和命名变量的位置，用 nm 命令可以看到内容：

```
$ nm a.out
08048334 t Letext
08049498 ? _DYNAMIC
08049570 ? _GLOBAL_OFFSET_TABLE_
........
080483f0 T main
08049590 b object.11
0804948c d p.3
U printf@GLIBC_2.0
```

从上面的输出的符号表的内容可看到 main 函数开始与十六进制偏移 080483f0。绝大部分符号是被编译器和操作系统内部使用的。第二列的“T”表示这是定义在对象文件中的函数，而“U”表示该函数没有被定义（即应该在链接时从另外的对象文件中被找到）。Nm 输出的完整解释可以在 GNU Binutils 手册中找到。

Nm 命令的最常应用是检查某个库是否包含的特定函数的定义，通过查找第二列为“T”项的函数名即可。

查找动态链接库

当程序用到共享库时，为了掉应库中的外部函数，程序需要在运行期动态地载入这些库。Ldd 命令用于检查可执行文件并显示它需要的共享库的列表。这些库被称为该可执行文件的共享库依赖。

例如，下面的命令演示了怎样找到 Hello World 程序的共享库依赖：

```
$ gcc -Wall hello.c
$ ldd a.out
libc.so.6 => /lib/libc.so.6 (0x40020000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

输出显示 Hello World 程序依赖于 C 库 libc(版本为 6 的共享库)和动态载入器库文件 ld-linux(版本为 2 的共享库)。

如果程序用到了外部库，比如数学函数库，它们也将被显示。例如，对 calc 程序应用 ldd 会产生如下输出：

```
$ gcc -Wall calc.c -lm -o calc
$ ldd calc
libm.so.6 => /lib/libm.so.6 (0x40020000)
libc.so.6 => /lib/libc.so.6 (0x40041000)
```

```
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

输出第一行显示该程序依赖数学函数库 libm(版本为 6 的共享库)，另外还依赖 C 库和动态载入器库。

ldd 命令也能够用于检查共享库本身，可以跟踪共享库依赖链。

获得帮助

如果你碰到了在本手册没有提到的问题，可以参考一些更详细地介绍GCC和语言相关主题的参考手册（见【进一步阅读】）。这些手册对通常的问题都有答案，仔细地学习它们通常都能找到解决方法。如果手册写的不够明了，寻求帮助的最佳办法是询问有经验的同事。

另外，业界有许多为 GCC 相关编程事务提供商业支持的按小时或以正进行的业务基础收费的公司和咨询师。在业务上获得高质量的技术支持还是很划算的。

为自由软件提供支持的公司目录和它们的变动率可以在GNU项目的站点[20]找到。自由软件在一个自由的市场中是可以获得商业上的技术支持的----提供服务的公司在质量和价格上进行竞争，而用户也不会被绑在任何特定的公司。相反，专有软件的技术支持通常仅能从原始卖主那儿获得。GCC的高级商业技术支持能从参与GCC编译器工具链开发的公司那儿获得。这些公司的列表可以在由出版社专为本书而设的网页的“开发公司”一节找到[21]。这些公司可以提供诸如扩展GCC以支持新的CPU或修补编译器的bug之类的服务。

[20] http://www.gnu.org/prep/service.html

[21] http://www.network-theory.co.uk/gcc/intro/

进一步阅读

GCC 的权威指南是由 GNU 出版社出版的官方参考手册，“Using GCC”：

Using GCC(GCC 3.3.1 版本)，由 Richard M. Stallman 和 GCC 开发者委员会撰写(GNU 出版社出版，ISBN 号为 1-882114-39-6)

该手册使用 GCC 工作的任何人都是必备的，因为它详细地介绍了每一个编译选项。注意该手册随着 GCC 新版本的发行而更新，所以在将来 ISBN 号可能改变。

如果你是使用 GCC 编程的新手，你也需要学习怎样使用 GNU 调试器 GDB，和怎样方便地用 GNU Make 来编译大型程序。这些工具在下面的手册中介绍：

Debugging with GDB:The GNU Source-Level Debugger，该书由 Richard M. Stallman，Roland Pesch，Stan Shebs 等撰写（GNU 出版社出版，ISBN 号为 1-882114-88-4）

GNU Make:A Program for Directing Recompilation，该书由 Richard M. Stallman 和 Roland McGrath 撰写（GNU 出版社出版，ISBN 号为 1-882114-82-5）

为了进行高效地 C 编程，对标准 C 库比较熟悉是必需的。下面的手册介绍了 GNU C 库中的所有函数：

The GNU C Library Reference Manual，该书由 Sandra Loosemore 和 Richard M. Stallman 等撰写(2 卷)(GNU 出版社出版，ISBN 号为 1-882114-22-1 和 1-882114-24-8)

要获得GNU出版社出版的最新打印版本的手册，请访问http://www.gnupress.org/网址。除了根据ISBN号通过绝大部分书店订购以外，手册可以在FSF站点用信用卡在线购买。GNU出版社出版的手册为自由软件组织和GNU项目募集资金。

关于 shell 命令，环境变量和 shell 用到的规则的信息，可以从下面的书中找到：

The GNU Bash Reference Manual，该书由 Chet Ramey 和 Brian Fox 撰写（Network Theory 有限公司出版，ISBN 号为 0-9541617-7-7）

另外的在本书中提到的GNU手册（比如象GNU gprof----The GNU Profiler and和The GNU Binutils 手册）在本书出版时还没有印刷版本。链接到出版社的站点可以找到本书的在线版本[22]。

你在GNU项目的官方站点上可以找到GCC的页面，位于http://www.gnu.org/software/gcc/。里面包括FAQ列表，GCC bug追踪数据库和关于GCC的许多其他有用信息。

市面上有许多关于介绍 C 和 C++语言的书。下面是两本标准的参考书：

The C Programming Language(ANSI 版本)，由 Brian W. Kernighan 和 Dennis Ritchie 撰写（ISBN 号为 0-13110362-8）

The C++ Programming Language(第三版)，由 Bjarne Stroustrup 撰写（ISBN 号为 0-20188954-4）

[22] http://www.network-theory.co.uk/gcc/intro/

任何在专业领域内使用 C 和 C++的程序员都应当获得一本官方的该语言标准的拷贝。
官方的 C 语言标准号是 ISO/IEC 9899:1990，是为 1990 年原来的 C 语言标准出版，并被 GCC 所实现。修订过的 C 语言标准是 ISO/IEC 9899:1999(大家熟知的 C99)，在 1999 年出版，GCC 支持其中的巨大部分特性（但不是全部）。
C++标准是 ISO/IEC 14882。IEEE 的浮点算数运算标准（IEEE-754）对涉及数学运算的编程也是很重要的。
这些标准的文档都能从相应的标准组织购买到。C 和 C++标准也有印刷版本：

The C Standard: Incorporating Technical Corrigendum 1（由 Wiley 出版，ISBN 号为 0-470-84573-2）

The C++ Standard（由 Wiley 出版,ISBN 号为 0-470-84674-7）

为了进一步学习，使用GCC的程序员可能想加入C和C++用户协会(ACCU)。ACCU是个非营利性的组织，几乎对编程的各个层面都有它专业精神的贡献，并被认为是这一领域的权威。要获得更多信息请访问http://www.accu.org站点。

ACCU 出版有两本关于 C 和 C++方面的杂志，并组织常规讨论会。对个人和想让它们的员工在职业开发领域中达到更高标准的公司而言，ACCU 每年的会员费是笔好的投资。

感谢

许多人对本书都有贡献，在这里记录他们的名字是重要的：

感谢 Gerald Pfeifer，他仔细地评阅和多次的建议才使得本书更好。
感谢 Andreas Jaeger，是他给予了有关 AMD64 和多架构支持的信息，以及许多有益的评阅。
感谢 David Edelsohn，是他提供了 POWER/PowerPC 系列处理器的信息。
感谢 Jamie Lokier 的研究工作。
感谢 Stephen Compall，为他对本书的修正。
感谢 Gerard Jungman，为他对本书的评阅。
感谢 Steven Rubin，是他制作了 the chip layout for the cover with Electric.
最重要的,当然是感谢 Richard Stallman，GNU 项目的奠基者，是他编写了 GCC 并使它成为自由软件。

译者跋

本书对已经熟悉 C/C++编程（我想大部分应该是在 Windows 平台上吧）的开发者，现在又想学习类 Unix 系统下开发的程序员是非常适合的。简洁，实用，毫无废话，再加上 Stallman 大师的推荐，虽然未到百页（英文版 100 多页），但绝对胜过国内同类书籍。

入门书难写，简洁，干净但能让人“入门”的书更是凤毛麟角。但本文绝对可称此种佳作，而且仅仅未满百页（中文版），实在难啊！

我翻译此书，纯粹是因为看到本书中 Stallman 大师的序言。本人从自由软件中学习了一些知识，但才疏学浅，干不了什么大事，写不了什么有大用的软件，无以回馈，翻译本书聊表寸心，希望对不太爱看英文技术书的开发者有所帮助。

对 Stallman 大师时常有“高山仰止”之叹，虽不完全赞同他的理念，但对大师的精神与技术，佩服得无以复加。大师说“**自由软件运动需要你的支持**”，我一小卒就吹吹号吧。

翻译中如有不准确或谬误之处，责任具在译者，请读者诸君 mail 通知。这里先作揖谢过了！

mailto:z-l-dragon@hotmail.com

Walter Zhou
2008-1-27，晚，初稿
2008-1-31，晚，更新一些链接
上海浦东周浦