自学通系列

Linux编程24学时教程

(美) Warren W. Gay 著
潇湘工作室 译

机械工业出版社
China Machine Press

本书循序渐进地介绍Linux编程技术，包括创建make文件、Linux过滤程序及错误处理和报告、创建Linux实用程序、调试程序、使用Linux及其编译器和GNU软件，以及信号量、进程、函数和函数库的使用等。

本书适用于懂得C语言初学Linux的程序员。

本书版权登记号：图字：01-1999-3454

图书在版编目(CIP)数据

Linux编程24学时教程 /（美）盖伊（Gay,W.W.）著；潇湘工作室译. - 北京：机械工业出版社，2000.5

(自学通系列)

书名原文：Sams Teach Yourself Linux Programming in 24 Hours

ISBN 7-111-08011-4

Ⅰ.L… Ⅱ.①盖… ②潇… Ⅲ.Linux操作系统程序设计 - 教材 Ⅳ.TP316.89

中国版本图书馆CIP数据核字(2000)第20859号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：李新阳

北京市密云县印刷厂印刷 · 新华书店北京发行所发行

2000年5月第1版第1次印刷

787mm × 1092mm 1/16 · 22.75印张

印数：0 001-6000册

定价：49.00元(附光盘)

前　言

本书介绍用C语言开发Linux应用程序的要点。阅读完本书时，你将能够编写应用于Linux操作系统的专业应用程序、客户程序和服务器守护程序。

本书的重点是对于C程序员来说相当重要的Linux主题。我们的课程经过认真的组织，其内容覆盖了早期的重要主题以及许多高级的特性。每一学时都以逻辑方式进行组织，其中包含许多示例，还有系统的讲解。此外，在本书中，你将会看到明显的Linux和GNU内容特征。

本书读者

如果你是一个初学者，将发现本书是一本很好的入门教程；如果你是一位有经验的程序员，将发现本书在程序开发方面是一本方便的参考书；如果你是一位Linux的爱好者，将发现本书包含许多实际的例子，帮助你从头开始编写高质量的程序。

本书约定

本书用不同的字体来区别程序代码和常用英文，同时，帮助你注意重要的概念。

在一行代码开始处的箭头，表示上一行代码太长，在印刷版面的一行放不下，当前行是上一行的继续。你应当在上一行的结尾继续输入箭头之后的所有字符。

表示与相应讨论有关的相关信息。

提供建议或教你一种做某事的简便方法。

提醒你存在潜在问题，并帮助你排除错误。

第1学时 入 门

1.1 Linux简介

本书着重于讲解Linux中的C语言程序开发。你将学到的大部分内容，可以直接移植到那些在工作中常见的商业UNIX平台。

本节课将介绍GNU gcc编译器，并告诉你为了正确地执行编译，需要进行哪些设置；介绍一些其他编译设置，在用GNU gdb调试器调试程序时，将会用到它们；另外，介绍核心文件，并概括出使用调试器的步骤。

在本节课中，你将学到如何使用GNU gcc编译器的警告级别，以节省时间，并使程序更可靠。并且可以学到一些技巧，这些技巧可以帮助你消除在每天的编程实践中都会出现的确保C结构有效的编译器警告。最后将执行一个短的过程，来检查gcc编译器的操作，确保它能正常工作。

在开始之前，区别Linux本身与用于Linux的工具是重要的。当你在Internet上发布问题或在Usenet上寻求建议时，这个区别是很有用的。通常称Linux为UNIX核心，核心管理文件系统、内存资源、进程和设备。

另一个重要的方面是用于Linux主机的开发工具集。它们包括GNU C编译器、GNU make命令和一些用于源代码控制的RCS工具。当你在Internet发布问题或在Usenet上寻求建议时，不要将这些工具和Linux核心混淆。

1.2 GNU gcc简介

用于Linux的C编译器是GNU gcc命令。通常，应当了解你正在使用的编译器版本，而且，这在要发布问题时尤其重要。通过执行以下命令，可以方便地得到版本信息：

```
bash$ gcc --version
2.7.2.3
```

在大多数UNIX平台上，通过cc命令来调用C编译器。在Linux系统中，cc命令和gcc命令指向同一个C编译器。

GNU gcc编译器常用于编译ANSI C代码，但是它也能接受老版本的非ANSI C代码。这是一个好消息，因为与一些商业C编译器不同的是，不需要对gcc进行附加设置。

1.2.1 选择GNU标准

C程序经常包含如下形式的头文件：

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

这个语句告诉编译器读入并编译文本文件/usr/include/stdio.h。实际上，C编译器还可以读

入和编译其他的几个文件，这取决于stdio.h是否包含其他的包含文件。当编译器在它的预处理器下处理这些文件时，将会编译不同的C源代码，或者已经包含了另一个源文件。这个动作是由编译器中定义宏的当前状态决定的。GNU C库的设计者仔细地设计了这些include文件，用来定义编程的结构，这些编程结构有一些微小的变化，以适应UNIX的不同标准。

你可能听人们说过这样的话，“对于标准来说，它好就好在有许多的标准！”你将要编写和编译的标准便是GNU标准。在Unix下，这个标准是GNU C 库可用的所有各种其他标准的超集。

在这些标准之间选择需要使用宏定义。在这种情况下，你必须简单地定义一个名为_GNU_SOURCE的宏。不必为这个宏赋值，这个宏的值只有在存在时才会使用。定义这个宏的方法是在编译器命令行使用-D选项:

```
bash$ gcc -D_GNU_SOURCE hello.c
```

使用-D实际上等效于下面的C语言语句:

```
#define _GNU_SOURCE
```

由编译器产生的默认可执行输出文件为a.out。

1.2.2 指定编译器输出

编译示例大多使用默认输出文件a.out。不过，正常情况下，可以明确地指定输出文件的文件名。这可以通过-o选项来完成:

```
bash$ gcc -D_GNU_SOURCE hello.c -o hello
```

1.2.3 运行编译后的程序

编译完一个程序，就可以运行它。为了运行它，可以使用外壳来调用它:

```
bash$ ./hello
```

上面命令行中的点和斜杠可能是多余的。不过，使用这种方法是一个好的习惯。当存在与可执行文件同名的文件时，这样做可以避免运行另一个文件。当你以root身份登录时，这个习惯还可以避免运行特洛伊木马程序的可能性。

1.3 用于调试的编译

在Linux下，标准的调试工具是gdb命令。你应当在系统上检查它的可用性，因为当开发新程序时，可能会用到它。在一个程序结束时，执行事后的分析是相当有用的。可以按以下命令检验它:

```
bash$ gdb --version
GDB 4.16 (i586-unknown-linux), Copyright 1996 Free Software
➥ Foundation, Inc.
bash$
```

如果得到如下结果，则表明gdb没有安装，或者外壳PATH环境变量设置不正确。

```
bash$ gdb
bash: gdb: command not found
bash$
```

无论选择什么样的调试工具，都需要设置编译器的选项，以保证将特定的调试信息写进最终的可执行文件中。当开发新程序，或调试有问题的程序时，通常在编译时都带有调试选项：

```
bash$ gcc -g -D_GNU_SOURCE hello.c -o hello
```

选项-g使得最终可执行文件包含一些附加信息，以便于调试器将机器指令和源程序对应起来。如果编译时不带-g选项，调试器就不能显示一些有用的信息。

大多数调试器有许多命令，这里只介绍一些重要的基础知识。

当一个程序运行并遇到错误时，Linux核心将终止这个程序。当这种情况发生时，操作系统将在当前目录下产生一个名为core的文件。这个core文件是错误发生时的内存映像，是执行postmortem分析的关键。

程序清单1-1显示了一个简单程序。第2行和第4行代码是有意编成引起内存冲突的。第2行设置指针cp为空指针(null)。第4行试图将有定义的点存储到空指针cp。因为指针cp为空，Linux核心将产生一个core文件，程序也将终止运行。

程序清单1-1 01LST01.c——具有属性的一个C程序

```
1:    static void fun1(void) {
2:        char *cp = 0;
3:
4:        *cp = '!'; /* Take that! */ (取值)
5:    }
6:    int main(int argc,char **argv) {
7:        fun1();
8:        return 0;
9:    }
```

编译这个程序并运行，可以看到它将终止。这时Linux核心产生了一个core文件以便于分析：

```
bash$ gcc -g -D_GNU_SOURCE 01LST01.c -o wipe_out
bash$ ./wipe_out
Segmentation fault (core dumped)
bash$
```

在这个过程中，细节是很重要的。可以看到程序终止在错误段。这是问题的第一个线索。错误段产生的原因是有一个严重的内存使用错误，这一般意味着程序试图操作一个无用指针，通常这个无用指针会是一个空指针。

在这种情况下，问题的来源在于程序清单1-1的第4行。指针cp为空，但程序却试图存储有定义的字符到这个指针。Linux操作系统不但禁止这种用法，而且会产生一个错误段，并输出core文件。

```
bash$ ls -l core
-rw-------   1 student1 user        98304 Nov 25 23:41 core
```

如果这个程序由几千个C语言函数组成，检查错误的来源是非常烦琐的。这样，算后检查(postmortem)显得非常重要。以下是算后检查的分析结果:

```
bash$ gdb ./wipe_out core
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for
➥ details.
GDB 4.16 (i586-unknown-linux), Copyright 1996 Free Software
➥ Foundation, Inc...
Core was generated by './wipe_out'.
Program terminated with signal 11, Segmentation fault.
#0  0x8048550 in fun1 () at 01LST01.c:4
4           *cp = '!'; /* Take that! */
(gdb) where
#0  0x8048550 in fun1 () at 01LST01.c:4
#1  0x8048568 in main (argc=1, argv=0xbffffb54) at 01LST01.c:7
(gdb) quit
bash$
```

执行算后检查时将调用gdb，并将前面出现的可执行文件名作为第1个命令行参数。第2个命令行参数必须为Core文件的名字。在调用调试器后，gdb同时分析可执行文件和Core文件。gdb将确认自身的版本为4.16，而且程序./wipe_out产生Core文件。它将重现由错误段产生Core文件的事实。最终，gdb显示错误发生在01LST01.c的第4行。然后将显示第4行、注释及所有内容。

有时，知道函数之间的调用关系对解释它最初如何被调用是很重要的，而其他时候，这个信息有助于重现错误。通过输入where命令将得到堆栈的跟踪信息。如上所示，main()在01LST01.c的第7行调用了fun1()，main()函数的参数甚至也显示了出来。所有这些有用的信息，显示了程序终止的真正原因。

要退出调试器，只要简单地输入quit命令，控制将返回外壳。gdb还有很多功能，但上面介绍的是最实用的。

1.4 检查编译器的警告级别

C编译器通常会报告一些消息，它们可分为错误消息和警告消息。错误消息是指那些只有改正才能成功编译的信息，而警告消息是用来指出程序员的一些不良编程习惯和程序在运行时可能发生的问题。

在gcc最高级警告级别下，编译器报告最小的差错，但不一定都完全正确。有时警告是为了确保C的编程习惯，有些开发者会在编译器中取消产生这些警告的设置。不过，这样做会阻止C编译器产生一些有用的建议。

最好的做法是采用可用的最高级警告级别。这样做可以迫使开发者在消除所有编译警告以后发布源代码。只有当你采用了别人的源代码，并且没有足够的时间处理这些警告时，才可以采用较低的警告级别。

通常在编译时，总是将警告级别设在最高级。处理警告消息的时间可以因调试程序时间的缩短而得到补偿。

下面的命令表示如何设置警告级别为最高级:

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

注意命令行中-Wall选项的使用。-W选项是用来为gcc编译器设置警告选项的，后面的参数设置可用的警告级别。在GNU gcc命令下，可以多次设置-W选项，这样，在不同的情况下，可以设置不同的警告级别。最好的习惯是只使用-Wall选项，并修改程序，直到所有的警告都完全解决。在本书中将采用这个方法，这有助于在学习过程中形成良好的C语言习惯。

大多数Linux命令行选项，在选项字母和选项参数之间不需要空格。例如，-Wall和-W all这两种设置选项是等价的。

1.5 处理编译器的警告消息

编译器使用高级警告级别时，将报告每一个可能的警告消息。一个低级的警告级别，只报告最重要的消息，而忽略掉其他消息。

在C编译器中使用高级警告级别时，有一个缺点，因为有时一些警告消息只是为了确保C语言的结构。不过，精心设计的编译器可以处理这些问题，GNU gcc在这方面就做得非常好。

1.5.1 关于赋值的警告

程序员一般喜欢C语言的表达简洁。这是指程序员常采用最少的语句和操作来完成一个任务，有时，在一步中既能进行赋值又能进行非零测试。例如程序清单1-2的if语句。

程序清单1-2 01LST02.c——IF语句中有关赋值的警告

```
1:    #include <string.h>
2:
3:    char *
4:    Basename(char *pathname) {
5:        char *cp;          /* Work Pointer */
6:
7:        if ( cp = strrchr(pathname,'/') )
8:            return cp + 1; /* Return basename pointer */
9:        return pathname;   /* No directory component */
10:   }
```

下面是程序清单1-2的编译会话结果:

```
bash$ gcc -c -D_GNU_SOURCE -Wall 01LST02.c
01LST02.c: In function 'Basename':
01LST02.c:7: warning: suggest parentheses around assignment used as
➥ truth value
bash$
```

在程序清单1-2 中，注意第7 行的语句。编译器提示这个语句是一个潜在的错误，因为C程序员通常采用比较运算符“==”而不是采用赋值符“=”来比较不同的值。编译器无法确认第7行是真正的赋值，还是要进行比较。在编译器发出警告后，如何处理这个警告将留给开发者来完成。

-c编译选项告诉编译器不要链接最终程序。结果是源代码被编译成中间目标文件，以便在后面的链接中使用。

注意，这个语句不能正确执行，或不能清楚地表达程序员的真实意图。有人可能会争辩说，那个if语句中的比较是正常的，而赋值是不合适的，应当发出一个错误消息。事实上，C语言定义了这样两种等效的表达方式。

编译器的编写者采用了聪明的技巧来处理这类事件。这种特定的情况是这样解决的：如果一个赋值运算符的运算如程序清单1-2所示的那样，因为它表明在编程方面可能存在错误，所以将给出一个警告消息。如果它确实表示一个错误，程序员将回过头来，将单个等号改为双等号，并重新编译代码。相反，如果这个赋值运算是正确的，程序员将用括号将赋值运算括起来。当这些工作完成后，编译器就认为程序员知道自己在干什么。

程序清单1-3和程序清单1-4显示了解决这种赋值的两种不同解决方案。

程序清单1-3 01LST03.c——解决IF语句警告的一种方式

```
1:    #include <string.h>
2:
3:    char *
4:    Basename(char *pathname) {
5:        char *cp;           /* Work Pointer */
6:
7:        if ( ( cp = strrchr(pathname,'/') ) )
8:            return cp + 1; /* Return basename pointer */
9:        return pathname;    /* No directory component */
10:   }
```

程序清单1-4 01LST04.c——避免赋值警告的首选方式

```
1:    #include <string.h>
2:
3:    char *
4:    Basename(char *pathname) {
5:        char *cp;     /* Work Pointer */
6:
7:        if ( ( cp = strrchr(pathname,'/') ) != 0 )
8:            return cp + 1; /* Return basename pointer */
9:        return pathname;    /* No directory component */
10:   }
```

注意在程序清单1-3 和1-4 的第7行中的附加圆括号。在这里，C语言本身不需要这个圆括号，但编译器采用这种方法，提示开发者注意他在干什么。程序清单1-3显示了一种GNU gcc编译器可接受的解决方案，而一些其他的商业编译器坚持采用程序清单1-4的结构。出于这个原因，程序清单1-4的方案应为首选，而且对读者来说也更容易理解。

不必为了优化而简化C语言的表达式。即使没有程序员的帮助，现在具有优化功能的编译器也可以有效地达到优化代码的效果。这样，使表达式更易读比采用最少的操作运算符更可取。

上面虽然只以C语言的if语句为例，但在C语言的其他语句中也存在同样的情况，比如

switch语句和while语句。

1.5.2 关于未用参数的警告

有些编译器提示未用参数。这是出于这样的考虑：如果定义了一个参数，就意味着要使用这个参数。问题的本质在于函数的参数定义了一个接口。定义这种没有真正使用的接口，是因为这个接口可能在将来会用到。

普遍存在的main()程序就是未用参数问题的一个例子。main()程序接口的定义通常如下所示：

```
extern int main(int argc,char *argv[]);
```

如果所编程序未用这些参数，难道意味着这些参数不存在？当然不是。这样，通常在函数中定义它们是可取的，无论这些参数能否用到。

不过，如果编译器提示未用参数，你就有可能去取消这些定义。因此，程序清单1-5表示了如何在不必删掉函数参数的情况下避免这种问题。

程序清单1-5 01LST05.c——适当地解决未用的参数

```
1:    #include <stdio.h>
2:    int main(int argc,char **argv) {
3:
4:        (void) argc;
5:        (void) argv;
6:
7:        puts("Hello World!");
8:        return 0;
9:    }
```

如果你了解C语言的编程规则，就一定知道在一个语句中引用它自身是完全有效的。正常情况下，因为它没有什么作用，所以这种结构是无用的。不过，它可以对编译器产生作用。确切的方法即是引用一个参数的值，然后通过void方式舍弃它，来告诉编译器参数被引用了，但你更乐于舍弃这个引用值。

当前的GNU gcc编译器不警告未用参数。这种处理可能是、也可能不是一种特色，这取决于你的观点。也许gcc的编写者选用了不警告未用参数的方式，这样程序员就不必从接口定义中去掉这些变量。不过，将来如果gcc在这种情况下会改变它的使用方式，你要为此作好准备。

1.5.3 解决关于未用变量的警告

有时，编译器会警告存在着代码中已声明但未使用的变量。这些警告可能强烈地诱使你立即从代码中去掉这些变量。不过，这样做之前，你一定要当心。

在去掉未用变量和缓冲时要特别小心。在你认为这些变量永远不会被用到之前，要确保你已经使用C预编译器全面检查了源代码。有时，在不同的宏设置下编译一个程序，将会用到这些变量声明。

未用变量的问题通常出现在可移植到不同UNIX平台的代码中，包括Linux。这是正常的，

即使有正确的C预处理器的帮助，最初的开发者也不会允许存在这些未用的变量。在实践中经常发生的情况是，人们将源代码分成不同的块，由几个人来修改，而从来不会在其他平台上全面测试这些变化。

为了说明这个问题，参见程序清单1-6。

程序清单1-6 01LST06.c——未用变量声明的示例

```
1:    #include <stdio.h>
2:    #include <unistd.h>
3:    #include <sys/types.h>
4:    /* #define SHOW_PID */ /* MACRO NOT DEFINED */
5:
6:    int main(int argc,char **argv) {
7:        pid_t PID;        /* Process ID */
8:
9:        (void) argc;
10:       (void) argv;
11:
12:   #ifdef SHOW_PID
13:       PID = getpid();   /* Get Process ID */
14:       printf("Hello World! Process ID is %d\n",
15:           (int)PID);
16:   #else
17:       puts("Hello World!");
18:   #endif
19:       return 0;
20:   }
```

程序清单1-6的编译会话结果如下：

```
bash$ gcc -c -D_GNU_SOURCE -Wall 01LST06.c
01LST06.c: In function 'main':
01LST06.c:7: warning: unused variable 'PID'
```

如果像上面所示的那样，将第4行作为注释来编译这个程序，编译器将提示第7行定义的变量PID没有使用。这是因为宏SHOW_PID没有定义，结果编译的是第17行，而不是第13到15行。如果你在意那个警告消息，并去掉第7行或将第7行作为注释，那么解决了当前的问题，但可能导致一个长期的问题。

如程序清单1-7所示，在第7行加上注释标记，并去掉第4行的注释标记(定义宏SHOW_PID)后，编译这个程序，编译器将编译第13~15行，而不是第17行。在这种情况下，编译器报告一个错误：变量PID没有声明。

程序清单1-7 01LST07.c——具有丢失变量声明的错误编译的程序

```
1:    #include <stdio.h>
2:    #include <unistd.h>
3:    #include <sys/types.h>
4:    #define SHOW_PID      /* MACRO IS DEFINED */
5:
6:    int main(int argc,char **argv) {
7:    /*  pid_t PID;        // Commented Out */
8:
9:        (void) argc;
10:       (void) argv;
```

```
11:
12:   #ifdef SHOW_PID
13:       PID = getpid();   /* Get Process ID */
14:       printf("Hello World! Process ID is %d\n",
15:           (int)PID);
16:   #else
17:       puts("Hello World!");
18:   #endif
19:       return 0;
20:   }
```

程序清单1-7的编译会话结果如下:

```
bash$ gcc -c -D_GNU_SOURCE -Wall 01LST07.c
01LST07.c: In function 'main':
01LST07.c:13: 'PID' undeclared (first use this function)
01LST07.c:13: (Each undeclared identifier is reported only once
01LST07.c:13: for each function it appears in.)
```

当遇到这个变量名字时，编译器的错误消息立刻表明第13行的变量PID没有定义。这表明在最初有警告消息时，取消变量声明是错误的解决办法。问题的本质在于要进行必要的C预编译处理。程序清单1-8给出了改正这个问题的正确方法。

程序清单1-8 01LST08.c——解决未用变量的正确方式

```
1:    #include <stdio.h>
2:    #include <unistd.h>
3:    #include <sys/types.h>
4:    #define SHOW_PID      /* MACRO IS DEFINED */
5:
6:    int main(int argc,char **argv) {
7:    #ifdef SHOW_PID
8:        pid_t PID;        /* Process ID */
9:    #endif
10:
11:       (void) argc;
12:       (void) argv;
13:
14:   #ifdef SHOW_PID
15:       PID = getpid();   /* Get Process ID */
16:       printf("Hello World! Process ID is %d\n",
17:           (int)PID);
18:   #else
19:       puts("Hello World!");
20:   #endif
21:       return 0;
22:   }
```

注意在附加的第7行和第9行之间的第8行声明了相应变量。这两个附加的预编译指令使得程序在编译时没有警告，无论宏SHOW_PID定义与否。

1.5.4 解决关于未用字符串的警告

未引用字符串常量也会引起警告。有时程序员将字符串常量留在程序中，这样未用字符串会成为可执行文件的一部分。通常会在程序中定义版本信息，这样可执行文件可以进行转储，并和源模块特定的版本相匹配。

这类警告的解决方案是用关键字const定义未用字符串为常量，编译器不警告未使用的常量。程序清单1-9显示了一个未引用的嵌入式RCS字符串。(注意：嵌入式RCS字符串在第2学时“管理源代码”介绍。)

为了减少编译器对未引用字符串常量的警告，可以简单地用关键字const定义未用字符串为常量。

程序清单1-9 01LST09.c——未引用RCS字符串的示例

```
1:    static char rcsid[] = "@(#)01LST09.c $Revision: 1.3$";
2:
3:    #include <stdio.h>
4:    int main(int argc,char **argv) {
5:
6:        (void) argc;
7:        (void) argv;
8:
9:        puts("Hello World!");
10:       return 0;
11:   }
```

程序清单1-9的编译会话结果如下：

```
bash$ gcc -c -D_GNU_SOURCE -Wall 01LST09.c
01LST09.c:1: warning: 'rcsid' defined but not used
```

注意，在第1行声明了字符串数组rcsid。在后面，在介绍源代码控制时，我们将会明显地看到声明这个数组的目的。在这个程序中没有引用这个字符串，因而编译器发布一个警告。你可简单地加一个关键字const来解决这个问题，如程序清单1-10所示。

程序清单1-10 01LST10.c——消除未用字符串常量警告

```
1:    static const char rcsid[] = "@(#)01LST10.c $Revision: 1.3$";
2:
3:    #include <stdio.h>
4:    int main(int argc,char **argv) {
5:
6:        (void) argc;
7:        (void) argv;
8:
9:        puts("Hello World!");
10:       return 0;
11:   }
```

1.6 测试C编译器

在开始Linux的编程之前，最好用一个简单的测试程序来测试一下编译器。通常在软件没有正确安装和管理时，软件本身或它的开发者将遭到责骂。这里，我们将再次使用熟悉的“Hello World”程序。这个程序将测试编译器是否正确设置并能正确工作。

当软件存在问题时，在向它的开发者发邮件或向Usenet发消息之前，首先

> 要确保你的软件安装完好并且操作正确。一般情况下，这类问题都是由于没有按照正确步骤操作引起的，尤其是软件的安装步骤。

用你喜欢的编辑软件(比如vi)输入简单的hello.c程序，或者使用配套光盘中的hello.c文件。程序清单1-11如下：

程序清单1-11 HELLO.C——简单的"Hello World" 签出(checkout)程序

```
1:    #include <stdio.h>
2:    int main(int argc,char **argv) {
3:
4:        (void) argc;
5:        (void) argv;
6:
7:        puts("Hello World!");
8:        return 0;
9:    }
```

> 如果你自己输入程序，记住不要输入行标志。在这里显示它们只是为了说明方便。

按照程序清单1-12所示编译、链接并执行这个程序，完成这个问题分析过程需要三步。

程序清单1-12 Hello.c的Linux编译、链接和执行

```
1:    bash$ gcc -c -Wall -D_GNU_SOURCE hello.c -o hello.o
2:    bash$ gcc hello.o -o hello
3:    bash$ ./hello
4:    Hello World!
5:    bash$ echo $?
6:    0
7:    bash$
```

程序清单1-12的第1行表示将源程序编译成目标程序hello.o，第2行表示链接，由目标文件和C库一起生成可执行文件hello。

第3行调用新编译的C程序并执行它，第4行表示程序的响应结果。最后，第5行用来显示程序的结束标志码，其值为0(见第6行)。如果可以毫无问题地执行到第7行，你就可以继续学习Linux的后续课程了。

解决编译器中的问题

如果在程序清单1-12的第1行遇到了问题，这通常表明编译器没有正确安装，或者/usr/include目录的包含文件丢失了或不正确。

如果第2行不能正确执行，这表明编译器、链接器或库文件有错误。通常都是C的库文件丢失了或没有正确安装。

如果编译和链接都正确，而在第3行不能正确执行，这表明系统中的库文件不正确或存在冲突。

第6行显示的值应当为0。如果不是0，首先用return语句检查hello.c文件的返回值是否为0。同时要确保main()函数的返回类型不是void。如果这些都正常，就说明C库存在错误。

只有当命令或程序刚执行完时，使用外壳变量“$?”才有效。

但愿你的编译器和链接器都正常，处理好这些是学习后面课程的基础。

1.7 课时小结

在这一课，我们熟悉了Linux gcc编译器，并检验了一些编译器选项，了解了定义C宏_GNU_SOURCE的原因，并看到了-g选项在调试中的作用。我们也学习了如何使用gdb调试器分析Core文件，并确定问题的来源。

另外，我们学习了在编译中使用高级警告级别的优点，也看到了使用高级警告级别的一些结果，并学习了如何避免不必要的警告。

1.8 专家答疑

问题：在main程序中，有些不会用到的参数，为什么还要声明它？

解答：在你的函数中，像接口本身那样声明它是很重要的。在未来的源代码版本中，函数的实现可能改变，这取决于定义的参数。

问题：为什么在C语言的源程序中过分注意效率不是一个好习惯？

解答：使程序易读和易懂是首要的，不必帮助编译器提高编译效率。对于编译效率来说，现在的编译器优化效果是相当好的。

问题：为什么当编译一个需要调试的程序时要用-g编译选项？

解答：-g选项可使编译器在目标代码中产生附加的信息。这些信息帮助gdb调试器定位源程序语句，并提供变量名称的信息等等。

1.9 课外作业

下面的练习有助于巩固在这节课中学到的知识，在开始下一课之前完成这些练习。

1.9.1 思考题

1) GNU gcc是编译器还是链接器或者两者都是？

2) 如何称呼一个*.c文件？如何称呼一个*.h文件？

3) 如何称呼一个*.o文件？

4) a.out是哪一类文件？

5) 编译器的错误消息和警告消息有什么不同？

6) 为什么在调用一个可执行文件时，最好加上点斜杠?为什么这一点对根目录下的用户尤其重要？

7) 用void类型转换意味着什么?为什么这样做是有用的？

1.9.2 练习题

1) 不用改变源代码，编译并执行程序清单1-6，除了显示Hello World信息外，还显示进程ID。

2) 再次使用程序清单1-6的源代码，加上选项-DSHOW_PID=13，编译并执行。与第1) 题的结果比较有没有不同?为什么？

第2学时　管理源代码

2.1　使用源代码管理的好处

UNIX平台一直被认为是一个拥有大量开发工具的平台。在它庞大的工具库中包括一些用来管理不同版本的源代码的工具，这些工具在Linux中也可以找到。

尽管拥有这些源代码管理工具，但是，很多程序员都不愿意使用它们。很多时候，他们都是等到参加了一个大型的项目或者与其他开发者合作时才会被迫使用这些工具。没有预先使用源代码管理工具是令人遗憾的，因为它们可以为个人开发者提供很多好处，当然也包括一个开发小组。

2.2　有哪些可以选择的工具

在Linux平台下，主要有以下三种选择：

- SCCS(Source Code Control System，源代码控制系统)，来自Marc Rochkind的The source code control system(源代码控制系统)，IEEE Trans. Software Engineering(软件工程)，1975
- RCS(Revision Control System，版本控制系统)，来自Walter F. Tichy的RCS——A System for Version Control,Software——Pratice & Experience(版本控制系统，软件——实践与经验) 15,7(1985年7月),637 ~ 654
- CVS(Concurrent Versions System，并发版本系统)，其作者包括：

Dick Grune：他是最早的cvs 外壳脚本的作者，该脚本发表在comp.sources.unix第六卷上(1986年12月)。Dick因提出了很多冲突解决算法而知名。

Brian Berliner：在Dick的工作的基础上，他于1989年4月设计和编写了cvs程序。

Jeff Polk：他协助Brian设计了CVS系统并提供了其他帮助。他还是checkin(1) 外壳脚本的作者，该脚本就是“cvs import”的前身。

SCCS是UNIX中最早的源代码管理工具之一。在本章中我不打算介绍它，因为它缺乏灵活性，不能生成源代码变化的不同分支(例如两个用户想同时修改同一段源代码)。SCCS只考虑了最简单的一种情况，那就是对于任何一个源文件，在任何时间都只有一个用户可以修改它。

RCS是作为SCCS更好的和更强大的更新产品出现的。RCS非常灵活，而且也很好用。我们在这里将要介绍的就是RCS。

CVS建立在RCS的框架的基础之上，并且可以在网络上运行。许多大型的Internet开发工作都是在一个可以被所有参加者访问的CVS服务器的帮助下进行的。不过CVS的建立和管理非常复杂，本书就不再涉及了。

2.3　为项目设置RCS

为了学习如何使用RCS，我们将开始编写一段程序，在第3学时中，还要进一步扩展这个

程序。这样现在就有了一个新的源模块可以进行试验。

一般情况下，为项目创建一个专用目录是明智的选择。创建这样一个专用目录使得我们可以把所有与项目有关的文件都保存在一起，而同时把所有与项目无关的文件分开。我们把创建一个新项目的步骤总结如下，并以程序清单2-1为示例:

1) 选择项目名。

2) 用pwd命令查看你现在所在的目录。如果现在不是在正确的目录下，那么可以用cd命令转到合适的目录中去。

3) 根据所选择的项目名，用mkdir命令创建一个新的目录。

4) 用cd命令进入新的项目目录。

程序清单2-1 创建项目目录

```
1:    bash$ cd
2:    bash$ pwd
3:    /home/student01
4:    bash$ mkdir dos_cvrt
5:    bash$ cd ./dos_cvrt
6:    bash$ pwd
7:    /home/student01/dos_cvrt
8:    bash$
```

程序清单2-1中介绍了如何为项目创建目录。根据程序概括的要点，具体步骤如下:

1) 选择项目名dos_cvrt。

2) 第1行和第2行介绍了如何建立和查看当前所在目录。第1行中的cd命令的作用的是移动到根目录下；pwd命令的作用是查看当前所在的目录。

3) 第4行介绍了如何创建名为dos_cvrt的项目目录。

4) 第5行介绍了如何移动到新建的项目目录下。第6行中使用了pwd命令以确认当前已在新建的项目目录下。

如果定义了CDPATH这个外壳变量，那么，当想移动到一个子目录下的时候，应该使用点斜杠表示法来表示子目录。CDPATH是可选的外壳变量，它记录了你可能经常进入的目录列表。这种表示方法使得只用根据缩写的目录名就可以进入这些目录。

在定义了CDPATH变量以后，有时，你会灵活地进入到目录中。点斜杠表示法可以确保能够相对于当前的目录来移动目录。

2.4 创建新的源文件

现在可以开始处理一个新的源文件了。既可以使用你喜爱的编辑器来创建如程序清单2-2所示的文件，也可以直接将本书所附光盘中的02LST02.c文件复制到新目录下，并取名为dos_cvrt.c。程序清单2-2中列出了这个程序的雏形。

程序清单2-2 02LST02.c——最初的DOS_CVRT程序

```
1:    /* dos_cvrt.c */
2:    #include <stdio.h>
```