Linux 平台的 C 开发环境介绍

王晓林

July 27, 2016

Contents

1 Linux 平台的经典 IDE **1**

2 GCC 使用入门 **3**
2.1 从 hello.c 到 hello.exe 5
2.2 常用编译选项 9
2.2.1 `-Wall` 9
2.2.2 `-E` 10
2.2.3 `-D` 11
2.2.4 更多选项 13

1 Linux 平台的经典 IDE

IDE 这个词大家都不陌生吧，Integrated Development Environment, 集成开发环境。说到 IDE，估计你会想到 Windows 平台的 Visual Studio，或者 Java 开发常用的 Eclipse。今天我向大家推荐一下世界上最好的 IDE。最好的 IDE？你一定开始怀疑我是在吹牛了吧。那么，好吧，我们先看看到底什么是 IDE。很简单，一个 IDE 无非包括如下一些功能模块：

- 一个编辑器；
- 一个编译器；
- 一个调试器；
- 其它一些辅助功能，比如用鼠标拖控件。

什么是最好的 IDE？那肯定是：

最好的 IDE = 最好的编辑器 + 最好的编译器 + 最好的调试器

上面我们提到的 VS 和 Eclipse 都没能把最好的东西集成到一起，所以，我说它们不是最好的。有哪个 IDE 做到这一点了吗？只有 Emacs。

Emacs 是一个有数十年悠久历史的开源编辑器。不折不扣地说，它也是世界上最强大的编辑器，因为它是模块化设计，你如果觉得它还缺少什么功能，那么

你可以给它添加一个新的功能模块。如此日积月累几十年，凡是你能想到的功能都已经添加好了。

Emacs 可以很方便地调用世界上最牛的编译器(gcc)和调试器(gdb)。

Emacs + gcc + gdb

这就是世界上最好的 IDE。也许你会说「Emacs 不能拖控件啊」。没错，但在我看，拖控件并不总是一个受人欢迎的功能，至少在系统编程的时候，它毫无用处。

而且，从学习的角度来说,「用鼠标编程」绝对是一个非常恶劣的习惯，因为这根本就是在逃避学习。「鼠标化的 IDE」隐藏了很多学生应该了解的技术细节。我们学院的绝大多数学生居然不知道 C 程序是要编译之后才能运行的。他们以为写好了程序，只要「按那个“感叹号”按钮」就可以了。这就是「鼠标教学」的成果。Emacs 可以帮助你克服「鼠标依赖」，强迫你熟练地使用键盘。

更重要的是，Emacs 不只是个 IDE，它是个 ICE(Integrated Computing Environment，这名字是我刚编出来的)。Emacs 的设计目标就是，你装了个 Unix 或者 Linux 系统，不需要装任何其它软件，只要装一个 Emacs 就够了，它能帮助你完成所有的任务。也就是说，除了编程，你还可以用它写论文、做幻灯片、浏览网页、收发邮件、聊天、听歌、看照片、玩游戏……目前，好像除了直接在 Emacs 里看电影还不行，其它的都实现了。

Emacs 如此「大一统」的设计目标显然有违 Unix 的设计原则，do one thing, and do it well (做一件事，并且把它真正做好)。但好在 Emacs 是模块化的，它的每一个功能模块都绝对遵循 do one thing, and do it well 原则。你不需要哪个功能，可以不装那个模块。

另外，还是从学习的角度来说，Emacs 的学习曲线貌似比其他 IDE 要长不少，但是你

- 不必学习 VC 去写 C/C++，
- 不必学习 eclipse 去写 Java，
- 不必学习 MS-Word 去写报告、幻灯片，
- 不必学习……

一句话，“Everything Emacs”，你可以省下大量不必要的学习时间。人生苦短，何必让你的生活被 VC/eclipse/MS-Word 搞得头昏脑胀呢？**简单而强大，本就是计科专业学生和非专业学生应有的不同。**

Emacs 绝对强大，但是否「方便」就不好说了。因为「方便」是一个很主观的概念。反正，作为一个 18 年的老用户，我肯定觉得方便。其他 IDE 太无聊了，那么花哨而庞大的东西，却只适用于应用层编程。既不能用来写论文，又不能做幻灯片，更不能用来听歌、玩游戏。生活也太没有乐趣了。

最后一点，Emacs 还是一个巨大的开放社区，在这里你能结识到更酷一些的程序员。

Emacs 入门还是很简单的，它自带了一个基础教程。打开 Emacs，按 `Ctrl-h t`, 教程就出现在你面前了。照着它边看边练，英文不太困难的话，一个小时应该可以走一遍了。之后，

- `Ctrl-h i m emacs` 就可以调出详细的 Emacs 使用手册;
- `Ctrl-h i m emacs lisp intro` 可以调出 Emacs Lisp 入门教程;
- `Ctrl-h i m elisp` 可以调出完整的 elisp 编程手册。

当然，Google 永远是你最好的帮手。Happy emacsing!

2 GCC 使用入门

GNU Compiler Collection (GCC) 是 GNU 自由软件项目开发出的一整套编译器，包括

- gcc: c 编译器
- g++: c++ 编译器
- gcj: java 编译器
- gfortran: Fortran 编译器
- GNAT: ada 编译器
- gccgo: Go 编译器
- 更多其它语言的编译器

注意 GCC 和 gcc 的区别。当我们说 GCC 的时候，是在说那一整套编译器；而当我们说 gcc 的时候，是在说 c 编译器。gcc 的开发者就是 GNU 自由软件运动的创始人，大名鼎鼎的 Richard Stallman。gcc 于 1987 年问世，自诞生以来就广受推崇。现在，它仍然是诸多 Unix-like 操作系统的标准配置。Linux 内核的数千万行源代码就是用 gcc 编译的。

gcc 很强大，它支持包括 x86, ARM 在内的数十种硬件架构，并且支持交叉编译，也就是说，你可以在 x86 平台上写程序，然后把它编译成能在 ARM 平台上运行的二进制文件。如果你的系统里已经装好了 gcc，那么你可以用 `man gcc | wc -l` 命令来数一数 gcc 手册的长度。这一万六千多行的庞大手册从侧面说明了 gcc 功能的强大。

不过，作为初学者，我们并不必关心 gcc 有多强大，少数几个简单的命令选项，就足以应付我们的 c 程序编译了。假设你有个 c 文件叫 `hello.c`,

```c
#include <stdio.h>

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

那么，在最通常的情况下，你只需要:

```
gcc hello.c
```

就可以得到一个名叫 a.out 的可执行文件了。如果你不喜欢 a.out 这个古老的名字，那么

```
gcc hello.c -o hello
```

就可以得到一个名叫 hello 的可执行文件了。你应该猜到了，选项"-o" 代表 output,"输出"的意思。要运行 hello 看看结果的话，

```
./hello
```

就可以了。./ 代表当前目录，也就是你的 hello.c 所在的目录。

当然，生活并不总是像 Hello, world! 这样简单。比如说，还是 hello.c,

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello, world!\n")
6   return 0;
7 }
```

你敲完 gcc hello.c, 一回车，意外地看到了如下一些东西:

```
1  hello.c: In function ‘main’:
2  hello.c:6:3: error: expected ‘;’ before ‘return’
3    return 0;
4    ^
```

OMG! 怎么办？第一，别慌；第二，别懒。其实，上面这几行输出并没有几个单词，而且差不多都认识，静下心来仔细看看，还是很好理解的嘛。

1. 第一行意思是说，在函数 main 里面发现了点问题；
2. 第二行和第三行具体给你指出了出错的地方，在第 6 行，第 3 列，r 的前面应该有个分号；
3. 第四行的 ^ 就是个向上的箭头，指向 r, 也就是问题点所在。

怎么样，不太困难吧？重新编辑你的 hello.c, 在 return 与) 之间加上分号，

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello, world!\n");
6   return 0;
7 }
```

然后再编译一下，

```
1  hello.c: In function ‘main’:
2  hello.c:5:3: error: stray ‘\357’ in program
```

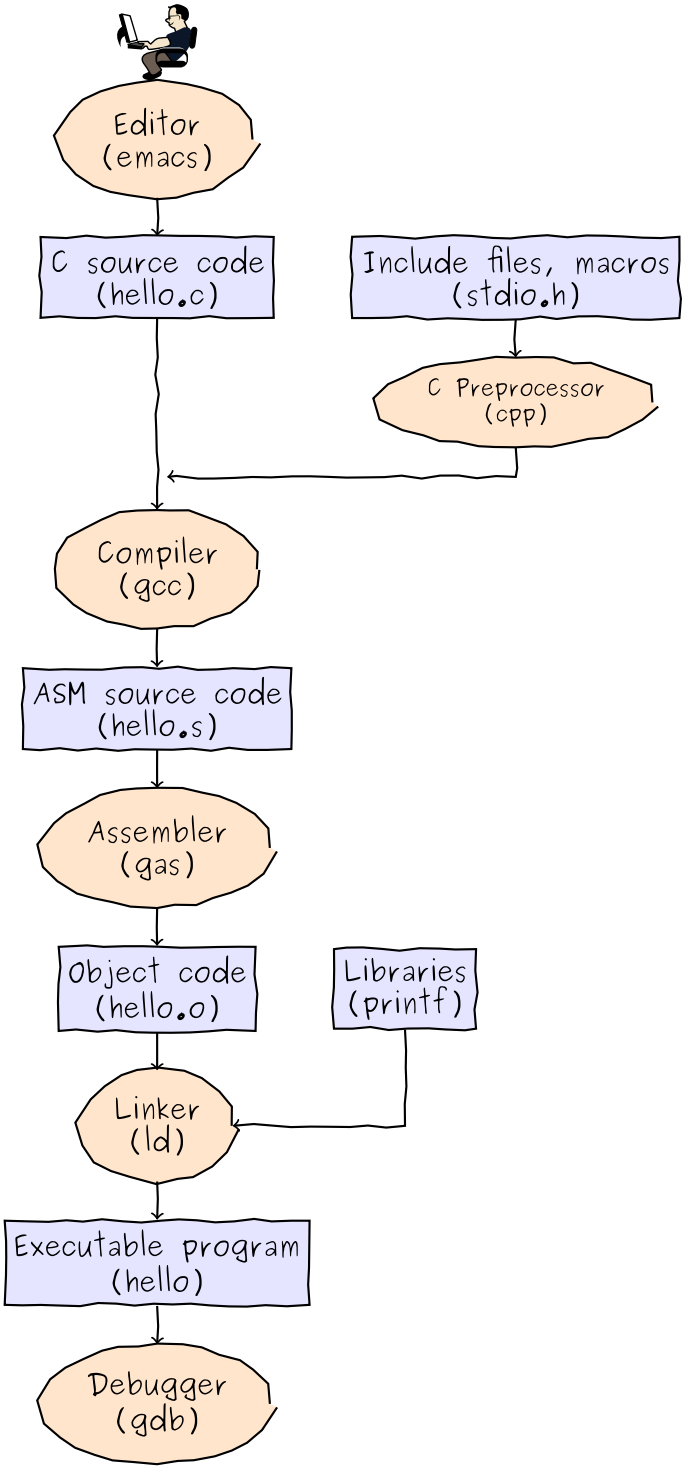
```
3       printf("Hello, world!\n");
4       ^
5   hello.c:5:3: error: stray ‘\274’ in program
6   hello.c:5:3: error: stray ‘\233’ in program
7   hello.c:6:3: error: expected ‘;’ before ‘return’
8     return 0;
9     ^
```

OMG!!! 怎么问题越来越多了？第一，别慌；第二，别懒；第三，别马虎。上面出错信息的最后三行你都见过了，显然，分号还是有问题。再仔细看看，那居然是个中文分号！你怎么可以这样马虎呢？

在初学者当中，类似上面这样的小错误是屡见不鲜的。怎么办呢？别慌、别懒、别马虎，静下心来做事情，这就够了。

2.1 从 hello.c 到 hello.exe

我们在命令行敲完 `gcc hello.c -o hello.exe`，然后一回车，不出错误的话，一个可执行文件，`hello.exe`，就诞生了。现在我们简要了解一下，敲完回车之后，电脑里到底发生了什么。换句话说，就是了解一下我们常说的「编译」到底是怎么回事。



上图中，椭圆框里面放的都是工具，包括

编辑器 我们用的是 Emacs，世界上最强大的编辑器。当然，你也可以用 nano, vim, 或者其它什么编辑器。毕竟写一个 `Hello, world!` 并不必非要用那么高大上的工具。但如果你以后想当个程序员，那么 Emacs 就应该是你的首选利器。

C 预处理器 C 预处理器 (c preprocessor) 负责处理程序中以 # 开头的程序语句，比如说：

- `#include<stdio.h>`。怎么处理？你肯定知道 `include` 是「包含」的意思，也肯定知道 `stdio.h` 是一个文件的名字[1]。那么，`#include<stdio.h>` 显然就是要把 `stdio.h` 文件的内容包含到你的程序中来。我们用的预处理器是 cpp，在命令行敲：

  ```
  cpp hello.c
  ```

  看到了吗？原来程序中的第一行 `#include<stdio.h>` 被扩展成了 800 多行。再比如说，

- `#define SQR(x) (x * x)`，这是一句「宏定义」，意思是说，在后面的程序里，凡是遇到 `SQR(x)`，就都给替换成 `(x * x)`. 这个替换工作，也是由 cpp 完成的。

C 编译器 我们用的是 gcc。编译器 (Compiler) 的工作是把 cpp 处理过的源程序翻译成一个汇编程序。如果你是头一次听说汇编语言，那么你应该立即去 Google 一下「汇编语言」。简而言之，汇编语言是一种比 C 更底层，也就是更靠近 CPU，的编程语言。如果你想搞操作系统开发，或者硬件驱动开发，那么汇编就是你必须掌握的编程语言。

与其把 C 翻译成汇编，为什么我们不直接用汇编来写程序呢？两个原因，

1. 用汇编写程序比用 C 要累得多。和汇编相比，C 是一门高级(或者说「高层」) 语言。所谓「高级」，通过上面的大流程图来看，就是离上面的那个程序员更近。其实也就是对人更友好的意思。举个简单的例子，

   ```
   i++
   ```

   如果用汇编来写，大概就成了下面这样

   ```
   movl    $0, -4(%rbp)
   addl    $1, -4(%rbp)
   movl    -4(%rbp), %eax
   ```

   显然，还是 `i++` 更友好些吧。通常来讲，用高级语言(比如 C)写一条程序语句就相当于好几条，甚至几十条，汇编语句。而一条汇编语句通常就对应一条机器指令。所以，用汇编些程序要累得多。

   通过下面的命令，就可以把我们的 `hello.c` 翻译成一个汇编程序 `hello.s`:

   ```
   gcc -S hello.c
   ```

   生成的 `hello.s` 就是下面这副样子：

   ```
   1 ^^I.file   "hello.c"
   2 ^^I.section        .rodata
   3 .LC0:
   4 ^^I.string "Hello, world!"
   5 ^^I.text
   ```

[1]在我们的 Debian GNU/Linux 系统中，它的完整路径是 `/usr/include/stdio.h`。

```
^^I.globl  main
^^I.type   main, @function
main:
.LFB0:
^^I.cfi_startproc
^^Ipushq   %rbp
^^I.cfi_def_cfa_offset 16
^^I.cfi_offset 6, -16
^^Imovq    %rsp, %rbp
^^I.cfi_def_cfa_register 6
^^Imovl    $.LC0, %edi
^^Icall    puts
^^Imovl    $0, %eax
^^Ipopq    %rbp
^^I.cfi_def_cfa 7, 8
^^Iret
^^I.cfi_endproc
.LFE0:
^^I.size   main, .-main
^^I.ident  "GCC: (Debian 5.4.0-6) 5.4.0 20160609"
^^I.section        .note.GNU-stack,"",@progbits
```

和前面的 `hello.c` 相比，我想你肯定和我一样，更愿意用 C 来写程序吧。

2. 用汇编语言写出的程序针对性很强，通常不能跨平台使用，可移植性差。针对 x86 机器写的汇编程序，在其他机器(比如 ARM, PowerPC, M68000, ...) 上就不能用。因为一条汇编指令就对应一条机器指令，而各 CPU 架构所支持的机器指令集都不一样，那么对应的汇编程序当然就无法跨平台使用了。反之，用高级语言写的程序就比较容易跨平台，可移植性比较好。

   所以，基于上述原因，通常我们会尽可能地选用高级语言来编程。之后，再利用汇编器，把对人友好的高级程序语句翻译成对机器友好的底层程序语句。

汇编器 汇编器 (Assembler) 是用来把人能看懂的汇编程序翻译成人读不懂，但机器能读懂的二进制程序，通常我们把它叫做目标文件(object file)，通常以 `.o` 结尾。我们用的汇编器是 GNU Assembler (gas)。

通常如果想要生成一个 `.o` 文件的话，我们用下面的命令：

```
gcc -c hello.c
```

这样，gcc 会调用 gas 帮我们生成一个目标文件。这时，你的目录里应该多了一个 `hello.o` 文件了。好奇的话，你可以 `cat hello.o` 来看看它的内容。我肯定你读不懂它，除非你是 CPU。用 `hd hello.o` 来看看它的内容，感觉会

稍好些，虽然还是看不懂。

链接器 链接器 (Linker) 是用来把若干个 `.o` 文件结合成一个可执行文件。我们用的链接器是 ld。

也许你会问,「我只写了一个 `hello.c`, 经过编译和汇编之后，只产生了一个而不是多个 `.o` 文件，还需要链接吗」? 的确，你只有一个 `.o` 文件，但在你的 `hello.c` 文件里还用到了别人的 `.o` 文件，比如说，`printf()` 函数并不是你写的吧？它存在于系统自带的函数库里。系统函数库里装的其实就是一大堆 `.o` 文件。这些 `.o` 文件里都是供我们调用的一个个函数，其中就包括 `printf()` 函数。所以，你自己的 `hello.o` 必须要和系统函数库中包含 `printf()` 的那个 `.o` 文件[2]链接，然后才能得到最终的可执行文件。

调试器 调试器 (Debugger) 是用来帮助我们找出可执行文件中的 bug。我们用的调试器是 gdb，它可以

- 分步执行程序
- 设置断点
- 追踪变量的值
- 查看堆栈
- 还有很多高深的功能

如果你的程序像 `Hello, world!` 那样简单，那么通常也就不会有什么 bug，自然也就不需要调试器了。

上面我们简单介绍了一下从 `hello.c` 到 `hello.exe` 的过程。了解这些基础知识，有助于我们加深对编程的理解，可以让我们在今后的学习中少走弯路。

2.2 常用编译选项

在上一节，我们看到在使用 gcc 编译 C 程序的时候，可以跟上一些选项，比如 `-o`, 后面可以给出可执行文件的名字。下面我们再介绍几个常用的编译选项。

2.2.1 `-Wall`

这个选项非常有用，应该随时都带着。大写的 `W` 代表 warning, `all` 就代表 all，那么 `-Warning` 就代表打开所有的告警。也就是说，编译过程中发现的算不上是「错误」的小毛病也都会被提示出来。有的时候，这些小毛病还是挺要命的，比如下面这个小程序 `wall.c`

```
1 #include <stdio.h>
2
3 int main (void)
4 {
```

[2]实际上是 `libc.so` 文件。`so` 代表 shared object，是 Unix 平台通用的动态链接函数库。想了解更多？去 Google 一下"shared object" 就知道了。

```
5   printf ("Two plus two is %f\n", 4);
6   return 0;
7 }
```

如果不带 -Wall 直接编译的话，gcc wall.c, 看不到任何错误提示。可是运行 a.out 的输出结果却是：

```
Two plus two is 0.000000
```

这个结果明显是错误的。如果编译时带上 -Wall 选项，gcc -Wall wall.c, 会看到如下输出：

```
wall.c: In function ‘main’:
wall.c:5:11: warning: format ‘%f’ expects argument of type ‘double’,
       but argument 2 has type ‘int’ [-Wformat=]
printf ("Two plus two is %f\n", 4);
        ^
```

问题被提示了出来，用 %f 的格式来输出整型数是不合情理的。记住，编译时永远带上 -Wall, 而且 W 必须大写！

2.2.2 -E

-E 这个选项是告诉 gcc, 调用完 cpp 就停下来。也就是说 gcc -E hello.c 和 cpp hello.c 是一回事。关于 cpp 我们前面已经提到过，它是一个 c preprocessor, 作用之一就是把源程序中的宏定义(Macro)扩展还原成它本来的字符串。宏定义是 C 编程中经常要用到的强大武器，而且编程大师们可以把它用得非常复杂。比如在 Linux 的内核源代码里就有下面这样的宏定义：

```
1 #define INIT_LIST_HEAD(ptr) do { \
2 ^^I   (ptr)->next = (ptr);(ptr)->prev= (ptr);  \
3 ^^I} while(0)
```

一个完整的 do-while 结构被 INIT_LIST_HEAD(prt) 代表了。你知道 do {} while(0) 中的花括号里是可以放很多程序语句的，那么你肯定也想到了，宏定义可以用来代表非常非常复杂的东西。

常识告诉我们，越复杂的东西越容易隐藏着 bug。-E 这个选项可以帮助我们排除宏定义中的 bug。比如下面这个小程序 macro.c 里用到了一个很简单的宏定义 SQR(x)

```
1 #include <stdio.h>
2 #define SQR(x) (x * x)
3 int main()
4 {
5     int counter;    /* counter for loop */
6     for (counter = 0; counter < 5; ++counter)
7     {
8 ^^Iprintf("x %d, x squared %d\n",
9 ^^I    counter+1, SQR(counter+1));
```

```
10      }
11      return (0);
12  }
```

SQR(x) 显然是要对 x 做平方运算。编译一下，gcc -Wall macro.c, 很顺利，没有任何出错迹象。现在运行一下 a.out, 看看结果：

```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

显然是错误的！问题就出在 SQR(x) 。借助 -E 把它还原扩展开，程序变成了这样：

```
1  /* 前面省略无数行 */
2  int main()
3  {
4      int counter;
5      for (counter = 0; counter < 5; ++counter)
6      {
7  ^^Iprintf("x %d, x squared %d\n",
8  ^^I    counter+1, (counter+1 * counter+1));
9      }
10     return (0);
11 }
```

看明白了吗？在第 8 行，SQR(counter+1) 被扩展成了 (counter+1 * counter+1), 而我们真正想要的是 ((counter+1) * (counter+1)), 所以程序中的宏定义不该是

```
1  #define SQR(x) (x * x)
```

而应该是

```
1  #define SQR(x) ((x) * (x))
```

如果你也喜欢宏定义，那么一定要记住 -E 这个编译选项。

2.2.3 -D

Debug 的时候，我们经常会在程序里加入一些 printf() 语句，借助它输出某些关键变量的值，帮助我们思考。在 bug 被解决之后，这些 printf() 语句也就没用了，如果要一个个地删除掉，实在是一件麻烦的事情。想省点事的话，你可以借助一下 -D 这个编译选项。比如，下面这个小程序 stackoverflow.c,

```
1  #include<stdio.h>
2
```

```
3 int i=0;
4
5 int main(void)
6 {
7 #ifdef DEBUG
8   printf("%d\t",i++);
9 #endif
10  main();
11  return 0;
12 }
```

主函数递归地调用它自己，这实在不是件有意义的工作。但如果你想知道调用多少次之后栈才会溢出，那么可以像上面程序中的第 8 行那样，利用 `printf()` 来输出计数器变量 `i` 的值。

第 7、9 两行是干什么用的呢？如果你正常编译这个小程序

```
gcc -Wall stackoverflow.c
```

然后运行 `./a.out`, 你只能看到如下一行输出，那就是著名的

```
Segmentation fault
```

很显然，栈溢出，递归程序就结束了，而且 `printf()` 没有发挥作用。但是，如果你像下面这样编译:

```
gcc -Wall -DDEBUG stackoverflow.c
```

之后再运行 `./a.out`, 怎么样？在我的电脑上，Segmentation fault 之前，=i= 最后的值是 523629。很显然，带上编译选项 `-DDEBUG` 之后，`printf()` 起作用了。现在，你该猜到 `-D` 和程序中的

```
#ifdef DEBUG
...
#endif
```

之间的关系了吧？`DEBUG` 也是个 Macro，`-DDEBUG` 就相当于在程序里写上

```
#define DEBUG 1
```

程序中的 `#ifdef DEBUG` 就是说「如果 `DEBUG` 有定义的话」，那么就执行之后的程序语句，直到看见 `#endif` 为止。

所以，编译时如果不带 `-D`, 那么 `DEBUG` 就没定义，于是 `#ifdef DEBUG` 这句判断结果就是 `false`, 于是它后面的 `printf()` 就不会被执行。反之，编译时带上 `-DDEBUG`, 那么 `DEBUG` 就有了定义，于是 `#ifdef DEBUG` 判断就返回 `true`, 于是 `printf()` 就发挥作用了。

如此一来，你再也不用为删除多余的 `printf()` 操心了，只需要操控 `-D` 这个小开关就可以达到目的了。

2.2.4 更多选项

gcc 的编译选项多如牛毛，但做为初学者，知道上面这些就算是入门了。随着学习的深入，更多的选项也会逐渐变成我们的常用选项。比如，

`-g` 如果你想用 gdb 来 debug 程序的话，编译时一定要带上它。

`-l` 如果你用到了外部函数库里的函数，那么编译时就要带上它，`l` 代表 link，链接的意思。

「那么，我怎么知道我到底要链接哪个函数库呢」? 答案很简单,「看手册」。比如说，我在程序里调用了 `pthread_create()` 函数用来产生一个新的线程，那么，显然你该看看 `pthread_create()` 的手册，具体了解一下这个函数的应用细节。

```
man pthread_create
```

手册的前几行如下:

```
NAME
   pthread_create - create a new thread

SYNOPSIS
   #include <pthread.h>

   int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg);

   Compile and link with -pthread.
```

它告诉你

1. 在程序中一定要有 `#include <pthread.h>`;
2. 从函数原型你知道
   (a) `pthread_create()` 一定要返回一个 `int`;
   (b) 调用这个函数必须提供 4 个指针类型的参数。
3. 上面的最后一行 `Compile and link with -pthread`, 明确告诉你编译的时候要带上 `pthread` 选项。

养成看手册的习惯，这可是程序员的基本功。

关于 Linux 平台上的 C 开发环境，我们简单介绍了编辑器 Emacs，和编译器 gcc。掌握了这两样神器，你就是个相当有前途的程序员了。另外还剩下一个调试神器 gdb 我们没有介绍。当年我的老师这样对我说,「像你这样两三百行的小程序，最好不要用调试器，静下心来，一行一行地读你自己的代码，用你自己的大脑来找出程序中的 bug，这是对你最好的训练」。Happy hacking!