gcc 编译器

一、 gcc 简介

目前 Linux 下最常用的 C 语言编译器是 gcc（GNU Compiler Collection），它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++和 Objective C 等语言编写的程序。gcc 不仅功能十分强大，结构也异常灵活。最值得称道的一点就是它可以通过不同的前端模块来支持各种语言，如 Java、Fortran、Pascal、Modula-3 和 Ada 等。gcc 是可以在多种硬件平台上编译出可执行程序的超级编译器，其编译效率与一般的编译器相比，平均效率要高 20%～30%。gcc 支持编译的一些源文件的后缀及其解释见表 1。

表 1

后缀名	所支持的语言
.c	C 原始程序
.C	C++原始程序
.cc	C++原始程序
.cxx	C++原始程序
.m	Objective-C 原始程序
.i	已经经过预处理的 C 原始程序
.ii	已经经过预处理的 C++原始程序
.s	组合语言原始程序
.S	组合语言原始程序
.h	预处理文件（标头文件）
.o	目标文件
.a	存档文件

在使用 gcc 编译程序时，编译过程可以被细分为 4 个阶段：

- 预处理（Pre-Processing） 处理以“#”为开头的命令
- 编译（Compiling） 将.c .i 等文件翻译成汇编代码
- 汇编（Assembling） 将汇编代码翻译成机器代码
- 链接（Linking）将生成的多个目标文件（.o 文件）连接起来，生成可执行文件

gcc 提供了 30 多条警告信息和 3 个警告级别，使用它们有助于增强程序的稳定性和可移植性。此外，gcc 还对标准的 C 和 C++语言进行了大量的扩展，提高了程序的执行效率，有助于编译器进行代码优化，能够减轻编程的工作量。

二、 使用 gcc

gcc 的版本可以用如下 gcc -v 命令查看：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.8/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 4.8.4-2ubuntu1~14.04.3' --with-bu
gurl=file:///usr/share/doc/gcc-4.8/README.Bugs --enable-languages=c,c++,java,go,d,fortran,objc,o
bj-c++ --prefix=/usr --program-suffix=-4.8 --enable-shared --enable-linker-build-id --libexecdir
=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/
c++/4.8 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-
debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --disable-libmudflap --enable-plugi
n --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-ja
va-home=/usr/lib/jvm/java-1.5.0-gcj-4.8-amd64/jre --enable-java-home --with-jvm-root-dir=/usr/li
b/jvm/java-1.5.0-gcj-4.8-amd64 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-4.8-amd64
--with-arch-directory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --en
able-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,
mx32 --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-
gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3)
```

实例 main.c

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

int main(int argc,char *argv[]) {
    printf("This is a test file\n");
    return 0;
}
```

第一步：进行预编译，使用-E 参数可以让 gcc 在预处理结束后停止编译过程；

gcc -E -o main.i main.c //对 C 文件做预处理

输出结果：

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 374 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 385 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 386 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 375 "/usr/include/features.h" 2 3 4
# 398 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 399 "/usr/include/features.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4









# 1 "/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stddef.h" 1 3 4
# 212 "/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
# 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4


typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
```

第二步：将 main.i 编译为汇编文件，这个可以通过使用参数-c 来完成；

gcc -S -o main.s main.i //进行编译，生成汇编文件

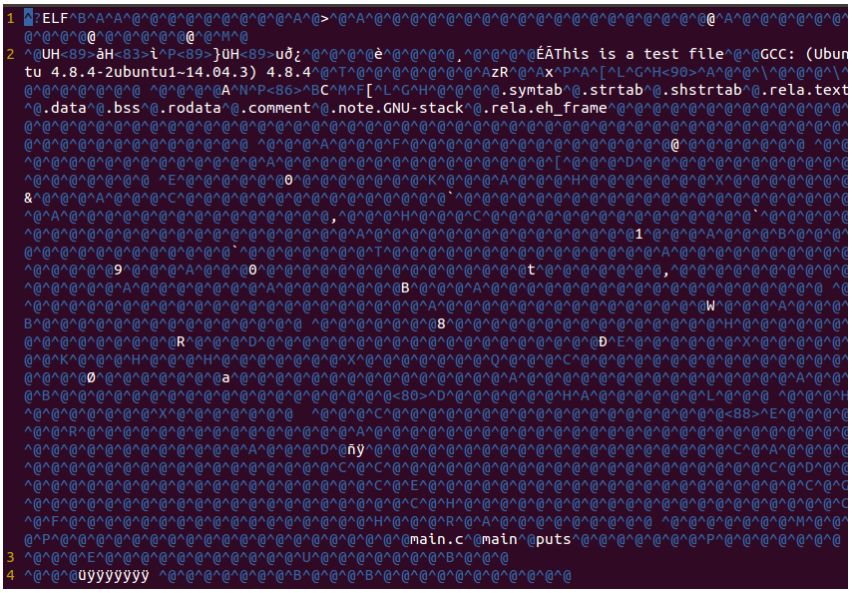
输出结果：

```
    .file   "main.c"
    .section    .rodata
.LC0:
    .string "This is a test file"
    .text
    .globl  main
    .type   main, @function
main:
.LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
    .section    .note.GNU-stack,"",@progbits
```

第三步：将汇编代码 main.s 编译成目标文件

gcc -c -o main.o main.s //对汇编代码编译成目标文件

输出结果：



第四步：将生成的目标文件链接成可执行文件

```
gcc -o main main.o        //生成可执行程序 main
```

输出结果：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# ./main
This is a test file
```

在采用模块化的设计思想进行软件开发时，通常整个程序是由多个源文件组成的，相应地就形成了多个编译单元，使用 gcc 能够很好地管理这些编译单元，使用 gcc 能够很好地管理这些编译单元。假设有一个由 david.c 和 xueer.c 两个源文件组成的程序，为了对它们进行编译，并最终生成可执行文件 davidxueer，可以使用下面的命令：

```
gcc david.c xueer.c  -o davidxueer
```

如果同时处理的文件不止一个，gcc 仍然会按照预处理、编译、汇编和链接的过程依次进行，如果深究起来，上面这条命令大致相当于依次执行如下 3 条命令：

```
gcc david.c  -o david.o
gcc sueer.c  -o sueer.o
gcc david.o sueer.o  -o davidsueer
```

三、 gcc 警告提示功能

gcc 包含完整的出错检查和警告提示功能，它们可以帮助 Linux 程序员尽快找到错误代码，从而写出更加专业和优美的代码。

实例 bad.c

```
1 #include <stdio.h>
2 void main(int argc,char *argv[]) {
3     long long int var = 1;
4     printf("It is not standard C code!\n");
5 }
```

当 gcc 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果加上了-pedantic 选项，那么扩展语法的地方将产生相应的警告信息：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# gcc -pedantic bad.c -o bad
bad.c:2:6: warning: return type of ‘main’ is not ‘int’ [-Wmain]
 void main(int argc,char *argv[]) {
      ^
bad.c: In function ‘main’:
bad.c:3:10: warning: ISO C90 does not support ‘long long’ [-Wlong-long]
     long long int var = 1;
          ^
```

需要注意的是，-pedantic 编译选项并不能保证被编译程序与 ANSI/ISO C 标准的完全兼容，它仅仅用来帮助 Linux 程序员离这个目标越来越近。换句话说，-pedantic 选项能够帮助程序员发现一些不符合 ANSI/ISO C 标准的代码，但不是全部。事实上只要 ANSI/ISO C 语言标准中要求进行编译器诊断的那些问题才有可能被 gcc 发现并提出警告。

除了-pedantic 之外，gcc 还有一些其他编译选项也能够产生有用的警告信息。这些选项大多数是以-W 开头，其中最有价值的当属-Wall，使用它能够使 gcc 产生尽可能多的警告信息。例如：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# gcc -Wall bad.c -o bad
bad.c:2:6: warning: return type of ‘main’ is not ‘int’ [-Wmain]
 void main(int argc,char *argv[]) {
      ^
bad.c: In function ‘main’:
bad.c:3:19: warning: unused variable ‘var’ [-Wunused-variable]
     long long int var = 1;
                   ^
```

gcc 给出的警告信息虽然从严格意义上来说不能算作是错误，但是很可能成为错误地栖身之所。一个优秀的 Linux 程序员应该尽量避免产生警告信息，使自己的代码始终保持简洁、优美和健壮的特性。

在处理警告方面，另一个常用的编译选项是-Werror，它要求 gcc 将所有的警告当成错误进行处理，这在使用自动编译工具（如 make 等）时非常有用。如果编译时带上-Werror 选项，那么 gcc 会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改。只有当相应的警告信息消除时，才可能将编译过程继续朝前进行。

对 Linux 程序员来讲，gcc 给出的警告信息都是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在 gcc 编译源代码时始终带上-Wall 选项，并把它逐渐培养成为一种习惯，这对找出常见的隐式编程错误很有帮助。

四、 库依赖

在 Linux 下使用 C 语言开发应用程序时，完全不使用第三方函数库的情况是比较少见的，通常来讲需要借助一个或者多个函数库的支持才能完成相应的功能。从程序员的角度看，函数库实际上就是一些头文件（.h）和库文件（.so 或者.a）的集合。虽然 Linux 下大多数函数都默认将头文件放在/usr/include/目录下，而库文件则放到/usr/lib/目录下，但并不是所有的情况都是这样。正因为如此，gcc 在编译的时候必须让编译器知道如何来查询所需要的头文件和库文件。

gcc 采用搜索目录的办法来查找所需要的文件，-I 选项可以向 gcc 的头文件搜索路径中添加新的目录。例如，如果在/home/david/include/目录下有编译时所需要的头文件，为了让 gcc 能够顺利地找到它们，就可以使用-I 选项：

```
gcc david.c -I /home/david/include -o david
```

同样，如果使用了不在标准位置的库文件，那么可以通过-L 选项向 gcc 的库文件搜索路径中添加新的目录。例如，如果在/home/david/lib/目录下有链接时所需要的库文件 libdavid.so，为了让 gcc 能够顺利地找到它，可以使用下面的命令：

```
gcc david.c -L /home/david/lib -ldavid -o david
```

值得详细解释一下的是-l 选项，它指示 gcc 去连接库文件 david.so。Linux 下的库文件在命名时有一个约定，那就是应该以 lib 三个字母开头。由于所有的库文件都遵循了同样的规范，因此在用-l 选项指定链接的库文件名时可以省去 lib 三个字母。也就是说 gcc 在对-ldavid 进行处理时，会自动去链接名为 libdavid.so 的文件。再举个例子：

```
gcc -o hello hello.c -I/home/hello/include -L/home/hello/lib -lworld
```

上面这句表示在编译 hello.c 时：

- -I /home/hello/include，表示将/home/hello/include 目录作为第一个寻找头文件的目录，寻找的顺序是：/home/hello/include->/usr/include->/usr/local/include
  - 也就是指定优先查找的目录，找不到的话查找默认目录

- -L /home/hello/lib，表示将/home/hello/lib 目录作为第一个寻找库文件的目录，寻找的顺序是：/home/hello/lib->/lib->/usr/lib->/usr/local/lib
  - 同上，也是指定优先查找的目录
- -lword,表示寻找动态链接库文件 libword.so（也就是文件名去掉前缀和后缀所代表的库文件）
  - 如果加上编译选项-static，表示寻找静态链接库文件，也就是 libword.a

对于第三方提供的动态链接库(.so)，一般将其拷贝到一个 lib 目录下(/usr/local/lib)，或者使用-L 来指定其所在目录， 然后使用-l 来指定其名称。

Linux 下的库文件分为两大类，分别是动态链接库（通常以.so 结尾）和静态链接库（通常以.a 结尾），两者的差别仅在于程序执行时所需要的代码是在运行时动态加载的还是在编译时静态加载的。默认情况下，gcc 在链接时优先使用动态链接库，只有动态链接库不存在时才考虑使用静态链接库。如果需要的话可以在编译时加上-static 选项，强制使用静态链接库。例如，如果/home/david/lib/目录下有链接时所需要的库文件 libfoo.so 和 libfoo.a，为了让 gcc 在连接时只用到静态链接库，可以使用下面的命令：

```
gcc foo.c -L /home/david/lib -static -ldavid -o david
```

五、 gcc 代码优化

代码优化是指编译器通过分析源代码，找出其中尚未达到最优的部分，然后对其进行重新组合，目的是为了改善程序的执行性能。gcc 提供的代码优化功能非常强大，它通过编译选项-On 来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 gcc 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 变化到 2 或 3。

编译时使用选项-O 可以告诉 gcc 同时减小代码的长度和执行时间，其效果等效于-O1。在这一级别上能够进行的优化类型虽然取决于目标处理器，但是一般都会包括线程跳转（Thread Jump）和延迟退栈（Deferred Stack Pops）两种优化。

选项-O2 告诉 gcc 除了完成所有-O1 级别的优化之外，同时要进行一些额外的调整工作，如处理器指令调度等。

选项-O3 则除了完成所有-O2 级别的优化之外，还包括循环展开和其他一些与处理器特性相关的优化工作。

通常来说，数字越大优化的等级越高，同时也意味着程序的运行速度越快。许多 Linux 程序员都喜欢使用-O2 选项，因为它在优化长度、编译时间和代码大小之间取得了一个比较理想的平衡点。

实例：count.c

```
#include <stdio.h>
int main(void) {
    double counter;
    double result;
    double temp;
    for (counter = 0;counter < 4000.0*4000.0*4000.0/20.0+2030;counter+=(5-3+2+1)/4){
        temp = counter / 1239;
        result = counter;
    }

    printf("Result is %lf\n",result);
    return 0;
}
```

首先不加任何优化选项进行编译：

```
gcc -Wall count.c -o count
```

借助 Linux 提供的 time 命令，可以大致统计出该程序在运行时所需要的时间：

```
time ./count
```

结果：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# time ./count
Result is 3200002029.000000

real    0m20.388s
user    0m19.164s
sys     0m0.000s
```

接下来使用优化选项来对代码进行优化：

```
gcc -Wall -O2 count.c -o count2
```

结果：

```
root@ubuntu:~/Documents/linux-c-learn/gcc-learn# time ./count2
Result is 3200002029.000000

real    0m6.117s
user    0m5.316s
sys     0m0.004s
```

对比两次执行的输出结果不难看出，程序的性能的确得到了很大幅度的改善，由原来的 20.388s 缩短到 6.117s，这个例子是专门针对 gcc 的优化功能而设计

的，因此优化前后程序的执行速度发生了很大的改变。尽管 gcc 的代码优化功能非常大，但作为一名优秀的 Linux 程序员，首先还是要力求能够手工编写高质量的代码。如果编写的代码简洁，并且逻辑性强，编译器就不会做更多的工作，甚至用不着优化。

优化虽然能够给程序带来更好的执行性能，但在如下一些场合中应该尽量避免优化代码。

- 程序开发的时候：优化等级越高，消耗在编译上的时间就越长，因此在开发的时候最好不要使用优化选项，只有到软件发行或开发结束的时候，才考虑对最终生成的代码进行优化。
- 资源受限的时候：一些优化选项会增加可执行代码的体积，如果程序在运行时能够申请到的内存资源非常紧张（如一些实时嵌入式设备），那就不要对代码进行优化，因为由此带来的负面影响可能会产生非常严重的后果。
- 跟踪调试的时候：在对代码进行优化的时候，某些代码可能会被删除或改写，或者为了取得更加的性能而进行重组，从而使跟踪和调试变得异常困难。

六、 加速

在将源代码变成可执行文件的过程中，需要经过许多中间的步骤，包含预处理、编译、汇编和链接。这些过程实际上是由不同的程序负责完成的。大多数情况下 gcc 可以为 Linux 程序员完成所有的后台工作，自动调入相应程序进行处理。

这样做有一个明显的缺点，就是 gcc 在处理每一个源文件的时候，最终都需要生成好几个零时文件才能完成相应的工作，从而无形中导致处理速度变慢。例如，gcc 在处理一个源文件时，可能需要一个临时文件来保存预处理的输出，一个临时文件来保存编译器的输出，一个临时来保存汇编器的输出，而读写这些临时文件显然需要耗费一定的时间。当软件项目变得非常庞大的时候，花费在这上面的代价可能会变得很大。

解决的办法是，使用 Linux 提供的一种更加高效的通信方式---管道。它可以用来同时连接两个程序，其中一个程序的输出将直接作为另一个程序的输入，这样就可以避免使用临时文件，但编译时却需要消耗更多的内存。

注意：在编译的过程中使用管道是有 gcc 的-pipe 选项决定的。下面的这条命令就是借助 gcc 的管道功能来提高编译速度的：

```
gcc  -pipe david.c  -o david
```

在编译小型工程时使用管道，编译时间上的差异可能还不是很明显，但在源代码非常多的大型工程中，差异将变得非常明显。

七、 gcc 常用选项

gcc 作为 Linux 下 C/C++重要的编译环境，功能强大，编译选项繁多。为了方便大家日后编译方便，在此将常用的选项及说明罗列出来，见表 2。

表 2

选项名	作用
-c	只编译不链接。编译器只是将输入的.c 等源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件
-S	只对文件进行编译，不汇编和链接
-E	只对文件进行预处理，不编译汇编和链接
-o output_filename	确定输出文件的名称为 output_filename，这个名称不能和源文件同名。如果不给出这个选项， gcc 就给出预设的可执行文件 a.out
-g	产生符号调试工具(GNU 的 gdb)所必要的符号信息，要想对源代码进行调试，就必须加入这个选项。g 也分等级，默认是-g2，-g1 是最基本的，-g3 包含宏信息。它不能与-o 选项联合使用
-DFOO=BAR	在命令行定义预处理宏 FOO，值为 BAR
-O	对程序进行优化编译、链接。采用这个选项，整个源代码会在编译、链接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、链接的速度就相应地要慢一些
-ON	指定代码的优化等级为 N，可取值为 0、 1、 2、 3； 00 没有优化， 03 优化级别最高
-Os	使用了-O2 的优化部分选项，同时对代码尺寸进行优化
-Idirname	将 dirname 目录加入到程序头文件搜索目录列表中，是在预编译过程中使用的参数
-L dirname	将 dirname 目录加入到库文件的搜索目录列表中
-l FOO	链接名为 libFOO 的函数库
-static	链接静态库
-ansi	支持 ANSI/ISO C 的标准语法，取消 GNU 语法中与该标准相冲突的部分
-w	关闭所有警告，不建议使用
-W	开启所有 gcc 能提供的警告
-werror	将所有警告转换为错误，开启该选项，遇到警告都会中止编译
-v	显示 gcc 执行时执行的详细过程， 以及 gcc 和相关程序的版本号

八、 gcc 的错误类型及对策

gcc 给出的错误信息一般可以分为四类，下面我们分别讨论其产生的原因和对策。

1. 第一类：C 语法错误

推荐一本由 Andrew Koenig 写的《C 陷阱与缺陷》

2. 第二类：头文件错误

错误信息：找不到头文件 head.h（can not find include file head.h）。这类错误时代么文件中包含的头文件有问题，可能的原因有头文件名错误、指定的头文件所在目录名错误等，也可能是错误地使用了双引号和尖括号。

3. 第三类：档案库错误

错误信息：连接程序找不到所需的函数库，例如：

ld: -lm:No such file or directory

这类错误是与目标文件相连接的函数库有错误，可能的原因是函数库名错误、指定的函数库所在的目录名称错误等。检查的方法是使用 find 命令在可能的目录中寻找相应的函数库名，确定档案库及目录的名称并修改程序中及编译选项中的名称。

4. 第四类：未定义符号

错误信息：有未定义的符号（Undefined symbol）。这类错误是在连续过程中出现的，可能有两种原因：一是用户自己定义的函数或者全局变量所在源代码文件，没有被编译、链接，或者干脆没有定义，这需要用户根据实际情况修改源程序，给出全局变量或者函数的定义体；二是未定义的符号是一个标准的库函数，在源程序中使用了该库函数，而连接过程中还没有给定相应的函数库的名称，或者是该档案库的目录名称有问题，这时需要使用档案库维护命令 ar 检查我们需要的库函数到底位于哪一个函数库中，确定之后，修改 gcc 连接选项中的-l 和-L 项。