第 5 章　使用 GCC 编译器

编译器就是将高级程序语言转换为执行效率更高的机器语言的程序。GCC 是 Linux 平台下最常用的编译程序，同时，在 Linux 平台下的嵌入式开发领域，GCC 也是使用最为普遍的一种编译器。本章将对 GCC 的安装及基本使用方法进行讲解。

5.1　安装 GCC

GCC（GNU Compiler Collection）是一套功能强大、性能优越的编程语言编译器，它是 GNU 计划的代表作品之一。目前，GCC 支持的体系结构有 40 多种，常见的有 x86 系列、ARM 系列、PowerPC 等。另外，GCC 还能运行在不同的操作系统上，如 Linux、Solaris、Windows 等。

Linux 下安装 GCC 有两种方式，以 RPM 等二进制形式在线安装或直接从源代码编译安装。编译安装对应初学者来说难度有点大并且容易出错。这里我们就介绍在线安装的方式。

Ubuntu 等基于 Debian 发行版 Linux 可以使用如下命令安装：

```
apt-get install gcc
```

Fedora 等基于 RPM 发行版 Linux 可以使用如下命令安装：

```
yum install gcc
```

在上述命令执行完毕后，读者可以通过编译一段 C 程序来检测 GCC 是否正常安装。也可以通过查看 GCC 版本来确认，命令如下：

```
gcc --version
```

5.2　GCC 常用选项

GCC 编译器可以使用不同的选项来编译程序，使得用户可以根据自己的需求来定制编译程序。下面我们就来学习常用的选项。

5.2.1　编译程序

本节通过使用不同的选项，编译一段代码来学习 GCC 常用的选项。GCC 编译器的基本选项如表 5.1 所示。

表 5.1　GCC 编译器的基本选项

选　　项	说　　明
-E	预处理后即停止，不进行编译、汇编及连接
-S	编译后即停止，不进行汇编及连接
-c	编译或汇编源文件，但不进行连接
-o file	指定输出文件为 file

在第 2 章中已经介绍过，程序的编译要经过预处理、编译、汇编，以及连接 4 个阶段。在预处理阶段，主要处理 C 语言源文件中的#ifdef、#include，以及#define 等命令。在预处理的过程中，GCC 会忽略掉不需要预处理的输入文件，该阶段会生成中间文件*.i。

对于如下的源程序：

```
01  /* example.c */
02  #include <stdio.h>
03  int main()
04  {
05      int i;
06      for(i=0;i<5;i++)
07      {
08          printf("%d\n",i);
09      }
10      return 0;
11  }
```

使用如下命令对上面的源文件进行预处理。

```
gcc -E example.c -o example.i
```

上面使用了两个选项：-E 和-o，其中-E 表示在预处理结束后即停止编译过程；-o 指定输出文件。前面的选项不同，输出的文件类型也不相同，可能为预处理后的 C 代码、汇编文件、目标文件或可执行文件，这里即为预处理后的 C 代码。

预处理后输出文件 example.i 的内容为：

```
01  # 1 "example.c"
02  # 1 "<built-in>"
03  # 1 "<命令行>"
04  # 1 "example.c"
05  # 1 "/usr/include/stdio.h" 1 3 4
06  # 28 "/usr/include/stdio.h" 3 4
07  … (中间部分省略)
08  extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
    __leaf__));
09  # 940 "/usr/include/stdio.h" 3 4
10
11  # 2 "example.c" 2
12  int main()
13  {
14   int i;
15   for(i=0;i<5;i++)
16   {
17    printf("%d\n",i);
18   }
19   return 0;
```

```
20  }
```

从上面的代码可以看出，GCC 对源文件所包含的头文件 stdio.h 进行了预处理。由于输出文件 example.i 比较长，上面只列出了部分内容。头文件 stdio.h 位于/usr/include 目录下，读者有兴趣的话可以查看该文件内容，并同 example.i 进行比对。

在编译阶段，输入的是中间文件*.i，编译后生成的是汇编语言文件*.s。这个阶段对应的 GCC 命令如下：

```
gcc -S example.i -o example.s
```

example.s 即为生成的汇编语言文件，其内容为：

```
01      .file   "example.c"
02      .section    .rodata
03  .LC0:
04      .string "%d\n"
05      .text
06      .globl  main
07      .type   main, @function
08  main:
09  .LFB0:
10      .cfi_startproc
11      pushl   %ebp
12      .cfi def cfa offset 8
13      .cfi_offset 5, -8
14      movl    %esp, %ebp
15      .cfi def cfa register 5
16      andl    $-16, %esp
17      subl    $32, %esp
18      movl    $0, 28(%esp)
19      jmp .L2
20  .L3:
21      movl    $.LC0, %eax
22      movl    28(%esp), %edx
23      movl    %edx, 4(%esp)
24      movl    %eax, (%esp)
25      call    printf
26      addl    $1, 28(%esp)
27  .L2:
28      cmpl    $4, 28(%esp)
29      jle .L3
30      movl    $0, %eax
31      leave
32      .cfi restore 5
33      .cfi def cfa 4, 4
34      ret
35      .cfi endproc
36  .LFE0:
37      .size   main, .-main
38      .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
39      .section    .note.GNU-stack,"",@progbits
```

有汇编语言基础的读者应该能看懂上面的语句，它实现了一个 for 循环，这里就不多做介绍。

在上面的例子中，是在预处理后 C 代码的基础上进行编译的，其实可以直接从源代码编译，这时应该使用小写的 s，即：

```
gcc -s example.c -o example.s
```

汇编是将输入的汇编语言文件转换为目标代码，这可以通过使用-c 选项来完成，对应的 GCC 命令如下：

```
gcc -c example.s -o example.o
```

注意：此时目标文件虽然是机器代码，但不可执行。

最后，将生成的目标文件与其他目标文件（或库文件）连接成可执行的二进制代码文件。这一步骤可以使用如下 GCC 命令来完成：

```
gcc example.o -o example
```

运行 example，输出结果为：

```
./example
0
1
2
3
4
```

在上面的例子中，从预处理、编译、汇编，以及连接一步步走下来，主要目的是讲解程序编译的整个过程以及 GCC 的各个选项，如果只需要最终的可执行文件，可以直接对源文件进行编译连接，对应的 GCC 命令如下：

```
gcc example.c -o example
```

对于一个程序的多个源文件进行编译连接时，可以使用如下格式：

```
gcc -o test first.c second.c third.c
```

该命令将同时编译 3 个源文件，即 first.c、second.c 和 third.c，然后将它们连接成一个可执行程序，名为 test。

注意：生成可执行文件时，被编译和连接的多个源文件中必须有且只能有一个 main()函数，因为 main()函数是该程序的入口。如果仅仅是把源文件编译成目标文件，而不进行连接，在这种情况下，main()函数不是必需的。

在上面的例子中，我们都给出了-o 选项，如果没有给出该选项，默认的输出结果为：预处理后的 C 代码被送往标准输出，即输出到屏幕，汇编文件为 example.s，目标文件为 example.o，而可执行文件为 a.out。

5.2.2　设置警告信息

GCC 可以通过使用不同的选项来指定编译过程中会提示的警告信息，GCC 编译器的警告选项如表 5.2 所示。

表 5.2　GCC 编译器的警告选项

选　项	说　明
-Wall	启用所有警告信息
-Werror	在发生警告时取消编译操作，即将警报看做是错误
-w	禁用所有警告信息

GCC 提供了非常丰富的警告信息，但前提是已经启用了它们，否则它不会报告检测到的问题。GCC 提供的-Wall 选项，可以在编译时把所有的警告信息都显示出来，进而帮助程序员发现一些潜在错误。

【实例 5-1】 下面给出一段代码，使用 GCC 进行编译，同时开启警告信息。

```
01  /* example1.c */
02  #include <stdio.h>
03  int main()
04  {
05      printf("hello world");
06  }
```

对上面的源程序进行编译连接：

```
gcc example1.c -o example1 -Wall
example1.c:5:1: 警告： 在有返回值的函数中，控制流程到达函数尾 [-Wreturn-type]
```

从上面的输出可以看到，GCC 给出了警告信息，意思是 main()函数的返回值被声明为 int，但实际上并没有显式地返回值。GCC 检测到了警告信息，但却继续编译并生成了可执行文件，这是因为警告只是针对程序结构的诊断信息，它不能说明程序一定有错误，只能说明程序存在一定风险，或者可能存在错误。

使用-Werror 选项后，GCC 发现源程序中存在可疑之处时，并不是简单地发出警告信息，而是将警告作为一个错误，进而中断整个编译过程，这样会得到高质量的代码。

如果用户不愿看到警告信息，可以使用-w 选项来禁用所有的警告，即告知 GCC 不要显示任何警告信息。

此外，GCC 中还提供了许多以-W 开头的选项，允许用户指定输出某个特定方面的警告信息，如表 5.3 所示。

表 5.3　指定输出警告信息选项

选　　项	说　　明
-Wcomment	出现注释嵌套时发出警告，即在“/*”之后，“*/”之前，出现了第二个“/*”
-Wconversion	如果程序中存在隐式类型转换，则发出警告
-Wformat	检查 printf 和 scanf 等格式化输入输出函数的格式字符串和参数类型的匹配情况，如果发现不匹配则发出警告
-Winline	如果函数不能被内联，则发出警告
-Wlong-long	如果使用了 long long 型数据，则发出警告
-Wmain	如果 main()函数的返回类型不是 int 型，或调用 main()函数时使用的参数数目不正确，则发出警告
-Wmissing-declarations	如果定义了全局函数，但没有在头文件中声明，则发出警告
-Wparentheses	在某些情况下，如果忽略了括号，编译器就会发出警告
-Wreturn-type	如果函数定义了返回类型，而默认类型是 int 型，编译器就发出警告。另外，如果它们所属的函数为非 void 型，不带返回值的 return 语句，也会给出警告信息
-Wundef	如果在#if 宏中使用了未定义的变量做判断，则发出警告
-Wuninitialized	如果使用的自动变量没有被初始化，则发出警告
-Wunused	如果声明的变量或 static 型函数没有使用，则发出警告

【实例 5-2】 下面使用 GCC 编译一段程序，来说明开启警告信息的必要性。

```
01  /* example2.c */
02  #include <stdio.h>
03  int main()
04  {
05    int i;
06    printf("%d\n",i);              /* 变量 i 在使用时未初始化 */
07    return 0;
08  }
```

对上面的源程序进行编译：

```
gcc example2.c -o example2
```

可以看到，编译并没有报错，运行可以执行文件 example2，输出结果为：

```
./example2
-1216622604
```

这不是我们想要的，如果在上面的编译中加入了-Wformat 或-Wall 选项，即：

```
gcc example2.c -o example2 -Wformat
```

或

```
gcc example2.c -o example1 -Wall
```

GCC 会给出如下警告信息：

```
example2.c: 在函数'main'中:
example2.c:6:9: 警告： 此函数中的'i'在使用前未初始化 [-Wuninitialized]
```

变量在使用时未初始化导致了警告，所以这是个非常有用的警告选项。

【实例 5-3】 下面使用 GCC 编译一段程序，使用-Wparentheses 选项对其中的括号进行检查。

```
01  /* example3.c */
02  #include <stdio.h>
03  int main()
04  {
05    int i=1;
06    int j=0;
07    int k=1;
08    if (k || i && j)
    /* 对于运算符优先级不是特别清楚的程序员会在这里出错,所以编译器会建议加上括号 */
09    {
10      ;
11    }
12    if (i == 1)
13      if (j == 1)
14        ;
15    else
            /* 这里的距离 else 最近的 if 语句,即 if (j == 1),为了避免错误,
            编译器会建议加上括号 */
16      ;
17    return 0;
18  }
```

对上面的源程序进行编译：

```
gcc example3.c -o example3 -Wparentheses
example3.c: 在函数'main'中:
example3.c:8:3: 警告:建议在'||'的操作数中出现的'&&'前后加上括号[-Wparentheses]
example3.c:12:6: 警告:建议显式地使用花括号以避免出现有歧义的'else'[-Wparentheses]
```

上面介绍的各个选项是用来开启各警告功能的，当然也可以通过设置将其关闭。例如，想要使用-Wall 来启用所有的警告信息，同时又想关闭 unused 警告，可以通过如下命令来实现：

```
$ gcc test.c -o test -Wall -Wno-unused
```

它的功能是开启除-Wunused 外的所有其他选项。

5.2.3　设置优化级别

GCC 具有优化代码的功能，它的优化功能也有多种不同的选项，主要的优化选项如表 5.4 所示。

表 5.4　优化选项

选　　项	说　　明
-O0	不进行优化处理
-O 或-O1	进行基本的优化，这些优化在大多数情况下都会使程序执行得更快
-O2	除了完成-O1 级别的优化外，还要一些额外的调整工作，如处理器指令调度等，这是 GNU 发布软件的默认优化级别
-O3	除了完成-O2 级别的优化外，还进行循环的展开以及其他一些与处理器特性相关的优化工作
-Os	生成最小的可执行文件，主要用在嵌入式领域

一般来讲，优化的级别越高，生成的可执行文件的运行速度也越快，但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。

这里推荐读者使用-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，它是最安全的优化选项。对于桌面应用，可以尝试-O3 选项，但不建议应用在服务器上，其实它们之间的速度差异并不是很明显，因为-O3 在优化时对循环进行了展开，这会使可执行文件增大，速度是否增加取决于特定环境。

-O2 选项已经启用绝大多数安全的优化选项，-O3 选项是在-O2 选项的基础上又增添了一些，其实用户也可以根据需要，在-O2 选项的基础自行添加，这样比直接使用-O3 选项更加安全。增添如下所述的选项。

- -finline-functions：允许编译器将一些简单的函数在其调用处展开；
- -funswitch-loops：将循环体中值不改变的变量移到循环体之外。

具体的命令格式为：

```
gcc -O2 -finline-functions example.c -o example
```

【实例 5-4】 下面给出一个实例，来查看 GCC 优化选项的效果。

```
/* example4.c */
#include <stdio.h>
int main()
{
    int i;
    int j;
    int x=0;
    for(i=0;i<100000;i++)
    {
        for(j=i;j>0;j--)
        {
            x+=x;
        }
    }
    return 0;
}
```

首先不加任何优化选项，对上面的源程序进行编译：

```
gcc example4.c -o example4
```

下面使用 Linux 系统下的 time 命令来统计程序的运行时间。

```
time ./example4
real    0m11.978s
user    0m11.969s
sys 0m0.004s
```

time 命令的输出结果由以下 3 个部分组成。

- real：程序的总执行时间，包括进程的调度、切换等时间；
- user：用户进程执行的时间；
- sys：内核执行的时间。

接下来使用优化选项-O2 对上面的源程序进行处理：

```
gcc -O2 example4.c -o example4
```

再次统计程序的运行时间：

```
time ./example4
real    0m0.002s
user    0m0.000s
sys 0m0.000s
```

从上面的输出结果可以看出，程序的性能得到大幅度的改善。

此外，还有一个比较重要的优化选项-march，它表示为特定的 CPU 类型编译二进制代码，进而取得最佳的优化效果。具体的命令格式为：

```
gcc --march=<CPU 类型> example.c -o example
```

CPU 类型如 pentium4、pentium4m、pentium-m 或 athlon64 等。

5.2.4　设置连接器

GCC 编译器提供的连接器也有多个选项，如表 5.5 所示。

表 5.5　GCC 编译器的连接器选项

选　　项	说　　明
-Idirectory	向 GCC 的头文件搜索路径中添加新的目录
-Ldirectory	向 GCC 的库文件搜索路径中添加新的目录
-llibrary	提示连接程序在创建可执行文件时包含指定的库文件
-static	强制使用静态链接库
-shared	生成动态库文件

在讲解 GCC 连接器选项之前，我们首先来区分头文件和库文件这两个基本概念。因为我们在实际编程中，发现很多读者对这两个概念的理解不是很清晰。

头文件包含变量和函数的声明，但没有定义函数的实现。例如，我们经常用到的头文件 stdio.h，其中就包含 printf 和 scanf 等格式化输入输出函数的声明，如果在代码中要用到这些函数，就需要包含该头文件。

函数的具体实现是在库文件中完成的。库文件可分为静态库和动态库，静态库是指编译连接时，将库文件的代码全部加入到可执行文件中，这样运行时就不需要库文件了，但此时生成的可执行文件比较大。静态库的后缀名一般为“.a”。动态库是指在编译连接时并不将库文件的代码加入到可执行文件中，而是在程序执行时，由运行时连接文件加载库文件，这样可以节省系统的开销。动态库的后缀名一般为“.so”。

在源程序中包含头文件时，如果所包含的头文件位于系统默认包含路径之内，只需给出头文件的名字即可，不需指定路径；如果所包含的头文件位于系统默认包含路径之外，则需要在编译时使用-I 选项来指定头文件的路径。例如：

```
gcc example.c -o example -I/home/xxx/include
```

头文件所对应的库文件，如果没有特别指定时，GCC 会到默认的搜索路径下进行查找。如果库文件不在上述目录中，在编译时就需要使用-L 选项来指定库文件的路径。例如：

```
gcc example.c -o example -L/home/xxx/lib
```

说明：Linux 系统中头文件的默认包含路径可以通过环境变量 C_INCLUDE_PATH 来设定，库文件的默认搜索路径可以通过环境变量 LIBRARY_PATH 来设定，程序运行时加载动态库的查找路径可以通过环境变量 LD_LIBRARY_PATH 来设定。

【实例 5-5】 下面使用 GCC 直接指定连接程序在创建可执行文件时包含的库文件。

```
01  /* example5.c */
02  #include <stdio.h>
03  #include <math.h>
04  int main()
05  {
06    int i = 2;
07    printf("i=%d\nsin(i)=%f\n", i, sin(i));
08    return 0;
09  }
```

对上面的源程序进行编译：

```
gcc example5.c -o example5
/tmp/cc7lQ6Bs.o: In function `main':
```

```
example5.c:(.text+0x19): undefined reference to `cos'
collect2: ld 返回 1
```

从上面可以看到，程序在连接时出现了错误。这是因为在 Linux 系统中，默认情况下只会连接 C 语言的标准库/usr/lib/libc.so，而与头文件 math.h 对应的数学库/usr/lib/libm.so，除非显式指定，否则不会被连接。

```
gcc example5.c -o example5 /usr/lib/i386-linux-gnu/libm.so
```

运行上面生成的可执行文件，输出结果为：

```
./example5
i=2
sin(i)=0.909297
```

GCC 编译器为连接函数库还提供了一个快捷的选项-l，命令的格式为：

```
gcc example5.c -o example5 -lm
```

它与上面指定库文件的全路径/usr/lib/libm.so 命令是等价的，这样避免了在命令行中书写很长的路径。上面之所以写为-lm，是因为在 Linux 系统下，库文件在命名时都遵循了一个规范，即以 lib 开头，因此在用-l 选项指定库文件名时可以省去 lib，也就是说 GCC 在对-lm 进行处理时，会自动去连接名为 libm.so 的库文件。

GCC 编译器在默认情况下使用动态库，但如果使用了-static 选项，连接器将忽略动态库，强制使用静态链接库，即使用如下命令：

```
gcc example5.c -o example5 -static -lm
```

此时静态库文件中的代码全部包含到可执行文件中，所以生成的可执行文件比较大。

与动态库连接的可执行文件只包含它需要的函数的引用表，而不是所有的函数代码，而且只有在程序执行时函数代码才会被拷贝到内存之中。这样可以使可执行文件比较小，进而节省了磁盘空间；更重要的是，如果库文件本身被更新了，不需要重新编译与它连接的源程序。

GCC 编译器提供了-shared 选项来生成动态库文件。

【实例 5-6】 下面首先创建一个动态库文件，然后在源程序中调用库文件中的函数，实现相应的功能。

```
01  /* hello.c */
02  #include <stdio.h>
03  void printhello()
04  {
05    printf("hello world\n");
06  }
```

对上面的源程序进行编译，生成动态库文件 abc.so。

```
gcc -shared -o libabc.so abc.c
```

在其他文件中调用该库文件。

```
01  /* example6.c */
02  #include <stdio.h>
03  extern void printhello();
```