第 1 章　编译与调试

1.1　编译的概念和理解

在进行 C 程序开发时，编译就是将编写的 C 语言代码变成可执行程序的过程，这一过程是由编译器来完成的。编译器就是完成程序编译工作的软件，在进行程序编译时完成了一系列复杂的过程。

1.1.1　程序编译的过程

在执行这一操作时，程序完成了复杂的过程。一个程序的编译，需要完成词法分析、语法分析、中间代码生成、代码优化、目标代码生成。本章将讲解这些步骤的作用与原理。

（1）词法分析。指的是对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号。然后把字符串的源程序改造成为单词符号串的中间程序。在编译程序时，这一过程是自动完成的。编译程序会对代码的每一个单词进行检查。如果单词发生错误，编译过程就会停止并显示错误。这时需要对程序中的错误进行修改。

（2）语法分析。语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语句。例如，需要检查表达式、赋值、循环等结构是否完整和符合使用规则。在语法分析时，会分析出程序中错误的语句，并显示出结果。如果语法发生错误，编译任务是不能完成的。

（3）中间代码生成。中间代码是源程序的一种内部表示，或称中间语言。程序进行词法分析和语法分析以后，将程序转换成中间代码。这一转换的作用是使程序的结构更加简单和规范。中间代码生成操作是一个中间过程，与用户是无关的。

（4）代码优化。代码优化是指对程序进行多种等价变换，使得从变换后的程序能生成更有效的目标代码。用户可以在编译程序时设置代码优化的参数，可以针对不同的环境和设置进行优化。

（5）目标代码生成。目标代码生成指的是产生可以执行的应用程序，这是编译的最后一个步骤。生成的程序是二进制的机器语言，用户只能运行这个程序，而不能打开这个文件查看程序的代码。

1.1.2　编译器

所谓编译器，是将编写出的程序代码转换成计算机可以运行的程序的软件。在进行 C 程序开发时，编写出的代码是源程序的代码，是不能直接运行的。需要用编译器编译成可以运行的二进制程序。

在不同的操作系统下面有不同的编译器。C 程序是可以跨平台运行的。但并不是说 Windows 系统下 C 语言编写的程序可以直接在 Linux 下面运行。Windows 下面 C 语言编写的程序，被编译成 exe 文件。这样的程序只能在 Windows 系统下运行。如果需要在 Linux 系统下运行，需要将这个程序的源代码在 Linux 系统重新编译。不同的操作系统下面有不同的编译器。Linux 系统下面编译生成的程序是不能在 Windows 系统上运行的。

1.2　gcc 编译器

gcc 是 Linux 下的 C 程序编译器，具有非常强大的程序编译功能。在 Linux 系统下，C 语言编写的程序代码一般需要通过 gcc 来编译成可执行程序。

1.2.1　gcc 编译器简介

Linux 系统下的 gcc 编译器（GNU C Compiler）是一个功能强大、性能优越的编译器。gcc 支持多种平台的编译，是 Linux 系统自由软件的代表作品。gcc 本来只是 C 编译器的，但是后来发展为可在多种硬体平台上编译出可执行程序的超级编译器。各种硬件平台对 gcc 的支持使得其执行效率与一般的编译器相比平均效率要高 20%~30%。gcc 编译器能将 C、C++源程序、汇程语言和目标程序进行编译链接成可执行文件。通过支持 make 工具，gcc 可以实施项目管理和批量编译。

经过多年的发展，gcc 已经发生了很大的变化。gcc 已经不仅仅能支持 C 语言，还支持 Ada 语言、C++语言、Java 语言、Objective C 语言、Pascal 语言、COBOL 语言等更多的语言集的编译。gcc 几乎支持所有的硬件平台，使得 gcc 对于特定的平台可以编译出更高效的机器码。

gcc 在编译一个程序时，一般需要完成预处理（preprocessing）、编译（compilation）、汇编（assembly）和链接（linking）过程。使用 gcc 编译 C 程序时，这些过程是使用默认的设置自动完成的，但是用户可以对这些过程进行设置，控制这些操作的详细过程。

1.2.2　gcc 对源程序扩展名的支持

扩展名指的是文件名中最后一个点的这个点以后的部分。例如下面是一个 C 程序源文件的扩展名。

```
5.1.c
```

那么这个文件的文件名是“5.1.c”，扩展名是“.c”。通常来说，源文件的扩展名标识源文件所使用的编程语言。例如 C 程序源文件的扩展名一般是“.c”。对编译器来说，扩展名控制着缺省语言的设定。在默认情况下，gcc 通过文件扩展名来区分源文件的语言类型。然后根据这种语言类型进行不同的编译。gcc 对源文件的扩展名约定如下所示。

- .c 为扩展名的文件，为 C 语言源代码文件。
- .a 为扩展名的文件，是由目标文件构成的库文件。
- .C,.cc 或.cpp 为扩展名的文件，标识为 C++源代码文件。
- .h 为扩展名的文件，说明文件是程序所包含的头文件。
- .i 为扩展名的文件，标识文件是已经预处理过的 C 源代码文件，一般为中间代码文件。

- .ii 为扩展名的文件，是已经预处理过的 C++源代码文件，同上也是中间代码文件。
- .o 为扩展名的文件，是编译后的目标文件，源文件生成的中间目标文件。
- .s 为扩展名的文件，是汇编语言源代码文件。
- .S 为扩展名的文件，是经过预编译的汇编语言源代码文件。
- .o 为扩展名的文件，是编译以后的程序目标文件（Object file），目标文件经过连接成可执行文件

此外，对于 gcc 编译器提供两种显示的编译命令，分别对应于编译 C 和 C＋＋源程序的编译命令。

1.3 C 程序的编译

本章以一个实例讲述如何用 gcc 编译 C 程序。在编译程序之前，需要用 VIM 编写一个简单的 C 程序。在编译程序时，可以对 gcc 命令进行不同的设置。

1.3.1 编写第一个 C 程序

本节将编写第一个 C 程序。程序实现一句文本的输出和判断两个整数的大小关系。本书中编写程序使用的编辑器是 VIM。程序编写步骤如下所示。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。

② 在终端中输入下面的命令，在用户根目录“root”中建立一个目录。

```
mkdir c
```

③ 在终端界面中输入“vim”命令，然后按“Enter”键，系统会启动 VIM。

④ 在 VIM 中按“i”键，进入到插入模式。然后在 VIM 中输入下面的程序代码。

```
#include <stdio.h>

int max(int i,int j )
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
}

void main()
{
   int i ,j,k;
   i=3;
   j=5;
   printf("hello ,Linux.\n");
```

```
    k=max(i,j);
    printf("%d\n",k);
}
```

⑤ 代码输入完成以后，按“Esc”键，返回到普通模式。然后输入下面的命令，保存文件。

```
:w /root/c/a.c
```

这时，VIM 会把输入的程序保存到 c 目录下的文件 a.c 中。

⑥ 再输入“:q”命令，退出 VIM。这时，已经完成了这个 C 程序的编写。

1.3.2　用 gcc 编译程序

上面编写的 C 程序，只是一个源代码文件，还不能作为程序来执行。需要用 gcc 将这个源代码文件编译成可执行文件。编译文件的步骤如下所示。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。这时进入的目录是用户根目录“/root”。然后输入下面的命令，进入到 c 目录。

```
cd c
```

② 上一节编写的程序就存放在这个目录中。输入“ls”命令可以查看这个目录下的文件。显示的结果如下所示。

③ 输入下面的命令，将这个代码文件编译成可执行程序。

```
gcc a.c
```

④ 查看已经编译的文件。在终端中输入“ls”命令，显示的结果如下所示。

```
a.c a.out
```

⑤ 输入下面的命令对这个程序添加可执行权限。

```
chmod +x a.out
```

⑥ 输入下面的命令，运行这个程序。

```
./a.out
```

⑦ 程序的运行结果如下所示。

```
hello ,Linux.
5
```

从上面的操作可知，用 gcc 可以将一个 C 程序源文件编译成一个可执行程序。编译以后的程序需要添加可执行的权限才可以运行。在实际操作中，还需要对程序的编译进行各种设置。

1.3.3 查看 gcc 的参数

gcc 在编译程序时可以有很多可选参数。在终端中输入下面的命令，可以查看 gcc 的这些可选参数。

```
gcc --help
```

在终端中显示的 gcc 的可选参数如下所示。进行程序编译时，可以设置下面的这些参数。

```
用法：gcc [选项] 文件...
选项：
-pass-exit-codes：在某一阶段退出时返回最高的错误码
--help：显示此帮助说明
--target-help：显示目标机器特定的命令行选项
-dumpspecs：显示所有内建 spec 字符串
-dumpversion：显示编译器的版本号
-dumpmachine：显示编译器的目标处理器
-print-search-dirs：显示编译器的搜索路径
-print-libgcc-file-name：显示编译器伴随库的名称
-print-file-name=<库>：显示 <库> 的完整路径
-print-prog-name=<程序>：显示编译器组件 <程序> 的完整路径
-print-multi-directory：显示不同版本 libgcc 的根目录
-print-multi-lib：显示命令行选项和多个版本库搜索路径间的映射
-print-multi-os-directory：显示操作系统库的相对路径
-Wa,<选项>：  将逗号分隔的 <选项> 传递给汇编器
-Wp,<选项>：将逗号分隔的 <选项> 传递给预处理器
-Wl,<选项>：将逗号分隔的 <选项> 传递给链接器
-Xassembler <参数>：将 <参数> 传递给汇编器
-Xpreprocessor <参数>：将 <参数> 传递给预处理器
-Xlinker <参数>：将 <参数> 传递给链接器
-combine：将多个源文件一次性传递给汇编器
-save-temps：不删除中间文件
-pipe：使用管道代替临时文件
-time：为每个子进程计时
-specs=<文件>：  用 <文件> 的内容覆盖内建的 specs 文件
-std=<标准>：指定输入源文件遵循的标准
--sysroot=<目录>：将 <目录> 作为头文件和库文件的根目录
-B <目录>：将 <目录> 添加到编译器的搜索路径中
-b <机器>：为 gcc 指定目标机器(如果有安装)
-V <版本>：运行指定版本的 gcc(如果有安装)
-v：显示编译器调用的程序
-###：与 -v 类似，但选项被引号括住，并且不执行命令
-E：仅作预处理，不进行编译、汇编和链接
-S：编译到汇编语言，不进行汇编和链接
-c：编译、汇编到目标代码，不进行链接
-o <文件>：输出到 <文件>
-x <语言>：指定其后输入文件的语言。允许的语言包括 c、c++、assembler 等。
以 -g、-f、-m、-O、-W 或 --param 开头的选项将由 gcc 自动传递给其调用的不同子进程。若要向这些进程传递其他选项，必须使用 -W<字母> 选项。
```

1.3.4　设置输出的文件

在默认情况下，gcc 编译出的程序为当前目录下的文件 a.out。-o 参数可以设置输出的目标文件。例如下面的命令，可以设置将代码编译成可执行程序 do。

```
gcc a.c -o do
```

也可以设置输出目录文件为不同的目录。例如下面的命令，是将目录文件设置成/tmp 目录下的文件 do。

```
gcc a.c -o /tmp/do
```

输入下面的命令，查看生成的目录文件。结果如下所示，在编译程序时生成的目录为/tmp 目录下的文件 do。

```
-rwxrwxr-x 1 root root 5109 12-28 13:33 /tmp/do
```

1.3.5　查看编译过程

参数-v 可以查看程序的编译过程和显示已经调用的库。输入下面的命令，在编译程序时输出编译过程。

```
gcc -v a.c
```

显示的结果如下所示。

```
使用内建 specs。
目标：i386-redhat-linux
配置为：../configure --prefix=/usr --mandir=/usr/share/man
 --infodir=/usr/share/info --enable-shared --enable-threads=posix
 --enable-checking=release --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions
--enable-languages=c,c++,objc,obj-c++,java,fortran,ada
--enable-java-awt=gtk --disable-dssi --enable-plugin
--host=i386-redhat-linux
线程模型：posix
gcc 版本 4.1.2 20070925 (Red Hat 4.1.2-33)
/usr/libexec/gcc/i386-redhat-linux/4.1.2/cc1
-quiet -v a.c -quiet -dumpbase a.c -mtune=generic -auxbase a -version
-o /tmp/cc8P7rzb.s
忽略不存在的目录“/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../i386-
redhat-linux/include”
#include "..." 搜索从这里开始：
#include <...> 搜索从这里开始：
 /usr/local/include
 /usr/lib/gcc/i386-redhat-linux/4.1.2/include
 /usr/include
搜索列表结束。
GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) (i386-redhat-linux)
由 GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) 编译。
GGC 准则：--param ggc-min-expand=64 --param ggc-min-heapsize=64394
```

```
    Compiler executable checksum: ab322ce5b87a7c6c23d60970ec7b7b31
    a.c: In function ‘main’:
    a.c:16: 警告:‘main’ 的返回类型不是 ‘int’
     as -V -Qy -o /tmp/ccEFPrYh.o /tmp/cc8P7rzb.s
    GNU assembler version 2.17.50.0.18 (i386-redhat-linux) using BFD version
version 2.17.50.0.18-1 20070731
    /usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2
    --eh-frame-hdr --build-id -m elf_i386 --hash-style=gnu -dynamic-linker
    /lib/ld-linux.so.2 /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o
```

从显示的编译过程可知，gcc 自动加载了系统的默认配置，调用系统的库函数完成了程序的编译过程。

1.3.6 设置编译的语言

gcc 可以对多种语言编写的源代码。如果源代码的文件扩展名不是默认的扩展名，gcc 就无法编译这个程序。可以用-x 选择来设置程序的语言。可以用下面的步骤来练习这一操作。

① 输入下面的命令，将 C 程序文件复制一份。

```
cp a.c a.u
```

② 复制出的文件 a.u 是一个 C 程序文件，但扩展名不是默认的扩展名。这时输入下面的命令编译这个程序。

```
gcc a.u
```

③ 显示的结果如下所示，表明文件的格式不能识别。

```
a.u: file not recognized: File format not recognized
collect2: ld 返回 1
```

④ 这时，用-x 参数设置编译的语言，命令如下所示。这样就可以正常地编译文件 a.u。

```
gcc -x 'c' a.u
```

需要注意的是，这里的 c 需要用单引号扩起来。当编译扩展名不是.c 的 C 程序时，需要使用-x 参数。

1.3.7 -asci 设置 ANSIC 标准

ANSIC 是 American National Standards Institute（ANSI：美国标准协会） 出版的 C 语言标准。使用这种标准的 C 程序可以在各种编译器和系统下运行通过。gcc 可以编译 ANSIC 的程序，但是 gcc 中的很多标准并不被 ANSIC 所支持。在 gcc 编译程序时，可以用-ansic 来设置程序使用 ANSIC 标准。例如下面的命令，在设置程序编译时，用 ANSIC 标准进行编译。

```
gcc -asci a.out
5.3.8
```

1.3.8 g++编译 C++程序

gcc 可以编译 C++程序。编译 C 程序和 C++程序时，使用的是不同的命令。编译 C++程

序时，使用的命令是 g++。该命令的使用方法与 gcc 是相似的。下面是使用 g++命令编译 C++程序的实例。

下面是一个 C++程序的代码，实现与 1.3.1 节程序同样的功能。C++程序的代码与 C 程序的代码非常相似。

```
#include <iostream>
int max(int i,int j )
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
}

int main()
{

   int i ,j,k;
   i=3;
   j=5;
   printf("hello ,Linux.\n");
   k=max(i,j);
   printf("%d\n",k);
   return(0);
}
```

输入下面的命令，编译这个 C++程序。

```
g++ 5.2.cpp -o 5.2.out
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x 5.2.ou
```

输入下面的命令，运行这个程序，程序的代码如下所示。

```
hello ,Linux.
5
```

从结果可知，这个程序与 1.3.1 节中的程序运行结果是相同的。

1.4　编译过程的控制

编译过程指的是 gcc 对一个程序进行编译时完成的内部处理和步骤。编译程序时会自动完成预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）和链接（Linking）4 个步骤。本节将讲解如何对这 4 个步骤进行控制。

1.4.1 编译过程简介

gcc 把一个程序的源文件，编译成一个可执行文件，中间包括很多复杂的过程。可以用图 1-1 来表示编译中 4 个步骤的作用和关系。

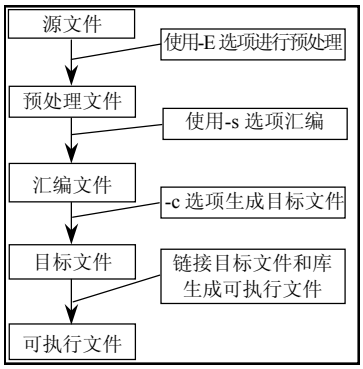

图 1-1 gcc 编译源文件到可执行文件的过程

在 4 个过程中，每一个操作都完成了不同的功能。编译过程的功能如下所示。

- 预处理：在预处理阶段，主要完成对源代码中的预编译语句（如宏定义 define 等）和文件包含进行处理。需要完成的工作是对预编译指令进行替换，把包含文件放置到需要编译的文件中。完成这些工作后，会生成一个非常完整的 C 程序源文件。
- 编译：gcc 对预处理以后的文件进行编译，生成以.s 为后缀的汇编语言文件。该汇编语言文件是编译源代码得到的汇编语言代码，接下来交给汇编过程进行处理。汇编语言是一种比 C 语言更低级的语言，直接面对硬盘进行操作。程序需要编译成汇编指令以后再编译成机器代码。
- 汇编：汇编过程是处理汇编语言的阶段，主要调用汇编处理程序完成将汇编语言汇编成二进制机器代码的过程。通常来说，汇编过程是将.s 的汇编语言代码文件汇编为.o 的目标文件的过程。所生成的目标文件作为下一步链接过程的输入文件。
- 链接：链接过程就是将多个汇编生成的目标文件以及引用的库文件进行模块链接生成一个完整的可执行文件。在链接阶段，所有的目标文件被安排在可执行程序中的适当的位置。同时，该程序所调用到的库函数也从各自所在的函数库中链接到程序中。经过了这个过程以后，生成的文件就是可执行的程序。

1.4.2 控制预处理过程

参数-E 可以完成程序的预处理工作而不进行其他的编译工作。下面的命令，可以将本章编写的程序进行预处理，然后保存到文件 a.cxx 中。

```
gcc -E -o a.cxx a.c
```

输入下面的命令，查看经过预处理以后的 a.cxx 文件。

```
vim a.cxx
```

可以发现，文件 a.cxx 约有 800 行代码。程序中默认包含的头文件已经被展开写到到这个

文件中。显示的文件 a.cxx 前几行代码如下所示。可见，在程序编译时，需要调用非常多的头文件和系统库函数。

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "a.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 335 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 360 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 361 "/usr/include/sys/cdefs.h" 2 3 4
# 336 "/usr/include/features.h" 2 3 4
# 359 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 5 "/usr/include/gnu/stubs.h" 2 3 4
# 1 "/usr/include/gnu/stubs-32.h" 1 3 4
# 8 "/usr/include/gnu/stubs.h" 2 3 4
# 360 "/usr/include/features.h" 2 3 4
# 29 "/usr/include/stdio.h" 2 3 4
```

1.4.3　生成汇编代码

参数-S 可以控制 gcc 在编译 C 程序时只生成相应的汇编程序文件，而不继续执行后面的编译。下面的命令，可以将本章中的 C 程序编译成一个汇编程序。

```
gcc -S -o a.s a.c
```

输入下面的命令，查看汇编文件 a.s。可以发现这个文件一共有 60 行代码。这些代码是这个程序的汇编指令。部分汇编程序代码如下所示。

```
    .file   "a.c"
    .text
.globl max
    .type   max, @function
max:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %eax
    cmpl    12(%ebp), %eax
    jle .L2
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    jmp .L4
.L2:
    movl    12(%ebp), %eax
```

```
        movl    %eax, -4(%ebp)
    .L4:
        movl    -4(%ebp), %eax
        leave
        ret
        .size   max, .-max
        .section    .rodata
    .LC0:
        .string "hello ,Linux."
    .LC1:
        .string "%d\n"
        .text
    .globl main
        .type   main, @function
    main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
    .......
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
        .section    .note.GNU-stack,"",@progbits
```

1.4.4 生成目标代码

参数-c 可以使得 gcc 在编译程序时只生成目录代码而不生成可执行程序。输入下面的命令，将本章中的程序编译成目录代码。

```
gcc -c -o a.o  a.c
```

输入下面的命令，查看这个目录代码的信息。

```
file a.o
```

显示文件 a.o 的结果如下所示，显示文件 a.o 是一个可重定位的目标代码文件。

```
a.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

1.4.5 链接生成可执行文件

gcc 可以把上一步骤生成的目录代码文件生成一个可执行文件。在终端中输入下面的命令。

```
gcc a.o -o aa.out
```

这时生成一个可执行文件 aa.out。输入下面的命令查看这个文件的信息。

```
file aa.out
```

显示的结果如下所示，表明这个文件是可在 Linux 系统下运行的程序文件。

```
aa.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, not stripped
```

1.5　gdb 调试程序

所谓调试，指的是对编好的程序用各种手段进行查错和排错的过程。进行这种查错处理时，并不仅仅是运行一次程序检查结果，而是对程序的运行过程、程序中的变量进行各种分析和处理。本节将讲解使用 gdb 进行程序的调试。

1.5.1　gdb 简介

gdb 是一个功能强大的调试工具，可以用来调试 C 程序或 C++程序。在使用这个工具进行程序调试时，主要使用 gdb 进行下面 5 个方面的操作。

- 启动程序：在启动程序时，可以设置程序运行环境。
- 设置断点：断点就是可以在程序设计时暂停程序运行的标记。程序会在断点处停止，用户便于查看程序的运行情况。这里的断点可以是行数、程序名称或条件表达式。
- 查看信息：在断点停止后，可以查看程序的运行信息和显示程序变量的值。
- 分步运行：可以使程序一个语句一个语句的执行，这时可以及时地查看程序的信息。
- 改变环境：可以在程序运行时改变程序的运行环境和程序变量。

1.5.2　在程序中加入调试信息

为了使用 gdb 进行程序调试，需要在编译程序中加入供 gdb 使用的调试信息。方法是在编译程序时使用一个-g 参数。在终端中输入下面的命令，在编译程序时加入调试信息。

```
gcc -g -o a.debug a.c
```

这时，编译程序 a.c，生成一个 a.bedug 的可执行程序。这个可执行程序中加入了供调试所用的信息。

1.5.3　启动 gdb

在调试文件以前，需要启动 gdb。在终端中输入下面的命令。

```
gdb
```

这时，gdb 的启动信息如下所示。这些提示显示了 gdb 的版本和版权信息。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

1.5.4 在 gdb 中加载需要调试的程序

使用 gdb 调试一个程序之前，需要加载这个程序。加载程序的命令是 file。在(gdb)提示符后面输入下面的命令加载程序 a.debug。

```
file a.debug
```

命令的运行结果如下所示，显示已经加载了这个文件，并且使用了系统库文件。

```
Reading symbols from /root/c/a.debug...done.
Using host libthread_db library "/lib/libthread_db.so.1".
```

1.5.5 在 gdb 中查看代码

用 gcc 命令编译程序加入了-g 命令以后，编译后的 a.debug 程序中加入了断点。可以用 list 命令显示程序的源代码和断点。下面的步骤是查看加入断点以后的代码。

① 在(gdb)提示符后面输入下面的命令。

```
list 1
```

② 这时，gdb 会显示第一个断点以前的代码。显示的代码如下所示。

```
1       #include <stdio.h>
2
3       int max(int i,int j )
4       {
5               if(i>j)
6               {
7                       return(i);
8               }
9               else
10              {
(gdb)
```

③ 这时，按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
11                      return(j);
12              }
13      }
14
15      void main()
16      {
17         int i ,j,k;
18         i=3;
19         j=5;
20         printf("hello ,Linux.\n");
(gdb)
```

④ 按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
21         k=max(i,j);
22         printf("%d\n",k);
```

```
23      }
(gdb)
```

1.5.6　在程序中加入断点

程序会运行到断点的位置停止下来，等待用户处理信息或者查看中间变量。如果自动设置的断点不能满足调试要求，可以用 break 命令增加程序的断点。例如需要在程序的第 6 行增加一个断点，可以输入下面的命令。

```
break 6
```

这时 gdb 显示的结果如下所示。

```
Breakpoint 1 at 0x8048402: file a.c, line 6.
```

输入下面的命令，在程序的第 18 行、19 行、21 行增加断点。

```
break 18
break 19
break 21
```

1.5.7　查看断点

命令 info breakpoint 可以查看程序中设置的断点。输入“info breakpoint”命令，结果如下所示。显示程序中所有的断点。

```
1   breakpoint     keep y   0x08048402 in max at a.c:6
2   breakpoint     keep y   0x08048426 in main at a.c:18
3   breakpoint     keep y   0x0804842d in main at a.c:19
4   breakpoint     keep y   0x08048440 in main at a.c:21
```

加上相应的断点编号，可以查看这一个断点的信息。例如下面的命令就是查看第二个断点。

```
info breakpoint 2
```

显示的结果如下所示。

```
2   breakpoint     keep y   0x08048426 in main at a.c:18
```

1.5.8　运行程序

gdb 中的 run 命令可以使这个程序以调试的模式运行。下面的步骤是分步运行程序，对程序进行调试。

① 在(ddb)提示符后输入“run”命令，显示的结果如下所示。

```
Starting program: /root/c/a.debug
warning:      Missing      the      separate      debug      info      file:
/usr/lib/debug/.build-id/ac/2eeb206486bb7315d6ac4cd64de0cb50838ff6.debug
warning:      Missing      the      separate      debug      info      file:
/usr/lib/debug/.build-id/ba/4ea1118691c826426e9410cafb798f25cefad5.debug
Breakpoint 2, main () at a.c:18
18        i=3;
```

② 结果显示了程序中的异常，并将异常记录到了系统文件中。然后在程序的第二个断点的位置第 18 行停下。

③ 这时输入“next”命令，程序会在下一行停下，结果如下所示。

```
19        j=5;
```

④ 输入“continue”命令，程序会在下一个断点的位置停下。结果如下所示。

```
Continuing.
Breakpoint 3, main () at a.c:19
21        k=max(i,j);
```

⑤ 输入“continue”命令，程序运行到结束。结果如下所示，表明程序已经运行完毕正常退出。

```
5
Program exited with code 02.
```

⑥ step 命令与 next 命令的作用相似，对程序实现单步运行。不同之处是，在遇上函数调用时，step 函数可以进行到函数内部。而 next 函数只是一步完成函数的调用 。

1.5.9 变量的查看

print 命令可以在程序的运行中查看一个变量的值。本节将用下面的步骤来讲解变量的查看方法。

① 输入下面的命令，运行程序。

```
run
```

② 程序在第一个断点位置停下。显示的结果如下所示。

```
Breakpoint 2, main () at a.c:18
18        i=3;
```

③ 程序进入第 18 行之前停下，并没有对 i 进行赋值。可以用下面的命令来查看 i 的值。

```
print i
```

④ 显示的结果如下所示，表示 i 现在只是一个任意值。

```
$5 = -1076190040
```

⑤ 输入下面的命令，使程序运行一步。

```
step
```

⑥ 显示的结果如下所示。

```
19        j=5;
```

⑦ 这时程序在 19 行以前停下，这时输入下面的命令，查看 i 的值。

```
print i
```

⑧ 这时显示的 i 的结果如下所示。表明 i 已经赋值为 3。

```
$6 = 3
```

⑨ 这时输入“step”命令，再次输入“step”命令，显示的结果如下所示。

```
21        k=max(i,j);
```

⑩ 这时输入“step”命令，会进入到子函数中，结果如下所示。这时，显示了传递给函数的变量和值。

```
max (i=3, j=5) at a.c:5
5             if(i>j)
```

⑪ 这时，输入“step”命令，显示的结果如下所示，表明函数会返回变量 j。

```
11                    return(j);
```

⑫ 输入下面的命令，查看 j 的值。

```
print j
```

⑬ 显示的结果如下所示，表明 j 的值为 5。

```
$7 = 5
```

⑭ 这时再运行两次“step”命令，显示的结果如下所示。

```
22        printf("%d\n",k);
```

⑮ 这时，输入下面的命令，查看 k 的值。

```
print k
```

⑯ 显示的结果如下所示，表明 k 的值为 5。

```
$8 = 5
```

⑰ 完成了程序的调试运行以后，输入“q”命令，退出 gdb。

1.6　程序调试实例

本节讲解一个程序调试实例。先编写一个程序，在程序运行时，发现结果与预想结果有些不同。然后用 gdb 工具进行调试，通过对单步运行和变量的查看，查找出程序的错误。

1.6.1　编写一个程序

本节将编写一个程序，要求程序运行时可以显示下面的结果。

```
1+1=2
2+1=3 2+2=4
3+1=4 3+2=5 3+3=6
4+1=5 4+2=6 4+3=7 4+4=8
```

很明显，这个程序是通过两次循环与一次判断得到的。程序中需要定义三个变量。下面用这个思路来编写这个程序。

① 打开一个终端。在终端中输入“vim”命令，打开 VIM。

② 在 VIM 中按“i”键，进入到插入模式。然后在 VIM 中输入下面的代码。

```
#include <stdio.h>
main()
{
    int i,j,k;
    for(i=1;i<=4;i++)
    {
        for(j=1;j<=4;j++);
        {
            if(i>=j)
            {
                k=i+j;
                printf("%d+%d=%d ",i,j,k);
            }
        }
        printf("\n");
    }
}
```

③ 在 VIM 中按“Esc”键，返回到普通模式。然后输入下面的命令，保存这个文件。

```
:w /root/c/test.c
```

④ 输入“:q”命令退出 VIM。很容易发现，在第二个循环后

1.6.2 编译文件

本节将对上一节编写的程序进行编译和运行。在运行程序时，会发现程序有错误。

① 在终端中输入下面的命令，编译这个程序。

```
gcc /root/c/test.c
```

② 程序可以正常编译通过，输入下面的命令，运行这个程序。

```
/root/c/a.out
```

③ 程序的显示结果是 4 个空行，并没有按照预想的要求输出结果。

④ 输入下面的命令，对这个程序进行编译。在编译加入-g 参数，为 gdb 调试做准备。

```
gcc -g -o test.debug 6.2.c
```

⑤ 这时，程序可以正常编译通过。输出的文件是 test.debug。这个文件中加入了文件调试需要的信息。

1.6.3 程序的调试

本节将讲述使用 gdb 对上一节编写的程序进行调试，查找出程序中的错误。

① 在终端中输入“gdb”命令，进入到 gdb，显示的结果如下所示。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
    GDB is free software, covered by the GNU General Public License, and you are
    welcome to change it and/or 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.
    This GDB was configured as "i386-redhat-linux-gnu".
```

② 导入文件。在 gdb 中输入下面的命令。

```
    file /root/c/test.debug
```

③ 这时显示的结果如下所示。表明已经成功加载了这个文件。

```
    Reading symbols from /root/c/test.debug...(no debugging symbols
found)...done.
    Using host libthread_db library "/lib/libthread_db.so.1".
```

④ 查看文件。在终端中输入下面的命令。

```
    list
```

⑤ 显示的文件查看结果如下所示。

```
    1       #include <stdio.h>
    2
    3       main()
    4       {
    5               int i,j,k;
    6               for(i=1;i<=4;i++)
    7               {
    8                       for(j=1;j<=4;j++);
    9                       {
    10                              if(i>=j)
    (gdb)
    11                              {
    12                                      k=i+j;
    13                                      printf("%d+%d=%d ",i,j,k);
    14                              }
    15                      }
    16                      printf("\n");
    17              }
    18      }
    (gdb)
    Line number 19 out of range; 6.2.c has 18 lines.
```

⑥ 在程序中加入断点。从显示的代码可知，需要在第 6 行、第 11 行、第 12 行和第 13 行加入断点。在 gdb 中输入下面的命令。

```
    break 6
    break 11
    break 12
    break 13
```

⑦ gdb 显示的添加断点的结果如下所示。

```
Breakpoint 1 at 0x8048405: file 6.2.c, line 6.
Breakpoint 2 at 0x8048429: file 6.2.c, line 11.
Breakpoint 3 at 0x8048429: file 6.2.c, line 12.
Breakpoint 4 at 0x8048432: file 6.2.c, line 13.
```

⑧ 输入下面的命令，运行这个程序。

```
run
```

⑨ 运行到第一个断点显示的结果如下所示。

```
Breakpoint 1, main () at 6.2.c:6
6             for(i=1;i<=4;i++)
```

⑩ 输入“step”命令，程序运行一步，结果如下所示。

```
8                     for(j=1;j<=4;j++);
```

⑪ 这说明程序已经进入了 for 循环。这时输入下面命令，查看 i 的值。

```
print i
```

⑫ 显示的结果如下所示。

```
$2 = 1
```

⑬ 这时再输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑭ 这时再输入“step”命令，显示的结果如下所示。

```
16                    printf("\n");
```

⑮ 这表明，在进行 j 的 for 循环时，没有反复执行循环体。这时再输入“step”命令，显示的结果如下所示。

```
 for(i=1;i<=4;i++)
```

⑯ 这表明，程序正常的进行了 i 的 for 循环。这是第二次执行 for 循环。

⑰ 输入“step”命令，显示的结果如下所示。

```
8                   for(j=1;j<=4;j++);
```

⑱ 这表明，程序执行到 for 循环。这时再次输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑲ 输入“step”命令，显示的结果如下所示。

```
16                    printf("\n");
```

⑳ 输入“step”命令，显示的结果如下所示。

```
 6             for(i=1;i<=4;i++)
```

㉑ 这说明，程序正常的进行了 i 的 for 循环，但是没有执行 j 的 for 循环。这一定是 j 的

for 循环语句有问题。这时就不难发现 j 的 for 循环后面多了一个分号。

㉒ 输入“q”命令，退出 gdb。

1.6.4　gdb 帮助的使用

gdb 有非常多的命令。输入“help”命令可以显示这些命令的帮助信息。本节将讲解帮助信息的使用。

① 在 gdb 输入“help”命令，显示的帮助信息如下所示。

```
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

② 上面的帮助信息显示，输入“help all”会输出所有帮助信息。

③ 在“help”命令后面加上一个命令名称，可以显示这个命令的帮助信息。例如输入“help file”，显示的 file 命令帮助信息如下所示。

```
Use FILE as program to be debugged.
It is read for its symbols, for getting the contents of pure memory,
and it is the program executed when you use the `run' command.
If FILE cannot be found as specified, your execution directory path
($PATH) is searched for a command of that name.
No arg means to have no executable file and no symbols.
```

1.7　gdb 常用命令

除了前面讲述的 gdb 命令以外，gdb 还有很多种命令。这些命令可以完成程序调试的各种功能。其他的常用命令含义如下所示。

- backtrace：显示程序中的当前位置和表示如何到达当前位置的栈跟踪（同义词：where）。
- breakpoint：在程序中设置一个断点。
- cd：改变当前工作目录。
- clear：删除刚才停止处的断点。

- commands：命中断点时，列出将要执行的命令。
- continue：从断点开始继续执行。
- delete：删除一个断点或监测点，也可与其他命令一起使用。
- display：程序停止时显示变量和表达式。
- down：下移栈帧，使得另一个函数成为当前函数。
- frame：选择下一条 continue 命令的帧。
- info：显示与该程序有关的各种信息。
- info break：显示当前断点清单，包括到达断点处的次数等。
- info files：显示被调试文件的详细信息。
- info func：显示所有的函数名称。
- info local：显示当函数中的局部变量信息。
- info prog：显示被调试程序的执行状态。
- info var：显示所有的全局和静态变量名称。
- jump：在源程序中的另一点开始运行。
- kill：异常终止在 gdb 控制下运行的程序。
- list：列出相应于正在执行的程序的源文件内容。
- next：执行下一个源程序行，从而执行其整体中的一个函数。
- print：显示变量或表达式的值。
- pwd：显示当前工作目录。
- pype：显示一个数据结构（如一个结构或 C++类）的内容。
- quit：退出 gdb。
- reverse-search：在源文件中反向搜索正规表达式。
- run：执行该程序。
- search：在源文件中搜索正规表达式。
- set variable：给变量赋值。
- signal：将一个信号发送到正在运行的进程。
- step：执行下一个源程序行，必要时进入下一个函数。
- undisplay display：命令的反命令，不要显示表达式。
- until：结束当前循环。
- up：上移栈帧，使另一函数成为当前函数。
- watch：在程序中设置一个监测点（即数据断点）。
- whatis：显示变量或函数类型。

1.8 编译程序常见的错误与问题

在编写程序时，无论是逻辑上还是语法上，不可能一次做到完全正确。于是在编译程序时，就会发生编译错误。本节将讲述程序编译时常见的错误类型与处理方法。

1.8.1　逻辑错误与语法错误

在编程时，出现的错误可能有逻辑错误和语法错误两种。这两种错误的发生原因和处理方法是不同的。本节将讲述这两种错误的处理方法。

- 逻辑错误指的是程序的设计思路发生了错误。这种错误在程序中是致命的，程序可能正常编译通过，但是结果是错误的。当程序正常运行而结果错误时，一般都是编程的思路错误。这时，需要重新考虑程序的运算方法与数据处理流程是否正确。
- 语法错误：语法错误指的是程序的思路正确，但是在书写语句时，发生了语句错误。这种错误一般是编程时不小心或是对语句的错误理解造成的。在发生语句错误时，程序一般不能正常编译通过。这时会提示错误的类型和错误的位置，按照这些提示改正程序的语法错误即可完成错误的修改。

1.8.2　C 程序中的错误与异常

C 程序中的错误，根据严重程序的不同，可以分为异常与警误两类。在编译程序时，这两种情况对编译的影响是不同的，对错误与异常的处理方式是不同的。

1．什么是异常

异常指的是代码中轻微的错误，这些错误一般不会影响程序的正常运行，但是不完全符合编程的规范。在编译程序时，会产生一个“警告”，但是程序会继续编译。下面的程序会使程序发生异常，在编译时产生一个警告错误。

- 在除法中，0 作除数。
- 在开方运算时，对负数开平方。
- 程序的主函数没有声明类型。
- 程序的主函数没有返回值。
- 程序中定义了一个变量，但是没有使用这个变量。
- 变量的存储发生了溢出。

2．什么是错误

错误指的是程序的语法出现问题，程序编译不能正常完成，产生一个错误信息。这时会显示错误的类型与位置。根据这些信息可以对程序进行修改。

1.8.3　编译中的警告提示

在编译程序时，如果发生了不严重的异常，会输出一个错告错误，然后完成程序的编译。例如下面的内容是一个程序在编译时产生的警告。

```
5.1.c: In function 'main':
5.1.c:16: 警告：‘main’ 的返回类型不是 ‘int’
5.1.c:18: 警告：被零除
```

这些的含义如下所示。

（1）“In function 'main':”表示发生的异常在 main 函数内。

（2）“5.1.c:16:”表示发生异常的文件是 5.1.c，位置是第 16 行。

（3）下面的信息是第 16 行的异常，表明程序的返回类型不正确。

```
‘main’ 的返回类型不是 ‘int’
```

（4）下面的警告信息表明程序的第 18 行有除数为 0 的错误。

```
5.1.c:18: 警告：被零除
```

1.8.4 找不到包含文件的错误

程序中的包含文件在系统或工程中一定要存在，否则程序编译时会发生致命错误。例如下面的语句包含了一个不正确的头文件。

```
#include <stdio1.h>
```

编译程序时，会发生错误，错误信息如下所示。

```
5.1.c:2:20: 错误：stdio2.h：没有那个文件或目录
```

1.8.5 错误地使用逗号

程序中逗号的含义是并列几个内容，形成某种算法或结构。程序中如果错误地使用逗号，会使程序在编译时发生致命错误。例如下面的代码，是程序中的 if 语句后面有一个错误的逗号。

```
int max(int i,int j )
{
    if(i>j),
    {
        return(i);
    }
    else
    {
        return(j);
    }
}
```

程序编译时输出的错误信息如下所示。表明 max 函数中逗号前面的表达式有错误，实际上的错误是多一个逗号。

```
5.1.c: In function ‘max’:
5.1.c:4: 错误：expected expression before ‘,’ token
5.1.c: In function ‘max’:
```

1.8.6 括号不匹配错误

程序中的引号、单引号、小括号、中括号、大括号等符号必须成对出现。这方面的错误会使程序发生符号不匹配的错误。发生这种错误后，编译程序往往不能理解代码的含义，也不能准确显示错误的位置，而是显示表达式错误。例如下面的代码，在最后一行上了一个花括号。

```
int max(int i,int j )
```

```
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
```

编译程序时，会显示下面的错误信息。

```
5.1.c:22: 错误: expected declaration or statement at end of input
```

1.8.7　小括号不匹配错误

程序中的小括号一般在一行内成对出现并且相匹配。小括号不匹配时，程序发生致命错误。例如下面的代码，第一行多了一个右半边括号。

```
if(i>j))
    {
        return(i);
    }
    else
    {
        return(j);
    }
```

编程程序时，会发生下面的错误。显示括号前面有错误，并且导致下面的 else 语句也有错误。

```
5.1.c:4: 错误: expected statement before ‘)’ token
5.1.c:8: 错误: expected expression before ‘else’
```

1.8.8　变量类型或结构体声明错误

程序中的变量或结构体的名称必须正确，否则程序会发生未声明的错误。例如下面的代码，用一个不存在的类型来声明一个变量。

```
ch a;
```

程序在运行时，会显示出这个变量错误，并且会显示有其他的错误。

```
5.1.c:17: 错误: ‘ch’ 未声明 (在此函数内第一次使用)
5.1.c:17: 错误: (即使在一个函数内多次出现，每个未声明的标识符在其
5.1.c:17: 错误: 所在的函数内只报告一次。)
5.1.c:17: 错误: expected ‘;’ before ‘a’
```

1.8.9　使用不存在的函数的错误

如果程序引用了一个不存在的函数，会使用程序发生严重的错误。例如下面的代码，引用了一个不存在的函数 add。

```
k=add(i,j);
```

程序显示的错误信息如下所示，表明在 main 函数中的 add 函数没有定义。

```
/tmp/ccYQfDJy.o: In function `main':
5.1.c:(.text+0x61): undefined reference to `add'
collect2: ld 返回 1
```

5.8.10 大小写错误

C 程序对代码的大小写是敏感的，不同的大小写代表不同的内容。例如下面的代码，将小写的“int”错误的写成了“Int”。

```
Int t;
```

程序显示的错误信息如下所示，表明“Int”类型不存在或未声明。发生这个错误时，会输出多行错误提示。

```
5.1.c:16: 错误：‘Int’ 未声明 (在此函数内第一次使用)
5.1.c:16: 错误：(即使在一个函数内多次出现，每个未声明的标识符在其
5.1.c:16: 错误：所在的函数内只报告一次。)
5.1.c:16: 错误：expected ‘;’ before ‘t’
```

1.8.11 数据类型的错误

程序中的某些运算，必须针对相应的数据类型，否则这个运算会发生数据类型错误。例如下面的代码，错误地将两个整型数进行求余运算。

```
float a,b;
a= a %b ;
```

程序编译时，输出下面的错误，表明“%”运算符的操作数无效。

```
5.1.c:19: 错误：双目运算符 % 操作数无效
```

1.8.12 赋值类型错误

任何一个变量，在赋值时必须使用相同的数据类型。例如下面的代码，错误地将一个字符串赋值给一个字符。

```
char c;
c="a";
```

程序编译时的结果如下所示，表明赋值时数据类型错误。

```
5.1.c:19: 警告：赋值时将指针赋给整数，未作类型转换
```

1.8.13 循环或判断语句中多加分号

分号在程序中的作用是表示一个语句结束。在程序的语句中用一个单独的分号表示一个空语句。但是在循环或判断结构的后面，一个分号会导致程序的逻辑发生错误。关于这些结构的使用方法，后面的章节将会详细讲到。下面的程序，在 for 语句的后面，错误的添加了一

个分号，导致程序不能正常地进行循环。

```
#include <stdio.h>
main()
{
  int sum, j;
  sum=0;
  for(j=0;j<11;j++);
  {
    sum=sum+j;
  }
printf("%d",sum);
}
```

这个程序的本意是要求出 10 以内的整数和。但是在 for 语句的后面，错误地使用了一个分号。这时，程序不能正确地进行循环，而是把分号作为一个语句进行循环，所以程序输出的结果为“11”。

1.9　小结

程序的编译和调试是编程的一个重要环节。本章讲解了 Linux 系统中 C 编程的编译器 gcc 和编译器 gdb 的使用。使用 gcc 时，需要对编译进行各种设置，需要理解 gcc 各项参数的作用。gdb 的学习重点是 gdb 单步运行程序的理解，通过程序的单步运行发现程序中的问题。