第 3 章　Linux 下的 C 编程基础

3.1　概述

3.1.1　C 语言简单回顾

C 语言是一种计算机程序设计语言。它既有高级语言的特点，又具有汇编语言的特点。它可以作为系统设计语言，编写工作系统应用程序，也可以作为应用程序设计语言，编写不依赖于计算机硬件的应用程序。因此，它的应用范围很广泛。

对操作系统和系统实用程序以及需要对硬件进行操作的场合，用 C 语言明显优于其他解释型高级语言，有一些大型应用软件也是用 C 语言编写的。

C 语言最早是由贝尔实验室的 Dennis Ritchie 为了 UNIX 的辅助开发而编写的，它是在 B 语言的基础上开发出来的。尽管 C 语言不是专门针对 UNIX 操作系统或机器编写的，但它与 UNIX 系统的关系十分紧密。由于 C 语言的硬件无关性和可移植性，它逐渐成为世界上使用最广泛的计算机语言。

为了进一步规范 C 语言的硬件无关性，1987 年，美国国家标准化协会（ANSI）根据 C 语言问世以来的各种版本对 C 的发展和扩充制定了新的标准，称为 ANSI C。ANSI C 比原来的标准 C 有了很大的进步，目前流行的 C 编译系统都是以它为基础的。

C 语言的成功并不是偶然的，它强大的功能和可移植性让它能在各种硬件平台上游刃有余。总体而言，C 语言具有如下特点：

（1）简洁紧凑、灵活方便。C 语言一共只有 32 个关键字，9 种控制语句，程序书写自由，主要用小写字母表示。它把高级语言的基本结构和语句与低级语言的实用性结合起来。C 语言可以像汇编语言一样对位、字节和地址进行操作，而这三者是计算机最基本的工作单元。

（2）运算符丰富。C 的运算符包含的范围很广泛，共有 34 个运算符。C 语言把括号、赋值、强制类型转换等都作为运算符处理。从而使 C 的运算类型极其丰富，表达式类型多样化，灵活使用各种运算符可以实现在其他高级语言中难以实现的运算。

（3）数据结构丰富。C 语言的数据类型有整型、实型、字符型、数组类型、指针类型、结构体类型、共用体类型等，能用来实现各种复杂的数据类型的运算，并引入了指针概念，使程序效率更高。另外 C 语言具有强大的图形功能，支持多种显示器和驱动器，且计算功能、逻辑判断功能强大。

（4）C 语言是结构式语言。结构式语言的显著特点是代码及数据的分隔化，即程序的各个部分除了必要的信息交流外彼此独立。这种结构化方式可使程序层次清晰，便于使用、维护和调试。C 语言是以函数形式提供给用户的，这些函数可方便地调用，并具有多种循环、条件语句控制程序流向，从而使程序完全结构化。

（5）C 语法限制不太严格，程序设计自由度大。虽然 C 语言也是强类型语言，但它的语

法比较灵活，允许程序编写者有较大的自由度。

（6）C 语言允许直接访问物理地址，可以直接对硬件进行操作。因此既具有高级语言的功能，又具有低级语言的许多功能，能够像汇编语言一样对位、字节和地址进行操作，而这三者是计算机最基本的工作单元，可以用来写系统软件。

（7）C 语言程序生成代码质量高，程序执行效率高。一般只比汇编程序生成的目标代码效率低 10%～20%。

（8）C 语言适用范围广，可移植性好。C 语言适合多种操作系统，如 DOS、Windows、Linux 等，也适合多种体系结构。

3.1.2　Linux 下的 C 语言编程环境概述

Linux 下的 C 语言程序设计与在其他环境中的 C 语言程序设计一样，主要涉及编辑器、编译器、调试器及项目管理器。现在先对这 4 种工具进行简单介绍，后面会对其一一进行讲解。

（1）编辑器。Linux 下的编辑器就如 Windows 下的 Word、记事本等一样，完成对所录入文字的编辑功能。Linux 中最常用的编辑器有 vi（vim）和 Emacs，它们功能强大，使用方便，广受编程爱好者的喜爱。在本书中，着重介绍 vi 和 Emacs。

（2）编译器。编译是指源代码转化生成可执行代码的过程。可见，编译过程是非常复杂的，它包括词法、语法和语义的分析，中间代码的生成和优化，符号表的管理和出错处理等。在 Linux 中，最常用的编译器是 Gcc 编译器。它是 GNU 推出的功能强大、性能优越的多平台编译器，其执行效率与一般的编译器相比平均要高 20%～30%，堪称为 GNU 的代表作品之一。

（3）调试器。调试器并不是代码执行的必备工具，而是专为方便程序员调试程序的。有编程经验的读者都知道，在编程过程中，往往调试所消耗的时间远远大于编写代码的时间。因此，有一个功能强大、使用方便的调试器是必不可少的。Gdb 是绝大多数 Linux 开发人员所使用的调试器，它可以方便地设置断点、单步跟踪等，足以满足开发人员的需要。

（4）项目管理器。Linux 中的项目管理器 Make 有些类似于 Windows 中 Visual C++里的“工程”，它是一种控制编译或者重复编译软件的工具。另外，它还能自动管理软件编译的内容、方式和时机，使程序员能够把精力集中在代码的编写上而不是在源代码的组织上。

3.2　vi 编辑器

vi 是 Linux 上最基本的文本编辑器，工作在字符模式下。由于不需要图形界面，使它成了效率很高的文本编辑器。尽管在 Linux 上也有很多图形界面的编辑器可用，但 vi 在系统和服务器管理中的效率是那些图形编辑器所无法比拟的。vim 是 vi 的加强版，比 vi 更容易使用。vi 的命令几乎全部都可以在 vim 上使用。

vi 是 visual interface 的简称，它在 Linux 上的地位就像 Edit 程序在 DOS 上的一样。它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。

3.2.1　vi 的工作模式

vi 有 3 种基本的工作模式：命令模式、插入模式和末行模式。

1. 命令模式

当用户启动 vi 后，vi 就处于命令模式。此时输入的任何字符都被当作编辑命令。如 i 表示插入命令，r 表示替换命令等。不管在什么时候，只要按一下 Esc 键，vi 就会回到命令模式。

2. 插入模式

在命令模式下，按字母 i、a、o、r 等就可以切换到插入模式。如果按的是字母 i，则在光标前插入文本；如果按的是字母 a，则在光标后插入文本。进入插入模式后用户即可输入或编辑文本。

3. 末行模式

在插入模式下，按 Esc 键回到命令模式，再按冒号（:）键就会转换到末行模式，此时光标停留在状态行上，并等待用户输入所需的末行模式的命令。用户可以用它来保存文件、装入另外的文件或退出 vi。

3.2.2 vi 的启动和退出

1. 启动 vi

在系统提示符下输入 vi 及文件名称后，就进入 vi 全屏幕编辑界面，如图 3-1 所示。

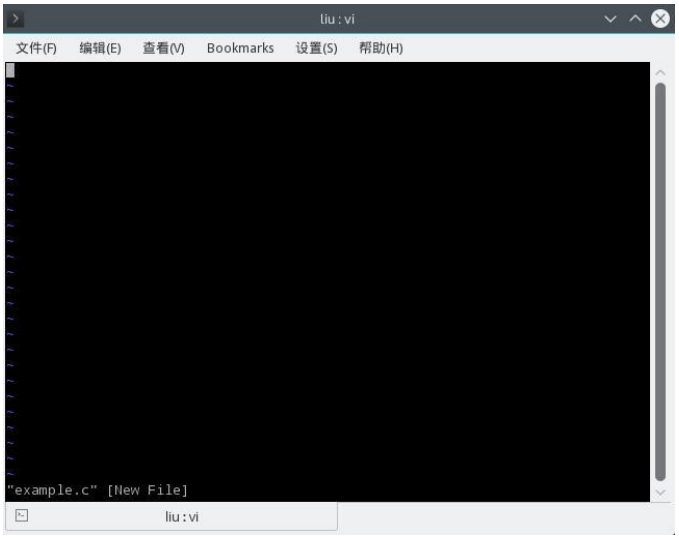

图 3-1 vi 编辑界面

example.c 是一个新文件，光标停留在左上角，在每一行开头都有一个“~”符号，表示空行。如果指定的文件存在，则打开该文件后在屏幕上显示的是该文件的内容，屏幕的最底行是 vi 的状态行，包括文件名、行数和字符数，如图 3-2 所示。

要特别注意的是，进入 vi 之后处于命令模式，切换到插入模式才能输入文字。在命令模式下按字母 i 便可进入插入模式，这时就可以开始输入文字了。

2. 退出 vi

当编辑完文件后，准备返回到 shell 状态时，需要执行退出 vi 的命令。在命令模式下按一下冒号键进入末行模式。

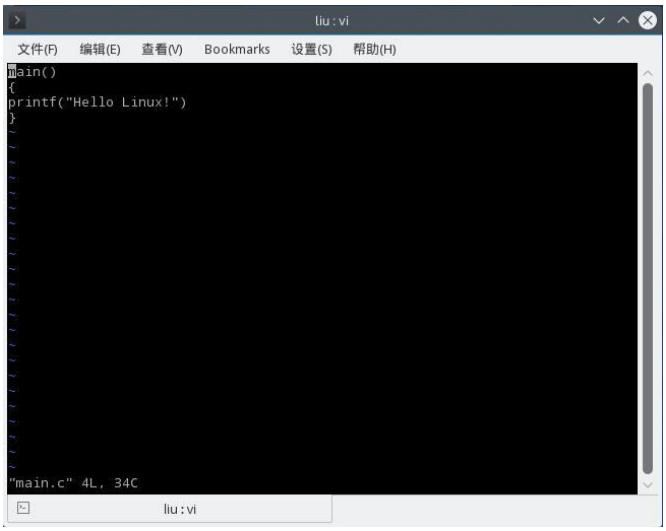

图 3-2　打开 main.c 的界面

（1）:wq：保存文件内容并退出 vi，回到 shell 状态。

（2）:q!：不保存内容强制退出 vi。

（3）:ZZ：仅当编辑的内容做过修改时，才将缓冲区的内容保存到文件。

（4）:x：与:ZZ 的功能相同。

3.2.3　文本输入

1. 插入命令

（1）i：在 i 命令之后输入的内容都插在光标位置之前，光标后的文本相应地向后移动。如果按 Enter 键，则插入新的一行或者换行。

（2）I：输入 I 命令后，光标移动到该行的行首，输入的相应文本则插入到行首的相应位置。

例如，原来屏幕显示为：

```
main()
{
  a,b;
  printf("Hello Linux!");
}
```

按 Esc 键进入命令模式，按下 I 键，光标就移到行首，显示为：

```
main()
{
  a,b;
  printf("Hello Linux!");
}
```

接着输入“int”，则在行首插入所输字符，显示为：

```
main()
{
  int a,b;
  printf("Hello Linux!");
}
```

2. 附加命令

（1）a：在 a 命令之后输入的内容都插在光标位置之后。

（2）A：输入 A 命令后，光标移动到该行的行尾，输入的相应文本则插入到行尾的相应位置。

3. 打开命令

（1）o：在光标所在行的下面新开辟一行，输入的文本就插入该行。

（2）O：在光标所在行的上面新开辟一行，输入的文本就插入该行。

4. 移动光标

要对正文内容进行修改，首先必须把光标移动到指定位置。移动光标的最简单的方式是按键盘上的上、下、左、右箭头键。除了这种最原始的方法外，用户还可以利用 vi 提供的众多字符组合键在正文中移动光标，迅速到达指定的行或列实现定位。移动光标命令如表 3-1 所示。

表 3-1　移动光标命令

命令	作用
k、j、h、l	等同于上、下、左、右箭头键
w	在指定行内右移光标，到下一个字的开头
e	在指定行内右移光标，到下一个字的末尾
b	在指定行内左移光标，到前一个字的开头
0	数字 0，左移光标，到本行的开头
$	右移光标，到本行的末尾
^	移动光标，到本行的第一个非空字符
H	将光标移到屏幕的最上行
nH	将光标移到屏幕的第 n 行
M	将光标移到屏幕的中间
L	将光标移到屏幕的最下行
nL	将光标移到屏幕的倒数第 n 行
Ctrl+b	在文件中向上移动一页（相当于 PageUp 键）
Ctrl+f	在文件中向下移动一页（相当于 PageDown 键）

3.2.4　文本修改

1. 删除与替换

（1）删除。

在插入模式下，用 Backspace 键来删除前面的字符，还可以用 Delete 键来删除当前字符。也可以在 vi 的命令模式下用一些命令来删除一个字符、一个单词或者整行内容等，其删除命令如表 3-2 所示。

对于命令 db，当用户在命令模式先按下 D 键，再按下 B 键，则会删除当前光标所在单词字符至前一个单词开始间的所有字符。对于命令 ndw，用户先按下一个数字键，再先后按下 D 键、W 键，则会删除当前光标所在单词字符至后 n 个单词开始间的所有字符。

表 3-2　删除命令

命令	作用
x	删除当前光标所在的字符
db	删除当前光标所在单词字符至前一个单词开始间的所有字符
ndb	删除当前光标所在单词字符至前 n 个单词开始间的所有字符
dw	删除当前光标所在单词字符至后一个单词开始间的所有字符
ndw	删除当前光标所在单词字符至后 n 个单词开始间的所有字符
d$（或 shift+d）	删除从当前光标至行尾的所有字符
dd	删除当前光标所在行
ndd	删除当前光标后 n 行

例如，原来屏幕显示为：

This is a test file!

按 Esc 键进入命令模式，再先后按 2、D、W 键后，则删除当前光标到 test 单词开始间的所有字符，显示为：

This test file!

（2）替换。

在 vi 的命令模式下还提供了一些命令来替换字符、单词或者进行整行替换，其替换命令如表 3-3 所示。

表 3-3　替换命令

命令	作用
r	替换当前光标所在的字符
R	替换字符序列
cw	替换一个单词
ce	同 cw
cb	替换光标所在的前一字符
c$	替换从当前光标至行尾的所有字符
cc	替换当前光标所在行

例如，原来 vi 编辑器显示为：

This is a test file!

按 Esc 键进入命令模式，再先后输入 RTEST 后，test 被 TEST 替换了，显示为：

This is a TEST file!

2. 复制、粘贴和剪切

（1）复制。

在 vi 编辑器中复制的方式有两种：鼠标方式和命令方式。鼠标方式与 Windows 操作系统的复制操作类似，vi 提供的复制命令如表 3-4 所示。

表 3-4　复制命令

命令	作用
yw	复制当前光标至下一个单词开始的内容
y$	复制当前光标至行尾的内容
yy 或 Y	复制整行

（2）粘贴。

与复制一样，vi 编辑器中粘贴的方式也有两种，且不同的复制方式对应不同的粘贴方式。粘贴方式同 Windows 操作系统的粘贴操作类似，vi 提供的粘贴命令很简单，有以下两种形式。

1）p：在当前光标后面粘贴。

2）P：在当前光标前面粘贴。

使用命令方式粘贴时，先将光标移动到相应位置，然后按下相应的粘贴命令键即可。

（3）剪切。

在 vi 编辑器中，所有的删除命令也是剪切命令，因为删除的内容都被送到剪贴板中。如果用户用剪切命令剪切，可将剪切的内容使用粘贴命令粘贴。

3. 撤消

使用撤消命令可撤消用户最后一次的操作。撤消命令很简单，有以下两种形式。

（1）u：取消上次的操作。

（2）U：可以恢复对光标所在行的所有改变。

4. 查找

vi 提供字符串查找功能，包括向前查找、向后查找、继续上一次查找等。当 vi 向前查找，从光标当前位置向前查找，当找到文本的开头时，它就到文本的末尾继续查找；当 vi 向后查找，从光标当前位置向后查找，当找到文本的末尾时，它就到文本的开头继续查找。

vi 提供的查找命令如表 3-5 所示。

表 3-5　查找命令

命令	作用
?str	向前查找字符串 str
/str	向后查找字符串 str
n	继续查找，找出 str 字符串下次出现的位置
N	与上一次相反的方向查找

例如，要在 main.c 文件中查找 Hello，在命令模式下输入“?Hello”，屏幕显示为：

```
main()
{
   int a,b;
   printf("Hello Linux!");
}
```

光标定位在 Hello 单词上。在查找过程中注意字母是区分大小写的，如果输入“?hello”命令是找不到的，因为文中没有单词 hello。

3.2.5　文件操作

1. 打开文件

（1）打开一个文件。

在 vi 中打开文件方法很简单，在命令模式下使用命令：vi file。其中 file 是指定路径的文件，如果没有指定路径，则默认为当前目录。

例如，输入 vi test 即可打开当前目录下的 test 文件，此时按字母 i 或 a 即可切换到插入模式进行文本输入。

（2）打开多个文件。

vi 能一次打开多个文件，使用命令：vi file1 file2。其中 file1 和 file2 是指定路径的两个文件，如果没有指定路径，则默认为当前目录。

例如，输入“vi test main.c”可以打开当前目录下的 test、main.c 文件。

首先将第一个文件 test 读入缓冲区并显示：

```
This is a test file!
```

按 Esc 键进入命令模式，按冒号键进入末行模式，再输入命令 next，则可以打开第二个文件 main.c 并显示：

```
main()
{
   int a,b;
   printf("Hello Linux!");
}
```

如果想返回到第一个文件，在命令模式下输入命令“: previous”或“: prev”，则又可以打开第一个文件并显示。

（3）打开多个窗口。

vi 也可以在多个窗口中打开多个文件，在命令模式下，使用命令：vi -o file1 file2，即可在两个窗口中分别打开 file1 和 file2。

2. 保存文件

用户在编辑过程中或者编辑完成后要对文件进行保存，在末行模式下，使用如下命令来保存文件。

（1）:w：将缓冲区的内容保存到当前文件中。

（2）:w file：将缓冲区的内容保存到名为 file 的文件中。如果用户另存为的 file 文件已经存在，则使用该命令保存时状态行会出现“File exists(add ! to override)”的提示，即需要使用:w! file 命令来强制覆盖。

（3）:w! file：强制将缓冲区的内容保存到名为 file 的文件中。

3.3　Emacs 编辑器

Emacs 编辑器不仅仅是一款功能强大的编辑器，而且是一款融合编辑、编译、调试于一体的开发环境。Emacs 编辑器既可以在图形界面下完成相应的工作，也可以在没有图形显示的终端环境下出色地工作。

Emacs 编辑器的使用和 vi 编辑器截然不同。Emacs 编辑器只有一种编辑模式，而且它的命令全靠功能键完成。因此，功能键的使用也就相当重要了。但 Emacs 却还使用一个不同于 vi 的“模式”，即各种辅助环境。当编辑普通文本时，使用的是“文本模式”；而当编写程序时，使用的则是 C 模式、shell 模式等。

3.3.1 Emacs 的基本操作

1. *启动* Emacs

在系统提示符下直接输入 Emacs 命令，则进入 Emacs 的欢迎界面，如图 3-3 所示。

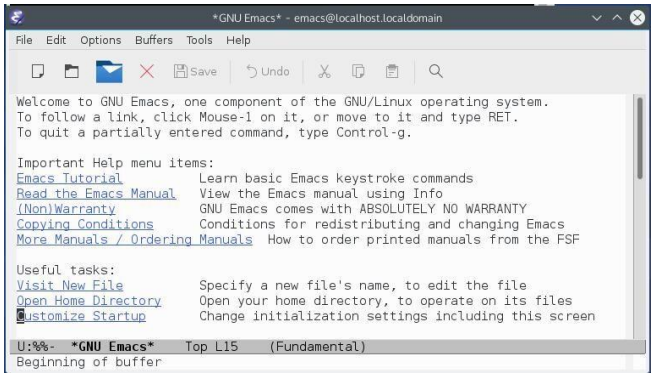

图 3-3 Emacs 的欢迎界面

可按任意键进入 Emacs 的工作窗口，也可以输入 Emacs 及文件名进入 Emacs 的工作窗口，工作窗口如图 3-4 所示。

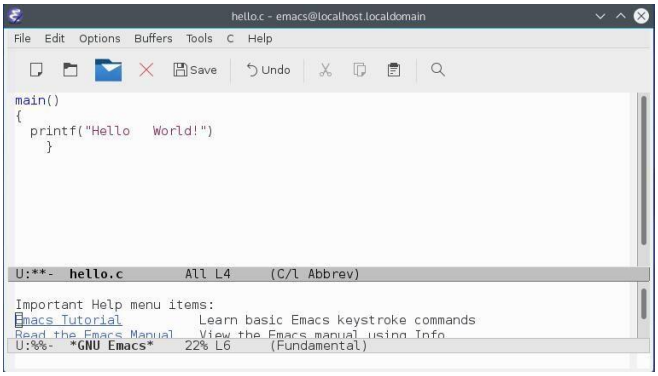

图 3-4 Emacs 的工作窗口

如果指定的文件不存在，则创建一个新文件。如果指定的文件存在，则打开该文件后在屏幕上显示的是该文件的内容。Emacs 的工作窗口分为上下两部分，上部为编辑窗口，底部为命令显示窗口，用户执行功能键的功能都会在底部有相应的显示，有时也需要用户在底部窗口中输入命令执行相应的操作。

2. *使用* Emacs

在进入 Emacs 后，即可进行文件的编辑。下面介绍 Emacs 中的基本编辑功能键。

Emacs 缩写注释：C-<chr>表示按住 Ctrl 键的同时键入字符<chr>，因此 C-f 就是按住 Ctrl

键的同时键入 f；M-<chr>表示当键入字符<chr>时同时按住 Meta 键或 Esc 键或 Alt 键（通常为 Alt 键）。

（1）复制、剪切和粘贴。

Emacs 复制文本包括选择复制区域和粘贴文本两步。选择复制区域的方法是：在复制起始点按 C-Shift-2 键使它成为一个表示点，再将光标移至复制结束点。按 M-w 键即可将复制起始点与结束点之间的文本复制到系统的缓冲区中。将光标移动到要粘贴的位置，按 C-y 键即可将其粘贴到指定位置。

以词和行为单位的剪切和粘贴功能键如表 3-6 所示。

表 3-6　剪切和粘贴功能键

命令	作用
M-Delete	剪切光标前面的单词
M-d	剪切光标前面的单词
M-k	剪切从光标位置到句尾的内容
C-k	剪切从光标位置到行尾的内容
C-y	将缓冲区中的内容粘贴到光标所在的位置
C-x u	撤消操作（先操作 C-x，接着再单击 u）

（2）查找文本。

Emacs 中查找文本的功能键如表 3-7 所示。

表 3-7　查找文本的功能键

命令	作用
C-s	查找光标以后的内容，并在对话框的“I-search:”后输入查找的字符串
C-r	查找光标以前的内容，并在对话框的“I-search backward:”后输入查找的字符串
C-s C-w	查找光标以后的内容，把光标所在的单词作为查找对象
C-r C-w	查找光标以前的内容，把光标所在的单词作为查找对象

（3）移动光标。

Emacs 中移动光标的功能键如表 3-8 所示。

表 3-8　移动光标的功能键

命令	作用
C-f	向前移动一个字符
C-b	向后移动一个字符
C-p	移动到上一行
C-n	移动到下一行
M-f	向前移动一个单词
M-b	向后移动一个单词
C-a	移动到行首
C-e	移动到行尾

（4）打开、保存和退出文件。

Emacs 中打开、保存和退出文件的功能键如表 3-9 所示。

表 3-9　打开、保存和退出文件的功能键

命令	作用
C-x C-f filename	打开一个 filename 文件
C-x C-s	保存文件
C-x C-c	退出 Emacs

3.3.2　Emacs 的编译概述

Emacs 不仅仅是一个强大的编辑器，它还是一个集编译、调试等于一体的工作环境。本节主要介绍 Emacs 作为编译器的最基本的概念。

1．Emacs 中的模式

Emacs 中并没有像 vi 中那样的“命令行”“编辑”模式，只有一种编辑模式。Emacs 的“模式”是指 Emacs 里的各种辅助环境。用 Emacs 打开某一文件时，Emacs 会判断文件的类型，并自动选择相应的模式。当然，用户也可以手动选择各种模式，使用功能键 M-x，然后再输入模式的名称，就启动了如图 3-5 所示的“C 模式”界面。

在 C 模式下，用户拥有“自动缩进”“注释”“预处理扩展”“自动状态”等强大功能。可以用 Tab 键自动地将当前行的代码进行适当的缩进，使代码结构清晰、美观。源代码要有良好的可读性，必须要有良好的注释。在 Emacs 中，用 M-可以产生一条右缩进的注释。C 模式下是“/* comments */”形式的注释，C++模式下是“//comments”形式的注释。当用户高亮选定某段文本，然后操作 C-c C-c 即可注释该段文字，如图 3-6 所示。

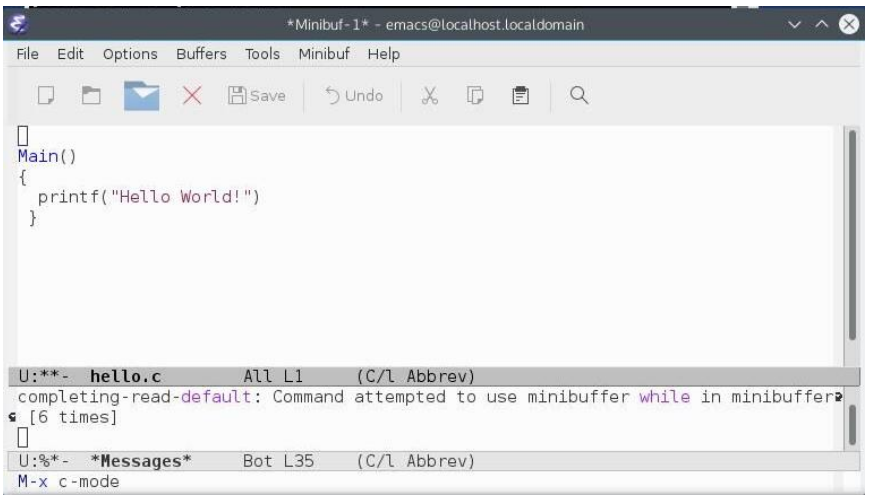

图 3-5　Emacs 的“C 模式”界面

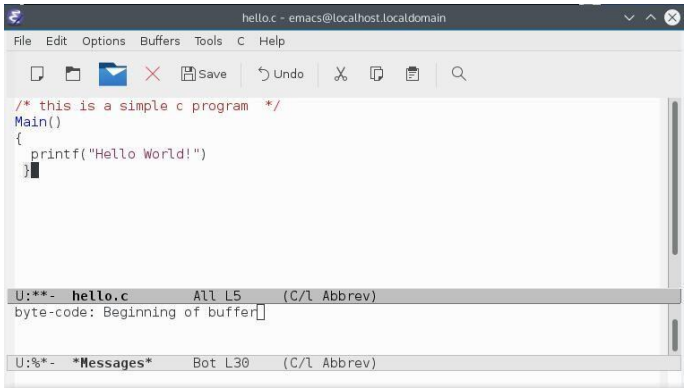

图 3-6　Emacs 的注释

2. Emacs 编译调试程序

Emacs 可以让程序员在 Emacs 环境里编译自己的软件。此时，编辑器把编译器的输出和程序代码连接起来。程序员可以像在 Windows 的其他开发工具中一样，将出错位置和代码定位联系起来。

Emacs 默认的编辑命令是对一个 make 的调用，用户可以打开 tool 下的 Compile 进行查看。Emacs 可以支持大量的工程项目，以方便程序员的开发。

另外，Emacs 为 Gdb 调试器提供了一个功能齐全的接口。在 Emacs 中使用 Gdb 的时候，程序员不仅能够获得 Gdb 用其他任何方式运行时所具有的全部标准特性，还可以通过接口增强而获得其他性能。

3.4　Gcc 编译器

3.4.1　Gcc 简介

Gcc 是 GNU 开源组织的一个项目，是一个用于编程开发的自由编译器。最初，Gcc 只是一个 C 语言编译器，它是 GNU C Compiler 的英文缩写。随着众多自由开发者的加入和 Gcc 自身的发展，如今的 Gcc 已经是一个包含众多语言的编译器了。其中包括 C、C++、Ada、Object C 和 Java 等。所以 Gcc 也由原来的 GNU C Compiler 变为 GNU Compiler Collection，也就是 GNU 编译器家族的意思。当然，如今的 Gcc 借助于它的特性，具有了交叉编译器的功能，即在一个平台下编译另一个平台的代码。

Linux 系统下的 Gcc（GNU C Compiler）是 GNU 推出的功能强大、性能优越的多平台编译器，是 GNU 的代表作品之一。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，其执行效率与一般的编译器相比平均要高 20%～30%。

虽然我们称 Gcc 是 C 语言的编译器，但使用 Gcc 由 C 语言源代码文件生成可执行文件的过程不仅仅是编译的过程，而需要经历 4 个相互关联的步骤：预处理（也称预编译，preprocessing）、编译（compilation）、汇编（assembly）和连接（linking）。

命令 gcc 首先调用 cpp 进行预处理，在预处理过程中，对源代码文件中的文件包含（include）、预编译语句（如宏定义 define 等）进行分析。接着调用 cc1 进行编译，这个阶段

根据输入文件生成以.o 为后缀的目标文件。汇编过程是针对汇编语言的步骤调用 as 进行工作，一般来讲，以.s 为后缀的汇编语言文件经过预编译和汇编之后都生成以.o 为后缀的目标文件。当所有的目标文件都生成之后，gcc 就调用 ld 来完成最后的关键性工作，这个阶段就是连接。在连接阶段，所有的目标文件被安排在可执行程序中的恰当位置，同时，该程序所调用到的库函数也从各自所在的档案库中连接到合适的地方。

Gcc 编译器能将 C、C++语言源程序和目标程序编译、连接成可执行文件，如果没有给出可执行文件的名称，Gcc 将生成一个名为 a.out 的文件。在 Linux 系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。而 Gcc 则通过后缀来区别输入文件的类别，表 3-10 列出了最常见的扩展名和 Gcc 解释它们的方式。

表 3-10　Gcc 最常见的扩展名及其解释

扩展名	解释
.c	C 语言源代码文件
.C、.cc、.cxx	C++源代码文件
.m	Objective-C 源代码文件
.a、.so	已编译的库文件
.h	程序所包含的头文件
.i	已经预处理过的 C 源代码文件
.ii	已经预处理过的 C++源代码文件
.o	编译后的目标文件
.s	汇编语言源代码文件
.S	经过预编译的汇编语言源代码文件

3.4.2　Gcc 的基本用法和选项

在使用 Gcc 编译器时，需要给出一系列必要的调用参数和文件名称。Gcc 编译器的调用参数约有 100 多个，其中多数参数可能根本就用不到，这里只介绍最基本、最常用的参数。

Gcc 最基本的用法是：gcc [选项] [文件名]

常用的选项如下。

（1）-c：只编译，不连接成为可执行文件，编译器只是由输入的.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。

（2）-o output_filename：确定输出文件的名称为 output_filename，同时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。

（3）-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，必须加入这个选项。

（4）-E：预处理后即停止，不进行编译、汇编及连接。

（5）-O：对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连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、连接的速度就相应地要慢一些。

（6）-O2：比-O 更好地优化编译、连接，当然整个编译、连接的过程会更慢。

（7）-Idirname：将 dirname 所指的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。C 程序中的头文件包含以下两种情况。

A）#include <stdio.h>

B）#include "myinc.h"

其中，A 类使用尖括号，B 类使用双引号。对于 A 类，预处理程序 cpp 在系统预设包含文件目录（如/usr/include）中搜寻相应的文件；而对于 B 类，cpp 在当前目录中搜寻头文件，这个选项的作用是告诉 cpp，如果在当前目录中没有找到需要的文件，就到指定的 dirname 目录中去寻找。在程序设计中，如果需要的这种包含文件分别分布在不同的目录中，就需要逐个使用-I 选项给出搜索路径。

（8）-Ldirname：将 dirname 所指出的目录加入到程序函数档案库文件的目录列表中，是在连接过程中使用的参数。在预设状态下，连接程序 ld 在系统的预设路径（如/usr/lib）中寻找所需要的档案库文件，这个选项告诉连接程序，首先到-L 指定的目录中去寻找，然后到系统预设的路径中寻找，如果函数库存放在多个目录下，就需要依次使用这个选项，给出相应的存放目录。

（9）-lname：在连接时，装载名字为 libname.a 的函数库，该函数库位于系统预设的目录或者由-L 选项确定的目录下。例如，-lm 表示连接名为 libm.a 的数学函数库。

下面的例子是使用 gcc 从源文件 test.c 生成一个 test 程序。源代码如下：

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello, this is a test program.\n");
    return 0;
}
```

现在，要编译并运行该程序，在终端执行命令：

```
[root@localhost root]# gcc test.c -o test
```

如果一切顺利，gcc 会返回到 shell 提示符状态。它编译、连接源文件 test.c 并创建一个名为 test 的二进制代码。

使用下面的命令运行该程序并可以得到输出。

```
[root@localhost root]# ./test
Hello, this is a test program.
```

3.4.3　编译多个源文件

许多重要的程序都是由多个源代码文件组成的，执行最后的连接之前，各个源文件都必须编译成目标文件。为此，要向 Gcc 传递要编译的每个源代码的文件名。可以使用如下 gcc 命令来编译多个源文件：

```
[root@localhost root]# gcc file1.c file2.c file3.c -o program
```

Gcc 将创建 file1.o、file2.o 和 file3.o，然后把它们连接在一起创建 program 目标文件。除此之外，还可以先分别使用 gcc 的-c 选项为每个文件创建目标文件，然后再把目标文件连接在一起来创建可执行文件。因此，刚才的单条命令就变成如下几条命令：

```
[root@localhost root]# gcc -c file1.c
[root@localhost root]# gcc -c file2.c
[root@localhost root]# gcc -c file3.c
[root@localhost root]# gcc file1.o file2.o file3.o –o program
```

使用多条命令的好处是避免重新编译没有发生变化的文件。如果仅修改了 file1.c 中的源代码，则不需要重新编译 file2.c 和 file3.c 来重新创建 program。在连接源文件创建可执行文件之前，单独编译源文件还可以避免长时间运行编译程序。如果某个源代码模块实际很长，那么在单个 gcc 调用中编译多个文件就要花费一些时间。

下面的例子是使用多个源文件来创建单个二进制可执行文件。该示例程序包括两个 C 源代码文件 main.c 和 msg.c，以及一个头文件 msg.h，下面是各文件的源代码。

main.c 的源代码如下：

```
#include <stdio.h>
#include "msg.h"
int main(int argc, char ** grgv)
{
    char msg_hi[] = {"Hi, programmer!"};
    char msg_bye[] = {"Goodbye, Programmer!"};
    printf("%s\n",msg_hi);
    prmsg(msg_bye);
    return 0;
}
```

msg.h 的源代码如下：

```
#ifndef MSG_H_
#define MSG_H_

void prmsg(char *msg);

#endif
```

msg.c 的源代码如下：

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

void prmsg(char *msg)
{
    printf("%s\n",msg);

}
```

使用如下命令创建 helloprogram：

```
[root@localhost root]# gcc msg.c main.c -o helloprogram
```

要分别创建目标文件，可以使用以下命令：

```
[root@localhost root]# gcc -c msg.c
[root@localhost root]# gcc -c main.c
```

然后，要从目标文件创建 helloprogram，使用以下命令：

```
[root@localhost root]# gcc msg.o main.o -o helloprogram
```

运行该程序，输出如下：

```
[root@localhost root]# ./helloprogram
Hi, programmer!
Goodbye, programmer!
```

3.5　Gdb 调试器

Gdb 是 GNU 发布的一个 Linux 下的程序调试工具，它是一个强大的命令行调试工具。命令行的强大在于可以形成执行序列和脚本。命令行软件的优势在于它们可以非常容易地集成在一起，使用几个简单的已有工具的命令就可以做出一个非常强大的功能。Linux 下的软件全是命令行形式的，Gdb 调试器的使用给程序开发提供了极大的便利。

一般来说，Gdb 主要帮助完成以下 4 方面的功能：

（1）启动程序，可以按照自定义的要求随心所欲地运行程序。

（2）可以让被调试的程序在所指定的断点处停住。

（3）当程序被停住时，可以检查此时程序中所发生的事。

（4）动态地改变程序的执行环境。

3.5.1　启动 Gdb

一般来说，Gdb 主要调试的是 C、C++的程序。要调试 C、C++的程序，首先在编译时必须把调试信息加入到可执行文件中。使用编译器（cc、Gcc、g++）的-g 参数可以做到这一点，如：

```
$ cc -g test.c -o test
$ g++ -g test.cpp -o test
```

如果没有-g，将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当用-g 把调试信息加入并成功编译目标代码以后，下面介绍如何用 Gdb 来调试程序。

启动 Gdb 的方法有以下几种：

（1）gdb program。program 也就是执行文件，一般在当前目录下。

（2）gdb core。用 gdb 同时调试一个运行程序和 core 文件，core 是程序非法执行 core dump 后产生的文件。

（3）gdb pid。如果程序是一个服务程序，那么可以指定这个服务程序运行时的进程 ID，Gdb 会自动进行调试。

下面提供一个有问题的程序：gdbtst.c。

源程序 gdbtst.c 如下：

```
#include <stdio.h>
#define INDEX 200
int func(int n)
{
   int sum=0,i;
   for(i=0; i<=n;i++)
   {
      sum+=i;
   }
   return sum;
}
```

```
void index_array(int array[])
{
   int i;
   for(i=0;i<INDEX;i++)
      array[i]=i;
}

int main()
{
int i,intarray[100];
 long result = 0;
 for(i=1; i<=100; i++)
 {
    result += i;
 }
 index_array(intarray);
 printf("result[1-100] = %d \n", result);
 printf("result[1-250] = %d \n", func(250));
 return 0;
}
```

首先使用 gcc -g 命令编译程序，然后再执行该程序，输出如下：

```
[root@localhost root]# gcc -g gdbtst.c -o gdbtst
[root@localhost root]# ./gdbtst
result[1-100] = 5050
result[1-250] = 31375
Segmentation fault (core dumped)
```

如果程序有 bug，则需要启动 Gdb 进行调试：

```
[root@localhost root]# gdb gdbtst
```

Gdb 初始化后，出现如图 3-7 所示的界面。

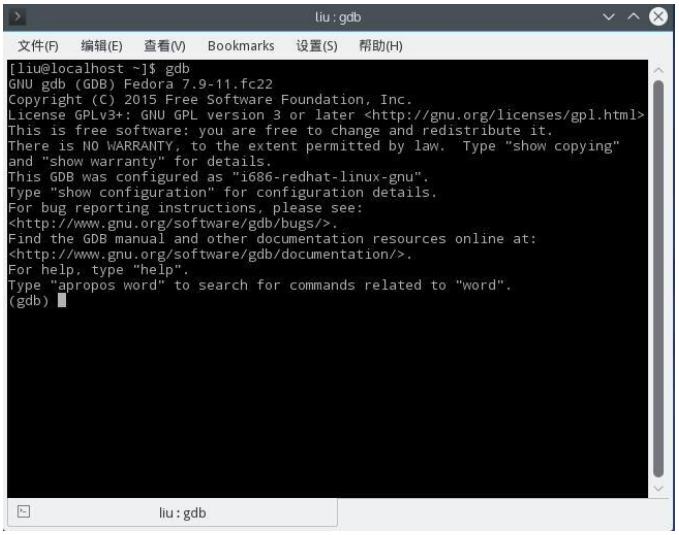

图 3-7　Gdb 初始化界面

Gdb 的命令很多，help 命令只是列出 Gdb 的命令种类，如果要查看种类中的命令，可以使用 help 命令，如 help breakpoints，查看设置断点的所有命令，如图 3-8 所示，也可以直接用 help 来查看命令的帮助。

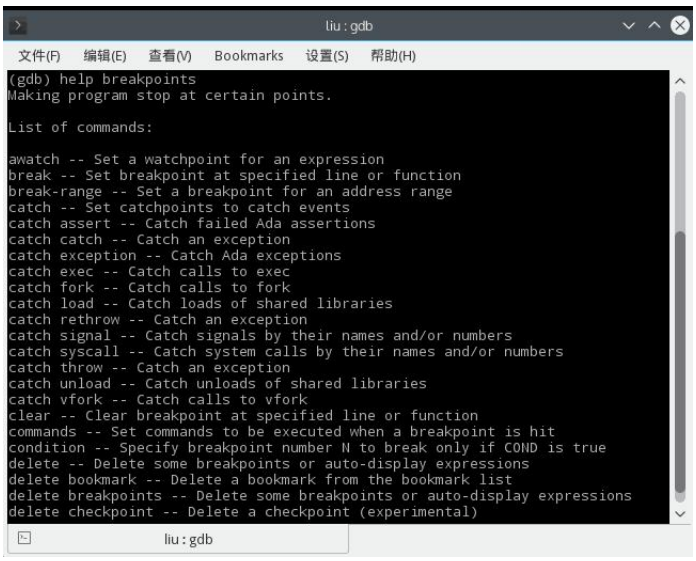

图 3-8　用 help 命令查看 breakpoints 的所有命令

在 Gdb 中输入命令时，可以不用打全命令，只用输入命令的前几个字符即可，但这几个字符必须能标识唯一的命令。在 Linux 下，也可以按两次 Tab 键来补齐命令的全称，如果有重复的，那么 Gdb 会将其列出来。

例如，输入 b，然后按两次 Tab 键，可以显示所有以 b 开头的命令：

```
(gdb) b
backtrace break bt
```

3.5.2　设置断点

调试有问题的代码，在某点上暂停执行通常很有用。设置断点，可以在执行流过程中查看给定点上的某些变量值，可以进行逐条代码执行，这对于查找程序的 bug 是十分有用的。

Gdb 中使用 break 命令来设置断点，该命令有如下 4 种形式。

（1）break line-number：使程序恰好在执行给定行之前停止。

（2）break function-name：使程序恰好在进入指定的函数之前停止。

（3）break line-or-function if condition：如果 condition（条件）是真，程序到达指定行或函数时停止。

（4）break routine-name：在指定例程的入口处设置断点。

如果该程序是由很多源文件构成的，可以在各个源文件中设置断点，而不是在当前的源文件中设置断点，方法如下：

```
(gdb) break filename:line-number
(gdb) break filename:function-name
```

要想设置一个条件断点，可以利用 break if 命令，方法如下：

```
(gdb) break line-or-function if expr
```

在前面的代码中，当变量 i 等于 50 时，用以下命令可以在 gdbtst 的第 29 行停止程序的执行：

```
(gdb) break 29 if i==50
Breakpoint 1 at 0x80483ba: file gdbtst.c, line 29.
(gdb) run
Starting program: /root/gdbtst

Breakpoint 1, main() at gdbtst.c:29
29 result += i;
(gdb) print i
$1 = 50
(gdb) print result
$2 = 1225
```

从以上代码可以看出，在第 29 行停止执行程序，用 print 检查变量 i 的输出值是 50，检查变量 result 此时的输出值是 1225。

要在达到断点后继续运行，可输入 continue 命令。如果要查看已经设置的断点，可使用 info breakpoints 命令。

3.6 Make 工程管理器

工程管理器是一个管理具有较多文件的项目的工具。工程管理器能够自动识别更新的文件代码，同时又不需要重复输入冗长的命令行。Make 工程管理器也是一个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入 Makefile 文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句即可。它大大提高了实际项目的工作效率，而且几乎所有 Linux 下的项目编程均会涉及到它。

Makefile 文件是 Make 工具程序的配置文件。Make 工具程序的主要用途是能自动地决定在一个含有很多源程序文件的大型程序中哪个文件需要被重新编译。为了使用 Make 程序，就需要 Makefile 文件来告诉 Make 要做些什么工作。通常，Makefile 文件会告诉 Make 如何编译和连接一个文件。当明确指出时，Makefile 还可以告诉 Make 运行何种命令（例如进行清理操作而删除某些文件）。

Make 的执行过程分为两个不同的阶段。在第一个阶段，它读取所有的 Makefile 文件以及包含 Makefile 的文件等，记录所有的变量及其值、隐式的或显式的规则，并构造出所有目标对象及其先决条件的一幅全景图。在第二个阶段，Make 就使用这些内部结构来确定哪个目标对象需要被重建，并且使用相应的规则来操作。

当 Make 重新编译程序时，每个修改过的 C 代码文件必须被重新编译。如果一个头文件被修改过了，那么为了确保正确，每一个包含该头文件的 C 代码程序都将被重新编译。每次编译操作都产生一个与源程序对应的目标文件（object file）。最终，如果任何源代码文件都被编译过了，那么所有的目标文件不管是刚编译完的还是以前就编译好的必须连接在一起以生成新的可执行文件。

Makefile 文件相当于程序编译过程中的批处理文件，是工具程序 Make 运行时的输入数据

文件。只要在含有 Makefile 的当前目录中键入 Make 命令，它就会依据 Makefile 文件中的设置对源程序或目标代码文件进行编译、连接或安装等活动。

Make 工具程序能自动地确定一个大程序系统中哪些程序文件需要被重新编译，并发出命令对这些程序文件进行编译。在使用 Make 之前，需要编写 Makefile 信息文件，该文件描述了整个程序包中各程序之间的关系，并针对每个需要更新的文件给出具体的控制命令。通常，执行程序是根据其目标文件进行更新的，而这些目标文件则是由编译程序创建的。一旦编写好一个合适的 Makefile 文件，那么在每次修改过程序系统中的某些源代码文件后，执行 Make 命令就能进行所有必要的重新编译工作。Make 程序是使用 Makefile 数据文件和代码文件的最后修改时间（Last-modification Time）来确定哪些文件需要进行更新的，对于每一个需要更新的文件它会根据 Makefile 中的信息发出相应的命令。在 Makefile 文件中，开头为#的行是注释行。文件开头部分的“=”赋值语句定义了一些参数或命令的缩写。

3.6.1　Makefile 的基本概念

1. Makefile 变量

Makefile 变量名不包括:、#、=前置空白和尾空白的任何字符串。同时，变量名中包含字母、数字以及下划线以外的情况应尽量避免，因为它们可能在将来被赋予特别的含义。变量名是大小写敏感的，如变量名 MAK、Mak 和 mak 代表不同的变量。

Makefile 中的变量均使用格式：$(VAR)。Makefile 变量分为用户自定义变量、预定义变量、自动变量和环境变量。自定义变量的值由用户自行设定，而预定义变量和自动变量通常为在 Makefile 中都会出现的变量，其中部分有默认值，也就是常见的设定值，当然用户可以对其进行修改。Make 在启动时会自动读取系统当前已经定义了的环境变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

表 3-11 和表 3-12 分别列出了 Makefile 中常用的自动变量和预定义变量。

表 3-11　Makefile 中常用的自动变量

命令	作用
$*	不包含扩展名的目标文件名称
$<	第一个依赖文件的名称
$+	所有的依赖文件
$?	所有时间戳比目标文件晚的依赖文件
$^	所有不重复的依赖文件
$%	如果目标是归档成员，则该变量表示目标的归档成员名称
$@	目标文件的完整名称

表 3-12　Makefile 中常用的预定义变量

命令	作用
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as

续表

命令	作用
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为$(CC)-E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm-f
ARFLAGS	库文件维护程序的选项
ASFLAGS	汇编程序的选项
CFLAGS	C 编译器的选项
CPPFLAGS	C 预编译器的选项
CXXFLAGS	C++编译器的选项
FFLAGS	FORTRAN 编译器的选项

2. Makefile 的基本结构

Makefile 是 Make 读入的唯一配置文件，在一个 Makefile 中通常包含如下内容：

（1）需要由 Make 工具创建的目标体（Target），通常是目标文件或可执行文件。

（2）要创建的目标体所依赖的文件（Dependency_file）。

（3）创建每个目标体时需要运行的命令（command）。

Makefile 的格式为：

```
target:dependency_file
command
```

例如，有两个文件 hello.c 和 hello.h，创建的目标体为 hello.o，执行的命令为 Ogcc -c hello.c，则对应的 Makefile 可写为：

```
hello.o: hello.c hello.h
gcc-c hello.c-o hello.o
```

接着就可以使用 Make 了。使用 Make 的格式为：make target，这样 Make 就会自动读入 Makefile，执行对应 Target 的 Command 语句，并会找到相应的依赖文件，代码如下：

```
[root@localhost makefile]# make hello.o
gcc -c hello.c -o hello.o
```

这样，Makefile 执行了 hello.o 对应的命令语句，并生成了 hello.o 目标体。

3. Makefile 的规则

Makefile 的规则是 Make 进行处理的依据，它包括了目标体、依赖文件及其之间的命令语句。一般地，Makefile 中的一条语句就是一个规则。在上面的例子中，显式地指出了 Makefile 中的规则关系，如$(CC) $(CFLAGS) -c $< -o $@，但为了简化 Makefile 的编写，Make 还定义了隐式规则和模式规则。

（1）隐式规则。

隐式规则能够告诉 Make 怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。Make 会自动搜索隐式规则目录来确

定如何生成目标文件。

如常见的隐式规则指出：所有.o 文件都可自动由.c 文件使用命令$(CC) $(CPPFLAGS) $(CFLAGS) -c file.c -o file.o 生成。这样 hello.o 就可以调用$(CC) $(CFLAGS) -c hello.c -o hello.o。

（2）模式规则。

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅只能够用 Make 默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化 Makefile 的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明，如%.o:%.c 表示所有的.o 文件由.c 文件生成：

```
%.o:%.c
$(CC) $(CFLAGS) -c $< -o $@
```

3.6.2　Make 管理器的使用

使用 Make 管理器非常简单，在 make 命令后键入目标名即可建立指定的目标：make target。此外 Make 还提供丰富的命令行选项，可以完成各种不同的功能。常用的 make 命令行选项如表 3-13 所示。

表 3-13　常用的 make 命令行选项

命令	作用
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
-i	忽略命令执行返回的出错信息
-s	沉默模式，在执行之前不输出相应的命令行信息
-n	非执行模式，输出所有执行命令，但并不执行
-t	更新目标文件
-p	输出所有宏定义和目标文件描述
-d	Debug 模式，输出有关文件和检测时间的详细信息

通过命令行选项中的 target，可指定 Make 要编译的目标，并且允许同时指定编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中的第一个目标文件。

3.7　使用 autotools

Linux 下工程管理器 Make 是可用于自动编译、连接程序的实用工具。现在要写一个 Makefile 文件，然后用 make 命令来编译、连接程序。Makefile 文件的作用就是让编译器知道要编译一个文件需要依赖其他的哪些文件。使用 autotools 系列工具可以完成系统配置信息的收集并自动生成 Makefile 文件，用户只需输入简单的目标文件、依赖文件、文件目录等。autotools 系列工具包括 aclocal、autoscan、autoconf、autoheader、automake 等 5 个工具。

autotools 的使用流程如下：

（1）手工编写 Makefile.am 文件。

（2）在源代码目录树的最高层运行 autoscan，然后手动修改 configure.scan 文件，并改名为 configure.ac 或 configure.in。

（3）运行 aclocal，它会根据 configure.ac 的内容生成 aclocal.m4 文件。

（4）运行 autoconf，它根据 configure.ac 和 aclocal.m4 的内容生成 configure 这个配置脚本文件。

（5）运行 automake --add-missing，它根据 Makefile.am 的内容生成 Makefile.in。

（6）运行 configure，它会根据 Makefile.in 的内容生成 Makefile 文件。

获得 Makefile 文件后，即可使用 make 程序来管理工程了。

有一个简单的工程，其目录和文件结构为：工程的最高层目录 tests 中有一个 hello.c 文件和 lib、include 两个子目录，在 lib 目录中有一个 print.c 文件，在 include 目录中有一个 print.h 文件。

首先，为该工程编写 automake 的输入配置脚本 Makefile.am。

其次，使用 autotools 工具为该工程创建 Makefile 文件，并编译该工程。这里共有 3 个目录，但只要在 tests 目录和 tests/lib 目录下分别创建 Makefile.am 文件，tests/include 不需要创建 Makefile.am 文件。文件的内容如下。

（1）hello.c 文件的内容：

```
#include "include/print.h"
int main(void)
{
        print("Hello, Beijing!\n");
        return 0;
}
```

（2）print.h 文件的内容：

```
void print(char *s);
```

（3）print.c 文件的内容：

```
#include "../include/print.h"
#include<stdio.h>
void print(char *string)
{
        printf("%s",string);
}
```

（4）tests 目录下的 Makefile.am 文件的内容：

```
SUBDIRS = lib
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = hello
hello_SOURCES = hello.c
hello_LDADD = ./lib/libprint.a
```

（5）lib 目录下的 Makefile.am 文件的内容：

```
noinst_LIBRARIES = libprint.a
libprint_a_SOURCES = print.c ../include/print.h
```

开始使用 aututools，步骤如下。

（1）执行 autoscan 命令，生成 configure.scan 文件，如图 3-9 所示。修改后的 configure.scan 文件的内容如图 3-10 所示，修改完后将文件重命名为 configure.ac 或 configure.in。

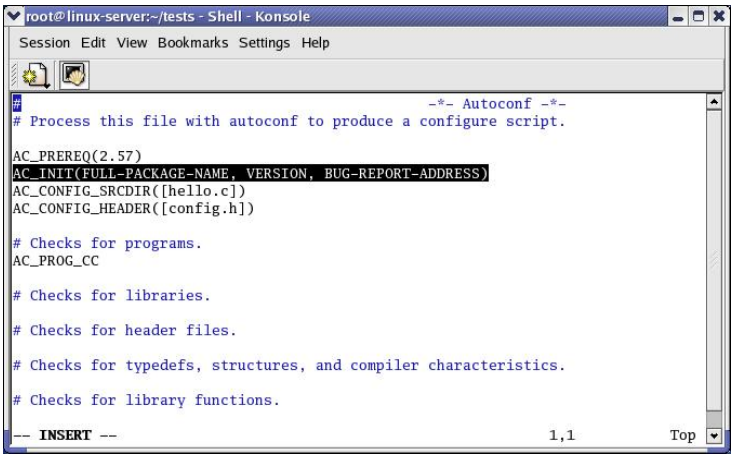

图 3-9　默认生成的 configure.scan 文件

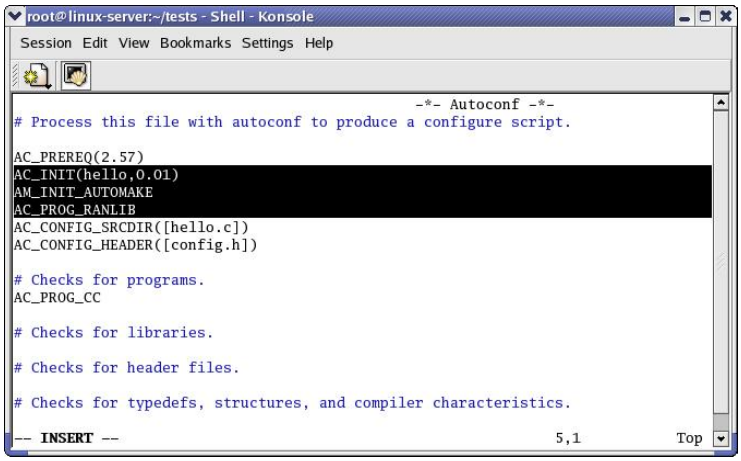

图 3-10　修改后的 configure.scan 文件

（2）先后执行 aclocal、autoconf、autoheader 命令。

（3）执行 automake --add-missing 命令，该步骤如果出现：

"Makefile.am:require file ./NEWS" not found"

"Makefile.am:require file "./README not found"

则运行 touch NEWS README * *（* *代表所缺失的文件，文件个数、名称因具体情况而定）。

（4）执行./configure 命令，即可生成 Makefile 文件。

（5）执行 make -f Makefile 命令编译文件，生成 hello 文件。

（6）输入./hello，运行结果如下：

```
[root@localhost tests]# ./hello
Hello, Beijing!
```

习题三

一、填空题

1．vi 编辑器有 3 种工作方式，即_____、_____和_____。

2．Makefile 变量分为_____变量、_____变量、_____变量和_____变量。

3．autotools 系列工具包括_____、_____、_____、_____和_____等 5 个工具。

二、简答题

1．简述 vi 的 3 种工作模式的转换方法。

2．什么是工程管理器？Makefile 的基本结构包括哪些？

3．Makefile 的模式规则有什么功能？

三、操作题

在 Linux 下使用 Gcc 编译器和 Gdb 调试器编写冒泡排序程序。