第四部分　调 试 工 具

第18章　程 序 调 试

将glib函数库、Gtk+构件库、Gnome库和GCC编译器结合起来可以用来开发非常复杂的应用程序，足以满足绝大多数的商业应用。但是这些还不足以成为一个完整的开发平台。还需要一个高效的调试器，特别是对较大型的应用程序，这一点更为重要。

Linux 包含了一个叫gdb的GNU调试程序。gdb 可以用来调试使C、C++以及Modula－2语言开发的程序；根据gdb维护者的计划，今后还将支持Fortran语言。gdb是一个强劲的调试器，提供了非常复杂的调试功能。它不仅能够用来调试 GUI应用程序，还可以用来调试非 GUI的程序、守护程序，甚至还可以将 gdb与正在运行的进程连接起来进行调试。可以用 gdb在程序运行时观察程序的内部结构和内存的使用情况。 gdb是基于字符的调试器；同时，还有一个图形界面的gdb版本，称为xxgdb。实际上， xxgdb是将gdb做了一个封装，并提供了一个图形接口，内部使用的还是gdb。

gdb是GNU项目的一部分，它是基于 GPL许可协议的。也就是说，只要遵从 GPL协议，就可以自由使用、修改、发布，且不需要为之付费。

下面是gdb和xxgdb 所提供的一些功能：

- 监视程序中变量的值。
- 设置断点以使程序在指定的代码行上停止执行。
- 让程序在指定条件下停止下来，检查程序的运行情况、表达式或变量的变化。
- 可以逐行执行程序代码。
- 运行中改变程序代码，可以直接体验修正 bug后的效果。

这里我们先介绍gdb，然后再介绍xxgdb。

18.1　用gdb调试应用程序

18.1.1　为调试程序做准备

一般大多数Linux的发布版本都包含了 gdb。安装时若选择“ 全部安装 ”或“ 安装为开发工作站 ”，就会安装gdb程序。在shell提示符下输入以下命令：

```
which gdb
```

如果安装了 gdb，将会返回 gdb的安装路径，一般是 /usr/bin/gdb，否则会什么也不显示。可以在Linux发布版本的光盘上找到 gdb的安装文件，一般是 gdb-4.18.rpm或者gdb-4.18.tar.gz。安装方法和普通的应用程序的安装方法一样，这里就不做介绍了。

要用gdb调试应用程序，当然首先得有应用程序。所以，要保证编写的应用程序没有语法错误，并且已经调试通过。同时，为了使 gdb正常工作，必须使程序在编译时包含调试信息。

调试信息包含了程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。

在编译时用 -g 选项打开调试选项。

18.1.2 获得gdb帮助

在命令行上输入gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话, 将启动 gdb, 可以在屏幕上看到类似的内容:

```
GNU gdb 4.18
Copyright 1998 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 "i586-pc-linux-gnu".
 (gdb)
```

启动gdb后, 可以在命令行上指定很多的选项。输入：

```
help
```

你可以获得 gdb的帮助信息。如果想要了解某个具体命令（比如 break）的帮助信息，在gdb提示符下输入下面的命令：

```
help break
```

屏幕上会显示关于break的帮助信息。从返回的信息得知， break是用于设置断点的命令。另一个获得gdb帮助的方法是浏览gdb的手册页。在Linux Shell提示符下输入：

```
man gdb
```

可以看到gdb的手册页。

18.1.3 gdb常用命令

还可以用下面的方式来运行gdb：

```
gdb filename
```

其中， filename是要调试的可执行文件。用这种方式运行 gdb时可以直接指定想要调试的程序。这和启动gdb后执行file filename命令效果完全一样。你也可以用 gdb去检查一个因程序异常终止而产生的core文件，或者与一个正在运行的程序相连。

gdb 支持很多的命令，还可以实现不同的功能。这些命令包含从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。下表列出了在使用 gdb进行调试时会用到的一些命令。想了解gdb的详细使用情况请参考 gdb的手册页。

file命令：装入想要调试的可执行文件。例如：

```
file  myapp
```

装入名为myapp的应用程序。要注意，必须指定要装入程序的完全路径，或者用 cd命令将工作目录转换到myapp所在目录上。

cd命令：改变工作目录。例如：

```
cd /root/Projects/myapp
```

将工作目录改变为/root/Projects/myapp。这和shell命令cd的作用相同。

pwd**命令：**返回当前工作目录。例如：

```
pwd
```

将返回当前的工作目录，比如/root/Projects/myapp。

run**命令：**执行当前被调试的程序。例如：

```
run
```

开始运行myapp程序。如果设置了断点，会在断点处挂起等待调试。有的应用程序在启动时可以指定选项，或者带参数运行。要调试这种情况，在 run后面加上选项参数表。

kill**命令：**停止正在调试的应用程序。

list**命令：**列出正在调试的应用程序的源代码。

break**命令：**设置断点。例如：

```
break 20
```

表示在第20行设置断点。程序运行到这一行时会挂起，并等待调试。一个应用程序可以设置多个断点。要注意的是，只能在可执行的代码行上设置断点，空行、注释等不可执行的行是不能设置断点的。如果应用程序由多个源程序组成，可以在不同文件中设置断点。例如：

```
break interface.c:30
```

在interface.c中的第30行设置断点。文件名和行号之间是一个冒号（:）。

tbreak**命令：**设置临时断点。它的语法与 break相同。区别在于用tbreak设置的断点执行一次之后立即消失。

watch**命令：**设置监视点，监视表达式的变化。例如：

```
watch mytitle
```

当mytitle（假定mytitle是一个变量）的值发生变化时，将应用程序挂起，并显示 mytitle的值。

awatch**命令：**设置读写监视点，当要监视的表达式被读或写时将应用程序挂起。它的语法与watch命令相同。

rwatch**命令：**设置读监视点，当监视表达式被读时将程序挂起，等待调试。此命令的语法与watch命令相同。

next**命令：**执行下一条源代码，但是不进入函数内部。也就是说，将一条函数调用作为一条语句执行。执行这个命令的前提是已经用 run开始了代码的执行。

step**命令：**执行下一条源代码，进入函数内部。如果调用了某个函数，会跳到函数所在的代码中，等候一步步执行。执行这个命令的前提是已经用 run开始执行代码。

display**命令：**在应用程序每次停止运行时打印表达式的值。

info break**命令：**显示当前断点列表，包括每个断点到达的次数。

info files**命令：**显示调试文件的信息。

info func**命令：**显示所有的函数名。

info local**命令：**显示当前函数的所有局部变量的信息。

info prog**命令：**显示调试程序的执行状态。

print**命令：**显示表达式的值。

delete**命令：**删除断点。指定一个断点号码，则删除指定断点。不指定参数则删除所有的断点。

shell命令：执行Linux Shell命令。例如：

```
shell ls
```

可以在不离开gdb的情况下，执行shell命令。

make命令：不退出gdb而重新编译生成可执行文件。

quit命令：退出gdb。

上面列出的命令只是gdb中最常用的一些命令。更多的命令可以在 gdb的手册页中找到。

gdb 支持很多与 Linux Shell 程序一样的命令编辑特征。可以像在 bash或tcsh里那样按Tab键让gdb补齐一个唯一的命令，如果不唯一， gdb会列出所有匹配的命令。你还可以用向上和向下的方向键上下翻动历史命令。

18.1.4 gdb 应用举例

下面用一个实例介绍怎样用 gdb调试程序。被调试的程序相当简单，但它展示了 gdb的典型应用。这也是一般介绍gdb时经常用到的一个例子。

下面列出了将被调试的程序。这个程序称为 greeting，它显示一个简单的问候，再用反序将它列出。

```
#include  <stdio.h>
main ()
{
  char my_string[] = "hello there";
  my_print (my_string);
  my_print2 (my_string);
}
void my_print (char *string)
{
  printf ("The string is %s\n", string);
}

void my_print2 (char *string)
{
  char *string2;
  int size, i;
  size = strlen (string);
  string2 = (char *) malloc (size + 1);
  for (i = 0; i < size; i++)
    string2[size - i] = string[i];
  string2[size+1] = `\0
  printf ("The string printed backward is %s\n", string2);
}
```

用gcc编译它：

```
gcc -o test test.c
```

程序执行时显示如下结果：

```
The string is hello there
The string printed backward is
```

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。期望的输出应该是：

```
The string printed backward is ereht olleh
```

毫无疑问，my_print2 函数没有正常工作。现在，让我们用 gdb看看问题究竟出在哪儿，先输入如下命令：

```
gdb greeting
```

如果输入命令时忘了把要调试的程序作为参数传给 gdb，可以在gdb提示符下用file命令加载它：

```
(gdb) file greeting
```

这个命令加载greeting可执行文件，就像在gdb命令行里加载它一样。

现在可以用gdb的run命令来运行greeting了。当它运行在gdb中时结果大约会像这样：

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样。可是，为什么反序打印没有工作呢？为了找出问题所在，我们可以在 my_print2函数的for语句后设一个断点。具体的做法是在 gdb提示符下执行三次list命令，列出源代码：

```
(gdb) list
(gdb) list
(gdb) list
```

每次执行list命令会列出10行代码。

第一次执行list命令的输出如下：

```
1       #include  <stdio.h>
2
3       main ()
4       {
5         char my_string[] = "hello there";
6
7         my_print (my_string);
8         my_print2 (my_string);
9       }
10
```

如果按下回车键，gdb 将再执行一次list命令，输出下列代码：

```
11      my_print (char *string)
12      {
13        printf ("The string is %s\n", string);
14      }
15
16      my_print2 (char *string)
17      {
18        char *string2;
19        int size, i;
20
```

再按一次回车将列出greeting 程序的剩余部分：

```
21        size = strlen (string);
```

```
22          string2 = (char *) malloc (size + 1);
23          for (i = 0; i < size; i++)
24            string2[size - i] = string[i];
25          string2[size+1] = '\0'
26          printf ("The string printed backward is %s\n", string2);
27 }
```

根据列出的源程序，可以看到应该将断点设在第 24行，在gdb命令行提示符下输入如下命令设置断点：

```
(gdb) break 24
```

gdb 将做出如下的响应：

```
Breakpoint 1 at 0x139: file greeting.c, line 24
(gdb)
```

现在再执行run命令，将产生如下的输出：

```
Starting program: /root/greeting
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24
24   string2[size-i]=string[i]
```

可以通过设置一个观察 string2[size - i]变量值的观察点来找出错误的产生原因，做法是键入如下语句：

```
(gdb) watch string2[size - i]
```

gdb将做出如下响应：

```
Watchpoint 2: string2[size - i]
```

现在可以用next命令来一步步的执行for循环了：

```
(gdb) next
```

经过第一次循环后，gdb告诉我们string2[size - i]的值是`h`。gdb显示如下信息：

```
Watchpoint 2, string2[size - i]
Old value = 0 '\000'
New value = 104 'h'
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23
23 for (i=0; i<size; i++)
```

这个值正是期望的。后来的数次循环的结果也都是正确的。当 i=10 时，表达式string2[size - i]的值等于`e`，size - i的值等于1，最后一个字符已经拷贝到新的字符串中了。

如果再把循环执行下去，会看到已经没有值分配给 string2[0]了，而它是新字符串的第一个字符，因为malloc函数在分配内存时把它们初始化为空 (null)字符。所以string2的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出。

找出了问题的所在，修正这个错误也就会变得很容易。可以把代码里写入 string2的第一个字符的偏移量改为 size - 1而不是size。这是因为string2的大小为12，但起始偏移量是0，串内的字符从偏移量0到偏移量10，偏移量11为空字符保留。

为了使代码正常工作有很多种修改办法。一种是另设一个比字符串的实际大小小 1的变量。下面是这种解决办法的代码：

```
#include  <stdio.h>
main ()
{
```

```
  char my_string[] = "hello there";
  my_print (my_string);
  my_print2 (my_string);
}

my_print (char *string)
{
  printf ("The string is %s\n", string);
}

my_print2 (char *string)
{
  char *string2;
  int size, size2, i;
  size = strlen (string);
  size2 = size -1;
  string2 = (char *) malloc (size + 1);

  for (i = 0; i < size; i++)
    string2[size2 - i] = string[i];
  string2[size] = `\0
  printf ("The string printed backward is %s\n", string2);
}
```

18.2 用xxgdb调试应用程序

xxgdb是一个X窗口下的调试器，它实际上是 gdb的一个封装。它的优点是不用记忆复杂的gdb命令，缺点是失去了 gdb的灵活性，且功能也不及 gdb强大。不过，普通应用程序使用xxgdb调试是非常方便的。

要知道是否安装了xxgdb，可在shell提示符下输入以下命令：

```
which xxgdb
```

如果已经安装了xxgdb，会返回xxgdb的路径，一般是/usr/X11R6/bin/xxgdb。

要想获得帮助，只需在shell提示符下输入如下命令：

```
man xxgdb
```

这样就可以获得xxgdb的手册页。你可以从中了解到 xxgdb的界面介绍，以及各个按钮的使用方法等。

在Linux Shell提示符下输入xxgdb，启动xxgdb，界面见图18-1。xxxgdb的界面主要由三个部分组成。最上面的部分是代码窗口，选中应用程序之后，应用程序的源代码将出现在该窗口中。中间部分是命令窗口，命令窗口中有一系列的按钮，对应于 xxgdb中的命令。最下面的部分是显示窗口，用于显示所执行的命令、执行结果和反馈信息。将鼠标放在分界线的黑色区域，可以调整各部分的相对大小。

启动xxgdb时，窗口标题是xxxgdb 1.12，而下面的显示窗口的提示却是 GDB 4.18，从这一点也能够看出xxxgdb实际上只是为gdb提供了一个图形界面。

xxgdb的命令窗口上的按钮根据功能可以大致分为以下几类：执行命令类，以各种方式运行应用程序；断点命令类，用于在程序中设置、取消断点；栈命令类，用于跟踪程序运行中

函数调用栈以及在栈之间移动；数据显示命令类，用于跟踪、显示表达式的值；其他命令类，比如加载文件、搜索、退出xxgdb等。

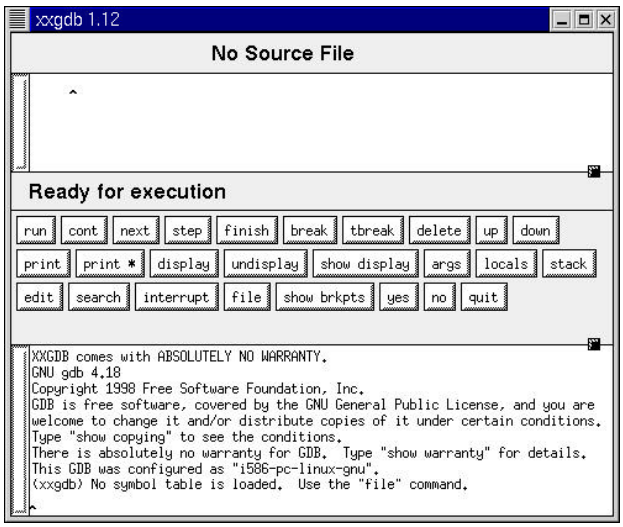

图18-1 用xxgdb调试程序

下面简要介绍xxgdb的几个重要调试功能。

1) 加载文件：点击file按钮，弹出一个选择文件对话框，如图 18-2所示：

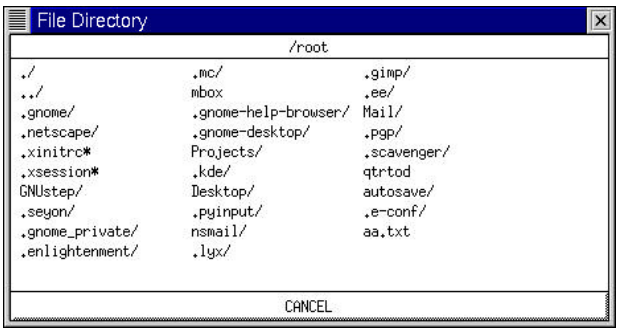

图18-2 在xxgdb中选择文件

单击对话框中的目录名，进入应用程序所在目录，然后选择要调试的应用程序的可执行文件。点击CANCEL可以取消此次操作。这时程序的源代码会显示在上部的代码窗口中。注意，如果应用程序有多个源文件，这里显示的是包含 main函数的文件。如果代码较长，代码窗口的左边会显示一个滚动条。用鼠标左键点击滚动条，代码向下滚动；用鼠标右键点击滚动条，代码向上滚动。

2) 加断点：在代码窗口中移动程序代码，找到要添加断点的程序行，在这一行的最前面单击鼠标左键，代码前面会出现一个“ ∧ ”符号，然后，点击 break按钮，这一行前面会出现一个红色的手形图片，表明这一行已经加上了断点。代码窗口和命令窗口之间会显示断点所在的文件以及行数。如果要设置临时断点（执行一次之后立即取消该断点），应该点击tbreak按钮。

如果程序由多个C源程序编译而成，想在其他文件，比如 interface.c中添加断点，需先将相应的文件加载进来。点击 file按钮，选择要跟踪的源文件，然后再在代码窗口中为它设置断点。

3) 显示所有的断点：点击 show brkpts按钮，下部的显示窗口中将显示所有的断点信息，包括断点号（根据设置的先后自动设置）、所在文件以及行号等。

4) 删除断点：在代码窗口选中断点，点击 delete按钮，可以删除该断点。

5) 运行应用程序：点击run按钮，立即运行程序。如果设置了断点，会在断点处挂起，等待发出其他调试命令。如果没有设置断点，将直接启动应用程序。

6) 继续运行：要从程序停止处继续运行，只需点击 cont按钮（cont实际是continue的简写）。

7) 单步执行：点击next，继续执行下一行代码，但是不进入函数中；点击 step按钮，继续下一行代码，如果遇到函数调用，会进入函数中。

8) 点击finish，会继续执行，直到程序返回。例如，正在某个函数中单步执行时，按下finish会立即执行函数直到函数返回。

9) 点击stack按钮会显示函数调用的堆栈，选择 up按钮移动调用栈的上一级，选择 down按钮移动调用栈的下一级。

10) 显示表达式的值。在代码中直到找到要监视的表达式，然后双击或按住鼠标左键拖过以选中它，鼠标左键点击 print按钮，将在显示窗口中显示该表达式的值。如果用鼠标右键点击print按钮，将在一个弹出的数据窗口中显示表达式的值。

11) 显示指针所指向的表达式的值。在代码窗口中选中要监视的指针，鼠标左键点击 print *按钮，将在显示窗口中显示该指针所指向的对象的信息。如果用鼠标右键点击 print *，将在一个弹出的数据窗口中显示这些信息。

12) 选中一个表达式，点击display按钮，会在显示窗口中显示表达式的值，且在程序每次挂起时更新一次。

13) 选择第12步中选中的表达式，点击undisplay，将取消上一步的工作。

14) 点击search按钮，会弹出一个查找对话框，在里面输入字符串，可以在程序代码中进行搜索。

15) 调试过程中， xxgdb可能会要求用户做出响应，比如对某种情况要求用户回答 y和n。点击yes按钮向调试器回答y，点击no按钮回答n。

16) 点击quit按钮将退出调试器。

上面介绍了xxgdb中的较重要的调试功能。用 xxgdb调试程序的步骤、方法和 gdb中的完全一样，这里就不重复介绍了。