CHhanker

Stay Hungry,Stay Foolish!

Linux学习--gdb调试

一. gdb常用命令：

命令	描述
backtrace（或bt）	查看各级函数调用及参数
finish	连续运行到当前函数返回为止，然后停下来等待命令
frame（或f） 帧编号	选择栈帧
info（或i） locals	查看当前栈帧局部变量的值
list（或l）	列出源代码，接着上次的位置往下列，每次列10行
list 行号	列出从第几行开始的源代码
list 函数名	列出某个函数的源代码
next（或n）	执行下一行语句
print（或p）	打印表达式的值，通过表达式可以修改变量的值或者调用函数
quit（或q）	退出gdb调试环境
set var	修改变量的值
start	开始执行程序，停在main函数第一行语句前面等待命令
step（或s）	执行下一行语句，如果有函数调用则进入到函数中

二.gdb学习小例：

```
#include <stdio.h>

int add_range(int low, int high)
{
        int i, sum;
        for (i = low; i <= high; i++)
                sum = sum + i;
        return sum;
}

int main(void)
{
        int result[100];
        result[0] = add_range(1, 10);
        result[1] = add_range(1, 100);
        printf("result[0]=%d\nresult[1]=%d\n", result[0], result[1]);
        return 0;
}
```

add_range函数从low加到high，在main函数中首先从1加到10，把结果保存下来，然后从1加到100，再把结果保存下来，最后打印的两个结果是：

```
result[0]=55
result[1]=5105
```

第一个结果正确[20]，第二个结果显然不正确，在小学我们就听说过高斯小时候的故事，从1加到100应该是5050。一段代码，第一次运行结果是对的，第二次运行却不对，这是很常见的一类错误现象，这种情况不应该怀疑代码而应该怀疑数据，因为第一次和第二次运行的都是同一段代码，如果代码是错的，那为什么第一次的结果能对呢？然而第一次和第二次运行时相关的数据却有可能不同，错误的数据会导致错误的结果。在动手调试之前，读者先试试只看代码能不能

昵称：hankers
园龄：3年9个月
粉丝：17
关注：3
+加关注

< 2012年12月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔

1. Linux学习--gdb调试
2. 解决单峰最值问题0.618法—hdu 4355
3. 一些数学公式
4. POJ 数学题目（转载）
5. 树形DP
6. 状态DP
7. 树形DP 状态DP
8. 贪心小练
9. 组合数学中的Polya定理
10. 图论题目分类

随笔分类(27)

Linux学习
PHP学习
Python学习
动态规划(7)
浪潮之颠
数学(8)
搜索(7)
算法小结(1)

看出错误原因，只要前面几章学得扎实就应该能看出来。

在编译时要加上-g选项，生成的可执行文件才能用gdb进行源码级调试：

```
$ gcc -g main.c -o main
$ gdb main
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

-g选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行，但并不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证gdb能找到源文件。gdb提供一个类似Shell的命令行环境，上面的(gdb)就是提示符，在这个提示符下输入help可以查看命令的类别：

```
(gdb) help
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.
```

也可以进一步查看某一类别中有哪些命令，例如查看files类别下有哪些命令可用：

```
(gdb) help files
Specifying and examining files.

List of commands:

add-shared-symbol-files -- Load the symbols from shared objects in the dynamic linker's link map
add-symbol-file -- Load symbols from FILE
add-symbol-file-from-memory -- Load the symbols out of memory from a dynamically loaded object file
cd -- Set working directory to DIR for debugger and program being debugged
core-file -- Use FILE as core dump for examining memory and registers
directory -- Add directory DIR to beginning of search path for source files
edit -- Edit specified file or function
exec-file -- Use FILE as program for getting contents of pure memory
file -- Use FILE as program to be debugged
forward-search -- Search for regular expression (see regex(3)) from last line listed
generate-core-file -- Save a core file with the current state of the debugged process
list -- List specified function or line
...
```

现在试试用list命令从第一行开始列出源代码：

```
(gdb) list 1
1       #include <stdio.h>
2
3       int add_range(int low, int high)
4       {
5               int i, sum;
```

贪心(2)
图论(2)
心路

随笔档案(28)

2012年12月 (1)
2012年8月 (9)
2012年7月 (1)
2012年3月 (1)
2012年2月 (9)
2012年1月 (7)

阅读排行榜

1. Linux学习--gdb调试(19233)
2. 组合数学中的Polya定理(919)
3. HDU 1025--LIS算法(764)
4. 记忆化搜索--HDU 1428(682)
5. 树形DP(301)

评论排行榜

1. 数学问题--POJ 2242(1)

推荐排行榜

1. Linux学习--gdb调试(3)
2. 组合数学中的Polya定理(1)
3. 数论总结---待续(1)

```
6               for (i = low; i <= high; i++)
7                       sum = sum + i;
8               return sum;
9       }
10
```

一次只列10行，如果要从第11行开始继续列源代码可以输入

```
(gdb) list
```

也可以什么都不输直接敲回车，gdb提供了一个很方便的功能，在提示符下直接敲回车表示重复上一条命令。

```
(gdb) （直接回车）
11      int main(void)
12      {
13              int result[100];
14              result[0] = add_range(1, 10);
15              result[1] = add_range(1, 100);
16              printf("result[0]=%d\nresult[1]=%d\n", result[0], result[1]);
17              return 0;
18
```

gdb的很多常用命令有简写形式，例如list命令可以写成l，要列一个函数的源代码也可以用函数名做参数：

```
(gdb) l add_range
1       #include <stdio.h>
2
3       int add_range(int low, int high)
4       {
5               int i, sum;
6               for (i = low; i <= high; i++)
7                       sum = sum + i;
8               return sum;
9       }
10
```

现在退出gdb的环境：

```
(gdb) quit
```

我们做一个实验，把源代码改名或移到别处再用gdb调试，这样就列不出源代码了：

```
$ mv main.c mian.c
$ gdb main
...
(gdb) l
5       main.c: No such file or directory.
        in main.c
```

可见gcc的-g选项并不是把源代码嵌入到可执行文件中的，在调试时也需要源文件。现在把源代码恢复原样，我们继续调试。首先用start命令开始执行程序：

```
$ gdb main
...
(gdb) start
Breakpoint 1 at 0x80483ad: file main.c, line 14.
Starting program: /home/akaedu/main
main () at main.c:14
14              result[0] = add_range(1, 10);
(gdb)
```

gdb停在main函数中变量定义之后的第一条语句处等待我们发命令，gdb列出的这条语句是即将执行的下一条语句。我们可以用next命令（简写为n）控制这些语句一条一条地执行：

```
(gdb) n
15              result[1] = add_range(1, 100);
(gdb) （直接回车）
16              printf("result[0]=%d\nresult[1]=%d\n", result[0], result[1]);
(gdb) （直接回车）
result[0]=55
result[1]=5105
17              return 0;
```

用n命令依次执行两行赋值语句和一行打印语句，在执行打印语句时结果立刻打出来了，然后停在return语句之前等待

我们发命令。虽然我们完全控制了程序的执行，但仍然看不出哪里错了，因为错误不在main函数中而在add_range函数中，现在用start命令重新来过，这次用step命令（简写为s）钻进add_range函数中去跟踪执行：

```
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Breakpoint 2 at 0x80483ad: file main.c, line 14.
Starting program: /home/akaedu/main
main () at main.c:14
14              result[0] = add_range(1, 10);
(gdb) s
add_range (low=1, high=10) at main.c:6
6               for (i = low; i <= high; i++)
```

这次停在了add_range函数中变量定义之后的第一条语句处。在函数中有几种查看状态的办法，backtrace命令（简写为bt）可以查看函数调用的栈帧：

```
(gdb) bt
#0  add_range (low=1, high=10) at main.c:6
#1  0x080483c1 in main () at main.c:14
```

可见当前的add_range函数是被main函数调用的，main传进来的参数是low=1, high=10。main函数的栈帧编号为1，add_range的栈帧编号为0。现在可以用info命令（简写为i）查看add_range函数局部变量的值：

```
(gdb) i locals
i = 0
sum = 0
```

如果想查看main函数当前局部变量的值也可以做到，先用frame命令（简写为f）选择1号栈帧然后再查看局部变量：

```
(gdb) f 1
#1  0x080483c1 in main () at main.c:14
14              result[0] = add_range(1, 10);
(gdb) i locals
result = {0, 0, 0, 0, 0, 0, 134513196, 225011984, -1208685768, -1081160480,
...
  -1208623680}
```

注意到result数组中有很多元素具有杂乱无章的值，我们知道未经初始化的局部变量具有不确定的值。到目前为止一切正常。用s或n往下走几步，然后用print命令（简写为p）打印出变量sum的值：

```
(gdb) s
7                       sum = sum + i;
(gdb) （直接回车）
6               for (i = low; i <= high; i++)
(gdb) （直接回车）
7                       sum = sum + i;
(gdb) （直接回车）
6               for (i = low; i <= high; i++)
(gdb) p sum
$1 = 3
```

第一次循环i是1，第二次循环i是2，加起来是3，没错。这里的$1表示gdb保存着这些中间结果，$后面的编号会自动增长，在命令中可以用$1、$2、$3等编号代替相应的值。由于我们本来就知道第一次调用的结果是正确的，再往下跟也没意义了，可以用finish命令让程序一直运行到从当前函数返回为止：

```
(gdb) finish
Run till exit from #0  add_range (low=1, high=10) at main.c:6
0x080483c1 in main () at main.c:14
14              result[0] = add_range(1, 10);
Value returned is $2 = 55
```

返回值是55，当前正准备执行赋值操作，用s命令赋值，然后查看result数组：

```
(gdb) s
15              result[1] = add_range(1, 100);
(gdb) p result
$3 = {55, 0, 0, 0, 0, 0, 134513196, 225011984, -1208685768, -1081160480,
...
  -1208623680}
```

第一个值55确实赋给了result数组的第0个元素。下面用s命令进入第二次add_range调用，进入之后首先查看参数和局

部变量：

```
(gdb) s
add_range (low=1, high=100) at main.c:6
6               for (i = low; i <= high; i++)
(gdb) bt
#0  add_range (low=1, high=100) at main.c:6
#1  0x080483db in main () at main.c:15
(gdb) i locals
i = 11
sum = 55
```

由于局部变量i和sum没初始化，所以具有不确定的值，又由于两次调用是挨着的，i和sum正好取了上次调用时的值，原来这跟例 3.7 “验证局部变量存储空间的分配和释放”是一样的道理，只不过我这次举的例子设法让局部变量sum在第一次调用时初值为0了。i的初值不是0倒没关系，在for循环中会赋值为0的，但sum如果初值不是0，累加得到的结果就错了。好了，我们已经找到错误原因，可以退出gdb修改源代码了。如果我们不想浪费这次调试机会，可以在gdb中马上把sum的初值改为0继续运行，看看这一处改了之后还有没有别的Bug：

```
(gdb) set var sum=0
(gdb) finish
Run till exit from #0  add_range (low=1, high=100) at main.c:6
0x080483db in main () at main.c:15
15              result[1] = add_range(1, 100);
Value returned is $4 = 5050
(gdb) n
16              printf("result[0]=%d\nresult[1]=%d\n", result[0], result[1]);
(gdb) （直接回车）
result[0]=55
result[1]=5050
17              return 0;
```

这样结果就对了。修改变量的值除了用set命令之外也可以用print命令，因为print命令后面跟的是表达式，而我们知道赋值和函数调用也都是表达式，所以也可以用print命令修改变量的值或者调用函数：

```
(gdb) p result[2]=33
$5 = 33
(gdb) p printf("result[2]=%d\n", result[2])
result[2]=33
$6 = 13
```


hankers
关注 - 3
粉丝 - 17
+加关注

« 上一篇：解决单峰最值问题0.618法—hdu 4355

posted @ 2012-12-07 11:11 hankers 阅读(19233) 评论(0) 编辑 收藏