GDB调试工具指南

王纯业
AnnCharles@tom.com

2003年08月18日

Contents

1 准备工作

在调试程序之前,现准备一个小程序,用于说明举例.
测试程序如下.

Listing 1: 测试代码test.cc

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  extern char **environ;
5  class SampleParentClass {
6    public:
7      virtual void PrintMe() {
8      printf("I am parent class.\n");
9      } virtual void PrintMe(const char *s) {
10     printf("I am parent class.\n%s\n", s);
11     }
12     class SampleParentClass *next;
13     const char *name;
14 };
15 class SampleChildClass:public SampleParentClass {
16     virtual void PrintMe() {
17     printf("I am child class.\nMy name is %s\n", name);
18     } virtual void PrintMe(const char *s) {
19     printf("I am child class.\nMy name is %s\n%s\n", name, s);
20     }
21 };
22 int foo(int a, int b)
23 {
24     int r = 0;
25     printf("a is %d,b is %d\n", a, b);
26     r = a + b;
27     return r;
28 }
29 int foo(int a)
30 {
31     printf("a is %d\n");
32     return a + 1;
33 }
34
35 int main(int argc, char **argv)
36 {
37     int a = 1;
38     int b = 3;
39     int i = 0;
40     char *p;
```

```
41      printf("Hello GDB\n");
42      printf("----------command line arguments-----\n");
43      for (i = 0, p = argv[i]; p != NULL; p = argv[++i]) {
44      printf("%d:\t%s\n", i, p);  //breakpoint
45      }
46      printf("----------enviroment----------\n");
47      for (i = 0, p = environ[i]; p != NULL; p = environ[++i]) {
48      printf("%d:\t%s\n", i, p);
49      }
50      // test overload function
51      foo(a, b);
52      foo(a);
53      // test class
54      SampleParentClass *obj;
55      obj = new SampleChildClass();
56      obj->name = "head";
57      obj->next = new SampleChildClass();
58      obj->next->name = "Charles";
59      obj->next->next = new SampleChildClass();
60      obj->next->next->name = "Smith";
61      obj->next->next->next = new SampleChildClass();
62      obj->next->next->next->name = "tail";
63      obj->next->next->next->next = NULL;
64      SampleParentClass *obj2;
65      for (obj2 = obj; obj2 != NULL; obj2 = obj2->next) {
66      obj2->PrintMe();     //breakpoint
67      obj2->PrintMe("I wanna say something.\n");
68      }
69      delete obj;
70      return 0;
71  }
```

编译这个程序

```
g++ -g -o t test.cc
```

-g的选项用于产生调试信息.

为什么要花精力学习一个调试工具.

因为我们知道开发过程中,在coding的阶段,写代码的时间要比调试代码的时间少得多. 有的时候,我们认为程序应该有这样的结果,可是偏偏程序的结果和我们相象的不一样, 我们通常花很长时间疑惑程序为什么没有按照我们预期的执行, 熟练掌握一个工具, 可以大大的提高我们的开发速度和效率.

2 运行程序

```
file <executable file name>
run
```

file 用于指定需要调试的可执行程序的文件名称,加载其调试信息. 也可以用gdb的命令行参数直接加载可执行文件.
run 这个命令用于启动程序.

2.1 设置命令行参数

有两种方法

- `set args <arg list>`
  用以下命令查看命令行参数.

  ```
  show args
  ```

2.2 设置环境变量

3 单步执行

- step 运行一步,会跳入函数里面.
- next 运行一行,不会跳入函数里面.
- finish 一直运行到函数返回.
- until 运行到某一行.

4 断点(breakpoint)

调试程序的概念就是要暂停程序的正常运行,查看当前的程序的状态. 断点就是程序的暂停的地方.

4.1 设置断点

有几种方法可以设置断点

- break ARGS
- tbreak ARGS
- hbreak ARGS
- thbreak ARGS
- rbreak REGEXP

先介绍break,这是最常用的. 其他方法类似,后面介绍他们和break不同点.
ARGS表示设置断点的地方. 有以下几种情况.

- 函数入口处暂停例如:

  ```
  break main
  ```

表示在main 函数开始的时候暂停运行.

如果因为重载(Overload),有多个函数有相同的名字,那么就不能用一个函数来标识一个断点位置了.

有两种情况,如果是重载类成员函数,如PrintMe, 那么会出来一个菜单,用户可以选择.

```
(gdb) break SampleChildClass::PrintMe
[0] cancel
[1] all
[2] SampleChildClass::PrintMe(char const*) at test.cc:25
[3] SampleChildClass::PrintMe() at test.cc:21
```

按0 , 表示不设置任何断点.

按1 , 表示设置所有断点.

按3 , 表示设置25处的断点.

按3 4, 表示设置3 和4 处的断点.

如果不是重载类成员函数, 也就是普通函数的重载, 可以通过指定不同入口参数类型来区分不同的重载函数,如:

```
break foo(int)
break foo(int,int)
```

- 在某个文件中的某一行

```
break test.cc:foo(int)
```

在test.cc中的foo函数的入口出停下来.

- 某一行

```
break 11
```

表示在当前源文件的第11行暂停. 什么是当前源文件? 当前源文件是指程序暂停处的所对应的源文件. 如果程序还没有起动,当前源文件是指含有main入口函数的源文件.

- 某源文件的某一行

```
break test.cc:11
```

在test.cc 中的第11行设为断点.

- 如果执行到某一行,在这一行的附近设置断

```
break +12
break -10
```

分别表示在当前行后12行处,和前10行处设置断点.

- 在某一个内存地址处暂停执行.

  ```
  break *0x80485ba
  ```

  内存地址必须是有效的代码段(Code Segment)的地址,不能是数据段(Data Segment),堆栈段(Stack Segment), 堆(Heap)的地址. 这种方法特别调试适合于没有调试信息的可执行文件.

tbreak 表示临时断点, 如果在这个断点暂停了,那么这个断点就被自动删除了.
hbreak 表示硬件辅助断点. 一般和具体硬件相关.
thbreak 表示硬件辅助的临时断点.是tbreak 和hbreak都复合.
rbreak 后面的参数是一个正则表达式, 凡是和该正则表达式相匹配的所有函数名称都设置成为断点.
正则表达式的语法和grep 命令的类似. 具体请参考grep命令. 正则表达式是Unix下的基本知识,很多程序都用到,如perl,vi,emacs,sed,awk,grep等等.
可以看到,设置断点时比较麻烦的,我写了一个小程序可以简化工作.在后面介绍.

4.2 查看断点

每次设置一个断点后,都会为每一个断点分配一个断点号,从1开始,依次递增.

```
info breakpoints
```

可以用用来查看设置的断点.

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080485a4 in foo(int) at test.cc:38
2   breakpoint     keep y   0x080485ca in main at test.cc:44
```

Num 表示断点号.
Type 表示断点类型.
Disp 表示断点的状态. del 表示断点暂停后(hit), 自动删除断点. keep 表示断点暂停后, 继续保持断点, dis 表示断点暂停后,关闭断点(disable).
Enb 表示断点是否起动(Enable). 程序运行到disable的断点,不会暂停. 可以用enable 命令起动断点. 用disable 命令关闭断点.
Address 和What 分别表示断点的代码地址和在源文件中的位置.
这个命令还会显示断点暂停的次数,每一次暂停的时候叫做一个hit. 还会显示断点的暂停的条件.

4.3 关闭和起动断点(Enable & Disable)

```
Enable <Breakpoint Number>
Disble <Breakpoint Numbe>
```

断点号可以有多个,用空格分隔.
还有进阶的命令

```
enable delete -- 起动断点,但是一旦在断点处暂停,就删除断点.
              用info breakpoints 命令查看断点时,该断点的Disp字段(field)是del
enable once -- 只起动一次断点,暂停后就关闭.
              用info breakpoints 命令查看断点时,该断点的Disp字段(field)是dis
```

4.4 条件断点

条件断点是指当程序运行到断点处, 只有在满足某些条件的时,才会暂停下来. 有以下方法设置断点的条件.

以上的例子中,设置断点,先运行程序,然后设置断点

- `break <ARGS> if <COND>`

  这种方法可以在设置断点的同时就设置断点的条件.

- `condition <Breakpoint Number> <COND>`

  这种办法用于对一个已知断点设置条件.

COND是任何合法的逻辑表达式. 什么是合法的逻辑表达式? 可以记住一条经验,在那个断点处,任何一个合法的C或C++ 逻辑表达式语句.也就是说,参数,局部变量,全局变量,甚至函数调用都可以. 但是注意如果逻辑表达式中含有函数调用, 那么要考虑到GDB 会调用这个函数,但是你的程序是不会调用的. 这样可能会有调试程序的时候和程序自己运行的时候,结果不一致.

4.5 删除断点

```
delete breakpoints <Breakpoint Number>
```

断点号可以有多个,用空格分隔.

如果不指定断点号,那么会删除所有断点.

5 查看变量

```
print /fmt <expr>
```

expr 是任何合法的表达式.

fmt 是格式定义,如下表.

x	十六进制格式
d	十进制格式
u	无符号整数
o	八进制格式
t	二进制格式
a	用十六进制格式打印地址,
c	字符格式
f	浮点格式

其中a 格式, 除了用十六进制打印内存地址外,同时在调试信息的符号表中查找一个和给定地址最近的调试已知符号,打印和这个符号的偏移量,

6 查看内存

```
x /<fmt> <ADDRESS>
```

Address 可以是任何合法的地址表达式,如

```
0x8799f000
p
&var
```

其中p 是在程序中是合法的指针,var是合法的变量.
fmt 是由三个部分组成, NFU,
N 表示查看的长度.
F 表示格式,和print 命令的格式是一样的.
U 表示单位. 可以是b h w g, 分别表示字节,半字(两个字节),字(四个字节),双字(8个字节)

7 查看源文件信息

```
info sources
```

8 查看当前源文件

9 如何查找源文件

一般情况下,可执行文件中含有调试信息,指定源代码中的位置,在哪个文件,哪一行上. 文件名称一般是相对路径.
GDB 通过source directory 来找到源文件的. 可以用

```
dir  NAME
```

添加一个查找路径.

```
show directories
```

查看souce directory.
dir 命令如果没有带参数,那么就清空source directory.

10 查看寄存器

11 关于frame

11.1 什么是frame

frame 是堆栈中的一段,表示一个函数调用.每次函数调用,传递参数,局部变量都保存在堆栈中, 在一个函数的可见范围内,叫做一个frame. 切换不同的frame,可以查看不同的局部变量和参数的值.

11.2 查看frame

```
backtrace or
bt
```

11.3 改变当前frame

```
frame <FRAME NO.>
up
down
```

例子

12 调试时改变变量

在程序暂停时,可以改变变量的值. 具体看例子.

13 调试时调用函数

具体看例子.

14 编写调试脚本

15 例子

有时觉得很多话都说不清楚,还不如例子来的清楚. 以下例子分别完成几个任务.

15.1 任务1,用调试程序打印命令行参数,和环境变量

```
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 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"...
(gdb) set args a b c d e f g                    (设置命令行参数)
(gdb) show args                                 (查看命令行参数)
Argument list to give program being debugged when it is started is "a b c d e f g ".
(gdb) unset env                                 (删除所有环境变量)
Delete all environment variables? (y or n) y
(gdb) set environment VAR1 = test               (设置两个环境变量)
(gdb) set environment VAR2 = var2 value
(gdb) show environment                          (显示两个环境变量)
VAR1=test
VAR2=var2 value
(gdb)break 51                                   (在51行处设断点)
Breakpoint 1 at 0x80486b8: file test.cc, line 51.
(gdb) list 51                                   (显示51行附近的源代码)
```

```
46          printf("-------------enviroment------------------\n");
47          for (i = 0, p = environ[i]; p != NULL; p = environ[++i]) {
48              printf("%d:\t%s\n", i, p);
49          }
50          // test overload function
51          foo(a, b);
52          foo(a);
53          // test class
54          SampleParentClass *obj;
55          obj = new SampleChildClass();
(gdb) run                                              (执行程序)
Starting program: /home/Charles/workplace/gdb_book/t a b c d e f g
Hello GDB                                              (程序输出)
-------------command line arguments-----
0:      /home/Charles/workplace/gdb_book/t
1:      a
2:      b
3:      c
4:      d
5:      e
6:      f
7:      g
-------------enviroment------------------
0:      VAR1=test
1:      PWD=/home/Charles/workplace/gdb_book
2:      VAR2=var2 value
3:      SHLVL=0

Breakpoint 1, main (argc=8, argv=0xbffffe94) at test.cc:51  (暂停了)
51          foo(a, b);
(gdb) set $i=0
(gdb) set variable p=argv[$i]                          (设置变量)
(gdb) while p!=0                                       (循环打印命令行参数)
 >print p=argv[$i++]
 >end
$1 = 0xbffffff61 "/home/Charles/workplace/gdb_book/t"
$2 = 0xbffffff84 "a"
$3 = 0xbffffff86 "b"
$4 = 0xbffffff88 "c"
$5 = 0xbffffff8a "d"
$6 = 0xbffffff8c "e"
$7 = 0xbffffff8e "f"
$8 = 0xbffffff90 "g"
$9 = 0x0
(gdb) print argv[$i]                                  (理解$i++的含义)
$20 = 0xbffffff92 "VAR1=test"
```

```
(gdb) set variable p=argv[$i]
(gdb) while p!=0                                    (继续打印环境变量)
 >print p=argv[$i++]
 >end
$21 = 0xbfffff92 "VAR1=test"
$22 = 0xbfffff9c "PWD=/home/Charles/workplace/gdb_book"
$23 = 0xbffffc1 "VAR2=var2 value"
$24 = 0xbffffd1 "SHLVL=0"
$25 = 0x0

(方法二,第一种方法改变了程序中p的值.下面的方法可以不用改变p的值.)
(gdb)  set $p=argv[0]
(gdb) set $i=0
(gdb) while $p!=0
 >print $p=argv[$i++]
 >end
$134 = 0xbffffff61 "/home/Charles/workplace/gdb_book/t"
$135 = 0xbffffff84 "a"
$136 = 0xbffffff86 "b"
$137 = 0xbffffff88 "c"
$138 = 0xbffffff8a "d"
$139 = 0xbffffff8c "e"
$140 = 0xbffffff8e "f"
$141 = 0xbffffff90 "g"
$142 = 0x0
(或者)
(gdb) set $i=0
(gdb) while argv[$i]!=0
 >print argv[$i++]
 >end
$226 = 0xbffffff61 "/home/Charles/workplace/gdb_book/t"
$227 = 0xbffffff84 "a"
$228 = 0xbffffff86 "b"
$229 = 0xbffffff88 "c"
$230 = 0xbffffff8a "d"
$231 = 0xbffffff8c "e"
$232 = 0xbffffff8e "f"
$233 = 0xbffffff90 "g"

(方法三,直接查看内存)
(gdb) x argv[0]
0xbffffff61:     0x6d6f682f
(gdb) x /10cb argv[0]     (以字节为单位(b),10个字节(10),用(c)字符格式显示)
0xbffffff61:     47 '/'  104 'h' 111 'o' 109 'm' 101 'e' 47 '/'  67 'C'  104 'h'
0xbffffff69:     97 'a'  114 'r'
(gdb) x /10tb                    (以二进制方式显示,调试编解码程序时有用)
```

```
0xbfffff6b:     01101100        01100101        01110011        00101111
01110111        01101111    01110010        01101011
0xbfffff73:     01110000        01101100
(gdb) x /10th     (以半字为单位)
0xbfffff7f:     0110101101101111        0111010000101111        0110000100000000        0110001000000
 0110001100000000        0110010000000000        0110010100000000        0110011000000000
0xbfffff8f:     0110011100000000        0101011000000000
(gdb)
```

15.2 任务2 打印链表

```
(gdb) break 64                          (设置断点)
Breakpoint 2 at 0x8048816: file test.cc, line 64.
(gdb) commands 2
(在刚刚设置的断点处设置命令,断点暂停后,会自动执行程序
断点号''2'', 可以省略)
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>silent                                (不打印调试信息)
>set $p=obj
>while $p!=0
 >print $p->name
 >set $p=$p->next
 >end
>cont                                   (继续执行.断点不停)
(这一点很有用,如果断点暂停会引起时间延迟,如果这个延迟对程序的运行结果
有很大的影响的话,会导致调试时的运行结果和直接运行时的结果不一样,用这个
方法,就可以最小化调试本身对程序的影响)
>end
a is 1,b is 3                           (程序的输出)
a is 1
$234 = 0x8048a2b "head"                 (断点暂停了, 断点命令的输出)
$235 = 0x8048a30 "Charles"
$236 = 0x8048a38 "Smith"
$237 = 0x8048a3e "tail"
I am child class.                        (程序继续执行)
My name is head
I am child class.
My name is head
I wanna say something.

I am child class.
My name is Charles
I am child class.
My name is Charles
I wanna say something.
```

```
I am child class.
My name is Smith
I am child class.
My name is Smith
I wanna say something.

I am child class.
My name is tail
I am child class.
My name is tail
I wanna say something.


Program exited normally.                    (程序结束)
```

15.3 任务3,堆栈

```
(gdb) del breakpoints                       (删除所有断点)
Delete all breakpoints? (y or n) y
(gdb) break foo(int)                 (在foo(int a)函数处设置断点,不是foo(int a,int b))
Breakpoint 3 at 0x80485a4: file test.cc, line 31.
(gdb) info breakpoints                      (显示所有断点)
Num Type           Disp Enb Address    What
3   breakpoint     keep y   0x080485a4 in foo(int) at test.cc:31
        breakpoint already hit 1 time
(gdb) run
Starting program: /home/Charles/workplace/gdb_book/t a b c d e f g
Hello GDB
-------------command line arguments-----
0:      /home/Charles/workplace/gdb_book/t
1:      a
2:      b
3:      c
4:      d
5:      e
6:      f
7:      g
-------------enviroment------------------
0:      VAR1=test
1:      PWD=/home/Charles/workplace/gdb_book
2:      VAR2=var2 value
3:      SHLVL=0
a is 1,b is 3

Breakpoint 3, foo(int) (a=1) at test.cc:31
```

```
31          printf("a is %d\n");
(gdb) bt                                (显示堆栈信息)
#0  foo(int) (a=1) at test.cc:31        (注意到但前执行到了test.cc的31行)
#1  0x080486d4 in main (argc=8, argv=0xbffffe94) at test.cc:52
#2  0x420158d4 in __libc_start_main () from /lib/i686/libc.so.6
(第一列是frame号码,0表示但前frame)
(gdb) info locals                       (查看所有局部变量)
No locals.
(gdb) info args                         (查看所有参数)
a = 1
(gdb) info frame                                    (查看当前frame)
Stack level 0, frame at 0xbffffdf8:                (frame level表示frame 号码)
 eip = 0x80485a4 in foo(int) (test.cc:31); saved eip 0x80486d4
 called by frame at 0xbffffe48
 source language c++.
 Arglist at 0xbffffdf8, args: a=1
 Locals at 0xbffffdf8, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffdf8, eip at 0xbffffdfc
(注意到第二行输出表示但前执行到了内存地址0xbffffdf8,在test.cc的31行处)
(gdb) up                                            (切换到上一个frame)
#1  0x080486d4 in main (argc=8, argv=0xbffffe94) at test.cc:52
52          foo(a);
(gdb) info frame
Stack level 1, frame at 0xbffffe48:
 eip = 0x80486d4 in main (test.cc:52); saved eip 0x420158d4
 called by frame at 0xbffffe68, caller of frame at 0xbffffdf8
 source language c++.
 Arglist at 0xbffffe48, args: argc=8, argv=0xbffffe94
 Locals at 0xbffffe48, Previous frame's sp is 0x0
 Saved registers:
  ebx at 0xbffffe44, ebp at 0xbffffe48, eip at 0xbffffe4c
(gdb) info locals
a = 1
b = 3
i = 4
p = 0x0
obj = (SampleParentClass *) 0x4212aa78
obj2 = (SampleParentClass *) 0x80484ea
(gdb) info args
argc = 8
argv = (char **) 0xbffffe94
(gdb) frame 0                           (选择最下面的frame)
#0  foo(int) (a=1) at test.cc:31
31          printf("a is %d\n");
(gdb) return 10                         (改变程序流程,直接返回10)
```

```
Make foo(int) return now? (y or n) y
#0  0x080486d4 in main (argc=8, argv=0xbffffe94) at test.cc:52
52          foo(a);
(gdb) call printf("this is test\0",foo(1,2))
(在程序暂停的时候,调用函数.)
a is 1,b is 2                              (foo(1,2)的输出)
$242 = 12                  (并没有真正输出,因为printf是缓存输出)
(gdb) call fflush(0)       (调用fflush()会真正输出)
this is test$243 = 0

(gdb)  disassemble                         (显示反汇编代码)
Dump of assembler code for function main:
0x80485ba <main>:       push   %ebp
0x80485bb <main+1>:     mov    %esp,%ebp
0x80485bd <main+3>:     push   %ebx
0x80485be <main+4>:     sub    $0x34,%esp
0x80485c1 <main+7>:     and    $0xfffffff0,%esp
0x80485c4 <main+10>:    mov    $0x0,%eax
0x80485c9 <main+15>:    sub    %eax,%esp
0x80485cb <main+17>:    movl   $0x1,0xfffffff8(%ebp)
0x80485d2 <main+24>:    movl   $0x3,0xfffffff4(%ebp)
0x80485d9 <main+31>:    movl   $0x0,0xfffffff0(%ebp)
0x80485e0 <main+38>:    sub    $0xc,%esp
0x80485e3 <main+41>:    push   $0x804899a
0x80485e8 <main+46>:    call   0x804849c <printf>
0x80485ed <main+51>:    add    $0x10,%esp
0x80485f0 <main+54>:    sub    $0xc,%esp
0x80485f3 <main+57>:    push   $0x80489c0
0x80485f8 <main+62>:    call   0x804849c <printf>
0x80485fd <main+67>:    add    $0x10,%esp
0x8048600 <main+70>:    movl   $0x0,0xfffffff0(%ebp)
0x8048607 <main+77>:    mov    0xfffffff0(%ebp),%eax
0x804860a <main+80>:    lea    0x0(,%eax,4),%edx
0x8048611 <main+87>:    mov    0xc(%ebp),%eax
0x8048614 <main+90>:    mov    (%eax,%edx,1),%eax
0x8048617 <main+93>:    mov    %eax,0xffffffec(%ebp)
0x804861a <main+96>:    cmpl   $0x0,0xffffffec(%ebp)
0x804861e <main+100>:   jne    0x8048622 <main+104>
0x8048620 <main+102>:   jmp    0x8048652 <main+152>
0x8048622 <main+104>:   sub    $0x4,%esp
0x8048625 <main+107>:   pushl  0xffffffec(%ebp)
0x8048628 <main+110>:   pushl  0xfffffff0(%ebp)
0x804862b <main+113>:   push   $0x80489ea
0x8048630 <main+118>:   call   0x804849c <printf>
0x8048635 <main+123>:   add    $0x10,%esp
0x8048638 <main+126>:   lea    0xfffffff0(%ebp),%eax
```

```
0x804863b <main+129>:   incl   (%eax)
0x804863d <main+131>:   mov    0xfffffff0(%ebp),%eax
0x8048640 <main+134>:   lea    0x0(,%eax,4),%edx
0x8048647 <main+141>:   mov    0xc(%ebp),%eax
---Type <return> to continue, or q <return> to quit---q
```

15.4 watchpoint 和display

watchpoint 是断点,意思是当某一个表达式发生变化时,程序暂停.
display 是自动打印变量. 和VC中的Watch窗口有的一比.
del display 和del watchpoint 分别是删除操作.
info display 和show display 分别是查看操作.

16 多线程调试

17 多进程调试