Linux 下 GDB 调试

本文写给主要工作在 Windows 操作系统下而又需要开发一些跨平台软件的程序员朋友，以及程序爱好者。

GDB 是一个由 GNU 开源组织发布的、UNIX/LINUX 操作系统下的、基于命令行的、功能强大的程序调试工具。

GDB 中的命令固然很多，但我们只需掌握其中十个左右的命令，就大致可以完成日常的基本的程序调试工作。

命令	解释	示例
file <文件名>	加载被调试的可执行程序文件。 因为一般都在被调试程序所在目录下执行 GDB，因而文本名不需要带路径。	(gdb) file gdb-sample
r	Run 的简写，运行被调试的程序。 如果此前没有下过断点，则执行完整个程序；如果有断点，则程序暂停在第一个可用断点处。	(gdb) r
c	Continue 的简写，继续执行被调试程序，直至下一个断点或程序结束。	(gdb) c
b <行号> b <函数名称> b *<函数名称> b *<代码地址> d [编号]	b: Breakpoint 的简写，设置断点。两可以使用“行号”“函数名称”“执行地址”等方式指定断点位置。 其中在函数名称前面加“*”符号表示将断点设置在“由编译器生成的 prolog 代码处”。如果不了解汇编，可以不予理会此用法。 d: Delete breakpoint 的简写，删除指定编号的某个断点，或删除所有断点。断点编号从 1 开始递增。	(gdb) b 8 (gdb) b main (gdb) b *main (gdb) b *0x804835c (gdb) d
s, n	s: 执行一行源程序代码，如果此行代码中有函数调用，则进入该函数； n: 执行一行源程序代码，此行代码中的函数调用也一并执行。 s 相当于其它调试器中的“Step Into (单步跟踪进入)”； n 相当于其它调试器中的“Step Over (单步跟踪)”。 这两个命令必须在有源代码调试信息的情况下才可以使用（GCC 编译时使用“-g”参数）。	(gdb) s (gdb) n
si, ni	si 命令类似于 s 命令，ni 命令类似于 n 命令。 所不同的是，这两个命令（si/ni）所针对的是汇编指令，而 s/n 针对的是源代码。	(gdb) si (gdb) ni
p <变量名称>	Print 的简写，显示指定变量（临时变量或全局变量）的值。	(gdb) p i (gdb) p nGlobalVar

display ... undisplay <编号>	display，设置程序中断后欲显示的数据及其格式。 例如，如果希望每次程序中断后可以看到即将被执行的下一条汇编指令，可以使用命令“display /i $pc” 其中 $pc 代表当前汇编指令，/i 表示以十六进行显示。当需要关心汇编代码时，此命令相当有用。 undispaly，取消先前的 display 设置，编号从 1 开始递增。	(gdb) display /i $pc (gdb) undisplay 1
i	Info 的简写，用于显示各类信息，详情请查阅“help i”。	(gdb) i r
q	Quit 的简写，退出 GDB 调试环境。	(gdb) q
help [命令名称]	GDB 帮助命令，提供对 GDB 名种命令的解释说明。 如果指定了“命令名称”参数，则显示该命令的详细说明；如果没有指定参数，则分类显示所有 GDB 命令，供用户进一步浏览和查询。	(gdb) help display

/add***********************************/**

j 命令　　回跳

ret　　设置返回值　　(例 ret 0/ret -1)

/add***********************************/**

废话不多说，下面开始实践。**gdb-sample.c**

```
1   #include <</font>stdio.h>
2
3   int nGlobalVar = 0;
4
5   int tempFunction(int a, int b)
6   {
7      printf("tempFunction is called, a = %d, b = %d \n", a, b);
8      return (a + b);
9   }
10
11  int main()
12  {
13     int n;
14        n = 1;
15        n++;
16        n--;
17
18        nGlobalVar += 100;
19        nGlobalVar -= 12;
20
```

```
21      printf("n = %d, nGlobalVar = %d \n", n, nGlobalVar);
22
23          n = tempFunction(1, 2);
24      printf("n = %d", n);
25
26      return 0;
27  }
28
```

```
gcc gdb-sample.c -o gdb-sample -g
```

使用参数 -g 表示将源代码信息编译到可执行文件中。如果不使用参数 -g，会给后面的 GDB 调试造成不便。当然，如果我们没有程序的源代码，自然也无从使用 -g 参数，调试/跟踪时也只能是汇编代码级别的调试/跟踪。

下面"gdb"命令启动 GDB，将首先显示 GDB 说明，不管它：

```
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 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)
```

上面最后一行"(gdb) "为 GDB 内部命令引导符，等待用户输入 GDB 命令。

下面使用"file"命令载入被调试程序 gdb-sample（这里的 gdb-sample 即前面 GCC 编译输出的可执行文件）：

```
(gdb) file gdb-sample
Reading symbols from gdb-sample...done.
```

上面最后一行提示已经加载成功。

下面使用"r"命令执行（Run）被调试文件，因为尚未设置任何断点，将直接执行到程序结束：

```
(gdb) r
Starting program: /home/liigo/temp/test_jmp/test_jmp/gdb-sample
n = 1, nGlobalVar = 88
tempFunction is called, a = 1, b = 2
n = 3
Program exited normally.
```

下面使用"b"命令在 main 函数开头设置一个断点（Breakpoint）：

```
(gdb) b main
Breakpoint 1 at 0x804835c: file gdb-sample.c, line 19.
```

上面最后一行提示已经成功设置断点，并给出了该断点信息：在源文件 gdb-sample.c 第 19 行处设置断点；这是本程序的第一个断点（序号为 1）；断点处的代码地址为 0x804835c（此值可

能仅在本次调试过程中有效）。回过头去看源代码，第 19 行中的代码为“n = 1”，恰好是 main 函数中的第一个可执行语句（前面的“int n;”为变量定义语句，并非可执行语句）。

再次使用“r”命令执行（Run）被调试程序：

```
(gdb) r
Starting program: /home/liigo/temp/gdb-sample

Breakpoint 1, main () at gdb-sample.c:19
19 n = 1;
```

程序中断在 gdb-sample.c 第 19 行处，即 main 函数是第一个可执行语句处。

上面最后一行信息为：下一条将要执行的源代码为“n = 1;”，它是源代码文件 gdb-sample.c 中的第 19 行。

下面使用“s”命令（Step）执行下一行代码（即第 19 行“n = 1;”）：

```
(gdb) s
20 n++;
```

上面的信息表示已经执行完“n = 1;”，并显示下一条要执行的代码为第 20 行的“n++;”。

既然已经执行了“n = 1;”，即给变量 n 赋值为 1，那我们用“p”命令（Print）看一下变量 n 的值是不是 1 ：

```
(gdb) p n
$1 = 1
```

果然是 1。（$1 大致是表示这是第一次使用“p”命令——再次执行“p n”将显示“$2 = 1”——此信息应该没有什么用处。）

下面我们分别在第 26 行、tempFunction 函数开头各设置一个断点（分别使用命令“b 26”“b tempFunction”）：

```
(gdb) b 26
Breakpoint 2 at 0x804837b: file gdb-sample.c, line 26.
(gdb) b tempFunction
Breakpoint 3 at 0x804832e: file gdb-sample.c, line 12.
```

使用“c”命令继续（Continue）执行被调试程序，程序将中断在第二 个断点（26 行），此时全局变量 nGlobalVar 的值应该是 88；再一次执行“c”命令，程序将中断于第三个断点（12 行，tempFunction 函数开头处），此时 tempFunction 函数的两个参数 a、b 的值应分别是 1 和 2：

```
(gdb) c
Continuing.

Breakpoint 2, main () at gdb-sample.c:26
26 printf("n = %d, nGlobalVar = %d \n", n, nGlobalVar);
(gdb) p nGlobalVar
$2 = 88
(gdb) c
Continuing.
n = 1, nGlobalVar = 88
```

```
Breakpoint 3, tempFunction (a=1, b=2) at gdb-sample.c:12
12 printf("tempFunction is called, a = %d, b = %d \n", a, b);
(gdb) p a
$3 = 1
(gdb) p b
$4 = 2
```

上面反馈的信息一切都在我们预料之中，哈哈~~~

再一次执行“c”命令（Continue），因为后面再也没有其它断点，程序将一直执行到结束：

```
(gdb) c
Continuing.
tempFunction is called, a = 1, b = 2
n = 3
Program exited normally.
```

有时候需要看到编译器生成的汇编代码，以进行汇编级的调试或跟踪，又该如何操作呢？

这就要用到 display 命令“display /i $pc”了（此命令前面已有详细解释）：

```
(gdb) display /i $pc
(gdb)
```

此后程序再中断时，就可以显示出汇编代码了：

```
(gdb) r
Starting program: /home/liigo/temp/test_jmp/test_jmp/gdb-sample

Breakpoint 1, main () at gdb-sample.c:19
19 n = 1;
1: x/i $pc 0x804835c : movl $0x1,0xfffffffc(%ebp)
```

看到了汇编代码，“n = 1;”对应的汇编代码是“movl $0x1,0xfffffffc(%ebp)”。

并且以后程序每次中断都将显示下一条汇编指定（“si”命令用于执行一条汇编代码—区别于“s”执行一行 C 代码）：

```
(gdb) si
20 n++;
1: x/i $pc 0x8048363 : lea 0xfffffffc(%ebp),%eax
(gdb) si
0x08048366 20 n++;
1: x/i $pc 0x8048366 : incl (%eax)
(gdb) si
21 n--;
1: x/i $pc 0x8048368 : lea 0xfffffffc(%ebp),%eax
(gdb) si
0x0804836b 21 n--;
1: x/i $pc 0x804836b : decl (%eax)
(gdb) si
23 nGlobalVar += 100;
1: x/i $pc 0x804836d : addl $0x64,0x80494fc
```

接下来我们试一下命令“b *<函数名称>”。

为了更简明，有必要先删除目前所有断点（使用“d”命令——Delete breakpoint）：

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb)
```

当被询问是否删除所有断点时，输入“y”并按回车键即可。

下面使用命令“b *main”在 main 函数的 prolog 代码处设置断点（prolog、epilog，分别表示编译器在每个函数的开头和结尾自行插入的代码）：

```
(gdb) b *main
Breakpoint 4 at 0x804834c: file gdb-sample.c, line 17.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/liigo/temp/test_jmp/test_jmp/gdb-sample

Breakpoint 4, main () at gdb-sample.c:17
17 {
1: x/i $pc 0x804834c : push %ebp
(gdb) si
0x0804834d 17 {
1: x/i $pc 0x804834d : mov %esp,%ebp
(gdb) si
0x0804834f in main () at gdb-sample.c:17
17 {
1: x/i $pc 0x804834f : sub $0x8,%esp
(gdb) si
0x08048352 17 {
1: x/i $pc 0x8048352 : and $0xfffffff0,%esp
(gdb) si
0x08048355 17 {
1: x/i $pc 0x8048355 : mov $0x0,%eax
(gdb) si
0x0804835a 17 {
1: x/i $pc 0x804835a : sub %eax,%esp
(gdb) si
19 n = 1;
1: x/i $pc 0x804835c : movl $0x1,0xfffffffc(%ebp)
```

此时可以使用“i r”命令显示寄存器中的当前值——“i r”即“Infomation Register”：

```
(gdb) i r
eax 0xbffff6a4 -1073744220
ecx 0x42015554 1107383636
```

```
edx 0x40016bc8 1073834952
ebx 0x42130a14 1108544020
esp 0xbffff6a0 0xbffff6a0
ebp 0xbffff6a8 0xbffff6a8
esi 0x40015360 1073828704
edi 0x80483f0 134513648
eip 0x8048366 0x8048366
eflags 0x386 902
cs 0x23 35
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x0 0
gs 0x33 51
```

当然也可以显示任意一个指定的寄存器值：

```
(gdb) i r eax
eax 0xbffff6a4 -1073744220
```

最后一个要介绍的命令是“q”，退出（Quit）GDB 调试环境：