GDB高级技巧：边Debug边修复BUG，无需修改代码，无需重新编译

原创 江南一散人 2020-06-08 23:29:06

友情提醒：本文介绍的调试技巧非常实用，但为了讲解清楚，篇幅较长，请耐心看完，我保证你定会有收获！

引言

程序调试时，你是否遇到过下面几种情况：

1、经过定位，终于找到了程序中的一个BUG，满心欢喜地以为找到了root cause，便迫不及待地修改源码，然后重新编译，重新部署。但验证时却发现，真正的问题并没有解决，代码中还隐藏着更多的问题。

2、调试时，我们找到代码中一个可疑的地方，但是不能100%确定这真的就是个BUG。要想确定，只能修改源码、重新编译、重新部署，然后重新运行验证。

3、已经找到了root cause，但不确定解决方案是否能正常工作，为了验证，不得不反复地修改代码、编译、部署。

对于大型项目，编译过程可能需要几十分钟，甚至几个小时，部署过程则更为复杂漫长！可想而知，如果调试过程中，不得不反复的修改源码，然后重新编译和部署，会是一项多么繁琐和浪费时间的事情！

那么，有没有一种更高效的调试手段，可以避免反复修改代码和编译呢?

当然有！本文将介绍一种GDB调试技巧，可以一边调试，一边修复Bug，可以在不修改代码、不重新编译的前提下即可修复BUG，验证我们的解决方案，大幅提高调试效率！

本文预期效果

如下图，冒泡排序程序中，有三个BUG：

```c
#include <stdio.h>

void bubble_sort(int a[], int n) /* 冒泡排序 */
{
    int i = 0, j = 0, tmp;

    for (i = 0; i <= n; i++) { /* BUG: 数组访问越界 */
        for (j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
```

```
14          }
15      }
16  }
17
18  int main(int argc, char *argv[])
19  {
20      int arr[10] = {64, 34, 25, 12, 22, 11, 90, 45, 86, 23};
21
22      bubble_sort(arr, sizeof(arr)); /* BUG: 取数组元素个数不能用sizeof */
23      for (int i = 0; i <= sizeof(arr); i++) { /* BUG: 数组访问越界 */
24          printf("%d ", arr[i]);
25      }
26      printf("\n");
27      return 0;
28  }
```


冒泡排序示例

图中已经把三个BUG都标注了出来。正常编译运行时，程序执行结果如下：

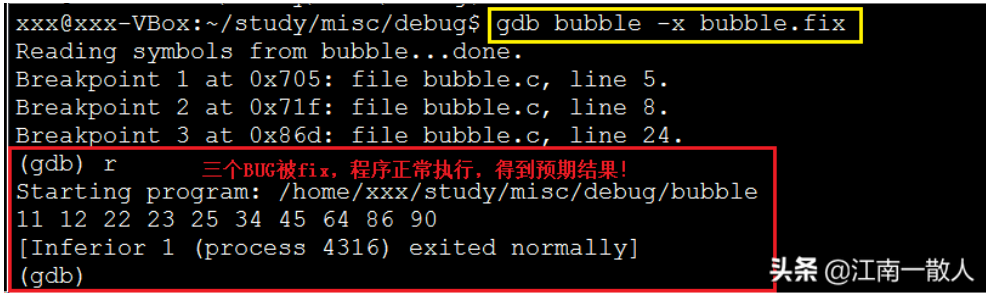
```
xxx@xxx-VBox:~/study/misc/debug$ ./bubble
-1432088366 -1410442298 -1332135194 -1315879440 -1315878929 -13158
78720 -1215874330 -478594330 -109326200 0 0 0 0 0 0 0 1 11 12 22
 23 25 32 34 45 64 86 90 22035 22035 22035 32601 32764 32764 33163
8784 871861023 1371300352 1371300360 1949715351 0
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)          普通方式执行，程序异常终止
xxx@xxx-VBox:~/study/misc/debug$
xxx@xxx-VBox:~/study/misc/debug$ gdb bubble
Reading symbols from bubble...done.
(gdb) r                        GDB中调试执行，仍然异常终止
Starting program: /home/xxx/study/misc/debug/bubble
-1772190383 -1772186130 -1022639941 -140485737 -7232 -7224 0 0 0 0
 0 0 0 0 1 11 12 22 23 25 32 34 45 64 86 90 21845 21845 21845 3276
7 32767 32767 343434853 392817664 414187661 862348901 1075340901 1
431651824 1431652335 1431652544 0   无法得到预期结果
*** stack smashing detected ***: <unknown> terminated

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:51
51      ../sysdeps/unix/sysv/linux/raise.c: No such file or direct
ory.
(gdb)
```

程序执行异常

不过是普通方式执行，还是在GDB中执行，程序都异常终止，无法得到正常结果。

但是，利用本文介绍的调试技巧，可以利用GDB给这个程序制作一个“热补丁”，在不修改代码、不重新编译的前提下，解决掉程序中的三个BUG，让程序正常执行，并得到预期结果！

最终效果，如下图所示：



打上“热补丁”后，程序正常执行

是不是很有趣呢？下面开始介绍！

GDB Breakpoint Command Lists

GDB支持断点触发后，自动执行用户预设的一组调试命令。使用方法：

```
commands [bp_id...]
  command-list
end
```

其中：

- commands是GDB内置关键字
- bp_id是断点的ID，也就是info命令显示出来的断点Num，可以指定多个，也可以不指定。当不指定时，默认只对最近一次设置的那个断点有效。

- command-list是用户预设的一组命令，当bp_id指定的断点被触发时，GDB会自动执行这些命令。
- end表示结束。

这个功能适用于各种类型的断点，如breakpoint、watchpoint、catchpoint等。

适用场景举例

利用GDB breakpoint commands lists这个特性可以做很多有趣的事情，本文仅列举其中的几个。

1、随时随地printf，不需修改代码和重新编译

看过我之前文章的朋友，应该还记得，我介绍过GDB的动态打印(Dynamic Printf)功能，可以用dprintf命令在代码的任意地方添加动态打印断点，并自动执行格式化打印操作，从而无需修改代码和重新编译就可以在代码中任意增加日志打印信息。

利用GDB breakpoint commands lists功能，可以实现一样的功能，而且除了打印之外，还可以做其它更多的操作，比如dump内存，dump寄存器等。

2、修改代码执行逻辑，避免修改代码和重新编译

在GDB中可以做很多有趣的事情，比如修改变量、修改寄存器、调用函数等，结合breakpoint command list功能，可以在调试的同时，修改程序执行逻辑，给程序打上“热补丁”。从而可以在调试过程中，快速修复Bug，避免重新修改代码和重新编译，大大提高程序调试的效率！

这是本文重点讲解的场景，稍后会演示如何利用这个功能，在GDB调试的过程中修复掉上文冒泡排序程序中的三个Bug。

3、实现自动化调试，提高调试效率

这个功能，结合GDB支持的脚本功能，以及自定义命令功能，可以实现调试自动化。

这涉及到GDB的很多其它知识，篇幅有限，不再展开讨论，以后更新专门文章讲解！感兴趣的童鞋，不妨右上角关注一下！

给冒泡排序打上“热补丁”

现在，我们利用GDB breakpoint command lists功能，给文中的冒泡排序程序打上“热补丁”，演示如何在不修改源码、不重新编译的前提下，解决掉程序中的3个BUG。

再看一下示例程序：

```
1  #include <stdio.h>
2
3  void bubble_sort(int a[], int n) /* 冒泡排序 */
4  {
5      int i = 0, j = 0, tmp;
6
7      for (i = 0; i <= n; i++) { /* BUG: 数组访问越界 */
8          for (j = 0; j < n - i - 1; j++) {
9              if (a[j] > a[j + 1]) {
10                 tmp = a[j];
11                 a[j] = a[j + 1];
12                 a[j + 1] = tmp;
```

```
13              }
14          }
15      }
16  }
17
18  int main(int argc, char *argv[])
19  {
20      int arr[10] = {64, 34, 25, 12, 22, 11, 90, 45, 86, 23};
21
22      bubble_sort(arr, sizeof(arr)); /* BUG: 取数组元素个数不能用sizeof */
23      for (int i = 0; i <= sizeof(arr); i++) { /* BUG: 数组访问越界 */
24          printf("%d ", arr[i]);
25      }
26      printf("\n");
27      return 0;
28  }
```


编译一下：

```
gcc -g bubble.c -o bubble
```

先用GDB加载运行一下：

```
xxx@host:~/study/misc/debug$ gcc -g bubble.c -o bubble
xxx@host:~/study/misc/debug$
xxx@host:~/study/misc/debug$ gdb bubble
Reading symbols from bubble...
(gdb) r
Starting program: /home/xxx/study/misc/debug/bubble
-136410261 -49022871 -7424 -7416 0 0 0 0 0 0 0 0 1 11 12 22 23 25 34 45 64
 86 90 21845 21845 21845 32767 32767 32767 524288 80177199 80181357 792066
153 1293285481 1368534328 1431654512 1431654986 1431655200 1441432744 1618
662144 0
*** stack smashing detected ***: <unknown> terminated

Program received signal SIGABRT, Aborted.
0x00007ffff7e05ed7 in raise () from /lib/x86_64-linux-gnu/libc.so.6
(gdb)
```


程序运行异常，符合我们的预期。

下面我们依次解决冒泡排序程序中的3个BUG。

1、解决第一个BUG

先解决第22行的BUG，也就是传递给了bubble_sort()错误的数组长度。

我们知道，在x64上，函数参数优先采用寄存器传递。那么，我们有这么几种方式可以选择：

1. 把断点设置在bubble_sort()入口第一条指令，然后直接修改存放数组长度n的那个寄存器中的值。
2. 把断点设置在bubble_sort()入口处(不必是第一条指令)，在第7行for循环之前，把存放数组长度的变量n的值改掉。
3. 把断点设置在main()函数第22行，也就是调用bubble_sort()的地方，然后以正确的参数手动调用bubble_sort()函数，并利用GDB的jump命令，跳过第22行代码的执行。

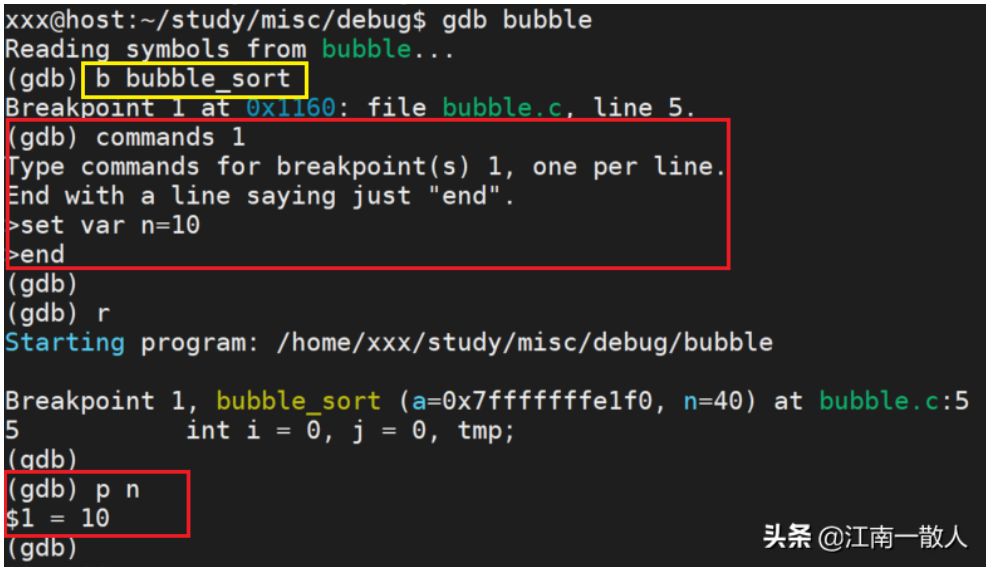
考虑到有些童鞋对x64 CPU不是非常了解，或者对GDB的jump命令不熟悉，我们采用第2种方

式。而且，这种方式也更简单通用。

我们先给bubble_sort()函数设置断点，然后利用commands命令预设一条命令，把变量n的值修改为10。命令如下：

```
b bubble_sort
commands 1
  set var n=10
end
```

设置完之后，用run命令开始运行程序。结果如下：



bubble_sort()处的断点被触发后，程序暂停，用print命令查看变量n的值，已经被修改成了正确的值：10。

可见，我们的设置是有效的。

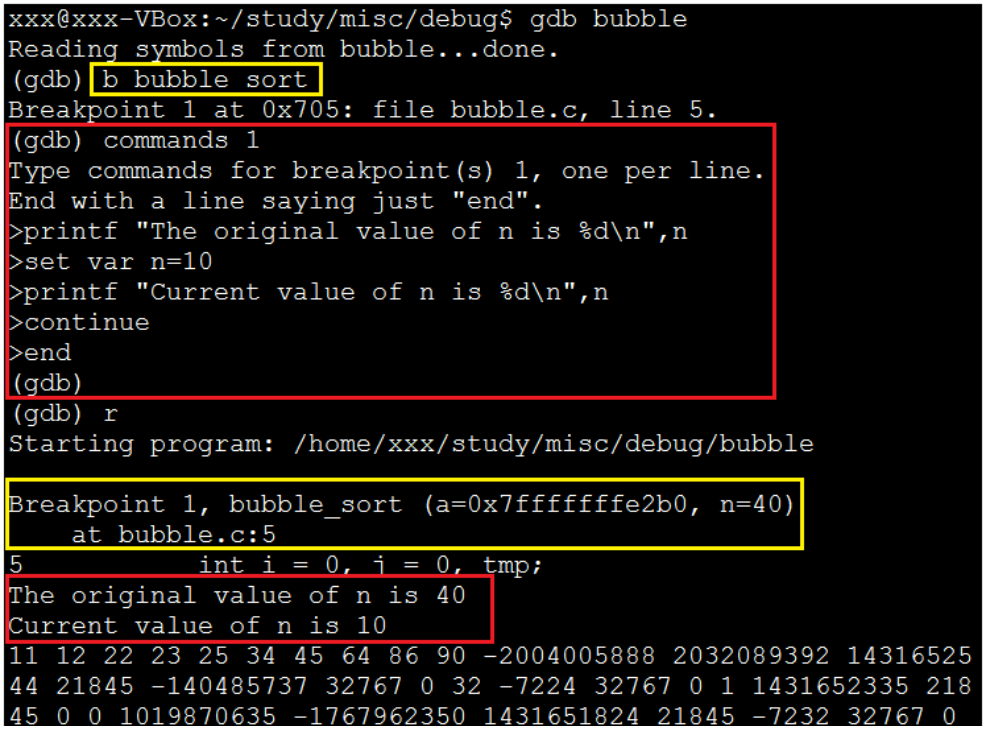
断点触发后，让程序自动恢复执行

那么，在bubble_sort()处断点被触发，变量n的值被修改之后，如何让程序自动恢复执行呢?

很简单，只需要在预设的命令中添加一个continue命令就可以了。为了证明我们的设置确实是生效的，我们在修改变量n的前后，各添加一个格式化打印语句，把变量n的值打印出来:

```
b bubble_sort
commands 1
  printf "The original value of n is %d\n",n
  set var n=10
  printf "Current value of n is %d\n",n
  continue
end
```

结果如下图:



```
0 0 0 1755970987 -1010148281 467140011 -1010152200 0
[Inferior 1 (process 3167) exited normally]
(gdb)
```


解决第一个BUG

从运行结果可以看出，断点被触发后，我们预设的语句被正确执行，变量n的值被修改为10，然后程序自动恢复执行。

到此，第一个BUG已经解决了。

2、解决第二个BUG

下面，我们解决第7行代码中的数组访问越界错误：数组的元素个数是n，但是bubble_sort()中第一个for循环的终止条件是i<=n，明显会造成访问越界，正确的条件应该是i<n。

要解决这个BUG也很简单，只需要在执行第8行代码之前，判断如果i的值等于n，就跳出循环。对于这个简单的程序，我们直接从bubble_sort()函数return就可以了。

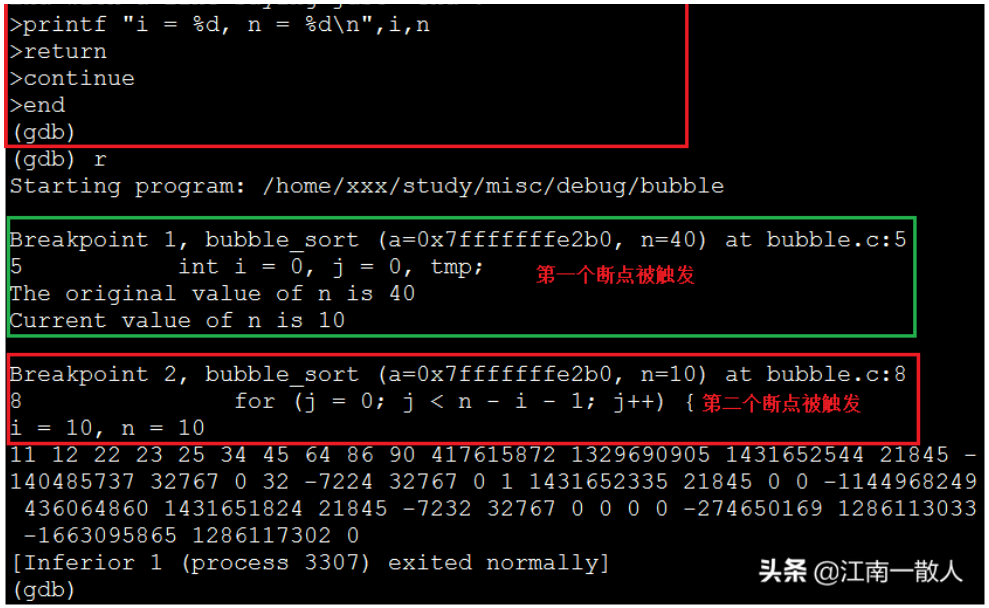
命令如下：

```
b 8 if i==n
command 2
  printf "i = %d, n = %d\n",i,n
  return
  continue
end
```

在第8行设置条件断点，当i==n时断点被触发，然后自动把i和n的值打印出来，再行return命令，从bubble_sort()返回，然后continue命令自动恢复程序执行。

执行结果如下图：

```
(gdb) b 8 if i==n
Breakpoint 2 at 0x55555555471f: file bubble.c, line 8.
(gdb) commands 2
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
```



解决第二个BUG

3、解决第三个BUG

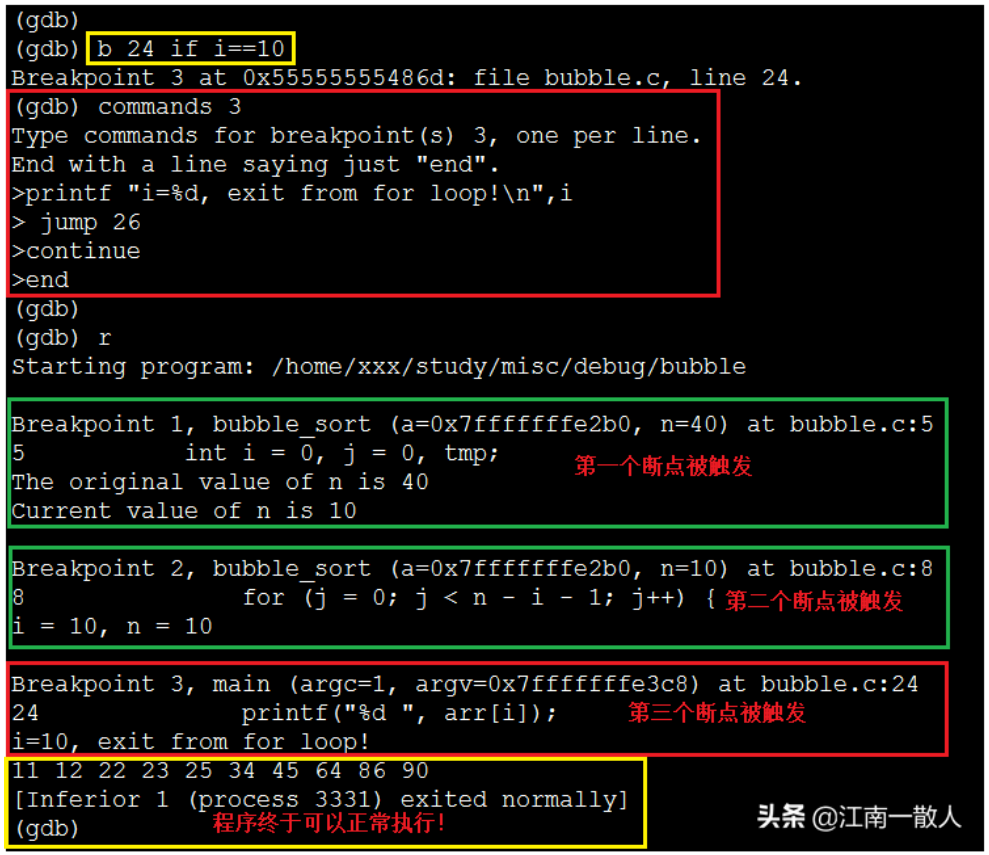
下面，解决最后一个BUG，第23行数组访问越界错误。

命令如下：

```
b 24 if i==10
commands 3
  printf "i=%d, exit from for loop!\n",i
  jump 26
  continue
end
```

与第二个BUG类似，在第24行设置条件断点，当==10时触发断点，然后退出循环，让程序跳转到第26行继续执行。

执行结果如下图所示：



解决第三个BUG

从图中可以看出，三个断点全部被触发，并且预设的命令都正常执行。

我们终于得到了正确的执行结果！

虽然，现在程序可以正常执行了，但是每次手动输入命令还是比较麻烦的。我之前文章介绍过，GDB支持调试脚本，从脚本中加载并执行调试命令。

下面，我们利用GDB脚本，来制作我们的“热补丁”脚本。

制作“热补丁”脚本

我们把上文中用来解决三个BUG的命令保存在一个脚本文件中：

```
vi bubble.fix
```

脚本内容如下图：

```
1   b bubble_sort
2   commands
3     set var n=10
4     continue
5   end
6
7   b 8 if i==n
8   commands
9     return
10    continue
11  end
12
13  b 24 if i==10
14  commands
15    jump 26
16    continue
17  end
```


bubble.fix 热补丁脚本

bubble.fix脚本中的命令，与上文在GDB中直接输入的命令有几个区别：

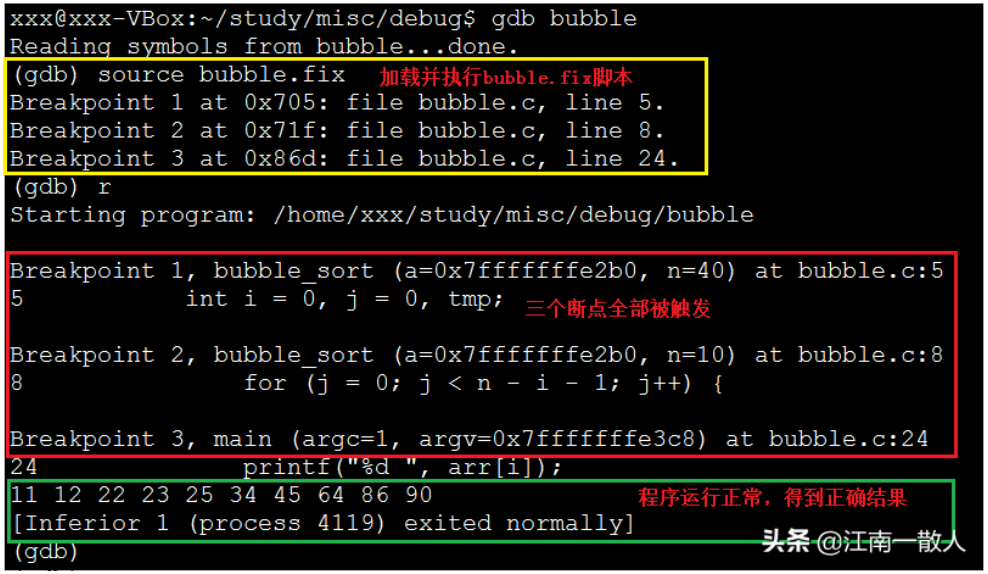
1. 删除了格式化打印信息。
2. 删除了commands后面的断点ID。上文讲过，commands后面的断点ID可以省略，表示对最近一次设置的断点有效。为了让脚本更加通用，每个commands都紧跟在break命令之

后，因此直接省略了断点ID。

GDB的脚本可以通过两种方式执行：

1. 启动GDB时，用-x参数指定要执行的脚本文件。
2. 启动GDB后，执行source命令执行指定的脚本。

下面，我们用第二种方式演示一下，如下图所示：



执行bubble.fix脚本

使用source命令加载并执行bubble.fix，然后用run命令执行程序，三个断点均被触发，且预设的命令全部被正确执行，最后程序运行正常，得到期望的结果！

我们现在可以利用我们制作的“热补丁”脚本，在不修改代码、不重新编译和部署的前提下，成功修复程序中的BUG！是不是很有趣呢?

不过，做到这种程度，还不算完美！

尽管得到了正确的结果，但程序执行时，总是会打印我们设置的断点信息，看起来还是有些视

觉干扰的。

最后，我们来解决这个问题，让我们的“热补丁”更加完美！

优化“热补丁”脚本，隐藏断点信息

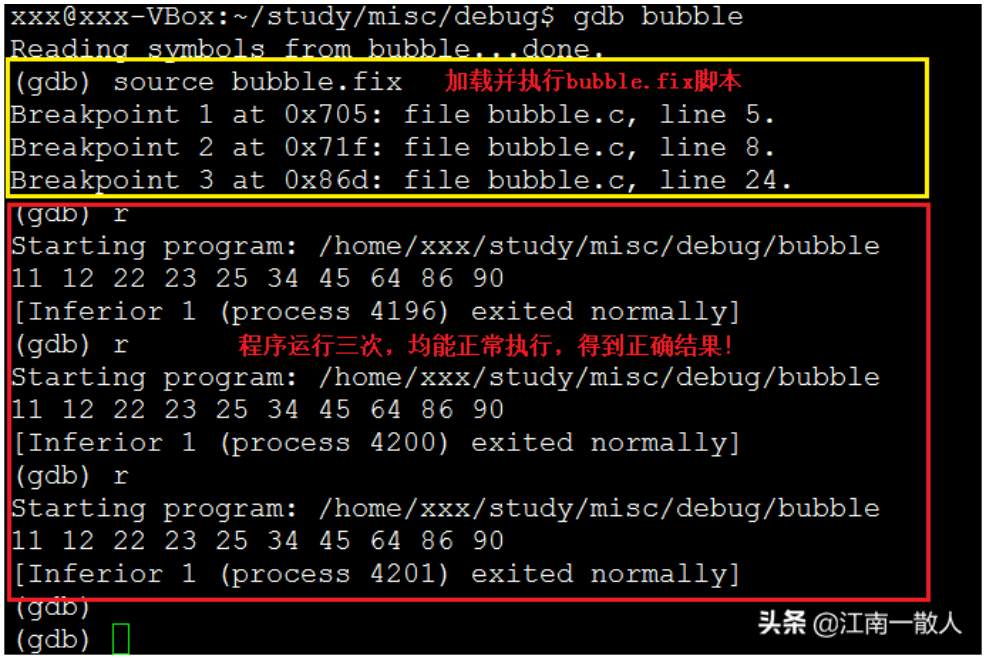
在预设的命令中，如果第一条命令是silent，断点被触发的打印信息会被屏蔽掉。

我们把bubble.fix做些修改，把silent命令加进去，如下图所示：

```
b bubble_sort
commands
  silent
  set var n=10
  continue
end

b 8 if i==n
commands
  silent
  return
  continue
end

b 24 if i==10
commands
  silent
  jump 26
  continue
end
```


最终版bubble.fix 脚本

然后，重新执行一下：



这样，看起来，清爽多了！

到此，我们终于实现了本文的目标：一边debug，一边修复BUG，避免反复修改代码、重新编译和部署、提高调试效率！

结语

本文重点介绍了如何利用GDB breakpoint command lists功能，制作“调试热补丁”，修改代码BUG。还可以利用这个功能，快速验证我们的猜想和解决方案，避免反复修改代码和重新编译。

巧用GDB breakpoint command lists功能，可以做很多有趣的事情，如实现调试自动化，提高调试效率等。

本文是调试系列专题文章的第七篇，如果对调试感兴趣的话，欢迎围观其它已更新的内容，相信一定会有收获的：

段错误(segmentation fault)：9种实用调试方法，你用过几种?

GDB动态打印：让你随时随地printf，不需修改代码，不需重新编译

调试引入的不确定性：必现的BUG神秘消失，断点改变代码执行逻辑

Linux调试技巧：GDB自定义命令，按需定制适合自己的调试工具

C语言：当GDB遇到复杂数据结构，两分钟带你掌握四个高效调试技巧

C语言：GDB调试时遇到宏定义怎么办？一个小技巧帮你一秒钟搞定

对编译、链接、OS内核等感谢的童鞋，欢迎围观另外一个系列专题：

你真的理解"Hello world"吗？ 从编译链接到OS内核系列专题（已更新三篇）

有任何疑问、建议，欢迎留言讨论！

原创不易，别忘了转发点赞，把知识分享给志同道合的朋友，谢谢！

对编译器、OS内核、性能调优、虚拟化等技术感兴趣的童鞋，欢迎右上角关注！

作为码农，我是个急性子，更是患有严重“懒癌”。最不喜欢做的事情，就是反复去折腾编译和部署的事情，尽管很多时候，可能仅仅是点几个按钮，敲几条命令而已，但真的不愿意浪费几十分钟甚至几个小时的时间去等待编译和部署的过程。曾经为了偷懒，避免调试时候反复修改代码和重新编译部署，自己开发了一个热补丁工具，可以在不中断程序执行的前提下，替换掉程序中的某个函数，只需要把新写的这个函数单独放到一个C文件中，单独编译一下就可以了。这样就可以快速fix掉一些BUG，或者验证一些想法，当时利用这个工具确实避免了很多重新编译和部署的工作。不过，可惜的是，那个工具有很多使用限制，有些情况下无法使用。后来才发现，原来利用GDB的breakpoint command lists功能居然也可以做类似的事情，而且通用性非常好！简直是如获至宝的感觉！可惜的是，这个功能大多数人都不知道！希望我这篇文章，可以让更多童鞋了解一下这个功能吧！应该对绝大多数Linux下的开发人员都有帮助！有任何疑问，欢迎留言讨论！觉得有用的话，别忘了点赞转发，分享给更多志同道合的朋友！谢谢！也请大家留言反馈，文章有哪些需要提高和完善的地方，帮我提高一下写作的能力，谢谢！

回复 · 4条回复 8

江南一散人 6月前

刚更新了一篇本文的续文，5种不同的方式，利用GDB给程序打“热补丁”，感兴趣的话，可以去看下。

回复 2

学无止境攻城狮 6月前

调试过程中讲一下关于汇编的东西

回复 · 5条回复 2

陈先生万寿无疆 7月前

用这个方法找到正确的值之后，最后是不是还是得改代码然后重新编译一次？这个做法是免去了改一下重新编译一次的麻烦，是这样吗

回复 · 2条回复 2

wdq689 7月前

期待更多linux下的使用技巧 老板 有没有不错的vim配置 开发环境

回复 · 2条回复 1

查看更多评论