

Valgrind+gdb 应用实战 20200630

零声学院 darren QQ 326873713

| | |
|--|----|
| 1. Valgrind 概述 | 2 |
| 1.1 体系结构..... | 2 |
| 1.2 编译安装..... | 3 |
| 1.3 Valgrind(memcheck)报错分类解析 | 3 |
| 2. Valgrind 使用 | 5 |
| 3. 利用 Memcheck 发现常见的内存问题..... | 7 |
| 3.1 使用未初始化的内存..... | 7 |
| 3.2 内存读写越界..... | 9 |
| 3.3 内存覆盖..... | 10 |
| 3.4 动态内存管理错误..... | 11 |
| 3.5 内存泄露..... | 14 |
| 4. GDB 调试器 | 17 |
| 4.1 调试信息与调试原理..... | 17 |
| 4.2 启动 GDB 调试 | 18 |
| 4.2.1 直接调试目标程序..... | 19 |
| 4.2.2 附加进程..... | 19 |
| 4.2.3 调试 core 文件..... | 20 |
| 4.3 常用简介..... | 24 |
| 4.4 常用命令实战..... | 25 |
| 4.4.1 run 命令 | 26 |
| 4.4.2 continue 命令 | 27 |
| 4.4.3 break 命令..... | 28 |
| 4.4.4 backtrace 与 frame 命令 | 30 |
| 4.4.5 info break、enable、disable 和 delete 命令 | 31 |
| 4.4.6 list 命令 | 33 |
| 4.4.7 print 和 ptype 命令..... | 37 |
| 4.4.8 ptype 命令 | 38 |
| 4.4.9 info 和 thread 命令..... | 38 |
| 4.4.10 next、step、until、finish、return 和 jump 命令 | 42 |
| 4.4.11 disassemble 命令 | 48 |
| 4.4.12 set args 和 show args 命令 | 49 |
| 4.4.13 tbreak 命令 | 50 |
| 4.4.14 watch 命令..... | 50 |
| 4.4.15 display 命令 | 53 |
| 4.5 调试技巧..... | 55 |
| 4.5.1 将 print 打印结果显示完整..... | 55 |
| 4.5.2 多线程下禁止线程切换..... | 55 |

| | |
|--------------------------------|----|
| 4.5.3 条件断点..... | 56 |
| 4.5.4 使用 GDB 调试多进程程序..... | 58 |
| 5. GDB TUI——在 GDB 中显示程序源码..... | 59 |
| 5.1 开启 GDB TUI 模式..... | 59 |
| 5.2 GDB TUI 模式常用窗口 | 59 |
| 5.3 常用快捷键..... | 61 |
| 5.4 窗口焦点切换..... | 62 |
| 6. 参考文档..... | 64 |

1. Valgrind 概述

官方参考文档: <http://valgrind.org/docs/manual/QuickStart.html>

辅助开发

1.1 体系结构

Valgrind 是一套 Linux 下，开放源代码（GPL V2）的**仿真调试工具**的集合。Valgrind 由内核（core）以及基于内核的其他调试工具组成。内核类似于一个框架（framework），它模拟了一个 CPU 环境，并提供服务给其他工具；而其他工具则类似于插件（plug-in），利用内核提供的服务完成各种特定的内存调试任务。Valgrind 的体系结构如下图所示：

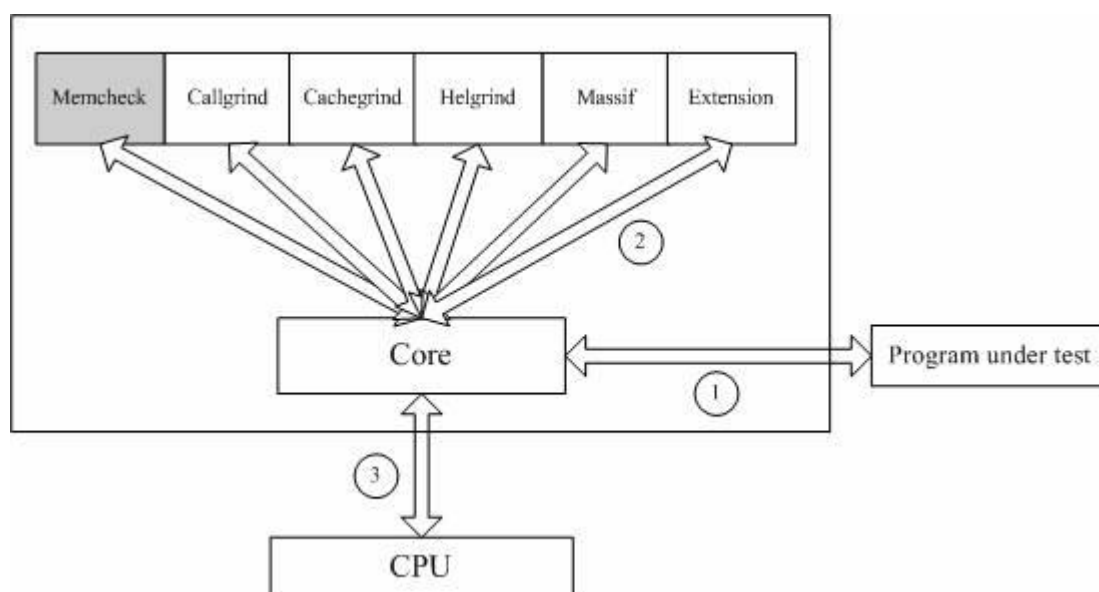


图 1 Valgrind 体系结构

Valgrind 体系结构

Valgrind 包括如下一些工具：

- **Memcheck**。这是 valgrind 应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。这也是本文将重点介绍的部分。
- **Callgrind**。它主要用来检查程序中函数调用过程中出现的问题。
- **Cachegrind**。它主要用来检查程序中缓存使用出现的问题。
- **Helgrind**。它主要用来检查多线程程序中出现的竞争问题。
- **Massif**。它主要用来检查程序中堆栈使用中出现的问题。
- **Extension**。可以利用 core 提供的功能，自己编写特定的内存调试工具。

1.2 编译安装

```
wget https://fossies.org/linux/misc/valgrind-3.15.0.tar.bz2
tar -jxvf valgrind-3.15.0.tar.bz2
cd valgrind-3.15.0
./configure
make
sudo make install
```

1.3 Valgrind(memcheck)报错分类解析

Valgrind(memcheck)包含这 7 类错误(高亮部分)，黑体为一般的错误提示：

1.illegal read/illegal write errors

Invalid read of size 4

2.use of uninitialised values

Conditional jump or move depends on uninitialised value(s)

3.use of uninitialised or unaddressable values in system calls

Syscall param write(buf) points to uninitialised byte(s)

4.illegal frees

Invalid free()

5.when a heap block is freed with an inappropriate deallocation function

Mismatched free() / delete / delete []

6.overlapping source and destination blocks

Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)

7.memory leak detection

7.1 Still reachable (cover case 1,2)

A start-pointer or chain of start-pointers to the block is found.memcheck won't report unless `--show-reachable=yes` is specified

内存指针还在还有机会使用或者释放，指针指向的动态内存还没有被释放就退出了

7.2 Definitely（明确地） lost (cover case 3)

no pointer to the block can be found memcheck won't report such blocks individually unless `--show-reachable=yes` is specified

确定的内存泄露，已经不能够访问这块内存

7.3 Indirectly lost (cover case 4,9)

the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost

指向该内存的指针都位于内存泄露处

7.4 Possibly lost (cover case 5,6,7,8)

a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer

可能的内存泄露，仍然存在某个指针能够访问某块内存，但该指针指向的已经不是该内存首位置

2. Valgrind 使用

第一步：准备好程序

为了使 **valgrind** 发现的错误更精确，如能够定位到源代码行，建议在编译时加上 **-g** 参数，编译优化选项请选择 **O0**，虽然这会降低程序的执行效率。

这里用到的示例程序文件名为：**sample.c**（如下所示），选用的编译器为 **gcc**。

生成可执行程序 **gcc -g -O0 sample.c -o sample**

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. void fun( )
4. {
5.     int *p = (int *)malloc(10*sizeof(int));
6.     p[10] = 0;
7. }
8.
9. int main(void)
10. {
11.     fun();
12.     return 0;
13. }
```

第二步：在 **valgrind** 下，运行可执行程序。

利用 **valgrind** 调试内存问题，不需要重新编译源程序，它的输入就是二进制的可执行程序。调用 **Valgrind** 的通用格式是：**valgrind [valgrind-options] your-prog [your-prog-options]**

Valgrind 的参数分为两类，一类是 **core** 的参数，它对所有的工具都适用；另外一类就是具体某个工具如 **memcheck** 的参数。**Valgrind** 默认的工具就是 **memcheck**，也可以通过“**--tool=tool name**”指定其他的工具。**Valgrind** 提供了大量的参数满足你特定的调试需求，具体可参考其用户手册。

这个例子将使用 **memcheck**，于是可以输入命令如下：**valgrind <Path>/sample**。

第三步：分析 **valgrind** 的输出信息。

以下是运行上述命令后的输出。

```
$ valgrind ./sample
==8378== Memcheck, a memory error detector
==8378== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8378== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8378== Command: ./sample
==8378==
```

```

==8378== Invalid write of size 4

==8378==      at 0x109153: fun (sample.c:6)
==8378==      by 0x109169: main (sample.c:11)
==8378== Address 0x4a48068 is 0 bytes after a block of size 40 alloc'd
==8378==      at 0x4837753: malloc (vg_replace_malloc.c:309)
==8378==      by 0x109146: fun (sample.c:4)
==8378==      by 0x109169: main (sample.c:11)
==8378==
==8378==
==8378== HEAP SUMMARY:
==8378==      in use at exit: 40 bytes in 1 blocks
==8378==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==8378==
==8378== LEAK SUMMARY:
==8378==      definitely lost: 40 bytes in 1 blocks
==8378==      indirectly lost: 0 bytes in 0 blocks
==8378==      possibly lost: 0 bytes in 0 blocks
==8378==      still reachable: 0 bytes in 0 blocks
==8378==      suppressed: 0 bytes in 0 blocks
==8378== Rerun with --leak-check=full to see details of leaked memory
==8378==
==8378== For lists of detected and suppressed errors, rerun with: -s
==8378== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

- 左边显示类似行号的数字（32372）表示的是 Process ID。
- 最上面的红色方框表示的是 **valgrind** 的版本信息。
- 中间红色方框表示 **valgrind** 通过运行被测试程序，发现的内存问题。通过阅读这些信息，可以发现：
 1. 这是一个对内存的非法写操作，非法写操作的内存是 4 bytes。
 2. 发生错误时的函数堆栈，以及具体的源代码行号。
 3. 非法写操作的具体地址空间。
- 最下面的红色方框是对发现的内存问题和内存泄露问题的总结。内存泄露的大小（40 bytes）也能够被检测出来。

示例程序显然有两个问题，一是 **fun** 函数中动态申请的堆内存没有释放；二是对堆内存的访问越界。这两个问题均被 **valgrind** 发现。

3. 利用 Memcheck 发现常见的内存问题

在 Linux 平台开发应用程序时，最常遇见的问题就是错误的使用内存，我们总结了常见了内存错误使用情况，并说明了如何用 valgrind 将其检测出来。

3.1 使用未初始化的内存

问题分析：

对于位于程序中不同段的变量，其初始值是不同的，全局变量和静态变量初始值为 0，而局部变量和动态申请的变量，其初始值为随机值。如果程序使用了为随机值的变量，那么程序的行为就变得不可预期。

下面的程序就是一种常见的，使用了未初始化的变量的情况。数组 a 是局部变量，其初始值为随机值，而在初始化时并没有给其所有数组成员初始化，如此在接下来使用这个数组时就潜在有内存问题。

badloop.c

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int a[5];
6.     int i, s;
7.     a[0] = a[1] = a[2] = a[3] = a[4] = 0;
8.     for (i = 0; i < 5; i++)
9.     {
10.         s += a[i];
11.     }
12.     if (s == 377)
13.     {
14.         printf("sum is %d\n", s);
15.     }
16.     return 0;
17. }
```

假设这个文件名为：badloop.c，生成的可执行程序为 badloop。用 memcheck 对其进行测试，输出如下。

```
$ valgrind ./badloop
==10490== Memcheck, a memory error detector
==10490== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10490== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==10490== Command: ./badloop
```

```

==10490==
==10490== Conditional jump or move depends on uninitialised value(s)
==10490==    at 0x1091A1: main (badloop.c:12)
==10490==
==10490==
==10490== HEAP SUMMARY:
==10490==    in use at exit: 0 bytes in 0 blocks
==10490==    total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10490==
==10490== All heap blocks were freed -- no leaks are possible
==10490==
==10490== Use --track-origins=yes to see where uninitialised values come from
==10490== For lists of detected and suppressed errors, rerun with: -s
==10490== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

输出结果显示，在该程序第 12 行中，程序的跳转依赖于一个未初始化的变量。准确的发现了上述程序中存在的问题。

3.1.1 补充：添加参数

(1) valgrind 对于 protobuf, stl 这样的 3 方库的兼容性不算太好，所以会造成输出一堆的 still reachable 字样。其实完全没有必要去纠结这些问题。可以肯定这些都是误判的。当你使用了这样的库的情况下，一般都会需要将检查 option 关闭掉。防止自己被爆出来的一堆的错误唬住了。信息太多反而阻碍自己的判断。

(2) 我们可以将测试结果指定输出文件。

```

valgrind --log-file=./valgrind_report.log --leak-check=full --show-leak-kinds=all --show-reachable=no --track-origins=yes ./执行程序

```

参数说明

■ -log-file

指定报告输出文件

■ -track-origins=yes

是否显示未定义的变量，在堆、栈中被定义没有被 initialised 的变量都被定义成 origins。默认是关闭这个 option 的。

■ -show-leak-kinds=all

这里可以支持的选项有[definite|possible]，一般只需要去关注 definite（明确的），possible 是可能会存在。

■ -leak-check=full

当服务器退出时是否收集输出内存泄漏，选项有[no|summary|full]这个地方我们将其设置成全输出，默认将会使用 summary 方式。

比如刚才的例子

```
valgrind --log-file=./valgrind_report.log --leak-check=full --show-leak-kinds=all --show-reachable=no --track-origins=yes ./sample
```

测试测试报告就输出到了当前目录的：valgrind_report.log

3.2 内存读写越界

问题分析：

这种情况是指：访问了你不应该/没有权限访问的内存地址空间，比如访问数组时越界；对动态内存访问时超出了申请的内存大小范围。下面的程序就是一个典型的数组越界问题。pt 是一个局部数组变量，其大小为 4，p 初始指向 pt 数组的起始地址，但在对 p 循环叠加后，p 超出了 pt 数组的范围，如果此时再对 p 进行写操作，那么后果将不可预期。

badacc.cpp

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main(void)
5. {
6.     int len = 4;
7.     int *pt = (int *)malloc(len * sizeof(int)); // 没有释放，内存泄露
8.     int *p = pt;
9.     for (int i = 0; i < len; i++)
10.    {
11.        p++;
12.    }
13.    *p = 5; // 无效写入
14.    printf("the value of p equal:%d\n", *p); // 无效读取
15.
16.    return 0;
17. }
```

结果分析：

假设这个文件名为 `badacc.cpp`，生成的可执行程序为 `badacc`，用 `memcheck` 对其进行测试，输出如下。

```
==10640== Invalid write of size 4
==10640==    at 0x10918F: main (badacc.c:13)
==10640== Address 0x4a48050 is 0 bytes after a block of size 16 alloc'd
==10640==    at 0x4837753: malloc (vg_replace_malloc.c:309)
==10640==    by 0x109164: main (badacc.c:7)
==10640==
==10640== Invalid read of size 4
==10640==    at 0x109199: main (badacc.c:14)
==10640== Address 0x4a48050 is 0 bytes after a block of size 16 alloc'd
==10640==    at 0x4837753: malloc (vg_replace_malloc.c:309)
==10640==    by 0x109164: main (badacc.c:7)
==10640==
the value of p equal:5
==10640==
==10640== HEAP SUMMARY:
==10640==    in use at exit: 16 bytes in 1 blocks
==10640== total heap usage: 2 allocs, 1 frees, 1,040 bytes allocated
==10640==
==10640== LEAK SUMMARY:
==10640==    definitely lost: 16 bytes in 1 blocks
==10640==    indirectly lost: 0 bytes in 0 blocks
==10640==    possibly lost: 0 bytes in 0 blocks
==10640==    still reachable: 0 bytes in 0 blocks
==10640==    suppressed: 0 bytes in 0 blocks
==10640== Rerun with --leak-check=full to see details of leaked memory
==10640==
==10640== For lists of detected and suppressed errors, rerun with: -s
==10640== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

输出结果显示，在该程序的第 15 行，进行了非法的写操作；在第 16 行，进行了非法读操作。准确地发现了上述问题。

3.3 内存覆盖

问题分析：

C 语言的强大和可怕之处在于其可以直接操作内存，C 标准库中提供了大量这样的函数，比如 `strcpy`, `strncpy`, `memcpy`, `strcat` 等，这些函数有一个共同的特点就是需要设置源地址 (`src`)，和目标地址(`dst`)，`src` 和 `dst` 指向的地址不能发生重叠，否则结果将不可预期。

下面就是一个 `src` 和 `dst` 发生重叠的例子。

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main(void)
5. {
6.     char x[50] = {0};
7.
8.     strncpy(x+20, x, 20); //ok
9.     strncpy(x+20, x, 21); // 这里检测不出问题。
10.    strncpy(x, x+20, 20); //ok
11.    strncpy(x, x+20, 21); // verlap
12.
13.    return 0;
14. }
```

结果分析：

假设这个文件名为 `badlap.cpp`，生成的可执行程序为 `badlap`，用 `memcheck` 对其进行测试，输出如下：

```
==10730== Source and destination overlap in strncpy(0x1fff000310, 0x1fff000324, 21)
==10730==    at 0x483ADD0: strncpy (vg_replace_strmem.c:552)
==10730==    by 0x109201: main (badlap.c:11)
==10730==
==10730==
==10730== HEAP SUMMARY:
==10730==    in use at exit: 0 bytes in 0 blocks
==10730==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10730==
==10730== All heap blocks were freed -- no leaks are possible
==10730==
==10730== For lists of detected and suppressed errors, rerun with: -s
==10730== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

输出结果显示上述程序中第 11 行，源地址和目标地址设置出现重叠。准确的发现了上述问题。

3.4 动态内存管理错误

问题分析：

常见的内存分配方式分三种：静态存储，栈上分配，堆上分配。全局变量属于静态存储，它们是在编译时就被分配了存储空间，函数内的局部变量属于栈上分配，而最灵活的内存使用方式当属堆上分配，也叫做内存动态分配了。常用的内存动态分配函数包括：`malloc`, `alloc`, `realloc`, `new` 等，动态释放函数包括 `free`, `delete`。

一旦成功申请了动态内存，我们就需要自己对其进行内存管理，而这又是最容易犯错误的。下面的一段程序，就包括了内存动态管理中常见的错误。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main(void)
5. {
6.     char *p = (char *)malloc(10);
7.     char *pt = p;
8.     for(int i = 0; i < 10; i++)
9.     {
10.         p[i] = 'z';
11.     }
12.
13.     delete p;
14.
15.     pt[1] = 'x';
16.     free(pt);
17.
18.     return 0;
19. }
```

常见的内存动态管理错误包括：

■ 申请和释放不一致

由于 C++ 兼容 C，而 C 与 C++ 的内存申请和释放函数是不同的，因此在 C++ 程序中，就有两套动态内存管理函数。一条不变的规则就是采用 C 方式申请的内存就用 C 方式释放；用 C++ 方式申请的内存，用 C++ 方式释放。也就是用 malloc/alloc/realloc 方式申请的内存，用 free 释放；用 new 方式申请的内存用 delete 释放。在上述程序中，用 malloc 方式申请了内存却用 delete 来释放，虽然这在很多情况下不会有问题，但这绝对是潜在的问题。

■ 申请和释放不匹配

申请了多少内存，在使用完成后就要释放多少。如果没有释放，或者少释放了就是内存泄露；多释放了也会产生问题。上述程序中，指针 p 和 pt 指向的是同一块内存，却被先后释放两次。

■ 释放后仍然读写

本质上说，系统会在堆上维护一个动态内存链表，如果被释放，就意味着该块内存可以继续被分配给其他部分，如果内存被释放后再访问，就可能覆盖其他部分的信息，这是一种严重的错误，上述程序第 15 行中就在释放后仍然写这块内存。

结果分析：

假设这个文件名为 **badmac.cpp**，生成的可执行程序为 **badmac**，用 **memcheck** 对其进行测试，输出如下。

```
==10902== Mismatched free() / delete / delete []
==10902==    at 0x4838F9E: operator delete(void*, unsigned long) (vg_replace_malloc.c:595)
==10902==    by 0x1091A6: main (badmac.cpp:13)
==10902== Address 0x4d8dc80 is 0 bytes inside a block of size 10 alloc'd
==10902==    at 0x4837753: malloc (vg_replace_malloc.c:309)
==10902==    by 0x109166: main (badmac.cpp:6)
==10902==
==10902== Invalid write of size 1
==10902==    at 0x1091AF: main (badmac.cpp:15)
==10902== Address 0x4d8dc81 is 1 bytes inside a block of size 10 free'd
==10902==    at 0x4838F9E: operator delete(void*, unsigned long) (vg_replace_malloc.c:595)
==10902==    by 0x1091A6: main (badmac.cpp:13)
==10902== Block was alloc'd at
==10902==    at 0x4837753: malloc (vg_replace_malloc.c:309)
==10902==    by 0x109166: main (badmac.cpp:6)
==10902==
==10902== Invalid free() / delete / delete[] / realloc()
==10902==    at 0x4838900: free (vg_replace_malloc.c:540)
==10902==    by 0x1091BD: main (badmac.cpp:16)
==10902== Address 0x4d8dc80 is 0 bytes inside a block of size 10 free'd
==10902==    at 0x4838F9E: operator delete(void*, unsigned long) (vg_replace_malloc.c:595)
==10902==    by 0x1091A6: main (badmac.cpp:13)
==10902== Block was alloc'd at
==10902==    at 0x4837753: malloc (vg_replace_malloc.c:309)
==10902==    by 0x109166: main (badmac.cpp:6)
==10902==
==10902==
==10902==
==10902== HEAP SUMMARY:
==10902==    in use at exit: 0 bytes in 0 blocks
==10902== total heap usage: 2 allocs, 3 frees, 72,714 bytes allocated
==10902==
==10902== All heap blocks were freed -- no leaks are possible
==10902==
==10902== For lists of detected and suppressed errors, rerun with: -s
==10902== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

输出结果显示，第 13 行分配和释放函数不一致；第 15 行发生非法写操作，也就是往释放后的内存地址写值；第 16 行释放内存函数无效。准确地发现了上述三个问题。

3.5 内存泄露

问题描述:

内存泄露 (Memory leak) 指的是, 在程序中动态申请的内存, 在使用完后既没有释放, 又无法被程序的其他部分访问。内存泄露是在开发大型程序中最令人头疼的问题, 以至于有人说, 内存泄露是无法避免的。其实不然, 防止内存泄露要从良好的编程习惯做起, 另外重要的一点就是要加强单元测试 (Unit Test), 而 memcheck 就是这样一款优秀的工具。

在一个单独的函数中, 每个人的内存泄露意识都是比较强的。但很多情况下, 我们都会对 malloc/free 或 new/delete 做一些包装, 以符合我们特定的需要, 无法做到在一个函数中既使用又释放。这个例子也说明了内存泄露最容易发生的地方: 即两个部分的接口部分, 一个函数申请内存, 一个函数释放内存。并且这些函数由不同的人开发、使用, 这样造成内存泄露的可能性就比较大了。这需要养成良好的单元测试习惯, 将内存泄露消灭在初始阶段。

文件名: badleak.cpp

```
#include <iostream>
#include <vector>

class Item
{
public:
    Item()
    {
        printf("构造 Item\n");
    }
    ~Item()
    {
        printf("析构 Item\n");
    }
};

int main(void)
{
    std::vector<Item*> vItem;

    for(int i= 0; i < 2; i++)
    {
        Item *item = new Item();
        vItem.push_back(item);
    }
}
```



```
    return 0;
}
```

结果分析:

假设上述文件名为 `badleak.cpp`，生成的可执行程序为 `badleak`，用 `memcheck` 对其进行测试，输出如下。

```
$ valgrind ./badleak
==9036== Memcheck, a memory error detector
==9036== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9036== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9036== Command: ./badleak
==9036==
构造 Item
构造 Item
==9036==
==9036== HEAP SUMMARY:
==9036==      in use at exit: 5 bytes in 5 blocks
==9036==    total heap usage: 11 allocs, 6 frees, 73,853 bytes allocated
==9036==
==9036== LEAK SUMMARY:
==9036==      definitely lost: 5 bytes in 5 blocks
==9036==      indirectly lost: 0 bytes in 0 blocks
==9036==      possibly lost: 0 bytes in 0 blocks
==9036==      still reachable: 0 bytes in 0 blocks
==9036==      suppressed: 0 bytes in 0 blocks
==9036== Rerun with --leak-check=full to see details of leaked memory
==9036==
==9036== For lists of detected and suppressed errors, rerun with: -s
==9036== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

改成智能指针的方式，文件名 `badleak2.cpp`

```
#include <iostream>
#include <vector>
#include <memory>

class Item
{
public:
    Item()
    {
        printf("构造 Item\n");
    }
};
```

```

    }
    ~Item()
    {
        printf("析构 Item\n");
    }
};

int main(void)
{
    std::vector<std::shared_ptr<Item>> vItem;

    for(int i= 0; i < 2; i++)
    {
        std::shared_ptr<Item> item = std::make_shared<Item>();
        vItem.push_back(item);
    }

    return 0;
}

```

此时就正常了

```

$ valgrind ./badleak2
==9097== Memcheck, a memory error detector
==9097== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9097== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9097== Command: ./badleak2
==9097==
构造 Item
构造 Item
析构 Item
析构 Item
==9097==
==9097== HEAP SUMMARY:
==9097==      in use at exit: 0 bytes in 0 blocks
==9097==    total heap usage: 6 allocs, 6 frees, 73,824 bytes allocated
==9097==
==9097== All heap blocks were freed -- no leaks are possible

```

4. GDB 调试器

官方参考文档: <http://www.gnu.org/software/gdb/documentation/>

GDB (GNU 项目调试器) 可以让您了解程序在执行时“内部”究竟在干些什么, 以及在程序发生崩溃的瞬间正在做什么。

GDB 做以下 4 件主要的事情来帮助您捕获程序中的 bug:

- 在程序启动之前指定一些可以影响程序行为的变量或条件
- 在某个指定的地方或条件下暂停程序
- 在程序停止时检查已经发生了什么
- 在程序执行过程中修改程序中的变量或条件, 这样就可以体验修复一个 bug 的成果, 并继续了解其他 bug

4.1 调试信息与调试原理

一般要调试某个程序, 为了能清晰地看到调试的每一行代码、调用的堆栈信息、变量名和函数名等信息, 需要调试程序含有调试符号信息。使用 `gcc` 编译程序时, 如果加上 `-g` 选项即可在编译后的程序中保留调试符号信息。举个例子, 以下命令将生成一个带调试信息的程序 `hello_server` (范例: `hello_server.c`)。

```
gcc -g -o hello_server hello_server.c
```

那么如何判断 `hello_server` 是否带有调试信息呢? 我们使用 `gdb` 来调试一下这个程序, `gdb` 会显示正确读取到该程序的调试信息, 在打开的 Linux Shell 终端输入 `gdb hello_server` 查看显示结果即可:

```
$ gdb ./hello_server
GNU gdb (Ubuntu 8.2-0ubuntu1) 8.2
Copyright (C) 2018 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
```

For bug reporting instructions, please see:

<<http://www.gnu.org/software/gdb/bugs/>>.

Find the GDB manual and other documentation resources online at:

<<http://www.gnu.org/software/gdb/documentation/>>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from ./hello_server...done.

(gdb)

Gdb 加载成功以后，会显示如下信息：

Reading symbols from ./hello_server...done.

即读取符号文件完毕，说明该程序含有调试信息。我们不加 -g 选项再试试：

gcc -g -o hello_server2 hello_server.c

Reading symbols from ./hello_server2...(no debugging symbols found)...done.

顺便提一下，除了不加 -g 选项，也可以使用 Linux 的 strip 命令移除掉某个程序中的调试信息，我们这里对 hello_server 使用 strip 命令试试：

\$ strip hello_server

##使用 strip 命令之前

-rwxrwxrwx 1 root root 18928 Nov 2 11:30 hello_server

##使用 strip 命令之后

-rwxrwxrwx 1 root root 14320 Nov 2 11:32 hello_server

在实际生成调试程序时，一般不仅要加上 -g 选项，也建议关闭编译器的程序优化选项。编译器的程序优化选项一般有五个级别，从 O0~O4 （注意第一个 O0，是字母 O 加上数字 0），O0 表示不优化，从 O1~O4 优化级别越来越高，O4 最高。这样做的目的是为了调试的时候，符号文件显示的调试变量等能与源代码完全对应起来。

4.2 启动 GDB 调试

GDB 调试主要有三种方式：

1. gdb filename 直接调试目标程序（gdb ./hello_server）

2. `gdb attach pid` 附加进程
3. `gdb filename corename` 调试 core 文件

4.2.1 直接调试目标程序

比如上一小节的
`gdb ./hello_server`

4.2.2 附加进程

在某些情况下，一个程序已经启动了，我们想调试这个程序，但是又不想重启这个程序。比如调试 redis。

```
lqf@ubuntu:~$ ps -ef | grep redis
lqf          2411    2397    0 11:38 pts/1    00:00:00 redis-server *:6379
```

得到 redis 进程 PID 为 2411，然后使用 `gdb attach 2411`，如果不是 root 权限需要加上 `sudo`，即是 `sudo gdb attach 2411`。

```
$ sudo gdb attach 2411
GNU gdb (Ubuntu 8.2-0ubuntu1) 8.2
For help, type "help".
Type "apropos word" to search for commands related to "word"...
attach: No such file or directory.
Attaching to process 2411
[New LWP 2415]
[New LWP 2416]
[New LWP 2417]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007f7867bf810f in epoll_wait (epfd=5, events=0x7f78676cd900, maxevents=10128,
    timeout=100) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
30  ../sysdeps/unix/sysv/linux/epoll_wait.c: No such file or directory.
(gdb)
```

当用 `gdb attach` 上目标进程后，调试器会暂停下来，此时可以使用 `continue` 命令让程序继续运行，或者加上相应的断点再继续运行程序。

当调试完程序想结束此次调试时，**而且不对当前进程 redis 有任何影响**，也就是说想让这个程序继续运行，可以在 GDB 的命令行界面输入 `detach` 命令让程序与 GDB 调试器分离，这样 redis 就可以继续运行了：

```
(gdb) detach
```

```
Detaching from program: /usr/local/bin/redis-server, process 2411
```

```
[Inferior 1 (process 2411) detached]
```

然后再退出 GDB:

(gdb) quit

4.2.3 调试 core 文件

有时候，服务器程序运行一段时间后会突然崩溃，这并不是我们希望看到的，需要解决这个问题。只要程序在崩溃的时候有 core 文件产生，就可以使用这个 core 文件来定位崩溃的原因。当然，Linux 系统默认是不开启程序崩溃产生 core 文件这一机制的，我们可以使用 `ulimit -c` 命令来查看系统是否开启了这一机制。

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 11036
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 11036
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

发现 `core file size` 那一行默认是 0，表示关闭生成 core 文件，可以使用“`ulimit` 选项名 设置值”来修改。例如，可以将 core 文件生成改成具体某个值（最大允许的字节数），这里我们使用 `ulimit -c unlimited`（`unlimited` 是 `-c` 选项值）直接修改成不限制大小。

将 `ulimit -c unlimited` 放入 `/etc/profile` 中，然后执行 `source /etc/profile` 即可立即生效。

即是：

- （1）将 `ulimit -c unlimited` 放入 `/etc/profile`
- （2）`source /etc/profile`
- （3）再次查看 `ulimit -a`

```
$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
```

| | |
|----------------------|-------------------------|
| scheduling priority | (-e) 0 |
| file size | (blocks, -f) unlimited |
| pending signals | (-i) 11036 |
| max locked memory | (kbytes, -l) 16384 |
| max memory size | (kbytes, -m) unlimited |
| open files | (-n) 1024 |
| pipe size | (512 bytes, -p) 8 |
| POSIX message queues | (bytes, -q) 819200 |
| real-time priority | (-r) 0 |
| stack size | (kbytes, -s) 8192 |
| cpu time | (seconds, -t) unlimited |
| max user processes | (-u) 11036 |
| virtual memory | (kbytes, -v) unlimited |
| file locks | (-x) unlimited |

范例测试 core_dump.c

```
#include <stdio.h>

int main(void)
{
    printf("hello world! dump core for set value to NULL pointer/n");

    *(char *)0 = 0;

    return 0;
}
```

编译运行

```
$ gcc -g -o core_dump core_dump.c
```

```
$ ./core_dump
```

Segmentation fault

用 ll 命令查看当前目录

```
lqf@ubuntu:~/0voice/gdb/gdb$ ll
total 144
drwxrwxrwx 2 lqf lqf 4096 Nov 4 10:27 ./
drwxrwxrwx 5 lqf lqf 4096 Nov 2 14:25 ../
-rw----- 1 lqf lqf 385024 Nov 4 10:24 core
-rwxrwxr-x 1 lqf lqf 18904 Nov 2 12:01 core_dump*
-rwxrwxrwx 1 lqf lqf 153 Nov 2 11:57 core_dump.c*
```

有产生 core 文件，使用 gdb 进行调试

```
$ gdb core_dump core
```

提示出错位置。

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from core_dump...done.
[New LWP 5040]
Core was generated by `./core_dump'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  main () at core_dump.c:7
--Type <RET> for more, q to quit, c to continue without paging--
7          *(char *)0 = 0;
(gdb) █
```

4.2.3.1 补充：Core Dump 的核心转储文件目录和命名规则

系统默认 corefile 是生成在程序的执行目录下或者程序启动调用了 chdir 之后的目录,我们可以通过设置生成 corefile 的格式来控制它，让其生成在固定的目录下（**对应的**
是自己的目录，不要直接写我的目录），并让每次启动后自动生效。

(1) 在/etc/sysctl.conf 写入 corefile 文件生成的目录。

```
kernel.core_pattern=/home/lqf/core_dump/core-%e-%p-%t
```

比如：

```
#####
# Magic system request Key
# 0=disable, 1=enable all, >1 bitmask of sysrq functions
# See https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html
# for what other values do
#kernel.sysrq=438
kernel.core_pattern=/home/lqf/core_dump/core-%e-%p-%t
```

文件末尾添加

(2) 创建应对的生成目录

```
mkdir /home/lqf/core_dump
```

(3) 然后执行生效

```
sudo sysctl -p /etc/sysctl.conf
```

其中：**/home/lqf/core_dump/** 对应自己要存放的目录，core-%e-%p-%t 文件格式

关于格式的的控制有如下几个参数：

| |
|--------------|
| %%: 相当于% |
| %p: 相当于<pid> |
| %u: 相当于<uid> |

%g: 相当于<gid>
%s: 相当于导致 dump 的信号的数字
%t: 相当于 dump 的时间
%e: 相当于执行文件的名称
%h: 相当于 hostname

(4) 可以使用 cat 去查看路径是否生效

```
cat /proc/sys/kernel/core_pattern
```

生效则显示:

```
lqf@ubuntu:~/core_dump$ cat /proc/sys/kernel/core_pattern  
/home/lqf/core_dump/core-%e-%p-%t
```

再次测试

(1) 先把原来的 core 文件先删除

(2) 然后执行 ./core_dump 程序

```
$ ./core_dump
```

Segmentation fault (core dumped)

(3) 查看 /home/lqf/core_dump

```
lqf@ubuntu:~/core_dump$ ll
```

total 124

```
drwxrwxr-x  2 lqf lqf   4096 Nov  4 11:06 ./
```

```
drwxr-xr-x 26 lqf lqf   4096 Nov  4 10:58 ../
```

```
-rw-----  1 lqf lqf 385024 Nov  4 11:05 core-core_dump-2725-1572836737
```

(4) 在(3)中可以看到生成的 corefile, 我们使用新生成的 corefile (注意引用完整的路径) 进行调试

```
$ gdb ./core_dump /home/lqf/core_dump/core-core_dump-2725-1572836737
```

GNU gdb (Ubuntu 8.2-0ubuntu1) 8.2

Reading symbols from ./core_dump...done.

[New LWP 2725]

Core was generated by './core_dump'.

Program terminated with signal SIGSEGV, Segmentation fault.

#0 main () at core_dump.c:7

```
7      *(char *)0 = 0;
```

(gdb)

4.3 退出 gdb

输入 `q` 或者 `Ctrl-d` 来退出。

4.4 常用简介

| 命令名称 | 命令缩写 | 命令说明 |
|-----------|--------|--|
| run | r | 运行一个程序 |
| continue | c | 让暂停的程序继续运行 |
| next | n | 运行到下一行 |
| step | s | 如果有调用函数，进入调用的函数内部，相当于 <code>step into</code> |
| until | u | 运行到指定行停下来 |
| finish | fi | 结束当前调用函数，到上一层函数调用处 |
| return | return | 结束当前调用函数并返回指定值，到上一层函数调用处 |
| jump | j | 将当前程序执行流跳转到指定行或地址 |
| print | p | 打印变量或寄存器值 |
| backtrace | bt | 查看当前线程的调用堆栈 |
| frame | f | 切换到当前调用线程的指定堆栈，具体堆栈通过堆栈序号指定 |
| thread | thread | 切换到指定线程 |
| break | b | 添加断点 |
| tbreak | tb | 添加临时断点 |
| delete | del | 删除断点 |
| enable | enable | 启用某个断点 |

| 命令名称 | 命令缩写 | 命令说明 |
|-------------|---------|----------------------|
| disable | disable | 禁用某个断点 |
| watch | watch | 监视某一个变量或内存地址的值是否发生变化 |
| list | l | 显示源码 |
| info | info | 查看断点 / 线程等信息 |
| ptype | ptype | 查看变量类型 |
| disassemble | dis | 查看汇编代码 |
| set args | | 设置程序启动命令行参数 |
| show args | | 查看设置的命令行参数 |

4.5 常用命令实战

以调试 redis 源码为例介绍常用的命令讲解。

1. 下载源码并解压

```
wget http://download.redis.io/releases/redis-6.0.3.tar.gz
```

```
tar zxvf redis-6.0.3.tar.gz
```

2. 进入 redis 源码目录并编译，注意编译时要生成调试符号并且关闭编译器优化选项。

```
cd redis-6.0.3
```

```
make CFLAGS="-g -O0" -j 2 （如果已经编译过先 make clean）
```

由于 redis 是纯 C 项目，使用的编译器是 gcc，因而这里设置编译器的选项时使用的是 CFLAGS 选项；如果项目使用的语言是 C++，那么使用的编译器一般是 g++，相对应的编译器选项是 CXXFLAGS。这点请读者注意区别。

另外，这里 makefile 使用了 -j 选项，其值是 2，表示开启 2 个进程同时编译，加快编译速度。

编译成功后，会在 `src` 目录下生成多个可执行程序，其中 `redis-server` 和 `redis-cli` 是需要调试的程序。进入 `src` 目录，使用 GDB 启动 `redis-server` 这个程序：

```
cd src
```

```
gdb ./redis-server
```

4.5.1 run 命令

默认情况下，`gdb filename` 命令只是附加的一个调试文件，并没有启动这个程序，需要输入 `run` 命令（简写为 `r`）启动这个程序：

```
(gdb) r
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
23155:C 30 Jun 2020 14:24:25.604 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
23155:C 30 Jun 2020 14:24:25.604 # Redis version=6.0.3, bits=64, commit=00000000, modified=0,
pid=23155, just started
23155:C 30 Jun 2020 14:24:25.604 # Warning: no config file specified, using the default config. In order
to specify a config file use /home/lqf/0voice/redis/redis-6.0.3/src/redis-server /path/to/redis.conf
23155:M 30 Jun 2020 14:24:25.608 * Increased maximum number of open files to 10032 (it was originally
set to 1024).
```

```
( _ _ _ _ _ ) Running in standalone mode
|_ _ _ _ _| Port: 6379
|_ _ _ _ _ / | PID: 23155
|_ _ _ _ _ / 
|_ _ _ _ _ 
|_ _ _ _ _ | http://redis.io
|_ _ _ _ _ 
|_ _ _ _ _ 
|_ _ _ _ _
```

```
23155:M 30 Jun 2020 14:24:25.612 # WARNING: The TCP backlog setting of 511 cannot be enforced
because /proc/sys/net/core/somaxconn is set to the lower value of 128.
23155:M 30 Jun 2020 14:24:25.612 # Server initialized
23155:M 30 Jun 2020 14:24:25.612 # WARNING overcommit_memory is set to 0! Background save may
```

```
fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf
and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
23155:M 30 Jun 2020 14:24:25.612 # WARNING you have Transparent Huge Pages (THP) support enabled
in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the
command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your
/etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
[New Thread 0x7ffff6624700 (LWP 23161)]
[New Thread 0x7ffff5e23700 (LWP 23162)]
[New Thread 0x7ffff5622700 (LWP 23163)]
[New Thread 0x7ffff4e21700 (LWP 23164)]
23155:M 30 Jun 2020 14:24:25.613 * Loading RDB produced by version 6.0.3
23155:M 30 Jun 2020 14:24:25.613 * RDB age 333454 seconds
23155:M 30 Jun 2020 14:24:25.613 * RDB memory usage when created 0.78 Mb
23155:M 30 Jun 2020 14:24:25.613 * DB loaded from disk: 0.000 seconds
23155:M 30 Jun 2020 14:24:25.613 * Ready to accept connections
```

这就是 `redis-server` 启动界面，假设程序已经启动，再次输入 `run` 命令则是重启程序。我们在 GDB 界面按 **Ctrl + C** 快捷键让 GDB 中断下来，再次输入 `run` 命令，GDB 会询问我们是否重启程序，输入 `yes` 确认重启。

```
^C
Thread 1 "redis-server" received signal SIGINT, Interrupt.
0x00007ffff7d2810f in epoll_wait (epfd=5, events=0x7ffff78de040, maxevents=10128,
    timeout=100) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
30 ../sysdeps/unix/sysv/linux/epoll_wait.c: No such file or directory.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) yes
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
```

4.5.2 continue 命令

当 GDB 触发断点或者使用 `Ctrl + C` 命令中断下来后，想让程序继续运行，只要输入 `continue` 命令即可（简写为 `c`）。当然，如果 `continue` 命令继续触发断点，GDB 就会再次中断下来。

```
^C
Thread 1 "redis-server" received signal SIGINT, Interrupt.
0x00007ffff7d2810f in epoll_wait (epfd=5, events=0x7ffff78de040, maxevents=10128,
    timeout=100) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
30 ../sysdeps/unix/sysv/linux/epoll_wait.c: No such file or directory.
```

```
(gdb) c
Continuing.
```

4.5.3 break 命令

break 命令（简称为 b）即我们添加断点的命令，可以使用以下方式添加断点：

- **break functionname**，在函数名为 functionname 的入口处添加一个断点；
- **break LineNo**，在当前文件行号为 LineNo 处添加一个断点；
- **break filename:LineNo**，在 filename 文件行号为 LineNo 处添加一个断点。

这三种方式都是我们常用的添加断点的方式。

在 redis main() 函数处添加一个断点：

```
(gdb) b main
Breakpoint 1 at 0x5555555a1fa0: file server.c, line 4969.
```

设置断点后重启程序

```
(gdb) r
Starting program: /home/lqf/0voice/gdb/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7ffffffe3b8) at server.c:4969
4969  int main(int argc, char **argv) { (gdb)
```

redis-server 默认端口号是 6379，绑定端口是需要调用 bind 函数，通过文件搜索可以找到相应位置文件，在 anet.c 455 行。

```
454 static int anetListen(char *err, int s, struct sockaddr *sa,
455     if (bind(s,sa,len) == -1) {
456         anetSetError(err, "bind: %s", strerror(errno));
457         close(s);
458         return ANET_ERR;
459     }
460
```

使用 break 命令在这个地方加一个断点：

```
(gdb) b anet.c:455
```

Breakpoint 2 at 0x555555585909: file anet.c, line 455.

由于程序绑定端口号是 redis-server 启动时初始化的，为了能触发这个断点，再次使用 run 命令重启下这个程序，GDB 第一次会触发 main() 函数处的断点，输入 continue 命令继续运行，接着触发 anet.c:455 处的断点：

```
(gdb) r
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

Breakpoint 1, main (argc=1, argv=0x7ffffffe3b8) at server.c:4969

```
4969 int main(int argc, char **argv) {
```

```
(gdb) c
```

Continuing.

```
23357:C 30 Jun 2020 14:37:58.196 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
```

```
23357:C 30 Jun 2020 14:37:58.197 # Redis version=6.0.3, bits=64, commit=00000000,
modified=0, pid=23357, just started
```

```
23357:C 30 Jun 2020 14:37:58.197 # Warning: no config file specified, using the default
config. In order to specify a config file use /home/lqf/0voice/redis/redis-6.0.3/src/redis-
server /path/to/redis.conf
```

```
23357:M 30 Jun 2020 14:37:58.200 * Increased maximum number of open files to 10032 (it
was originally set to 1024).
```

```
Breakpoint 2, anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0,
len=28, backlog=511) at anet.c:455
```

```
455 if (bind(s,sa,len) == -1) {
```

```
(gdb)
```

anet.c:455 对应的代码：

```
454 static int anetListen(char *err, int s, struct sockaddr *sa,
455     if (bind(s,sa,len) == -1) {
456         anetSetError(err, "bind: %s", strerror(errno));
457         close(s);
458         return ANET_ERR;
459     }
460
461     if (listen(s, backlog) == -1) {
462         anetSetError(err, "listen: %s", strerror(errno));
463         close(s);
464         return ANET_ERR;
465     }
466     return ANET_OK;
467 }
```

现在断点停在第 455 行，所以当前文件就是 anet.c，可以直接使用“break 行号”添加断点。例如，可以在第 458 行、464 行、466 行分别加一个断点，看看这个函数执行完毕后走哪个 return 语句退出，则可以执行：

```
(gdb) b 458
Breakpoint 3 at 0x555555585955: file anet.c, line 458.
(gdb) b 464
Breakpoint 4 at 0x5555555859a3: file anet.c, line 464.
(gdb) b 466
Breakpoint 5 at 0x5555555859aa: file anet.c, line 466.
```

添加好这三个断点以后，我们使用 continue 命令继续运行程序，发现程序运行到第 466 行中断下来（即触发 Breakpoint 5）：

```
(gdb) c
Continuing.

Breakpoint 5, anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0,
    len=28, backlog=511) at anet.c:466
466     return ANET_OK;
```

说明 redis-server 绑定端口号并设置侦听（listen）成功，我们可以再打开一个 SSH 窗口，验证一下，发现 6379 端口确实已经处于侦听状态了。

```
lqf@ubuntu:~$ lsof -i:6379
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
redis-ser 7379 lqf     6u   IPv6  63292    0t0  TCP *:6379 (LISTEN)
```

4.5.4 backtrace 与 frame 命令

backtrace 命令（简写为 bt）用来查看当前调用堆栈。接上，redis-server 现在中断在 anet.c:466 行，可以通过 backtrace 命令来查看当前的调用堆栈：

```
(gdb) bt
#0  anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0, len=28,
    backlog=511) at anet.c:466
#1  0x000055555559535e in _anetTcpServer (err=0x555555b5be68 <server+680> "", port=6379,
    bindaddr=0x0, af=10, backlog=511) at anet.c:501
```



```
#2 0x000055555559544f in anetTcp6Server (err=0x555555b5be68 <server+680> "", port=6379,
bindaddr=0x0, backlog=511) at anet.c:524
#3 0x000055555559ba26 in listenToPort (port=6379, fds=0x555555b5bd34 <server+372>,
count=0x555555b5bd74 <server+436>) at server.c:2648
#4 0x000055555559c1e2 in initServer () at server.c:2792
#5 0x00005555555a264d in main (argc=1, argv=0x7ffffffe3b8) at server.c:5128
```

这里一共有 6 层堆栈，最顶层是 `main()` 函数，最底层是断点所在的 `anetListen()` 函数，堆栈编号分别是 #0~#5，如果想切换到其他堆栈处，可以使用 **frame** 命令（简写为 **f**），该命令的使用方法是“**frame** 堆栈编号（编号不加 #）”。在这里依次切换至堆栈顶部，然后再切换回 #0 练习一下：

```
(gdb) f 1
#1 0x000055555559535e in _anetTcpServer (err=0x555555b5be68 <server+680> "", port=6379,
bindaddr=0x0, af=10, backlog=511) at anet.c:501
501         if (anetListen(err,s,p->ai_addr,p->ai_addrlen,backlog) == ANET_ERR) s = ANET_ERR;
(gdb) f 2
#2 0x000055555559544f in anetTcp6Server (err=0x555555b5be68 <server+680> "", port=6379,
bindaddr=0x0, backlog=511) at anet.c:524
524     return _anetTcpServer(err, port, bindaddr, AF_INET6, backlog);
(gdb) f 3
#3 0x000055555559ba26 in listenToPort (port=6379, fds=0x555555b5bd34 <server+372>,
count=0x555555b5bd74 <server+436>) at server.c:2648
2648         fds[*count] = anetTcp6Server(server.neterr,port,NULL,
(gdb) f 4
#4 0x000055555559c1e2 in initServer () at server.c:2792
2792         listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
(gdb) f 5
#5 0x00005555555a264d in main (argc=1, argv=0x7ffffffe3b8) at server.c:5128
5128     initServer();
```

4.5.5 info break、enable、disable 和 delete 命令

在程序中加了很多断点，而我们想查看加了哪些断点时，可以使用 **info break** 命令（简写为 **info b**）

```
(gdb) info b
```

| Num | Type | Disp | Enb | Address | What |
|-----|-------------------------------|------|-----|--|------|
| 1 | breakpoint | keep | y | 0x00005555555a1fa0 in main at server.c:4969 | |
| | breakpoint already hit 1 time | | | | |
| 2 | breakpoint | keep | y | 0x00005555555950b2 in anetListen at anet.c:455 | |
| | breakpoint already hit 1 time | | | | |
| 3 | breakpoint | keep | y | 0x00005555555950fe in anetListen at anet.c:458 | |

```

4      breakpoint      keep y    0x00005555559514c in anetListen at anet.c:464
5      breakpoint      keep y    0x000055555595153 in anetListen at anet.c:466
      breakpoint already hit 1 time

```

通过上面的内容片段可以知道，目前一共增加了 5 个断点，相应的断点信息比如每个断点的位置（所在的文件和行号）、内存地址、断点启用和禁用状态信息也一目了然。如果我们想禁用某个断点，使用“**disable 断点编号**”就可以禁用这个断点了，被禁用的断点不会再被触发；同理，被禁用的断点也可以使用“**enable 断点编号**”重新启用。

```

(gdb) disable 1
(gdb) info b
Num      Type      Disp Enb Address                               What
1        breakpoint keep n    0x00005555555a1fa0 in main at server.c:4969
      breakpoint already hit 1 time
2        breakpoint keep y    0x0000555555950b2 in anetListen at anet.c:455
      breakpoint already hit 1 time
3        breakpoint keep y    0x0000555555950fe in anetListen at anet.c:458
4        breakpoint keep y    0x00005555559514c in anetListen at anet.c:464
5        breakpoint keep y    0x000055555595153 in anetListen at anet.c:466
      breakpoint already hit 1 time

```

使用 `disable 1` 以后，第一个断点的 `Enb` 一栏的值由 `y` 变成 `n`，重启程序也不会再次触发

```

(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
23451:C 30 Jun 2020 14:48:07.095 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
23451:C 30 Jun 2020 14:48:07.096 # Redis version=6.0.3, bits=64, commit=00000000, modified=0,
pid=23451, just started
23451:C 30 Jun 2020 14:48:07.096 # Warning: no config file specified, using the default config. In
order to specify a config file use /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
/path/to/redis.conf
23451:M 30 Jun 2020 14:48:07.099 * Increased maximum number of open files to 10032 (it was
originally set to 1024).

Breakpoint 2, anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0,
len=28, backlog=511) at anet.c:455
455      if (bind(s,sa,len) == -1) {
(gdb)

```

如果 `disable` 命令和 `enable` 命令不加断点编号, 则分别表示禁用和启用所有断点:

```
(gdb) disable
(gdb) info b
```

| Num | Type | Disp | Enb | Address | What |
|-------------------------------|------------|------|-----|---------------------|-----------------------------|
| 1 | breakpoint | keep | n | 0x000055555555a1fa0 | in main at server.c:4969 |
| 2 | breakpoint | keep | n | 0x000055555555950b2 | in anetListen at anet.c:455 |
| breakpoint already hit 1 time | | | | | |
| 3 | breakpoint | keep | n | 0x000055555555950fe | in anetListen at anet.c:458 |
| 4 | breakpoint | keep | n | 0x0000555555559514c | in anetListen at anet.c:464 |
| 5 | breakpoint | keep | n | 0x00005555555595153 | in anetListen at anet.c:466 |

使用“`delete` 编号”可以删除某个断点, 如 `delete 2 3` 则表示要删除的断点 2 和断点 3:

```
(gdb) delete 2 3
(gdb) info b
```

| Num | Type | Disp | Enb | Address | What |
|-----|------------|------|-----|---------------------|-----------------------------|
| 1 | breakpoint | keep | n | 0x00005555555591308 | in main at server.c:4969 |
| 4 | breakpoint | keep | n | 0x000055555555859a3 | in anetListen at anet.c:464 |
| 5 | breakpoint | keep | n | 0x000055555555859aa | in anetListen at anet.c:466 |

同样的道理, 如果输入 `delete` 不加命令号, 则表示删除所有断点。

4.5.6 list 命令

```
(gdb) list
```

如果不带任何参数的话, 该命令会接着打印上次 `list` 命令打印出代码后面的代码。如果是第一次执行 `list` 命令则会显示当前正在执行代码位置附近的代码。

```
(gdb) list -
```

如果参数是一个减号的话, 则和前面刚好相反, 会打印上次 `list` 命令打印出代码前面的代码。

```
(gdb) list LOCATION
```

`list` 命令还可以带一个代码位置作为参数, 顾名思义, 这样的话就会打印出该代码位置附近的代码。这个代码位置的定义和在 `break` 命令中定义的相同, 可以是一个行号:

```
(gdb) list 100 #列出当前代码文件中第 100 行附近代码
```

(gdb) `list tcpdump.c:450` #列出 tcpdump.c 文件中第 450 行附近代码

也可以是一个函数名：

(gdb) `list main` #列出当前代码文件中 main 函数附近代码

(gdb) `list inet.c:pcap_lookupdev` #列出 inet.c 代码文件中指定函数附近代码

list 命令还可以指定要显示代码的具体范围：

(gdb) `list FIRST, LAST`

这里 FIRST 和 LAST 都是具体的代码位置，此时该命令将显示 FIRST 到 LAST 之间的代码。可以不指定 FIRST 或者 LAST 参数，这样的话就将显示 LAST 之前或者 FIRST 之后的代码。注意，即使只指定一个参数也要带逗号，否则就编程前面的命令，显示代码位置附近的代码了。

list 命令默认只会打印出 10 行源代码，如果觉得不够，可以使用如下命令修改：

(gdb) `set listsize COUNT`

这样的话，下次 list 命令就会显示 COUNT 行源代码了。如果想查看这个参数当前被设置成多少，可以使用如下命令：

(gdb) `show listsize`

还有一个非常有用的命令，如果你想看程序中一共定义了哪些函数，可以使用下面的命令：

(gdb) `info functions`

这个命令会显示程序中所有函数的名词，参数格式，返回值类型以及函数处于哪个代码文件中。

`list` 命令（简写为 `l`）可以查看当前断点处的代码。使用 `frame` 命令切换到刚才的堆栈 #4 处，然后输入 `list` 命令看下效果：

```
(gdb) c
Continuing.

Breakpoint 5, anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0,
    len=28, backlog=511) at anet.c:466
466     return ANET_OK;

(gdb) bt
```

```

(gdb) bt
#0  anetListen (err=0x555555b5be68 <server+680> "", s=6, sa=0x555555d70ca0, len=28,
    backlog=511) at anet.c:466
#1  0x000055555559535e in _anetTcpServer (err=0x555555b5be68 <server+680> "", port=6379,
    bindaddr=0x0, af=10, backlog=511) at anet.c:501
#2  0x000055555559544f in anetTcp6Server (err=0x555555b5be68 <server+680> "", port=6379,
    bindaddr=0x0, backlog=511) at anet.c:524
#3  0x0000555555595ba26 in listenToPort (port=6379, fds=0x555555b5bd34 <server+372>,
    count=0x555555b5bd74 <server+436>) at server.c:2648
#4  0x000055555559c1e2 in initServer () at server.c:2792
#5  0x00005555555a264d in main (argc=1, argv=0x7fffffff3b8) at server.c:5128
(gdb) f 4
#4  0x000055555559c1e2 in initServer () at server.c:2792
2792             listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
(gdb) |
2787         }
2788         server.db = zmalloc(sizeof(redisDb)*server.dbnum);
2789
2790         /* Open the TCP listening socket for the user commands. */
2791         if (server.port != 0 &&
2792             listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
2793             exit(1);
2794         if (server.tls_port != 0 &&
2795             listenToPort(server.tls_port,server.tlsfd,&server.tlsfd_count) == C_ERR)
2796             exit(1);
(gdb)

```

断点停在第 2792 行，输入 list 命令时，会显示第 2792 行前后的 10 行代码。

再次输入 list 命令试一下，则往后查阅代码：

```

(gdb) |
2797
2798         /* Open the listening Unix domain socket. */
2799         if (server.unixsocket != NULL) {
2800             unlink(server.unixsocket); /* don't care if this fails */
2801             server.sofd = anetUnixServer(server.neterr,server.unixsocket,
2802                 server.unixsocketperm, server.tcp_backlog);
2803             if (server.sofd == ANET_ERR) {
2804                 serverLog(LL_WARNING, "Opening Unix socket: %s", server.neterr);
2805                 exit(1);
2806             }
(gdb) |

```

```

2807         anetNonBlock(NULL,server.sofd);
2808     }
2809
2810     /* Abort if there are no listening sockets at all. */
2811     if (server.ipfd_count == 0 && server.tlsfd_count == 0 && server.sofd < 0) {
2812         serverLog(LL_WARNING, "Configured to not listen anywhere, exiting.");
2813         exit(1);
2814     }
2815
2816     /* Create the Redis databases, and initialize other internal state. */
(gdb)

```

代码继续往后显示 10 行，也就是说，第一次输入 **list** 命令会显示断点处前后的代码，继续输入 **list** 指令会以递增行号的形式继续显示剩下的代码行，一直到文件结束为止。当然 **list** 指令还可以往前和往后显示代码，命令分别是“**list+ (加号)**”和“**list- (减号)**”：

```

(gdb) l -
2797
2798     /* Open the listening Unix domain socket. */
2799     if (server.unixsocket != NULL) {
2800         unlink(server.unixsocket); /* don't care if this fails */
2801         server.sofd = anetUnixServer(server.neterr,server.unixsocket,
2802             server.unixsocketperm, server.tcp_backlog);
2803         if (server.sofd == ANET_ERR) {
2804             serverLog(LL_WARNING, "Opening Unix socket: %s", server.neterr);
2805             exit(1);
2806         }
(gdb) l -
2787     }
2788     server.db = zmalloc(sizeof(redisDb)*server.dbnum);
2789
2790     /* Open the TCP listening socket for the user commands. */
2791     if (server.port != 0 &&
2792         listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
2793         exit(1);
2794     if (server.tls_port != 0 &&
2795         listenToPort(server.tls_port,server.tlsfd,&server.tlsfd_count) == C_ERR)
2796         exit(1);

```

可以在试试 **list +20** 和 **list -20**。

4.5.7 print 和 ptype 命令

通过 `print` 命令（简写为 `p`）我们可以在调试过程中方便地查看变量的值，也可以修改当前内存中的变量值。切换当前断点到堆栈 **#4**，然后打印以下三个变量。

```
(gdb) f 4
#4  0x0000555555559c1e in initServer () at server.c:2792
2792         listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
(gdb) l
2787     }
2788     server.db = zmalloc(sizeof(redisDb)*server.dbnum);
2789
2790     /* Open the TCP listening socket for the user commands. */
2791     if (server.port != 0 &&
2792         listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
2793         exit(1);
2794     if (server.tls_port != 0 &&
2795         listenToPort(server.tls_port,server.tlsfd,&server.tlsfd_count) == C_ERR)
2796         exit(1);
(gdb) p server.port
$1 = 6379
(gdb) p server.ipfd
$2 = {0 <repeats 16 times>}
(gdb) p server.ipfd_count
$3 = 0
```

这里使用 `print` 命令分别打印出 `server.port`、`server.ipfd`、`server.ipfd_count` 的值，其中 `server.ipfd` 显示 “{0 \<repeats 16 times\>}”，这是 GDB 显示字符串或字符数据特有的方式，当一个字符串变量或者字符数组或者连续的内存值重复若干次，GDB 就会以这种模式来显示以节约空间。

print 命令不仅可以显示变量值，也可以显示进行一定运算的表达式计算结果值，甚至可以显示一些函数的执行结果值。

举个例子，我们可以输入 `p &server.port` 来输出 `server.port` 的地址值，如果在 C++ 对象中，可以通过 `p this` 来显示当前对象的地址，也可以通过 `p *this` 来列出当前对象的各个成员变量值，如果有三个变量可以相加（假设变量名分别叫 `a`、`b`、`c`），可以使用 `p a + b + c` 来打印这三个变量的结果值。

假设 `func()` 是一个可以执行的函数，`p func()` 命令可以输出该变量的执行结果。举一个最常用的例子，某个时刻，某个系统函数执行失败了，通过系统变量 `errno` 得到一个错误码，则可以使用 `p strerror(errno)` 将这个错误码对应的文字信息打印出来，这样就不用费劲地去 `man` 手册上查找这个错误码对应的错误含义了。

`print` 命令不仅可以输出表达式结果，同时也可以修改变量的值，我们尝试将上文中的端口号从 6379 改成 6400 试试：

```
(gdb) p server.port=6400
$24 = 6400
(gdb) p server.port
$25 = 6400
(gdb)
```

总结起来，利用 `print` 命令，我们不仅可以查看程序运行过程中的各个变量的状态值，也可以通过临时修改变量的值来控制程序的行为。

4.5.8 ptype 命令

`ptype`，顾名思义，其含义是“print type”，就是输出一个变量的类型。例如，我们试着输出 Redis 堆栈 #4 的变量 `server` 和变量 `server.port` 的类型：

```
(gdb) ptype server
type = struct redisServer {
    pid_t pid;
    char *configfile;
    char *executable;
    char **exec_argv;
    int hz;
    redisDb *db;
    ...省略部分字段...
(gdb) ptype server.port
type = int
```

可以看到，对于一个复合数据类型的变量，`ptype` 不仅列出了这个变量的类型（这里是一个名叫 `redisServer` 的结构体），而且详细地列出了每个成员变量的字段名，方便我们去查看每个变量的类型定义。

4.5.9 info 和 thread 命令

在前面使用 `info break` 命令查看当前断点时介绍过，`info` 命令是一个复合指令，还可以用来查看当前进程的所有线程运行情况。下面以 `redis-server` 进程为例来演示一下，

使用 `delete` 命令删掉所有断点，然后使用 `run` 命令重启一下 `redis-server`，等程序正常启动后，我们按快捷键 `Ctrl+C` 中断程序，然后使用 `info thread` 命令来查看当前进程有哪些线程，分别中断在何处：

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/lqf/0voice/gdb/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
7653:C 02 Nov 15:33:27.959 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
7653:C 02 Nov 15:33:27.959 # Redis version=6.0.3, bits=64, commit=00000000, modified=0,
pid=7653, just started
7653:C 02 Nov 15:33:27.959 # Warning: no config file specified, using the default config. In order to
specify a config file use /home/lqf/0voice/gdb/redis-6.0.3/src/redis-server /path/to/redis.conf
7653:M 02 Nov 15:33:27.962 * Increased maximum number of open files to 10032 (it was originally
set to 1024).
[New Thread 0x7ffff69ff700 (LWP 7654)]
[New Thread 0x7ffff61fe700 (LWP 7655)]
[New Thread 0x7ffff59fd700 (LWP 7656)]
7653:M 02 Nov 15:33:27.965 * Ready to accept connections
^C
Thread 1 "redis-server" received signal SIGINT, Interrupt.
0x00007ffff7d3410f in epoll_wait (epfd=5, events=0x7ffff6ad7380, maxevents=10128,
timeout=100)
    at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
30 ../sysdeps/unix/sysv/linux/epoll_wait.c: No such file or directory.
(gdb) info threads
    Id      Target Id                                Frame
* 1      Thread 0x7ffff7c08f80 (LWP 23500) "redis-server" 0x00007ffff7d2810f in epoll_wait
(epfd=5, events=0x7ffff78de040, maxevents=10128, timeout=100)
    at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
 2      Thread 0x7ffff6624700 (LWP 23503) "bio_close_file" futex_wait_cancelable (
private=<optimized out>, expected=0, futex_word=0x555555b42708 <bio_newjob_cond+40>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
 3      Thread 0x7ffff5e23700 (LWP 23504) "bio_aof_fsync"  futex_wait_cancelable (
private=<optimized out>, expected=0, futex_word=0x555555b42738 <bio_newjob_cond+88>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
 4      Thread 0x7ffff5622700 (LWP 23505) "bio_lazy_free"  futex_wait_cancelable (
private=<optimized out>, expected=0, futex_word=0x555555b42768
<bio_newjob_cond+136>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
```

```
5 Thread 0x7fff4e21700 (LWP 23506) "jemalloc_bg_thd" futex_wait_cancelable (  
private=<optimized out>, expected=0, futex_word=0x7fff7a073b0)  
at ../sysdeps/unix/sysv/linux/futex-internal.h:88
```

通过 `info thread` 的输出可以知道 `redis-server` 正常启动后，一共产生了 5 个线程，包括一个主线程和四个工作线程，线程编号（`Id` 那一列）分别是 5、4、3、2、1。

注意 虽然第一栏的名称叫 `Id`，但第一栏的数值不是线程的 `Id`，第三栏括号里的内容（如 `LWP 23500`）中，23500 这样的数值才是当前线程真正的 `Id`。Light Weight Process（轻量级进程），即是我们所说的线程。

怎么知道线程哪个线程是主线程，现在有 5 个线程，也就有 5 个调用堆栈，如果此时输入 `backtrace` 命令查看调用堆栈，由于当前 `GDB` 作用在线程 1，因此 `backtrace` 命令显示的一定是线程 1 的调用堆栈：

```
(gdb) bt  
#0 0x00007ffff7d2810f in epoll_wait (epfd=5, events=0x7ffff78de040, maxevents=10128,  
timeout=100) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30  
#1 0x00005555555592e32 in aeApiPoll (eventLoop=0x7ffff780b480, tvp=0x7fffffe210)  
at ae_epoll.c:112  
#2 0x00005555555593c0d in aeProcessEvents (eventLoop=0x7ffff780b480, flags=27)  
at ae.c:447  
#3 0x00005555555593f18 in aeMain (eventLoop=0x7ffff780b480) at ae.c:539  
#4 0x000055555555a27ee in main (argc=1, argv=0x7fffffe3b8) at server.c:5175
```

由此可见，堆栈 #4 的 `main()` 函数也证实了上面的说法，即线程编号为 1 的线程是主线程。

如何切换到其他线程呢？可以通过“`thread` 线程编号”切换到具体的线程上去。例如，想切换到线程 2 上去，只要输入 `thread 2` 即可，然后输入 `bt` 就能查看这个线程的调用堆栈了：

```
(gdb) thread 2  
[Switching to thread 2 (Thread 0x7fff6624700 (LWP 23503))]  
#0 futex_wait_cancelable (private=<optimized out>, expected=0,  
futex_word=0x555555b42708 <bio_newjob_cond+40>)  
at ../sysdeps/unix/sysv/linux/futex-internal.h:88  
88 ../sysdeps/unix/sysv/linux/futex-internal.h: No such file or directory.
```

因此利用 `info thread` 命令就可以调试多线程程序，当然用 `GDB` 调试多线程程序还有一个很麻烦的问题，我们将在后面的 `GDB` 高级调试技巧中介绍。请注意，当把 `GDB`

当前作用的线程切换到线程 2 上之后，线程 2 前面就被加上了星号：

```
(gdb) info threads
  Id   Target Id                                     Frame
  1     Thread 0x7ffff7c08f80 (LWP 23500) "redis-server" 0x00007ffff7d2810f in
epoll_wait (epfd=5, events=0x7ffff78de040, maxevents=10128, timeout=100)
    at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
* 2     Thread 0x7ffff6624700 (LWP 23503) "bio_close_file" futex_wait_cancelable (
    private=<optimized out>, expected=0, futex_word=0x555555b42708
    <bio_newjob_cond+40>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
  3     Thread 0x7ffff5e23700 (LWP 23504) "bio_aof_fsync" futex_wait_cancelable (
    private=<optimized out>, expected=0, futex_word=0x555555b42738
    <bio_newjob_cond+88>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
  4     Thread 0x7ffff5622700 (LWP 23505) "bio_lazy_free" futex_wait_cancelable (
    private=<optimized out>, expected=0, futex_word=0x555555b42768
    <bio_newjob_cond+136>)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
  5     Thread 0x7ffff4e21700 (LWP 23506) "jemalloc_bg_thd" futex_wait_cancelable (
    private=<optimized out>, expected=0, futex_word=0x7ffff7a073b0)
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
```

info 命令还可以用来查看当前函数的参数值，组合命令是 info args，我们找个函数值多一点的堆栈函数来试一下：

```
(gdb) bt
#0  0x00007ffff7d2810f in epoll_wait (epfd=5, events=0x7ffff78de040, maxevents=10128,
    timeout=100) at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
#1  0x00005555555592e32 in aeApiPoll (eventLoop=0x7ffff780b480, tvp=0x7fffff210)
    at ae_epoll.c:112
#2  0x00005555555593c0d in aeProcessEvents (eventLoop=0x7ffff780b480, flags=27)
    at ae.c:447
#3  0x00005555555593f18 in aeMain (eventLoop=0x7ffff780b480) at ae.c:539
#4  0x000055555555a27ee in main (argc=1, argv=0x7fffff3b8) at server.c:5175
(gdb) info 2
Undefined info command: "2". Try "help info".
(gdb) f 2
#2  0x00005555555593c0d in aeProcessEvents (eventLoop=0x7ffff780b480, flags=27)
    at ae.c:447
```

```
447         numevents = aeApiPoll(eventLoop, tvp);  
(gdb) info args  
eventLoop = 0x7ffff780b480  
flags = 27  
(gdb)
```

上述代码片段切回至主线程 1，然后切换到堆栈 #2，堆栈 #2 调用处的函数是 `aeProcessEvents()`，一共有两个参数，使用 `info args` 命令可以输出当前两个函数参数的值，参数 `eventLoop` 是一个指针类型的参数，对于指针类型的参数，GDB 默认会输出该变量的指针地址值，如果想输出该指针指向对象的值，在变量名前面加上 `*` 解引用即可，这里使用 `p *eventLoop` 命令：

```
(gdb) p *eventLoop  
$7 = {maxfd = 7, setsize = 10128, timeEventNextId = 1, lastTime = 1593500832,  
      events = 0x7ffff7878340, fired = 0x7ffff78c9f40, timeEventHead = 0x7ffff7823190,  
      stop = 0, apidata = 0x7ffff7877be0, beforeSleep = 0x55555559a4ef <beforeSleep>,  
      afterSleep = 0x55555559a6da <afterSleep>, flags = 0}
```

如果还要查看其成员值，继续使用 变量名 -> 字段名 即可，在前面学习 `print` 命令时已经介绍过了，这里不再赘述。

上面介绍的是 `info` 命令最常用的三种方法，更多的方法使用 `help info` 查看。

4.5.10 next、step、until、finish、return 和 jump 命令

这几个命令是 GDB 调试程序时最常用的几个控制流命令，因此放在一起介绍。`next` 命令（简写为 `n`）是让 GDB 调到下一条命令去执行，这里的下一条命令不一定是代码的下一行，而是根据程序逻辑跳转到相应的位置。

4.5.10.1 next 命令

范例 `next.c`:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 0;
6      if (a == 7)
7      {
8          print("a is equal to 7.\n");
9      }
10
11     int b = 10;
12     print("b = %d.\n", b);
13     return 0;
14 }

```

如果当前 GDB 中断在上述代码第 6 行,此时输入 `next` 命令 GDB 将调到第 11 行,因为这里的 `if` 条件并不满足。

这里有一个小技巧,在 GDB 命令行界面如果直接按下回车键,默认是将最近一条命令重新执行一遍,因此,当使用 `next` 命令单步调试时,不必反复输入 `n` 命令,直接回车就可以了。

`next` 命令用调试的术语叫“单步步过”(step over),即遇到函数调用直接跳过,不进入函数体内部。而下面的 `step` 命令(简写为 `s`)就是“单步步入”(step into),顾名思义,就是遇到函数调用,进入函数内部。举个例子,在 `redis-server` 的 `main()` 函数中有个叫 `spt_init(argc, argv)` 的函数调用,当我们停在这一行时,输入 `s` 将进入这个函数内部。

```

// 除去不相关的干扰,代码有删除
int main(int argc, char **argv) {
    struct timeval tv;
    int j;
    /* We need to initialize our libraries, and the server configuration. */
    spt_init(argc, argv);
    setlocale(LC_COLLATE, "");
    zmalloc_set_oom_handler(redisOutOfMemoryHandler);
    srand(time(NULL)^getpid());
    gettimeofday(&tv, NULL);
    char hashseed[16];
    getRandomHexChars(hashseed, sizeof(hashseed));
    dictSetHashFunctionSeed((uint8_t*)hashseed);
}

```

```
server.sentinel_mode = checkForSentinelMode(argc,argv);
initServerConfig();
moduleInitModulesSystem();
//省略部分无关代码...
}
```

4.5.10.2 step 命令

使用 `b main` 命令在 `main()` 处加一个断点，然后使用 `r` 命令重新跑一下程序，会触发刚才加在 `main()` 函数处的断点，然后使用 `n` 命令让程序走到 `spt_init(argc, argv)` 函数调用处，再输入 `s` 命令就可以进入该函数了：

```
(gdb) b main
Breakpoint 6 at 0x5555555a1fa0: file server.c, line 4969.
(gdb) info b
Num      Type           Disp Enb Address              What
6        breakpoint    keep y   0x00005555555a1fa0 in main at server.c:4969
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
要先在 main 函数打断点 b main 再 r 重新启动
Breakpoint 6, main (argc=1, argv=0x7fffffffe3b8) at server.c:4969
4969  int main(int argc, char **argv) {
(gdb) |
4964      return ((!server.cluster_enabled && server.masterhost == NULL) ||
4965              (server.cluster_enabled && nodeIsMaster(server.cluster->myself)));
4966  }
4967
4968
4969  int main(int argc, char **argv) {
```

```

4970      struct timeval tv;
4971      int j;
4972
4973      #ifdef REDIS_TEST
(gdb) n
5001      spt_init(argc, argv);
(gdb) s
spt_init (argc=1, argv=0x7fffffffe3b8) at setproctitle.c:153
153      char **envp = environ;
(gdb) bt
#0  spt_init (argc=1, argv=0x7fffffffe3b8) at setproctitle.c:153
#1  0x000055555555a1fc0 in main (argc=1, argv=0x7fffffffe3b8) at server.c:5001
进入了 spt_init 函数

```

4.5.10.3 return 和 finish 命令

实际调试时，我们在某个函数中调试一段时间后，不需要再一步步执行到函数返回处，希望直接执行完当前函数并回到上一层调用处，就可以使用 `finish` 命令。与 `finish` 命令类似的还有 `return` 命令，`return` 命令的作用是结束执行当前函数，还可以指定该函数的返回值。

这里需要注意一下二者的区别：`finish` 命令会执行函数到正常退出该函数；而 `return` 命令是立即结束执行当前函数并返回，也就是说，如果当前函数还有剩余的代码未执行完毕，也不会执行了。

4.5.10.4 until 命令

实际调试时，还有一个 `until` 命令（简写为 `u`）可以指定程序运行到某一行停下来，还是以 `redis-server` 的代码为例：

```

2739 void initServer(void) {
2740     int j;
2741
2742     signal(SIGHUP, SIG_IGN);
2743     signal(SIGPIPE, SIG_IGN);
2744     setupSignalHandlers();
2745
2746     if (server.syslog_enabled) {
2747         openlog(server.syslog_ident,
2748                 server.syslog_facility);
2749     }

```

这是 redis-server 代码中 initServer() 函数的一个代码片段，位于文件 server.c 中，当停在第 2742 行，想直接跳到第 2746 行，可以直接输入 `u 2746`，这样就能快速执行完中间的代码。当然，也可以先在第 2746 行加一个断点，然后使用 `continue` 命令运行到这一行，但是使用 `until` 命令会更简便。

```
b initServer
```

```
u 2746
```

4.5.10.5 Jump 命令

jump 命令基本用法是：

`jump <location>`

- 该命令会带一个参数，即要跳转到的代码位置，可以是源代码的行号：

(gdb) `jump 555` #跳转到源代码的第 555 行的位置

- 可以是相对当前代码位置的偏移量：

(gdb) `jump +10` #跳转到距当前代码下 10 行的位置

- 也可以是代码所处的内存地址：

(gdb) `jump *0x12345678` #跳转到位于该地址的代码处

注意，在内存地址前面要加“*”。还有，`jump` 命令不会改变当前程序调用栈的内容，所以当你从一个函数跳到另一个函数时，当函数运行完返回进行退栈操作时就会发生错误，因此最好还是在同一个函数中进行跳转。

`location` 可以是程序的行号或者函数的地址，`jump` 会让程序执行流跳转到指定位

置执行，当然其行为也是不可控制的，例如您跳过了某个对象的初始化代码，直接执行操作该对象的代码，那么可能会导致程序崩溃或其他意外行为。`jump` 命令可以简写成 `j`，但是不可以简写成 `jmp`，其使用有一个注意事项，即如果 `jump` 跳转到的位置后续没有断点，那么 GDB 会执行完跳转处的代码会继续执行。举个例子：

```
1 int somefunc()
2 {
3     //代码 A
4     //代码 B
5     //代码 C
6     //代码 D
7     //代码 E
8     //代码 F
9 }
```

假设我们的断点初始位置在行号 3 处（代码 A），这个时候我们使用 `jump 6`，那么程序会跳过代码 B 和 C 的执行，执行完代码 D（跳转点），程序并不会停在代码 6 处，而是继续执行后续代码，因此如果我们想查看执行跳转处的代码后的结果，需要在行号 6、7 或 8 处设置断点。

有时候也可以用来测试一些我们想要执行的代码（正常逻辑不太可能跑到），比如
范例：jump.c

```

1  //jump.c
2  #include <stdio.h>
3  int main()
4  {
5      int a = 0;
6      if (a != 0)
7      {
8          printf("if condition\n");
9      }
10     else
11     {
12         printf("else condition\n");
13     }
14
15     return 0;
16 }

```

我们想执行 12 行的代码。

则

b main

jump 12

就会将 else 分支执行。

4.5.11 disassemble 命令

当进行一些高级调试时，我们可能需要查看某段代码的汇编指令去排查问题，或者是在调试一些没有调试信息的发布版程序时，也只能通过反汇编代码去定位问题，那么 disassemble 命令就派上用场了。

```

(gdb) bt
#0  initServer () at server.c:2746
#1  0x000055555555a264d in main (argc=1, argv=0x7ffffffe3b8) at server.c:5128
(gdb) disassemble
Dump of assembler code for function initServer:
0x0000555555559bfb4 <+0>:  push    %rbp
0x0000555555559bfb5 <+1>:  mov     %rsp,%rbp

```

```

0x000055555559bfb8 <+4>: push    %rbx
0x000055555559bfb9 <+5>: sub     $0x18,%rsp
0x000055555559bfbd <+9>: mov     $0x1,%esi
0x000055555559bfc2 <+14>: mov     $0x1,%edi
0x000055555559bfc7 <+19>: callq   0x55555558e8a0 <signal@plt>
0x000055555559bfcc <+24>: mov     $0x1,%esi
0x000055555559bfd1 <+29>: mov     $0xd,%edi
0x000055555559bfd6 <+34>: callq   0x55555558e8a0 <signal@plt>
0x000055555559bfdb <+39>: callq   0x5555555a16e0 <setupSignalHandlers>

```

4.5.12 set args 和 show args 命令

很多程序需要我们传递命令行参数。在 GDB 调试中，很多人会觉得可以使用 `gdb filename args` 这种形式来给 GDB 调试的程序传递命令行参数，这样是不行的。正确的做法是在用 GDB 附加程序后，在使用 `run` 命令之前，使用“`set args` 参数内容”来设置命令行参数。

还是以 `redis-server` 为例，Redis 启动时可以指定一个命令行参数，它的默认配置文件位于 `redis-server` 这个文件的上一层目录，因此我们可以在 GDB 中这样传递这个参数：`set args ../redis.conf`（即文件 `redis.conf` 位于当前程序 `redis-server` 的上一层目录），可以通过 `show args` 查看命令行参数是否设置成功。

```

(gdb) set args ../redis.conf
(gdb) show args
Argument list to give program being debugged when it is started is "../redis.conf".
(gdb)

```

如果单个命令行参数之间含有空格，可以使用引号将参数包裹起来。

```

(gdb) set args "999 xx" "hu jj"
(gdb) show args
Argument list to give program being debugged when it is started is "\"999 xx\" \"hu jj\"".
(gdb)

```

如果想清除掉已经设置好的命令行参数，使用 `set args` 不加任何参数即可。

```

(gdb) set args

```

```
(gdb) show args
```

```
Argument list to give program being debugged when it is started is "".
```

```
(gdb)
```

4.5.13 **tbreak** 命令

tbreak 命令也是添加一个断点，第一个字母“t”的意思是 **temporarily**（临时的），也就是说这个命令加的断点是临时的，所谓临时断点，就是一旦该断点触发一次后就会自动删除。添加断点的方法与上面介绍的 **break** 命令一模一样，这里不再赘述。

这里文档就不再描述，大家自行测试即可。

4.5.14 **watch** 命令

watch 命令是一个强大的命令，它可以用来监视一个变量或者一段内存，当这个变量或者该内存处的值发生变化时，GDB 就会中断下来。被监视的某个变量或者某个内存地址会产生一个 **watch point**（观察点）。

watch 命令的使用方式是“**watch** 变量名或内存地址”，一般有以下几种形式：

■ 形式一：整型变量

```
int i;
```

```
watch i
```

■ 形式二：指针类型

```
char *p;
```

```
watch p 与 watch *p
```

注意：**watch p** 与 **watch *p** 是有区别的，前者是查看 ***(&p)**，是 **p** 变量本身；后者是 **p** 所指内存的内容。我们需要查看地址，因为目的是要看某内存地址上的数据是怎样变化的。

■ 形式三：**watch** 一个数组或内存区间

```
char buf[128];
```

```
watch buf
```

这里是对 **buf** 的 128 个数据进行了监视，此时不是采用硬件断点，而是用软中断实现的。用软中断方式去检查内存变量是比较耗费 CPU 资源的，精确地指明地址是硬件中断。

注意：当设置的观察点是一个局部变量时，局部变量无效后，观察点也会失效。在观察点失效时 GDB 可能会提示如下信息：

Watchpoint 2 deleted because the program has left the block in which its expression is valid.

文件名 watch.c

```
#include <stdio.h>
#include <string.h>

char mem[8];
char buf[128];

void initBuf(char *pBuf)
{
    int i, j;
    mem[0] = '0';
    mem[1] = '1';
    mem[2] = '2';
    mem[3] = '3';
    mem[4] = '4';
    mem[5] = '5';
    mem[6] = '6';
    mem[7] = '7';
    //ascii table first 32 is not printable
    for (i = 2; i < 8; i++)
    {
        for (j = 0; j < 16; j++)
            pBuf[i * 16 + j] = i * 16 + j;
    }
}

void prtBuf(char *pBuf)
{
    int i, j;
    for (i = 2; i < 8; i++)
    {
        for (j = 0; j < 16; j++)
            printf("%c ", pBuf[i * 16 + j]);
        printf("\n");
    }
}

int main()
```

```

{
    initBuf(buf);
    prtBuf(buf);
    return 0;
}

```

测试

```

$ gdb ./watch
GNU gdb (Ubuntu 8.2-0ubuntu1) 8.2

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./watch...done.
(gdb) b main
Breakpoint 1 at 0x1248: file watch.c, line 39.
(gdb) watch mem
Hardware watchpoint 2: mem
(gdb) c
The program is not being run.
(gdb) r
Starting program: /mnt/hgfs/ubuntu/vip/20200630-gdb/src/gdb/watch

Breakpoint 1, main () at watch.c:39
39      initBuf(buf);
(gdb) c
Continuing.

Hardware watchpoint 2: mem

Old value = "\000\000\000\000\000\000\000"
New value = "0\000\000\000\000\000\000"
initBuf (pBuf=0x55555558060 <buf> "") at watch.c:11
11      mem[1] = '1';
(gdb) c
Continuing.

Hardware watchpoint 2: mem

Old value = "0\000\000\000\000\000\000"

```

```

New value = "01\000\000\000\000\000"
initBuf (pBuf=0x555555558060 <buf> "") at watch.c:12
12      mem[2] = '2';
(gdb) c
Continuing.

Hardware watchpoint 2: mem

Old value = "01\000\000\000\000\000"
New value = "012\000\000\000\000\000"
initBuf (pBuf=0x555555558060 <buf> "") at watch.c:13
13      mem[3] = '3';
(gdb) c
Continuing.

.....省略部分

Hardware watchpoint 2: mem

Old value = "012345\000"
New value = "0123456"
initBuf (pBuf=0x555555558060 <buf> "") at watch.c:17
17      mem[7] = '7';
(gdb)

```

watch i 的问题：是可以同时去 **watch**，只是局部变量需要进入到相应的起作用范围才能 **watch**。比如 **initBuf** 函数的 **i**。

问题来了，如果要取消 **watch** 怎么办？

先用 **info watch** 查看 **watch** 的变量，然后根据编号使用 **delete** 删除相应的 **watch** 变量。

```

(gdb) info watch

```

| Num | Type | Disp | Enb | Address | What |
|-----|---------------|------|-----|---------|------|
| 3 | hw watchpoint | keep | y | | mem |

```

(gdb) delete 3

```

4.5.15 display 命令

display 命令监视的变量或者内存地址，每次程序中断下来都会自动输出这些变量

或内存的值。例如，假设程序有一些全局变量，每次断点停下来我都希望 GDB 可以自动输出这些变量的最新值，那么使用“display 变量名”设置即可。

```
Program received signal SIGINT, Interrupt.
0x00007ffff71e2483 in epoll_wait () from /lib64/libc.so.6
(gdb) display $ebx
1: $ebx = 7988560
(gdb) display /x $ebx
2: /x $ebx = 0x79e550
(gdb) display $eax
3: $eax = -4
(gdb) b main
Breakpoint 8 at 0x4201f0: file server.c, line 4696.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/lqf/0voice/redis/redis-6.0.3/src/redis-server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 8, main (argc=1, argv=0x7fffffe4e8) at server.c: 4969
4003      int main(int argc, char **argv) {
3: $eax = 4325872
2: /x $ebx = 0x0
1: $ebx = 0
(gdb)
```

上述代码中，使用 `display` 命令分别添加了寄存器 `ebp` 和寄存器 `eax`，`ebp` 寄存器分别使用十进制和十六进制两种形式输出其值，这样每次程序中断下来都会自动把这些值打印出来，可以使用 `info display` 查看当前已经自动添加了哪些值，使用 `delete display` 清除全部需要自动输出的变量，使用 `delete display 编号` 删除某个自动输出的变量。

```
(gdb) delete display
Delete all auto-display expressions? (y or n) n
```



```
(gdb) delete display 3
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
2:   y  $ebp
1:   y  $eax
```

4.6 调试技巧

4.6.1 将 `print` 打印结果显示完整

当使用 `print` 命令打印一个字符串或者字符数组时，如果该字符串太长，`print` 命令默认显示不全的，我们可以通过在 GDB 中输入 `set print element 0` 命令设置一下，这样再次使用 `print` 命令就能完整地显示该变量的所有字符串了。

4.6.2 多线程下禁止线程切换

假设现在有 5 个线程，除了主线程，工作线程都是下面这样的一个函数：

```
void thread_proc(void* arg)
{
    //代码行 1
    //代码行 2
    //代码行 3
    //代码行 4
    //代码行 5
    //代码行 6
    //代码行 7
    //代码行 8
    //代码行 9
    //代码行 10
    //代码行 11
    //代码行 12
    //代码行 13
    //代码行 14
    //代码行 15
}
```

为了能说清楚这个问题，我们把四个工作线程分别叫做 A、B、C、D。

假设 GDB 当前正在处于线程 A 的代码行 3 处，此时输入 `next` 命令，我们期望的是调试器跳到代码行 4 处；或者使用“u 代码行 10”，那么我们期望输入 `u` 命令后调试器可以跳转到代码行 10 处。

但是在实际情况下，GDB 可能会跳转到代码行 1 或者代码行 2 处，甚至代码行 13、代码行 14 这样的地方也是有可能的，这不是调试器 bug，这是多线程程序的特点，当我们从代码行 4 处让程序 `continue` 时，线程 A 虽然会继续往下执行，但是如果此时系统的线程调度将 CPU 时间片切换到线程 B、C 或者 D 呢？那么程序最终停下来时，处于代码行 1 或者代码行 2 或者其他地方就不奇怪了，而此时打印相关的变量值，可能就不是我们需要的线程 A 的相关值。

为了解决调试多线程程序时出现的这种问题，GDB 提供了一个在调试时将程序执行流锁定在当前调试线程的命令：`set scheduler-locking on`。当然也可以关闭这一选项，使用 `set scheduler-locking off`。除了 `on/off` 这两个值选项，还有一个不太常用的值叫 `step`，这里就不介绍了。

4.6.3 条件断点

在实际调试中，我们一般会用到三种断点：普通断点、条件断点和硬件断点。

硬件断点又叫数据断点，这样的断点其实就是前面课程中介绍的用 `watch` 命令添加的部分断点（为什么是部分而不是全部，前面介绍原因了，`watch` 添加的断点有部分是通过软中断实现的，不属于硬件断点）。硬件断点的触发时机是监视的内存地址或者变量值发生变化。

普通断点就是除去条件断点和硬件断点以外的断点。

下面重点来介绍一下条件断点，所谓条件断点，就是满足某个条件才会触发的断点，这里先举一个直观的例子：

```
void do_something_func(int i)
{
    i++;
    i = 100 * i;
}

int main()
{
    for(int i = 0; i < 10000; ++i)
```

```

{
    do_something_func(i);
}

return 0;
}

```

在上述代码中，假如我们希望当变量 `i=5000` 时，进入 `do_something_func()` 函数追踪一下这个函数的执行细节。此时可以修改代码增加一个 `i=5000` 的 `if` 条件，然后重新编译链接调试，这样显然比较麻烦，尤其是对于一些大型项目，每次重新编译链接都需要花一定的时间，而且调试完了还得把程序修改回来。

有了条件断点就不需要这么麻烦了，添加条件断点的命令是 `break [lineNo] if [condition]`，其中 `lineNo` 是程序触发断点后需要停下的位置，`condition` 是断点触发的条件。这里可以写成 `break 11 if i==5000`，其中，`11` 就是调用 `do_something_func()` 函数所在的行号。当然这里的行号必须是合理行号，如果行号非法或者行号位置不合理也不会触发这个断点。

```

(gdb) break 11 if i==5000
Breakpoint 2 at 0x400514: file test1.c, line 10.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/testgdb/test1

Breakpoint 1, main () at test1.c:9
9          for(int i = 0; i < 10000; ++i)
(gdb) c
Continuing.

Breakpoint 2, main () at test1.c:11
11          do_something_func(i);
(gdb) p i
$1 = 5000

```

把 `i` 打印出来，GDB 确实是在 `i=5000` 时停下来了。

添加条件断点还有一个方法就是先添加一个普通断点，然后使用“`condition` 断点编号断点触发条件”这样的方式来添加。添加一下上述断点：

```

(gdb) b 11
Breakpoint 1 at 0x400514: file test1.c, line 11.
(gdb) info b
Num      Type      Disp Enb Address          What
1        breakpoint keep y   0x0000000000400514 in main at test1.c:11
(gdb) condition 1 i==5000

```

```
(gdb) r
Starting program: /root/testgdb/test1
y

Breakpoint 1, main () at test1.c:11
11          do_something_func(i);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7_4.2.x86_64
(gdb) p i
$1 = 5000
(gdb)
```

4.6.4 使用 GDB 调试多进程程序

这里说的多进程程序指的是一个进程使用 Linux 系统调用 `fork()` 函数产生的子进程，没有相互关联的进程就是普通的 GDB 调试，不必刻意讨论。

在实际的应用中，如有这样一类程序，如 **Nginx**，对于客户端的连接是采用多进程模型，当 **Nginx** 接受客户端连接后，创建一个新的进程来处理这一路连接上的信息来往，新产生的进程与原进程互为父子关系，那么如何用 GDB 调试这样的父子进程呢？一般有两种方法：

（1）用 GDB 先调试父进程，等子进程 `fork` 出来后，使用 `gdb attach` 到子进程上去，当然这需要重新开启一个 `session` 窗口用于调试，`gdb attach` 的用法在前面已经介绍过了；

（2）GDB 调试器提供了一个选项叫 `follow-fork`，可以使用 `show follow-fork mode` 查看当前值，也可以通过 `set follow-fork mode` 来设置是当一个进程 `fork` 出新的子进程时，GDB 是继续调试父进程还是子进程（取值是 `child`），默认是父进程（取值是 `parent`）。

```
(gdb) show follow-fork mode
Debugger response to a program call of fork or vfork is "parent".
(gdb) set follow-fork child
(gdb) show follow-fork mode
Debugger response to a program call of fork or vfork is "child".
(gdb)
```

5. GDB TUI——在 GDB 中显示程序源码

官方参考文档: <http://sourceware.org/gdb/onlinedocs/gdb/TUI.html>

5.1 开启 GDB TUI 模式

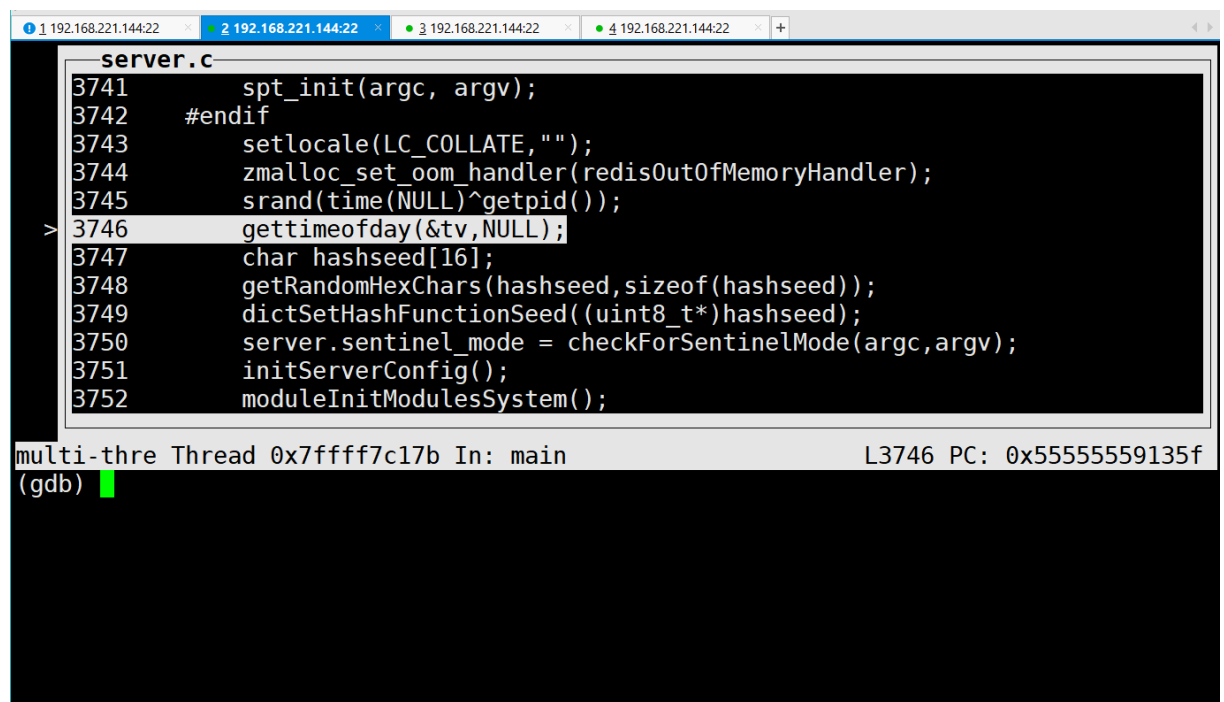
开启 GDB TUI 模式有两个方法。

方法一: 使用 `gdbtui` 命令或者 `gdb-tui` 命令开启一个调试。

`gdbtui -q` 需要调试的程序名

方法二: 直接使用 GDB 调试代码, 在需要的时候使用切换键 **Ctrl + x**, 然后按 **a** (两个步骤是连在一起执行), 进入常规 GDB 和 GDB TUI 的来回切换。

5.2 GDB TUI 模式常用窗口

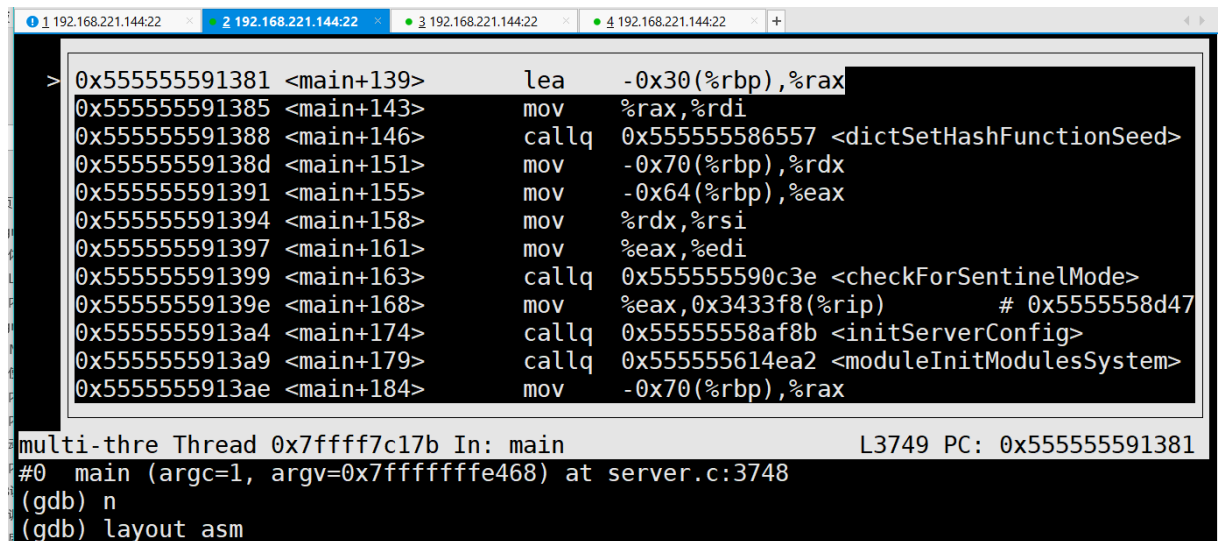


The screenshot shows the GDB TUI interface. The main window displays the source code of `server.c` with line numbers 3741 to 3752. Line 3746 is highlighted, showing the function `gettimeofday(&tv, NULL);`. Below the source code, a status bar indicates the current thread is `multi-thre Thread 0x7ffff7c17b In: main` and the program counter is `L3746 PC: 0x55555559135f`. At the bottom, there is a command window labeled `(gdb)` with a green cursor.

默认情况下, GDB TUI 模式会显示 `command` 窗口和 `source` 窗口, 如上图所示, 还有其他窗口, 如下列举的四个常用的窗口:

- (cmd) command 命令窗口, 可以输入调试命令
- (src) source 源代码窗口, 显示当前行、断点等信息
- (asm) assembly 汇编代码窗口
- (reg) register 寄存器窗口

可以通过“layout + 窗口类型”命令来选择自己需要的窗口，例如，在 cmd 窗口输入 layout asm 则可以切换到汇编代码窗口。



```
> 0x55555591381 <main+139>    lea    -0x30(%rbp),%rax
0x55555591385 <main+143>    mov    %rax,%rdi
0x55555591388 <main+146>    callq 0x55555586557 <dictSetHashFunctionSeed>
0x5555559138d <main+151>    mov    -0x70(%rbp),%rdx
0x55555591391 <main+155>    mov    -0x64(%rbp),%eax
0x55555591394 <main+158>    mov    %rdx,%rsi
0x55555591397 <main+161>    mov    %eax,%edi
0x55555591399 <main+163>    callq 0x55555590c3e <checkForSentinelMode>
0x5555559139e <main+168>    mov    %eax,0x3433f8(%rip)    # 0x5555558d47
0x555555913a4 <main+174>    callq 0x5555558af8b <initServerConfig>
0x555555913a9 <main+179>    callq 0x555555614ea2 <moduleInitModulesSystem>
0x555555913ae <main+184>    mov    -0x70(%rbp),%rax

multi-thre Thread 0x7ffff7c17b In: main          L3749 PC: 0x55555591381
#0  main (argc=1, argv=0x7ffffffe468) at server.c:3748
(gdb) n
(gdb) layout asm
```

layout 命令还可以用来修改窗口布局，在 cmd 窗口中输入 help layout，常见的有：

```
Usage: layout prev | next | <layout_name>
Layout names are:
src    : Displays source and command windows.
asm    : Displays disassembly and command windows.
split : Displays source, disassembly and command windows.
regs   : Displays register window. If existing layout
         is source/command or assembly/command, the
         register window is displayed. If the
         source/assembly/command (split) is displayed,
         the register window is displayed with
         the window that has current logical focus.
```

另外，可以通过 winheight 命令修改各个窗口的大小，如下所示：

```
(gdb) help winheight
Set the height of a specified window.
Usage: winheight <win_name> [+ | -] <#lines>
Window names are:
src    : the source window
cmd    : the command window
asm    : the disassembly window
regs   : the register display

##将代码窗口的高度扩大 5 行代码
winheight src + 5
##将代码窗口的高度减小 4 代码
```

5.3 常用快捷键

C-代码 Ctrl 键，下面介绍的皆为组合键，比如：

C-x a，即是先按 **Ctrl + x** 两个键，然后再去按 **a** 键。

| 组合键 | 说明 |
|---------------|---|
| C-x a 或 C-x A | 进入或退出 TUI 模式 |
| C-x 1 | 这里是数字 1，仅在一个窗口中使用 TUI 布局 |
| C-x 2 | 使用至少两个窗口的 TUI 布局。 |
| C-x o | 切换活动窗口，当窗口被选中时则可以使用上下左右键，上翻、下翻键进行查看信息查看，刷新则用 Ctrl+L（信息显示混乱是使用该组合键进行刷新）。 |
| C-x s | 单键模式和 TUI 模式的切换。 单键模式可以直接通过快捷键调试程序。 |

注：单键模式的调试提供了常用的调试快捷键。

单键模式对应的快捷键：

| 快捷键 | 说明 |
|-----|---|
| c | 继续执行 |
| d | 向下层查看堆栈 |
| u | 向上层查看堆栈 |
| f | 执行完当前函数后返回，和常规 gdb 的 finish 命令一致 |
| n | 和常规 gdb 的 next 命令一致。 |
| s | 和常规 gdb 的 step 命令一致。 |
| i | stepi. The shortcut letter 'i' stands for "step Into" |
| o | nexti. The shortcut letter 'o' stands for "step Over" |
| r | 和常规 gdb 的 run 命令一致。 |

| | |
|---|-----------|
| v | 查看局部变量 |
| w | 提示当前调试的位置 |
| q | 退出单键模式 |

5.4 窗口焦点切换

| 快捷键方式 | 说明 |
|-------|------|
| C-x o | 下一窗口 |

| 输入命令方式 | 说明 |
|-------------------|-------|
| focus src 或 fs s | 源码窗口 |
| focus cmd 或 fs c | 命令窗口 |
| focus asm 或 fs a | 汇编窗口 |
| focus regs 或 fs r | 寄存器窗口 |
| focus next 或 fs n | 下一窗口 |
| focus prev 或 fs p | 上一窗口 |

6. GCC 优化级别

gcc 默认提供了 5 级优化选项：

■ **-O/-O0**:无优化(默认)

■ **-O1**:使用能减少目标文件大小以及执行时间并且不会使编译时间明显增加的优化。该模式在编译大型程序的时候会花费更多的时间和内存。在**-O1**下：编译会尝试减少代码体积和代码运行时间，但是并不执行会花费大量时间的优化操作。

■ **-O2**: 包含**-O1** 的优化并增加了不需要在目标文件大小和执行速度上进行折衷的优化。GCC 执行几乎所有支持的操作但不包括空间和速度之间权衡的优化，编译器不执行循环展开以及函数内联。这是推荐的优化等级，除非你有特殊的需求。**-O2** 会比**-O1** 启用多一些标记。与**-O1** 比较该优化**-O2** 将会花费更多的编译时间当然也会生成性能更好的代码。

■ **-Os**:专门优化目标文件大小,执行所有的不增加目标文件大小的**-O2** 优化选项。同时**-Os** 还会执行更加优化程序空间的选项。这对于磁盘空间极其紧张或者 CPU 缓存较小的机器非常有用。但也可能产生些许问题，因此软件树中的大部分 **ebuild** 都过滤掉这个等级的优化。使用**-Os** 是不推荐的。

■ **-O3**: 打开所有**-O2** 的优化选项并且增加 **-finline-functions**, **-funswitch-loops**, **-fpredictive-commoning**, **-fgcse-after-reload** and **-ftree-vectorize** 优化选项。这是最高最危险的优化等级。用这个选项会延长编译代码的时间，并且在使用 **gcc4.x** 的系统里不应全局启用。自从 **3.x** 版本以来 **gcc** 的行为已经有了极大地改变。在 **3.x**，**-O3** 生成的代码也只是比**-O2** 快一点点而已，而 **gcc4.x** 中还未必更快。用**-O3** 来编译所有的软件包将产生更大体积更耗内存的二进制文件，大大增加编译失败的机会或不可预知的程序行为（包括错误）。这样做将得不偿失，记住过犹不及。在 **gcc 4.x** 中使用**-O3** 是不推荐的。

7. 参考文档

官方参考文档: <http://valgrind.org/docs/manual/QuickStart.html>

官方参考文档: <http://www.gnu.org/software/gdb/documentation/>

官方参考文档: <http://sourceware.org/gdb/onlinedocs/gdb/TUI.html>

GDB 常用命令 https://blog.csdn.net/Roland_Sun/article/details/42460663