Linux环境代码调试(摘录&笔记)

Author:BG2EDG
Time:2022-01-30

写在前面

本文档主要参考《C/C++代码调试的艺术》一书，将其中针对Linux环境的调试方法进行整理，并增加了一些扩展内容。文档中绝大多数指令都实际运行并进行了纠错，并将原书中一些例程进行修改，以适配队内特点，建议配合使用。该文档仍在开发，最终目的是形成一套完整教程并纳入培训体现，希望大家发现问题或有好的内容分享及时pull request。

Part 1 GDB调试

概念辨析

学习Linux开发知识时，经常会出现GNU，GCC等一些长相相近的名词，这里先对我们使用的工具名词概念和相互关系进行辨析。

- GNU
  GNU是一个自由软件操作系统，包括GNU软件包（专门由GNU工程发布的程序）和由第三方发布的自由软件。GNU有一个很酷的名字和一个很宏伟的理想，如果想深入了解其背后的故事请搜索“Richard Matthew Stallman”，或者到GNU官网)了解。
- GCC
  GCC全称为the GNU Compiler Collection，意为“GNU编译工具集”，包含为GNU系统提供的编译器、链接器、组装器等一系列工具，可用于编译 C, C++, Objective-C等语言编写的程序。详情可到GCC官网l了解。
- gcc&g++
  gcc是GNU C Compiler的缩写，意为GNU C语言编译器，g++则指GNU C++语言编译器。注意此处的gcc是小写，含义和大写的GCC有很大不同。从本质上讲，gcc和g++并不是真正的编译器，二者只是GCC里面的两个工具，在编译C/C++程序时，它们会调用真正的编译器对代码进行编译。二者的区别如下表：

区别	gcc	g++
文件后缀处理方式	.c作为C程序，.cpp作为C++程序	.c和.cpp均作为C++程序
链接方式	/	自动链接C++库
预处理宏	/	会自动添加一些预处理宏

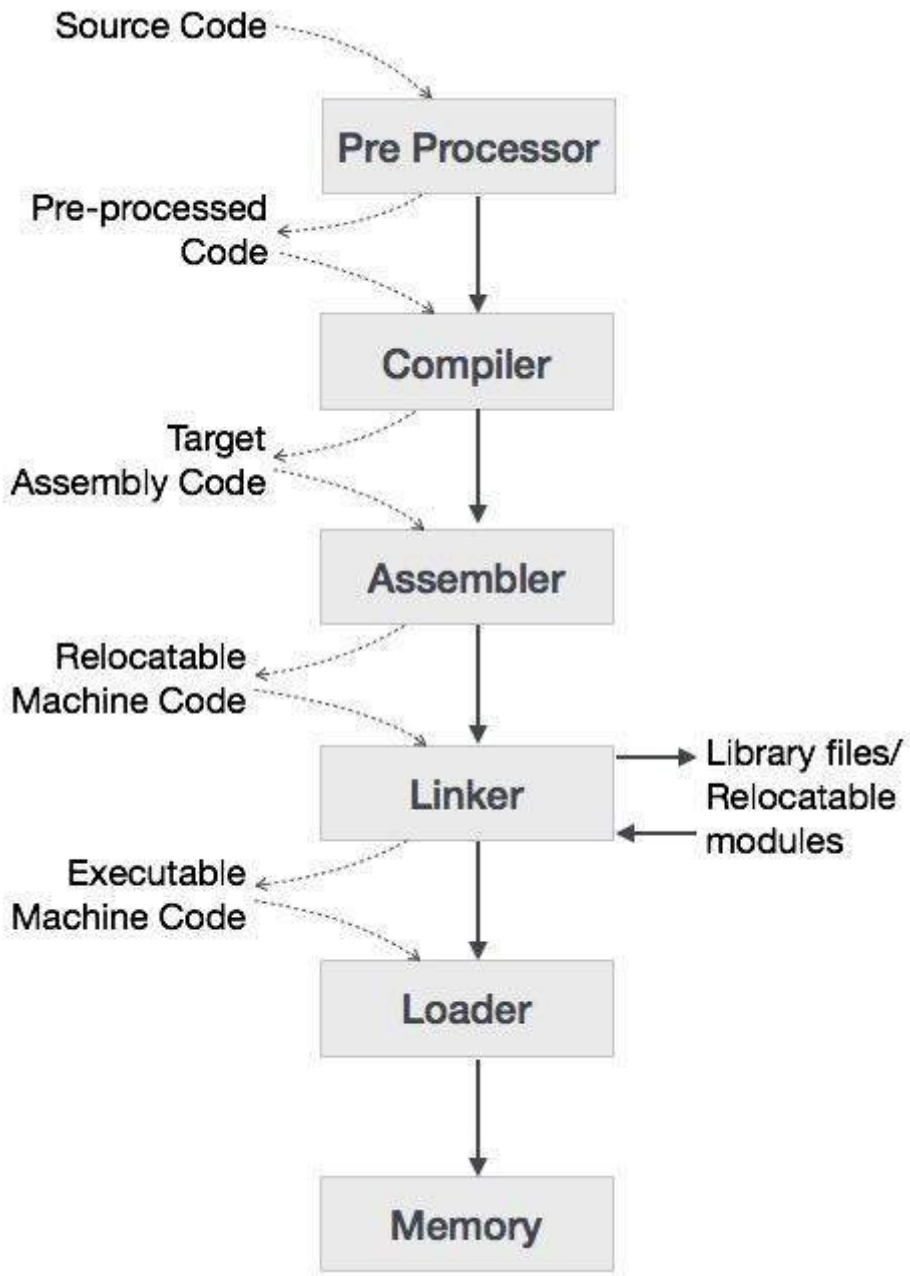
- GDB
  GDB(the GNU Project debugger)为GNU调试器，是本篇文档的主角之一。GDB可以做四件事，总结为“起止查换”，详细解释见下表：

four	things
起	设置可能改变行为的初始条件(Start your program, specifying anything that might affect its behavior)
止	让程序在特定条件下停止(Make your program stop on specified conditions)
查	查看程序停止时曾发生过什么(Examine what has happened, when your program has stopped)
换	更换条件观察bug对行为的影响(Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another)

(无关宏旨的拓展知识) GDB主要使用ptrace来调试程序，它是Linux中用于进程跟踪的一个系统调用。ptrace可以控制子进程，包括读写子进程的内存、寄存器、控制子进程的执行等等。若要在C/C++代码中调用ptrace()函数，可添加```#include <sys/ptrace.h>```包含头文件。
更多GDB知识可到[GDB官网](http://www.sourceware.org/gdb/)了解。

关系说明

首先通过一张图介绍C/C++程序从编写到执行需要经历的几个阶段：



- 预处理器(preprocessor)对由C/C++编写的源代码（高阶语言），完成宏处理，文件包含，语言扩展等功能。
- gcc/g++编译器(compiler)将程序编译成汇编程序（低阶语言）。
- 汇编器(assembler)将程序翻译成机器码对象(object)。
- 连接器(linker)将程序的不同部分连接成一个整体（可执行机器码）。
- 加载器(loader)加载到内存，程序开始执行。

编译和链接是靠gcc和g++编译器完成的，如果编译和链接阶段的源文件太多，相互之间的关系复杂，如果直接使用gcc/g++编译代码会很麻烦。Linux系统中的**Make**工具就是为了解决这一问题出现的，通过编写一个**Makefile**文件指定构建程序的方法，Make能依据Makefile批处理编译过程，生成可执行文件。然而Makefile的编写也很麻烦，于是**CMake**工具就出现了。通过编写一个**CMakeList**，cmake就能依据规则自动生成Makefile文件或project文件。总结一下这几个工具的关系：
CMakeList ->*输入*-> **CMake** ->*输出*-> **Makefile**
->*输入*-> **Make** ->*调度*-> **gcc/g++** ->*生成* ->**可执行文件**

GDB常用功能

支持的功能	描述
调试执行	逐语句、逐过程执行
显示、搜索源代码	查看、查找源代码信息
断点管理	设置断点、查看断点等
查看数据	在调试状态下查看变量数据、内存数据等
运行时修改变量值	在调试状态下修改某个变量的值
调用堆栈管理	查看堆栈信息
线程管理	调试多线程程序，查看线程信息
进程管理	调试多个进程
崩溃转储(coredump)分析	分析coredump文件
多方式启动调试	用不同的方式调试进程，比如加载参数启动、附加到进程等

GDB使用

1. 启动/终止gdb

- 将代码编译获得二进制可执行文件，然后在其目录下运行 `gdb 程序名` 。
- 若程序已经开始运行，可将通过附加进程的方式对程序进行调试。新建一个终端，输入 `ps aux | grep 程序名` 找到程序的进程号pid，然后执行 `gdb attach pid` 将gdb附加到进程。若提示无权限，则在指令前加上 `sudo` 。
- 进入gdb后会先输出一大段提示，按回车后终端提示变成 **(gdb)**。gdb操作和终端很像，只要输入指令就可以执行相关操作。若要退出gdb运行指令 `q` 或 `quit` 。

2. 启动/终止程序

- 若程序运行需要传入参数，现在gdb中使用 `set args 参数0 参数1 ……` 指令设置输入参数。然后执行 `r` 或 `run` 启动程序。执行 `kill` 强制停止程序。
- 程序若处于中断状态，执行 `c` 或 `continue` 恢复程序运行，
- `finish` 用于子函数调试，在main()中使用无效。效果是在子函数内运行完剩余指令，然后回到调用该子函数的父函数位置停下。
- `step` 用于单步执行，逐行执行代码，遇到子函数时会进入该函数内部，并在子函数第一行代码处停止。
- `next` 用于逐过程执行，遇到子函数时不会进入，而是跳过执行后面一行。
- `jump` 执行跳转功能时要保证后面程序还能继续正常运行，不能任意跳转，否则会导致程序崩溃。

3. 断点

- 程序运行到==断点(breakpoints)==位置会暂停，称为命中。我们可以在程序暂停时查看或修改一些程序信息，进而判断程序行为是否符合预期。
- 设置断点的指令要在程序运行前执行，运行时无法设置断点。
- 设置断点位置的方式可以指定代码行号、函数名、指令地址，详见常用指令表-断点管理。

- ==对只有定义没被执行的函数打端点，由于找不到执行函数，会显示pending==
- ==break -1中的当前行指list光标的前一个==
- ==临时断点(tbreak)==只命中一次，然后会被自动删除，再次调用则不会再命中。
- 设置一次断点后会显示该断点的编号(从1开始)，可以使用`disable`或`enable`指令某编号的断点进行禁用或启用。
- 删除断点的方式有`clear`和`delete`二者的区别：`clear`命令受到当前栈帧制约，删除的是将要执行的下一处指令的断点；`delete`命令是全局的，不受栈帧影响。`clear`不能删除观察点和捕获点；`delete`则全可以删除。
- 有些BUG只有在某个变量值或者几个因素同时发生变化时出现，==观察点(watchpoints)==可以用来发现或定位该类型的bug。gdb中用`watch`指令添加观察点，且可以通过`delete`、`disable`和`enable`指令对观察点操作。
- watch需要配合使用，不能单独使用
- ==捕获点(catchpoint)==指程序在发生某件事时，gdb能捕获这些事件并使程序停止执行。

4. 查看信息

- 查看各种信息的指令包含`info`、`print`、`display`、`list`、`x`、`ptype`等几种，具体效果参考附表，建议自己探索一遍。
- `list`看源代码，每调用一次，查看区间会向后平移一个listsize。使用`b +偏移`指令时，当前行定义为当前代码区间后面一行（个人测试结论）。
- ptype的可选参数包含以下几种：
    - /r：以原始数据的方式显示，不会替代一些typedef定义
    - /m：查看类时，不显示类的方法，只显示成员变量
    - /M：显示类的方法（默认选型）
    - /t：不打印typedef数据
    - /o：打印结构体字段的偏移量和大小
- 寄存器==TODO==

5. 栈帧

栈帧这个名词涉及了一种叫栈的数据结构，以及计算机函数调用时机器的底层实现方法。数据结构的部分自行补习，我们这里只把栈帧的调用过程介绍给大家。

首先对栈帧的一些特点介绍。当程序进行函数调用时，这次的调用信息，如调用位置、调用函数参数、局部变量等内容，称为栈帧。每一个函数调用，产生一个栈帧；函数调用结束，栈帧也结束出栈。我们把所有栈帧组成的信息抽象成一种结构，称为调用栈。

这里通过一个简化例子说明栈帧和函数调用的关系，比如我有一个最简单的hello word程序，外层一个main()主函数，里面cout输出hello world。

```
#include <iostream>
using namespace std;
int main()
{
    cout << "hello world!";
    cout << endl;
    return 0;
}
```

- 程序开始运行时，调用了main()函数，此时只有他被调用，因此产生一个栈帧。
- 接着运行到第5行cout这个函数，调用cout时产生了第二个栈帧。此时main()函数还没结束，只是我们从main()的任务中跳到了他的子函数任务中，因此第一个栈帧没被销毁，仍占用存

储。所以，我们的调用栈里此时有两个栈帧。

- cout中肯定有其他函数实现运算符<<的运算，但我们先假设里面没再调用其他函数。把hello word处理完了，然后第二个栈帧就销毁出栈了。此时调用栈里只有一个栈帧。
- 下面来到第6行的cout这个函数，调用cout生成一个栈帧，顺序上记为第三个栈帧。在处理完`endl`之前，调用栈一直有2个栈帧。
- 当第二个cout处理完，第三个栈帧销毁。函数返回0正常退出后，第一个栈帧销毁。

GDB查看栈回溯命令`backtrace`，执行后效果如图，包含了程序执行到的位置、参数等帧信息。帧排列顺序与函数调用顺序相反：编号从0开始，表示当前正在执行的函数（栈顶），1表示调用当前函数的函数，以此类推。

使用`frame`指令切换当前帧，使用`info frame`查看帧的详细信息。

6. 线程管理

- 线程调试中，有些指令是只针对某个线程生效的，指令`break 行号(函数名) thread 线程号`可以为指定线程设置断点，如果多个线程中都执行同一函数，则只有制定的线程才会命中。指令`thread apply 线程号 命令`可以把指令应用于制定的线程。
- 不同线程的调用栈是分开的，可以使用`thread 线程号`切换到要查看调用栈的线程，然后使用`bt`指令回溯栈帧。使用`info threads`可以查看所有线程信息，线程编号前有*的是当前线程。
- 多线程调试例程为test_multithread，使用boost库内的thread_group实现多个线程开启和管理。

7. gdb界面相关

- 界面的一些显示设置常以`set`开头，窗口管理指令则常以`layout`开头，相关指令见附表。
- gdb中可以调用一些C函数，还可以使用`shell 指令`调用系统终端的外部指令。

Part2 其他调试技巧/工具

1. ASan内存管理

ASan(Address Sanitizer)是一个C/C++内存错误检测器，可以发现如内存泄漏、释放后再次使用、堆内存溢出、栈溢出等内存问题。gcc 4.8以上的版本内嵌有该功能，使用时只需在gcc编译时添加编译选项`-fsanitize=address`，或在CMakeList里面添加指令`set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsanitize=address")`。
根据例程test_memory提供了五种常见问题的报错，示例信息如下：

- 内存泄漏

  例程中new_test()函数new了100个int型变量但没回收；malloc_test()函数malloc分配了100个char型变量但没回收。信息解读如下：

  问题类型：memory leaks，内存泄露。

  内存泄露大小及位置：第一个new_test()函数中的new[]操作导致了400B内存泄露；第二个malloc_test()函数中的malloc操作导致了100B内存泄露。

  总结: 内存泄露情况总结，2个位置总共泄露500B内存。

  ==注：例程中内存泄露量很小，然而车上代码要持续长时间运行，即便很小的内存泄露也会不断累积，导致严重的后果，所以一定要在平时编程时培养内存管理的意识。另外，通过程序的输出可以看出，程序是能够正常结束的（例程中其他四种情况也能正常退出程序），如果不对

代码进行认真检查或使用内存管理工具进行分析，我们甚至没有办法发现此类问题。==

```
❯ ./TestMemory
memory test start!
malloc_test
new_test
memory test end!

=================================================================
==29259==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 400 byte(s) in 1 object(s) allocated from:
    #0 0x7fcb9c338608 in operator new[](unsigned long) (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe0608)
    #1 0x563f811e4813 in new_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1813)
    #2 0x563f811e4bc0 in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bc0)
    #3 0x7fcb9baffbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

Direct leak of 100 byte(s) in 1 object(s) allocated from:
    #0 0x7fcb9c336b40 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xdeb40)
    #1 0x563f811e4854 in malloc_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1854)
    #2 0x563f811e4bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #3 0x7fcb9baffbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

SUMMARY: AddressSanitizer: 500 byte(s) leaked in 2 allocation(s).
```

- 堆溢出

例程中heap_buffer_overflow_test()函数分配了10字节内存，然后向其中复制22字节的内容 。信息解读如下：

问题类型：heap buffer overflow，堆溢出。

溢出大小及位置：WRITE of size是写入了22字节大小的数据，由线程T0执行。

堆内存分配位置地址：地址0x60200000001a是已分配内存区域[0x602000000010,0x60200000001a]右侧的第0个地址。

堆内存分配代码位置和分配者：分配10字节区域的函数是heap_buffer_overflow_test()函数中的new[]操作，由线程T0执行。

总结: 堆溢出。

```
❯ ./TestMemory
memory test start!
heap_buffer_overflow_test
=================================================================
==29450==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000001a at pc 0x7f128317e3a6 bp 0x7ffd8f634950 sp 0x7ffd8f6340f8
WRITE of size 22 at 0x60200000001a thread T0
    #0 0x7f128317e3a5  (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x663a5)
    #1 0x558c93af58b7 in heap_buffer_overflow_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x18b7)
    #2 0x558c93af5bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #3 0x7f12829bfbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #4 0x558c93af56f9 in _start (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x16f9)

0x60200000001a is located 0 bytes to the right of 10-byte region [0x602000000010,0x60200000001a)
allocated by thread T0 here:
    #0 0x7f12831f8608 in operator new[](unsigned long) (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe0608)
    #1 0x558c93af5895 in heap_buffer_overflow_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1895)
    #2 0x558c93af5bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #3 0x7f12829bfbf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.4+0x663a5)
```

剩下的信息是关于内存具体情况的说明，对调试意义不大。后面有些情况也可能出现此类信息，不再做说明，想了解可以参考这篇博客：[理解ASAN的shadow memory和读懂报错信息](#)。

```
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c047fff8000: fa fa 00[02]fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==29450==ABORTING
```

- 栈溢出

```
❯ ./TestMemory
memory test start!
stack_buffer_overflow_test
=================================================================
==29603==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffe524c1750 at pc 0x55aa79f1f9f8 bp 0x7ffe524c16e0 sp 0x7ffe524c16d0
READ of size 4 at 0x7ffe524c1750 thread T0
    #0 0x55aa79f1f9f7 in stack_buffer_overflow_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x19f7)
    #1 0x55aa79f1fbbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #2 0x7f5cc9148bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #3 0x55aa79f1f6f9 in _start (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x16f9)

Address 0x7ffe524c1750 is located in stack of thread T0 at offset 80 in frame
    #0 0x55aa79f1f8dd in stack_buffer_overflow_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x18dd)

  This frame has 1 object(s):
    [32, 72) 'test' <== Memory access at offset 80 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x19f7) in stack_buffe
r_overflow_test()
```

- 全局内存溢出

```
❯ ./TestMemory
memory_test_start!
global_buffer_overflow_test
=================================================================
==29841==ERROR: AddressSanitizer: global-buffer-overflow on address 0x561b65859734 at pc 0x561b65657ad5 bp 0x7fff0d223070 sp 0x7fff0d223060
READ of size 4 at 0x561b65859734 thread T0
    #0 0x561b65657ad4 in global_buffer_overflow_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1ad4)
    #1 0x561b65657bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #2 0x7f92939c7bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #3 0x561b656576f9 in _start (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x16f9)

0x561b65859734 is located 4 bytes to the right of global variable 'global_data' defined in '/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/cha
pter_6.2/test_memory.cpp:30:5' (0x561b658595a0) of size 400
SUMMARY: AddressSanitizer: global-buffer-overflow (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1ad4) in global_buf
fer_overflow_test()
```

- 检查释放后继续使用

```
❯ ./TestMemory
memory_test_start!
use_after_free_test
=================================================================
==29964==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010 at pc 0x562170183b7e bp 0x7ffc88ee09d0 sp 0x7ffc88ee09c0
READ of size 1 at 0x602000000010 thread T0
    #0 0x562170183b7d in use_after_free_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1b7d)
    #1 0x562170183bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #2 0x7ff22c7c1bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)
    #3 0x562170183bf9 in _start (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x16f9)

0x602000000010 is located 0 bytes inside of 10-byte region [0x602000000010,0x60200000001a)
freed by thread T0 here:
    #0 0x7ff22cffb480 in operator delete[](void*) (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe1480)
    #1 0x562170183b49 in use_after_free_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1b49)
    #2 0x562170183bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #3 0x7ff22c7c1bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

previously allocated by thread T0 here:
    #0 0x7ff22cffa608 in operator new[](unsigned long) (/usr/lib/x86_64-linux-gnu/libasan.so.4+0xe0608)
    #1 0x562170183b1a in use_after_free_test() (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1b1a)
    #2 0x562170183bbb in main (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1bbb)
    #3 0x7ff22c7c1bf6 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21bf6)

SUMMARY: AddressSanitizer: heap-use-after-free (/home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_6.2/TestMemory+0x1b7d) in use_after_fre
e_test()
```

2. GDB调试发行版

- ==TODO==:生成调试版和发行版的g++和cmake。g++中-g选型和cmake中的set编译模式的区别还在研究。g++编译时不加-g，报错没有源码行数，最多显示到函数名。
- 直接对发行版程序进行调试时，gdb会提示(no debbugging symbols found)，使用l命令和bt命令看不到源码信息。解决这个问题的关键是让gdb了解哪里有调试信息，书中提供的两种方案都需要编译两次，一次生成包含调试符号的调试版，一次生成不包含调试符号的发行版。
- 方法一：从调试版提取调试符号。
  在终端中运行`object --only-keep-debug 调试版程序名 调试版程序名.symbol`
  启动gdb时执行`gdb --symbol=调试版程序名.symbol -exec=发行版程序名`
- 方法二：使用调试版作为符号源
  启动gdb时执行`gdb --symbol=调试版程序名 -exec=发行版程序名`

3. GDB内核转储调试

- Linux系统中的转储文件一般称为内核转储(core dump)，转储文件通常是指程序崩溃时对程序各项状态的记录文件，包含程运行时内存状态、调用栈状态、寄存器状态、线程等。
- 与发行版调试类似，内核转储文件通常都是在客户环境下产生的，可以通过指定调试版代码加载符号以进行调试，需要做到调试版与发行版匹配。
- 开启内核转储存储
  `ulimit -c`查看内核转储容许大小，如果是0则无内核转储输出。可以运行`ulimit -c unlimited`设置容许大小为无限，unlimited可以用数字代替以指定大小，单位为字节。`ulimit`指令仅对当前终端有效，如果想让每个终端都有此设置，运行`sudo gedit /etc/profile`，在打开的文件中的空白行添加`ulimit -c unlimited`并保存退出。然后重启或运行`source /etc/profile`使之生效。
- 设置转储文件名称格式
  该设置无法在终端中调用gedit等文本编辑器修改，需要运行`sudo -i`指令进入root模式。然后运行`echo -e "/home/hitcrt/corefile/core-%e-%s-%p-%t" > /proc/sys/kernel/core_pattern`设置文件储存操作和命名格式，名称含义可参考博客使用coredump帮助解决segmentation fault的问题。注意root模式下的操作都具有最高权限，所以要非常谨慎！更改完毕后运行`exit`退出root模式。
  测试发现此更改重启后会被覆盖，暂时没有找到解决方案。

- 完成设置后运行例程test_coredump，会提示段错误并完成核转储。按照前述设置，转储文件位置在/home/hitcrt/corefile。运行 `gdb 调试版程序名 转储文件名`，即可以看到相关信息。部分输出信息如下图，首先从调试版程序中读取了调试符号，然后是产生转储文件的程序名，程序崩溃的原因，以及崩溃时运行到的位置。

```
Reading symbols from TestCoreDump...done.
[New LWP 10179]
Core was generated by `./TestCoreDump'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000557dbdb45aca in crash_test () at /home/hitcrt/CRT/codes/test/cpp_debug/book_debug-master/chapter_8.2/test_core_dump.cpp:8
8               strcpy(str, "test");
(gdb)
```

- gdb的大多数操作都能用于转储文件调试，唯一显著的不同是不能继续执行代码，只能查看信息。

4.backward-cpp查找段错误

```
Backward is a beautiful stack trace pretty printer for C++.
```

Backward是一个堆栈跟踪打印器，是一个开源的C++项目。与使用GDB不同，backward是一个C++库，使用时只需添加头文件并增加几行代码即可。项目使用步骤如下：
(1) 依赖安装

```
sudo apt-get install binutils-dev
sudo apt-get install libdw-dev
```

(2)然后进入backward-cpp使用cmake安装

```
mkdir build && cd build
cmake .. && make -j16
sudo make install
```

针对CMakeLists的修改有两种方式：
方法一：将backward-cpp复制到项目目录下，与程序一起编译，生成静态库(不推荐)

```
add_subdirectory(/path/to/backward-cpp)

# This will add backward.cpp to your target
add_executable(mytarget mysource.cpp ${BACKWARD_ENABLE})

# This will add libraries, definitions and include directories needed by
backward
# by setting each property on the target.
add_backward(mytarget)
```

方法二：find_package()找到backward位置，动态链接到项目中（推荐）

```
list(APPEND CMAKE_MODULE_PATH /path/to/backward-cpp)
find_package(Backward)

# This will add libraries, definitions and include directories needed by
backward
# through an IMPORTED target.
target_link_libraries(mytarget PUBLIC Backward::Backward)
```

不增加以上内容直接编译，输出只有segment fault，增加以后则可以输出问题代码行号。

5. GDB调试动态库

- 静态库和动态库最本质的区别就是：该库是否被编译进目标（程序）内部。
- 静态（函数）库 一般扩展名为.a或.lib，这类库在编译的时候会直接整合到目标程序中，所以利用静态函数库编译成的文件会比较大，这类函数库最大的优点就是编译成功的可执行文件可以独立运行，而不再需要向外部要求读取函数库的内容；但如果函数库更新，需要重新编译。
- 动态函数库动态函数库的扩展名一般为.so或.dll，与静态函数库被整个捕捉到程序中不同，动态函数库在编译的时候，在程序里只有一个“指向”的位置而已，也就是说当可执行文件需要使用到函数库的机制时，程序才会去读取函数库来使用，即可执行文件无法单独运行。这样从产品功能升级角度方便升级，只要替换对应动态库即可，不必重新编译整个可执行文件。
- Linux中的动态库命名规则如下：

> lib[动态库名].so.主版本号.次版本号.发行版本号

- gcc在链接时寻找的名称仍是不包含版本号的文件名。如果要使用带有版本号的文件名，要建立一个软连接，指向真实动态库的文件名。
- Linux查找动态库的顺序
    - 可执行文件指定的路径，可在CMakeList中设置
    - /usr/lib、/lib、/usr/local/lib
    - LD_LIBRARY_PATH环境变量中设定的路径，只影响当前shell
    - /etc/ld.so.conf配置的路径，添加后要执行ldconfig指令使其生效
- 使用gdb对可执行文件调试时，如果要调试动态库的代码，设置断点时会提示处于pending状态。这是因为动态库只有函数执行时才会被载入，等到程序运行到该函数时断点才从pending状态离开，等到运行到该断点位置再命中。
- 以静态链接方式调用动态库时，调试符号中的库函数除了显示函数名，还能显示代码行号；动态链接方式调用动态库，则只能显示函数名。其他并无差异。
- 附加到进程调试动态库时，要保证在可执行文件的目录下打开终端运行gdb，否则无法加载调试符号。

6. GDB远程调试

- 远程GDB调试使用gdb server，安装指令 `sudo apt-get install gdbserver` 。
- 目标机运行指令 `gdbserver 目标机IP:监听端口号` 直接启动程序，或使用 `gdbserver 目标机IP:监听端口号 --attach 进程号` 附加到进程。然后调试机执行 `target remote 目标机IP:监听端口号` ，远程连接成功。执行 `detach` 调试机与目标机断开，执行 `quit` 目标机上测试程序也会结束。
- gdb server启动后，一旦调试机退出调试，gdb server也会退出，再次调试要重新启动gdb server。

7. 死锁调试-==TODO==

死锁调试
什么是死锁？p150
发现死锁
直接现象：程序卡在某处/始终不停止
检查方法：查看多个线程，均包含lock_wait()
常见死锁原因p156
互斥条件
保持和请求条件
避免死锁p158

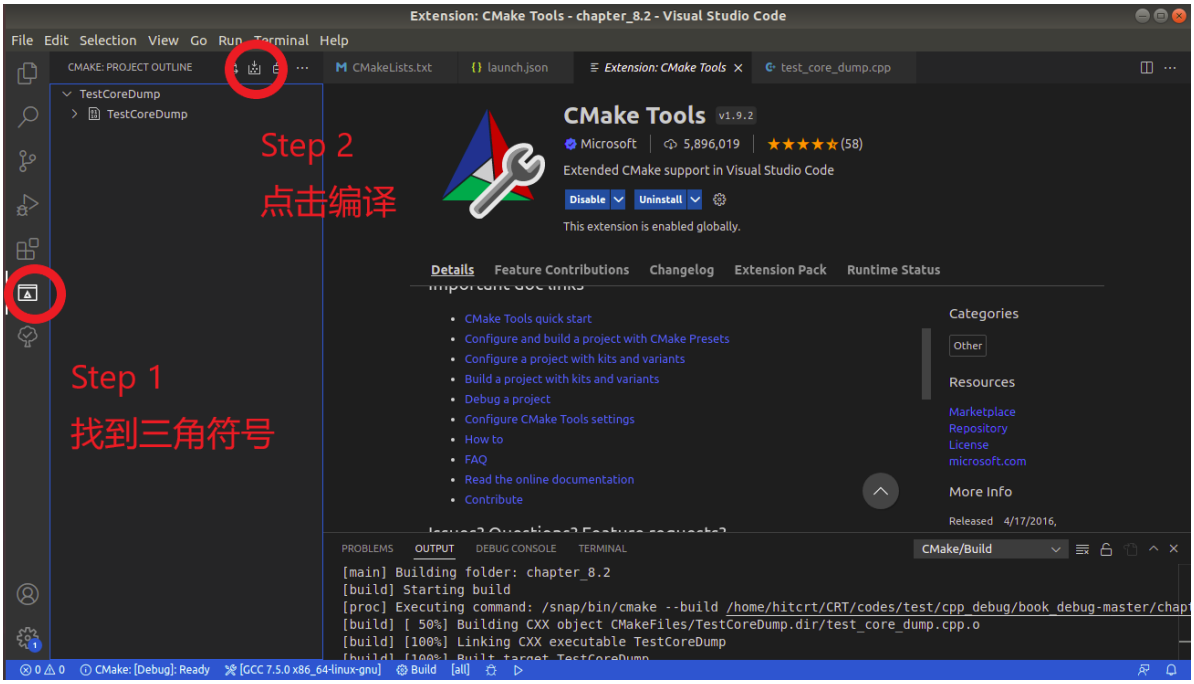
顺序使用

控制作用域

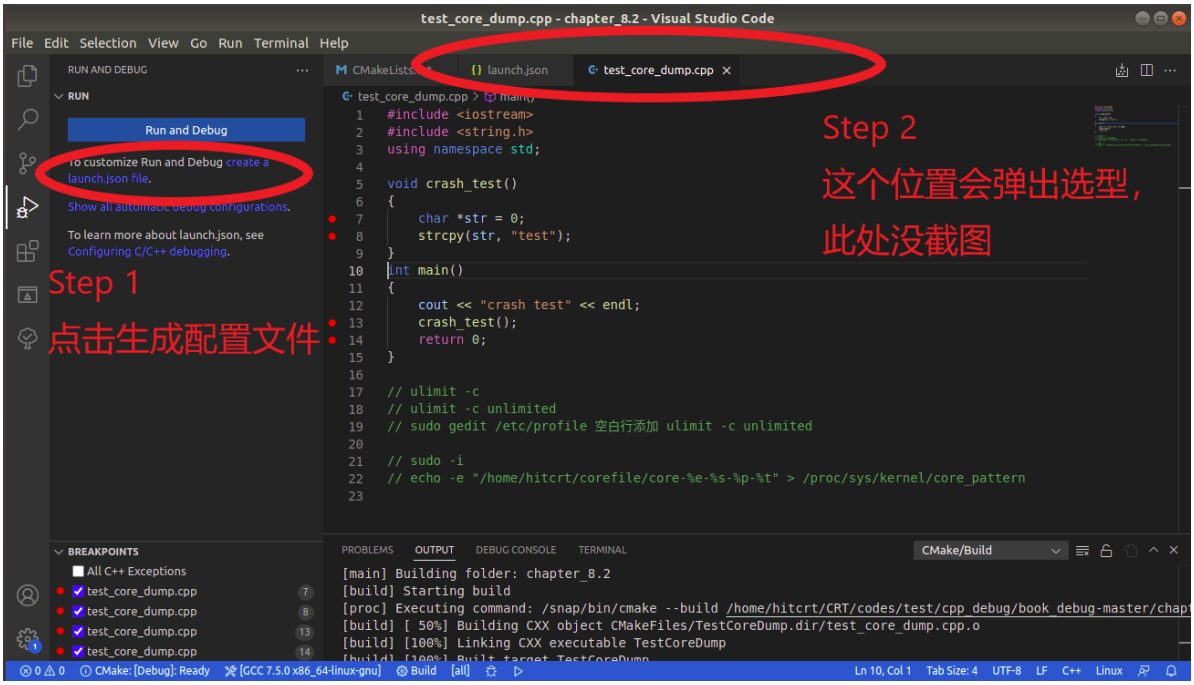
Part3 VS Code+CMake配置GDB调试

前两部分对终端使用gdb做了较为详细的介绍，如果觉得终端调试不习惯，也可以使用编辑器中的插件进行调试。但编辑器调试归根结底也是使用gdb，只是界面更加直观漂亮，如果不掌握概念也没法得心应手地使用。下面介绍VSCode中如何设置GDB调试。

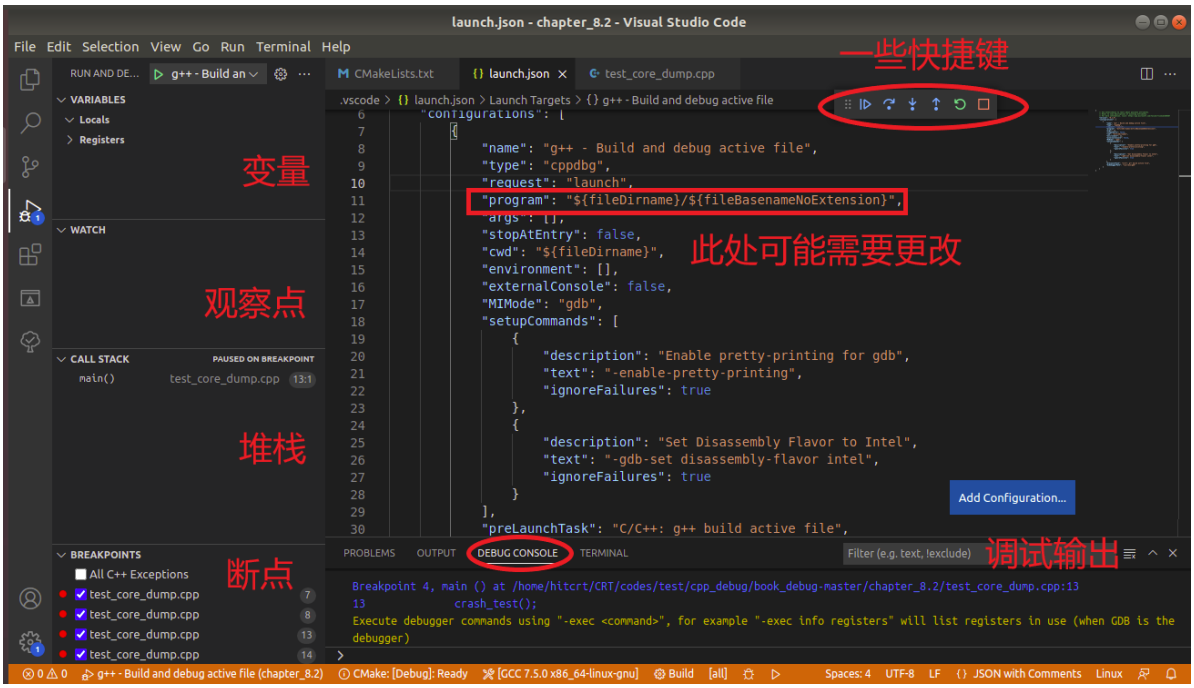
首先安装C/C++插件和CMake Tools插件，之后打开项目文件夹，可以看到左侧带有三角标识的按键，界面中的按键可以进行cmake及编译，不必从终端中输入指令。



之后点击调试按钮，点击create launch.json蓝字，然后到软件上方依次选择GDB，g++，生成调试配置文件。



如果program项不是程序名，要改为程序名。保存后可看见如下界面。



附录：GDB常用指令表

启动/停止/执行

命令	功能
run (r)	启动程序
run *命令行参数*	以传入参数的方式启动程序
run > *输出文件*	将输出重定向到输出文件
continue (c)	继续运行，直到下一个断点
continue *数量*	继续执行，忽略指定数量的命中次数
kill	停止程序
quit (q)	退出gdb
finish	跳出当前函数,在最外层帧使用无效
step(s)	逐语句执行，执行遇到的每个语句
step *步数*	逐语句执行步数
next(n)	逐过程执行，会跳过子函数
next *数量*	逐过程执行指定行数的代码
where	显示当前执行的具体函数和代码行
jump *代码行号(函数地址)*	跳转到某行或某个函数执行

断点管理

命令	功能
break(b) *函数名*	为函数设置断点，执行到该位置则命中该断点
break *类名*::*函数名*	在某个类的成员函数上设置断点，该函数必须被调用过
break *文件名*:*函数名*	在文件名指定的函数上设置断点，该函数必须被调用过
rbreak(rb) *正则表达式*	使用正则表达式对满足命名规则的函数设置断点
break *代码行号*	在某一代码行上设置断点，不执行行号句
break *文件名*:*行号*	在文件名指定的代码行上设置断点
break **地址*	在指定地址设置断点，要在程序运行后使用地址
break +*偏移量*	在当前代码行加上偏移量的位置设置断点
break -*偏移量*	在当前代码行减去偏移量的位置设置断点
break *行号(函数名)* if *条件*	设置条件断点
tbreak	设置临时断点
disable *断点范围*	禁用指定范围的断点
enable *断点范围*	启用指定范围断点
enable once *断点编号*	启用指定断点一次
enable count *命中次数N 断点编号*	启用指定断点，命中N次后禁用
ignore *断点编号 忽略次数N*	忽略断点前N次命中，第N+1次命中时才暂停
delete(d)	删除所有断点，包括观察点和捕获点
delete *断点编号*	删除指定编号断点
delete *断点范围*	删除指定范围断点
clear	删除所有断点
clear *函数名*	删除该函数的断点
clear *行号*	删除行号对应的断点
watch *变量(表达式)*	添加观察点
rwatch *变量(表达式)*	添加读取观察点，变量或表达式被读取时程序发生中断
awatch *变量(表达式)*	添加读写观察点，变量或表达式被读取或写入时程序发生中断
catch throw	执行到throw语句时中断
catch catch	执行到catch语句块时会中断，即代码捕获异常时中断
catch load	加载动态库时中断

命令	功能
catch unload	卸载动态库时中断
tcatch *事件*	设置临时捕获点

调用栈

命令	功能
backtrace(bt)	显示调用栈信息
bt *栈帧数*	显示指定数量的栈帧（栈顺序）
bt - 栈帧数	显示指定数量的栈帧（调用顺序）
backtrace full	显示所有栈帧的局部变量
frame	显示当前帧编号等信息
frame(f) *帧编号*	切换锁到指定编号的帧
up	切换帧，将当前帧增大1
down	切换帧，将当前帧减少1
up *帧数量*	切换帧，将当前帧增大指定数量
down *帧数量*	切换帧，将当前帧减少指定数量

线程管理

命令	功能
thread *线程号*	切换当前要查看的线程
info(i) thread	查看当前线程
b 行号(函数名) thread *线程号*	为指定线程设置断点
thread apply 线程号 命令	对指定线程执行命令

查看源代码

命令	功能
list (l)	查看源代码
list *行号*	显示指定行号代码
list *函数名*	显示指定函数的代码
list -	往前显示代码
list *开始,结束*	显示指定区间的代码
list *文件名:行号*	显示指定文件名的指定行代码
set listsize *数字*	设置显示的代码行数
show listsize	查看一次显示的代码行数
directory(dir) *目录名*	添加目录到源代码搜索路径中
directory(dir)	清空添加到源代码搜索目录的目录
show directories(dir)	查看源代码搜索目录
search *正则表达式*	搜索源码（从前向后）
reverse-search *正则表达式*	搜索源码（从后向前）

查看信息

命令	功能
info breakpoints(break)	查看所有断点信息
ib	查看所有断点信息
info break *断点编号*	查看指定断点编号的断点信息
info watchpoints	查看所有观察点信息
info registers	查看所有整型寄存器信息
info threads	查看所有线程信息
info frame	查看当前帧的信息
info args	查看当前帧的参数
info locals	查看当前帧的局部变量

查看变量

命令	功能
x *地址*	查看指定地址的内存
x *局部变量名*	查看指定变量地址的内存
x /*选型 地址*	以特定的格式查看指定地址的内存
print(p) *变量名*	查看变量
display *变量名*	自动显示变量
p *文件名:变量名*	查看指定文件的变量
ptype *可选参数 变量*	查看变量类型
ptype *可选参数 数据类型*	查看类型（或类）的详细信息

gdb使用

命令	功能
set logging on	设置日志开关
set logging off	
show logging	
set logging file *日志文件名*	设置日志文件名，默认名称为gdb.txt
set print array on	数组显示是否友好开关，默认是关闭的
set print array off	
show print array	
set print array-indexes on	显示数组索引开关，默认是关闭的
set print array-indexes off	
show print array-indexes	
set print pretty on	格式化结构体，默认是关闭的
set print pretty off	
show print pretty	
set print union on	联合体开关，默认是关闭的
set print union off	
show print union	
layout next	显示下一个窗口
layout prev	显示前一个窗口
layout src	只显示源代码窗口
layout asm	只显示汇编窗口
layout split	显示源码和汇编窗口
layout regs	显示寄存器窗口，与源码、汇编一起窗口显示
focus next	设置窗口为活动窗口
refresh	刷新屏幕
update	更新源代码窗口

注：括号表示缩写或简略写法

例程清单

```
.
|-- test_coredump 内核转储测试
|   |-- CMakeLists.txt
|   `-- test_core_dump.cpp
|-- test_memory 内存泄露测试
|   |-- CMakeLists.txt
|   |-- TestMemory
|   `-- test_memory.cpp
`-- test_multithread 多线程测试
|-- CMakeLists.txt
`-- test_multithread.cpp
```

参考文献

[1] 张海洋. C/C++代码调试的艺术[M]. 北京:人民邮电出版社, 2021

[2] SunnyYang. GCC G++ Make CMake自我科普[EB/OL]. https://www.cnblogs.com/jiy-for-you/p/7282011.html, 2017-08-03/2021-02-04

拓展阅读

Debugging with GDB(在more文件夹中)

gdb 中step,next与finish的区别

正则表达式GitHub-ziishaned-learn-regex: Learn regex the easy way

栈和堆

GDB frame和backtrace命令：查看栈信息