绪论

本文分为两个部分，第一部分是在某次论文分享活动中分享的Debug经验PPT，主要介绍常见bug以及debug手段。PPT核心是想告诉大家debug手段优先级"求上为日志，求中为GDB，求下为print"。第二部为GDB使用说明书，包括Core dump、GDB命令以及多线程GDB。

武汉光电国家研究中心

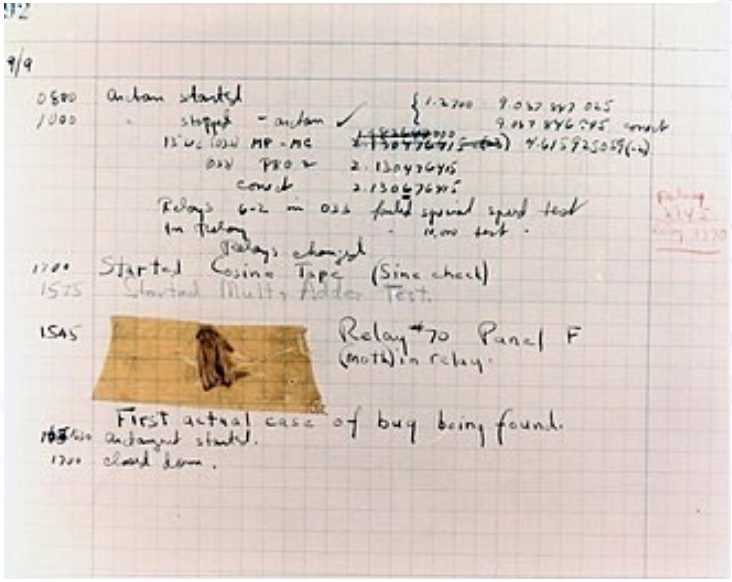
Debug 经验分享

报告人： 刘必勇

日期：2023年xx月xx号

世界上第一个BUG

- 1947年9月9日，葛丽丝·霍普（Grace Hopper）发现了第一个电脑bug。有一次Mark II突然宕机，整队团队都搞不清电脑为何不能正常运作。经过大家深度挖掘，发现原来有飞蛾意外飞入一台电脑引起故障。



常见BUG类型

- 逻辑错误
  - 当出现逻辑错误（logic bug）时，这意味着代码在语法上是正确的，但它的行为不符合你的预期。这可能是因为算法错误、条件错误、循环错误、类型转换错误、字符串处理错误等。

```
int main() {
    int num1, num2, sum;
   std::cout << "Enter the first number: ";
    std::cin >> num1;
    std::cout << "Enter the second number: ";
    std::cin >> num2;
    // 计算它们的和
    sum = num1 - num2;  // 逻辑错误：应该是相加而不是相减
    // 显示结果
    std::cout << "The sum of " << num1 << " and " << num2 << " is: " << sum << std::endl;
    return 0;
}
```

常见BUG类型

- 空指针
  - 空指针错误是在试图访问或操作一个空指针（nullptr）时引发的错误（释放空指针）。

```
void printValue(int* ptr) {
    // 尝试打印指针指向的值，未检查空指针
    std::cout << "The value is: " << *ptr << std::endl;
}

int main() {
    int* ptr = nullptr;  // 声明一个空指针

    // 将空指针传递给函数，未检查空指针
    printValue(ptr);

    return 0;
}
```

常见BUG类型

- 野指针
  - 野指针是指指针变量被赋予了一个未知的或无效的地址，这样的指针可能会导致程序出现不可预测的错误。（越界访问数组、使用未初始化的变量、使用已释放的内存）

```
#include <iostream>

int main() {
    int* wildPtr;  // 未初始化的指针

    // 尝试解引用未初始化的指针（野指针错误）
    std::cout << "The value is: " << *wildPtr << std::endl;

    return 0;
}
```

常见BUG类型

- 内存泄漏
  - 内存泄漏是指在程序中动态分配的内存没有被释放的情况。如果在程序执行过程中分配了内存（例如使用 new 或 malloc），但没有相应地释放它（例如使用 delete 或 free），就会发生内存泄漏。

```
int main() {
    // 内存泄漏示例
    int* dynamicArray = new int[5];  // 分配一个整数数组
    // 使用动态分配的数组
    for (int i = 0; i < 5; ++i) {
        dynamicArray[i] = i;
    }
    // 没有释放分配的内存
    // delete[] dynamicArray;  // 注释掉这行以模拟内存泄漏
    // 尝试访问已释放的内存（悬挂指针）
    std::cout << "Accessing freed memory: " << dynamicArray[0] << std::endl;

    return 0;
}
```

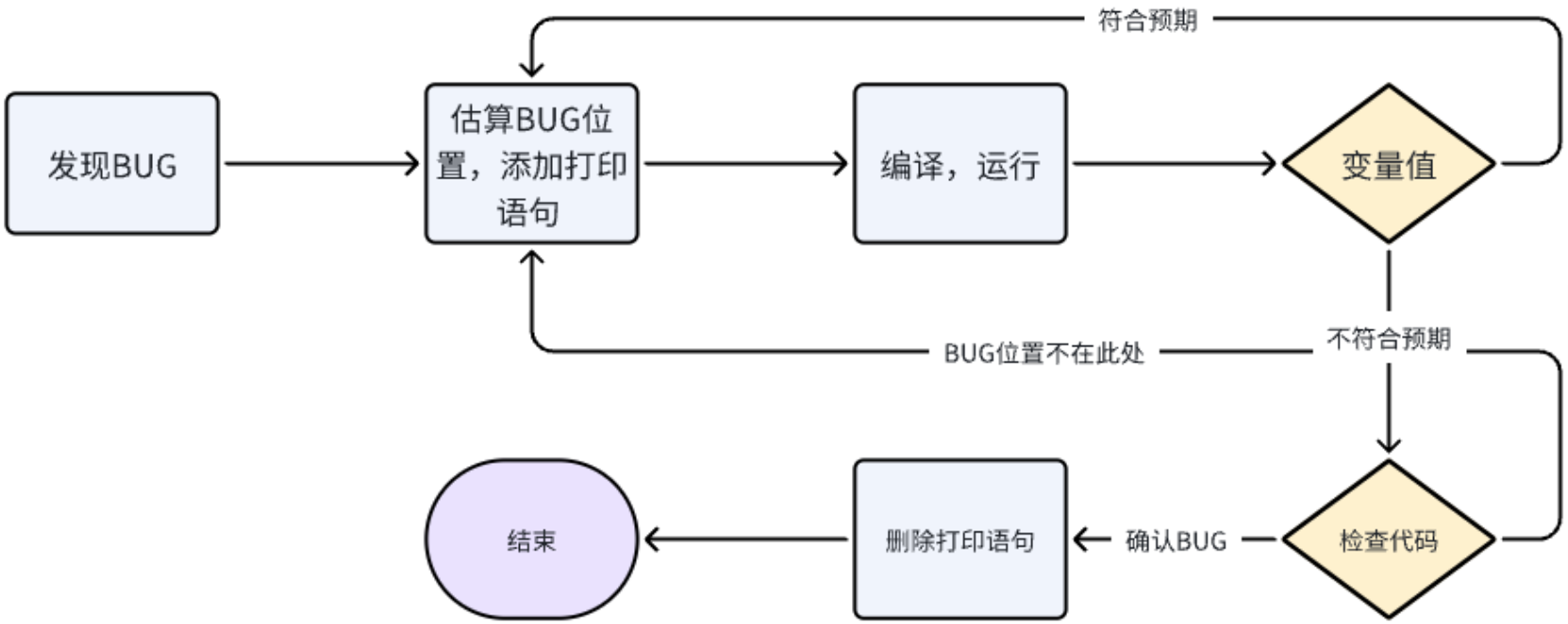
常见BUG类型

- 死锁
  - 死锁（Deadlock）是多线程或多进程编程中一种常见的 bug，它发生在两个或多个进程（线程）之间，每个进程都在等待对方释放某个资源，从而导致所有的进程都无法继续执行。死锁通常涉及互斥、持有和等待这三个条件。

```
std::mutex mutex1;
std::mutex mutex2;
void threadFunction1() {
    std::lock_guard<std::mutex> lock1(mutex1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(mutex2);
    std::cout << "Thread 1 executed successfully" << std::endl;
}
void threadFunction2() {
    std::lock_guard<std::mutex> lock2(mutex2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(mutex1);
    std::cout << "Thread 2 executed successfully" << std::endl;
}
int main() {
    std::thread t1(threadFunction1);    std::thread
t2(threadFunction2);
    t1.join();    t2.join();
    return 0;
}
```

DEBUG方法 -- 终端打印变量值

- 使用终端打印（console printing）是一种常见的调试方法。你可以使用标准输出流 std::cout 来输出信息到终端



DEBUG方法 -- 终端打印变量值

存在问题：

- Bug位置不清楚，需要反复多次加打印语句
- Bug变量不清楚，需要反复调整打印变量
- 需要反复多次运行，在编译以及运行时间长的情况下，非常浪费时间
- 找到Bug后需要删掉printf语句
- 只能处理极其简单的bug，一旦出现并发bug非常棘手，bug出现具有偶然性

DEBUG方法 -- GDB

- 启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。
- 可让被调试的程序在你所指定的调置的断点处停住。
- 当程序被停住时，可以检查此时你的程序中所发生的事。
- 你可以改变你的程序，将一个BUG产生的影响修正从而测试其他BUG。

```
GNU gdb (GDB) 12.1
Copyright (C) 2022 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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".
(gdb) help
List of classes of commands:

aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

DEBUG方法 -- GDB

存在问题：

- 学习曲线陡峭
- 多线程调试复杂

DEBUG方法 -- Core Dump文件

- Core Dump文件（核心转储文件）是在程序发生崩溃或异常终止时生成的一种文件。它包含了在程序崩溃时内存的快照，提供了程序崩溃时的内存状态和堆栈信息。这对于调试和分析程序故障非常有用。

```
TEG_23_76_sles10_64:/data/coredump # readelf -h core
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              CORE (Core file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         25
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

DEBUG方法 --日志

- 日志调试是通过记录程序执行过程中的关键信息到日志文件，以便开发者在后期分析和排查问题时进行调试的一种方法。合理的日志记录可以为开发人员提供关于程序运行状态、变量值、错误信息等方面的详细信息。
- 优点：适用于所有场景
- 缺点：写代码要时刻加日志

```
rootservice.log.20231117124436401:[2023-11-17 10:57:55.267633] INFO  [RS]
create_normal_tenant (ob_ddl_service.cpp:22632) [716373][DDLQueueTh0][T0][YB427F000001-
00060A504906126E-0-0] [lt=23] [CREATE_TENANT] STEP 2. finish create tenant(ret=0,
ret="OB_SUCCESS", tenant_id=1001, cost=15760027)
rootservice.log.20231117124436401:[2023-11-17 10:58:10.868829] INFO  [RS]
create_normal_tenant (ob_ddl_service.cpp:22632) [716373][DDLQueueTh0][T0][YB427F000001-
00060A504906126E-0-0] [lt=19] [CREATE_TENANT] STEP 2. finish create tenant(ret=0,
ret="OB_SUCCESS", tenant_id=1002, cost=15546709)
```

```
if (!mem_dump.is_inited()) {
    ret = OB_NOT_INIT;
    LOG_WARN("not inited", K(ret));
  } else if (OB_FAIL(read_cmd(buf, len, real_size))) {
    LOG_WARN("read cmd failed", K(ret));
  } else if(FALSE_IT(cmd.assign_ptr(buf, static_cast<int32_t>(real_size)))) {
  } else if (OB_FAIL(parser.parse(cmd, parse_result))) {
    LOG_WARN("parse failed", K(cmd), K(ret));
```

C++日志库推荐 - spdlog

- 大多数开源代码都有自己的日志模块，直接用即可。
- 如果没有，则可以用高性能简易C++日志库**spdlog**，

https://github.com/gabime/spdlog

```
#include "spdlog/spdlog.h"

int main()
{
    auto logger = spdlog::basic_logger_mt("basic_logger", "logs/basic-log.txt");
    spdlog::set_level(spdlog::level::debug); // Set global log level to debug
    spdlog::set_level(spdlog::level::off); //
    spdlog::debug("This message should be displayed..");

    // change log pattern
    spdlog::set_pattern("[%H:%M:%S %z] [%n] [%^---%L---%$] [thread %t] %v");

    // Compile time log levels
    // define SPDLOG_ACTIVE_LEVEL to desired level
    SPDLOG_TRACE("Some trace message with param {}", 42);
    SPDLOG_DEBUG("Some debug message");
}
```

DEBUG方法 –内存泄漏检测工具

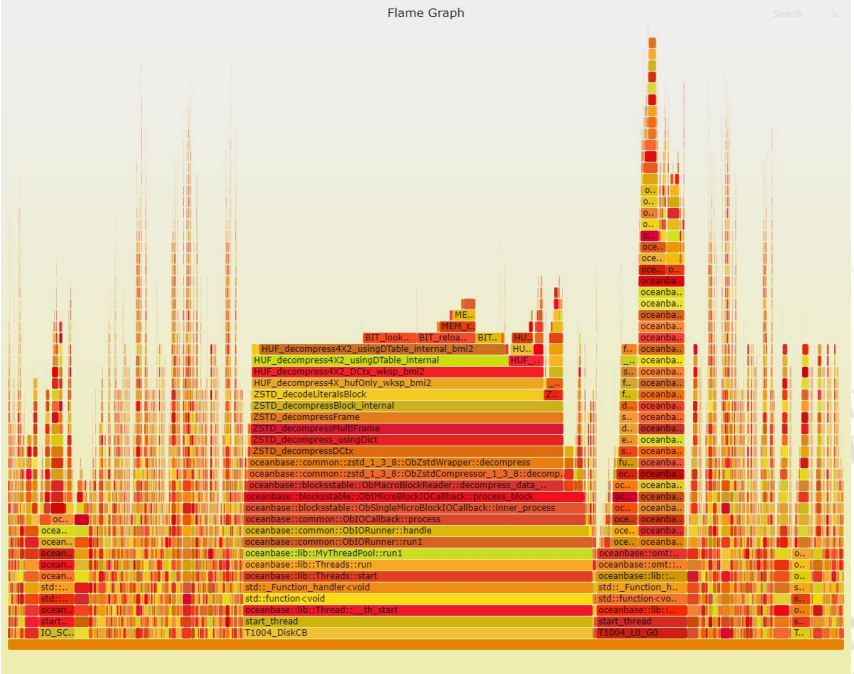
- 内存泄漏检测工具是一类用于检测程序中是否存在内存泄漏问题的软件工具。这些工具可以在程序运行时或在编译时进行检测，帮助开发者发现并修复内存泄漏。
  - **Valgrind 是一个强大的内存分析工具，可以检测内存泄漏、使用未初始化的内存、访问已释放的内存等问题。它提供了多个工具，其中 Memcheck 是最常用的用于内存泄漏检测的工具之一。**
  - **AddressSanitizer 是 Clang 和 GCC 编译器的一个特性，用于检测内存错误，包括内存泄漏。它通过在运行时注入额外的代码来进行检测，可以捕获对未分配或已释放内存的访问。**

Debug Example – core dump报错

Debug Example –内存泄漏报错

One More Thing – 性能调试

- 火焰图（Flame Graph）是一种可视化性能分析工具，最初由Brendan Gregg开发。它主要用于可视化程序的调用栈和函数的执行时间，以便更容易地识别性能瓶颈和优化机会。火焰图通常用于分析软件的CPU使用情况。



One More Thing – 有用链接

- gdb/core dump文件

https://yd4fppb7u2.feishu.cn/wiki/wikcniG6IeKRqm1mLqNno9uzgbg?from=from_copylink

- 火焰图

https://zhuanlan.zhihu.com/p/402188023

- valgrind

http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/

- AddressSanitizer

https://github.com/google/sanitizers/wiki/AddressSanitizer

谢谢聆听

GDB

Core Dump 调试

想象一种场景：一个复杂 C++程序运行了半个小时突然终止，报错信息为 core dump，简单思考一下原因很有可能是访问空指针导致，那么如何定位访问这个空指针的代码所在的位置？如果你直接使用 gdb 逐行运行肯定要到明年，此时你就可以使用 core-dump 文件调试，直接定位到空指针的代码。

核心文件（core file），也称为核心转存（core dump），是操作系统在进程运行的过程中异常终止或崩溃，操作系统会将程序当时的内存状态记录下来，保存在一个文件中，这种行为就叫做 Core Dump（中文有的翻译成“核心转储”)。我们可以认为 core dump 是“内存快照”，但实际上，除了内存信息之外，还有些关键的程序运行状态也会同时 dump 下来，例如寄存器信息（包括程序指针、栈指针等）、内存管理信息、其他处理器和操作系统状态和信息。core dump 对于编程人员诊断和调试程序是非常有帮助的，因为对于有些程序错误是很难重现的，例如指针异常，而 core dump 文件可以再现程序出错时的情景。

Segmentation fault (core dumped)多为内存不当操作造成。空指针、野指针的读写操作，数组越界访问，破坏常量等。如对链表的新增和释放包括赋值等等，如出现不当操作都有可能造成程序崩溃。对每个指针声明后进行初始化为 NULL 是避免这个问题的好办法。排除此问题的最好办法则是调试。

开启 core dump

当程序 core dump 时，可能会产生 core 文件，它能够很大程序帮助我们定位问题。但前提是系统没有限制 core 文件的产生。可以使用命令 limit -c 查看:

```
Plaintext
$ ulimit -c
0
```

如果结果是 0，那么恭喜你，即便程序 core dump 了也不会有 core 文件留下。我们需要让 core 文件能够产生:

```
Plaintext
$ ulimit -c unlimited  #表示不限制 core 文件大小
```

设置 core 文件的名称和文件路径

默认生成路径：输入可执行文件运行命令的同一路径下
默认生成名字：默认命名为 core。新的 core 文件会覆盖旧的 core 文件

```
Python
cat /proc/sys/kernel/core_pattern #查看默认 core 文件路径及名字
echo "/mnt/core-%e-%p-%t" > /proc/sys/kernel/core_pattern #临时修改 core 文件路径及名字
```

长期保存配置，切换到 root 用户，执行#vim /etc/sysctl.conf ，然后在 sysctl.conf 文件中添加下面两句话：

```
Python

      kernel.core_pattern = /mnt/core-%e-%p
      kernel.core_uses_pid = 0
```

保存 sysctl.conf 文件后退出，并执行 sysctl -p 即时生效。

阅读 coredump 文件

如何“阅读”coredump 文件

1. 编译，执行。编译时最好加-g 选项，能记录更多信息

2. 运行 gdb 阅读 core 文件，命令为“gdb 二进制程序路径 coredump 文件路径”，这时就进入 gdb 的提示符“(gdb)”。下面 bt 代码打印堆栈信息，最上方栈对应代码即为出现错误代码，如下对应 cd_test.c:13

```
C++
(gdb) bt
#0  0x080483eb in div (div_i=4, div_j=0) at cd_test.c:13
#1  0x08048422 in sub (sub_i=2, sub_j=1) at cd_test.c:24
#2  0x08048461 in add (add_i=1, add_j=0) at cd_test.c:37
#3  0x08048498 in main (argc=1, argv=0xbfba3264) at cd_test.c:53
```

编译时未加-g 选项，gdb 查看状态

```
C++
Core was generated by `./cd_test'.
Program terminated with signal 8, Arithmetic exception.
#0  0x080483eb in div ()
(gdb) bt
```

```
#0  0x080483eb in div ()     对比带-g 选项的少了些什么？
#1  0x08048422 in sub ()
#2  0x08048461 in add ()
#3  0x08048498 in main ()
```

扩展阅读

如何定位 core dump 问题 https://xie.infoq.cn/article/7f0f2f44e16338c6bae46a47a

GDB 教程

本文参考资料 GDB cheetsheet GDB 进阶 GDB 调试入门指南知乎

想象一种场景：如果此时我们已经定位报错出现的位置，如一个变量值不符合预期，但是思考后发现导致这个变量值异常的位置不在这里，我们应该如何处理呢？此时可以利用 GDB 穿梭于函数调用栈中查看变量的异常情况，如果错误不在当前函数，则可进入父函数即调用当前函数的函数，查看各个变量值的情况。

启动调试

前言

GDB（GNU Debugger）是 UNIX 及 UNIX-like 下的强大调试工具，可以调试 ada, c, c++, asm, minimal, d, fortran, objective-c, go, java,pascal 等语言。本文以 C 程序为例，介绍 GDB 启动调试的多种方式。

哪类程序可被调试

对于 C 程序来说，需要在编译时加上-g 参数，保留调试信息，否则不能使用 GDB 进行调试。 但如果不是自己编译的程序，并不知道是否带有-g 参数，如何判断一个文件是否带有调试信息呢？判断方式如下

- gdb 文件

例如：

```
Plaintext
$ gdb helloworld
Reading symbols from helloWorld...(no debugging symbols
found)...done.
```

如果没有调试信息，会提示 no debugging symbols found。 如果是下面的提示：

```
Plaintext
Reading symbols from helloWorld...done.
```

则可以进行调试。

- readelf 查看段信息

例如：

```
Plaintext
$ readelf -S helloWorld|grep debug
  [28] .debug_aranges    PROGBITS         0000000000000000
0000106d
  [29] .debug_info       PROGBITS         0000000000000000
0000109d
  [30] .debug_abbrev     PROGBITS         0000000000000000
0000115b
  [31] .debug_line       PROGBITS         0000000000000000
000011b9
  [32] .debug_str        PROGBITS         0000000000000000
000011fc
```

helloWorld 为文件名，如果没有任何 debug 信息，则不能被调试。

- file 查看 strip 状况

下面的情况也是不可调试的：

```
Plaintext
$ file helloWorld
helloWorld: (省略前面内容) stripped
```

如果最后是 stripped，则说明该文件的符号表信息和调试信息已被去除，不能使用 gdb 调试。但是 not stripped 的情况并不能说明能够被调试。

调试方式运行程序

程序还未启动时，可有多种方式启动调试。

调试启动无参程序

例如：

```
Plaintext
```

```
$ gdb helloWorld
(gdb)
```

输入 run 命令，即可运行程序

调试启动带参程序

假设有以下程序，启动时需要带参数:

```
Plaintext
#include<stdio.h>
int main(int argc,char *argv[]){
    if(1 >= argc)
    {
        printf("usage:hello name\n");
        return 0;
    }
    printf("Hello World %s!\n",argv[1]);
    return 0 ;
}
```

编译:

```
Plaintext
$ gcc -g -o hello hello.c
```

这种情况如何启动调试呢？需要设置参数:

```
Plaintext
$ gdb hello
(gdb)run 编程珠玑
Starting program: /home/shouwang/workspaces/c/hello 编程珠玑
Hello World 编程珠玑!
[Inferior 1 (process 20084) exited normally]
(gdb)
```

只需要 run 的时候带上参数即可。 或者使用 set args，然后在用 run 启动:

```
Plaintext
$ gdb hello
(gdb) set args 编程珠玑
(gdb) run
Starting program: /home/hyb/workspaces/c/hello 编程珠玑
Hello World 编程珠玑!
```

```
[Inferior 1 (process 20201) exited normally]
(gdb)
```

调试已运行程序

如果程序已经运行了怎么办呢？ 首先使用 ps 命令找到进程 id：

```
Plaintext
$ ps -ef|grep 进程名
```

或者：

```
Plaintext
$ pidof 进程名
```

- attach 方式

假设获取到进程 id 为 20829，则可用下面的方式调试进程：

```
Plaintext
$ gdb
(gdb) attach 20829
```

接下来就可以继续你的调试啦。

可能会有下面的错误提示：

```
Plaintext
Could not attach to process.  If your uid matches the uid of the
target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope,
or try
again as the root user.  For more details, see /etc/sysctl.d/10-
ptrace.conf
ptrace: Operation not permitted.
```

解决方法，切换到 root 用户： 将/etc/sysctl.d/10-ptrace.conf 中的

```
Plaintext
kernel.yama.ptrace_scope = 1
```

修改为

```
Plaintext
```

```
kernel.yama.ptrace_scope = 0
```

- 直接调试相关 id 进程

还可以是用这样的方式 gdb program pid，例如:

```
Plaintext
gdb hello 20829
```

或者:

```
Plaintext
gdb hello --pid 20829
```

已运行程序没有调试信息

为了节省磁盘空间，已经运行的程序通常没有调试信息。但如果又不能停止当前程序重新启动调试，那怎么办呢？还有办法，那就是同样的代码，再编译出一个带调试信息的版本。然后使用和前面提到的方式操作。对于 attach 方式，在 attach 之前，使用 file 命令即可:

```
Plaintext
$ gdb
(gdb) file hello
Reading symbols from hello...done.
(gdb)attach 20829
```

断点设置

前言

上节 GDB 调试指南-启动调试我们讲到了 GDB 启动调试的多种方式，分别应用于多种场景。今天我们来介绍一下断点设置的多种方式。

为何要设置断点

在介绍之前，我们首先需要了解，为什么需要设置断点。我们在指定位置设置断点之后，程序运行到该位置将会“暂停”，这个时候我们就可以对程序进行更多的操作，比如查看变量内容，堆栈情况等等，以帮助我们调试程序。

查看已设置的断点

在学习断点设置之前，我们可以使用 info breakpoints 查看已设置断点：

```
Plaintext
info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004005fc in printNum2 at
test.c:17
    breakpoint already hit 1 time
2       hw watchpoint  keep y                      a
    breakpoint already hit 1 time
    ignore next 3 hits
```

它将会列出所有已设置的断点，每一个断点都有一个标号，用来代表这个断点。例如，第 2 个断点设置是一个观察点，并且会忽略三次。

断点设置

根据行号设置断点

```
Plaintext
b 9  #break 可简写为 b
```

或者

```
Plaintext
b test.c:9
```

程序运行到第 9 行的时候会断住。

根据函数名设置断点

同样可以将断点设置在函数处：

```
Plaintext
b printNum
```

程序在调用到 printNum 函数的时候会断住。

根据条件设置断点

假设程序某处发生崩溃，而崩溃的原因怀疑是某个地方出现了非期望的值，那么你就可以在这里断点观察，当出现该非法值时，程序断住。这个时候我们可以借助 gdb 来设置条件断点，例如：

```
Plaintext
break test.c:23 if b==0
```

当在 b 等于 0 时，程序将会在第 23 行断住。 它和 condition 有着类似的作用，假设上面的断点号为 1，那么：

```
Plaintext
condition 1 b==0
```

会使得 b 等于 0 时，产生断点 1。而实际上可以很方便地用来改变断点产生的条件，例如，之前设置 b==0 时产生该断点，那么使用 condition 可以修改断点产生的条件。

根据规则设置断点

例如需要对所有调用 printNum 函数都设置断点，可以使用下面的方式：

```
Plaintext
rbreak printNum*
```

所有以 printNum 开头的函数都设置了断点。而下面是对所有函数设置断点：

#用法：rbreak file:regex

```
Plaintext
rbreak .
rbreak test.c:. #对 test.c 中的所有函数设置断点
rbreak test.c:^print #对以 print 开头的函数设置断点
```

建立捕捉断点

捕捉断点的作用是，监控程序中某一事件的发生，例如程序发生某种异常时、某一动态库被加载时等等，一旦目标时间发生，则程序停止执行。

建立捕捉断点的方式很简单，就是使用 catch 命令，其基本格式为：

(gdb) catch event

其中，event 参数表示要监控的具体事件。对于使用 GDB 调试 C、C++ 程序，常用的 event 事件类型如表 1 所示。

event 事件	含 义
throw [exception]	当程序中抛出 exception 指定类型异常时，程序停止执行。如果不指定异常类型（即省略 exception），则表示只要程序发生异常，程序就停止执行。
	当程序中捕获

点击图片可查看完整电子表格

设置临时断点

假设某处的断点只想生效一次，那么可以设置临时断点，这样断点后面就不复存在了：

```
Plaintext
tbreak test.c:10  #在第 10 行设置临时断点
```

跳过多次设置断点

假如有某个地方，我们知道可能出错，但是前面 30 次都没有问题，虽然在该处设置了断点，但是想跳过前面 30 次，可以使用下面的方式：

```
Plaintext
ignore 1 30
```

其中，1 是你要忽略的断点号，可以通过前面的方式查找到，30 是需要跳过的次数。这样设置之后，会跳过前面 30 次。再次通过 info breakpoints 可以看到：

```
Plaintext
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004005e8 in printNum2 at
test.c:16
    ignore next 30 hits
```

根据表达式值变化产生断点

有时候我们需要观察某个值或表达式，知道它什么时候发生变化了，这个时候我们可以借助 watch 命令。例如：

```
Plaintext
watch a
```

这个时候，让程序继续运行，如果 a 的值发生变化，则会打印相关内容，如：

```
Plaintext
Hardware watchpoint 2: a
Old value = 12
New value = 11
```

但是这里要特别注意的是，程序必须运行起来，否则会出现：

```
Plaintext
No symbol "a" in current context.
```

因为程序没有运行，当前上下文也就没有相关变量信息。

rwatch 和 awatch 同样可以设置观察点前者是当变量值被读时断住，后者是被读或者被改写时断住。

禁用或启动断点

有些断点暂时不想使用，但又不想删除，可以暂时禁用或启用。例如：

```
Plaintext
disable  #禁用所有断点
disable bnum #禁用标号为 bnum 的断点
enable  #启用所有断点
enable bnum #启用标号为 bnum 的断点
enable delete bnum  #启动标号为 bnum 的断点，并且在此之后删除该断点
```

断点清除

断点清除主要用到 clear 和 delete 命令。常见使用如下：

```
Plaintext
clear   #删除当前行所有 breakpoints
clear function  #删除函数名为 function 处的断点
clear filename:function #删除文件 filename 中函数 function 处的断点
clear lineNum #删除行号为 lineNum 处的断点
clear f:lename: lineNum #删除文件 filename 中行号为 lineNum 处的断点
delete  #删除所有 breakpoints,watchpoints 和 catchpoints
```

```
delete bnum #删除断点号为 bnum 的断点
```

变量查看

前言

在启动调试以及设置断点之后，就到了我们非常关键的一步-查看变量。GDB 调试最大的目的之一就是走查代码，查看运行结果是否符合预期。既然如此，我们就不得不了解一些查看各种类型变量的方法，以帮助我们进一步定位问题。

普通变量查看

打印基本类型变量，数组，字符数组

最常见的使用便是使用 print（可简写为 p）打印变量内容。 例如，打印基本类型，数组，字符数组等直接使用 p 变量名即可：

```
Plaintext
(gdb) p a
$1 = 10
(gdb) p b
$2 = {1, 2, 3, 5}
(gdb) p c
$3 = "hello,shouwang"
(gdb)
```

当然有时候，多个函数或者多个文件会有同一个变量名，这个时候可以在前面加上函数名或者文件名来区分：

```
Plaintext
(gdb) p 'testGdb.h'::a
$1 = 11
(gdb) p 'main'::b
$2 = {1, 2, 3, 5}
(gdb)
```

这里所打印的 a 值是我们定义在 testGdb.h 文件里的，而 b 值是 main 函数中的 b。

打印指针指向内容

如果还是使用上面的方式打印指针指向的内容，那么打印出来的只是指针地址而已，例如：

```
Plaintext
(gdb) p d
$1 = (int *) 0x602010
(gdb)
```

而如果想要打印指针指向的内容，需要解引用：

```
Plaintext
(gdb) p *d
$2 = 0
(gdb) p *d@10
$3 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb)
```

从上面可以看到，仅仅使用*只能打印第一个值，如果要打印多个值，后面跟上@并加上要打印的长度。 或者@后面跟上变量值：

```
Plaintext
(gdb) p *d@a
$2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb)
```

由于 a 的值为 10，并且是作为整型指针数据长度，因此后面可以直接跟着 a，也可以打印出所有内容。

另外值得一提的是，$可表示上一个变量，而假设此时有一个链表 linkNode，它有 next 成员代表下一个节点，则可使用下面方式不断打印链表内容：

```
Plaintext
(gdb) p *linkNode
(这里显示 linkNode 节点内容)
(gdb) p *$.next
(这里显示 linkNode 节点下一个节点的内容)
```

如果想要查看前面数组的内容，你可以将下标一个一个累加，还可以定义一个类似 UNIX 环境变量，例如：

```
Plaintext
(gdb) set $index=0
(gdb) p b[$index++]
$11 = 1
(gdb) p b[$index++]
$12 = 2
(gdb) p b[$index++]
```

```
$13 = 3
```

这样就不需要每次修改下标去打印啦。

按照特定格式打印变量

对于简单的数据，print 默认的打印方式已经足够了，它会根据变量类型的格式打印出来，但是有时候这还不够，我们需要更多的格式控制。常见格式控制字符如下：

- x 按十六进制格式显示变量。
- d 按十进制格式显示变量。
- u 按十六进制格式显示无符号整型。
- o 按八进制格式显示变量。
- t 按二进制格式显示变量。
- a 按十六进制格式显示变量。
- c 按字符格式显示变量。
- f 按浮点数格式显示变量。

还是以辅助程序来说明，正常方式打印字符数组 c：

```
Plaintext
(gdb) p c
$18 = "hello,shouwang"
```

但是如果我们要查看它的十六进制格式打印呢？

```
Plaintext
(gdb) p/x c
$19 = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x73, 0x68, 0x6f, 0x75,
0x77, 0x61,
  0x6e, 0x67, 0x0}
(gdb)
```

但是如果我们想用这种方式查看浮点数的二进制格式是怎样的是不行的，因为直接打印它首先会被转换成整型，因此最终会得到 8：

```
Plaintext
(gdb) p e
$1 = 8.5
(gdb) p/t e
$2 = 1000
```

```
(gdb)
```

那么就需要另外一种查看方式了。

查看内存内容

examine(简写为 x)可以用来查看内存地址中的值。语法如下：

```
Plaintext
x/[n][f][u] addr
```

其中：

- n 表示要显示的内存单元数，默认值为 1
- f 表示要打印的格式，前面已经提到了格式控制字符
- u 要打印的单元长度
- addr 内存地址

单元类型常见有如下：

- b 字节
- h 半字，即双字节
- w 字，即四字节
- g 八字节

我们通过一个实例来看，假如我们要把 float 变量 e 按照二进制方式打印，并且打印单位是一字节：

```
Plaintext
(gdb) x/4tb &e
0x7fffffffdbd4:    00000000    00000000    00001000    01000001
(gdb)
```

可以看到，变量 e 的四个字节都以二进制的方式打印出来了。

自动显示变量内容

假设我们希望程序断住时，就显示某个变量的值，可以使用 display 命令。

```
Plaintext
(gdb) display e
1: e = 8.5
```

那么每次程序断住时，就会打印 e 的值。要查看哪些变量被设置了 display，可以使用：

```
Plaintext
(gdb)into display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  b
2:   y  e
```

如果想要清除可以使用

```
Plaintext
delete display num #num 为前面变量前的编号,不带 num 时清除所有。
```

或者去使能：

```
Plaintext
disable display num  #num 为前面变量前的编号，不带 num 时去使能所有
```

查看寄存器内容

```
Plaintext
(gdb)info registers
rax            0x0    0
rbx            0x0    0
rcx            0x7ffff7dd1b00    140737351850752
rdx            0x0    0
rsi            0x7ffff7dd1b30    140737351850800
rdi            0xffffffff    4294967295
rbp            0x7fffffffdc10    0x7fffffffdc10
(内容过多未显示完全)
```

单步调试

前言

前面通过《启动调试》,《断点设置》,《变量查看》,我们已经了解了 GDB 基本的启动，设置断点，查看变量等，如果这些内容你还不知道，建议先回顾一下前面的内容。在启动调试设置断点观察之后，没有我们想要的信息怎么办呢？这个时候，就需要单步执行或者跳过当前断点继续执行等等。而本文所说的单步调试并非仅仅指单步执行，而是指在你的控制之下，按要求执行语句。

单步执行-next

next 命令（可简写为 n）用于在程序断住后，继续执行下一条语句，假设已经启动调试，并在第 12 行停住，如果要继续执行，则使用 n 执行下一条语句，如果后面跟上数字 num，则表示执行该命令 num 次，就达到继续执行 n 行的效果了：

```
Plaintext
$ gdb gdbStep   #启动调试
(gdb)b 25       #将断点设置在 12 行
(gdb)run        #运行程序
Breakpoint 1, main () at gdbStep.c:25
25        int b = 7;
(gdb) n     #单步执行
26        printf("it will calc a + b\n");
(gdb) n 2   #执行两次
it will calc a + b
28        printf("%d + %d = %d\n",a,b,c);
(gdb)
```

从上面的执行结果可以看到，我们在 25 行处断住，执行 n 之后，运行到 26 行，运行 n 2 之后，运行到 28 行，但是有没有发现一个问题，为什么不会进入到 add 函数内部呢？那就需要用到另外一个命令啦。

单步进入-step

对于上面的情况，如果我们想跟踪 add 函数内部的情况，可以使用 step 命令（可简写为 s），它可以单步跟踪到函数内部，但前提是该函数有调试信息并且有源码信息。

```
Plaintext
$ gdb gdbStep    #启动调试
(gdb) b 25       #在 12 行设置断点
Breakpoint 1 at 0x4005d3: file gdbStep.c, line 25.
(gdb) run        #运行程序
Breakpoint 1, main () at gdbStep.c:25
25        int b = 7;
(gdb) s
26        printf("it will calc a + b\n");
(gdb) s     #单步进入，但是并没有该函数的源文件信息
_IO_puts (str=0x4006b8 "it will calc a + b") at ioputs.c:33
33    ioputs.c: No such file or directory.
(gdb) finish    #继续完成该函数调用
Run till exit from #0  _IO_puts (str=0x4006b8 "it will calc a +
b")
```

```
    at ioputs.c:33
it will calc a + b
main () at gdbStep.c:27
27        int c = add(a,b);
Value returned is $1 = 19
(gdb) s        #单步进入，现在已经进入到了 add 函数内部
add (a=13, b=57) at gdbStep.c:6
6        int c = a + b;
```

从上面的过程可以看到，s 命令会尝试进入函数，但是如果没有该函数源码，需要跳过该函数执行，可使用 finish 命令，继续后面的执行。如果没有函数调用，s 的作用与 n 的作用并无差别，仅仅是继续执行下一行。它后面也可以跟数字，表明要执行的次数。

当然它还有一个选项，用来设置当遇到没有调试信息的函数，s 命令是否跳过该函数，而执行后面的。默认情况下，它是会跳过的，即 step-mode 值是 off:

```
Plaintext
(gdb) show step-mode
Mode of the step operation is off.
(gdb) set step-mode on
(gdb) set step-mode off
```

还有一个与 step 相关的命令是 stepi（可简写为 si），它与 step 不同的是，每次执行一条机器指令:

```
Plaintext
(gdb) si
0x0000000000400573    6       int c = a + b;
(gdb) display/i $pc
1: x/i $pc
=> 0x400573 <add+13>:    mov    -0x18(%rbp),%eax
(gdb)
```

继续执行到下一个断点-continue

我们可能打了多处断点，或者断点打在循环内，这个时候，想跳过这个断点，甚至跳过多次断点继续执行该怎么做呢？可以使用 continue 命令（可简写为 c）或者 fg，它会继续执行程序，直到再次遇到断点处:

```
Plaintext
$ gdb gdbStep
(gdb)b 18    #在 count 函数循环内打断点
(gdb)run
```

```
Breakpoint 1, count (num=10) at gdbStep.c:18
18            i++;
(gdb) c      #继续运行，直到下一次断住
Continuing.
1

Breakpoint 1, count (num=10) at gdbStep.c:18
18            i++;
(gdb) fg     #继续运行，直到下一次断住
Continuing.
2

Breakpoint 1, count (num=10) at gdbStep.c:18
18            i++;
(gdb) c 3    #跳过三次
Will ignore next 2 crossings of breakpoint 1.  Continuing.
3
4
5

Breakpoint 1, count (num=10) at gdbStep.c:18
18            i++;
```

继续运行到指定位置-until

假如我们在 25 行停住了，现在想要运行到 29 行停住，就可以使用 until 命令（可简写为 u）：

```
Plaintext
$ gdb gdbStep
(gdb)b 25
(gdb)run
(gdb) u 29
it will calc a + b
3 + 7 = 10
main () at gdbStep.c:29
29        count(c);
(gdb)
```

可以看到，在执行 u 29 之后，它在 29 行停住了。它利用的是临时断点。

跳过执行--skip

skip 可以在 step 时跳过一些不想关注的函数或者某个文件的代码:

```
Plaintext
$ gdb gdbStep
(gdb) b 27
Breakpoint 1 at 0x4005e4: file gdbStep.c, line 27.
(gdb) skip function add    #step 时跳过 add 函数
Function add will be skipped when stepping.
(gdb) info skip   #查看 step 情况
Num     Type           Enb What
1       function       y   add
(gdb) run
Starting program: /home/hyb/workspaces/gdb/gdbStep
it will calc a + b

Breakpoint 1, main () at gdbStep.c:27
27        int c = add(a,b);
(gdb) s
28        printf("%d + %d = %d\n",a,b,c);
(gdb)
```

可以看到，再使用 skip 之后，使用 step 将不会进入 add 函数。 step 也后面也可以跟文件:

```
Plaintext
(gdb)skip file gdbStep.c
```

这样 gdbStep.c 中的函数都不会进入。

其他相关命令:

- skip delete [num] 删除 skip
- skip enable [num] 使能 skip
- skip disable [num] 去使能 skip

其中 num 是前面通过 info skip 看到的 num 值，上面可以带或不带该值，如果不带 num，则针对所有 skip，如果带上了，则只针对某一个 skip。

源码查看

前言

我们在调试过程中难免要对照源码进行查看，如果已经开始了调试，而查看源码或者编辑源码却要另外打开一个窗口，那未免显得太麻烦。文本将会介绍如何在 GDB 调试模式下查看源码或对源码进行编辑。

列出源码

首先要介绍的就是 list 命令（可简写为 l），它用来打印源码。

直接打印源码

例如：

```
Plaintext
$ gdb main
(gdb) l
1    //main.c
2    #include<stdio.h>
3    #include"test.h"
4    int main(void)
5    {
6        printf("it will print from 5 to 1\n");
7        printNum(5);
8        printf("print end\n");
9
10        printf("it will print 1 to 5\n");
(gdb)
```

直接输入 l 可从第一行开始显示源码，继续输入 l，可列出后面的源码。后面也可以跟上+或者-，分别表示要列出上一次列出源码的后面部分或者前面部分。

列出指定行附近源码

l 后面可以跟行号，表明要列出附近的源码：

```
Plaintext
(gdb) l 9
4    int main(void)
5    {
6        printf("it will print from 5 to 1\n");
7        printNum(5);
8        printf("print end\n");
9
10        printf("it will print 1 to 5\n");
11        printNum1(5);
12        printf("print end\n");
13        return 0;
```

在这里，l 后面跟上 9，表明要列出第 9 行附近的源码。

列出指定函数附近的源码

这个很容易理解，而使用也很简单，l 后面跟函数名即可，例如：

```
Plaintext
(gdb) l printNum
1    #include"test.h"
2    void printNum(int n)
3    {
4        if( n < 0)
5            return;
6        while(n > 0)
7        {
8            printf("%d\n",n);
9            n--;
10        }
```

在这里，l 后面跟上函数名 printNum，它便列出了 printNum 函数附近的源码。

设置源码一次列出行数

不知道你有没有发现，在列出函数源码的时候，它并没有列全，因为 l 每次只显示 10 行，那么有没有方法每次列出更多呢？ 我们可以通过 listsize 属性来设置，例如设置每次列出 20 行：

```
Plaintext
(gdb) set listsize 20
(gdb) show listsize
Number of source lines gdb will list by default is 20.
```

这样每次就会列出 20 行，当然也可以设置为 0 或者 unlimited，这样设置之后，列出就没有限制了，但源码如果较长，查看将会不便。

列出指定行之间的源码

list first,last 例如，要列出 3 到 15 行之间的源码：

```
Plaintext
(gdb) l 3,15
3    {
4        if( n < 0)
5            return;
6        while(n > 0)
7        {
```

```
8            printf("%d\n",n);
9            n--;
10        }
11    }
12
13    void printNum1(int n)
14    {
15        if( n < 0)
```

启始行和结束行号之间用逗号隔开。两者之一也可以省略，例如：

```
Plaintext
(gdb) list 3,
3    {
4        if( n < 0)
5            return;
6        while(n > 0)
7        {
8            printf("%d\n",n);
9            n--;
10        }
11    }
12
```

省略结束行的时候，它列出从开始行开始，到指定大小行结束，而省略开始行的时候，到结束行结束，列出设置的大小行，例如默认设置为 10 行，则到结束行为止，总共列出 10 行。前面我们也介绍了修改和查看默认列出源码行数的方法。

列出指定文件的源码

前面执行 l 命令时，默认列出 main.c 的源码，如果想要看指定文件的源码呢？可以

```
Plaintext
l location
```

其中 location 可以是文件名加行号或函数名，因此可以使用：

```
Plaintext
(gdb) l test.c:1
1    #include"test.h"
2    void printNum(int n)
3    {
4        if( n < 0)
5            return;
```

```
6         while(n > 0)
7         {
8             printf("%d\n",n);
9             n--;
10         }
(gdb)
```

来查看指定文件指定行，或者指定文件指定函数：

```
Plaintext
(gdb) l test.c:printNum1
9             n--;
10        }
11    }
12
13    void printNum1(int n)
14    {
15        if( n < 0)
16            return;
17        int i = 1;
18        while(i <= n)
(gdb)
```

或者指定文件指定行之间：

```
Plaintext
(gdb) l test.c:1,test.c:3
1     #include"test.h"
2     void printNum(int n)
3     {
(gdb)
```

指定源码路径

在查看源码之前，首先要确保我们的程序能够关联到源码，一般来说，我们在自己的机器上加上-g 参数编译完之后，使用 gdb 都能查看到源码，但是如果出现下面的情况呢？

源码被移走

例如，我现在将 main.c 移动到当前的 temp 目录下，再执行 l 命令：

```
Plaintext
(gdb) l
```

```
1    main.c: No such file or directory.
(gdb)
```

它就会提示找不到源码文件了，那么怎么办呢？ 我们可以使用 dir 命名指定源码路径，例如：

```
Plaintext
(gdb) dir ./temp
Source directories searched:
/home/hyb/workspaces/gdb/sourceCode/./temp:$cdir:$cwd
```

这个时候它就能找到源码路径了。我这里使用的是相对路径，保险起见，你也可以使用绝对路径。

更换源码目录

例如，你编译好的程序文件，放到了另外一台机器上进行调试，或者你的源码文件全都移动到了另外一个目录，怎么办呢？当然你还可以使用前面的方法添加源码搜索路径，也可以使用 set substitute-path from to 将原来的路径替换为新的路径，那么我们如何知道原来的源码路径是什么呢？借助 readelf 命令可以知道：

```
Plaintext
$ readelf main -p .debug_str
  [     0]  long unsigned int
  [    12]  short int
  [    1c]  /home/hyb/workspaces/gdb/sourceCode
  [    40]  main.c
（显示部分内容）
```

main 为你将要调试的程序名，这里我们可以看到原来的路径，那么我们现在替换掉它：

```
Plaintext
(gdb) set substitute-path /home/hyb/workspaces/gdb/sourceCode
/home/hyb/workspaces/gdb/sourceCode/temp
(gdb) show substitute-path
List of all source path substitution rules:
  `/home/hyb/workspaces/gdb/sourceCode' ->
`/home/hyb/workspaces/gdb/sourceCode/temp'.
(gdb)
```

设置完成后，可以通过 show substitute-path 来查看设置结果。这样它也能在正确的路径查找源码啦。

需要注意的是，这里对路径做了字符串替换，那么如果你有多个路径，可以做多个替

换。甚至可以对指定文件路径进行替换。

最后你也可以通过 unset substitute-path [path]取消替换。

编辑源码

为了避免已经启动了调试之后，需要编辑源码，又不想退出，可以直接在 gdb 模式下编辑源码，它默认使用的编辑器是/bin/ex，但是你的机器上可能没有这个编辑器，或者你想使用自己熟悉的编辑器，那么可以通过下面的方式进行设置：

```
Plaintext
$ EDITOR=/usr/bin/vim
$ export EDITOR
```

/usr/bin/vim 可以替换为你熟悉的编辑器的路径，如果你不知道你的编辑器在什么位置，可借助 whereis 命令或者 witch 命令查看：

```
Plaintext
$ whereis vim
vim: /usr/bin/vim /usr/bin/vim.tiny /usr/bin/vim.basic
/usr/bin/vim.gnome /etc/vim /usr/share/vim
/usr/share/man/man1/vim.1.gz
$ which vim
/usr/bin/vim
```

设置之后，就可以在 gdb 调试模式下进行编辑源码了，使用命令 edit location，例如：

```
Plaintext
(gdb)edit 3  #编辑第三行
(gdb)edit printNum #编辑 printNum 函数
(gdb)edit test.c:5 #编辑 test.c 第五行
```

可自行尝试，这里的 location 和前面介绍的一样，可以跟指定文件的特定行或指定文件的指定函数。 编辑完保存后，别忘了重新编译程序：

```
Plaintext
(gdb)shell gcc -g -o main main.c test.c
```

这里要注意，为了在 gdb 调试模式下执行 shell 命令，需要在命令之前加上 shell，表明这是一条 shell 命令。这样就能在不用退出 GDB 调试模式的情况下编译程序了。

另外一种模式

启动时，带上 tui(Text User Interface)参数，会有意想不到的效果，它会将调试在多个

文本窗口呈现：

```
Plaintext
gdb main -tui
```

查看栈信息

当程序因某种异常停止运行时，我们要做的就是找到程序停止的具体位置，分析导致程序停止的原因。

对于 C、C++ 程序而言，异常往往出现在某个函数体内，例如 main() 主函数、调用的系统库函数或者自定义的函数等。要知道，程序中每个被调用的函数在执行时，都会生成一些必要的信息，包括：

- 函数调用发生在程序中的具体位置；
- 调用函数时的参数；
- 函数体内部各局部变量的值等等。

这些信息会集中存储在一块称为“栈帧”的内存空间中。也就是说，程序执行时调用了多少个函数，就会相应产生多少个栈帧，其中每个栈帧自函数调用时生成，函数调用结束后自动销毁。

注意，这些栈帧所在的位置也不是随意的，它们集中位于一个大的内存区域里，我们通常将其称为栈区或者栈。

GDB 调试器并没有套用地址标识符的方式来管理栈帧。对于当前调试环境中存在的栈帧，GDB 调试器会按照既定规则对它们进行编号：当前正被调用函数对应的栈帧的编号为 0，调用它的函数对应栈帧的编号为 1，以此类推。

frame 命令的常用形式有 2 个：

选择栈深度

根据栈帧编号或者栈帧地址，选定要查看的栈帧，语法格式如下：

```
Plaintext
(gdb) frame spec
```

该命令可以将 spec 参数指定的栈帧选定为当前栈帧。spec 参数的值，常用的指定方法有 3 种：

- 通过栈帧的编号指定。0 为当前被调用函数对应的栈帧号，最大编号的栈帧对应的函数通常就是 main() 主函数；
- 借助栈帧的地址指定。栈帧地址可以通过 info frame 命令（后续会讲）打印出的

信息中看到；

- 通过函数的函数名指定。注意，如果是类似递归函数，其对应多个栈帧的话，通过此方法指定的是编号最小的那个栈帧。

除此之外，对于选定一个栈帧作为当前栈帧，GDB 调试器还提供有 up 和 down 两个命令。其中，up 命令的语法格式为：

```
Plaintext
(gdb) up n
```

其中 n 为整数，默认值为 1。该命令表示在当前栈帧编号（假设为 m）的基础上，选定 m+n 为编号的栈帧作为新的当前栈帧。

相对地，down 命令的语法格式为：

```
Plaintext
(gdb) down n
```

其中 n 为整数，默认值为 1。该命令表示在当前栈帧编号（假设为 m）的基础上，选定 m-n 为编号的栈帧作为新的当前栈帧。

显示栈存储内容

我们可以查看当前栈帧中存储的信息：

显示当前栈内容

(gdb) info frame

该命令会依次打印出当前栈帧的如下信息：

- 当前栈帧的编号，以及栈帧的地址；
- 当前栈帧对应函数的存储地址，以及该函数被调用时的代码存储的地址
- 当前函数的调用者，对应的栈帧的地址；
- 编写此栈帧所用的编程语言；
- 函数参数的存储地址以及值；
- 函数中局部变量的存储地址；
- 栈帧中存储的寄存器变量，例如指令寄存器（64 位环境中用 rip 表示，32 为环境中用 eip 表示）、堆栈基指针寄存器（64 位环境用 rbp 表示，32 位环境用 ebp 表示）等。 除此之外，还可以使用 `info args` 命令查看当前函数各个参数的值；使用 `info locals` 命令查看当前函数中各局部变量的值。

查看所有栈内容

backtrace 命令用于打印当前调试环境中所有栈帧的信息，常用的语法格式如下：

```
Plaintext
(gdb) backtrace [-full] [n]
```

其中，用 [ ] 括起来的参数为可选项，它们的含义分别为：

- n：一个整数值，当为正整数时，表示打印最里层的 n 个栈帧的信息；n 为负整数时，那么表示打印最外层 n 个栈帧的信息；
- -full：打印栈帧信息的同时，打印出局部变量的值。

扩展阅读

GDB 调试动态链接库

1. 编译动态库时加入-g 选项

check 动态库是否加入-g，则表示成功

```
C++
gdb libxxx.so
reading symbos form xxxx
```

2. 在 main 处设置断点，并运行程序，

```
C++
(gdb) b main # 设置入口断点
(gdb) r [可执行程序的命令行参数] # 启动调试
(gdb) load <要调试的动态库，如 test.so>        # 将动态库加载入内存

(gdb) dir <要调试的动态库的源码路径，如 ./src>

(gdb) sharedlibrary <要调试的动态库> # 将动态库的符号读入 gdb，为了你能找到变量和函数名

(gdb) breakpoint <要调试动态库的断点位置,如 src/test.cpp:100 >

(gdb) c # 运行至设置的动态库中的断点
```

GDB 多线程调试

复杂的程序往往都涉及多线程，这里简单介绍一下多线程 GDB，但是遇到多线程 DEBUG 还是更推荐打印日志的方式

调试多线程的相关指令

- 显示所有进程

```
Plaintext
info threads
```
这条指令显示可以调试的所有线程，gdb 会为每个线程分配一个 ID,这个 ID 和线程 ID 不同，ID 号一般从 1 开始。

- 切换线程指令

```
Plaintext
```
为什么要切换线程呢？因为 gdb 默认在调试多线程程序时只跟踪主线程，而新创建的线程都阻塞在线程创建函数 pthread_create 处。如果我们不切换，那么我们就无法调试我们在主线程中所创建的线程。

```
        thread ID
```
这条指令用来切换当前调试的线程为指定的 ID 号的线程，ID 为 GDB 为线程所分配的 ID.

- 多个线程控制指令

```
Plaintext
//指定多个线程执行 command 指令，其中 command 为 gdb 中的指令
thread apply ID1 ID2 ...IDn command
//指定所有线程执行 command 指令，其中 command 为 gdb 中的指令
thread apply all command
```

- 线程锁

```
Plaintext
//线程锁
  set scheduler-locking off|on
```
off:不锁定任何线程，当程序继续运行的时候如果有断点，那么就把所有的线程都暂停下来，直到你指定某个线程继续执行，如果在当前线程下使用 **continue** 的话那么会启动所有线程（GDB 默认）；

on:打开线程锁，锁定其他线程，只有当前线程执行。

non-stop 模式
为了对上面的这种一个线程中断在一个断点上，其他所有的线程都会被 freeze.
gdb v7.0 引入了 non-stop 模式，在这个模式下：
1. 当某个或多个线程在一个断点上，其他线程仍会并行运行
2. 你可以选择某个被中断的线程，只让他运行。
3. non-stop 模式表示不停止模式，除了断点有关的进程会被停下来，其他线程会继续执行。
设置 non-stop 模式，打开 gdb 后，在开始 r 之前，首先连续输入下面的指令

```
set target-async 1
set pagination off
set non-stop on
```