不只是gdb

> 本文提供基础的gdb应用指南，以及鄙人的一些小小的调试经验。

宏观理解一下调试

大一过了，可能还有同学不知道调试是什么，只是说着debug，但是实际上是一遍一遍地看着自己写的代码。这样子的程序开发效率是很低的。而且，开始实现agenda后，我们可以发现，肉眼debug的方法不再十分适用，因为大家都觉得自己写的算法很对。

因此我们应该要学会调试，这里说的调试，**大概指的是借助机器找到我们代码里面的缺漏**，一定要知道 `调试 ！= 会用gdb` ，重要的是要掌握调试的一些套路与经验，gdb只是一个给我们调试的工具而已。因此我们在了解gdb前应该要更新一些观念（调试大神可以跳过：））

1. bug

- bug 有两种 ：
    - 灾难性的： 出现了这种bug之后程序会直接崩溃，比如说，段错误。
    - 非灾难性的 ：出现这种bug并不会使程序崩溃，但是程序运行的结果与我们预期的并不一样，比如说，一些算法的实现出现了错误。
- bug 的另外一种分类方法：
    - 可重复的：就是可以不断复现的错误。
    - 不可重复的：偶然出现的不可重复的bug，笔者现在没有遇到过，可能是涉及到的代码工程还不够大，只是看书看到。

2.断点

在调试工具中，程序运行到断点行就会停下。

3.调试的一般思路

> 调试通俗来讲应该是：不断地重复，在程序运行到某个地方的时候停下来看看那个时候各个值有没有符合预期

1. 重现bug

要重现bug，**首先采用与bug第一次出现时完全相同的输入**。注意不要只执行触发操作，因为bug可能是由整个操作序列产生的。

当能一致地重现这个bug时，应**尝试找出触发这个bug的最小序列代码**，可以从仅包含触发操作的最小序列开始，然后慢慢扩大至覆盖启动时的完整序列代码，直到bug被触发时。 –> **这会得到重现这个bug的最简单高效的测试用例。**

2.调试可重复的bug

目标是找到触发这个问题的准确的代码行。

正所谓，“眼见为实”，调试可重复的bug，很多情况下是自己的算法实现出现问题，这种bug十分隐秘，因为可能是你在coding中任何出现的小问题。这个时候，不要过于相信自己，事实上，你的代码输出就是错的，**所以就不要抱怨自己的代码没有问题了**，立刻开始调试吧。

调试这种类型的bug主要有两种方法：

1. **记录调试信息**，通过观察调试信息来判断bug出现的地点,也就是有一部分同学喜欢用的cout大法，在一定的位置把一些相关量的内容输出出来。不过若手边有一个现成的调试器，不建议加上调试信息，因为调试信息的修改与还原比较麻烦 （ 当然可以使用宏定义来简化这个过程，详细情况可以自行百度 ）。
2. **直接使用调试器来进行调试**， 比如说，我们等一下要谈到的gdb： ）

3. 调试不可重复的bug

这种bug的调试非常困难，因为**很难重现bug出现时的情景**（只能使用自己的经验进行大胆的猜测）。。。所以调试这种bug，手边没有什么特别厉害的调试工具时，只能够通过检查代码来发现问题，不过有趣的时，这种方法时时也竟会比较有效。因为我们带着刚才发生bug的视角再来重新审视一遍代码时，往往会有很大可能发现出问题所在。
不过，**我们在这里并不建议长时间盯着代码看，而是手工跟踪代码的执行路径**

4.调试内存问题

不要解除对nullptr指针的引用
一般内存错误的检查：

1. 确保每一个new的调用都匹配了一个delete的调用。同样，每一个对malloc、alloc和calloc的调用都要匹配一个对free的调用。每一个new[]的调用也要匹配一个对delete[]的调用。**为了避免双重释放内存或使用已释放的内存，建议释放内存后将指针设置为空**
2. **检查缓冲区的溢出**，每次迭代访问一个数组或者读写一个c风格的字符串的时候，验证没有越过数组尾部访问的内存。通过使用STL容器和字符串通常可以避免此类问题
3. **检查无效指针的解除引用**
4. 确保总是在类的构造函数中初始化指针数据成员，既可以在构造函数中分配内存也可以将指针设置为nullptr。

5. **在堆栈上声明指针的时候，一定要确保在声明中初始化指针**，如：

```
T* p = nullptr;
//或 T* p = new T;
//不要 T* p;
```

（ps：内存错误并不总是会立刻出现，它有可能是程序中各个代码段综合作用的结果，往往需要调试器和一点耐心才可以。）

4.测例

测试的样例，并不是随意想的，重点是要想**边界情况**，即**极限情况**

5.找到适合自己的调试方法

除了gdb，或许，各位比较喜欢那种图形界面方式的，像VC、BCB等IDE的调试，或者是google-test，我们应该要在平时的调试实践中积累经验，选择适合自己使用的方法。

---

gdb的介绍

> gdb 博大精深，鄙人不知自己掌握了多少，但是，基本的使用还是能够保证的，这里提供的是我自己使用的比较多的，自己觉得比较有用的命令 ： ）

1.GDB概述

GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具。主要调试C/C++等用gcc/g++编译出来的程序。

一般来说，GDB主要帮忙你完成下面四个方面的功能：

- 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。（断点可以是条件表达式）
- 当程序被停住时，可以检查此时你的程序中所发生的事。
- 动态的改变你程序的执行环境。

2. 基本使用

前期准备

（通过一个例子来说明）

1. 在编译指令里记得加上 `-g` 参数，这个会使得编译出来的程序可以使用gdb进行调试
   ```
   $ g++ -g test.cpp -o test
   ```
2. gdb载入可执行文件，这里有两个方法

   直接运行 `gdb test`
   或：

   ```
   gdb + file test
   ```

```
$ gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.h
tml>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copyin
g"
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".
(gdb) gdb tes
Undefined command: "gdb".  Try "help".
(gdb) gdb file test
Undefined command: "gdb".  Try "help".
(gdb) file test
Reading symbols from test...done.
(gdb)
```

只有导入成功了之后才能进行调试!!

基本命令

基本命令是整个gdb的使用基础，如果一开始记不住，可以把它记下来，要用的时候就查阅，逐渐熟能生巧。

- `l` ：输出代码，输出当前上下各10行的代码
- `b 行数` ：在那一行上设置断点
- `r`：运行整个程序，直至遇到断点

- `s` : 单步继续执行（进入函数）
- `n` : 单步继续进行 (不进入函数，直接把子函数一并运行完)
- `q` : 退出gdb
- `d num` ： num为断点编号， 删除该断点
- `c` : continue，继续运行直至下一个断点
- `p exp` ： 查看变量exp的内容
- `k` ：kill掉当前运行的程序（然后再用r来重新调试）

3. 进阶使用

breakpoint进阶

- `b 行数 if i > 9` :增加条件， 当 i > 9的时候，程序运行到num行会停下来。
- `info break` ： 查看断点信息，包括所有断点号，禁用情况等
- 删除breakpoint：
    - `clear 行数` ：删除该行上的所以断点
    - `d` ：删除所有的断点
    - `d 1-6` ：删除编号是1 ~ 6的断点

- 禁用断点相关：
    - `disable num` :禁用某个断点
    - `disable` :全部禁用
    - `enable num` :启用某个断点
    - `enable` ：全部启用

- 在其他文件里面设置断点
    - `b 文件名 ： 函数名/行数` ：for example:

```
(gdb) b AgendaService.cpp : AgendaService::startAgenda()
Breakpoint 1 at 0x40a0d4: file src/AgendaService.cpp, line 208.
```

设置观察点（数据断点）：

`watch 变量/表达式`

- 当观察点的内容有变化的时候，即立刻停止
- 用delete来删除观察点
- `info watchpoints` 来查看所有观察点的信息

p 命令的说明

- p 后的变量必须是全局变量或者是当前可见的局部变量，如果两者同名，则优先查看局部变量

- p 还可以显示其他数据结构，比如说，在调试agenda的时候，可以直接 `p startdate` ，这样就可以直接看startdate这个Date类里面的全部内容了，是不是比cout startdate的所有成员方便很多呢？：）

ptype 命令

通过 `ptype var` 可以查看var的数据类型

bt 命令

bt 查看程序crash堆栈信息
当程序被停住了，你需要做的第一件事就是查看程序是在哪里住的。当你的程序调用了一个函数，函数的地址，函数参数，数内的局部变量都会被压入“栈”（Stack）中。你可以用GDB令来查看当前的栈中的信息。
for exmaple:

```
(gdb) bt
 #0 func (n=250) at tst.c:6
 #1 0x08048524 in
 main (argc=1, argv=0xbffff674) at tst.c:30
 #2 0x40040Arrayed in __libc_start_main () from /lib/libc.so.6
```

这个命令在两种情况下非常有用：

1. 程序崩溃的时候，这个时候查看最近的堆栈消息，可以非常快地查看到是在哪里出错的。
2. 当你确认了一个函数有bug，这个函数被其他很多函数调用，这个时候，程序在这个函数停下来的时候使用bt命令，就可以知道是哪个函数调用了有问题的函数。（这个样例由**帅气的伟铭师兄**提供：））

display 命令

嫌每一次都要用 `p` 来查看同一个变量？可以使用
`display + var` 来使用自动显示var的值
for example :

```
#include <iostream>

int main(int argc, char const *argv[]) {
  for (int i = 0; i < 5; ++i) {
    std::cout << "haha" << std::endl;
  }
  return 0;
}
```

使用gdb 的display后的结果是：

```
Breakpoint 1, main (argc=1, argv=0x7fffffffdd08) at test.cpp:5
5       std::cout << "haha" << std::endl;
(gdb) display i
1: i = 0
(gdb) n
haha
4     for (int i = 0; i < 5; ++i) {
1: i = 0
(gdb) n

Breakpoint 1, main (argc=1, argv=0x7fffffffdd08) at test.cpp:5
5       std::cout << "haha" << std::endl;
1: i = 1
(gdb) n
haha
4     for (int i = 0; i < 5; ++i) {
1: i = 1
(gdb) n

Breakpoint 1, main (argc=1, argv=0x7fffffffdd08) at test.cpp:5
5       std::cout << "haha" << std::endl;
1: i = 2
```

这样就可以在每一步都看到i的值了。

不小心用s进入了不想进入的函数怎么办？

几个方法：

1. 耐心地按n直至函数执行完：）
2. `return` ：直接终止当前运行的函数，但没有提供返回值，可以在 `return` 的命令后面加上值var,强行指定返回var这个值。
3. `finish` ：直接模拟完成当前运行的函数，有返回值。

`set` 命令

举例:
`set tol=100` 这样可以在程序运行时把tol设置为100

4.学习思路

再一次说一下：

> 调试通俗来讲应该是：不断地重复，在程序运行到某个地方的时候停下来看看那个时候各个值有没有符合预期

所以，有关**断点的设置**和**数据输出**的命令应该是gdb的核心内容，，同学们应该重点掌握。

可能第二部分，大家全部看完有点头晕，但是，学习gdb是一个渐进的过程。初学gdb，可以先把命令记录下来，先尝试着使用几个基本的命令，熟练后再学习新的命令，一般新的命令在使用的时候是减少操作的次数，提高效率的。用得多了，自然也就熟了。

关键是要**自己动手，自己动手，自己动手！**

最后，祝大家早日熟练掌握gdb ：)

更加深入地了解gdb