gdb使用技巧

什么是gdb?

调试程序
程序中出现的语法错误可以借助编辑器解决，但逻辑错误只能靠自己解决。实际场景中解决逻辑错误最高效的方法，就是借助调试工具对程序进行调试。
所谓调试（Debug）就是让代码一步一步慢慢执行，跟踪程序的运行过程。比如，可以让程序停在某个地方，查看当前所有变量的值，或者**内存**中的数据，也可以让程序一次只执行一条或者几条语句，看看程序导致执行了那些代码。
也就是说，通过调试程序，我们可以监控程序执行的每一个细节，包括变量的值、函数的调用过程、内存中数据、线程的调度等，从而发现隐藏的错误或者低效的代码。

gdb的作用

GDB 全程 GNU symbolic debugger，从名称上不难看出，它诞生于 GNU 计划（同时诞生的还有 GCC、 Emacs 等），是 Linux 下常用的程序调试器。发展至今， GDB 已经迭代了诸多版本，当下 GDB 支持调试多种编程语言编写的程序，包括 C、 C++、 Go、 Objective-C、 OpenCL、 Ada 等。实际场景中, GDB 更常用于调试 C 和 C++ 程序。
总的来说，借助 GDB 调试器可以实现以下几个功能：

- 程序启动时，可以按照我们自定义的要求运行程序，例如设置参数和环境变量；
- 可以使调试程序在指定代码处暂停运行，并查看当前程序的运行状态（例如当前变量的值，函数执行结果等），即支持断点调试；

- 程序执行过程中，可以改变某个变量的值，还可以改变代码的执行顺序，从而尝试修改程序中出现的逻辑错误。

gdb的使用

编译时需要使用 gcc/g++ -g 选项编译源文件，才可生成满足 GDB 要求的可执行文件。
调试命令缩写：

- break(b) point
  在源码指定的某一行设置断点，其中 point 用于指定具体打断点的位置。
- run(r)
  执行被调用程序，会自动在第一个断点处暂停执行。
- continue(c)
  当程序在某一断点处停止后，用该指令可以继续执行，直到遇到断点或者程序结束。
- next(n)
  令程序一行代码一行代码的执行。
- step(s)
  如果有调用函数，进入调用函数的内部；否则和 next 调试命令功能一样。
- until(u)
  当你厌倦了在一个循环体内单步跟踪时，单纯使用 until 命令，可以运行程序直到退出循环体。
- until(u) n
  until n 命令中，n为某一行的行号，该命令会使程序运行至第n行代码处停止。
- print(p) value
  打印指定变量的值，其中 value 指的是某一变量名。
- list(l)
  显示源程序代码的内容，包括各行代码所在的行号。按 [Enter] 可以显示更多的行。
- finsh(fi)
  结束当前正在执行的函数，并在跳出函数后暂停程序的执行。
- return value
  结束当前调用函数并返回指定值，到上一层函数调用处停止程序执行。
- jump(j)
  使程序从当前要执行的代码处，直接跳到指定位置处继续执行后续的代码。
- quit(q)
  终止调试。

有一个示例程序如下所示：

```cpp
// main.cpp
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    int n = 1;
    while (n <= 100) {
        sum += n;
        n++;
    }
    cout << "sum = " << sum << endl;

    return 0;
}
```

编译:

```
$ g++ -g -o test main.cpp
$ ls
main.cpp test
```

启动gdb调试

```
$ gdb test
GNU gdb (Wind River Linux Sourcery CodeBench 4.8-45) 7.6
Copyright (C) 2013 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-wrs-linux-gnu".
For bug reporting instructions, please see:
<support@windriver.com>...
Reading symbols from /home/ngos/my_learn/test...done.
(gdb)
```

`gdb` 启动时会默认打印一堆免责条款，通过添加 `--silent` 或者 `-q` or `--quiet` 选项，可将这部分信息屏蔽掉。
通过下面的示例运行上述调试命令

```
$ gdb test -q <-- 启动gdb进行调试
Reading symbols from /home/ngos/my_learn/test...done.
(gdb) l       <-- 显示带行号的源代码
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6        int sum = 0;
7        int n = 1;
8        while (n <= 100) {
9            sum += n;
10           n++;
(gdb)          <-- 输入[Enter]查看更多源码
11       }
12       cout << "sum = " << sum;
13
14       return 0;
15   }
(gdb) b 8      <-- 在第8行处设置断点
Breakpoint 1 at 0x4008d6: file main.cpp, line 8.
(gdb) r        <-- 运行程序，直到遇到断点
Starting program: /home/ngos/my_learn/test
warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, main () at main.cpp:8
8               while (n <= 100) {
(gdb) print n <-- 查看代码中变量n的值
$1 = 1        <-- 当前n的值为1，$1 表示该变量，表示变量所在的存储区的名称
(gdb) b 14
Breakpoint 2 at 0x400911: file main.cpp, line 14.
(gdb) n       <-- 单步执行程序
9            sum += n;
(gdb) n       <-- 单步执行程序
10           n++;
(gdb) c       <-- 继续执行程序

Continuing.
sum = 5050

Breakpoint 2, main () at main.cpp:14
14       return 0;
(gdb) print sum <-- 查看sum的值
$2 = 5050       <-- 当前sum的值为5050
(gdb) q         <-- 退出调试
A debugging session is active.

    Inferior 1 [process 3364] will be killed.
```

```
Quit anyway? (y or n) y     <--确实是否退出调试，y为退出，n为不退出
```

接下俩分别介绍各个命令的用法。

GDB断点调试

启动程序

break命令

设置断点

`break` 命令（可以用 `b` 代替）常用的语法格式有以下两种：

- `break location` // `b location`
- `break ... if cond` // `b ... if cond`

第一种格式中，point用于指定断点的具体位置，其表示方法有很多种，如下所示：

- `linenum`
  linenum是一个整数，表示要打断点处代码的行号。程序中各行代码都有对应的行号，可以通过执行 `l` （小写的 `L` ）命令看到。
- `filename:linenum`
  file表示源程序文件名；linenum为整数，表示具体行数。
  整体的意思是在指定文件filename的第linenum行处打断点。
- `+offset`
- `-offset`
  offset为整数（假设值为2），
  +offset表示以当前程序处暂停位置（例如第4行）为准，向后数offset行处（第6行）打断点；
  -offset表示以当前程序暂停位置为准，向前数offset行处（第2行）打断点。
- `function`
  function表示程序中包含的函数的函数名，即break命令会在该函数的开头位置打断点，程序会执行到该程序的第一行处暂停
- `filename:function`
  filename表示文件名，function表示程序中的函数名。
  整体的意思是在指定文件filename中function函数的开头位置打断点。

第二种格式中， `...` 可以是上述的所有参数的值，用于指定打断点的具体位置； `cond` 为某个表达式。
整体的含义是：

> 每次程序执行到 `...` 位置时都计算 `cond` 的值，如果值为 `TRUE` ,则程序在该处暂停，反之，程序继续执行。

断点小技巧

1.查看断点信息

```
info breakpoint [list...]
```

或者

```
info break [list...]
```

其中 `list` （可以查看多个断点，形如 `info breakpoint list1 list2 ...` ）表示查看指定序号的断点信息，若省略则表示查看所有的断点信息。

```
(gdb) info breakpoint  <-- 查看所有的断点信息
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400925 in main() at main.cpp:17
    breakpoint already hit 1 time
2       breakpoint     keep y   0x000000000040092d in main() at main.cpp:19
    breakpoint already hit 24 times
3       breakpoint     keep y   0x00000000004008c8 in call_func() at main.cpp:6
(gdb) info breakpoint 1  <-- 查看序号为1的断点信息
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400925 in main() at main.cpp:17
    breakpoint already hit 1 time
```

说明

> - Num
>
> 断点编号。 `m.n` ， m表示多进程调试时的进程编号，n则是对应进程的断点。
> - Type
>
> 断点类型：行级别调试断点、监测数据变化的监测断点，捕获事件的断点。
> - Disp
> - Enb
>
> 描述信息：当前断点是触发后禁用或者是触发后删除。
> - Address
>
> 断点位置，可能是程序虚拟地址，位置信息等，可能是未来断点pending，一般是共享库上的断点。multiple表示一个断点多个位置。
> - What
>
> 断点信息，即设置时断点的条件，比如作用的行号、已经命中的次数等。

删除断点

- clear命令

  语法： `clear location`

参数location通常为某一行代码的行号或者某个具体的函数名。当location参数为某个函数的函数名时，表示删除位于该函数入口处的所有断点。

- delete命令
  语法： `delete [breakpoint] [list...]`
  其中breakpoint参数可有可无，其中list（可以删除多个断点，形
  如 `delete [breakpoint] list1 list2 ...` ）表示删除指定序号的断点，若省略则表示删除所有的断点。

禁用断点

所谓禁用，就是使目标断点暂时失去作用，必要时可以再次将其激活，恢复断点原有的功能。
禁用断点可以使用disable命令，语法格式如下：
`disable [breakpoint] [list...]`
其中breakpoint参数可有可无，其中list（可以禁用多个断点，形
如 `disable [breakpoint] list1 list2 ...` ）表示禁用指定序号的断点，若省略则表示禁用所有的断点。
举例说明：

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004008ce in call_func(int, int) at main.cpp:6
2       breakpoint     keep y   0x0000000000400924 in main(int, char**) at main.cpp:15
3       breakpoint     keep y   0x000000000040096d in main(int, char**) at main.cpp:24
(gdb) disable 1 2
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00000000004008ce in call_func(int, int) at main.cpp:6
2       breakpoint     keep n   0x0000000000400924 in main(int, char**) at main.cpp:15
3       breakpoint     keep y   0x000000000040096d in main(int, char**) at main.cpp:24
```

可以看到，对于disable命令禁用的断点， `Disp End` 用参数 `n` 表示其处于禁用状态，用参数 `y` 表示该断点处于激活状态。
对于禁用的断点，可以使用 `enable` 命令激活，该命令的语法格式有多种，分别应有不同的功能：

- `enable [breakpoint] [list...]`
  激活用 `[list...]` 参数指定的多个断点，如果不设定 `[list...]` 表示激活所有禁用的断点。
- `enable [breakpoint] once [list...]`
  临时激活以 `[list...]` 为编号的多个断点，但断点只能使用一次，之后自动回到禁用状态。
- `enable [breakpoint] count [list...]`
  临时激活以 `[list...]` 为编号的多个断点，断点可以使用count次，之后进入禁用状态。
- `enable [breakpoint] delete [list...]`
  激活用 `[list...]` 参数指定的多个断点，但断点只能使用1次，之后被永久删除。

其中breakpoint参数可有可无，其中list（可以激活多个断点）表示激活指定序号的断点，若省略则表示激活所有的断点。

举例说明：

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00000000004008ce in call_func(int, int) at main.cpp:6
2       breakpoint     keep n   0x0000000000400924 in main(int, char**) at main.cpp:15
3       breakpoint     keep y   0x000000000040096d in main(int, char**) at main.cpp:24
(gdb) enable breakpoint 1 2
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004008ce in call_func(int, int) at main.cpp:6
2       breakpoint     keep y   0x0000000000400924 in main(int, char**) at main.cpp:15
3       breakpoint     keep y   0x000000000040096d in main(int, char**) at main.cpp:24
```