

- HaHack
- [Archive](#)
  - [Categories](#)
  - [Tags](#)
  - [wiki](#)
- [Subscribe](#)
    - [RSS](#)
    - [WeChat](#)
    - [Toutiao](#)
  - [About](#)

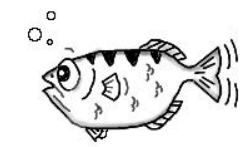
# 调试

- [跟踪调试: GDB](#)
  - [为调试做好准备](#)
  - [在 GDB SHELL 中调试](#)
  - [常用命令](#)
  - [GDB 之歌](#)
  - [高级功能](#)
  - [GDB的可视化前端](#)
  - [DDD](#)
- [内存调试: valgrind](#)
  - [Valgrind 体系结构](#)
  - [使用 Valgrind](#)
- [无调试器调试](#)
  - [core dump](#)
  - [printf 调试](#)
  - [assert 断言](#)
  - [调试宏](#)
- [总结](#)
- [深入阅读](#)
- [Comments](#)

## 跟踪调试: GDB

gdb 是由 GNU 软件系统社区提供的调试器，同 gcc 配套组成了一套完整的开发环境，可移植性很好，支持非常多的体系结构并被移植到各种系统中（包括各种类 Unix 系统与 Windows 系统里的 MinGW 和 Cygwin ）。此外，除了 C 语言之外，gcc/gdb 还支持包括 C++、Objective-C、Ada 和 Pascal 等各种语言后端的编译和调试。gcc/gdb 是 Linux 和许多类 Unix 系统中的标准开发环境，Linux 内核也是专门针对 gcc 进行编码的。

gdb 的吉祥物是专门捕杀 bug 的射手鱼：



For a fish, the archer fish is known to shoot down bugs from low hanging plants by spitting water at them.

### 为调试做好准备

通常，在为调试而编译时，我们会(在尽量不影响程序行为的情况下)关掉编译器的优化选项(-o)， 并打开调试选项(-g)。另外，-Wall 选项打开所有 warning，也可以发现许多问题，避免一些不必要的 bug：

```
1 $ gcc -g -Wall program.c -o program
```

-g选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行，但并不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证gdb能找到源文件。

如果使用Makefile进行编译，可以在执行make时使用CFLAGS=-g选项，将可以为每次编译都添加-g选项。详见[直接在命令行定义Makefile变量](#)

### 在 GDB SHELL 中调试

编译好之后用 gdb 启动要调试的程序：

```
1 $ gdb program
```

之后会出现 gdb 的 shell，输入

```
1 run args
```

即可启动程序，args 是传递给程序的命令行参数。当然，在启动之前，通常会先设置断点，并熟悉一下相关的命令。

### 常用命令

以下列出一些常用的命令(括号里的字符表示缩写，具体语法请参见帮助文档)：

命令	描述
backtrace (或bt)	查看各级函数调用及参数，与where命令等效。
break (或b)	设置断点。break if 可以设置断点在满足某个条件时才激活，如break 9 if num != 0。
delete (或d)	delete 是一系列命令，最常用的且删除断点。delete 1 删除断点 1，delete 0 删除所有断点。

delete (或d)  
disable  
condition  
continue (或c)  
finish  
frame (或f) 帧编号  
info (或i)  
list (或l)  
list 行号  
list 函数名  
next (或n)  
print (或p)  
quit (或q)  
set var  
start  
step (或s)  
quit (或q)  
run (或r)  
display 变量名  
undisplay 跟踪显示号  
watch  
x

delete 是一系列命令，取消用的定期删除断点 delete breakpoints 断点号，个加断点与删除断点有断点。  
和 delete 相似，但只是禁用而不是删除。  
为断点指定条件。例如，cond 1 argc==1 为编号为 1 的断点设置条件 argc==1。  
连续运行直到下个断点  
连续运行到当前函数返回为止，然后停下来等待命令  
选择栈帧  
info 是一系列命令，其中最常用的是列出所有断点的 info breakpoints 和查看当前栈帧局部变量的 info locals  
列出源代码，接着上次的位置往下列，每次列10行  
列出从第几行开始的源代码  
列出某个函数的源代码  
单步执行下一行语句  
打印表达式的值，通过表达式可以修改变量的值或者调用函数  
退出gdb调试环境  
修改变量的值  
开始执行程序，停在main函数第一行语句前面等待命令  
单步执行下一行语句，如果有函数调用则进入到函数中  
退出调试器。  
启动程序，可以给程序传递命令行参数，也支持重定向操作  
跟踪查看某个变量，每次停下来都显示它的值。  
可以取消跟踪显示。  
设置观察点。  
从某个位置开始打印存储单元的内容，全部当成字节来看，而不区分哪个字节属于哪个变量

一个典型的 gdb 调试的过程是启动程序，run 运行，程序出错退出，用 where 查到出错的位置，设置断点，重新运行程序，然后在出错的附件使用 next 和 step 单步跟踪，并查出问题所在。

GDB 之歌

这是一首改编自经典儿歌 [ABC](#) 的歌曲，出自[GNU](#)。不仅有趣，而且还介绍了 run、print、set、quit 几个有用的功能。不过个人觉得D和B两句歌词的后半句应该对换，你觉得呢？;-)

Let’ s start at the very beginning, a very good place to start,  
  
When you’ re learning to sing, its Do, Re, Mi;  
  
When you’ re learning to code, its G, D, B.  
  
(background) G, D, B.  
  
The first three letters just happen to be, G, D, B.  
  
(background) G, D, B.  
  
(Chorus)  
  
G!,  
  
GNU!, it’ s Stallman’ s hope,  
  
D,  
  
a break I set myself.  
  
B,  
  
debug that rotten code,  
  
Run,  
  
a far, far way to go.  
  
Print,  
  
to see what you have done,  
  
Set,  
  
a patch that follows print.  
  
Quit,  
  
and recompile your code - - -  
  
That will bring it back to G, D, B,  
  
(Resume from the Chorus)

高级功能

gdb 有一些很强大的高级功能，特别是在 7.0 里加入了逆向调试和Python 脚本的支持。

逆向调试，顾名思义就是逆向地执行程序，相关指令包括 reverse-continue、reverse-next 和 reverse-step 等。逆向执行实际并不是在“执行”程序，而是把程序“回滚”到之前的状态，考虑最简单的情况，一个变量赋值，要逆向执行，就是要恢复其被赋值之前的值，如果对于整个程序而言的话，那么为了能够回滚，程序每执行一步的所有状态都需要记录下来，即便是通过一些增量式的存储技术，也是相当大的数据量，而且程序的执行速度可能会被进一步拖慢。

逆向调试在某些情况下也许会比较有用，比如一个很难跟踪的 bug，单步执行的时候不小心多按了一下 step，这个时候就希望能够 reverse-step 一下，否则就又要从头跟踪一遍了。另一方面，逆向调试并不是万金油(且不说目前 gdb 的这个技术还处在比较初步的阶段)，设想，如果一条程序语句的功能是发送一封 email，那 reverse-next 一下难道还能把它再撤回来吗？

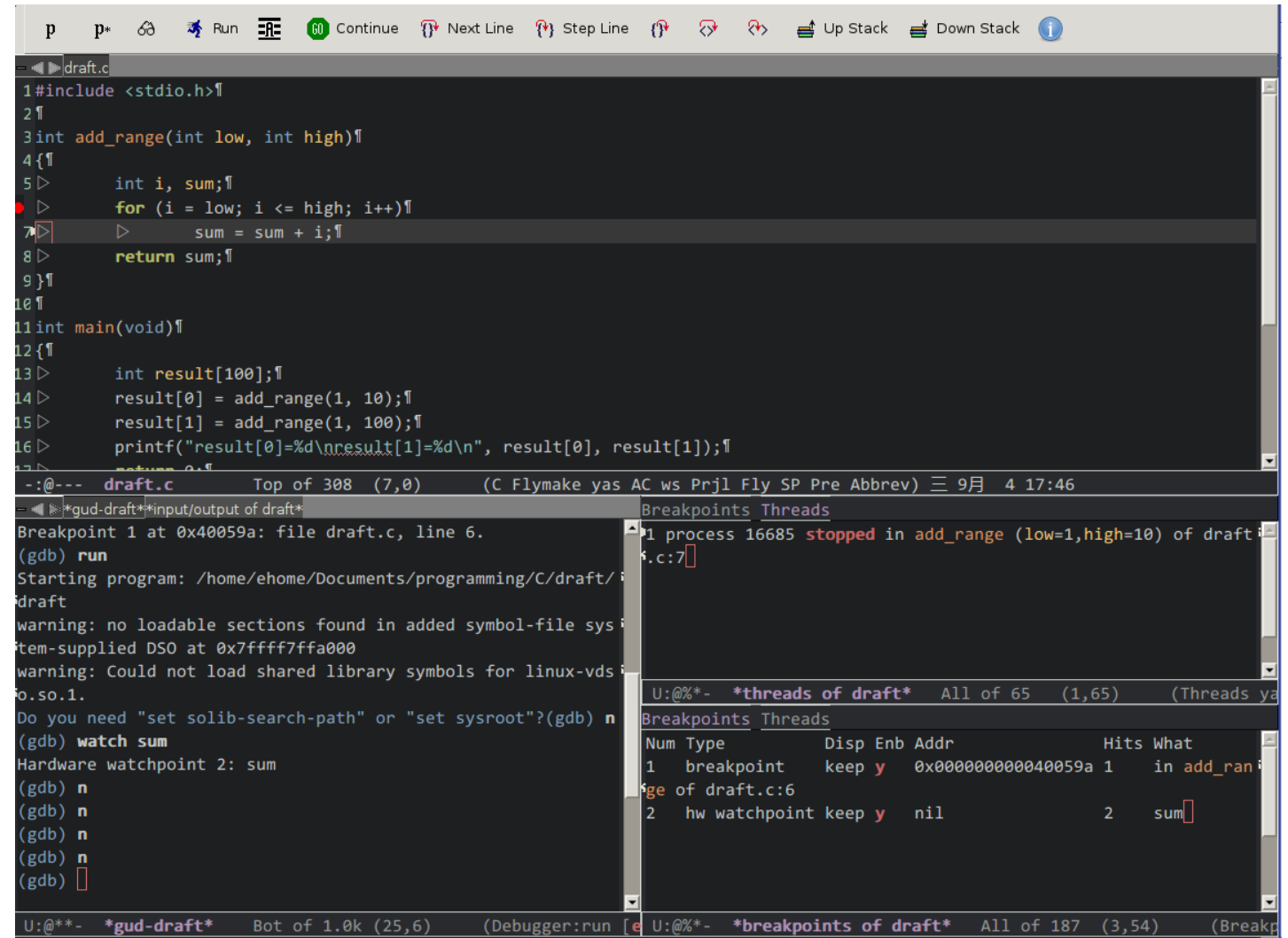
关于 Python 脚本支持，指的是可以在 gdb 里写一些 Python 脚本来控制调试过程，例如，考虑一个复杂的断点条件“某个变量等于特定值并且栈的前一帧的函数名是 foobar”之类的，没法用 C 语言表达式表达出来的，就可以写一个Python 函数来测试断点条件是否满足。

GDB的可视化前端

直接使用 gdb 有一些不太方便的地方，比如查看源代码很麻烦，而且断点的设置也有点痛苦(如果要具体到某一行的话，需要手工输入文件名、行号)。可以使用可视化的工具来减轻这种痛苦。

EMACS 中使用 GDB

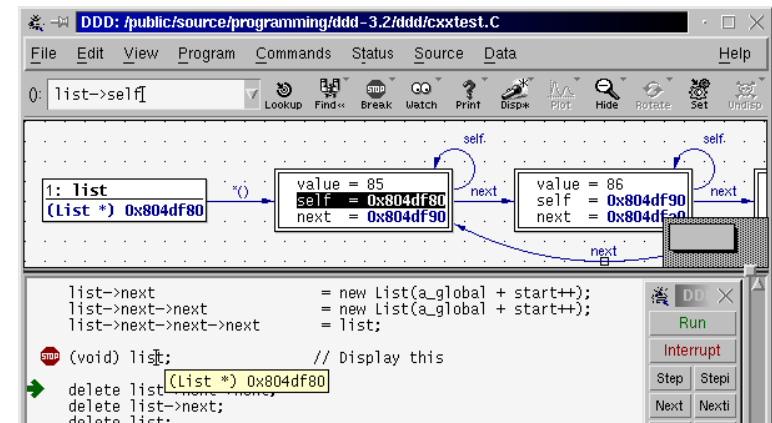
Emacs是一个功能齐全，可扩展性很好的编辑器，它也对 gdb 有比较好的支持，可以当作一个轻量级的 gdb 前端使用。在 Emacs 使用 M-x gdb 即可启动 gdb 调试器，启动之后工具栏会发生变化，并且菜单栏会出现相应的菜单项，与 Visual Studio 类似。一个典型的 Emacs 调试窗口如下图所示：

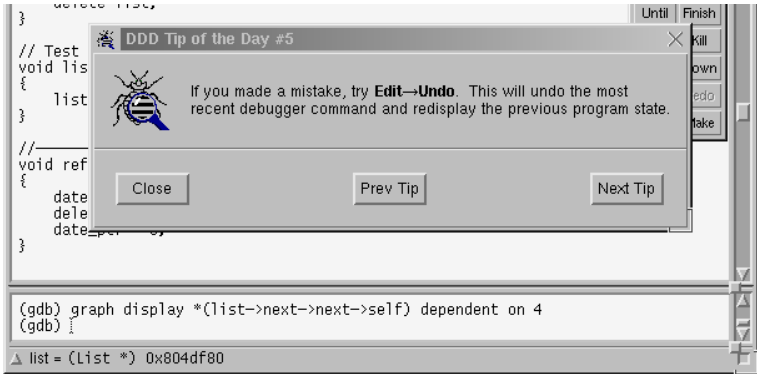


虽然如此，通常调试时还是直接在图中的 gdb shell 中输入 gdb 命令，而不是使用工具栏。比直接使用 gdb shell 更好的地方在于现在可以直接同步地看到带语法高亮的正在执行的源代码(黑色的小箭头)，并且可以像 Visual Studio 那样通过直接在代码左边点击来添加和删除断点(如果是在不支持鼠标操作的终端中，可以使用 C-x C-a C-b 和 C-x C-a C-d 来做同样的操作)。

DDD

[GNU DDD](#) 也是一个可视化的调试器前端，它支持 GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake 以及 Python 的调试器 pydb。相关的用法，可以参考[官方教程](#)。





内存调试：valgrind

C 语言的其中一个强大的地方就在于可以直接对内存进行许多底层的操作，同时这个特性也成为了一个相当繁荣的 bug 家族——内存 bug 滋生的温床。常见的内存 bug 通常包括：

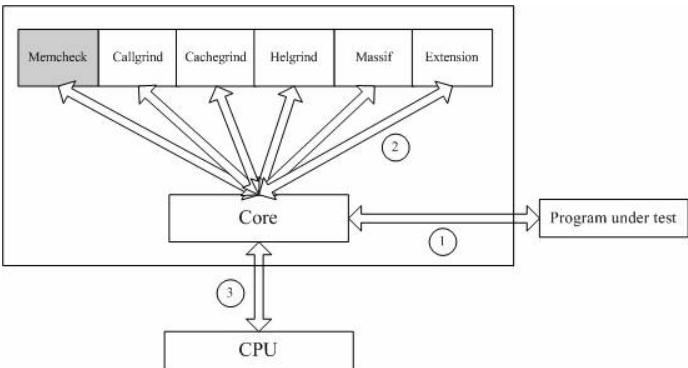
- 内存泄漏:这是需要长期运行的服务程序的最致命的杀手之一。
- 缓冲区溢出:众所周知的安全隐患，非常大一部分系统安全问题(包括入侵、感染等)都是由缓冲区溢出引起的。结果现在 Visual C++编译器对 C 标准库里的诸如 gets、strcpy 等函数一律报 C4996 警告，可见微软对缓冲区溢出的问题也是咬牙切齿啊。
- 野指针，或者是非法内存访问。是另一类非常常见的错误，如果运气好，被操作系统及时捕捉到了异常行为，则情况还比较乐观。在 Linux 下通常会表现为程序出现 segmentation fault(段错误)而退出。
- 内存未初始化，相比它的前面几位兄弟，这个 bug 似乎名气要小一点，但是绝对也是无孔不入，不太出名是因为它通常没有自己独特的表现形式，但是他善于制造混乱，产生各种其他的诡异问题，并且由于未初始化的内容通常是随机的，因此在不同的时间、地点运行有可能会产生不同的结果，不易重现，很难调试。



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 体系结构

Valgrind 的体系结构如下图所示：



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中出现的问题。
- 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。

## 使用 Valgrind

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（需要使用 `-g` 选项编译）。调用Valgrind的通用格式是：

```
1 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。例如：

```
1 $ valgrind --tool=memcheck --leak-check=yes ./program
```

valgrind 会运行程序，并把生成的报告打印到终端上，输出类似于这样：

```
1 ==19389== Invalid write of size 1
2 ==19389== at 0x4C22F0F: strcpy (in ...)
3 ==19389== by 0x400692: copy_string (buggy.c:11)
4 ==19389== by 0x4006B6: main (buggy.c:16)
5 ==19389== Address 0x5179053 is 0 bytes after a block of size 19 alloc'd
6 ==19389== at 0x4C21E03: malloc (in ...)
7 ==19389== by 0x400663: copy_string (buggy.c:6)
8 ==19389== by 0x4006B6: main (buggy.c:16)
```

前面的 19389 是程序的进程标识符(Pid)，在同时跟踪多个程序时可以用于区分，这里可以忽略。输出的内容非常详细，可读性很好，这里的输出大致是说分配了 19 字节的数据，但是却多写了一个字节。同时还给出了出错的源文件以及行号，一般定位之后就很容易查出问题了。

其他的一些输出就不一一列在这里了，一般都能看得懂，另外，valgrind 的一些高级用法以及如何用来做 profiling，可以参考[官方文档](#)。

## 无调试器调试

调试器的出现固然极大地改善了可怜的程序员们的生活水平，然而调试器也并不总是扮演救世主的角色，例如，在有复杂竞争条件的多线程程序或者分布式程序中，调试器所能起的作用通常都不大。另外，调试运行和正常运行的程序实际上是有一定的差异的，有些神奇的 bug，当你以正常方式运行程序时，它跑出来作威作福，可以当你以调试模式运行程序的时候，它就躲得无影无踪了。更为极端的情况是没有调试器可以用，如果 gdb 的开发人员需要用 gdb 来调试 gdb 都还可以接受的话，那么 Linux kernel 的开发人员就真的是悲剧了。

因此，很多时候，我们需要在没有调试器的情况下进行调试，幸运的是，在这样的情况下，也是有一些约定的方法可以遵循的。

### core dump

在 Linux 下，程序如果出现段错误退出，会产生 core dump 文件，默认情况下被 ulimit 禁用了这个功能，运行下面这个命令：

```
1 ulimit -c 5000
```

将允许系统产生 5kB 以内的 core dump 文件，可以根据自己的需求调整大小，并写到 shell 的启动脚本里。系统生成的 core dump 文件通常就是叫做 core，包含了程序出错时的整个状态，用 gdb 加载 core 文件：

```
1 gdb program core
```

就可以进行一些事后分析，例如，可以通过 backtrace 命令查出出错时的调用栈，并查看一些变量的值等，通常对于定位 bug 有很大的帮助。

### printf 调试

printf 调试泛指通过记录程序执行状态来做调试的方法。具体来说，通常我们对于程序的行为和状态都有一些期望的值，通过将程序运行时的实际值打印出来，与期望的情况进行对比，就可以逐渐找到问题的所在。然而这种方法操作起来却有一些相当繁琐的地方：

首先，由于不知道问题出在哪里，又不能在所有的地方都添加输出语句(输出信息太多的话，要找到问题就变得困难了)，所以通常会在可疑的地方添加输出语句，！如果结果发现猜错了，就需要换一个地方或者扩大范围，修改代码，重新编译，运行，再查看新的输出结果。对于编译时间很长(例如，有很多模版代码的 C++程序)的情况，整个过程会变得相当痛苦，因为可能需要重复很多次，并且许多时间都是在做无聊的等待。其次，如果找到了问题所在，是不是要删除那些状态输出语句呢？过多的输出是会影响程序运行性能的，特别是打印到终端上。这些输出语句可能遍布代码的各个角落，要全部清除也不容易，而且，万一以后遇到了类似的 bug 呢？可能还要再写一遍这些类似的输出语句。另一个选择就是把他们注释掉。但是，无论如何，代码会被改得越来越乱。

避免让代码变乱的解决方案是使用标准化的工具，例如，最简单的情况，可以使用下面这样的宏：

```
1 #ifdef DEBUG
2 #define LOG(args) printf args
3 #else
4 #define LOG(args) ((void) 0)
5 #endif /* DEBUG */
```

需要记录信息的时候，使用

```
1 LOG(("a = %d, b = %d\n", a, b));
```

就可以了(注意双重括号是必要的)，需要调试的时候，只要定义 DEBUG，就可以得到调试输出，而调试结束之后可以直接去掉 DEBUG 的定义，这样 LOG 宏在编译的时候就会变成空语句，也就不会产生任何输出了。即使想要移除这些调试语句，由于它们都有统一的格式，因此也可以方便地进行自动化处理。

对于更为复杂的项目，可以使用一些第三方的成熟的日志库来满足更复杂的需求，实现更灵活的控制。

总的来说，printf 调试主要用在两种情况下：

- 对于简单的情况，编程者知道哪里出了问题。

- 过于复杂的情况：调试器已经无能为力了，例如一些分布式的程序。

## assert 断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况，为程序增加诊断功能。

```
1 void assert(int expression)
```

当assert(expression)执行时，如果表达式的值为0，那么 assert 宏将在标准出错输出流 stderr 输出一条如下所示的信息：

```
1 Assertion failed: 表达式, file 文件名, line nnn
```

然后调用 abort 终止执行。其中的源文件名和行号来自于预处理程序宏 \_\_FILE\_\_ 和 \_\_LINE\_\_。

如果在头文件 assert.h 被包含时定义了宏 NDEBUG，那么宏 assert 将被忽略。

下例是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
1 void *memcpy(void *pvTo, const void *pvFrom, size_t size)
2 {
3     assert((pvTo != NULL) && (pvFrom != NULL)); // 使用断言
4     byte *pbTo = (byte *) pvTo;           // 防止改变 pvTo 的地址
5     byte *pbFrom = (byte *) pvFrom;        // 防止改变 pvFrom 的地址
6     while(size -- > 0 )
7         *pbTo ++ = *pbFrom ++ ;
8     return pvTo;
9 }
```

在程序里许多地方插入断言也没有关系，断言在正常的时候并不会产生输出，而且在去掉调试选项之后，断言会编译为空语句，不会影响最终程序的性能。另外，断言通常是对程序状态的一个客观描述，还可以起到注释的作用。因此在代码中保留合适的断言是比较推荐的做法。

断言出错之后立即退出，而 printf 则需要事后再去分析和寻找问题。然而太过于暴力也算是断言的一个缺点，因为 bug 有大小疾缓，有时候让程序能持续稳定地运行也是很重要的，因此除非特别严重的时候，人们通常会倾向于使用更加温和的记录日志的方式来记录下潜在的 bug，而不是直接结束程序。

使用断言的规则：

- 使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。
- 在函数的入口处，使用断言检查参数的有效性(合法性)。
- 在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。
- 一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果“不可能发生”的事情的确发生了，则使用断言进行报警。

func 变量

在打印调试信息时除了文件名和行号之外还可以打印出当前函数名，C99引入一个特殊的标识符\_\_func\_\_支持这一功能。这个标识符应该是一个变量名而不是宏定义，不属于预处理的范畴，但它的作用和\_\_FILE\_\_、\_\_LINE\_\_类似，所以放在一起讲。例如：

```
1
2 #include <stdio.h>
3 void myfunc(void)
4 {
5     printf("%s\n", __func__);
6 }
7 int main(void)
8 {
9     myfunc();
10    printf("%s\n", __func__);
11    return 0;
12 }
13
```

输出：

```
1 $ gcc main.c
2 $ ./a.out
3 myfunc
4 main
```

## 调试宏

Mongrel 的作者 Zed A. Shaw 编写了[一个更为实用的调试宏](#)，内容只有如下短短几行：

```
1
2
3
4
5
6 #ifndef __dbg_h__
7 #define __dbg_h__
8 #include <stdio.h>
9 #include <errno.h>
10 #include <string.h>
11 #ifdef NDEBUG
12 #define debug(M, ...)
13 #else
14 #define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: " M "\n", __FILE__, __LINE__, ##_VA_ARGS_)
15 #endif
16 #define clean_errno() (errno == 0 ? "None" : strerror(errno))
```



```
17 #define clean_errno( errno -- v : none : stderr(errno))
18 #define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##_VA_ARGS__ )
19 #define log_warn(M, ...) fprintf(stderr, "[WARN] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##_VA_ARGS__ )
20 #define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n", __FILE__, __LINE__, ##_VA_ARGS__ )
21 #define check(A, M, ...) if(!(A)) { log_err(M, ##_VA_ARGS__); errno=0; goto error; }
22 #define sentinel(M, ...) { log_err(M, ##_VA_ARGS__); errno=0; goto error; }
23 #define check_mem(A) check((A), "Out of memory.")
24 #define check_debug(A, M, ...) if(!(A)) { debug(M, ##_VA_ARGS__); errno=0; goto error; }
25 #endif
26
27
28
29
30
```

将它保存为 `dbg.h` 就可以在需要调试的地方引入该文件然后调用预定义的几个调试函数：

- `debug`: 当没有预定义 `NDEBUG` 宏时，调用形如 `debug("format", arg1, arg2)` 的语句将可以像 `fprintf` 一样输出内容到 `stderr`。如果预定义了 `NDEBUG`，则调用 `debug` 函数将不会产生任何输出；
- `clean_errno`: 获得一个更安全且可读的 `errno` 版本。通常作为其他几个调试函数的参数；
- `log_err`、`log_warn`、`log_info`: 产生日志输出。和 `debug` 函数类似，但是不能通过设置 `NDEBUG` 来跳过执行；
- `check`: 非常有用的宏，可以检查条件 `A` 是否成立。如果不成立，将错误 `M` 输出到日志（利用 `log_err` 宏），并跳转到函数的错误处理部分（使用 `error`: 标号标记的语句段）。
- `sentinel`: 另一个实用的宏。用于放到一个不该被执行的函数里面。如果该函数被执行，则会打印一个错误信息，并跳转到函数的错误处理部分 `error`。常用的用法是将它放到 `if` 语句或 `switch` 语句中不该执行的边界条件里，例如 `default`: 语句段中；
- `check_mem`: 确保一个指针是有效的指针（不是空指针），如果该指针为空，则提示“Out of memory.”错误信息；
- `check_debug`: 和 `check` 类似，但是底层执行的是 `debug` 宏而非 `log_err` 宏，因此可以通过设置 `NDEBUG` 来禁用这些输出，但仍然会进行错误检查和处理。

## 总结

1. 在绝大多数的情况下，使用[调试宏](#)来诊断和修复跟逻辑语句相关的错误。
2. 使用 [Valgrind](#) 来捕捉所有跟内存相关的错误；
3. 对于上面两个工具无法解决的诡异问题，或者在某些紧急的场合被迫尽可能多的获取错误相关信息的时候，才使用 [gdb](#)。

## 深入阅读

1. [\[PDF\]Pluskid: C语言调试技巧](#)
2. [GDB: The GNU Project Debugger](#)
3. [The LLDB Debugger](#)
4. [Valgrind Documentation](#)
5. [应用 Valgrind 发现 Linux 程序的内存问题](#)
6. [Zed's Awesome Debug Macros](#)
7. [Debugging Code](#)
8. [使用 Cachegrind 和 Callgrind 进行性能调优](#)

- •

•
- [Prev](#)  
[Wiki](#)  
[Next](#)

## Comments

2 Comments


hahack

1 Login


Recommend

Share

按从旧到新排序




Join the discussion...

 feifei435 • 1年前

拜读你的文章后很有收获。谢谢。

^ | v

• Reply • Share

 jlinux • 1年前

太棒了，大神

^ | v

• Reply • Share

 Subscribe

 在您的网站上使用Disqus

 Add

 Add

 隐私

© 2016 Joseph Pan with help from [Hexo](#) and . 