常用的 gdb 命令

backtrace　显示程序中的当前位置和表示如何到达当前位置的栈跟踪（同义词：where）
breakpoint　在程序中设置一个断点
cd　改变当前工作目录
clear　删除刚才停止处的断点
commands　命中断点时，列出将要执行的命令
continue　从断点开始继续执行
delete　删除一个断点或监测点；也可与其他命令一起使用
display　程序停止时显示变量和表达时
down　下移栈帧，使得另一个函数成为当前函数
frame　选择下一条 continue 命令的帧
info　显示与该程序有关的各种信息
jump　在源程序中的另一点开始运行
kill　异常终止在 gdb　控制下运行的程序
list　列出相应于正在执行的程序的原文件内容
next　执行下一个源程序行，从而执行其整体中的一个函数
print　显示变量或表达式的值
pwd　显示当前工作目录
pype　显示一个数据结构（如一个结构或 C++类）的内容
quit　退出 gdb
reverse-search　在源文件中反向搜索正规表达式
run　执行该程序
search　在源文件中搜索正规表达式
set　variable　给变量赋值
signal　将一个信号发送到正在运行的进程
step　执行下一个源程序行，必要时进入下一个函数
undisplay　display 命令的反命令，不要显示表达式
until　结束当前循环
up　上移栈帧，使另一函数成为当前函数
watch　在程序中设置一个监测点（即数据断点）
whatis　显示变量或函数类型

GDB 命令分类详解

一：列文件清单 2
二：执行程序 2
三：显示数据 2
四：断点(breakpoint) 3
五．断点的管理 3
六．变量的检查和赋值 4
七．单步执行 4
八．函数的调用 4
九．机器语言工具 4
十．信号 4
十一.原文件的搜索 5
十二. UNIX 接口 5
十三. 命令的历史 5
十四. GDB 帮助 5
十五. GDB 多线程 6
十六. GDB 使用范例 7

一：列文件清单

1. List

(gdb) list line1,line2

二：执行程序

要想运行准备调试的程序，可使用 run 命令，在它后面可以跟随发给该程序的任何参数，包括标准输入和标准输出说明符(<和>)和外壳通配符 （*、？、[、]）在内。

如果你使用不带参数的 run 命令，gdb 就再次使用你给予前一条 run 命令的参数，这是很有用的。

利用 set args 命令就可以修改发送给程序的参数，而使用 show args 命令就可以查看其缺省参数的列表。

（gdb）set args –b –x

(gdb) show args

backtrace 命令为堆栈提供向后跟踪功能。

Backtrace 命令产生一张列表，包含着从最近的过程开始的所以有效过程和调用这些过程的参数。

三：显示数据

利用 **print** 命令可以检查各个变量的值。

(gdb) print p (p 为变量名)

whatis 命令可以显示某个变量的类型

(gdb) whatis p

type = int *

print 是 gdb 的一个功能很强的命令，利用它可以显示被调试的语言中任何有效的表达式。表达式除了包含你程序中的变量外，还可以包含以下内容：

- 对程序中函数的调用

  (gdb) print find_entry(1,0)

- 数据结构和其他复杂对象

  (gdb) print *table_start

  $8={e=reference='\000',location=0x0,next=0x0}

- 值的历史成分

  (gdb)print $1 ($1 为历史记录变量,在以后可以直接引用 $1 的值)

- 人为数组

人为数组提供了一种去显示存储器块（数组节或动态分配的存储区）内容的方法。早期的调试程序没有很好的方法将任意的指针换成一个数组。就像对待参数一样， 让我们查看内存中在变量 h 后面的 10 个整数，一个动态数组的语法如下所示：

base@length

因此，要想显示在 h 后面的 10 个元素，可以使用 h@10：

(gdb)print h@10

$13=(-1,345,23,-234,0,0,0,98,345,10)

四：断点(breakpoint)

break 命令（可以简写为 b）可以用来在调试的程序中设置断点，该命令有如下四种形式：

- break line-number 使程序恰好在执行给定行之前停止。
- break function-name 使程序恰好在进入指定的函数之前停止。
- break line-or-function if conditio 如果 condition（条件）是真，程序到达指定行或函数时停止。
- break routine-name 在指定例程的入口处设置断点

如果该程序是由很多原文件构成的，你可以在各个原文件中设置断点，而不是在当前的原文件中设置断点，其方法如下：

(gdb) break filename:line-number

(gdb) break filename:function-name

要想设置一个条件断点，可以利用 break if 命令，如下所示：

(gdb) break line-or-function if expr

例：

(gdb) break 46 if testsize==100

从断点继续运行：**countinue** 命令

五．断点的管理

1．显示当前 gdb 的断点信息：

(gdb) info break

他会以如下的形式显示所有的断点信息：

Num Type Disp Enb Address What

1 breakpoint keep y 0x000028bc in init_random at qsort2.c:155

2 breakpoint keep y 0x0000291c in init_organ at qsort2.c:168

(gdb)

2. 删除指定的某个断点：

(gdb) delete breakpoint 1

该命令将会删除编号为 1 的断点，如果不带编号参数，将删除所有的断点

(gdb) delete breakpoint

3.禁止使用某个断点

(gdb) disable breakpoint 1

该命令将禁止断点 1,同时断点信息的 (Enb)域将变为 n

4．允许使用某个断点

(gdb) enable breakpoint 1

该命令将允许断点 1,同时断点信息的 (Enb)域将变为 y

5．清除原文件中某一代码行上的所有断点

(gdb)clean number

注：number 为原文件的某个代码行的行号

六．变量的检查和赋值

- whatis: 识别数组或变量的类型
- ptype: 比 whatis 的功能更强，他可以提供一个结构的定义
- set variable: 将值赋予变量
- print: 除了显示一个变量的值外，还可以用来赋值

七．单步执行

next 不进入的单步执行

step 进入的单步执行

finish 如果已经进入了某函数，而想退出该函数返回到它的调用函数中，可使用命令 **finish**

八．函数的调用

- **call** name 调用和执行一个函数

  (gdb) call gen_and_sork(1234,1,0)

  (gdb) call printf(“abcd”)

  $1=4

- **finish** 结束执行当前函数，显示其返回值（如果有的话）

九．机器语言工具

有一组专用的 gdb 变量可以用来检查和修改计算机的通用寄存器，gdb 提供了目前每一台计算机中实际使用的 4 个寄存器的标准名字：

$pc 程序计数器

$fp 帧指针（当前堆栈帧）

$sp 栈指针

$ps 处理器状态

十．信号

gdb 通常可以捕捉到发送给它的大多数信号，通过捕捉信号，它就可决定对于正在运行的进程要做些什么工作。例如，按 CTRL-C 将中断信号发送给 gdb， 通常就会终止 gdb。但是你或许不想中断 gdb，真正的目的是要中断 gdb 正在运行的程序，因此，gdb 要抓住该信号并停止它正在运行的程序，这样就可以 执行某些调试操作。

Handle 命令可控制信号的处理，他有两个参数，一个是信号名，另一个是接受到信号时该作什么。

几种可能的参数是：

nostop 接收到信号时，不要将它发送给程序，也不要停止程序。

stop 接受到信号时停止程序的执行，从而允许程序调试；显示一条表示已接受到信号的消息（禁止使用消息除外）

print 接受到信号时显示一条消息

noprint 接受到信号时不要显示消息（而且隐含着不停止程序运行）

pass 将信号发送给程序，从而允许你的程序去处理它、停止运行或采取别的动作。

nopass 停止程序运行，但不要将信号发送给程序。

例如，假定你截获 SIGPIPE 信号，以防止正在调试的程序接受到该信号，而且只要该信号一到达，就要求该程序停止，并通知你。要完成这一任务，可利用如 下命令：

(gdb) handle SIGPIPE stop print

请注意，UNIX 的信号名总是采用大写字母！你可以用信号编号替代信号名

如果你的程序要执行任何信号处理操作，就需要能够测试其信号处理程序，为此，就需要一种能将信号发送给程序的简便方法，这就是 signal 命令的任务。该命令的参数是一个数字或者一个名字，如 SIGINT。假定你的程序已将一个专用的 SIGINT（键盘输入，或 CTRL-C；信号 2）信号处理程序设置成采 取某个清理动作，要想测试该信号处理程序，你可以设置一个断点并使用如下命令：

（gdb） signal 2

continuing with signal SIGINT(2)

该程序继续执行，但是立即传输该信号，而且处理程序开始运行.

十一. 原文件的搜索

Search text: 该命令可显示在当前文件中包含 text 串的下一行。

Reverse-search text: 该命令可以显示包含 text 的前一行。

十二. UNIX 接口

shell 命令可启动 UNIX 外壳，CTRL-D 退出外壳，返回到 gdb.

十三. 命令的历史

为了允许使用历史命令，可使用 **set history expansion on** 命令

(gdb) set history expansion on

十四. GDB 帮助

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有：

* aliases：命令别名
* breakpoints：断点定义；
* data：数据查看；
* files：指定并查看文件；
* internals：维护命令；
* running：程序执行；
* stack：调用栈查看；
* statu：状态查看；
* tracepoints：跟踪程序执行。

键入 help 后跟命令的分类名，可获得该类命令的详细清单。

十五. GDB 多线程

先介绍一下 GDB 多线程调试的基本命 令。

- **info threads**

显示当前可调试的所有线程，每个线程会有一个 GDB 为其分配的 ID，后面操作线程的时候会用到这个 ID。前面有*的是当前调试的线程。

- **thread ID**

切换当前调试的线程为指定 ID 的线程。

- **break thread_test.c:123 thread all**

在所有线程中相应的行上设置断点

- **thread apply ID1 ID2 command**

让一个或者多个线程执行 GDB 命令 command。

- **thread apply all command**

让所有被调试线程执行 GDB 命令 command。

- **set scheduler-locking off|on|step**

估计是实际使用过多线程调试的人都可以发现，在使用 step 或者 continue 命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试 程序执行呢？通过这个命令就可以实现这个需求。

off 不锁定任何线程，也就是所有线程都执行，这是默认值。

on 只有当前被调试程序会执行。

step 在单步的时候，除了 next 过一个函数的情况(熟悉情况的人可能知道，这其实是一个设置断点然后 continue 的行为)以外，只有当前线程会执行。

在介绍完基本的多线程调试命令后，大概介绍一下 GDB 多线程调试的实现思路。

比较主要的代码是 thread.c，前面介绍的几个命令等都是在其中实现。

thread_list 这个表存储了当前可调试的所有线程的信息。

函数 add_thread_silent 或者 add_thread(不同版本 GDB 不同)用来向 thread_list 列表增加一个线程的信息。

函数 delete_thread 用来向 thread_list 列表删除一个线程的信息。

上面提到的这 2 个函数会被有线程支持的 target 调用，用来增加和删除线程，不同的 OS 对线程的实现差异很大，这么实现比较好的保证了 GDB 多线程调试 支持的扩展性。

函数 info_threads_command 是被命令 info threads 调用的，就是显示 thread_list 列表的信息。

函数 thread_command 是被命令 thread 调用，切换当前线程最终调用的函数是 switch_to_thread，这个函数会先将当前调试线 程变量 inferior_ptid，然后对寄存器和 frame 缓冲进行刷新。

函数 thread_apply_command 被命令 thread apply 调用，这个函数的实际实现其实很简单，就是先切换当前线为指定线程，然后调用函数 execute_command 调用指定函数。

比较特别的是 set scheduler-locking 没有实现在 thread.c 中，而是实现在控制被调试程序执行的文件 infrun.c 中。

对其的设置会保存到变量 scheduler_mode 中，而实际使用这个变量的函数只有用来令被调试程序执行的函数 resume。在默认情况下， 传递给 target_resume 的变量是 resume_ptid，默认情况下其的值为 RESUME_ALL，也就是告诉 target 程序执行的时候所有 被调试线程都要被执行。而当 scheduler_mode 设置为只让当前线程执行的时候，resume_ptid 将被设置为 inferior_ptid， 这就告诉 target 只有 inferior_ptid 的线程会被执行。

最后特别介绍一下 Linux 下多线程的支持，基本的调试功能在 linux-nat.c 中，这里有对 Linux 轻量级别进程本地调试的支持。但是其 在调试多线程程序的时候，还需要对 pthread 调试的支持，这个功能实现在 linux-thread-db.c 中。对 pthread 的调试要通过调用 libthread_db 库来支持。

这里有一个单独的 target"multi-thread"，这个 target 有 2 点很特别:

第一，一般 target 的装载是在调用相关 to_open 函数的时候调用 push_target 进行装载。而这个 target 则不同，在其初始化 的时候，就注册了函数 thread_db_new_objfile 到库文件 attach 事件中。这样当 GDB 为调试程序的动态加载库时候 attach 库文 件的时候，就会调用这个函数 thread_db_new_objfile。这样当 GDB 装载 libpthread 库的时候，最终会装载 target"multi-thread"。

第二，这个 target 并没有像大部分 target 那样自己实现了全部调试功能，其配合 linux-nat.c 的代码的功能，这里有一个 target 多层 结构的设计，要介绍的比较多，就不详细介绍了。

最后介绍一下我最近遇见的一个多线程调试和解决。

基本问题是在一个 Linux 环境中，调试多线程程序不正常，info threads 看不到多线程的信息。

我先用命令 maintenance print target-stack 看了一下 target 的装载情况，发现 target"multi-thread"没有被装载，用 GDB 对 GDB 进行调试，发现在 函数 check_for_thread_db 在调用 libthread_db 中的函数 td_ta_new 的时候，返回了 TD_NOLIBTHREAD，所 以没有装载 target"multi-thread"。

在时候我就怀疑是不是 libpthread 有问题，于是检查了一下发现了问题，这个环境中的 libpthread 是被 strip 过的，我想可能 就是以为这个影响了 td_ta_new 对 libpthread 符号信息的获取。当我换了一个没有 strip 过的 libpthread 的时候，问题果然解决 了。

最终我的解决办法是拷贝了一个.debug 版本的 libpthread 到 lib 目录中，问题解决了。 多线程如果 dump，多为段错误，一般都涉及内存非法读写。可以这样处理，使用下面的命令打开系统开关，让其可以在死掉的时候生成 core 文件。

ulimit -c unlimited

这样的话死掉的时候就可以在当前目录看到 core.pid(pid 为进程号)的文件。接着使用 gdb:

gdb ./bin ./core.pid

进去后，使用 bt 查看死掉时栈的情况，在使用 frame 命令。

还有就是里面某个线程停住，也没死，这种情况一般就是死锁或者涉及消息接受的超时问题(听人说的，没有遇到过)。遇到这种情况，可以使用：

gcore pid （调试进程的 pid 号）

手动生成 core 文件，在使用 pstack(linux 下好像不好使)查看堆栈的情况。如果都看不出来，就仔细查看代码，看看是不是在 if，return，break，continue 这种语句操作是忘记解锁，还有嵌套锁的问题，都需要分析清楚了。

十六. GDB 使用范例

清单 一个有错误的 C 源程序 bugging.c

代码:

```
----------
1  #include
2
3  static  char  buff  [256];
4  static  char*  string;
5  int  main  ()
6  {
7       printf  ("Please  input  a  string:  ");
8       gets  (string);
9       printf  ("\nYour  string  is:  %s\n",  string);
10  }
```

上面这个程序非常简单，其目的是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 string，因此，编译并运行之后，将出现 Segment Fault 错误：

```
$ gcc -o bugging -g bugging.c
$ ./bugging
Please input a string: asfd
Segmentation fault (core dumped)
```

为了查找该程序中出现的问题，我们利用 gdb，并按如下的步骤进行：

1. 运行 gdb bugging 命令，装入 bugging 可执行文件；
2. 执行装入的 bugging 命令 run；
3. 使用 where 命令查看程序出错的地方；
4. 利用 list 命令查看调用 gets 函数附近的代码；
5. 唯一能够导致 gets 函数出错的因素就是变量 string。用 print 命令查看 string 的值；
6. 在 gdb 中，我们可以直接修改变量的值，只要将 string 取一个合法的指针值就可以了，为此，我们在第 8 行处设置断点 break 8；
7. 程序重新运行到第 8 行处停止，这时，我们可以用 set variable 命令修改 string 的取值；
8. 然后继续运行，将看到正确的程序运行结果