**常用的GDB命令**

backtrace(bt) 显示程序中的当前位置和当前位置的栈跟踪（同where）

breakpoint(b) 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(c) 从断点开始继续执行

delete(d) 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(i) 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(k) 异常终止在gdb 控制下运行的程序

list(l) 列出相应于正在执行的程序的原文件内容

next(n) 执行下一个源程序行，从而执行其整体中的一个函数

print(p) 显示变量或表达式的值

pwd 显示当前工作目录

ptype 显示一个数据结构（如一个结构或C++类）的内容

quit(q) 退出gdb

reverse-search 在源文件中反向搜索正规表达式

run(r) 执行该程序

search 在源文件中搜索正规表达式

set var=x 给变量赋值x

signal 将一个信号发送到正在运行的进程

step(s) 执行下一个源程序行，必要时进入下一个函数

thread(t) 切换到指定线程，t tid

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(w) 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

source 定位源代码路径，用法source src-path

help(h) 显示当前帮助

具体可参考《GDB QUICK REFERENCE》，非常棒！

另外 gcore pid 对正在运行的进程产生一个core文件

一. 列文件清单

l list(l)

(gdb)list line1,line2

二. 执行程序

要想运行准备调试的程序，可使用run(r)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(bt) 命令为堆栈提供向后跟踪功能。

backtrace(bt) 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**

**1. 利用print(p)命令可以检查各个变量的值**

(gdb) print var (var为变量名)

whatis 命令可以显示某个变量的类型

(gdb) whatis var

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。当你使用print命令时，可以用一个参数/F来选择输出的打印格式。F可以是以下的一些值：

'x' 16进制整数格式

'd' 有符号十进制整数格式

'u' 无符号十进制整数格式

'f' 浮点数格式表达式

通常在gdb调试时要打印出一些字符串的内容，通过p str打印字符串时，通常有长度的限制，我测试linux机器上默认为200个，但实际输出的长度str\_len可能大于该值，结果不能够完全输出，而进行了省略，通过命令set print element 0就可以了。

**print除了显示变量外，还可以显示以下内容：**

l **对程序中函数的调用**

(gdb) print find\_entry(1,0)

l **数据结构和其他复杂对象**

(gdb) print \*table\_start

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

l **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)

l **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)

2. 使用info查看信息

（1）查看当前源程序：info source

（2）看堆栈信息：info stack

（3）查看当前的参数：info args

（4）查看当前所有变量：frame n ; info locale;

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：

l break line-number 使程序恰好在执行给定行之前停止。

l break function-name 使程序恰好在进入指定的函数之前停止。

l break line-or-function if condition如果condition（条件）是真，程序到达指定行或函数时停止。

l **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(缩写c)命令

五. 断点的管理

1．Info(i)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

2. delete(d)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

l whatis: 识别数组或变量的类型

l ptype: 比whatis的功能更强，他可以提供一个结构的定义

l set variable: 将值赋予变量

l print: 除了显示一个变量的值外，还可以用来赋值

七. 设置监视点watch（w）

当你调试一个很大的程序，并且在跟踪一个关键的变量时，发现这个变量不知在哪儿被改动过，如何才能找到改动它的地方。这时你可以使用watch命令。简单地说，监视点可以让你监视某个表达式或变量，当它被读或被写时让程序断下。watch命令的用法如下：watch EXPRESSION

watch指令是监视被写的，当你想监视某个表达式或变量被读的话，需要使用rwatch指令，具体用法是一样的。要注意的是，监视点有硬件和软件两种方式，如果可能Linux尽可能用硬件方式，因为硬件方式在速度上要大大快于软件方式。软件方式由于要在每次执行一条指令后都要检查所要监视的值是否被改变，因此它的执行速度会大大降低。同时它也无法设置成被读时让程序断下，因为读操作不会改变值，所以GDB无法检测到读操作。幸运的是，目前的PC机基本都支持硬件方式。如果你想确认一下你的机器是否支持硬件，你可以在调试程序时用watch设置一个监视点，如果GDB向你显示：Hardware watchpoint NUM: EXPR，那么你可以放心了，你的机器支持硬件方式。

八. 检查内存值x

检查内存值的指令是x，x是examine的意思。用法：x /NFU ADDR，其中N代表重复数，F代表输出格式，U代表每个数据单位的大小。U可以取如下值：

b ：字节(byte)

h ：双字节数值

w ：四字节数值

g ：八字节数值

因此，上面的指令可以这样解释：从ADDR地址开始，以F格式显示N个U数值。例如：x/4ub 0x4000，会以无符号十进制整数格式(u)显示四个字节(b)，0x4000，0x4001，0x4002，0x4003。

九. 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

十. 函数的调用

Ø 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

Ø finish 结束执行当前函数，显示其返回值（如果有的话）

十一. 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

十二. 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.

十三. 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。

十四. UNIX接口

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

十五. 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

十六. 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。

**十七. GDB多线程**

先介绍一下GDB多线程调试的基本命令。

**Ø 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。

**Ø thread ID**

切换当前调试的线程为指定ID的线程。

**Ø break thread\_test.c:123 thread all**

在所有线程中相应的行上设置断点

**Ø thread apply ID1 ID2 command**

让一个或者多个线程执行GDB命令command。

**Ø thread apply all command**

让所有被调试线程执行GDB命令command。

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

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。

Ø thread\_list这个表存储了当前可调试的所有线程的信息。

Ø 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。

Ø 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。

Ø 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。

Ø 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。

Ø 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"。

在这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，再使用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。

十八. 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。

十九. 附录一：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 命 令    缩写    用 法    作 用  help    h    h command    显示命令的帮助  run    r    r [args]    运行要调试的程序 args为要运行程序的参数  step    s    s [n]    步进,n为步进次数。如果调用了某个函数，会跳入函数内部。  next    n    n [n]    下一步,n为下一步的次数  continue    c    c    继续执行程序  list    l    l / l+ / l-    列出源码  break    b    b address    在地址address上设置断点  b function    此命令用来在某个函数上设置断点。  b linenum    在行号为linenum的行上设置断点。程序在运行到此行之前停止  b +offset  b -offset    在当前程序运行到的前几行或后几行设置断点。offset为行号  b filename:linenum    在文件名为filename的原文件的第linenum行设置断点  b filename:function    在文件名为filename的原文件的名为function的函数上设置断点。当你的多个文件中可能含有相同的函数名时必须给出文件名  watch    w    w exp    监视表达式的值  kill    k    k    结束当前调试的程序  print    p    p exp    打印表达式的值      p/fmt exp    fmt为以下值  x 十六进制 d 十进制  u 无符号数 o八进制  t 二进制 a 十六进制打印  c 字符格式 f浮点数  output    o    o exp    同print,但是不输出下一行的语句  ptype      ptype struct    输出一个struct结构的定义  whatis      whatis var    命令可以显示某个变量的类型  x    x/(length)(format)(size) addr  x/6(o/d/x/u/c/t)(b/h/w)  例如：x/4xw    按一定格式显示内存地址或变量的值  pwd      pwd    显示当前路径  delete    d    d num    删除编号为num的断点和监视  disable      disable n    编号为n的断点暂时无效  enable      enable n    与disable 相反  display      display expr    暂停，步进时自动显示表达式的值  finish        执行直到函数返回执行直到当前stack返回  return        强制从当前函数返回  where        命令用来查看执行的代码在什么地方中止  backtrace    bt      显示函数调用得所有栈框架（stack frames）的 踪迹和当前函数的参数的值。  quit    q      退出调试程序  frame        shell      shell ls    执行shell命令  make        不退出gdb而重新编译生成可执行文件  disassemble        显示反汇编代码  thread      thread thread\_no    用来在线程之间的切换  set      set width 70    就是把标准屏幕设为70列    set var=54    设置变量的值。  forward/search    search string    从当前行向后查找匹配某个字符串的程序行  reverse-search      forward/search相反，向前查找字符串。使用格式同上  up/down        上移/下移栈帧，使另一函数成为当前函数  info    i    i breakpoint    显示当前断点列表    i reg[ister]    显示寄存器信息    i threads    显示线程信息    i func    显示所有的函数名  i local    显示当前函数的所有局部变量的信息  i prog    显示调试程序的执行状态  i watch exp    为表达式（变量）exp设置一个观察点。一但表达式值有变化时，马上停住程序。  i proc    显示进程的概要信息  info proc mappings    报告你进程所能访问的地址范围。  info proc times    你进程和子进程的开始时间，用户时间(user CPU time),和系统CPU时间。  info proc id    报告有关进程id的信息  info proc status    报告你进程的一般状态信息。如果进程停止了。这个报告还包括停止的原因和收到的信号  info proc all    显示上面proc命令这些命令返回的所有信息 |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |