Advanced Programming in the UNIX Environment — *A Brief Introduction of* gdb

Hop Lee
hoplee@bupt.edu.cn

Contents

1 gdb 介绍

- 本文内容参考自LinuxSir Forum
- Linux 的大部分特色源自于 shell 的 GNU 调试器，也称作gdb。
- gdb可以让您查看程序的内部结构、打印变量值、设置断点，以及单步调试源代码。它是功能极其强大的工具，适用于修复程序代码中的问题。
- 一般来说，gdb主要帮忙你完成下面四个方面的功能：
  1. 启动你的程序，可以按照你的自定义的要求随心所欲地运行程序。
  2. 可让被调试的程序在你所指定的断点处停住。（断点可以是条件表达式）
  3. 当程序被停住时，可以检查此时程序中所发生的事。
  4. 动态地改变你程序的执行环境。
- 开始之前
  - 良好的习惯提高你的调试效率
  - 使用固定的名词缩写规则
  - 给你的函数、变量起一个好名字
  - 充分利用大小写分割单词
  - 利用缩进做好程序的排版
  - 做好设计后再动手写代码
- 调试之前
  - 一般来说gdb主要调试的是 C/C++ 的程序。
  - 要调试 C/C++ 的程序，首先在编译时必须要把调试信息加到可执行文件中。在gcc（或g++）下使用额外的-g选项来编译程序可以做到这一点。
  - 如果编译时没有-g选项，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。
- 关于 core dump
  - 当程序执行了非法的指令的时候，为了帮助调试，操作系统将相应的内存中的内容导出到一个文件供调试之用。
  - Unix 命令—file可以用于检查core文件是由那个程序生成的
  - 通过检查core文件，结合程序的调试信息，我们可以找到大多数程序中的 bug。

2 启动 gdb

- 启动gdb的方法：
  - 在 shell 中，直接运行gdb
    直接运行gdb可以进入gdb环境，由于没有加载需要调试的程序，需要通过file内部命令来进行加载。
  - 在 shell 中执行命令：

```
1 gdb [path][program]
```

    执行文件的搜索路径是可选的
    通常这种执行方法是为了在调试状态中运行程序以发现问题所在
  - 在 shell 中执行命令：

```
1 gdb [path][program] [path][core]
```

    当程序运行出错，我们需要检查core文件中提供的错误信息的时候使用这种方法调试程序
  - 在 shell 中执行命令：

```
1 gdb [path][program] [process ID]
```

    如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程 ID。gdb会自动 attach 上去，并调试它。

3 gdb 常用选项

- -symbols或-s
  从指定文件中读取符号表。
- -se *file*
  从指定文件中读取符号表信息，并把他用在可执行文件中。
- -core或-c
  调试时 core dump 的core文件。
- -directory或-d
  加入一个源文件的搜索路径。默认搜索路径是环境变量中PATH所定义的路径。

4 gdb 常用内部命令

- gdb的命令很多，gdb把之分成许多个种类。help命令只是例出gdb的命令种类，如果要看种类中的命令，可以使用help *inner-cmd*命令，如：help *breakpoints*，查看设置断点的所有命令。

  aliases 其它内部命令的别名

  breakpoints 使得程序在指定的位置停止

  data 检查数据

  files 指定及检查文件

  internals 内部命令维护

`obscure` 含糊的特性

`running` 运行程序

`stack` 检查栈的内容

`status` 状态调查

`support` 支持工具

`tracepoints` 无须停止程序而对程序的执行过程进行跟踪

`user-defined` 用户自定义的内部命令

- gdb中，输入内部命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该具有唯一性，不会引起混淆。
- 你可以敲击两次TAB键来补齐命令的全称，如果有重复的，那么gdb会把它们都列出来。补全机制对函数名、变量名也有效。
- 下面将介绍常用的一些gdb内部命令。

4.1 退出及 shell

- 要退出gdb时，只用发quit或命令简称q就行了。
- 在gdb环境中，你可以执行 UNIX 的 shell 的命令，使用gdb的shell命令来完成，它将调用 UNIX 的 shell 来执行，环境变量SHELL中定义的 UNIX 的 shell 将会被用来执行，如果SHELL没有定义，那就使用 UNIX 的标准 shell：/bin/sh。（在 M$ Windows 中使用command.com或cmd.exe）
- 可以在gdb中执行make命令来重新编译自己的程序。这个命令等价于shell make。

4.2 暂停/恢复程序运行

- 调试程序中，暂停程序运行是必须的，gdb可以方便地暂停程序的运行。你可以设置程序在哪行停住，在什么条件下停住，在收到什么信号时停往等等。以便于你查看运行时的变量，以及运行时的流程。
- 当进程被gdb停住时，你可以使用info *program*来查看程序的是否在运行、进程号以及被暂停的原因。
- 在gdb中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（WatchPoint）、捕捉点（CatchPoint）、信号（Signals）、线程停止（Thread Stops）。
- 如果要恢复程序运行，可以使用c或是continue命令。

4.2.1 设置断点（BreakPoint）

- 用break命令来设置断点。
- 在进入指定函数时停住：

  break *FUNCTION*

  C++ 中可以使用class::function 或function(type,type) 格式来指定函数名。
- 在指定行号停住：

  break *LINENUM*
- 在当前行号的前面或后面的*offset*行停住。*offset*为自然数。

  break +*offset*

  break -*offset*

- 在源文件*filename*的*linenum*行处停住：

  `break` *FILENAME:LINENUM*

- 在源文件*filename*的*function*函数的入口处停住：

  `break` *FILENAME:FUNCTION*

- 在程序运行的内存地址*ADDRESS*处停住：

  `break` **ADDRESS*

- `break`命令没有参数时，表示在下一条指令处停住：

  `break`

- 在条件成立时停住，*...*可以是上述的参数，*COND*表示条件。比如在循环体中，可以设置`break if` *i*=100，表示当*i*为 100 时停住程序。

  `break` *... if COND*

- 查看断点时，可使用`info`命令，如下所示：（注：*n*表示断点序号）

  `info breakpoints` [*n*]

  `info break` [*n*]

4.2.2 设置观察点（WatchPoint）

- 观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

- 为表达式（变量）*EXPR*设置一个观察点。一量表达式值有变化时，马上停住程序：

  `watch` *EXPR*

- 当表达式（变量）*EXPR*被读时，停住程序：

  `rwatch` *EXPR*

- 当表达式（变量）*EXPR*的值被读或被写时，停住程序：

  `awatch` *EXPR*

- 列出当前所设置了的所有观察点：

  `info watchpoints`

4.2.3 设置捕捉点（CatchPoint）

- 你可设置捕捉点来补捉程序运行时的一些事件。如：载入共享库（动态链接库）或是 C++ 的异常。

- 当*EVENT*发生时，停住程序：

  `catch` *EVENT*

- *EVENT*可以是下面的内容：

  1. `throw`：一个 C++ 抛出的异常
  2. `catch`：一个 C++ 捕捉到的异常
  3. `exec`：调用系统调用`exec`时
  4. `fork`：调用系统调用`fork`时
  5. `vfork`：调用系统调用`vfork`时

6. `load`或`load` *LIBNAME*：载入共享库（动态链接库）时
7. `unload`或`unload` *LIBNAME*：卸载共享库（动态链接库）时

- 上述后五种事件只在 HP-UX 系统下有效。
- 只设置一次捕捉点，当程序停住以后，该点被自动删除：

  `tcatch` *EVENT*
- 命令`info catch`可以列出当前所有的捕捉点。

4.2.4 维护停止点

- 上面说了如何设置程序的停止点，`gdb`中的停止点也就是上述的三类。在`gdb`中，如果你觉得已定义好的停止点没有用了，你可以使用`delete`、`clear`、`disable`、`enable`这几个命令来进行维护。
- 清除所有的已定义的停止点：

  `clear`
- 清除所有设置在函数上的停止点：

  `clear` *FUNCTION*

  `clear` *FILENAME:FUNCTION*
- 清除所有设置在指定行上的停止点：

  `clear` *LINENUM*

  `clear` *FILENAME:LINENUM*
- 删除指定的断点，*BREAKPOINTS*为断点序号。如果不指定断点序号，则表示删除所有的断点。*RANGE*表示断点序号的范围（如：3-7）。其简写命令为`d`：

  `delete` [*BREAKPOINTS*] [*RANGE*...]
- 比删除更好的一种方法是 disable 停止点，disable 了的停止点，`gdb`不会删除，当你还需要时，enable 即可，就好像回收站一样。
- disable 所指定的停止点，*BREAKPOINTS*为停止点序号。如果什么都不指定，表示 disable 所有的停止点。简写命令是`dis`。

  `disable` [*BREAKPOINTS*] [*RANGE*...]
- enable 所指定的停止点，*BREAKPOINTS*为停止点序号。

  `enable` [*BREAKPOINTS*] [*RANGE*...]
- enable 所指定的停止点一次，当程序停止后，该停止点马上被`gdb`自动 disable。

  `enable` [*BREAKPOINTS*] `once` *RANGE*...
- enable 所指定的停止点一次，当程序停止后，该停止点马上被`gdb`自动删除。

  `enable` [*BREAKPOINTS*] `delete` *RANGE*...

4.2.5 停止条件维护

- 前面在说到设置断点时，我们提到过可以设置一个条件，当条件成立时，程序自动停止，这是一个非常强大的功能，这里，我想专门说说这个条件的相关维护命令。
- 一般来说，为断点设置一个条件，我们使用`if`关键词，后面跟其断点条件。并且，条件设置好后，我们可以用`condition`命令来修改断点的条件。（只有`break`和`watch`命令支持`if`，`catch`目前暂不支持`if`）
- 修改断点号为*BNUM*的停止条件为*EXPR*。

  `condition` *BNUM EXPR*
- 清除断点号为*BNUM*的停止条件。

  `condition` *BNUM*
- 还有一个比较特殊的维护命令`ignore`，你可以指定程序运行时，忽略停止条件几次。

  `ignore` *BNUM COUNT*

  表示忽略断点号为*BNUM*的停止条件*COUNT*次。

4.2.6 为停止点设定运行命令

- 我们可以使用`gdb`提供的`command`命令来设置停止点的运行命令。也就是说，当运行的程序在被停止住时，我们可以让其自动运行一些别的命令，这很有利行自动化调试。对基于`gdb`的自动化调试是一个强大的支持。
- 为断点号*BNUM*指写一个命令列表。当程序被该断点停住时，`gdb`会依次运行命令列表中的命令：

  `commands` [*BNUM*]

  ... COMMAND-LIST ...

  `end`
- 例如：

```
1 break foo if x>0
2 commands
3   printf "x is %d\n",x
4   continue
5 end
```

  断点设置在函数`foo`中，断点条件是*x*>0，如果程序被断住后，也就是，一旦*x*的值在`foo`函数中大于 0，`gdb`会自动打印出*x*的值，并继续运行程序。
- 如果你要清除断点上的命令序列，那么只要简单的执行一下`commands`命令，并直接再打个`end`就行了。

4.2.7 断点菜单

- 在 C++ 中，可能会重复出现同一个名字的函数若干次（函数重载），在这种情况下，`break`不能告诉`gdb`要停在哪个函数的入口。当然，你可以使用`break`也就是把函数的参数类型告诉`gdb`，以指定一个函数。否则的话，`gdb`会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了。如：

```
1 (gdb) b String::after
2 [0] cancel
3 [1] all
4 [2] file:Str.cc; line number:867
```

```
[3] file:Str.cc; line number:860
[4] file:Str.cc; line number:875
[5] file:Str.cc; line number:853
[6] file:Str.cc; line number:846
[7] file:Str.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file Str.cc, line 867.
Breakpoint 2 at 0xb344: file Str.cc, line 875.
Breakpoint 3 at 0xafcc: file Str.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

可见，gdb列出了所有after的重载函数，你可以选一下列表编号就行了。0 表示放弃设置断点，1 表示所有函数都设置断点。

4.2.8 恢复程序运行和单步调试

- 当程序被停住了，你可以用continue命令恢复程序的运行直到程序结束，或下一个断点到来。也可以使用step或next命令单步跟踪程序。
- 下面的命令恢复程序运行，直到程序结束，或是下一个断点到来。其中*IGNORE-COUNT*表示忽略其后的断点次数。continue,c, fg三个命令都是一样的意思：

  continue [*IGNORE-COUNT*]

  c [*IGNORE-COUNT*]

  fg [*IGNORE-COUNT*]

- 单步跟踪：

  step [*COUNT*]

  如果有函数调用，它会进入该函数。进入函数的前提是，此函数被编译有 debug 信息。很像 VC 等工具中的**step in**。后面可以加*COUNT*也可以不加，不加表示一条条地执行，加表示执行后面的*COUNT*条指令，然后再停住。

- 下面的命令同样是单步跟踪，如果有函数调用，它不会进入该函数。很像 VC 等工具中的**step over**。后面可以加*COUNT*也可以不加，不加表示一条条地执行，加表示执行后面的*COUNT*条指令，然后再停住：

  next [*COUNT*]

- 下面的命令打开step-mode模式，于是，在进行单步跟踪时，程序不会因为没有 debug 信息而不停住。这个参数有很利于查看机器码：

  set step-mode

  set step-mode *on*

- 关闭step-mode模式。

  set step-mod *off*

- 运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息：

  finish

- 当你厌倦了在一个循环体内单步跟踪时，下面这个命令可以运行程序直到退出循环体：

  until或u

- 单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi和nexti可以单步执行机器指令：

  stepi或si

  nexti或ni

  与之一样有相同功能的命令是display/i$pc，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）

4.2.9 信号（Signals）

- 信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是 UNIX，比较重要应用程序一般都会处理信号。
- gdb有能力在你调试程序的时候处理任何一种信号，你可以告诉gdb需要处理哪一种信号。你可以要求gdb收到你所指定的信号时，马上停住正在运行的程序，以供你进行调试。你可以用gdb的handle命令来完成这一功能。
- 在gdb中定义一个信号处理。信号可以以SIG开头或不以SIG开头，可以用定义一个要处理信号的范围（如：SIGIO-SIGKILL，包括SIGIO, SIGIOT, SIGKILL三个信号），也可以使用关键字all来标明要处理所有的信号。一旦被调试的程序接收到信号，运行程序马上会被gdb停住，以供调试。

  handle *SIGNAL KEYWORDS...*

  关键字*KEYWORDS*可以是以下的一个或多个。

  - 当被调试的程序收到信号时，gdb不会停住程序的运行，但会打出消息告诉你收到这种信号：
    nostop
  - 当被调试的程序收到信号时，gdb会停住你的程序：
    stop
  - 当被调试的程序收到信号时，gdb会显示出一条信息：
    print
  - 当被调试的程序收到信号时，gdb不会告诉你收到信号的信息：
    noprint
  - 当被调试的程序收到信号时，gdb不处理信号。这表示，gdb会把这个信号交给被调试程序会处理：
    pass
    noignore
  - 当被调试的程序收到信号时，gdb不会让被调试程序来处理这个信号：
    nopass
    ignore
- 查看有哪些信号正在被gdb检测：

  info signals

  info handle

4.2.10 线程（Thread Stops）

- 如果你程序是多线程的话，你可以定义你的断点是否在所有的线程上，或是在某个特定的线程。gdb很容易帮你完成这一工作。

- 在下面的语法中，*LINESPEC*指定了断点设置在的源程序的行号。*THREADNO*指定了线程的 ID，注意，这个 ID 是gdb分配的，你可以通过`info threads`令来查看正在运行程序中的线程信息。如果你不指定`thread`则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件：

  `break` *LINESPEC* `thread` *THREADNO*

  `break` *LINESPEC* `thread` *THREADNO* `if` ...

- 例如：

```
1 (gdb) break frik.c:13 thread 28 if bartab > lim
```

- 当你的程序被gdb停住时，所有的运行线程都会被停住。这方便你你查看运行程序的总体情况。而在你恢复程序运行时，所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

4.3 查看栈信息

- 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停住的。当你的程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。你可以用gdb命令来查看当前的栈中的信息。

  下面是一些查看函数调用栈信息的gdb命令：

- 打印当前的函数调用栈的所有信息：

  `backtrace`

  `bt`

- 例如：

```
1 (gdb) bt
2 #0 func (n=250) at tst.c:6
3 #1 0x08048524 in main (argc=1, argv=0xbffff674) at tst.c:30
4 #2 0x400409ed in __libc_start_main () from /lib/libc.so.6
```

  从上面可以看出函数的调用栈信息：`__libc_start_main --> main() --> func()`

- 只打印栈顶上*N*层的栈信息，*N*是一个正整数：

  `backtrace` *N*

  `bt` *N*

- 只打印栈底下*N*层的栈信息，-*N*是一个正整数：

  `backtrace` -*N*

  `bt` -*N*

- 一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈：

  `frame` *N*

  `f` *N*

  *N*是一个从 0 开始的整数，是栈中的层编号。比如：0 表示栈顶，1 表示栈的第二层。

- 向栈的上面移动*N*层，可以不给出*N*，表示向上移动一层。

  `up`

- 向栈的下面移动*N*层，可以不给出*N*，表示向下移动一层。

  down

- 上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：

  select-frame对应于frame命令。

  up-silently对应于up命令。

  down-silently对应于down命令。

- 查看当前栈层的信息，你可以用以下gdb命令：

  frame或f

  会打印出这些信息：栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。

- 下面的命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内部地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等：

  info frame

  info f

- 例如：

```
(gdb) info f
Stack level 0, frame at 0xbffff5d4:
eip = 0x804845d in func (tst.c:6); saved eip 0x8048524
called by frame at 0xbffff60c
source language c.
Arglist at 0xbffff5d4, args: n=250
Locals at 0xbffff5d4, Previous frame's sp is 0x0
Saved registers:
ebp at 0xbffff5d4, eip at 0xbffff5d8
```

- 打印出当前函数的参数名及其值：

  info args

- 打印出当前函数中所有局部变量及其值：

  info locals

- 打印出当前的函数中的异常处理信息：

  info catch

4.4 查看源程序

4.4.1 显示源代码

- gdb可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，gdb会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码。

- 显示程序第 LINENUM 行的周围的源程序。

  list *LINENUM*

- 显示函数名为 FUNCTION 的函数的源程序。

  `list` *FUNCTION*

- 显示当前行后面的源程序。

  `list`

- 显示当前行前面的源程序。

  `list -`

- 一般是打印当前行的上 5 行和下 5 行，如果显示函数是是上 2 行下 8 行，默认是 10 行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

- 设置一次显示源代码的行数为*COUNT*：

  `set listsize` *COUNT*

- 查看当前`listsize`的设置：

  `show listsize`

- 显示从*FIRST*行到*LAST*行之间的源代码：

  `list` *FIRST,LAST*

- 显示从当前行到*LAST*行之间的源代码：

  `list` *,LAST*

- 显示从*FIRST*行到当前行之间的源代码：

  `list` *FIRST,*

- 往后显示源代码：

  `list +`

- 一般来说在`list`后面可以跟以下这们的参数：
  - *NUMBER*行号；
  - <+*offset*> 当前行号的正偏移量；
  - <-*offset*> 当前行号的负偏移量；
  - *FILENAME:NUMBER*哪个文件的哪一行；
  - *FUNCTION*函数名；
  - *FILENAME:FUNCTION*哪个文件中的哪个函数；
  - <**ADDRESS*> 程序运行时的语句在内存中的地址。

4.4.2 搜索源代码

- `gdb`还提供了源代码搜索的命令：
- 前向搜索：

  `forward-search` *REGEXP*

  `search` *REGEXP*

- 反向搜索：

  `reverse-search` *REGEXP*

- 其中，*REGEXP*是正则表达式。

4.4.3 指定源文件的路径

- 某些时候，用-g编译过后的执行程序中只是包括了源文件的名字，没有路径名。gdb提供了可以让你指定源文件的路径的命令，以便gdb进行搜索。
- 下面的命令添加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX 下你可以使用 ":" 作为分隔符，Windows 下你可以使用 ";"：

  directory *DIRNAME*

  dir *DIRNAME*
- 清除所有的自定义的源文件搜索路径信息：

  directory
- 显示定义了的源文件搜索路径：

  show directories

4.4.4 源代码的内存

- 你可以使用info line命令来查看源代码在内存中的地址。其后可以跟 "行号"、"函数名"、"文件名: 行号"、"文件名: 函数名"，这个命令会打印出所指定的源码在运行时的内存地址，如：

```
(gdb) info line tst.c:func
Line 5 of "tst.c" starts at address 0x8048456
and ends at 0x804845d .
```

- 还有一个命令disassemble可以让你查看源程序的当前执行的机器码，这个命令会把目前内存中的指令 dump 出来。如下面的示例表示查看函数func的汇编代码。

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 : push %ebp
0x8048451 : mov %esp,%ebp
0x8048453 : sub $0x18,%esp
0x8048456 : movl $0x0,0xfffffffc(%ebp)
0x804845d : movl $0x1,0xfffffff8(%ebp)
0x8048464 : mov 0xfffffff8(%ebp),%eax
0x8048467 : cmp 0x8(%ebp),%eax
0x804846a : jle 0x8048470
0x804846c : jmp 0x8048480
0x804846e : mov %esi,%esi
0x8048470 : mov 0xfffffff8(%ebp),%eax
0x8048473 : add %eax,0xfffffffc(%ebp)
0x8048476 : incl 0xfffffff8(%ebp)
0x8048479 : jmp 0x8048464
0x804847b : nop
0x804847c : lea 0x0(%esi,1),%esi
0x8048480 : mov 0xfffffffc(%ebp),%edx
0x8048483 : mov %edx,%eax
0x8048485 : jmp 0x8048487
0x8048487 : mov %ebp,%esp
0x8048489 : pop %ebp
0x804848a : ret
End of assembler dump.
```

4.5 查看运行时数据

- 在你调试程序时，当程序被停住时，你可以使用print命令（简写命令为p），或是同义命令inspect来查看当前程序的运行数据。print命令的格式是：

  print [*EXPR*]

  print /F [*EXPR*]

  其中*EXPR*是表达式，是你所调试的程序的语言的表达式（gdb可以调试多种编程语言），/F是输出的格式，比如，如果要把表达式按 16 进制的格式输出，那么就是/x。

4.5.1 表达式

- print和许多gdb的命令一样，可以接受一个表达式，gdb会根据当前的程序运行的数据来计算这个表达式，既然是表达式，那么就可以是当前程序运行中的常量、变量、函数等内容。可惜的是gdb不能使用你在程序中所定义的宏。
- 表达式的语法应该是当前所调试的语言的语法，由于 C/C++ 是一种大众型的语言，所以，本文中的例子都是关于 C/C++ 的。
- 在表达式中，有几种gdb所支持的操作符，它们可以用在任何一种语言中。
  - @ 是一个和数组有关的操作符，在后面会有更详细的说明。
  - :: 指定一个在文件或是一个函数中的变量。
  - {*TYPE*} *ADDR*表示一个指向内存地址 *ADDR*的类型为*TYPE*的一个对象。

4.5.2 程序变量

- 在gdb中，你可以随时查看以下三种变量的值：
  1. 全局变量（所有文件可见的）
  2. 静态全局变量（当前文件可见的）
  3. 局部变量（当前 Scope 可见的）
- 如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量，也就是说，如果一个全局变量和一个函数中的局部变量同名时，如果当前停止点在函数中，用print显示出的变量的值会是函数中的局部变量的值。如果此时你想查看全局变量的值时，你可以使用 “::“操作符：

  file::*variable*

  function::*variable*

  可以通过这种形式指定你所想查看的变量，是哪个文件中的或是哪个函数中的。
- 当然，“::” 操作符会和 C++ 中的发生冲突，gdb能自动识别 “::“ 是否是 C++ 的操作符，所以你不必担心在调试 C++ 程序时会出现异常。
- 另外，需要注意的是，如果你的程序编译时开启了优化选项，那么在用gdb调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。这个是很正常的，因为优化程序会删改你的程序，整理你程序的语句顺序，剔除一些无意义的变量等，所以在gdb调试这种程序时，运行时的指令和你所编写指令就有不一样，也就会出现你所想象不到的结果。对付这种情况时，需要在编译程序时关闭编译优化。一般来说，几乎所有的编译器都支持编译优化的开关，例如，GNU 的 C/C++ 编译器gcc，你可以使用-gstabs选项来解决这个问题。关于编译器的参数，还请查看编译器的使用说明文档。

4.5.3 数组

- 有时候，你需要查看一段连续的内存空间的值。比如数组的一段，或是动态分配的数据的大小。你可以使用gdb的“@”操作符，“@”的左边是第一个内存的地址的值，“@”的右边则你你想查看内存的长度。例如，你的程序中有这样的语句：

```
1 int *array=(int *)malloc(len * sizeof(int));
```

于是，在gdb调试过程中，你可以以如下命令显示出这个动态数组的取值：

p *array@len*

@ 的左边是数组的首地址的值，也就是变量array所指向的内容，右边则是数据的长度，其保存在变量len中，其输出结果，大约是下面这个样子的：

```
1 (gdb) p *array@len
2 $1 = { 2,  4,  6,  8, 10, 12, 14, 16, 18, 20,
3       22, 24, 26, 28, 30, 32, 34, 36, 38, 40}
```

- 如果是静态数组的话，可以直接用print数组名，就可以显示数组中所有数据的内容了。

4.5.4 输出格式

- 一般来说，gdb会根据变量的类型输出变量的值。但你也可以自定义gdb的输出的格式。gdb的数据显示格式：

x	按十六进制格式显示变量
d	按十进制格式显示变量
u	按十六进制格式显示无符号整型
o	按八进制格式显示变量
t	按二进制格式显示变量
a	按十六进制格式显示变量
c	按字符格式显示变量
f	按浮点数格式显示变量

```
1  (gdb) p i
2  $21 = 101
3  (gdb) p/a i
4  $22 = 0x65
5  (gdb) p/c i
6  $23 = 101 'e'
7  (gdb) p/f i
8  $24 = 1.41531145e-43
9  (gdb) p/x i
10 $25 = 0x65
11 (gdb) p/t i
12 $26 = 1100101
```

4.5.5 查看内存

- 你可以使用examine命令（简写是x）来查看内存地址中的值。x命令的语法如下所示：

x/NFU *ADDR*

N、F、U是可选的参数。

- N 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。
- F 表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是s，如果地十是指令地址，那么格式可以是i。
- U 表示从当前地址往后请求的字节数，如果不指定的话，gdb默认是 4 个字节。u参数可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字节，g表示八字节。当我们指定了字节长度后，gdb会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。

*ADDR*表示一个内存地址。

- N/F/U三个参数可以一起使用。例如，命令：x/3uh 0x54320 表示，从内存地址 0x54320 读取内容，h表示以双字节为一个单位，3 表示三个单位，u表示按十六进制显示。

4.5.6 自动显示

- 你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的gdb命令是display：

  display *EXPR*

  display/FMT *EXPR*

  display/FMT *ADDR*

  *EXPR*是一个表达式，FMT表示显示的格式，*ADDR*表示内存地址，当你用display设定好了一个或多个表达式后，只要你的程序被停下来，gdb会自动显示你所设置的这些表达式的值。

- 格式i和s同样被display支持，一个非常有用的命令是：

  display/i $*pc*

  $*pc*是gdb的环境变量，表示着指令的地址，/i则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

- 下面是一些和display相关的gdb命令：

  undisplay *DNUMS ...*

  delete display *DNUMS ...*

  删除自动显示，*DNUMS*意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

- disable和enalbe不删除自动显示的设置，而只是让其失效和恢复：

  disable display *DNUMS ...*

  enable display *DNUMS ...*

- 查看disable设置的自动显示的信息。gdb会打出一张表格，向你报告当然调试中设置了多少个自动显示设置，其中包括，设置的编号，表达式，是否 enable：

  info display

4.5.7 设置显示选项

- gdb中关于显示的选项比较多，这里我只例举大多数常用的选项。
- 打开地址输出，当程序显示函数信息时，gdb会显出函数的参数地址。系统默认为打开的：

  set print address

  set print address *on*

- 例如：

```
(gdb) f
#0 set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
at input.c:530
530 if (lquote != def_lquote)
```

- 关闭函数的参数地址显示：

  `set print address off`

- 例如：

```
(gdb) set print addr off
(gdb) f
#0 set_quotes (lq="<<", rq=">>") at input.c:530
530 if (lquote != def_lquote)
```

- 查看当前地址显示选项是否打开：

  `show print address`

- 打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。这个选项默认是关闭的。与之相关的两个命令如下：

  `set print array`

  `set print array on`

  `set print array off`

  `show print array`

- 下面这个选项主要是设置数组的,如果你的数组太大了,那么就可以指定一个*NUMBER-OF-ELEMENTS*来指定数据显示的最大长度，当到达这个长度时，gdb就不再往下显示了。如果设置为 0，则表示不限制：

  `set print elements NUMBER-OF-ELEMENTS`

- 查看 print elements 的选项信息：

  `show print elements`

- 如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为 off：

  `set print null-stop`

- 如果打开 print pretty 这个选项，那么当gdb显示结构体时会比较漂亮：

  `set print pretty on`

- 例如：

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

- 关闭printf pretty这个选项：

  set print pretty *off*

  gdb显示结构体时会如下显示：

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1},
meat = 0x54 "Pork"}
```

- 查看gdb是如何显示结构体的。

  show print pretty

- 设置字符显示，是否按“\nnn”的格式显示，如果打开，则字符串或字符数据按\nnn显示：

  set print sevenbit-strings

- 查看字符显示开关是否打开：

  show print sevenbit-strings

- 设置显示结构体时，是否显式其内的联合体数据：

  set print union

- 例如有以下数据结构：

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} \
  Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly} \
  Bug_forms;
struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};
struct thing foo = {Tree, {Acorn}};
```

- 当打开这个开关时，执行p *foo*命令后，会如下显示：

```
$1 = {it = Tree, form = {tree = Acorn,
bug = Cocoon}}
```

- 当关闭这个开关时，执行p *foo*命令后，会如下显示：

```
$1 = {it = Tree, form = {...}}
```

- 查看联合体数据的显示方式：

  show print union

- 在 C++ 中，如果一个对象指针指向其派生类，如果打开这个选项，gdb会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，gdb就不管虚函数表了。这个选项默认是 off：

  set print object

- 查看对象选项的设置：

  `show print object`

- 这个选项表示，当显示一个 C++ 对象中的内容是，是否显示其中的静态数据成员。默认是 on：

  `set print static-members`

- 查看静态数据成员选项设置：

  `show print static-members`

- 当此选项打开时，gdb将用比较规整的格式来显示虚函数表时。其默认是关闭的：

  `set print vtbl`

- 查看虚函数显示格式的选项：

  `show print vtbl`

4.5.8 历史记录

- 当你用gdb的print查看程序运行时的数据时，你每一个print都会被gdb记录下来。gdb会以 `$1`，`$2`，`$3` … 这样的方式为你每一个print命令编上号。于是，你可以使用这个编号访问以前的表达式，如 `$1`。这个功能所带来的好处是，如果你先前输入了一个比较长的表达式，如果你还想查看这个表达式的值，你可以使用历史记录来访问，省去了重复输入。

4.5.9 环境变量

- 你可以在gdb的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个gdb的变量很简单只需。使用gdb的set命令。gdb的环境变量和 UNIX 一样，也是以 `$` 起头。如：

  `set $`*foo* `=` *`*object_ptr`*

- 使用环境变量时，gdb会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组：

  `show convenience`

- 该命令查看当前所设置的所有的环境变量。

- 环境变量和程序变量的交互使用是一个比较强大的功能，将使得程序调试更为灵活便捷。例如：

  `set $`*i* `= 0`

  `print` *bar* `[$`*i*`++]->`*contents*

  于是，你就不必`print` *bar* `[0]->`*contents*, `print` *bar* `[1]->`*contents*地输入命令了。输入这样的命令后，只用敲回车，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

4.5.10 查看寄存器

- 要查看寄存器的值，很简单，可以使用如下命令：

  `info registers`

- 查看寄存器的情况（除了浮点寄存器）：

  `info all-registers`

- 查看所有寄存器的情况（包括浮点寄存器）：

  `info registers`

- 查看所指定的寄存器的情况：

- 寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址（ip），程序的当前堆栈地址（sp）等等。你同样可以使用print命令来访问寄存器的情况，只需要在寄存器名字前加一个 $ 符号就可以了。如：p $*ip*。

4.6 改变程序的执行

- 一旦使用gdb挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在gdb中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，比如，你可以在程序的一次运行中走遍程序的所有分支。

4.6.1 修改变量值

- 修改被调试程序运行时的变量值，在gdb中很容易实现，使用gdb的print命令即可完成。如：

```
(gdb) print x=4
```

x=4这个表达式是 C/C++ 的语法，意为把变量x的值修改为 4，如果你当前调试的语言是 Pascal，那么你可以使用 Pascal 的语法：x:=4。

- 在某些时候，很有可能你的变量和gdb中的参数冲突，如：

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

- 因为set width是gdb的命令，所以出现了 “Invalid syntax in expression” 的设置错误，此时，你可以使用set var命令来告诉gdb，*width*不是你gdb的参数，而是程序的变量名，如：

```
(gdb) set var width=47
```

- 另外，还可能有些情况，gdb并不报告这种错误，所以保险起见，在你改变程序变量取值时，最好都使用set var格式的gdb命令。

4.6.2 跳转执行

- 一般来说，被调试程序会按照程序代码的运行顺序依次执行。gdb提供了乱序执行的功能，也就是说，gdb可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由gdb的jump命令来完成：

- 指定下一条语句的运行点。

  jump *LINESPEC*

  其中*LINESPEC*可以是文件的行号，可以是 FILE:LINE 格式，可以是 +NUM 这种偏移量格式。它用于表式着下一条运行语句从哪里开始。

- 或者，直接跳转到指定的代码行的内存地址：

  jump **ADDRESS*

- 注意，jump命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序 Core Dump。所以最好是同一个函数中进行跳转。

- 熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump命令也就是改变了这个寄存器中的值。于是，你可以使用set $*pc*来更改跳转执行的地址。如：

  set $*pc* = 0x485

4.6.3 产生信号

- 使用singal命令，可以产生一个信号给被调试的程序。如：中断信号 Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用gdb产生一个信号，这种精确地在某处产生信号非常有利程序的调试。
- 语法是：signal *SIGNAL*，UNIX 的系统信号通常从 1 到 31。所以*SIGNAL*的取值也在这个范围。
- single命令和 shell 的kill命令不同，系统的kill命令发信号给被调试程序时，是由gdb截获的，而single命令所发出一信号则是直接发给被调试程序的。

4.6.4 强制函数返回

- 如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用return命令强制函数忽略还没有执行的语句并返回。
- 使用return命令取消当前函数的执行，并立即返回，如果指定了*EXPR*，那么该表达式的值会作为函数的返回值。

  return

  return *EXPR*

4.6.5 强制调用函数

- 命令语法：

  call *EXPR*

- 表达式*EXPR*可以是一函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值类型是 void，那么就不显示。
- 另一个相似的命令也可以完成这一功能—print，print后面可以跟表达式，所以也可以用他来调用函数，print和call的不同是：如果函数返回值类型 void，call不显示、print显示函数返回值，并把该值存入历史数据中。

5 其它

其他 Unix 环境下的调试工具

- 集成环境
  - Purify (Rational 公司的商业产品）
  - Insure++ (Parasoft 公司的商业产品）
  - ddd (DataDisplayDebuger, 自由软件，结合gdb使用)
- 函数库

  Mpatrol（检查内存使用的函数库）

  http://www.cbmamiga.demon.co.uk/mpatrol/

6 后记

- gdb是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX 下的软件全是命令行的，这给程序开发提代供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。
- 于是 UNIX 下的软件比 M$ Windows 下的软件更能有机地结合，各自发挥各自的长处，组合成更为强劲的功能。而 M$ Windows 下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。在这里并不是要和 M$ Windows 做个什么比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。（看到这句话时，希望各位千万再也不要认为我就是“鄙视图形界面”，和我抬杠了）
- 我是根据版本为 5.1.1 的gdb所写的这篇文章，所以可能有些功能已被修改，或是又有更为强劲的功能。
- 文中所罗列的gdb的功能时，我只是罗列了一些带用的gdb的命令和使用方法，其实，我这里只讲述的功能大约只占gdb所有功能的 60% 吧，详细的文档，还是请查看gdb的帮助和使用手册吧。
- gdb的自动调试的功能真的很强大，试想，我们在 UNIX 下写个脚本，让脚本自动编译我的程序，被自动调试，并把结果报告出来，调试成功，自动 checkin 源码库。一个命令，编译带着调试带着 checkin，多爽啊。只是gdb对自动化调试目前支持还不是很成熟，只能实现半自动化，真心期望着gdb的自动化调试功能的成熟。

The End of gdb Introduction.