

1、GDB 启动

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，我们必须要把调试信息加到可执行文件中。

使用编译器（cc/gcc/g++）的 `-g` 参数可以做到这一点。如：

```
> cc -g hello.c -o hello
```

```
> g++ -g hello.cpp -o hello
```

如果没有`-g`，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。当你用`-g`把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用gdb来调试他。

启动GDB的方法有以下几种：

1、gdb <program>

program也就是你的执行文件，一般在当然目录下。

2、gdb <program> core

用gdb同时调试一个运行程序和core文件，core是程序非法执行后core dump后产生的文件。

3、gdb <program> <PID>

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，并调试他。

program应该在PATH环境变量中搜索得到。

GDB启动时，可以加上一些GDB的启动开关，详细的开关可以用gdb `-help`查看。

我在下面只例举一些比较常用的参数：

`-symbols <file>`

`-s <file>`

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 <file>`

`-c <file>`

调试时core dump的core文件。

`-directory <directory>`

`-d <directory>`

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。

2、GDB的命令概貌

启动gdb后，就你被带入gdb的调试环境中，就可以使用gdb的命令开始调试程序了，gdb的命令可以使用help命令来查看，如下所示：

```
/home/hchen> gdb
```

```
GNU gdb 5.1.1
```

```
Copyright 2002 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are  
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

```
This GDB was configured as "i386-suse-linux".
```

```
(gdb) help
```

```
List of classes of commands:
```

```
aliases -- Aliases of other commands
```

```
breakpoints -- Making program stop at certain points
```

```
data -- Examining data
```

```
files -- Specifying and examining files
```

```
internals -- Maintenance commands
```

```
obscure -- Obscure features
```

```
running -- Running the program
```

```
stack -- Examining the stack
```

```
status -- Status inquiries
```

```
support -- Support facilities
```

```
tracepoints -- Tracing of program execution without stopping the program
```

```
user-defined -- User-defined commands
```

```
Type "help" followed by a class name for a list of commands in that class.
```

```
Type "help" followed by command name for full documentation.
```

```
Command name abbreviations are allowed if unambiguous.
```

```
(gdb)
```

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 命令，如：help breakpoints，查看设置断点的所有命令。也可以直接help 来查看命令的帮助。

gdb中，输入命令时，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要标志着一个唯一的命令，在Linux下，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 会把其例出来。

示例一：在进入函数func时，设置一个断点。可以敲入break func，或是直接就是b func

```
(gdb) b func
```

```
Breakpoint 1 at 0x8048458: file hello.c, line 10.
```

示例二：敲入b按两次TAB键，你会看到所有b打头的命令：

```
(gdb) b
```

```
backtrace break bt
```

```
(gdb)
```

示例三：只记得函数的前缀，可以这样：

(gdb) b make_ <按TAB键>

(再按下一次TAB键，你会看到:)

make_a_section_from_file make_environ

make_abs_section make_function_type

make_blockvector make_pointer_type

make_cleanup make_reference_type

make_command make_symbol_completion_list

(gdb) b make_

GDB把所有make开头的函数全部例出来给你查看。

示例四：调试C++的程序时，有可以函数名一样。如：

(gdb) b 'bubble(M-?

bubble(double, double) bubble(int, int)

(gdb) b 'bubble(

你可以查看到C++中的所有的重载函数及参数。（注：M-?和“按两次TAB键”是一个意思）

要退出 gdb 时，只用发 quit 或命令简称 q 就行了。

3、GDB中运行UNIX的shell程序

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：

shell <command string>

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Windows中使用Command.com或cmd.exe）

还有一个gdb命令是make：

make <make-args>

可以在gdb中执行make命令来重新build自己的程序。这个命令等价于“shell make <make-args>”。

4、在GDB中运行程序

当以gdb 方式启动gdb后，gdb会在PATH路径和当前目录中搜索的源文件。如要确认gdb是否读到源文件，可使用l或list命令，看看gdb是否能列出源代码。

在 gdb 中，运行程序使用 r 或是 run 命令。程序的运行，你有可能需要设置下面四方面的事。

4.1、程序运行参数。

`set args` 可指定运行时参数。（如：`set args 10 20 30 40 50`）

`show args` 命令可以查看设置好的运行参数。

4.2、运行环境。

`path`

可设定程序的运行路径。

`show paths` 查看程序的运行路径。

`set environment varname [=value]` 设置环境变量。如：`set env USER=hchen`

`show environment [varname]` 查看环境变量。

4.3、工作目录。

`cd <dir>`

相当于shell的cd命令。

`pwd` 显示当前的所在目录。

4.4、程序的输入输出。

`info terminal` 显示你程序用到的终端的模式。

使用重定向控制程序输出。如：`run > outfile`

`tty` 命令可以指写输入输出的终端设备。如：`tty /dev/ttyb`

5、调试已运行的程序

两种方法：

1、在UNIX下用ps查看正在运行的程序的PID（进程ID），然后用gdb PID格式挂接正在运行的程序。

2、先用gdb 关联上源代码，并进行gdb，在gdb中用attach命令来挂接进程的PID。并用detach 来取消挂接的进程。

6、暂停 / 恢复程序运行（断点调试）

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。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停住等等。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，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。

6.1、设置断点（BreakPoint）

我们用`break`命令来设置断点。正面有几点设置断点的方法：

`break`

在进入指定函数时停住。C++中可以使用`class::function`或`function(type, type)`格式来指定函

数名。

`break <function>`

在指定行号停住。

`break +offset`

`break -offset`

在当前行号的前面或后面的offset行停住。offset为自然数。

`break filename:linenum`

在源文件filename的linenum行处停住。

`break filename:function`

在源文件filename的function函数的入口处停住。

`break *address`

在程序运行的内存地址处停住。

`break`

`break`命令没有参数时，表示在下一条指令处停住。

`break ... if <condition>`

...可以是上述的参数，condition表示条件，在条件成立时停住。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。

查看断点时，可使用`info`命令，如下所示：（注：n表示断点号）

`info breakpoints [n]`

`info break [n]`

6.2、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：

`watch <expr>`

为表达式（变量）`expr`设置一个观察点。一旦表达式值有变化时，马上停住程序。

`rwatch <expr>`

当表达式（变量）`expr`被读时，停住程序。

`awatch <expr>`

当表达式（变量）的值被读或被写时，停住程序。

`info watchpoints`

列出当前所设置的所有观察点。

6.3、设置捕捉点（CatchPoint）

你可设置捕捉点来捕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。设置捕捉点的格式为：

`catch <event>`

当`event`发生时，停住程序。`event`可以是下面的内容：

- 1、`throw` 一个C++抛出的异常。（`throw`为关键字）
- 2、`catch` 一个C++捕捉到的异常。（`catch`为关键字）
- 3、`exec` 调用系统调用`exec`时。（`exec`为关键字，目前此功能只在HP-UX下有用）
- 4、`fork` 调用系统调用`fork`时。（`fork`为关键字，目前此功能只在HP-UX下有用）
- 5、`vfork` 调用系统调用`vfork`时。（`vfork`为关键字，目前此功能只在HP-UX下有用）
- 6、`load` 或 `load` 载入共享库（动态链接库）时。（`load`为关键字，目前此功能只在HP-UX下有用）
- 7、`unload` 或 `unload` 卸载共享库（动态链接库）时。（`unload`为关键字，目前此功能只在HP-UX下有用）

`tcatch`

只设置一次捕捉点，当程序停住以后，应点被自动删除。

6.4、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。

`clear`

清除所有的已定义的停止点。

`clear` <function>

`clear` <filename:function>

清除所有设置在函数上的停止点。

`clear` <linenum>

`clear` <filename:linenum>

清除所有设置在指定行上的停止点。

`delete` [breakpoints] [range...]

删除指定的断点，breakpoints为断点号。如果不指定断点号，则表示删除所有的断点。range表示断点号的范围（如：3-7）。其简写命令为d。

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` [breakpoints] [range...]

disable所指定的停止点，breakpoints为停止点号。如果什么都不指定，表示disable所有的停止点。简写命令是dis。

`enable` [breakpoints] [range...]

enable所指定的停止点，breakpoints为停止点号。

`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disable。

`enable` [breakpoints] delete 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。

6.5、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）

`condition` <bnum> <expression>

修改断点号为bnum的停止条件为expression。

`condition` <bnum>

清除断点号为bnum的停止条件。

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。

6.6、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。

```
commands [bnum]
```

```
... command-list ...
```

```
end
```

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。

例如：

```
break foo if x>0
```

```
commands
```

```
printf "x is %d\n", x
```

```
continue
```

```
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，并直接在打个end就行了。

6.7、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：

```
(gdb) b String::after
```

```
[0] cancel
```

```
[1] all
```

```
[2] file:String.cc; line number:867
```

```
[3] file:String.cc; line number:860
```

```
[4] file:String.cc; line number:875
```

```
[5] file:String.cc; line number:853
```

```
[6] file:String.cc; line number:846
```

```
[7] file:String.cc; line number:735
```

```
> 2 4 6
```

```
Breakpoint 1 at 0xb26c: file String.cc, line 867.
```

```
Breakpoint 2 at 0xb344: file String.cc, line 875.
```

```
Breakpoint 3 at 0xafcc: file String.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。

6.8、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。

continue [ignore-count]

c [ignore-count]

fg [ignore-count]

恢复程序运行，直到程序结束，或是下一个断点到来。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。

step <count>

单步跟踪，如果有函数调用，他会进入该函数。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。

next <count>

同样单步跟踪，如果有函数调用，他不会进入该函数。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。

set step-mode

set step-mode on

打开step-mode模式，于是，在进行单步跟踪时，程序不会因为缺少debug信息而不停住。这个参数很利于查看机器码。

set step-mod off

关闭step-mode模式。

finish

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。

until 或 u

当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

stepi 或 si

nexti 或 ni

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。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）

6.9、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。UNIX定义了许多信号，比如SIGINT表示中断字符信号，也就是Ctrl+C的信号，SIGBUS表示硬件故障的信号；SIGCHLD表示子进程状态改变信号；SIGKILL表示终止程序运行的信号，等等。信号量编程是UNIX下非常重要的一种技术。

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。

```
handle <signal> <keywords...>
```

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，表示处理从SIGIO信号到SIGKILL的信号，其中包括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。其可以是以下几种关键字的一个或多个。

```
nostop
```

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。

```
stop
```

当被调试的程序收到信号时，GDB会停住你的程序。

```
print
```

当被调试的程序收到信号时，GDB会显示出一条信息。

```
noprint
```

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。

```
pass
```

```
noignore
```

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。

```
nopass
```

```
ignore
```

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。

```
info signals
```

```
info handle
```

查看有哪些信号在被GDB检测中。

6.10、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。

```
break <linespec> thread <threadno>
break <linespec> thread <threadno> if ...
```

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”命令来查看正在运行程序中的线程信息。如果你不指定thread <threadno>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。如：

```
(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。

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。

```
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。

7、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：

`backtrace`

`bt`

打印当前的函数调用栈的所有信息。如：

`(gdb) bt`

#0 func (n=250) at tst.c:6

#1 0x08048524 in main (argc=1, argv=0xbffff674) at tst.c:30

#2 0x400409ed in __libc_start_main () from /lib/libc.so.6

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

`backtrace`

`bt`

n是一个正整数，表示只打印栈顶上n层的栈信息。

`backtrace <-n>`

`bt <-n>`

-n表一个负整数，表示只打印栈底下n层的栈信息。

如果你要查看某一层的信息，你需要在切换当前的栈，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：frame 0，表示栈顶，frame 1，表示栈的第二层。

`up <n>`

表示向栈的上面移动n层，可以不打n，表示向上移动一层。

`down <n>`

表示向栈的下面移动n层，可以不打n，表示向下移动一层。

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。

`info frame`

`info f`

这个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存地址。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。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
```

```
info args
```

打印出当前函数的参数名及其值。

```
info locals
```

打印出当前函数中所有局部变量及其值。

```
info catch
```

打印出当前的函数中的异常处理信息。

8、查看源程序

8.1、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。还是来看一看查看源代码的GDB命令吧。

```
list <linenum>
```

显示程序第linenum行的周围的源程序。

```
list <function>
```

显示函数名为function的函数的源程序。

```
list
```

显示当前行后面的源程序。

```
list -
```

显示当前行前面的源程序。

一般是打印当前行的上5行和下5行，如果显示函数是上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。

```
set listsize <function>
```

设置一次显示源代码的行数。

```
show listsize
```

查看当前listsize的设置。

list命令还有下面的用法:

list <first>, <last>

显示从first行到last行之间的源代码。

list , <last>

显示从当前行到last行之间的源代码。

list +

往后显示源代码。

一般来说在list后面可以跟以下这们的参数:

行号。

<+offset> 当前行号的正偏移量。

<-offset> 当前行号的负偏移量。

<filename:linenum>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。

8.2、搜索源代码

不仅如此, GDB还提供了源代码搜索的命令:

forward-search <regexp>

search <regexp>

向前面搜索。

reverse-search <regexp>

全部搜索。

其中, 就是正则表达式, 也主一个字符串的匹配模式, 关于正则表达式, 我就不在这里讲了, 还请各位查看相关资料。

8.3、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。

directory <dirname ... >

dir <dirname ... >

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“:”, Windows下你可以使用“;”。

directory

清除所有的自定义的源文件搜索路径信息。

show directories

显示定义了的源文件搜索路径。

8.4、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。`info line`后面可以跟“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`）你可以查看源程序的当前执行时的机器码，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 $0x0,0xffffffffc(%ebp)
0x804845d : movl $0x1,0xffffffff8(%ebp)
0x8048464 : mov 0xffffffff8(%ebp),%eax
0x8048467 : cmp 0x8(%ebp),%eax
0x804846a : jle 0x8048470
0x804846c : jmp 0x8048480
0x804846e : mov %esi,%esi
0x8048470 : mov 0xffffffff8(%ebp),%eax
0x8048473 : add %eax,0xffffffffc(%ebp)
0x8048476 : incl 0xffffffff8(%ebp)
0x8048479 : jmp 0x8048464
0x804847b : nop
0x804847c : lea 0x0(%esi,1),%esi
0x8048480 : mov 0xffffffffc(%ebp),%edx
0x8048483 : mov %edx,%eax
0x8048485 : jmp 0x8048487
0x8048487 : mov %ebp,%esp
0x8048489 : pop %ebp
0x804848a : ret
```

```
End of assembler dump.
```

9、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>
```

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），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。

9.1、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const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。（而关于用GDB调试其它语言的章节，我将在后面介绍）

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。

{}

表示一个指向内存地址的类型为type的一个对象。

9.2、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。例如，查看文件f2.c中的全局变量x的值：

```
gdb) p 'f2.c'::x
```

当然，“::”操作符会和C++中的发生冲突，GDB能自动识别“::”是否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。

9.3、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，“@”的右边则是你想查看内存的长度。例如，你的程序中有这样的语句：

```
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：

```
(gdb) p *array@len
```

```
$1 = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 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。

9.4、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。

例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用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。

```
(gdb) p i
```

```
$21 = 101
```

```
(gdb) p/a i
```

```
$22 = 0x65
```

```
(gdb) p/c i
```

```
$23 = 101 'e'
```

```
(gdb) p/f i
$24 = 1.41531145e-43
(gdb) p/x i
$25 = 0x65
(gdb) p/t i
$26 = 1100101
```

9.5、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/<n/f/u> <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，如果地址是指令地址，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bytes。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会从指定内存地址开始，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。

9.6、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 display <dnums...>
```

```
enable display <dnums...>
```

disable和enable不删除自动显示的设置，而只是让其失效和恢复。

```
info display
```

查看display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。

9.7、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。

```
set print address
```

```
set print address on
```

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，如：

```
(gdb) f
```

```
#0 set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
```

```
at input.c:530
```

```
530 if (lquote != def_lquote)
```

```
set print address off
```

关闭函数的参数地址显示，如：

```
(gdb) set print addr off
```

```
(gdb) f
```

```
#0 set_quotes (lq="<<", rq=">>") at input.c:530
```

```
530 if (lquote != def_lquote)
```

```
show print address
```

查看当前地址显示选项是否打开。

```
set print array
```

```
set print array on
```

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，我就不再多说了。

```
set print array off
```

```
show print array
```

```
set print elements
```

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。

```
show print elements
```

查看print elements的选项信息。

```
set print null-stop
```

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 pretty on
```

如果打开printf pretty这个选项，那么当GDB显示结构体时会比较漂亮。如：

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

```
set print pretty off
```

关闭printf pretty这个选项，GDB显示结构体时会如下显示：

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

```
show print pretty
```

查看GDB是如何显示结构体的。

```
set print sevenbit-strings
```

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，如“\065”。

```
show print sevenbit-strings
```

查看字符显示开关是否打开。

```
set print union
```

设置显示结构体时，是否显式其内的联合体数据。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 = {...}}
```

```
show print union
```

查看联合体数据的显示方式

```
set print object
```

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。

```
show print object
```

查看对象选项的设置。

```
set print static-members
```

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。

```
show print static-members
```

查看静态数据成员选项设置。

```
set print vtbl
```

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。

```
show print vtbl
```

查看虚函数显示格式的选项。

9.8、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。

9.9、GDB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。

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

例如：

```
set $i = 0
```

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

于是，当你就不必，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。

十、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 <regname ...>
```

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 \$eip。

10、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。

10.1、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

set命令只能设置内置类型以及POD类型的变量{}}
不过我们可以通过其他方式改变某个类对象的值，比如调用一个辅助函数，该函数可以改变类对象的值，在gdb用call命令或是print命令调用该函数，这样就可以在gdb中动态改变某个类对象的值了。

在GDB中访问类对象的成员是不检查可访问性的，比如可以直接访问某个类对象中的private成员（包括成员变量或成员函数）

时，最好都使用set var格式的GDB命令。

10.2、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：

```
jump <linespec>
```

指定下一条语句的运行点。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。表示着下一条运行语句从哪里开始。

```
jump <address>
```

这里的 <address> 是代码行的内存地址。

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
```

10.3、产生信号量

使用singal命令，可以产生一个信号量给被调试的程序。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产生一个信号量，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 <singal>，UNIX的系统信号量通常从1到15。所以取值也在这个范围。

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。

10.4、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。

```
return
```

```
return <expression>
```


使用return命令取消当前函数的执行，并立即返回，如果指定了，那么该表达式的值会被认作函数的返回值。

10.5、强制调用函数

`call <expr>`

表达式中可以是一函数，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void，call则不显示，print则显示函数返回值，并把该值存入历史数据中。

11、在不同语言中使用GDB

GDB支持下列语言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般说来，GDB会根据你所调试的程序来确定当然的调试语言，比如：发现文件名后缀为“.c”的，GDB会认为是C程序。文件名后缀为“.C, .cc, .cp, .cpp, .cxx, .c++”的，GDB会认为是C++程序。而后缀是“.f, .F”的，GDB会认为是Fortran程序，还有，后缀为如果是“.s, .S”的会认为是汇编语言。

也就是说，GDB会根据你所调试的程序的语言，来设置自己的语言环境，并让GDB的命令跟着语言环境的改变而改变。比如一些GDB命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。例如C/C++中对指针的语法是*p，而在Modula-2中则是p^。并且，如果你当前的程序是由几种不同语言一同编译成的，那到在调试过程中，GDB也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，真是体贴开发人员的一种设计。

下面是几个关于GDB语言环境的命令：

`show language`

查看当前的语言环境。如果GDB不能识为你所调试的编程语言，那么，C语言被认为是默认的环境。

`info frame`

查看当前函数的程序语言。

`info source`

查看当前文件的程序语言。

如果GDB没有检测出当前的程序语言，那么你也可以手动设置当前的程序语言。使用set language命令即可做到。

当set language命令后什么也不跟的话，你可以查看GDB所支持的语言种类：

(gdb) set language

The currently understood settings are:

local or auto Automatic setting based on source file

c Use the C language

c++ Use the C++ language

asm Use the Asm language

chill Use the Chill language

fortran Use the Fortran language

java Use the Java language

modula-2 Use the Modula-2 language

pascal Use the Pascal language

scheme Use the Scheme language

于是你可以在 set language 后跟上被列出来的程序语言名，来设置当前的语言环境。