RT Embedded http://www.kontronn.com

# 常用 GDB 命令中文速览


## 目录

- break -- 在指定的行或函数处设置断点，缩写为 `b`
- info breakpoints -- 打印未删除的所有断点，观察点和捕获点的列表，缩写为 `i b`
- disable -- 禁用断点，缩写为 `dis`
- enable -- 启用断点
- clear -- 清除指定行或函数处的断点
- delete -- 删除断点，缩写为 `d`
- tbreak -- 设置临时断点，参数同 `break`，但在程序第一次停住后会被自动删除
- watch -- 为表达式（或变量）设置观察点，当表达式（或变量）的值有变化时，暂停程序执行
- step -- 单步跟踪，如果有函数调用，会进入该函数，缩写为 `s`
- reverse-step -- 反向单步跟踪，如果有函数调用，会进入该函数
- next -- 单步跟踪，如果有函数调用，不会进入该函数，缩写为 `n`
- reverse-next -- 反向单步跟踪，如果有函数调用，不会进入该函数
- return -- 使选定的栈帧返回到其调用者
- finish -- 执行直到选择的栈帧返回，缩写为 `fin`
- until -- 执行直到达到当前栈帧中当前行后的某一行（用于跳过循环、递归函数调用），缩写为 `u`

- continue -- 恢复程序执行，缩写为 `c`
- print -- 打印表达式 EXP 的值，缩写为 `p`
- x -- 查看内存
- display -- 每次程序停止时打印表达式 EXP 的值（自动显示）
- info display -- 打印早先设置为自动显示的表达式列表
- disable display -- 禁用自动显示
- enable display -- 启用自动显示
- undisplay -- 删除自动显示项
- help -- 打印命令列表（带参数时查找命令的帮助），缩写为 `h`
- attach -- 挂接到已在运行的进程来调试
- run -- 启动被调试的程序，缩写为 `r`
- backtrace -- 查看程序调用栈的信息，缩写为 `bt`
- ptype -- 打印类型 TYPE 的定义

---

## break

使用 `break` 命令（缩写 `b`）来设置断点。

用法：

- `break` 当不带参数时，在所选栈帧中执行的下一条指令处设置断点。
- `break <function-name>` 在函数体入口处打断点，在 C++ 中可以使用 `class::function` 或 `function(type, ...)` 格式来指定函数名。
- `break <line-number>` 在当前源码文件指定行的开始处打断点。
- `break -N` `break +N` 在当前源码行前面或后面的 `N` 行开始处打断点，`N` 为正整数。
- `break <filename:linenum>` 在源码文件 `filename` 的 `linenum` 行处打断点。
- `break <filename:function>` 在源码文件 `filename` 的 `function` 函数入口处打断点。
- `break <address>` 在程序指令的地址处打断点。
- `break ... if <cond>` 设置条件断点，`...` 代表上述参数之一（或无参数），`cond` 为条件表达式，仅在 `cond` 值非零时暂停程序执行。

## info breakpoints

查看断点，观察点和捕获点的列表。

用法：

- `info breakpoints [list...]`
- `info break [list...]`
- `list...` 用来指定若干个断点的编号（可省略），可以是 `2`， `1-3`， `2 5` 等。

## disable

禁用一些断点。参数是用空格分隔的断点编号。要禁用所有断点，不加参数。

禁用的断点不会被忘记，但直到重新启用才有效。

用法：

- `disable [breakpoints] [list...]`
- `breakpoints` 是 `disable` 的子命令（可省略），`list...` 同 `info breakpoints` 中的描述。

## enable

启用一些断点。给出断点编号（以空格分隔）作为参数。没有参数时，所有断点被启用。

用法：

- `enable [breakpoints] [list...]` 启用指定的断点（或所有定义的断点）。
- `enable [breakpoints] once list...` 临时启用指定的断点。GDB 在停止您的程序后立即禁用这些断点。
- `enable [breakpoints] delete list...` 使指定的断点启用一次，然后删除。一旦您的程序停止，GDB 就会删除这些断点。等效于用 `tbreak` 设置的断点。

`breakpoints` 同 `disable` 中的描述。

## clear

在指定行或函数处清除断点。参数可以是行号，函数名称或 `*` 跟一个地址。

用法：

- `clear` 当不带参数时，清除所选栈帧在执行的源码行中的所有断点。
- `clear <function>`, `clear <filename:function>` 删除在命名函数的入口处设置的任何断点。
- `clear <linenum>`, `clear <filename:linenum>` 删除在指定的文件指定的行号的代码中设置的任何断点。
- `clear <address>` 清除指定程序指令的地址处的断点。

## delete

删除一些断点或自动显示表达式。参数是用空格分隔的断点编号。要删除所有断点，不加参数。

用法： `delete [breakpoints] [list...]`

## tbreak

设置临时断点。参数形式同 `break` 一样。

除了断点是临时的之外，其他同 `break` 一样，所以在命中时会被删除。

## watch

为表达式设置观察点。

用法： `watch [-l|-location] <expr>` 每当一个表达式的值改变时，观察点就会暂停程序执行。

如果给出了 `-l` 或者 `-location`，则它会对 `expr` 求值并观察它所指向的内存。例如，`watch *(int *)0x12345678` 将在指定的地址处观察一个 4 字节的区域（假设 int 占用 4 个字节）。

## step

单步执行程序，直到到达不同的源码行。

用法： `step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

警告：如果当控制在没有调试信息的情况下编译的函数中使用 `step` 命令，则执行将继续进行，直到控制到达具有调试信息的函数。 同样，它不会进入没有调试信息编译的函数。

要执行没有调试信息的函数，请使用 `stepi` 命令，详见后文。

## reverse-step

反向单步执行程序，直到到达另一个源码行的开头。

用法： `reverse-step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

## next

单步执行程序，执行完子程序调用。

用法： `next [N]`

与 `step` 不同，如果当前的源代码行调用子程序，则此命令不会进入子程序，而是将其视为单个源代码行，继续执行。

## reverse-next

反向步进程序，执行完子程序调用。

用法： `reverse-next [N]`

如果要执行的源代码行调用子程序，则此命令不会进入子程序，调用被视为一个指令。

参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

## return

您可以使用 `return` 命令取消函数调用的执行。如果你给出一个表达式参数，它的值被用作函数的返回值。

用法： `return <expression>` 将 `expression` 的值作为函数的返回值并使函数直接返回。

## finish

执行直到选定的栈帧返回。

用法： `finish` 返回后，返回的值将被打印并放入到值历史记录中。

## until

执行直到程序到达当前栈帧中当前行之后（与 break[14] 命令相同的参数）的源码行。此命令用于通过一个多次的循环，以避免单步执行。

用法：`until <location>` 或 `u <location>` 继续运行程序，直到达到指定的位置，或者当前栈帧返回。

## continue

在信号或断点之后，继续运行被调试的程序。

用法： `continue [N]` 如果从断点开始，可以使用数字 `N` 作为参数，这意味着将该断点的忽略计数设置为 `N - 1`(以便断点在第 N 次到达之前不会中断)。如果启用了非停止模式（使用 `show non-stop` 查看），则仅继续当前线程，否则程序中的所有线程都将继续。

## print

求值并打印表达式 EXP 的值。可访问的变量是所选栈帧的词法环境，以及范围为全局或整个文件的所有变量。

用法：

- `print [expr]` 或 `print /f [expr]` `expr` 是一个（在源代码语言中的）表达式。

默认情况下，`expr` 的值以适合其数据类型的格式打印；您可以通过指定 `/f` 来选择不同的格式，其中 `f` 是一个指定格式的字母；详见输出格式[17]。

如果省略 `expr`，GDB 再次显示最后一个值。

要以每行一个成员带缩进的格式打印结构体变量请使用命令 `set print pretty on`，取消则使用命令 `set print pretty off`。

可使用命令 `show print` 查看所有打印的设置。

## x

检查内存。

用法： `x/nfu <addr>` 或 `x <addr>` `n`、`f` 和 `u` 都是可选参数，用于指定要显示的内存以及如何格式化。`addr` 是要开始显示内存的地址的表达式。

`n` 重复次数（默认值是 1），指定要显示多少个单位（由 `u` 指定）的内存值。

`f` 显示格式（初始默认值是 `x`），显示格式是 `print('x','d','u','o','t','a','c','f','s')` 使用的格式之一，再加 `i`（机器指令）。

`u` 单位大小，`b` 表示单字节，`h` 表示双字节，`w` 表示四字节，`g` 表示八字节。

例如：

`x/3uh 0x54320` 表示从地址 0x54320 开始以无符号十进制整数的格式，双字节为单位来显示 3 个内存值。

`x/16xb 0x7f95b7d18870` 表示从地址 0x7f95b7d18870 开始以十六进制整数的格式，单字节为单位显示 16 个内存值。

## display

每次程序暂停时，打印表达式 EXP 的值。

用法： `display <expr>`，`display/fmt <expr>` 或 `display/fmt <addr>fmt` 用于指定显示格式。像 print[20] 命令里的 `/f` 一样。

对于格式 `i` 或 `s`，或者包括单位大小或单位数量，将表达式 `addr` 添加为每次程序停止时要检查的内存地址。

## info display

打印自动显示的表达式列表，每个表达式都带有项目编号，但不显示其值。

包括被禁用的表达式和不能立即显示的表达式（当前不可用的自动变量）。

## undisplay

取消某些表达式在程序暂停时的自动显示。参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示取消所有自动显示表达式。

`delete display` 具有与此命令相同的效果。

## disable display

禁用某些表达式在程序暂停时的自动显示。禁用的显示项目不会被自动打印，但不会被忘记。 它可能稍后再次被启用。

参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示禁用所有自动显示表达式。

## enable display

启用某些表达式在程序暂停时的自动显示。

参数是重新显示的表达式的编号（使用 `info display` 查询编号）。不带参数表示启用所有自动显示表达式。

## help

打印命令列表。

您可以使用不带参数的 `help`（缩写为 `h`）来显示命令的类别名的简短列表。
使用 `help <class>` 您可以获取该类中的各个命令的列表。使用 `help <command>` 显示如何使用该命令。

## attach

挂接到 GDB 之外的进程或文件。该命令可以将进程 ID 或设备文件作为参数。

对于进程 ID，您必须具有向进程发送信号的权限，并且必须具有与调试器相同的有效的 uid。

用法： `attach <process-id>` GDB 在安排调试指定的进程之后做的第一件事是暂停该进程。
无论是通过 `attach` 命令挂接的进程还是通过 `run` 命令启动的进程，您都可以使用的 GDB 命令来检查和修改挂接的进程。

## run

启动被调试的程序。

可以直接指定参数，也可以用 set args[24] 设置（启动所需的）参数。
例如： `run arg1 arg2 ...` 等效于

```
1. set args arg1 arg2 ...
2. run
```

还允许使用 `>`、 `<` 或 `>>` 进行输入和输出重定向。

## backtrace

打印整体栈帧信息。

- `bt` 打印整体栈帧信息，每个栈帧一行。
- `bt n` 类似于上，但只打印最内层的 n 个栈帧。
- `bt -n` 类似于上，但只打印最外层的 n 个栈帧。
- `bt full n` 类似于 `bt n`，还打印局部变量的值。

`where` 和 `info stack`(缩写 `info s`) 是 `backtrace` 的别名。调用栈信息类似如下：

```
1. (gdb) where
2. #0  vconn_stream_run (vconn=0x99e5e38) at lib/vconn-
   stream.c:232
3. #1  0x080ed68a in vconn_run (vconn=0x99e5e38) at
   lib/vconn.c:276
4. #2  0x080dc6c8 in rconn_run (rc=0x99dbbe0) at lib/rconn.c:513
5. #3  0x08077b83 in ofconn_run (ofconn=0x99e8070,
   handle_openflow=0x805e274 <handle_openflow>) at
   ofproto/connmgr.c:1234
6. #4  0x08075f92 in connmgr_run (mgr=0x99dc878,
   handle_openflow=0x805e274 <handle_openflow>) at
   ofproto/connmgr.c:286
7. #5  0x08057d58 in ofproto_run (p=0x99d9ba0) at
   ofproto/ofproto.c:1159
8. #6  0x0804f96b in bridge_run () at vswitchd/bridge.c:2248
9. #7  0x08054168 in main (argc=4, argv=0xbf8333e4) at
   vswitchd/ovs-vswitchd.c:125
```

## ptype

打印类型 TYPE 的定义。

用法： `ptype[/FLAGS] TYPE-NAME | EXPRESSION`

参数可以是由 `typedef` 定义的类型名， 或者 `struct STRUCT-TAG` 或者 `class CLASS-NAME` 或者 `union UNION-TAG` 或者 `enum ENUM-TAG`。

根据所选的栈帧的词法上下文来查找该名字。

类似的命令是 `whatis`，区别在于 `whatis` 不展开由 `typedef` 定义的数据类型，而 `ptype` 会展开，举例如下：

```
1. /* 类型声明与变量定义 */
2. typedef double real_t;
3. struct complex {
4.     real_t real;
5.     double imag;
6. };
7. typedef struct complex complex_t;
8.
9. complex_t var;
10.  real_t *real_pointer_var;
```

这两个命令给出了如下输出：

```
1. (gdb) whatis var
2. type = complex_t
3. (gdb) ptype var
4. type = struct complex {
5.     real_t real;
6.     double imag;
7. }
8. (gdb) whatis complex_t
9. type = struct complex
10.  (gdb) whatis struct complex
11.  type = struct complex
12.  (gdb) ptype struct complex
13.  type = struct complex {
14.      real_t real;
15.      double imag;
```

```
16.  }
17.  (gdb) whatis real_pointer_var
18.  type = real_t *
19.  (gdb) ptype real_pointer_var
20.  type = double *
```