GDB 学习笔记

---- GDB 7.2.50.20101009

1. 调用 GDB

可以通过命令行来调用 GBD，下面是 GDB 的命令行形式：

```
gdb [options] [program [core|pid]]
```

通常运行 GDB 的方式是：

gdb program	调试可执行文件 program。
gdb program core	调试可执行文件 program 以及相关的 core 文件。
gdb program 1234	调试一个正在运行的进程 program，1234 是这个进程的 id。

GDB 接受的选项：（斜体字如 *file* 是选项参数）

-symbols *file* -s *file*	从 *file* 中读取符号表。
-exec *file* -e *file*	将 *file* 作为可执行文件，并且和 core dump 生成的纯数据文件一起使用。
-se *file*	从 *file* 中读取符号表，并将它作为可执行文件。*File* 与在 gdb 参数中指定的可执行程序的作用的相同的。
-core *file* -c *file*	将 *file* 作为 core dump 生成的文件。
-pid *number* -p *number*	使用 *number* 作为进程 ID，作用与 attach 的参数一致。
-command *file* -x *file*	从 *file* 文件中执行命令，作用与 source 命令的参数一致。
-eval-command *command* -ex *command*	执行 gdb 命令 *command*。
-directory *directory* -d *directory*	将 *directory* 添加到搜索源代码和脚本的路径中。
-readnow -r	立刻读取全部的符号表，而不是等到需要的时候再读取。
-nx -n	不执行来自于任何初始化文件的命令。通常 gdb 处理完命令行选项和参数之后执行这些文件中的命令。
-quiet -silent -q	不打印出介绍和版本信息。
-batch	运行批量模式。在成功的处理所有的使用-x 指定的命令文件和来自于初始化文件中的命令后返回 0，如果执行命令文件中的 gdb 命令时出现错误，返回非 0。
-batch-silent	运行批量模式，但是完全不现实介绍和版本信息。
-return-child-result	返回被 gdb 调试的进程的推出代码。除非发生下面的异常情况： 1)gdb 异常退出。由于参数错误或者 gdb 内部错误造成的，这样的情况下 gdb 的返回值和没有使用-return-child-result 选项一致。 2)用户明确的指定了推出码，如 quit 1。 3)子进程没有运行或者没有被允许终止，这种情况返回-1。
-nowindows -nw	如果 gdb 默认使用 GUI，那么这个选项告诉 gdb 仅仅使用命令行接口。如果 GUI 不可用那么这个选项被忽略。
-windows	如果 gdb 包含 GUI，告诉 gdb 尽量使用 GUI。

-w	
-cd *directory*	使用 *directory* 作为 gdb 的工作目录，而不是当前目录。
-fullname -f	告诉 gdb 在标准中输出完整的文件名和行号。当 GNU Emacs 将 gdb 作为子进程运行的时候设置这些选项。
-epoch	当 Epoch Emacs 将 gdb 作为子进程运行的时候设置这个选项。主要是使 gdb 的输出符合 Epoch 的格式。
-annotate *level*	设置 gdb 内部的 *annotate level*。和 set annotate *level* 命令的效果是一致的。
--args	改变命令行的解释，这个选项将可执行文件后面的选项都作为可执行文件的选项传递给它。这个选项停止了 gdb 的命令行解析，例如： gdb --args gcc -O2 -c foo.c 可以用来调试命令行选项为-O2 -c foo.c 时的 gcc。
-baud *bps* -b *bps*	设置使用 gdb 调试远程目标时的串行口的波特率。
-l *timeout*	设置使用 gdb 调试远程目标时候的链接超时时间。
-tty *device* -t *device*	使用设备 *device* 作为被调试程序的输入和输出。
-tui	启动时使用文本用户接口，与 gdbtui 效果一致。
-interpreter *interp*	使用 *interp* 作为控制程序或者设备的接口。
-write	允许对可执行文件和 core 文件读写。
-statistics	这个选项导致 gdb 在完成每一个命令时打印关于时间和内存使用情况的统计信息。
-version	打印 gdb 版本信息。
-help -h	列出所有的选项以及简要的解释。

2. 退出 GDB

quit [*expression*] q Ctrl-D	退出 gdb，如果要制定 gdb 的推出代码请使用 *expression*。

3. 执行 shell 命令

shell *command string*	调用标准的 shell 并执行命令 *command string*。环境变量 SHELL 决定执行的 shell，如没有这个环境变量则使用标准的 shell。
make *make-args*	使用特殊的目标运行 make 程序，与 shell make *make-args* 一致。

4. log 输出

gdb 的 log 记录的 gdb 命令的输出结果。

set logging *switch*	设置 log 开关，*switch* 是：

	on：打开 log。 off：关闭 log。
set logging file *file*	将当前 log 文件的名字改为 *file*，默认的名字为 gdb.txt。
set logging overwrite *switch*	设置 log 文件的覆盖开关，switch 是： on：允许覆盖。 off： 不允许覆盖。
set logging redirect *switch*	设置 log 文件的重定向开关，默认情况下 gdb 输出显示在终端上也输出到 log 中，switch 是： on：仅仅输出到 log 文件中。 off：同时输出到终端和 log 文件中。
show logging	显示当前的 log 设置。

5. 命令帮助措施

命令补全：(<TAB>表示 tab 键)

<TAB>	补全命令，变量等等 gdb 中可以出现的符号。有唯一前缀的时候直接补全，有多个可选前缀的时候第一次响铃提示，第二次列出所有的可选符号。
<ALT>-? <ALT>-<SHIFT>-/	显示所有的可选补全方案。

帮助命令：

help h	列出所有的命令类别。
help *class*	列出所有属于 *class* 类别的命令。
help *command*	列出如何使用 *command* 的信息。
apropos *args*	搜索所有的 gdb 命令和文档，列出所有与正则表达式 *args* 匹配的内容。
complete *args*	列出所有以 *args* 开头的命令的补全方案。

信息：

info i	info 命令主要是显示与被 gdb 调试的程序相关的各种信息。
set	set 命令设置变量，命令，属性等的值。
show	show 命令主要是现实与 gdb 本身有关的信息。

6. 启动被调试的程序

启动程序：

run *args* r *args*	run 命令在 gdb 中启动要调试的程序。运行 run 命令前 gdb 必须已经确定了要调试的程序。*args* 是传递给要调试的程序的命令行选析。
start *args*	与 run 命令相同，不同的就是使用 start 启动程序后，程序停止主过程的开始处(C/C++语言编写的程序停在 main 函数开头)。
set exec-wrapper *wrapper*	gdb 使用包装程序 *wrapper* 来启动被调试的程序。

show exec-wrapper	显示当前被设置的包装程序。
unset exec-wrapper	删除被设置的包装程序。
set disable-randomization *switch*	设置本地应用程序虚拟地址空间随机选择，*switch* 是： on：关闭本地应用程序虚拟地址空间随机选择。 off：打开本地应用程序虚拟地址空间随机选择。 不给出 *switch* 与 set disable-randomization on 效果一致。有些时候应用程序单独执行的时候有可能有问题，但是 gdb 调试却没有问题，这可能是程序映射到某个地址的时候才会出现问题，但是 gdb 默认是关闭虚拟地址空间的随机选择的，所以调试这样的情况的时候要使用 set disable-randomization off 来打开虚拟地址空间随机选择，然 gdb 调试的程序与实际中运行的程序行为一致，才有可能再次出现问题。
show disable-randomization	显示当前的本地程序虚拟地址空间随机选择的设置。

程序的参数：

set args *args*	将 *args* 作为下次程序运行时使用的参数。当使用 run 或者 start 启动程序的时候，如果没有使用参数则程序使用 *args* 作为参数。如果 run 或者 start 带有参数则使用 run 或者 start 的参数为实际的参数。
show args	显示下次程序运行的时候使用的参数，这个参数是由 set args 和 run 或者 start 命令的参数决定的。

程序的环境变量：(这个换将变量只影响被调试的程序不影响 gdb)

path *directory*	将 *directory* 添加到 PATH 环境变量的前面并传递给要调试的程序。gdb 搜索被调试的程序也使用这个 PATH 环境变量。
show path	显示 gdb 搜索被调试的程序的路径列表。
show environment [*varname*] show env [*varname*]	显示环境变量 *varname* 的值。
set environment *varname* [=*value*] set env *varname* [=*value*]	将环境变量 *varname* 设置为 *value*。如果没有给出 *value* 那么 *varname* 的只是空值。
unset environment *varname* unset env *varname*	删除环境变量 *varname*。set evn *varname*=和 unset env *varname* 是不同的，前者将环境变量 *varname* 设置为空，后者删掉了环境变量 *varname*。

工作目录：

cd *directory*	将 *directory* 设置为 gdb 的当前工作目录。
pwd	显示 gdb 的当前工作目录。

输入和输出：

info terminal	显示被调试的程序使用的输入输出终端的信息。可以通过 run 命令来重定向输入输出，如：run > output。是将程序的输出重定向到 output 中。
tty *device* set inferior-tty *device*	将被调试的程序的输入输出设备设置为 *device*。
show inferior-tty	显示当前被调试程序的输入输出设备。

7. 调试运行的进程

attach *pid*	链接到 id 为 *pid* 的进程。
detach	将被调试的进程分离。如不使用 detach 命令而是直接关闭 gdb，那么这个进程也会被 gdb 关闭。
kill	杀掉在 gdb 中运行的进程。

8. 调试多个进程

gdb 可以在一个会话中调试多个进程：(gdb 将每一个进程称为下属进程 inferior)

info inferiors	显示当前 gdb 管理的所有的下属进程的信息。gdb 显示的内容包括，1)gdb 赋予的下属进程号。2)系统的进程标识符。3)下属进程的名字。带有*的是当前的下属信息。例如： (gdb) info inferiros Num Description Executable 2 process 3207 hello * 1 process 3411 goodbye
inferior *infno*	将 *infno* 切换为当前的下属进程，*infno* 是 gdb 赋予的下属进程号。
add-inferior [-copies *n*] [-exec *executable*]	将 *executable* 的 *n* 个拷贝添加到当前的调试会话中。*N* 默认是 1。
clone-inferior [-copies *n*] [*infno*]	克隆 *n* 个下属进程号为 *infno* 的下属进程。*n* 默认是 1，*infno* 默认是当前下属进程。
remove-inferior *infno*	移除下属进程 *infno*。这个命令不能够移除正在运行的下属进程。(描述不是<null>的下属进程)
detach inferior *infno*	从当前调试会话中范例下属进程 *infno*。分离后的下属集成的实体仍然在当前会话中，但是描述为<null>，可以使用 remove-inferior 来删除。
kill inferior *infno*	杀死下属进程 *infno*。杀死之后下属进程仍然保留在当前会话中，但是描述为<null>，可以使用 remove-inferiro 来删除。
set print inferior-events *switch*	当一个下属进程在 gdb 的控制下启动或者退出或者被分离，gdb 会发出提示，这个命令就是用来控制这个提示的，*switch* 是： on：允许在一个下属进程启动或者退出或者被分离的时候发出提示。 off：关闭提示。 当没有给出 switch 的时候等于 set print inferior-events on。
show print inferior-events	显示当前的 print inferiro-events 的设置。

gdb 可以在一些平台上调试通过 fork 生成的多个进程，默认的情况下当使用 fork 生成子进程的时候，gdb 调试父进程，并且子进程无限制的运行。下面是相关的一些命令：

set follow-fork-mode *mode*	设置调用 fork 后 gdb 的调试模式，*mode* 是： parent：fork 之后调试原来的进程，子进程无限制的运行，这是默认情况。 child：fork 之后调试子进程，父进程无限制的运行。
show follow-fork-mode	显示 fork 之后的 gdb 调试模式。
set detach-on-fork *mode*	设置 fork 之后子进程(或者是父进程，这依赖于 follow-fork-mode)被分离还是依然被 gdb 控制，*mode* 是：

	on：没有被调试的进程被分离，并且不依赖于 gdb 运行，这是默认情况。 off：两个进程都在 gdb 的控制之下，一个进程被调试，另一个被挂起。
show detach-on-fork	显示 fork 之后 gdb 对于两个进程的控制情况。
set follow-exec-mode *mode*	设置 fork 后执行 exec 调用新的可执行文件，新的镜像是否替换原有的进程，*mode* 是： new：gdb 创建一个新的下属进程绑定到新的进程中，运行 exec 的进程依然存在。 same：新的进程替换原有进程的镜像，这个默认形式。

9. 调试多线程

当 gdb 识别出一个线程的时候，它使用[New *systag*]这样的形式显示一个线程提示。*systag* 的样式与系统相关，例如在 GNU/Linux 上是[New Thread 46912507313328 (LWP 25582)]，在 SGI 系统上是[New process 386]，下面是进程相关的命令：

info threads	显示当前进程中的线程信息。Gdb 为每一个线程显示如下信息：1)gdb 赋予的线程号。2)系统的线程描述符。3)线程的当前堆栈概要。前面带有*的是当前线程。例如： (gdb) info threads 3 process 35 thread 27 0x34e5 in sigpause() 2 process 35 thread 23 0x34e5 in sigpause() * 1 process 35 thread 13 main (argc=1, argv=0x7ffffff8) at threadtest.c:68
thread *threadno*	将线程 *threadno* 切换到当前线程。*threadno* 是 gdb 赋予的线程号。
thread apply [*threadno*] [*all*] *command*	对于一个或者多个线程实行命令 *command*。*threadno* 是 gdb 赋予的线程号。*all* 表示全部线程。*threadno* 还可以使用范围如 2-5。
set print thread-events *switch*	这个控制着当 gdb 注意到线程启动或者线程结束的时候是否打印消息，*switch* 是： on：打印消息。 off：不打印消息。 没有给出 *switch* 与 set print thread-events on 一致。
show print thread-events	显示 print thread-events 的设置。
set libthread-db-search-path [*path*]	设置 libthread_db 的搜索路径，如果省略了 *path* 则将 libthread-db-search-path 设置为空。
show libthread-db-search-path	显示 libthread_db 的搜索路径。
set debug libthread-db *switch*	控制 libthread_db 相关事件的现实，switch 是： 1：显示。 0：不显示。
shwo debug libthread-db	显示 libthread_db 相关事件的设置。
set scheduler-locking *switch*	设置 All-Stop 模式下线程调度锁模式，*switch* 是： off：其他的线程不会被锁定，可以在任何时候运行。 on：当下属进程重启之后，只有当前线程可以运行。只有在当前线程执行相对长时间的时候其他线程才有机会运行。
show scheduler-locking	显示线程调度所模式。

set schedule-multiple *switch*	设置 All-Stop 模式中多个进程中的多个线程的调度锁模式，*switch* 是： on：所有的进程中的所有线程都允许运行。 off：只有当前进程中的线程允许运行。
show schedule-multiple	显示多个进程中多个线程的调度锁模式。
set non-stop *switch*	设置 non-stop 模式，*switch* 是： on：允许 non-stop 模式。 off：关闭 non-stop 模式。
show non-stop	显示 non-stop 模式的设置。
set target-async *switch*	设置后台执行，*switch* 是： on：允许后台执行模式。 off：关闭后台执行模式。
show target-async	显示后台执行模式。
interrupt [-a]	挂起运行的程序。在 all-stop 模式下，interrupt 挂起整个进程，在 non-stop 模式下，interrupt 挂起当前线程，要在 non-stop 模式下挂起所有线程使用 interrupt -a。

10. 回退

gdb 可以使用某些机制让程序回退到以前的一个点，这样从这个点开始就可以重新执行下面的代码。

checkpoint 实际上是在某一点 gdb 为当前的进程复制了一个进程，你可以从 checkpoint 开始继续调试代码，当你发现需要重新调试的时候，你不用重新启动程序，只需要切换到另一个检查点就可以再次重新开始从 checkpoint 位置调试代码。checkpoint 只能生成一个 checkpoint，要需要多个 checkpoint 就要在同样的地方运行多次 checkpoint。每一个 checkpoint 只能用一次，用过之后就不能再次使用了。

checkpoint	在代码的当前位置设置检查点，一次只能设置一个。gdb 给每一个检查点赋予一个检查点号。
info checkpoints	显示所有的检查点。依次显示检查点号，进程 id，代码地址，行号： 2 process 1632 at 0x40051a, file gdb_c.c, line 19 1 process 1613 at 0x40051a, file gdb_c.c, line 19 * 0 process 1610 (main process) at 0x40051a, file gdb_c.c, line 19 带有*的表示当前进程。
restart *checkpoint-id*	切换到 *checkpoint-id* 的进程中，重新从检查点开始运行，这时的变量，寄存器，调用栈都退回到检查点的值。
delete checkpoint *checkpoint-id*	删除检查点 *checkpoint-id*。

下面的命令是程序的执行做 undo 操作，也就是从当前点向回运行直到遇到断点或者是信号。

reverse-continue [*ignore-count*] rc [*ignore-count*]	从当前停止位置向回运行，知道碰到一个断点或者是信号
reverse-step [*count*]	与 step 命令类似，不过是向相反方向运行。
reverse-stepi [*count*]	与 stepi 命令类似，不过是向相反方向运行。
reverse-next [*count*]	与 next 命令类似，不过是向相反方向运行。
reverse-nexti [*count*]	与 nexti 命令类似，不过是向相反方向运行。

reverse-finish	与 finish 命令类似，不过是向相反的方向运行，并且停在函数调用开始的位置。
set exec-direction *mode*	设置程序运行的方向，*mode* 是： reverse：程序的执行方向为反方向，受影响的命令包括 step, stepi, next, nexti, continue, finish, 但是 return 命令不受影响。 forward：程序的执行方向为向前执行，这个默认的。

11. 断点，观察点，捕获点

断点：

break [*location*] [thread *threadnum*] b [*location*] [thread *threadnum*]	在 *location* 设置断点，如没有给出 location 那么在下一条要执行的指令上设置断点。*threadnum* 是指定的线程。
break [*location*] [thread *threadnum*] if *cond* b [*location*] [thread *threadnum*] if *cond*	在 *location* 上设置条件断点，条件为 *cond*。*Cond* 是语言相关的条件表达式。*threadnum* 是指定的线程。
tbreak *args*	一次性断点，当这个断点被触发后就会被自动删除。*Args* 与 break 命令的参数相同。
hbreak *args*	设置硬件断点，普通断点是修改软件指令，但是硬件断点是在不修改指令的情况下插入断点。不是所有的硬件都支持。
thbreak *args*	设置一次性硬件断点。
rbreak [*file:*]*regex*	为匹配正则表达式 *regex* 的函数设置非条件断点，并且打印出设置的断点的列表。一点这些断点被设置了，它们与使用 break 命令设置的断点没什么区别。*file* 用来限定设置断点的文件名。
info breakpoints [*n*] info break [*n*] info b [*n*]	列举出所有的断点，观察点，捕获点，依次列出的是断点的号，类型(断点，观察点，捕获点)，配置，是否可用，地址，源代码的位置，例如： Num Type Disp Enb Address What 2 breakpoint keep y 0x400513 in main at gdb_c.c:17 3 hw watchpoint keep y con 当不知道断点的地址的时候 Address 显示<PENDING>，直到共享库被加载后才知道这个地址，此外当为 C++语言中的重载函数设置断点的时候会显示<MULTIPLE>。
set breakpoint pending *mode*	这个命令控制当不确定断点位置的时候，gdb 的行为，*mode* 是： auto：当 gdb 不能够找到断点的位置的时候，询问你是否创建未确定的断点。这是默认行为。 on：当 gdb 不能够找到断点位置的时候，自动的创建未确定的断点。 off：当 gdb 不能够找到断点位置的时候，提示错误。
show breakpoint pending	显示当 gdb 不能够找到断点位置的时候的行为。
set breakpoint auto-hw *switch*	设置断点的时候是否自动选择断点的类型，*switch* 是： on：设置断点的时候，gdb 自动选择是否使用硬件断点。这个是默认行为。 off：设置断点的时候，gdb 不自动选择使用硬件断点，断点类型是根据命令确定，hbreak 命令总是选择硬件断点。
set breakpoint always-instered *mode*	这个命令设置了当用户使用断点的时候，gdb 是否立即是断点

	指令生效，*mode* 是： on：当用户添加，改变或者退出断点，那么断点指令立即插入到目标代码中。(这个立即是指在用户执行命令之后马上对于断点指令起作用) off：当用户添加新断点后，只有在目标程序再次运行起来之后在插入断点指令(这个再次运行不是指目标程序再次从 main 开始运行，而是例如当前正停止在某一行，然后用户添在下面的代码某处添加了一个新断点，那么用户继续指向查看名利如 p 等，这个时候断点质量并没有添加到目标程序中，当用户使用 c，n，s 等能够让程序指向起来的指令后，新断点才真正的插入到程序中)。删除的断点，质量在程序停下来之后被删掉。 auto：gdb 自己决定什么时候插入或者删除断点质量。这个是默认情况。

观察点是特殊的断点，当观察点观察的量发生改变的时候，它停止程序运行：

watch [-l | -location] *expr* [thread *threadnum*]	为表达式 *expr* 设置观察点。thread *threadnum* 表示只有线程 *threadnum* 改变了表达式 *expr* 的时候 gdb 才停止程序，其他线程修改 *expr* 时 gdb 不会停止程序。-location 表示观察点观察的是以 *expr* 的结果为地址的内存中的值，当 *expr* 的结果指向的内存中的数据改变的时候，gdb 停止程序。
rwatch [-l | -location] *expr* [thread *threadnum*]	只有在 expr 中的数据被读取的时候，gdb 才停止程序。
awatch [-l | -location] *expr* [thread *threadnum*]	只有在 expr 中的数据被写入的时候，gdb 才停止程序。
info watchpoints	列出所有的观察点，格式与 info breakpoints 一致。
set can-use-hw-watchpoints *switch*	控制 gdb 是否为使用硬件观察点，*switch* 是： 0：不使用硬件观察点。 1：使用硬件观察点。
show can-use-hw-watchpoints	显示 gdb 是否使用硬件观察点。

捕获点是特殊的断点，当某些特殊的事件发生时，它停止程序运行：

ecatch *event*	当 *event* 发生的时候，gdb 停止程序的运行，*event* 是： throw：当抛出 C++异常的时候，停止程序。 catch：当捕获 C++异常的时候，停止程序。 exception [unhandled | *name*]：当产生 Ada 异常的时候，*name* 指出了特定的异常，如果没有给出 *name*，那么当 Ada 产生异常的时候 gdb 就停止程序。unhandled 是当没有被捕获的异常的时候 gdb 停止程序。 assert：当 Ada 断言失败的时候，gdb 停止程序。 exec：当调用 exec 的时候，gdb 停止程序。 syscall [*name* | *number*]：当调用系统调用或者从系统调用中返回的时候，gdb 停止程序。可以使用系统调用的名字或者数字来表示系统调用，否则捕获所有的系统调用。 fork：调用 fork 的时候，gdb 停止程序。 vfork：调用 vfork 的时候，gdb 停止程序。
tcatch *event*	设置一个一次性的捕获点，*event* 与 catch 一致。程序停止之后这个捕获点就被自动的删除了。

断点操作：

clear [*location*]	删除 *location* 中的所有断点。*location* 是下面的形

	式： clear *function* clear *filename*:*function* 　　删除函数 *function* 中的所有断点。 clear *linenum* clear *funname*:*linenum* 　　删除 *linenum* 行或者 *linenum* 行包含的代码中的所有断点。 如果没有指定 *location* 那么删除调用堆栈当前级别的所有断点。
delete [breakpoints] [*range*...] d [breakpoints] [*range*...]	删除 *range* 指定的断点，观察点，捕获点。没有指定 *range* 则删除所有的断点，观察点，捕获点。
disable [breakpoints] [*range*...] dis [breakpoints] [*range*...]	停用 *range* 指定的断点，观察点，捕获点。没有指定 *range* 则停用所有的断点，观察点，捕获点。
enable [breakpoints] [[*range*...] | (once | delete) *range*...]	停用 *range* 指定的断点，观察点，捕获点。没有指定 *range* 则停用所有的断点，观察点，捕获点。once 和 delete 启动的临时断点，当断点停止之后，使用 once 启动的断点自动停用，使用 delete 启动的断点自动被删除。
condition *bnum* [*expression*]	为断点，观察点，捕获点 *bnum* 指定停止条件。*expression* 是与语言相关的条件表达式，如果没有给出 *expression* 就是删掉了断点 *bnum* 的条件。
ignore *bnum count*	设置断点 *bnum* 的忽略测试 *count*。当到达断点的次数少于 *count* 的时候，仅仅增加到达次数而不停止。
commands [*range*...] [... *command-list* ...] end	为 *range* 指定的断点，观察点，捕获点添加命令列表，如果不给出 *command-list* 则删掉 *range* 指定的断点，观察点，捕获点的命令列表，如果不给出 *range* 则指定的是最后一个被设置的断点，观察点，捕获点。
save breakpoints [*filename*]	将断点定义保存在文件 *filename* 中。

12. 继续运行以及单步运行

continue [*ignore-count*] c [*ignore-count*] fg [*ignore-count*]	继续运行程序。*ignore-count* 指的是继续运行中忽略的断点的次数。
step [*count*] s [*count*]	执行程序中的 *count* 条语句。step 命令会进入函数调用中。如果没有给出 *count* 则执行一条语句。
next [*count*] n [*count*]	执行程序中的 *count* 条语句。next 命令不会进入函数调用中。如果没有给出 *count* 则执行一条语句。
set step-mode *switch*	设置对于不包含调试信息的函数调用，step 的行为，*switch* 是： on：进入不包含调试信息的函数调用中。 off：跳过不包含调试信息的函数调用。 没有给出 *switch* 与 set step-mode on 是一致的。
show step-mode	显示对于不包含调试信息的函数调用，step 命令的行为。

finish fin	一直运行到当前函数运行结束并打印出返回值。
until [*location*] u [*location*]	一直运行到 *location* 指定的位置或者是当前的调用堆栈结束。不给出 *location* 则一直运行到源代码的当前行结束。
advance *location*	一直运行到 *location* 指定的位置结束。不会受到当前调用堆栈的影响。
stepi [*args*] si [*args*]	单步执行下一条指令。*args* 与 step 中的一致。
nexti [*args*] ni [*args*]	单步执行下一条指令。*args* 与 next 中的一致。

step 和 stepi 的区别看下面的例子，有这样一段代码：

```
28:  cond = 100;
29:  if (cond > 50) { printf("GT 50\n"); }
30:  else
31:  {
32:      printf("LG 50\n");
33:  }
34:  return 0;
```

这行代码，当调试到第 29 行的时候，使用 step 命令的结果是：

```
29    if (cond > 50) { printf("GT 50\n"); }
(gdb) step
GT 50
34    return 0;
```

但是使用 stepi 命令的结果是：

```
29    if (cond > 50) { printf("GT 50\n"); }
(gdb) stepi
0x0000000000400578    29    if (cond > 50) { printf("GT 50\n"); }
(gdb) stepi
0x000000000040057a    29    if (cond > 50) { printf("GT 50\n"); }
(gdb) stepi
0x0000000000400581    29    if (cond > 50) { printf("GT 50\n"); }
(gdb) stepi
GT 50
0x0000000000400586    29    if (cond > 50) { printf("GT 50\n"); }
(gdb) stepi
34    return 0;
```

step 是以行为单位的，stepi 是以指令为单位的。next 和 nexti 也是同样的。

13. 信号

当一个型号到达的时候 gdb 会停止你的程序，然后等待你对于信号的处理。

info signals [*sig*] info handle [*sig*]	不给出 *sig* 的时候打印出 gdb 能够处理的信号，给出 *sig* 只是答应信号 *sig* 相关的信息。
handle *signal* [*keywords*...]	改变 gdb 处理信号 *signal* 的方式。*signal* 可以是信号数或者信号名。all 代表全部的信号。*keywords* 是信号的处理方式： nostop：当信号到来的时候 gdb 不会停止程序，只是打印消息告诉你信号已经到了。

	stop：当信号到来的时候 gdb 停止程序，并且打印消息。 print：当信号发生的时候打印消息。 noprint：完全不管信号的发生，隐含使用 nostop。 pass noignore：将这个信号发送到你的程序让你的程序来处理。 nopass ignore：gdb 不会让你的程序知道这个信号发生了。

14. observer 模式

observer 模式是不允许 gdb 修改程序的状态。

set observer *switch*	设置 observer 模式状态，*switch* 是： on：打开 observer 模式。 off：关闭 observer 模式。
show observer	显示 observer 模式。
set may-write-registers *switch*	控制 gdb 试图对于寄存器的值的修改行为，*switch* 是： on：允许修改。 off：不允许修改。 on 是默认情况。
show may-write-registers	显示 may-write-registers 的模式。
set may-write-memory *switch*	控制 gdb 试图对于内存中的值的修改行为，*switch* 是： on：允许修改。 off：不允许修改。 on 是默认情况。
show may-write-memory	显示 may-write-memory 的模式。
set may-insert-breakpoints *switch*	控制 gdb 试图插入断点，影响所有的断点，包括 gdb 内部断点，*switch* 是： on：允许插入断点。 off：不允许插入断点。 on 是默认情况。
show may-insert-breakpoints	显示 may-insert-breakpoints 模式。
set may-insert-tracepoints *switch*	控制 gdb 试图插入常规跟踪点，*switch* 是： on：允许插入。 off：不允许插入。 on 是默认模式。
show may-insert-tracepoints	显示 may-insert-tracepoints 的模式。
set may-insert-fast-tracepoints *switch*	控制 gdb 试图插入快速跟踪点，*switch* 是： on：允许插入。 off：不允许插入。 on 是默认情况。
set may-interrupt *switch*	控制 gdb 是否允许中断运行的程序，*switch* 是： on：允许中断。 off：不允许中断。 on 是默认情况。
show may-interrupt	显示 may-interrupt 模式。

15. 进程执行的记录和回放

target record target rec	开启进程记录和回放目标。只有在程序运行之后才能够开启进程记录和回放。
record stop rec stop	停止进程记录和回放目标。
record save *filename* rec save *filename*	将记录保存在文件 *filename* 中，默认的名字是 gdb_record.*pid*，*pid* 是进程 id。
record restore *filename* rec restore *filename*	重新执行来自于文件 *filename* 中的可执行记录。*filename* 必须是 record save 创建的。
set record insn-number-max *limit*	设置记录指令的上限，默认是 200000。如果 *limit* 是正数，这个值表示记录指令的最大值，当超过这个值以后后面的指令将会顶替掉前面记录的指令。如果 *limit* 是 0，gdb 不会删除任何记录的指令，这是无限制的情况。
show record insn-number-max	显示记录指令的上限。
set record stop-at-limit *switch*	设置指令到达记录上限之后的行为，*switch* 是： on：指令到达上限之后，gdb 停止程序并且询问是否继续，如果继续那么 gdb 将会记录新的指令并且丢弃旧指令。这个默认情况。 off：指令到达上限后，gdb 自动的删除旧指令并且记录新指令，而不询问用户。
show record stop-at-limit	显示指令达到上限之后 gdb 的行为。
set record memory-query *switch*	控制由于指令导致 gdb 不能够记录内存改变，这个时候的行为，*switch* 是： on：gdb 询问是否停止进程。 off：gdb 自动忽略对内存的影响。
show record memory-query	显示 memory-query 的设置。
info record	显示进程记录的统计量。
record delete	在回放模式中，删除接下来的执行记录，并且开始一个新的记录。

16. 检查调用栈

frame [*args*] f [*args*]	从调用栈的一层移动到另一层，并打印出你选择的调用栈。*args* 是地址或者是调用栈的序号，没有给出 *args* 打印出当前的调用栈的情况。
select-frame [*args*]	与 frame 命令相同，但是不打印出任何信息。
backtrace [full] [*n*] bt [full] [*n*] where info stack	显示整个调用栈的内容，每一行打印此层调用栈。full 表示打印出栈的局部变量。*n*>0 表示打印出 *n* 层内层调用栈，*n*<0 表示打印出 *n* 层外层调用栈。Where 和 info stack 与命令 bt 相同。
set backtrace past-main *switch*	设置 backtrace 命令是否显示用户入口点(main 函数)以外的调用栈，*switch* 是： on：显示 main 函数以外的调用栈。

	off：不显示 main 函数以外的调用栈。这是默认情况。 不给出 *switch* 与 set backtrace past-main on 相同。
show backtrace past-main	显示 backtrace past-main 的规则。
set backtrace past-entry *switch*	设置 backtrace 命令是否显示应用程序内部入口点以外的调用栈，*switch* 是： on：显示内部入口点函数以外的调用栈。 off：不显示内部入口点函数以外的调用栈。这是默认情况。 不给出 *switch* 与 set backtrace past-entry on 相同。
show backtrace past-entry	显示 backtrace past-entry 的规则。
set backtrace limit *n*	设置 backtrace 显示级数的限制，0 表示没有限制。
show backtrace limit	显示 backtrace 的显示级数。
up [*n*]	调用栈向上移动 *n* 层，*n* 默认是 1。
down [*n*] do [*n*]	调用栈向下移动 *n* 层，*n* 默认是 1。
up-silently [*n*]	调用栈向上移动 *n* 层，*n* 默认是 1。这个命令不打印任何消息。
down-silently [*n*]	调用栈向下移动 *n* 层，*n* 默认是 1。这个命令不打印任何消息。
info frame [*args*] i f [*args*]	打印相应调用栈的详细描述。
info args	打印出当前调用堆栈的参数。
info locals	打印出当前调用堆栈的局部变量。
info catch	打印出当前调用堆栈的异常处理。

17. 检查源代码

list l list *linenum* list *function* list *first*,*last* list *first*, list ,*last* list + list -	打印出当前位置的源代码，默认是 10 行。*linenum* 打印出以给出的行号为中心的 10 行代码。*function* 打印出以给出的行号为中心的 10 行代码。*first*,*last* 给出了打印代码的行号范围。*first*,表示打印从 *first* 开始的 10 行。,*last* 表示打印到 *last* 结尾的 10 行。+打印当前这 10 行代码的下一个 10 行代码。-打印出当前这 10 行代码的上一个 10 行。不给出参数则依次向下打印代码每次默认 10 行。
set linesize *count*	设置 list 命令的默认打印行数。
show linesize	显示当前 list 命令的默认打印行数。
forward-search *regexp* search *regexp*	向前搜索源代码，并打印出第一个符合 *regexp* 正则表达式的源代码。
reverse-search regexp	向后搜索源代码，并打印出第一个符合 *regexp* 正则表达式的源代码。

编辑源代码：

edit *location*	编辑从 location 开始的源代码，location 是： *number*：编辑从文件的指定行号开始的代码。 *function*：编辑从指定函数体开始的代码。 默认使用 ex 编辑器，可以通过 EDITOR 环境变量修改编辑器。

show linesize	显示当前 list 命令的默认打印行数。

添加源代码搜索目录：

directory [*dirname*] dir [*dirname*]	将 *dirname* 添加到源代码搜索目录中，多个目录可以使用目录分隔符(UNIX 上是:Windows 上是;)。如果不给出 *dirname* 那么就将搜索目录设置为默认值。
show directories	打印源代码搜索目录。
set substitute-path *from to*	定义源代码搜索目录的替换规则，多个替换规则按顺序使用。例如你把源代码/usr/src/foo.c 转移到了/mnt/cross/foo.c 则使用 set substitute-path /usr/src /mnt/cross 来定义替换规则。
unset substitute-path [*path*]	删除 *path* 指定的替换规则，*path* 要与上面命令中的 *from* 相同。不给出 *path* 则删除所有的替换规则。
show substitute-path [*path*]	显示指定的替换规则，*path* 与上面命令中的 *from* 一致。没有给出 *path* 显示所有的替换规则。

18. 源代码和机器码

info line *linespec*	打印出指定行号 *linespec* 的源代码的指令开始的地址和结束的地址。
disassemble [/r | /m] [*start*,(*end* | +*length*)]	将一段范围内的内存中的内容以机器指令的形式显示出来。/m 同时显示源代码，/r 同时显示内存中的内容。*start*,(*end* | +*length*)用来指定内存的范围。没有指定范围则显示全部进程的的地址空间。
set disassembly-flavor *instruction-set*	设置汇编指令集，*instruction-set* 是： intel：intel 指令集。 att：AT&T 指令集。 这都是针对 x86 体系结构的，att 是默认选项。
show disassembly-flavor	显示汇编指令集。
set disassemble-next-line *mode*	控制当进程停止的时候 gdb 是否显示将要执行的语句的汇编指令，*mode* 是： on：显示下面要执行的源代码的汇编指令。 auto：在不能够显示源代码的情况下显示汇编指令。 off：不显示汇编指令。
show disassemble-next-line	显示当前的 disassemble-next-line 的设置。

19. 指定位置

有很多 gdb 命令都需要制定位置，通常可以使用下面这些方式指定一个位置：

linenum	*linenum* 是源代码中的行号。
+*offset* -*offset*	相对于当前位置的偏移量。
filename:*linenum*	指定源文件中的行号。
function	指定函数体开始的位置。

filename:function	指定文件中的函数体还是的位置。
label	指定源代码中的标签。
**address*	指定程序的地址。*address* 可能是： *expression*：当前语言的有效的表达式。 *funcaddr*：当前语言的函数地址。 *'filename'::funcaddr*：指定文件中的函数地址。

20. 检查数据

检查表达式

print [*/f*] [*expr*] p [*/f*] [*expr*] inspect [*/f*] [*expr*]	用来查看数据的内容。表达式 *expr* 以格式*/f* 显示。不指定 *expr* 则显示上一个被检查的表达式的值。
set multiple-symbols *mode*	这个命令调整当遇到有歧义的表达式的时候 gdb 的行为，*mode* 是： all：当遇到有歧义的表达式的时候，gdb 选择所有的可能性。这个默认情况。 ask：gdb 使用一个可选的列表来询问用户。 cancel：gdb 报告由于歧义的表达式造成的错误。
show multiple-symbols	显示当前 gdb 对于歧义符号的处理。

print 命令的输出格式(*/f*)：

/x	将数据作为 16 进制整数进行输入。
/d	有符号整型。
/u	无符号整型。
/o	八进制整数。
/t	二进制整数。
/a	地址，包括 16 进制的地址和调用栈符号偏移量。例如： p/a 0x400508 $1 = 0x400508 <__do_global_dtors_aux+8>
/c	作为字符输出。
/f	作为浮点数输出。
/s	作为字符串输出。
/r	raw 格式输出。基于 Python 的格式打印。

检查数组

@	二进制操作符，用来指定人为的数组。例如源代码中有一个动态分配的数组：
	int* array = (int*)malloc(len * sizeof(int));
	调试的时候使用人为构造的数组可以查看所有的内容。
	p *array@len
	这样的显示结果就是假设 len == 5
	$1 = {0, 0, 0, 0, 0}
	如果你的数组是保存指针类型的数组如：
	abc_t** array = (abc_t**)malloc(len * sizeof(abc_t*));
	p *array@len 的结果是：
	$1 = {0x601070, 0x601090, 0x6010b0, 0x6010d0, 0x6010f0}

	如果你想查看每个指针指向的结构的内容(假设每个结构都含有 a，b，c 三个整型成员)： p **array@len $1 = {{a=0, b=0, c=0}, {a=0, b=0, c=0}, {a=0, b=0, c=0}, {a=0, b=0, c=0}, {a=0, b=0, c=0}}

set $i = 0
p *array*[$i++]->*a*
	设置辅助变量来查看数组内容。与上面的例子相同，你可以使用这个命令来变量整个数组：
	例如源文件中的相关代码是(忽略了异常处理代码)：
	abc_t** array = (abc_t**)malloc(len * sizeof(abc_t*));
	for(i = 0; i < len; ++i)
	{
	array[i] = (abc_t*)malloc(sizeof(abc_t));
	array[i]->a = array[i]->b = array[i]->c = i;
	}
	这样调试命令为：
	set $i = 0
	p array[$i++]->a
	<RET>
	<RET>
	每次回车之后，都显示下一个数据：
	$1 = 0
	$2 = 1
	$3 = 2
	...
	如果你想检查整个结果：
	set $i = 0
	p *array[$i++]
	<RET>
	<RET>
	输出的结果：
	$1 = {a = 0, b = 0, c = 0}
	$2 = {a = 1, b = 1, c = 1}
	$3 = {a = 2, b = 2, c = 2}
	...

检查内存

x [/*nfu*] [*addr*]	检查内存。选项： *n*：是重复的次数，默认是 1。 *f*：是显示的格式，与 print 命令的格式相同，此外还添加了 i，显示指令。x 是默认情况。 *u*：显示单元的大小： b：字节。 h：半字(两个字节)。 w：字(四个字节)，这个是默认情况。 g：大字(八个字节)。 *addr*：开始显示的地址。
find [/*un*] *start_addr*, *+len*, *val* ... find [/*un*] *start_addr*, *end_addr*, *val* ...	在指定的内存地址中搜索 *val* 以及以后的变量给出的字节序列。*u* 搜索单元的大小，与上面的 *u* 相同。*n* 是显示的配置数，默认是完全显示。

自动显示

display [/*f*] [*addr* | *expr*]	当程序停止的时候，自动显示变量或者地址的内容。显示的格式使用 print 和 x 的格式。*addr* 是显示的地址，*expr* 是显示的表达式。不指定格式和地址以及表

	达式显示当前自动显示列表中的变量和地址的值。
undisplay *dnums* delete display *dnums*	从自动显示列表中删除 *dnums* 指定的自动显示单元。
disable display *dnums*	关闭自动显示单元 *dnums*。
enable display *dnums*	打开自动显示单元 *dnums*。
info display	显示自动显示列表的情况。

21. 显示设置

set print address *switch*	在显示的时候打印内存地址，*switch* 是： on：打印出相关的内存地址。 off：只显示内容不打印地址。 不给出 *switch* 与 set print address on 是一致的。
show print address	显示当前的 print address 的设置。
set print symbol-filename *switch*	控制 gdb 在显示符号的时候是否显示符号所在的源文件名以及行号，switch 是： on：显示相关的文件名和行号。 off：不显示相关的文件名和行号。这是默认项。
show print symbol-filename	显示当前的 print symbol-filename 的设置。
set print max-symbolic-offset *max-offset*	设置打印符号的最大地址偏移。0 表示没有限制，这是默认情况。
show print max-symbolic-offset	显示打印符号地址的最大偏移。
set print array *switch*	控制显示数组的方式，*switch* 是： on：以更可读的方式显示数组。 off：以普通的方式显示数组。这是默认项。 没有给出 *switch* 与 set print array on 是一致的。
show print array	显示 gdb 显示数组的方式。
set print array-indexes *switch*	控制在显示数组的同时是否显示索引，*switch* 是： on：显示索引。 off：不显示索引，这是默认选项。 没有给出 *switch* 与 set print array-indexes on 是一致的。
show print array-indexes	显示 gdb 显示数组的时候是否显示索引。
set print elements *number-of-elements*	设置 gdb 显示数组元素的限制，当 gdb 显示数组元素达到限制的时候，它会停止显示，后面用省略符号表示。gdb 启动的时候的值是 200。0 表示没有限制。
show print elements	显示 gdb 显示数组元素的个数限制。
set print frame-arguments *value*	控制 gdb 显示调用栈(函数调用)的时候任何显示参数，*value* 是： all：显示所有参数的值。 scalars：仅仅显示数值型的参数，复杂的参数如数组，结构，联合等这样的参数都使用...来代替。这是默认情况。 none：不显示任何参数，完全使用...代替所有的参数。
show print frame-arguments	显示 gdb 如何显示调用栈的参数。
set print repeats *n*	设置抑制显示重复数组元素的门槛。默认是 10。*n* 为 0 表示没

	有限制，所有的重复数组元素都会被显示出来。
show print repeats	显示抑制显示重复数组元素的门槛值。
set print null-stop *switch*	控制 gdb 在显示字符数组的时候遇到 NULL 是否停止显示。 *switch* 是： on：停止显示。 off：不停止显示。这是默认情况。 没有给出 *switch* 与 set print null-stop on 一致。
show print null-stop	显示当前 gdb 在显示字符数组是遇到 NULL 的行为。
set print pretty *switch*	控制 gdb 以可读的方式打印结构。*switch* 是： on：以方便阅读的方式打印结构。 off：以紧凑的方式打印结构。这是默认选项。
show print pretty	显示 gdb 打印结构的方式。
set print sevenbit-strings *switch*	设置 7 位字符的打印，*switch* 是： on：以 7 位的方式打印字符，如果超过 7 位则以\nnn 的方式打印出来。 off：打印 8 位字符。这个是默认情况。
show print sevenbit-strings	显示 gdb 是否显示 7 位字符。
set print union *switch*	设置 gdb 是否显示在其他结构体或者其他联合中的联合体， *switch* 是： on：显示。这是默认设置。 off：不显示，使用{...}代替。
show print union	显示 gdb 是否显示在其他结构或者联合中的联合体。

还有一些关于显示 C++相关对象的命令，以后有机会使用 C++的时候在整理。

22. 历史记录和便利变量

$ $*n* $$ $$*n*	$*n* 引用历史记录号码为 *n* 的历史记录。$引用的是刚刚使用的历史记录值。$$*n* 引用的是从结尾开始的第 *n* 个历史记录。$$2 和$$1 都与$$相同，$$0 等于$。下面是历史的一些有用的使用方法： 1)当你刚刚 p 命令显示一个指针。接下来你想显示指针指向的结构，可以使用命令： p *$ 2)如果你有一个结构，用其中的一个成员 next 指向下一个，这样形成一个链表。你可以使用命令： p *$.next 来打印下一个，然后可以通过<RET>来按链表顺序打印整个链表。
show values [*n* | +]	打印出 10 个历史记录，*n* 表示打印以 *n* 为中心的 10 个历史记录，+表示接着上面打印的历史记录，再次打印 10 个历史记录。没有参数表示打印最近的 10 个历史记录。
$*name*	gdb 提供的便利变量，*name* 是变量的名字，它不能够是数字也不能够保留字(寄存器名等)。使用命令 set $*name* = 1234

	这样的命令来设置便利变量，便利变量没有类型，任何类型的值都可以赋值给它，在赋值之前它的值是 void。 便利变量的一个使用就是作为索引来打印数组中的数据： set $i = 0 p bar[$i++] 然后按<RET>就可以打印整个数组中的值。
$_	这是保留的便利变量，被 gdb 自动创建，用来保存 x 命令检查的最后的地址。$_是 void*类型。
$__	这是保留便利变量，被 gdb 自动创建，用来保存 x 命令检查的最后的地址中的值。
$_exitcode	这是保留便利变量，被 gdb 自动创建，用来保存当调试结束的时候程序的退出代码。
$_sdata	这是保留便利变量，被 gdb 自动创建，用来保存额外收集的静态跟踪点数据。
$_siginfo	这是保留便利变量，被 gdb 自动创建，用来保存额外的信号信息。
$_tlb	这是保留便利变量，当调试的程序以本地程序运行在 Windows 上或者连接到支持 qGetTIBAddr 的 gdbserver 上的时候，这个变量就自动创建。这变量保存进程信息块的地址。
show convenience show conv	打印到目前为止正在使用的便利变量。
init-if-undefined $*variable* = *expression*	如果变量$*variable* 没有被初始化，那么将它赋值为 *expression*。这在用户定义的命令中很有用。
help function	打印所有的便利函数。

23. 寄存器以及硬件相关信息

$*register*	通过$来引用寄存器，*register* 是寄存器的名字，可以通过 info registers 来查看机器相关的寄存器的名字和值。
info registers [*regname*]	打印名字为 *regname* 的寄存器的值，没有提供 *regname* 就打印所有的寄存器的名字和值，除了浮点寄存器和向量寄存器。
info all-registers	打印所有的寄存器的名字和值，包括浮点寄存器和向量寄存器。
$pc	标准寄存器名字(如果不与结构的寄存器规范记法冲突)，表示程序计数寄存器。
$sp	标准寄存器名字(如果不与结构的寄存器规范记法冲突)，表示堆栈指针寄存器。
$fp	标准寄存器名字(如果不与结构的寄存器规范记法冲突)，表示当前调用栈指针寄存器。
$ps	标准寄存器名字(如果不与结构的寄存器规范记法冲突)，表示处理器状态寄存器。
info float	打印浮点硬件相关的信息，精确的内容和布局依赖于浮点芯片。
info vector	打印向量单元的信息，精确的内容和布局依赖于向量硬件。

24. 操作系统相关信息

info udot	打印出内核为正准备调试的程序维护的 struct user 结构的内容。
info auxv	打印出下属进程的辅助向量，这个下属进程可以是运行的进程也可以是 core 文件。
info os	打印出目标机器上的 OS 的可用信息。
info os processes	打印出目标机器上的进程列表。对于每一个进程打印出进程的 id，用户名和相应的命令。

25. 内存区域

mem *lower upper attributes ...*	定义一段内存区域，*lower* 和 *upper* 指定了内存区域的范围，*upper*==0 是特殊情况，它表明使用内存的最大上限(在 16 位目标上上 0xffff，在 32 位目标上是 0xffffffff，等等)。*attributes* 是属性： ro：只读。 wo：只写。 rw：读写，这是默认情况。 8：使用 8 位内存访问。 16：使用 16 位内存访问。 32：使用 32 位内存访问。 64：使用 64 位内存访问。 catch：允许缓存目标内存。 nocatch：不允许缓存目标内存，这是默认属性。
mem auto	丢弃用户对于内存区域的改变并且使用目标提供的内存支持。
delete mem *nums ...*	从 gdb 监视的内存区域列表中删除内存区域 *nums*。
disable mem *nums ...*	在 gdb 监视的内存区域列表中禁用内存区域 *nums*。
dnable mem *nums ...*	在 gdb 监视的内存区域列表中启用内存区域 *nums*。
info mem	打印所有定义的内存区域列表。
set mem inassessible-by-default *switch*	设置 gdb 对于没有列在内存区域列表中的内存的访问情况，switch 是： on：禁止访问没有列举在内存区域列表中的内存区域，这是默认情况。 off：将没有列举在内存区域列表中的内存作为 RAM 使用。
show mem inasscessible-by-default	显示 db 对于没有列在内存区域列表中的内存的访问情况。

26. 内存和文件之间的数据交换

dump [*format*] memory *filename start_addr end_addr* dump [*format*] value *filename expr*	将内存或者数据中的内容拷贝到文件 *filename* 中。*format* 是生成的文件的格式： binary：原始的二进制格式。 ihex：Intel hex 格式。 srec：Motolora S-record 格式。 tekhex：Tektronix Hex 格式。 没有给出 *format* 使用 binary 格式。*start_addr* 和 *end_addr* 是内存地址。*expr* 是表达式。
append [binary] memory *filename start_addr end_addr*	将内存地址 *start_addr* 和 *end_addr* 中的内容或者是

append [binary] value *filename expr*	变量 *expr* 的值添加到文件 *filename* 末尾。这个命令只能使用原始的二进制格式。
restore *filename* [binary] *bias start end*	将文件 *filename* 中的内容重新读取到内存中，restore 可以识别任何二进制格式，但是要使用原始的二进制格式必须使用 binary 关键字。*bias* 如果不是 0 那么将它作为文件的开始位置。*start* 和 *end* 如果不是 0，那么它们指定的范围都是以 *bias* 为标准的。
generate-core-file [*file*] gcore [*file*]	产生当前进程的 core 文件。可选项 *file* 指定了输出的文件名。如果没有指定，输出的文件名是 core.*pid*。

27. 字符集

set charset *charset*	将主机字符集和目标字符集设置成 *charset*。使用 set charset <TAB><TAB> 将会列出可以使用的字符集。
show charset	显示当前的主机字符集和目标字符集的设置。
set host-charset *charset*	设置当前的主机字符集。
show host-charset	显示当前的主机字符集。
set target-charset *charset*	设置目标字符集。
show target-charset	显示目标字符集。
set target-wide-charset *charset*	设置目标宽字符集。影响 wchar_t 类型。
show target-wide-charset	显示目标宽字符集。

28. 调试远程目标

set remotecache *switch*	这个命令不做任何事情，只是为了提供向老版本的兼容性。
show remotecache	显示过时的 remotecache 标志。
set stack-cache *switch*	堆栈访问缓存开关，switch 是： on：启用堆栈访问的缓存。默认情况 off：停用堆栈访问的缓存。
show stack-cache	显示对于堆栈访问的缓存情况。
info dcache [*linenum*]	打印关于数据缓存的实现信息。如果指定了 *linenum* 那么以 16 进制的方式显示缓存内容。
set romote *medium*	设置与远程目标链接的媒介，*medium* 是： *serial-device*：使用串行线连接远程目标。 [tcp:]*host*:*port*：使用 TCP 链接远程目标。 udp:*host*:*port*：使用 UDP 链接远程目标。 | *command*：在后台运行 *command* 并且与远程目标链接。
detach	释放远程目标，远程进程结束，释放之后 gdb 可以连接另一个远程目标了。
disconnect	与 detach 命令类似，但是 disconnect 之后远程进程并不结束，再次链接之后

	仍然从停止的位置开始。
monitor *cmd*	直接给远程监视器发送命令 *cmd*。
remote put *hostfile targetfile*	将主机文件系统中的文件拷贝到目标系统中。
remote get *targetfile hostfile*	将目标系统中的文件下载到主机系统中。
remote delete *targetfile*	从目标系统中删除文件。

29. C 预处理器宏

要使用 gcc 的-g3 选项才能够在编译后的可执行文件中包含预处理宏的信息，这样 gdb 的预处理宏命令才能有效。否则这些命令不起作用。

macro expand *expression* macro exp *expression*	显示包含宏调用的表达式 *expression* 进行宏扩展之后的结构，只是显示不会实际运行，表达式不必是合法的表达式。
macro expand-once *expression* macro exp1 *expression*	显示包含宏调用的表达式 *expression* 进行宏扩展之后的结构，只是显示不会实际运行，表达式不必是合法的表达式。这个命令与上面命令的不同之处在于它只扩展最外层的宏，宏定义内部调用的宏不会在扩展。
info macro *macro*	显示宏 *macro* 的定义以及定义的文件或者命令行。
macro define *macro replacement-list* macro define *macro*(*arglist*) *replacement-list*	这个命令定义一个对象宏或者是函数宏，这些宏只能应用于 gdb 调试过程中。
macro undef *macro*	删除由 macro define 定义的宏。它不能够删除代码或者命令行中定义的宏。
macro list	列出所有的 macro define 定义的宏。

30. 跟踪点

跟踪点是用来调试那些实时性质的程序，这些程序调试器的延迟可能会使程序得不到想要的结果，即使正确的代码也会产生错误，甚至失败。跟踪点就是在程序运行的过程中得到数据并且在运行之后检查这些记录的信息来查看问题所在。跟踪点也是特殊的断点，可以用标准的断点命令来管理跟踪点，但是有些命令是不适用的。

创建和删除跟踪点

trace *location* [if *cond*] tr *location* [if *cond*]	设置跟踪点。*cond* 表示跟踪点收集数据的条件。
ftrace *location* [if *cond*]	设置快速跟踪点，但并不是所有的平台都支持。
strace *location* [if *cond*]	设置静态跟踪点，但并不是所有平台都支持。
delete tracepoint [*num*] del tr [*num*]	删除一个或者多个跟踪点，不给出 *num* 则删除所有的跟踪点。
info tracepoints [*num*]	列举跟踪点 *num* 的信息，如果没有指定 *num* 那么列举到目前为止的全部跟踪点的信息。
info static-tracepoint-markers	列举静态跟踪点标记的信息。
save tracepoints *filename* save-tracepoints *filename*	将当前定义的跟踪点保存到文件 *filename* 中。

禁用和启用跟踪点

disable tracepoint [*num*]	禁用跟踪点 *num*，如果没有给出 *num*，则禁用所有的跟踪点。

enable tracepoint [*num*]	启用跟踪点 *num*，如果没有给出 *num*，则启用所有的跟踪点。
passcount [*n* [*num*]]	设置跟踪点的通过计数，这是自动停止一个跟踪实验(而不是停止程序)的方法，在第 *n* 次通过跟踪点 *num* 的时候自动的停止收集数据。如果 *num* 没有指定，那么命令设置的是最近定义的跟踪点。如果没有参数给出那么直到用户显示的停止程序，跟踪点才停止。

跟踪状态变量

跟踪状态变量是由目标端代码创建和管理的，并且保存在目标端，这个变量通常是 64 位的有符号整数，它与 gdb 便利变量使用同样的名字空间$，所以它们之间会有冲突。

tvariable $*name* [= *expression*]	这个命令创建一个新的跟踪状态变量$*name*，*expression* 为可选的初始化表达式。如果$*name* 是一个已经存在的跟踪状态变量，那么这个命令将使用 *expression* 覆盖原有的值。默认值是 0。
info tvariables	显示所有的目标状态变量和它们的值，以及当前运行的跟踪实验的目标状态变量。
delete tvariable [$*name* ...]	删除给出的跟踪状态变量，没有给出参数则删除所有的跟踪状态变量。

为跟踪点设置动作

actions [*num*]	为跟踪点 *num* 设置动作，这些动作都是在跟踪点被触发的时候执行的，如果没有指定 *num* 则为刚刚设置的跟踪点设置动作。这个命令执行之后会出现命令提示符例如： (gdb) trace foo (gdb) actions > collect $regs > collect $args > end (gdb) 你可以在命令提示符中设置命令，并使用 end 结束设置。下面的几个命令都是可以应用在 actions 名利中的。
collect expr1, expr2, ...	收集表达式给出的变量的值，这个命令接受以逗号分隔的有效表达式，此外它还接受下列特殊命令： $regs：搜集所有寄存器。 $args：搜集所有函数参数。 $locals：搜集所有局部变量。 $_sdata：搜集静态跟踪点标记的数据。只应用于静态跟踪点。
teval expr1, expr2, ...	计算表达式的值。这个计算结果会被丢弃，所以它的主要用处是为了给跟踪状态变量赋值。
while-stepping *n* stepping *n* ws *n*	在跟踪点后面执行 *n* 测单步跟踪，例如： > while-stepping 12 > collect $regs, myglobal > end > 使用 end 命令结束 while-stepping。
set default-collect expr1, expr2, ...	设置触发每个跟踪点时默认搜集的变量表达式。
show default-collect	显示在触发每个跟踪点时默认搜集的变量表达式。

跟踪实验

tstart	开始跟踪实验并收集数据，这个命令有一个副作用就是丢弃所有先前运行的跟踪实验搜集在缓冲区中的数据。

tstop	结束跟踪实验并且停止搜集数据。
tstatus	这个命令显示当前跟踪数据收集的情况。

跟踪行为

set disconnected-tracing *switch*	当 gdb 与远程目标断开链接(主要是处理未预期的结果)，选择是否继续运行。*switch* 是： on：继续运行。 off：停止运行。
show disconnected-tracing	显示 gdb 与远程目标断开连接的时候，是否继续运行。
set circular-trace-buffer *switch*	选择是否使用循环缓冲区，switch 是： on：使用循环缓冲区。 off：使用线性缓冲区。
show circular-trace-buffer	显示当前的缓冲区类型。

使用跟踪数据

tfind [*mode*]	在缓存区中查找跟踪数据的快照。*mode* 是： start：在缓冲区中查找第一个快照。与 tfine 0 相同。 none： end：停止调试跟踪数据快照。 *num*：查找号码为 *num* 的快照，号码从 0 开始。 没有给出参数：表示查找下一个快照。 -：查找上一个快照。 tracepoint *num*：查找与跟踪点 *num* 相关的下一个快照。 pc *addr*：查找与 *addr* 相关联的 pc 的下一个快照。 outside *addr1*, *addr2*：查找在限定范围(*addr1*, *addr2*)之外的 pc 的下一个快照。 range *addr1*, *addr2*：查找在限定范围(*addr1*, *addr2*)之内的 pc 的下一个快照。 line [*file*:]*n*：查找与指定源代码行 *n* 相关的下一个快照。
tdump	显示当前跟踪快照搜集的所有的数据。

跟踪点便利变量

(int)$trace_frame	当前跟踪快照号，-1 表示没有被选择的快照。
(int)$tracepoint	当前跟踪快照的跟踪点。
(int)$trace_line	当前跟踪快照的行号。
(char[])$trace_file	当前跟踪快照的源文件。
(char[])$trace_func	包含$tracepoint 的函数名字。

跟踪文件

tsave [-r] *filename*	将跟踪数据保存在文件 *filename* 中，默认的情况下 *filename* 是主机的文件系统。如果目标支持-r 选项直接将数据保存在目标的文件系统上。
target tfile *filename*	使用跟踪数据文件 *filename*。

31. 调试大程序镜像

32. 调试多语言代码

33. 检查符号表

set case-sensitive *switch*	设置 gdb 在检查符号表的时候的大小写敏感情况。*switch* 是： on：大小写敏感。 off：大小写不敏感。 auto：设置为自动，这个情况根据语言来设定。
show case-sensitive	显示 gdb 检查符号表时的大小写敏感设置。
info address *symbol*	描述符好 *symbol* 存储的位置。对于寄存器变量它显示的是存储的寄存器，对于非寄存器变量它显示的是存储的调用栈的偏移量。
info symbol *addr*	打印出地址 *addr* 存储的符好的名字。
whatis [*arg*]	打印出 *arg* 的数据类型。没有指定参数则打印$的数据类型。

ptype [*arg*]	与 whatis 命令类似但是它打印出的是数据类型的详细信息。例如结构
	typedef struct _tagabc_t
	{
	int a;
	int b;
	}abc_t;
	abc_t v;
	(gdb) whatis v
	type = struct _tagabc_t
	(gdb) ptype v
	type =struct _tagabc_t {
	int a;
	int b;
	}

info types *regexp* info types	打印与正则表达式 regexp 匹配的所有类型的信息，没有参数打印程序中支持所有类型的信息。
info scope *location*	打印出所有与位置 *location* 相关的局部变量和参数的信息。
info source	打印出当前源代码文件的信息。
info sources	打印出当前调试程序的所有的源代码文件的信息。
info functions [*regexp*]	打印出名字与 *regexp* 匹配的函数的名字和数据类型，没有给出 *regexp* 则打印出全部函数。
info variables [*regexp*]	打印名字与 *regexp* 匹配的变量的名字和数据类型，没有给出 *regexp* 则打印出全部变量。
set opaque-type-resolution *switch*	控制 gdb 是否解释在一个文件中使用的一种指向结构或者联合的指针，但是结构或者联合的完整定义在另一个文件中，switch 是： on：gdb 自动解释这种指针。 off：gdb 不解释这种指针。
show opaque-type-resolution	显示 gdb 是否自动解释 opaque 类型。

34. 在调试的过程中修改程序的执行

为变量赋值

set variable *var* = *value* set var *var* = *value*	将变量 *var* 赋值为 *value*。
set {*type*}*addr* = *value*	将地址 *addr* 按照 *type* 类型赋值成 *value*。例如 set {int}0x80307 = 34 是将地址 0x80307 解释成 int 类型并赋值为 34。

跳转

jump *linespec* jump *location*	在 *linespec* 行或者位置 *location* 继续运行代码。这个效果与 set $pc = 0x485 相同，但是 set $pc 使用的全局的地址，jump 只限于程序内部地址。

给程序发送信号

signal *signal*	当程序停止的时候使用这个命令重新运行程序并立即给程序发送信号 *signal*，可以是数字也可以是信号名。例如 signal 2 和 signal SIGINT。 signal 0 表示运行程序但是不发送信号。

从函数中返回

return [*expression*]	从函数中返回，*expression* 最为返回值。这个命令与 finish 不同，finish 事将函数执行结束，这个命令是在当前函数的当前位置退出。

调用程序函数

print *expr*	计算表达式 *expr* 并显示返回值。*expr* 可以含有对于被调试程序中的函数的调用。
call *expr*	计算表达式 *expr* 并不显示 void 返回值。E*xpr* 可以含有对于被调试程序中的函数的调用。
set unwindonsignal *switch*	设置当被调试程序受到信号后，gdb 是否退出当前调用堆栈，switch 是： on：gdb 退回到产生信号的函数调用之前的调用栈。 off：gdb 不进行任何调用栈操作。这是默认情况。
show unwindonsignal	显示当前的 unwindonsignal 的设置。

给程序打补丁

set write *switch*	设置 gdb 对于程序的读写方式，switch 是： on：gdb 以读写的方式打开程序或者是 core 文件。 off：gdb 以只读的方式打开程序或者是 core 文件。
show write	显示 gdb 对于程序的读写方式。

35. gdb 文件

file [-readnow] [*filename*]	读取文件 *filename* 作为调试的程序，读取符号表和内存内容。如果没有给出 *filename* 则 gdb 丢弃可执行文件和符号表的任何信息。-readnow 是控制是否立即读取符号表。
exec-file [*filename*] target exec [*filename*]	指定 *filename* 作为可被调试的执行程序，而不是符

	号表。忽略 *filename* 表示丢弃当前的可执行程序。
symbol-file [-readnow] [*filename*]	从文件 *filename* 中读取符号表，忽略 *filename* 则表示丢弃当前调试程序的符号表。-readnow 表示立即读取符号表。
core-file [*filename*] target core [*filename*] core	指定 core 文件。
add-symbol-file *filename* [-s*section*] *address* [-readnow]	从 *filename* 中读取额外的符号表信息。通过 -s*section* 给出显示的段名字和地址。
add-symbol-file-from-memory *address*	从动态链接对象的镜像地址 *address* 中读取符号表。
section *section addr*	这个命令改变可执行文件的短的基本地址。
info files info target	打印当前的目标，包括可执行文件和 core 文件的名字和读取的符号表文件的名字。
set trust-readonly-section switch	设置 gdb 只读段的性质，switch： on：只读段中的内容不会改变。 off：只读段中的内容可能会改变。这是默认选项。
show trust-readonly-section	显示 gdb 只读段设置。
set auto-solib-add *mode*	控制自动读取共享库符号表，*mode* 是： on：当进程运行的时候，或者 gdb 链接到一个正在运行的进程的时候，自动读取共享库中的符号表。这个是默认项。 off：gdb 不会自动读取符号表，必须使用 sharedlibrary 来手工读取。
show auto-solib-add	显示现在的共享库符号表读取模式。
info share *regexp* info sharedlibrary *regexp*	打印与 *regexp* 匹配的当前加载的共享库的名字。忽略 *regexp* 则打印所有的加载的共享库的名字。
share *regexp* sharedlibrary *regexp*	加载与 *regexp* 匹配的共享库的符号表。忽略 *regexp* 则加载所有的共享库。
nosharedlibrary	卸载所有的共享库，丢弃所有来自于共享库的符号表。
set stop-on-solib-events *switch*	控制 gdb 对于共享库事件的相应，常见的共享库事件就是加载和卸载，*switch* 是： on：当共享库事件发生时 gdb 停止。 off：gdb 忽略共享库事件。
show stop-on-solib-events	显示 gdb 对于共享库实现的相应方式。
set sysroot *path*	将路径 *path* 作为系统的跟目录。任何共享库前面都会加上 *path* 路径。如果 *path* 中带有 remote:串，gdb 将会从远端系统上获得目标库。
show sysroot	显示当前的共享库前缀。
set solib-search-path *path*	设置共享库搜索路径，*path* 是冒号分隔的路径，solib-search-path 用在 sysroot 之后。
show solib-search-path	显示当前的共享库搜索路径。
set target-file-system-kind *kind*	设置目标文件系统类型。*kind* 是： unix：目标系统类型为 unix 类型的文件系统。

	dos-based：目标系统类型为基于 DOS 的文件系统。 auto：让 gdb 使用目标系统类型来推断文件系统。 这是默认选项。
set debug-file-directory *directories*	设置单独调试信息文件的搜索目录。
show debug-file-directory	显示单独调试信息文件的搜索目录。
save gdb-index *directory*	为 gdb 知道的每一个符号表文件建立一个索引文件，名字是符号表文件后面加.gdb-index，并且保存在 *directory* 中。
set data-directory *directory*	设置 gdb 辅助数据目录。
show data-directory	显示 gdb 数据数据目录。

36. 指定调试目标环境

set architecture *arch*	设置目标体系结构 *arch*。*Arch* 的值可以是 auto 用来表示支持的结构中的一个。
show architecture	显示当前的目标体系结构。
target *type parameters*	设置与 gdb 主机环境链接的目标机器或者处理器。
help target [*name*]	显示所有可用的目标。给出参数 *name* 则显示指定的目标以及需要的参数。
set gnutarget *args*	用来指定 BFD 库记录的文件格式。
show gnutarget	显示 gnutarget 记录的文件格式。
target remote *medium*	指定远程协议媒体。
target sim [*simargs*]...	内建的 cpu 模拟器，gdb 包含多种体系结构的模拟器。
target nrom *dev*	NetROM ROM 仿真器。
set hash *switch*	这个命令控制当下载到远端模拟器的时候是否显示 hash 标记#。
show hash	显示当前的 hash 标记的状态。
set debug monitor *switch*	开启或者禁止显示 gdb 和远端监视器的链接消息。
show debug monitor	显示当前的 debug monitor 的设置。
load *filename*	通过下载或者动态链接，是 *filename* 在远端系统上可以调试。
set endian *mode*	设置目标字节序，*mode* 是： big：将目标字节序设置成 big-endian。 little：将目标字节序设置成 little-endian。 auto：根据可执行文件设置目标字节序。
show endian	显示目标字节序。