GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。如果你是在 UNIX 平台下做软件，你会发现 GDB 这个调试工具有比 VC、BCB 的图形化调试器更强大的功能。同时 GDB 也具有例如 ddd 这样的图形化的调试端。

一般来说，GDB 主要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的运行程序。

(2)可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）

(3)当程序被停住时，可以检查此时你的程序中所发生的事。

(4)动态的改变你程序的执行环境。

兴趣是最好的老师，这里先整理总结一下在调试的过程中经常遇到的问题。带着这些问题进行学习和实践可以有助于加深印象

(1)如何打印变量的值？(print var)

(2)如何打印变量的地址？(print &var)

(3)如何打印地址的数据值？(print *address)

(4)如何查看当前运行的文件和行？(backtrace)

(5)如何查看指定文件的代码？(list file:N)

(6)如何立即执行完当前的函数，但是并不是执行完整个应用程序？(finish)

(7)如果程序是多文件的，怎样定位到指定文件的指定行或者函数？(list file:N)

(8)如果循环次数很多，如何执行完当前的循环？(until)

(9)多线程如何调试？(???)

[举例]

==========

*启动 gdb

$gdb

这样可以和 gdb 进行交互了。

*启动 gdb，并且分屏显示源代码：

$gdb -tui

这样,使用了'-tui'选项，启动可以直接将屏幕分成两个部分，上面显示源代码，比用 list 方便多了。这时候使用上下方向键可以查看源代码,想要命令行使用上下键就用[Ctrl]n 和[Ctrl]p.

*启动 gdb 调试指定程序 app：

$gdb app

这样就在启动 gdb 之后直接载入了 app 可执行程序，需要注意的是，载入的 app 程序必须在编译的时候有 gdb 调试选项，例如'gcc -g app app.c',注意，如果修改了程序的源代码，但是没有编译，那么在 gdb 中显示的会是改动后的源代码，但是运行的是改动前的程序，这样会导致跟踪错乱的。

*启动程序之后，再用 gdb 调试：

$gdb <program> <PID>

这里，<program>是程序的可执行文件名，<PID>是要调试程序的 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。

*启动程序之后，再启动 gdb 调试：

$gdb <PID>

这里，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,<PID>是要调试程序的 PID.这样 gdb 就附加到程序上了，但是现在还没法查看源代码,用 file 命令指明可执行文件就可以显示源代码了。

**启动 gdb 之后的交互命令：

交互命令支持[Tab]补全。

*显示帮助信息：

(gdb) help

*载入指定的程序：

(gdb) file app

这样在 gdb 中载入想要调试的可执行程序 app。如果刚开始运行 gdb 而不是用 gdb app 启动的话可以这样载入 app 程序，当然编译 app 的时候要加入-g 调试选项。

*重新运行调试的程序：

(gdb) run

要想运行准备调试的程序，可使用 run 命令，在它后面可以跟随发给该程序的任何参数，包括标准输入和标准输出说明符(<和>)和 shell 通配符（*、？、[、]）在内。

*修改发送给程序的参数：

(gdb) set args no

这里，假设我使用"r yes"设置程序启动参数为 yes，那么这里的 set args 会设置参数 argv[1]为 no。

*显示缺省的参数列表：

(gdb) show args

*列出指定区域(n1 到 n2 之间)的代码：

(gdb) list n1 n2

这样,list 可以简写为 l,将会显示 n1 行和 n2 行之间的代码，如果使用-tui 启动 gdb，将会在相应的位置显示。如果没有 n1 和 n2 参数，那么就会默认显示当前行和之后的 10 行，再执行又下滚 10 行。另外，list 还可以接函数名。

一般来说在 list 后面可以跟以下这们的参数：

<linenum> 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。

*执行下一步：

(gdb) next

这样，执行一行代码，如果是函数也会跳过函数。这个命令可以简化为 n.

*执行 N 次下一步：

(gdb) next N

*执行上次执行的命令：

(gdb) [Enter]

这里，直接输入回车就会执行上次的命令了。

*单步进入：

(gdb) step

这样，也会执行一行代码，不过如果遇到函数的话就会进入函数的内部，再一行一行的执行。

*执行完当前函数返回到调用它的函数：

(gdb) finish

这里，运行程序，直到当前函数运行完毕返回再停止。例如进入的单步执行如果已经进入了某函数，而想退出该函数返回到它的调用函数中，可使用命令 finish.

*指定程序直到退出当前循环体：

(gdb) until

或(gdb) u

这里，发现需要把光标停止在循环的头部，然后输入 u 这样就自动执行全部的循环了。

*跳转执行程序到第 5 行：

(gdb) jump 5

这里，可以简写为"j 5"需要注意的是，跳转到第 5 行执行完毕之后，如果后面没有断点则继续执行，而并不是停在那里了。

另外，跳转不会改变当前的堆栈内容，所以跳到别的函数中就会有奇怪的现象，因此最好跳转在一个函数内部进行,跳转的参数也可以是程序代码行的地址,函数名等等类似 list。

*强制返回当前函数:

(gdb) return

这样，将会忽略当前函数还没有执行完毕的语句，强制返回。return 后面可以接一个表达式，表达式的返回值就是函数的返回值。

*强制调用函数：

(gdb) call <expr>

这里,<expr>可以是一个函数，这样就会返回函数的返回值，如果函数的返回类型是 void 那么就不会打印函数的返回值,但是实践发现，函数运行过程中的打印语句还是没有被打印出来。

*强制调用函数 2：

(gdb) print <expr>

这里，print 和 call 的功能类似，不同的是，如果函数的返回值是 void 那么 call 不会打印返回值，但是 print 还是会打印出函数的返回值并且存放到历史记录中。

*在当前的文件中某一行（假设为 6）设定断点：

(gdb) break 6

*设置条件断点：

(gdb) break 46 if testsize==100

这里，如果 testsize==100 就在 46 行处断点。

*检测表达式变化则停住：

(gdb) watch i != 10

这里，i != 10 这个表达式一旦变化，则停住。watch <expr> 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(也是一种断点)。

*在当前的文件中为某一函数(假设为 func)处设定断点：

(gdb) break func

*给指定文件（fileName）的某个行（N）处设置断点：

(gdb) break fileName:N

这里，给某文件中的函数设置断点是同理的。

*显示当前 gdb 断点信息：

(gdb) info breakpoints

这里，可以简写为 info break.会显示当前所有的断点，断点号，断点位置等等。

*删除 N 号断点：

(gdb) delete N

*删除所有断点：

(gdb) delete

*清除行 N 上面的所有断点：

(gdb) clear N

*继续运行程序直接运行到下一个断点：

(gdb) continue

这里，如果没有断点就一直运行。

*显示当前调用函数堆栈中的函数：

(gdb) backtrace

命令产生一张列表，包含着从最近的过程开始的所有有效过程和调用这些过程的参数。当然，这里也会显示出当前运行到了哪里(文件，行)。

*查看当前调试程序的语言环境：

(gdb) show language

这里，如果 gdb 不能识别你所调试的程序，那么默认是 c 语言。

*查看当前函数的程序语言：

(gdb) info frame

*显示当前的调试源文件：

(gdb) info source

这样会显示当前所在的源代码文件信息,例如文件名称，程序语言等。

*手动设置当前的程序语言为 c++:

(gdb) set language c++

这里，如果 gdb 没有检测出你的程序语言，你可以这样设置。

*查看可以设置的程序语言：

(gdb) set language

这里，使用没有参数的 set language 可以查看 gdb 中可以设置的程序语言。

*终止一个正在调试的程序：

(gdb) kill

这里，输入 kill 就会终止正在调试的程序了。

*print 显示变量(var)值：

(gdb) print var

这里，print 可以简写为 p,print 是 gdb 的一个功能很强的命令，利用它可以显示被调试的语言中任何有效的表达式。表达式除了包含你程序中的变量外，还可以包含函数调用,复杂数据结构和历史等等。

*用 16 进制显示(var)值：

(gdb) print /x var

这里可以知道，print 可以指定显示的格式，这里用'/x'表示 16 进制的格式。

可以支持的变量显示格式有：

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。

*如果 a 是一个数组，10 个元素，如果要显示则：

(gdb) print *a@10

这样，会显示 10 个元素，无论 a 是 double 或者是 int 的都会正确地显示 10 个元素。

*修改运行时候的变量值：

(gdb) print x=4

这里，x=4 是 C/C++的语法，意为把变量 x 值改为 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。

*显示一个变量 var 的类型：

(gdb) whatis var

*以更详细的方式显示变量 var 的类型：

(gdb) ptype var

这里，会打印出 var 的结构定义。

**

[其他]

==========

*在 Qt4.x 环境中打印 QString msg;的 msg 变量：

步骤如下：

1)定义一个宏 printqstring

```
define printqstring
   printf "(QString)0x%x (length=%i): \"",&$arg0,$arg0.d->size
   set $i=0
   while $i < $arg0.d->size
      set $c=$arg0.d->data[$i++]
      if $c < 32 || $c > 127
            printf "\\u0x%04x", $c
      else
            printf "%c", (char)$c
      end
   end
   printf "\"\n"
```

end

2)(gdb) printqstring msg

这里，这个宏可以在 gdb 中直接定义，据说也可以写到$HOME/.gdbinit, 这样每次启动自动加载。

*调试同时指明生成 core 文件：

$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。当程序非法崩溃的时候会产生一个 core 文件，然后使用这个命令，会直接定位到发生程序崩溃的位置。注意：有时需要设置系统命令“ulimit -c unlimited”才能产生 core 文件。

*查看当前调试程序的语言环境：

(gdb) show language

这里，如果gdb不能识别你所调试的程序，那么默认是c语言。

*查看当前函数的程序语言：

(gdb) info frame

*显示当前的调试源文件：

(gdb) info source

这样会显示当前所在的源代码文件信息,例如文件名称，程序语言等。