GDB命令

- 显示帮助信息：
  (gdb) help
  后面加命令可查看命令的帮助，如`(gdb) help x`：
  `(gdb) help <func>`
- 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。
- 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（缩写为r）**，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。
- **examine**：（简写为x）
  examine命令查看内存地址处的值。x命令的语法如下所示：
  `(gdb) x/<n/f/u> <address>`
  n、f、u是可选的参数（可都有）。
  n是一个正整数，表示需要显示的内存单元的个数，也就是说从当前地址向后显示几个内存单元的内容，一个内存单元的大小由后面的u定义。
  f 表示显示的格式，参见下面。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；不写就默认是十六进制。
  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。
  表示一个内存地址。（也可以是函数名？检查函数的前若干个字节）
  注意：严格区分n和u的关系，n表示单元个数，u表示每个单元的大小。
  n/f/u三个参数可以一起使用。
  例如：`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表示输出三个单位，u表示按无符号十进制显示。
  `x 0x1000`查看1000处的十六进制数；`x/s 0x24000`显示0x24000地址处的字符串；`x/xg 0x402470`查看0x402470处的地址（注意地址是8字节16进制数）；`x/8xg`

`0x402470`查看从`0x402470`开始的个字节地址（16进制数）。

> *Examine memory: x/FMT ADDRESS.*
> *ADDRESS is an expression for the memory address to examine.*
> *FMT is a repeat count followed by a format letter and a size letter.*
> *Format letters are **o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left)**.*
> *Size letters are **b(byte), h(halfword), w(word), g(giant, 8 bytes)**.*
> *The specified number of objects of the specified size are printed*
> *according to the format.*

- **print**：（简写为p）
  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。

> *其它示例：*
> *`print $rax`：以十进制输出%rax中的内容*
> *`print /t $rax`：以二进制输出%rax中的内容*
> *`print /x $rax`：以十六进制输出%rax中的内容*
> *`print 0x100`：输出0x100的十进制表示*
> *`print /x 555`：输出555的十六进制表示*
> *`print /x ($rsp+8)`：以十六进制输出%rsp的内容加上8*
> *`print *(long*)0x123456`：输出位于地址0x123456的长整数*
> *`print *(long*)($rsp+8)`：输出位于地址%rsp+8的长整数*
> *`print (char*)0x123456`：输出位于地址0x123456的字符串（x/s）*

- 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。var可以是变量、寄存器（如`print $rax`）。
  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（不带显示格式则默认为十进制）。
  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。

> *字符串较长时，print可能无法全部显示。需设置输出上限`print elements`为无限（0）：*
> *(gdb) set print elements 0*
> *(gdb) show print elements*

- 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：
  行号。
  <+offset> 当前行号的正偏移量。
  <-offset> 当前行号的负偏移量。
  哪个文件的哪一行。
  函数名。
  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。

> ***next***：*继续执行语句，但是跳过子程序的调用。后可接数字表示跳过几条程序（不加则默认为一行）*
> ***nexti***：*单步执行语句，但和next不同的是，它会跟踪到子程序的内部，但不打印出子程序内部的语句。后可接数字。*
> ***step***：*与next类似，但是它会跟踪到子程序的内部，而且会显示子程序内部的执行情况。后可接数字。即`Single stepping until exit fromt this function`。*
> ***stepi***：*与step类似，但是比step更详细，是nexti和step的结合，每次跳过一行汇编。后可接数字。*

- 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
- 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。即`Run till exit from this function`。
- 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
  这里,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
  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。
- **break**（简写为b）
  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。
- 在当前的文件中为某一函数(假设为func)入口处设定断点：
  (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。
- 给某地址处设置断点
  (gdb) break *0x400540
- 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。
- 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 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)。
- **delete**：（简写为d）
  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。
- 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。