**常用GDB指令**

0.7372017.03.25 19:55:07字数 1,867阅读 8,178

**概述**

GDB是一个由GNU开源组织发布的、UNIX/[Linux](https://link.jianshu.com/?t=http://lib.csdn.net/base/linux)操作系统下的、基于命令行的、功能强大的程序调试工具。

一般来说，GDB主要帮忙你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。

虽然没有像VC、Eclipse、IDEA、Xcode等IDE那样的图形化操作环境，但是借助于调试命令GDB可以完成几乎所有你想要的功能，而且GDB的命令操控特性也特别适用于UNIX/Linux的命令行开发环境。

本文不介绍GCC编译器的编译选项和CoreDump的配置，仅介绍部分日常开发中的常用调试命令 。

**常用命令**

1. **file <文件名>**  
   加载被调试的可执行程序文件。因为一般都在被调试程序所在目录下执行GDB，因而文本名不需要带路径。

(gdb) file file-name

1. **attach <PID>**  
   关联指定进程。

(gdb) attach 1024 //关联进程号为1024的进程进行调试

1. **l**  
   List的简写，列出当前位置之后的10行代码；list line\_number: 列出line\_number之后的十行代码。

(gdb) l

1. **r**  
   Run的简写，运行被调试的程序。如果此前没有下过断点，则执行完整个程序；如果有断点，则程序暂停在第一个可用断点处。

(gdb) r

1. **c**  
   Continue的简写，继续执行被调试程序，直至下一个断点或程序结束。

(gdb) c

1. **b <行号>  
   b <函数名称>  
   b \*<函数名称>  
   b \*<代码地址>  
   d [编号]**  
   b是Breakpoint的简写，设置断点。两可以使用“行号”“函数名称”“执行地址”等方式指定断点位置。  
   其中在函数名称前面加\*符号表示将断点设置在“由编译器生成的prolog代码处”。如果不了解汇编，可以不予理会此用法。  
   break ... if ...：条件中断。  
   d是Delete breakpoint的简写，删除指定编号的某个断点，或删除所有断点。断点编号从1开始递增。

(gdb) b 8

(gdb) b main

(gdb) b \*main

(gdb) b \*0x804835c

(gdb) d

1. **bt**  
   backtrace的简写，列出调用栈。

(gdb) bt

1. **s**  
   执行一行源程序代码，如果此行代码中有函数调用，则进入该函数。相当于其它调试器中的“Step Into (单步跟踪进入)”。  
   这个命令必须在有源代码调试信息的情况下才可以使用（GCC编译时使用“-g”参数）。

(gdb) s

1. **n**  
   执行一行源程序代码，此行代码中的函数调用也一并执行。相当于其它调试器中的“Step Over (单步跟踪)”。  
   这个命令必须在有源代码调试信息的情况下才可以使用（GCC编译时使用“-g”参数）。

(gdb) n

1. **si**  
   si命令类似于s命令，但针对汇编指令。

(gdb) si

1. **ni**  
   ni命令类似于n命令，但针对汇编指令。

(gdb) ni

1. **p <变量名称>**  
   Print的简写，显示指定变量（临时变量或全局变量）的值。

(gdb) p i

(gdb) p nGlobalVar

1. **x**  
   和print命令需要指定变量不同，x命令需要指定内存地址。

(gdb) help x

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) and s(string).

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.

Defaults for format and size letters are those previously used.

Default count is 1. Default address is following last thing printed

with this command or "print".

(gdb) x /6cb 0x804835c //打印地址0x804835c起始的内存内容，连续6个字节，以字符格式输出。

1. **display ...  
   undisplay <编号>**  
   display，设置程序中断后欲显示的数据及其格式。  
   例如，如果希望每次程序中断后可以看到即将被执行的下一条汇编指令，可以使用命令display /i $pc，其中 $pc 代表当前汇编指令，/i 表示以十六进行显示。当需要关心汇编代码时，此命令相当有用。  
   undispaly，取消先前的display设置，编号从1开始递增。

(gdb) display /i $pc

(gdb) undisplay 1

1. **i**  
   Info的简写，用于显示各类信息，详情请查阅“help i”。

(gdb) help i

info address -- Describe where symbol SYM is stored

info all-registers -- List of all registers and their contents

info args -- Argument variables of current stack frame

info auto-load -- Print current status of auto-loaded files

info auto-load-scripts -- Print the list of automatically loaded Python scripts

info auxv -- Display the inferior's auxiliary vector

info bookmarks -- Status of user-settable bookmarks

info breakpoints -- Status of specified breakpoints (all user-settable breakpoints if no argument)

info checkpoints -- IDs of currently known checkpoints

info classes -- All Objective-C classes

info common -- Print out the values contained in a Fortran COMMON block

info copying -- Conditions for redistributing copies of GDB

info dcache -- Print information on the dcache performance

info display -- Expressions to display when program stops

info extensions -- All filename extensions associated with a source language

info files -- Names of targets and files being debugged

info float -- Print the status of the floating point unit

info frame -- All about selected stack frame

info frame-filter -- List all registered Python frame-filters

info functions -- All function names

info handle -- What debugger does when program gets various signals

info inferiors -- IDs of specified inferiors (all inferiors if no argument)

info line -- Core addresses of the code for a source line

info locals -- Local variables of current stack frame

info macro -- Show the definition of MACRO

info macros -- Show the definitions of all macros at LINESPEC

info mem -- Memory region attributes

info os -- Show OS data ARG

info pretty-printer -- GDB command to list all registered pretty-printers

info probes -- Show available static probes

info proc -- Show /proc process information about any running process

info program -- Execution status of the program

info record -- Info record options

info registers -- List of integer registers and their contents

info scope -- List the variables local to a scope

info selectors -- All Objective-C selectors

info set -- Show all GDB settings

info sharedlibrary -- Status of loaded shared object libraries

info signals -- What debugger does when program gets various signals

info skip -- Display the status of skips

info source -- Information about the current source file

info sources -- Source files in the program

info stack -- Backtrace of the stack

info static-tracepoint-markers -- List target static tracepoints markers

info symbol -- Describe what symbol is at location ADDR

info target -- Names of targets and files being debugged

info tasks -- Provide information about all known Ada tasks

info terminal -- Print inferior's saved terminal status

info threads -- Display currently known threads

info tracepoints -- Status of specified tracepoints (all tracepoints if no argument)

info tvariables -- Status of trace state variables and their values

info type-printers -- GDB command to list all registered type-printers

info types -- All type names

info variables -- All global and static variable names

info vector -- Print the status of the vector unit

info vtbl -- Show the virtual function table for a C++ object

info warranty -- Various kinds of warranty you do not have

info watchpoints -- Status of specified watchpoints (all watchpoints if no argument)

info win -- List of all displayed windows

(gdb) i r

1. **q**  
   Quit的简写，退出GDB调试环境。

(gdb) q

1. **help [命令名称]**  
   GDB帮助命令，提供对GDB名种命令的解释说明。  
   如果指定了“命令名称”参数，则显示该命令的详细说明；如果没有指定参数，则分类显示所有GDB命令，供用户进一步浏览和查询。

(gdb) help display

**学会使用帮助文档**

上述的命令其实都有更复杂的使用方法，可以通过help命令查看帮助，比如我们使用help p查看print命令的详细说明。

(gdb) help p

Print value of expression EXP.

Variables accessible are those of the lexical environment of the selected

stack frame, plus all those whose scope is global or an entire file.

$NUM gets previous value number NUM. $ and $$ are the last two values.

$$NUM refers to NUM'th value back from the last one.

Names starting with $ refer to registers (with the values they would have

if the program were to return to the stack frame now selected, restoring

all registers saved by frames farther in) or else to debugger

"convenience" variables (any such name not a known register).

Use assignment expressions to give values to convenience variables.

{TYPE}ADREXP refers to a datum of data type TYPE, located at address ADREXP.

@ is a binary operator for treating consecutive data objects

anywhere in memory as an array. FOO@NUM gives an array whose first

element is FOO, whose second element is stored in the space following

where FOO is stored, etc. FOO must be an expression whose value

resides in memory.

EXP may be preceded with /FMT, where FMT is a format letter

but no count or size letter (see "x" command).

上述帮助文档提到@参数，这个参数可以让你像查看数组内容一样打印连续内存的数据对象。“@”的左边是第一个内存的地址的值，“@”的右边则你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：

(gdb) p \*array@len

**动手实践**

从网上随便扒了一段代码保存为main.c。

#include <stdio.h>

int g\_var = 0;

static int \_add(int a, int b) {

printf("\_add callad, a:%d, b:%d\n", a, b);

return a+b;

}

int main(void) {

int n = 1;

printf("one n=%d, g\_var=%d\n", n, g\_var);

++n;

--n;

g\_var += 20;

g\_var -= 10;

n = \_add(1, g\_var);

printf("two n=%d, g\_var=%d\n", n, g\_var);

return 0;

}

**编译**

记得加-g参数

gcc -g -Wall -o main main.c

**执行**

结果如下：

./main

one n=1, g\_var=0

\_add callad, a:1, b:10

two n=11, g\_var=10

**开始调试**

gdb main

GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-94.el7

Copyright (C) 2013 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "x86\_64-redhat-linux-gnu".

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>...

Reading symbols from /media/sf\_temp/gdb-sample/main...(no debugging symbols found)...done.

1. **l**查看源码

(gdb) l

2

3 int g\_var = 0;

4

5 static int \_add(int a, int b) {

6 printf("\_add callad, a:%d, b:%d\n", a, b);

7 return a+b;

8 }

9

10 int main(void) {

11 int n = 1;

1. **b**下断点

(gdb) b 17

Breakpoint 1 at 0x40058d: file main.c, line 17.

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x000000000040058d in main at main.c:17

1. **r**运行

(gdb) r

Starting program: /media/sf\_temp/gdb-sample/main

one n=1, g\_var=0

Breakpoint 1, main () at main.c:17

17 g\_var += 20;

断点生效了，也可以试试条件断点

(gdb) b 17 if n==1

Breakpoint 1 at 0x40058d: file main.c, line 17.

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x000000000040058d in main at main.c:17

stop only if n==1

(gdb) r

Starting program: /media/sf\_temp/gdb-sample/main

one n=1, g\_var=0

Breakpoint 1, main () at main.c:17

17 g\_var += 20;

1. 试试**Continue**命令

(gdb) b 19

Breakpoint 2 at 0x4005ab: file main.c, line 19.

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x000000000040058d in main at main.c:17

breakpoint already hit 1 time

2 breakpoint keep y 0x00000000004005ab in main at main.c:19

(gdb) c

Continuing.

Breakpoint 2, main () at main.c:19

19 n = \_add(1, g\_var);

1. **s**进入\_add函数

(gdb) s

\_add (a=1, b=10) at main.c:6

6 printf("\_add callad, a:%d, b:%d\n", a, b);

1. **n**单步执行

(gdb) n

\_add callad, a:1, b:10

7 return a+b;

1. **p**打印变量值

(gdb) p a+b

$1 = 11

1. **bt**打印调用栈

(gdb) bt

#0 \_add (a=1, b=10) at main.c:7

#1 0x00000000004005bd in main () at main.c:19

1. **info**打印详细信息

(gdb) info f

Stack level 0, frame at 0x7fffffffe4c0:

rip = 0x400552 in \_add (main.c:7); saved rip 0x4005bd

called by frame at 0x7fffffffe4e0

source language c.

Arglist at 0x7fffffffe4b0, args: a=1, b=10

Locals at 0x7fffffffe4b0, Previous frame's sp is 0x7fffffffe4c0

Saved registers:

rbp at 0x7fffffffe4b0, rip at 0x7fffffffe4b8

(gdb) info b

Num Type Disp Enb Address What

1 breakpoint keep y 0x000000000040058d in main at main.c:17

breakpoint already hit 1 time

2 breakpoint keep y 0x00000000004005ab in main at main.c:19

breakpoint already hit 1 time

(gdb) info args

a = 1

b = 10

(gdb) info registers

rax 0x17 23

rbx 0x0 0

rcx 0x7fffffe9 2147483625

rdx 0x7ffff7dd8a00 140737351879168

rsi 0x7ffff7ff8000 140737354104832

rdi 0x0 0

rbp 0x7fffffffe4b0 0x7fffffffe4b0

rsp 0x7fffffffe4a0 0x7fffffffe4a0

r8 0x7ffff7a64938 140737348258104

r9 0x16 22

r10 0x0 0

r11 0x246 582

r12 0x400440 4195392

r13 0x7fffffffe5b0 140737488348592

r14 0x0 0

r15 0x0 0

rip 0x400552 0x400552 <\_add+37>

eflags 0x202 [ IF ]

cs 0x33 51

ss 0x2b 43

ds 0x0 0

es 0x0 0

fs 0x0 0

gs 0x0 0

1. **x**打印内存信息

(gdb) p a

$4 = 1

(gdb) p &a

$5 = (int \*) 0x7fffffffe4ac

(gdb) x /1db 0x7fffffffe4ac

0x7fffffffe4ac: 1

1. **q**退出调试

(gdb) q

A debugging session is active.

Inferior 1 [process 2916] will be killed.

Quit anyway? (y or n) y

**gdbtui**

单纯使用l命令查看源码在单步调试过程中十分不方便，所以官方提供了gdbtui这个工具，可以将调试界面分栏，实时显示源码。

**其他**

CoreDump的调试也是必备技能，但是使用的主要命令逃不开上述几个例子，更多的是要依赖个人经验，结合寄存器数据和内存数据进行分析。

至于GDB的多线程多进程调试实际开发中使用机会很少, 也就老鸟会用上些。这部分可以调试，不好调试，一般一调估计小半天就走了。  
常用的命令如下：

info threads

thread id

set follow-thread-mode parent/child

set scheduler-locking on/off

attach pid

分别是查看、切换、设置同步调试和加载进程。

默认设置下，在调试多进程程序时GDB只会调试主进程。但是GDB（>V7.0）支持多进程的分别以及同时调试，换句话说，GDB可以同时调试多个程序。只需要设置follow-fork-mode(默认值：parent)和detach-on-fork（默认值：on）即可。

| **follow-fork-mode** | **detach-on-fork** | **说明** |
| --- | --- | --- |
| parent | on | 只调试主进程（GDB默认） |
| child | on | 只调试子进程 |
| parent | off | 同时调试两个进程，gdb跟主进程，子进程block在fork位置 |
| child | off | 同时调试两个进程，gdb跟子进程，主进程block在fork位置 |

设置方法：set follow-fork-mode [parent|child] set detach-on-fork [on|off]  
查询正在调试的进程：info inferiors  
切换调试的进程： inferior <infer number>  
添加新的调试进程： add-inferior [-copies n] [-exec executable] ,可以用file executable来分配给inferior可执行文件。  
其他：remove-inferiors infno， detach inferior

我是咕咕鸡，一个还在不停学习的全栈工程师。  
热爱生活，喜欢跑步，家庭是我不断向前进步的动力。