GDB 工具的基本用法

赵宏智 整理
北京交通大学计算机与信息技术学院

GDB 工具是 Unix/Linux 系统上标配的调试工具，主要有如下几个功能：

1、启动程序，可以按照程序员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所指定的断点处停住。（断点可以是条件表达式）

3、当程序被停住时，可以通过 dump 文件来检查此时程序当前所发生的事。Dump 文件是进程的内存镜像，可以把程序的执行状态（包括内存状态、CPU 寄存器状态等）通过调试器保存到 dump 文件中，然后使用调试器加以查看和分析。

4、可以修改程序，将一个 BUG 修正从而测试其他 BUG。

一、GDB 工具的安装

在当前 Linux 系统下敲入 gdb 命令，看是否能够执行，如果能够执行，则表明已经安装了 gdb 工具。如果不能执行，则还需要安装 gdb 工具，在联网的情况下使用: yum install gdb 命令进行网络安装即可。可以使用 which gdb 查看是否安装成功，或者直接在命令行界面下输入 gdb 命令。

二、GDB 工具的常用命令

在使用 gdb 工具来调试可执行程序之前，需要在 gcc 编译时，带上-g 选项才能产生具有调试符号的程序，才能供 gdb 进行有源码的调试。典型如：gcc –g xxx.c xxx.S –o xxx。在产生带调试符号的可执行程序之后，即可使用 gdb 工具中的各项命令来对其分析和调试。

GDB 工具的命令很多，这里仅给出常用的一些命令。注意：（）里的小写字母为命令的简写形式。更多的命令可以自行查阅相关资料。

1， list(l)命令：列出多行源代码。如果代码较长，可以连续多个 list 命令来查看。

2， break(b)命令：设置断点，break 命令之后可接源文件行、函数名称、地址等，用法有：

- break [line-num] ：按代码行来设置断点，例如：break 3
- break [function-name]：按函数名来设置断点，例如：break main
- break *地址：按地址来设置断点，地址可以通过 disassemble 来查看，地址形如“0X400”;例如：break *0x400
- break if <condition>: E.g. break if i=10： 条件成立时程序停住
- info break：查看断点

3， start：开始执行程序，在第一条可执行语句处停下来。

4， run(r)命令： 运行程序，直到第一个断点或程序结束；

- 如果程序运行不需要输入参数，则直接输入：run

- 如果程序运行需要输入参数，则形如：run argv[1] argv[2]

5， 单步执行命令：

- step(s)命令：执行下一条语句，如果该语句为函数调用，则执行所调用函数执行其第一行语句；
- next(n)命令：执行下一条语句，如果该下一条语句为函数调用，不会进入函数内部执行(即不会一步步地调试函数内部语句)。
- continue(c)命令：继续程序的运行，直到下一个断点。如果没有下一个端点，则程序执行结束。

6， info(i)命令：

- info registers 查看 CPU 各寄存器的值；
- info args 查看当前函数参数的值
- info locals 查看当前函数中局部变量的值
- info frame 查看当前栈帧的详细信息

函数调用栈由连续的栈帧组成。每个栈帧记录一个函数调用的信息，这些信息包括函数参数，函数变量，函数运行地址。当程序启动后，栈中只有一个帧，这个帧就是 main 函数的帧。我们把这个帧叫做初始化帧或者叫做最外层帧。每当一个函数被调用，一个新帧将被建立，每当一个函数返回时，函数帧将被剔除。如果函数是个递归函数，栈中将有很多帧是记录同一个函数的。但前执行的函数的帧被称作最深帧，这个帧是现存栈中最近被创建的帧。在程序内部，函数栈帧用函数的地址来标记。gdb 为所有存活的栈帧分配一个数字编号，最深帧的编号是 0，被它调用的帧的编号就是 1。

7，x 命令：查看内存的地址以及所存储的内容；

语法如下： x/<n/f/u> <addr>

n、f、u 是可选的参数。

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。

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。

注意： n 表示单元个数，u 表示每个单元的大小。

<addr>表示一个内存地址。

n/f/u 三个参数可以一起使用。例如：

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 表示按十六进制显示。

例如：x/20xw &a 显示变量 a 处开始的 20 个内存单元，16 进制，4 字节每单元

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。输出格式

一般来说，GDB 会根据变量的类型输出变量的值。但你也可以自定义 GDB 的输出的格式。例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用 GDB 的数据显示格式：

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。

8，disassemble(disass)命令：是反汇编命令，可以对具体的函数进行反汇编。

- disassemble function_name：查看函数 function_name 对应的汇编代码；
- 添加/r 选项，即（gdb） disassemble /r function_name：在前述基础上增加显示机器码；
- 添加/m 选项，即（gdb） disassemble /m function_name：在前述基础上增加显示源码（源码是 C 语句则显示 C 代码，源码是汇编语句则显示汇编代码）；
- 也可以同时添加/rm 选项；

9，display 命令：跟踪查看某个变量，每次单步停下来都显示该变量当前的值。 undisplay 命令则是取消跟踪观察变量。

10，watch 命令： 监视变量值的变化。当被设置的观察点变量发生变化时，打印显示！

与 display 命令一直显示某个变量的值的方式相比，用 watch 命令只显示变量值的变化情况，如果变量值不变则不显示，调试效率要高很多。

11，file 命令：在当前程序运行完毕退出之后，可以使用 file 命令重新载入待调试程序，重新进行调试。典型格式是：file executablefilenam。

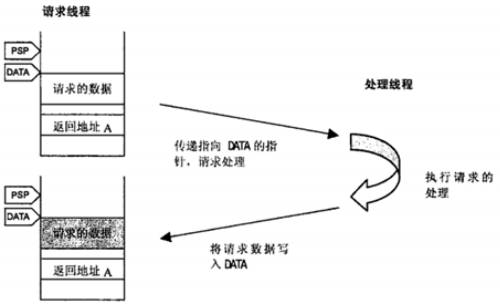
12，kill 命令：终止当前的调试过程；

13，backtrace(bt)命令：查看当前函数及其被调用的信息

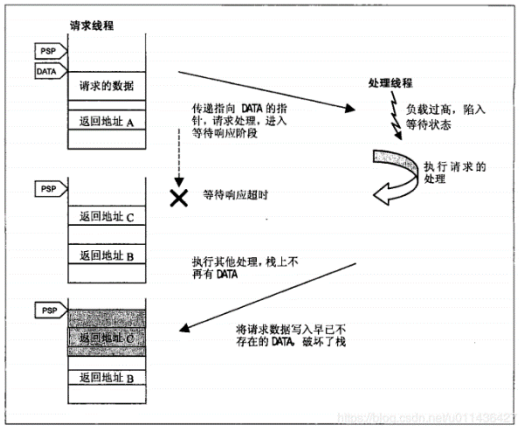
backtrace 命令是依据进程栈里保存的函数返回地址来显示函数间调用关系的，可以用于回溯函数调用栈。如果 bt 命令下所显示的某些函数不正确或者出现段错误，则可以认为是栈被破坏了，还可以看到是在哪个函数的调用处产生的段错误。

线程间的冲突是导致进程栈被破坏的原因之一。举一个例子来展示栈被破坏的过程：如下图所示，在线程间的数据处理上传递了栈的指针，导致了其他线程向该地址写入了数据。而这个其他线程向栈内写入数据的操作被推迟了，推迟后的写操作写入了一个其它线程的栈空间，从而导致了栈破坏

正常时，应用程序行为如下：



问题时，应用程序的行为如下：



当因为上述原因出现段错误的时候，执行 backtrace，如下图所示，可以看到是哪里的调用，产生的这个段错误。？？()就表明调用当前函数的上一级函数是有误的，其原因可能是进程栈被破坏所导致的。

```
(gdb) bt
#0 0x00000003b4869ac80 in nanossleep () from /lib64/libc.so.6
#1 000ee1c2000ee1c1 in ?? ()
#2 000ee1c2000ee1c3 in ?? ()
#3 000ee1c2000ee1c5 in ?? ()
```

14，help(h)命令：查看命令帮助，具体命令查询为输入 help+命令；

15，quit(q)命令或 ctrl+c 命令：退出 gdb 环境

三、示例代码

1，示例代码 1：

C 语言-ArmV8 汇编混合编程，实现两个数的相加：

```
------------example.c---C代码主函数main--------
#include <stdio.h>
#include <stdlib.h>
typedef unsigned int u32;
extern int plus(u32, u32);
int main()
{
    u32 a = 2;
    u32 b = 3;

    printf("%d\n", plus(a, b));

    return 0;
}
```

----------plus.S-----汇编代码函数 plus----------

```
#include "plus.h"
ENTRY(plus)
add w0, w1, w0
ret
ENDPROC(plus)
```

----------plus.h----------汇编代码的头文件----------

```
#ifndef ENTRY
#define ENTRY(name) \
    .globl name ; \
    .align 4 ; \
    name:
#endif

/* If symbol 'name' is treated as a subroutine (gets called, and returns)
 *  * then please use ENDPROC to mark 'name' as STT_FUNC for the benefit of
 *   * static analysis tools such as stack depth analyzer.
 *
 *   */
#ifndef ENDPROC
#define ENDPROC(name) \
    .type name, @function ; \
    .size name, .-name
#endif
```

```
[root@ecs-102d test]# ls
example.c  plus.h  plus.S
```

注意：用gcc 编译汇编的文件，汇编文件扩展名必须是S，不能是s，s只做编译未做预处理。

2，对例 1 代码进行带调试信息的编译：

gcc –g example.c plus.S –o example

生成带调试信息的可执行文件 example

```
[root@ecs-102d test]# gcc -g example.c plus.S -o example
[root@ecs-102d test]# ls
example  example.c  plus.h  plus.S
```

2，示例代码 2：

纯汇编程序 hello.s（或者 hello.S，在这里不区分 s 的大小写）

```
.text    @ 这个定义可选；
.global tart1
tart1:
    mov x0,#0
    ldr x1,=msg
```

```
    mov x2,len
    mov x8,64
    svc #0

    mov x0,123
    mov x8,93
    svc #0

.data
msg:
    .ascii "Hello World!\n"
len=.-msg
```

对该hello.s采用gcc进行编译，使用了-g 和 –nostartfiles选项，编译生成了带调试符号表的可执行文件hello。

```
[root@ecs-102d helloworld]# ls
hello.s
[root@ecs-102d helloworld]# gcc -g hello.s -nostartfiles -o hello
/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000400250
[root@ecs-102d helloworld]# ls
hello  hello.s
```

如果不采用-nostartfiles 选项，则会出现如下错误：

```
[root@ecs-102d helloworld]# gcc -g hello.s  -o hello
/usr/lib/gcc/aarch64-redhat-linux/4.8.5/../../../../lib64/crt1.o: In function `_start':
(.text+0x30): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

即找不到主函数 main 或者入口函数的错误，这是采用 gcc 编译器时会出现的问题。如果 gcc 编译时采用了 –nostartfiles 选项，则可以避免该错误。

3，示例代码 3 采用 gcc 来编译

如果将示例代码 2 中的“tart1”修改为“_start”，即如下的示例代码 3：

```
# h2.s
.global _start
_start:
    mov x0,#0
    ldr x1,=msg
    mov x2,len
    mov x8,64
    svc #0

    mov x0,123
    mov x8,93
    svc #0

.data
msg:
    .ascii "Hello World!\n"
len=.-msg
```

在采用 gcc 对其进行编译时就会出现对“_start”多个定义导致定义冲突的问题：

```
[root@ecs-102d helloworld]# gcc -g h2.s -o h2
/tmp/ccn5qY1a.o: In function `_start':
/root/helloworld/h2.s:4: multiple definition of `_start'
/usr/lib/gcc/aarch64-redhat-linux/4.8.5/../../../../lib64/crt1.o:(.text+0x0): first defined here
/usr/lib/gcc/aarch64-redhat-linux/4.8.5/../../../../lib64/crt1.o: In function `_start':
(.text+0x30): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

此时可以添加“-nostartfiles 选项”来避免该问题。需要说明的是，由于采用了-nostartfiles 选项，那么在带有调试信息的可执行文件 hello 中就不再含有程序从哪里开始 start 的信息了，此时 gdb 中就不能使用 start 命令来执行程序的第一行代码了。但是可以采用 break 1 和 run 组合的方式来代替 start 命令寻找程序的第一行并开始执行。

4，示例代码 3 采用 as 和 ld 命令来编译

as 和 ld 是 linux 下的汇编命令和连接命令，可以用其来编译和连接汇编源代码，生成可执行程序。由于 gcc 在执行时会默认使用自身库中的 crt1.o 中的_start 作为程序的入口点，就会与示例代码中定义的“_start”入口点冲突，导致示例 3 中的问题，对具体过程感兴趣的同学可以参考“http://www.doc88.com/p-9847334917507.html”。使用 as 和 ld 这两个命令可以避免 gcc 编译时产生的“_start”多定义冲突问题。

为了生成带调试符号表的可执行程序来在 gdb 环境下进行调试，需要在使用 as 命令时使用-gstabs 选项。这两个命令使用示例如下：

```
[root@ecs-102d helloworld]# as -gstabs h2.s -o h2.o
[root@ecs-102d helloworld]# ld h2.o -o h2
```

这样就可以生成带调试符号表的可执行文件 h2。

综上所述，如果要对含有”_start”标号的汇编代码进行带调试符号表的编译，为避免_start 多定义冲突问题，有两种方式：一是使用 gcc –g –nostartfiles xx.s –o xx 命令来编译；二是使用 as –gstabs xx.s –o xx.o 命令和 ld xx.o –o xx 命令来编译和链接。在 gdb 环节下对这两种方式生成的可执行文件进行调试时，采用 break 1 和 run 命令来取代 start 命令来开始调试可执行文件的第一行代码。

四、采用 GDB 工具来调试含调试信息的可执行文件 example

1，采用 gdb 来调试 example 文件

在命令行输入： gdb example，回车后出现：

```
[root@ecs-102d test]# gdb example
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-119.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 "aarch64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/test/example...done.
(gdb)
```

如果 gcc 编译时不使用-g 选项，则出现“no debugging symbols found”字样，无法对其进行 gdb 调试。

```
Reading symbols from /root/test/example...(no debugging symbols found)...done.
```

2，用多个 list 命令查看多行源代码

```
(gdb) list
6
7       extern int plus(u32, u32);
8
9       int main()
10      {
11          u32 a = 2;
12          u32 b = 3;
13
14          printf("%d\n", plus(a, b));
15
(gdb) list
16          return 0;
17      }
```

3，在代码的第 9 行插入第一个断点

```
(gdb) list
2       #include <stdio.h>
3       #include <stdlib.h>
4
5       typedef unsigned int u32;
6
7       extern int plus(u32, u32);
8
9       int main()
10      {
11          u32 a = 2;
(gdb) list
12          u32 b = 3;
13
14          printf("%d\n", plus(a, b));
15
16          return 0;
17      }
(gdb) break 9
Breakpoint 1 at 0x400608: file example.c, line 9.
```

在此基础上，再使用 run 命令，则表示程序先执行到第 9 行处就暂停。

4，采用 start 命令来运行第一行代码

```
(gdb) start
Temporary breakpoint 1 at 0x400608: file example.c, line 11.
Starting program: /root/test/example

Temporary breakpoint 1, main () at example.c:11
11          u32 a = 2;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.aarch64
```

5，采用 next 命令来单步执行

```
(gdb) next
12          u32 b = 3;
(gdb) next
14          printf("%d\n", plus(a, b));
(gdb) next
5
16          return 0;
(gdb) next
17      }
```

注意观察：next 命令不展开 plus 函数内部的代码来进行单步，直接跳过。

6，采用 step 命令来单步执行

```
(gdb) step
12          u32 b = 3;
(gdb) step
14          printf("%d\n", plus(a, b));
(gdb) step
plus () at plus.S:4
4       add w0, w1, w0
(gdb) step
5       ret
(gdb)
```

注意观察：step 命令展开 plus 函数内部的代码来进行单步。

7，在程序单步执行过程中，可以：

- 使用 info locals 命令来查看当前函数中局部变量的值

```
(gdb) step
5
main () at example.c:16
16          return 0;
(gdb) i locals
a = 2
b = 3
(gdb)
```

当前函数 main 中的局部变量为 a 和 b。

- 使用 info registers 命令来查看处理器中各寄存器的值。

```
(gdb) info registers
x0             0x3      3
x1             0xffffffffff658   281474976708184
x2             0xffffffffff668   281474976708200
x3             0x0      0
x4             0x400600 4195840
x5             0xc4548c42253258dc       -4299657528512194340
x6             0x0      0
x7             0xb1aba9a0b8bba7ff       -5644231200219617281
x8             0x2f2f2f2f2f2f2f2f       3399988123389603631
x9             0x18     24
x10            0xffffffff       4294967295
x11            0x90000000       2415919104
x12            0x90     144
x13            0xf      15
x14            0xffffbe7fffa8   281473877802920
x15            0x1915d7 1643991
x16            0xffffbe631624   281473875908132
x17            0x420000 4325376
x18            0xffffffffff420   281474976707616
x19            0x0      0
x20            0x0      0
x21            0x4004a0 4195488
x22            0x0      0
x23            0x0      0
x24            0x0      0
x25            0x0      0
x26            0x0      0
x27            0x0      0
x28            0x0      0
x29            0xffffffffff4f0   281474976707824
x30            0xffffbe631714   281473875908372
sp             0xffffffffff4f0   0xffffffffff4f0
pc             0x400618 0x400618 <main+24>
cpsr           0x60200000       1612709888
fpsr           0x0      0
fpcr           0x0      0
```

- 使用 x 命令来查看 example 程序中变量 a 和 b 所在的内存地址（用&a 和&b 表示）以及取值情况，在下图中采用了 x/x &a 以及 x/x &b 命令。

```
Breakpoint 1, main () at example.c:11
11          u32 a = 2;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.aarch64
(gdb) x/x &a
0xffffffff50c: 0x00000000
(gdb) next
12          u32 b = 3;
(gdb) x/x &a
0xffffffff50c: 0x00000002
(gdb) x/x &b
0xffffffff508: 0x00000000
(gdb) x/x &b
0xffffffff508: 0x00000000
```

- 可以使用 disassemble 命令来查看函数如 plus 的汇编代码

```
(gdb) disassemble plus
Dump of assembler code for function plus:
   0x0000000000400640 <+0>:     add     w0, w1, w0
   0x0000000000400644 <+4>:     ret
End of assembler dump.
(gdb)
```

- 可以使用 display 命令来在每一步查看待观察的变量值，比如 display a

```
(gdb) display a
2: a = 0
(gdb) next
12          u32 b = 3;
2: a = 2
(gdb) next
14          printf("%d\n", plus(a, b));
2: a = 2
(gdb) next
5
16          return 0;
2: a = 2
(gdb)
```

在每一次 next 之后，都会显示变量 a 的值。

- 使用 watch 命令来在变量 a 的值的变化情况

```
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Temporary breakpoint 5 at 0x400608: file example.c, line 11.
Starting program: /root/test/example

Temporary breakpoint 5, main () at example.c:11
11          u32 a = 2;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.aarch64
(gdb) watch a
Hardware watchpoint 6: a
(gdb) next
Hardware watchpoint 6: a

Old value = 0
New value = 2
main () at example.c:12
12          u32 b = 3;
(gdb) next
14          printf("%d\n", plus(a, b));
(gdb) next
5
16          return 0;
(gdb) next
17      }
(gdb) next

Watchpoint 6 deleted because the program has left the block in
which its expression is valid.
```

可以看出，在使用 watch a 命令之后，只有当变量 a 的值发生变化时才在屏幕上打印出变化前的 a 值和变化后的 a 值，其它时刻不在屏幕上打印 a 的值。在监控效率上，使用 watch 命令要比 display 命令的效率高的多。

- 使用 backtrace 命令来在观察当前函数被调用的情况

```
(gdb) step
14          printf("%d\n", plus(a, b));
(gdb) step
plus () at plus.S:4
4       add w0, w1, w0
(gdb) backtrace
#0  plus () at plus.S:4
#1  0x0000000000400624 in main () at example.c:14
(gdb)
```

当前函数（用“#0”标志）plus 被上一级函数（用“#1”）main 所调用。

- 使用 info frame 命令来查看当前栈帧的信息

```
(gdb) info frame
Stack level 0, frame at 0xffffffff510:
 pc = 0x400608 in main (example.c:11); saved pc 0xffffbe631714
 source language c.
 Arglist at 0xffffffff4f0, args:
 Locals at 0xffffffff4f0, Previous frame's sp is 0xffffffff510
 Saved registers:
  x29 at 0xffffffff4f0, x30 at 0xffffffff4f8
(gdb) step
12          u32 b = 3;
(gdb) step
14          printf("%d\n", plus(a, b));
(gdb)
plus () at plus.S:4
4       add w0, w1, w0
(gdb)
5       ret
(gdb) info frame
Stack level 0, frame at 0xffffffff4f0:
 pc = 0x400644 in plus (plus.S:5); saved pc 0x400624
 called by frame at 0xffffffff510
 source language asm.
 Arglist at 0xffffffff4f0, args:
 Locals at 0xffffffff4f0, Previous frame's sp is 0xffffffff4f0
(gdb)
```

五、作业题：

参照第四节，用 gdb 的各项命令来查看示例 2 代码编译生成的 hello 可执行文件。