# 后台启动程序的打印

后台启动的打印一般都看不见，在启动脚本.sh文件中将echo的内容输出到/dev/console即可

|  |
| --- |
| echo "------hello-------" > /dev/console |

# printf重定向

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>      int main()  {  fflush(stdout);  setvbuf(stdout,NULL,\_IONBF,0);  printf("test stdout\n");  int save\_fd = dup(STDOUT\_FILENO); // 保存标准输出 文件描述符 注:这里一定要用 dup 复制一个文件描述符. 不要用 = 就像是Winodws下的句柄.  int fd = open("test1.txt",(O\_RDWR | O\_CREAT), 0644);  dup2(fd,STDOUT\_FILENO); // 用我们新打开的文件描述符替换掉 标准输出  printf("test file\n");    //再恢复回来标准输出. 两种方式  //方法1 有保存 标准输出的情况  //dup2(save\_fd,STDOUT\_FILENO);    //方法2 没有保存 标准输出的情况  int ttyfd = open("/dev/tty",(O\_RDWR), 0644);  dup2(ttyfd,STDOUT\_FILENO);  printf("test tty\n");  } |
| #include <stdlib.h>  #include <stdio.h>  #include <unistd.h>    int main()  {  fflush(stdout);  setvbuf(stdout,NULL,\_IONBF,0);  printf("test stdout\n");  freopen("test1.txt","w",stdout); //注: 不要使用这类的代码 stdout = fopen("test1.txt","w"); 这样的话输出很诡异的. 最好使用 freopen 这类的函数来替换它.  printf("test file\n");  freopen("/dev/tty","w",stdout);  printf("test tty\n");  } |

# slub debug

static unsigned long kmem\_cache\_flags(unsigned long object\_size,

unsigned long flags, const char \*name,

void (\*ctor)(void \*))

{

/\*

\* Enable debugging if selected on the kernel commandline.

\*/

//if (slub\_debug && (!slub\_debug\_slabs || (name &&

// !strncmp(slub\_debug\_slabs, name, strlen(slub\_debug\_slabs)))))

// flags |= slub\_debug;

flags |= SLAB\_RED\_ZONE | SLAB\_POISON | SLAB\_STORE\_USER | SLAB\_DEBUG\_FREE;

return flags;

}

# echo 8 > /proc/sys/kernel/printk

# 内存地址改变产生硬中断

==================================

中断处理函数

==================================

#include <linux/perf\_event.h>

#include <linux/hw\_breakpoint.h>

struct perf\_event \* \_\_percpu \*sample\_hbp;

struct perf\_event\_attr attr;

struct page \*my\_page;

static void sample\_hbp\_handler(struct perf\_event \*bp,

                                   struct perf\_sample\_data \*data,

                                   struct pt\_regs \*regs)

{

         unregister\_wide\_hw\_breakpoint(sample\_hbp);

         dump\_stack();

}

==================================

中断监控函数

==================================

static int iii=0;

static int \_\_videobuf\_dc\_alloc(struct device \*dev,

                                   struct videobuf\_dma\_contig\_memory \*mem,

                                   unsigned long size, gfp\_t flags)

{

         mem->size = size;

         mem->vaddr = dma\_alloc\_coherent(dev, mem->size,

                                               &mem->dma\_handle, flags);

if(iii==0){

         iii++;

         hw\_breakpoint\_init(&attr);

**attr.bp\_addr = &(mem->vaddr); //监控内存地址**

         attr.bp\_len = HW\_BREAKPOINT\_LEN\_8;

         attr.bp\_type = HW\_BREAKPOINT\_W;

         sample\_hbp = register\_wide\_hw\_breakpoint(&attr, sample\_hbp\_handler, NULL);

         if (IS\_ERR((void \_\_force \*)sample\_hbp)) {

                   printk("register\_wide\_hw\_breakpoint fail\n");

         }

}

         if (!mem->vaddr) {

                   dev\_err(dev, "memory alloc size %ld failed\n", mem->size);

                   return -ENOMEM;

         }

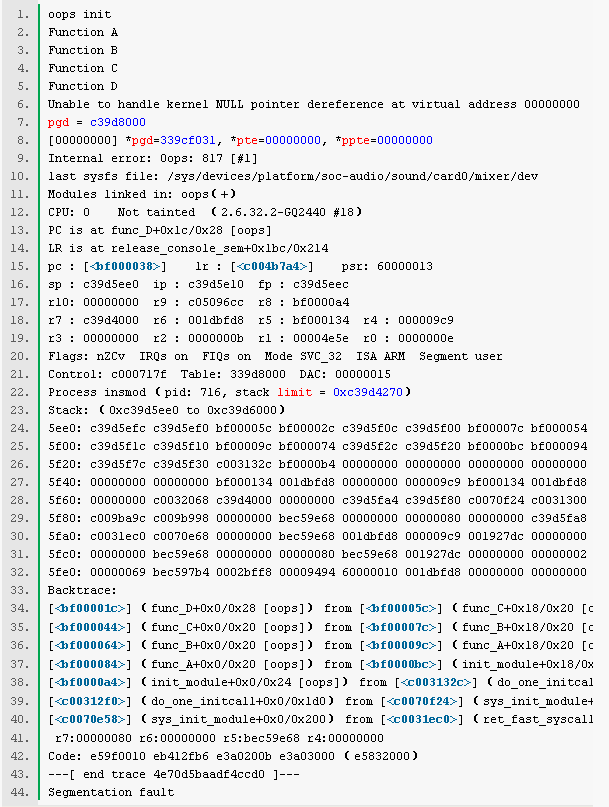
         dev\_dbg(dev, "dma mapped data is at %p (%ld)\n", mem->vaddr, mem->size);

         return 0;

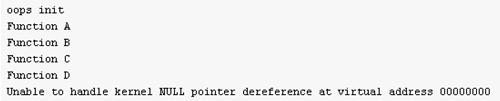
}

# 内核空指针

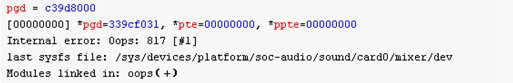
## 内核错误信息：



## 错误分析：



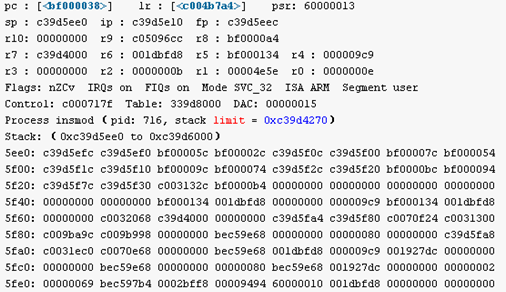
前面5行是加载模块时，由模块中的printk打印出来的语句。但发现在打印"Function D"后，内核提示"Unable to handle kernel NULL pointer dereference at virtual address 00000000"。很显然，这说明在打印出"Function D"之后，模块因为访问了空指针而出错。如果这个时候进入参考代码会发现，在调用printk（"Function D"）之后，仅有一条语句"\*p =a+5"，由此可以很轻松地确定，非法的地址访问就出现在那里了。当然，在实际开发中，不会有这么简单的情况，因此，还得找出其他更有效的定位方案。



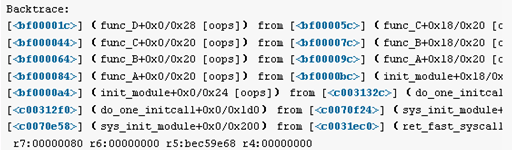
这些信息，和这个模块的调试没多大关系，它们是虚拟内存页目录、页表信息、oops错误号以及最后访问的sysfs文件等。



它直接地告诉了我们，oops出错时，PC是位于func\_D函数标号之后的0x1c处（怎么去寻找它？后面会进行分析。另外请思考后面的0x28代表什么？）。



寄存器信息之后是栈信息，但这里还用不上，先略过。



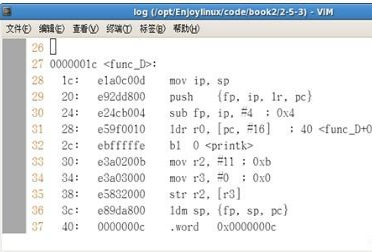
最后的部分，也就是Backtrace标号开始的地方，它是oops的精华。它表示回溯信息，也告诉调试者在oops出错之前，模块调用了那些函数。当然，在本实例中，可以看到模块调用了func\_D后就出错了，显然错误就在func\_D中了。



Code标号开始的字段记录了模块出错前最后几条机器码，其中被括号括起来的就是oops出错对应的机器码。

## 解决问题

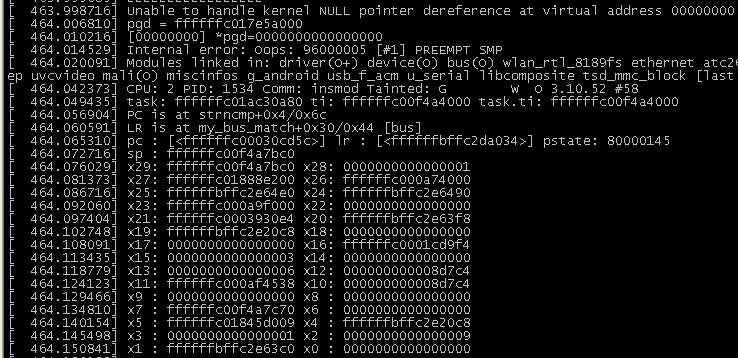
根据上面的分析，可以使用反汇编来确定出错的位置。在RHEL5中，使用命令：arm-linux-objdump -D -S oops.ko >log，将模块文件反汇编到log中，使用vim打开该文件log，直接找到func\_D标号处，如图6-17所示。



根据前面的信息，出错位置应该在func\_D+0x1c处，func\_D在0x1c，所以出错地址应该是0x38。看看这句汇编代码，前面的语句将寄存器 r3赋值为0，然后这句又试图将寄存器r2的值存入到r3指向的地址处，也就是向0地址写。因此出错。再来看看这句出错代码对应的机器码 e5832000，显然就是之前在opps的Code字段中看到的被括起来的那个。

修改内核Makefile， 增加编译选项-g后，再次编译模块。然后将新得到的oops.ko反汇编，使用命令：arm-linux-objdump -D -S oops.ko >log。用vim打开log文件。这时，通过汇编混合C语言调试信息的结果，结合前面的分析，可以很轻松的定位到C语言的错误语句就出现在"\*p = a + 5"处.

# 内核空指针



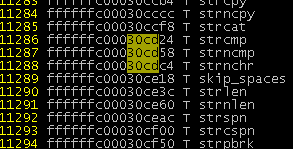
1,根据PC值，找到导致错误的指令。

pc = ffffffc00030cd5c它属于什么的地址？是内核的地址，还是通过insmod加载的驱动程序的地址？

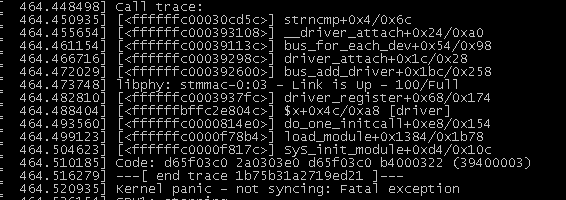
先判断是否属于内核的地址 : 看 内核编译makefile目录下的 System.map(编译完内核都会发现在内核根目录下面多出来一个System.map文件)

确定内核的函数的地址范围 : ffffffc000080000 ~ ffffffc000d57000。

#cat System.map



根据PC指针可以确定是内核的strncmp引起的。



# Segmentation fault (core dumped)

## 使用core文件调试程序

**看下面的例子：**

/\*core\_dump\_test.c\*/

#include <stdio.h>

char \*str = "test";

void core\_test(){

str[1] = 'T';

}

int main(){

core\_test();

return 0;

}

**编译：**

gcc –g core\_dump\_test.c -o core\_dump\_test

如果需要调试程序的话，使用gcc编译时加上-g选项，这样调试core文件的时候比较容易找到错误的地方。

**执行：**

**./core\_dump\_test**

段错误

运行core\_dump\_test程序出现了“段错误”，但没有产生core文件。这是因为系统默认core文件的大小为0，所以没有创建。可以用ulimit命令查看和修改core文件的大小。

**ulimit -c**

**ulimit -c 1000**

**ulimit -c 1000**

-c 指定修改core文件的大小，1000指定了core文件大小。也可以对core文件的大小不做限制，如：

**ulimit -c unlimited**

**ulimit -c unlimited**

如果想让修改永久生效，则需要修改配置文件，如 .bash\_profile、/etc/profile或/etc/security/limits.conf。

**再次执行： 会生成core文件**

**./core\_dump\_test**

段错误 (core dumped)

**ls core.\***

core.6133

可以看到已经创建了一个core.6133的文件.6133是core\_dump\_test程序运行的进程ID。

**调式core文件**

core文件是个二进制文件，需要用相应的工具来分析程序崩溃时的内存映像。

file core.6133

core.6133: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style, from 'core\_dump\_test'

**在Linux下可以用GDB来调试core文件。**

gdb core\_dump\_test core.6133

GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)

Copyright 2003 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux-gnu"...

Core was generated by `./core\_dump\_test'.

Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/tls/libc.so.6...done.

Loaded symbols for /lib/tls/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

#0 0x080482fd in core\_test () at core\_dump\_test.c:7

7 str[1] = 'T';

(gdb) **where**

#0 0x080482fd in core\_test () at core\_dump\_test.c:7

#1 0x08048317 in main () at core\_dump\_test.c:12

#2 0x42015574 in \_\_libc\_start\_main () from /lib/tls/libc.so.6

GDB中键入where，就会看到程序崩溃时堆栈信息（当前函数之前的所有已调用函数的列表（包括当前函数），gdb只显示最近几个），我们很容易找到我们的程序在最后崩溃的时候调用了core\_dump\_test.c 第7行的代码，导致程序崩溃。注意：在编译程序的时候要加入选项-g。您也可以试试其他命令，　如　fram、list等。更详细的用法，请查阅GDB文档。

core文件创建在什么位置

在进程当前工作目录的下创建。通常与程序在相同的路径下。但如果程序中调用了chdir函数，则有可能改变了当前工作目录。这时core文件创建在chdir指定的路径下。有好多程序崩溃了，我们却找不到core文件放在什么位置。和chdir函数就有关系。当然程序崩溃了不一定都产生core文件。

什么时候不产生core文件

在下列条件下不产生core文件：

( a )进程是设置-用户-ID，而且当前用户并非程序文件的所有者；

( b )进程是设置-组-ID，而且当前用户并非该程序文件的组所有者；

( c )用户没有写当前工作目录的许可权；

( d )文件太大。core文件的许可权(假定该文件在此之前并不存在)通常是用户读/写，组读和其他读。

利用GDB调试core文件，当遇到程序崩溃时我们不再束手无策。

# libmediaplayerservice.so文件启动失败

源代码一致，但一个so文件启动正常，另一个so文件启动失败。

so文件位置： android/out\_703d\_ebox/target/product/gs703d/symbols/system/lib

1. **查看so文件信息，发现两个so文件的符号表一致**

readelf -a libmediaplayerservice.so > /home/yhong/readelf.txt

1. **反汇编.so文件，发现两个so文件的反汇编文件有一处不一样，涉及到地址0003fe80。**

arm-none-linux-gnueabi-objdump -S libmediaplayerservice.so > /home/yhong/dump.txt

1. **查看libmedia.so中地址为0003fe80对应的源文件信息。发现代码没有异常，那可能是编译链接时部分信息优化时没有更新。**

addr2line -e libmediaplayerservice.so 0003fe80

查看0003fe80对应的源文件信息：

android/frameworks/av/media/libmediaplayerservice/MediaRecorderClient.cpp:303

**//MediaRecorderClient.cpp:303**

**303 MediaRecorderClient::MediaRecorderClient(const sp<MediaPlayerService>& service, pid\_t pid)**

**304 {**

**305 ALOGV("Client constructor");**

**306 mPid = pid;**

**307 mRecorder = new StagefrightRecorder;**

**308 mMediaPlayerService = service;**

**309 }**

1. **最终解决办法：**

cd /home/yhong/android/frameworks/av/media/libmediaplayerservice

**mm -B //mm -B重新连接所有文件**

//可能原因：

编译库时，可能部分库之间的连接没有更新，导致库被修改，而其他库并未察觉

# gdb & gdbserver

开发板：

gdbserver 192.168.183.127:2345 ./test\_debug

其中192.168.183.127：本开发板的ip

123：端口号，自己随便写的

./test\_debug：要调试的程序

PC机：

gdb ./test-debug target remote 192.168.183.127:2345

gdbserver 192.168.90.180:2345 ./onvifsvr &

gdb ./onvifsvr target remote 192.168.91.106:2345

# gdb sample

arm-linux-androideabi-gdb ./onvifsvr core

源程序：tst.c

1 #include

2

3 int func(int n)

4 {

5 int sum=0,i;

6 for(i=0; i<7; i++) {

8 sum+=i;

9 }

10 return sum;

11 }

12

13

14 main()

15 {

16 int i;

17 long result = 0;

18 for(i=1; i<=100; i++)

19 {

20 result += i;

21 }

22

23 printf("result[1-100] = %d /n", result );

24 printf("result[1-250] = %d /n", func(250) );

25 }

编译生成执行文件：（Linux下）

hchen/test> cc -g tst.c -o tst

使用GDB调试：

hchen/test> gdb tst <---------- 启动GDB

GNU gdb 5.1.1

Copyright 2002 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-suse-linux"...

(gdb) l <-------------------- l命令相当于list，从第一行开始例出原码。

1 #include

2

3 int func(int n)

4 {

5 int sum=0,i;

6 for(i=0; i 7 {

8 sum+=i;

9 }

10 return sum;

(gdb) <-------------------- 直接回车表示，重复上一次命令

11 }

12

13

14 main()

15 {

16 int i;

17 long result = 0;

18 for(i=1; i<=100; i++)

19 {

20 result += i;

(gdb) break 16 <-------------------- 设置断点，在源程序第16行处。

Breakpoint 1 at 0x8048496: file tst.c, line 16.

(gdb) break func <-------------------- 设置断点，在函数func()入口处。

Breakpoint 2 at 0x8048456: file tst.c, line 5.

(gdb) info break <-------------------- 查看断点信息。

Num Type Disp Enb Address What

1 breakpoint keep y 0x08048496 in main at tst.c:16

2 breakpoint keep y 0x08048456 in func at tst.c:5

(gdb) r <--------------------- 运行程序，run命令简写

Starting program: /home/hchen/test/tst

Breakpoint 1, main () at tst.c:17 <---------- 在断点处停住。

17 long result = 0;

(gdb) n <--------------------- 单条语句执行，next命令简写。

(gdb) s <--------------------- 单条语句执行，step into。

18 for(i=1; i<=100; i++)

(gdb) n

20 result += i;

(gdb) n

18 for(i=1; i<=100; i++)

(gdb) n

20 result += i;

(gdb) c <--------------------- 继续运行程序，continue命令简写。

Continuing.

result[1-100] = 5050 <----------程序输出。

Breakpoint 2, func (n=250) at tst.c:5

5 int sum=0,i;

(gdb) n

6 for(i=1; i<=n; i++)

(gdb) p i <--------------------- 打印变量i的值，print命令简写。

$1 = 134513808

(gdb) n

8 sum+=i;

(gdb) n

6 for(i=1; i<=n; i++)

(gdb) p sum

$2 = 1

(gdb) n

8 sum+=i;

(gdb) p i

$3 = 2

(gdb) n

6 for(i=1; i<=n; i++)

(gdb) p sum

$4 = 3

(gdb) bt <--------------------- 查看函数堆栈。

#0 func (n=250) at tst.c:5

#1 0x080484e4 in main () at tst.c:24

#2 0x400409ed in \_\_libc\_start\_main () from /lib/libc.so.6

(gdb) finish <--------------------- 退出函数。

Run till exit from #0 func (n=250) at tst.c:5

0x080484e4 in main () at tst.c:24

24 printf("result[1-250] = %d /n", func(250) );

Value returned is $6 = 31375

(gdb) c <--------------------- 继续运行。

Continuing.

result[1-250] = 31375 <----------程序输出。

Program exited with code 027. <--------程序退出，调试结束。

(gdb) q <--------------------- 退出gdb。

hchen/test>

# Gdb cmd

===============================================================================

run（r） //运行程序，当遇到断点后，程序会在断点处停止运行，等待用户输入下一步的命令。

continue（c） //继续执行，到下一个断点处（或运行结束）

next（n） //单步跟踪程序，当遇到函数调用时，直接调用，不进入此函数体；

step（s） //单步调试如果有函数调用，则进入函数；与命令n不同，n是不进入调用的函数的

until //运行程序直到退出循环体; / until+行号： 运行至某行

finish //运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

call 函数(参数) //调用“函数”，并传递“参数”，如：call gdb\_test(55)

quit( q ) //退出gdb

break n（b n） //在第n行处设置断点 ;可以带上代码路径和代码名称： b OAGUPDATE.cpp:578）

break func //在函数func()的入口处设置断点，如：break cb\_button

delete 断点号n //删除第n个断点

disable 断点号n //暂停第n个断点

enable 断点号n //开启第n个断点

clear 行号n //清除第n行的断点

info breakpoints（info b） //显示当前程序的断点设置情况

list（l） //列出程序的源代码，默认每次显示10行。

list 行号 //将显示当前文件以“行号”为中心的前后10行代码，如：list 12

list 函数名 //将显示“函数名”所在函数的源代码，如：list main

list //不带参数，将接着上一次 list 命令的，输出下边的内容。

print 表达式 (p ) //表达式”可以是任何当前正在被测试程序的有效表达式，包括数字，变量甚至是函数调用。

print a //将显示整数 a 的值

print ++a //将把 a 中的值加1,并显示出来

print name //将显示字符串 name 的值

print gdb\_test(22) //将以整数22作为参数调用 gdb\_test() 函数

print gdb\_test(a) //将以变量 a 作为参数调用 gdb\_test() 函数

display 表达式 //在单步运行时将非常有用，使用display命令设置一个表达式后，它将在每次单步进行指令后，紧接着输出被设置的表达式及值。如： display a

watch 表达式 //设置一个监视点，一旦被监视的“表达式”的值改变，gdb将强行终止正在被调试的程序。如： watch a

where/bt //当前运行的堆栈列表

set args 参数 //指定运行时的参数

show args //查看设置好的参数

info program //来查看程序的是否在运行，进程号，被暂停的原因

layout //用于分割窗口，可以一边查看代码，一边测试：

layout src //显示源代码窗口

layout asm //显示反汇编窗口

layout regs //显示源代码/反汇编和CPU寄存器窗口

layout split //显示源代码和反汇编窗口

Ctrl + L //刷新窗口

===============================================================================

break：断点

break 10 //设置本地行号，即list命令可见的行号

break test.c:10 //设置指定文件的行号

break main //函数，可以是自定义函数也可是库函数，如open

break test.c:main //指定文件中的函数

break system

break open

break \*0x40fe00 //设置地址为0x40fe00位置的断点：可通过info add function查看函数的地址

break 10 if index == 3 //当变量index为3时，设置断点为第10行

info break //查看断点信息

tbreak //跟break类似，只在断点停一次，过后会自动将断点删除

rbreak list\_\* //即在所有以 list\_ 为开头字符的函数地方都设置断点

delete 5 //删除断点5

===============================================================================

info

info add RTSPServer::RTSPClientSession::handleCmd\_SETUP //查看指定函数的地址： 0x40fe00

break \*0x40fe00 //设置地址为0x40fe00处为断点

list \*0x40fe00 //查看0x40fe00处的源码

directory 源代码路径 //加载源代码

===============================================================================

directory: 增加源文件路径

directory /home/yhong/src/live/live\_20161101/liveMedia

# Gdb cmd

|  |  |  |
| --- | --- | --- |
| **命令** | **解释** | **示例** |
| 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 gdb-sample |
| 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 |
| 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 |
| b <行号> b <函数名称> b \*<函数名称> b \*<代码地址>  d [编号] | b: 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。  d: 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 |
| s, n | 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； 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。  s 相当于其它调试器中的“Step Into (单步跟踪进入)”； n 相当于其它调试器中的“Step Over (单步跟踪)”。  这两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 (gdb) n |
| si, ni | si命令类似于s命令，ni命令类似于n命令。所不同的是，这两个命令（si/ni）所针对的是汇编指令，而s/n针对的是源代码。 | (gdb) si (gdb) ni |
| 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 |
| 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 |
| i | Info的简写，用于显示各类信息，详情请查阅“help i”。 | (gdb) i r |
| q | Quit的简写，退出GDB调试环境。 | (gdb) q |
| 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 |

# End