MySQL突然崩溃？教你用gdb解剖core文件，快速锁定“元凶”!

首页推荐

原创 szrsu 2025-03-13 743

gdb工具介绍

程序运行过程中可能会异常终止或崩溃，系统会把程序挂掉时的内存状态记录下来，写入core文件（core dump），可以通过gdb工具来分析core文件。GDB(GNU Debugger)是Linux下的一款C/C++程序调试工具，通过在命令行中执行相应的命令实现程序的调试。

GDB主要有以下功能：

- 设置断点
- 单步调试
- 查看变量的值
- 动态改变程序的执行环境
- 分析崩溃程序产生的core文件

安装gdb

```
# yum install gdb gcc -y
# gdb --version
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
```

测试程序

```
# vi test.c

#include<stdio.h>

int main(){
       int arr[4] = {1,2,3,4};
       int i=0;
       for(i=0;i<4;i++){
              printf("%d\n",arr[i]);
       }
       return 0;
}

###编译的时候带上-g,才能用gdb
# gcc -g test.c
# ./a.out
1
2
3
4
```

gdb用法

```
# gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.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 /root/a.out...done.
(gdb)

### r 命令，让程序跑起来
(gdb) r
Starting program: /root/a.out
1
2
3
4
[Inferior 1 (process 1656) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64
##quit退出
(gdb) quit
```

szrsu
LV.6
关注
79 文章
227 粉丝
398K+ 浏览量
获得了 2816 次点赞
内容获得 758 次评论
获得了 1675 次收藏

热门文章

MySQL中清除binlog的几种方法
2023-01-16 6836浏览

postgresql 主从切换过程
2023-01-07 6263浏览

postgresql 逻辑复制配置实战
2023-01-31 3272浏览

ORACLE RAC 扩容ASM磁盘组空间详细步骤
2024-05-15 2870浏览

openGauss 一主一备 从5.0 LTS 版本升级至 6.0 LTS 版本实战
2024-11-07 2508浏览

在线实训环境入口

MySQL在线实训环境
查看详情 »

最新文章

OGG Veridata 23c：实时数据比对利器，让异构数据库同步无忧！
2025-07-27 122浏览

别再折腾环境了，独家 Veridata 23c Docker 镜像，1分钟开启数据比对！
2025-07-22 90浏览

PostgreSQL性能分析神器，让你成为优化高手！
2025-07-09 206浏览

Oracle 自增列终极指南，三种用法一文打尽！
2025-06-17 394浏览

PG与Oracle的桥梁：手把手教你配置oracle_fdw实现跨库访问
2025-04-25 269浏览

目录

- gdb工具介绍
  - 安装gdb
  - 测试程序
  - gdb用法
  - gdb的骚操作
- gdb分析MySQL core文件
  - 系统打开coredump

gdb最常用的10个命令：

```
        break [file:]functiop
            Set a breakpoint at function (in file).
        ### b break 打断点
              函数的地方（函数名字）
              在第几行打断点
        run [arglist]
            Start your program (with arglist, if specified).
        ### run r 运行程序
        bt  Backtrace: display the program stack.

        print expr
            Display the value of an expression.
        ###print 打印变量
           arr[0]
           &arr[0]

        c   Continue running your program (after stopping, e.g. at a breakpoint).

        next   ###执行一条程序，不进入函数内部
            Execute next program line (after stopping); step over any function calls in the

        edit [file:]function
            look at the program line where it is presently stopped.

        list [file:]function   ###列出源代码的一部分(10行)
            type the text of the program in the vicinity of where it is presently stopped.

        step        ### step s进去某一个具体的函数调试
            Execute next program line (after stopping); step into any function calls in the

        help [name]
            Show information about GDB command name, or general information about using GDB.

        quit   ###退出gdb环境
            Exit from GDB.
```

打断点两种方式：

（1）函数的地方

（2）在第几行打断点

```
(gdb) b main        -->在这个main函数打断点
Breakpoint 1 at 0x400535: file test.c, line 4.
## list查看源代码
(gdb) list
1       #include<stdio.h>
2
3       int main(){
4               int arr[4] = {1,2,3,4};
5               int i=0;
6               for(i=0;i<4;i++){
7                       printf("%d\n",arr[i]);
8               }
9               return 0;
10      }
## 在上面的代码第7行打断点
(gdb) b 7
Breakpoint 2 at 0x400561: file test.c, line 7.
## 查看已设置的断点
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400535 in main at test.c:4  -->在第4行打了断点
2       breakpoint     keep y   0x0000000000400561 in main at test.c:7  -->在第7行打了断点
```

运行程序：

```
(gdb) r
Starting program: /root/a.out

Breakpoint 1, main () at test.c:4
4               int arr[4] = {1,2,3,4};
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64
(gdb) n                    -->往下走一行
5               int i=0;
(gdb) p arr[0]   -->打印变量的值
$1 = 1
(gdb) p &arr[0]  -->打印变量的arr[0]的地址
$3 = (int *) 0x7fffffffe4b0

(gdb) p arr[1]
$2 = 2
(gdb) p &arr[1]
$4 = (int *) 0x7fffffffe4b4 -->打印变量的arr[1]的地址，发现比arr[0]多了4个字节

(gdb) n
6               for(i=0;i<4;i++){
(gdb) n

Breakpoint 2, main () at test.c:7
7                       printf("%d\n",arr[i]);
```

-步进

```
# cp test.c test1.c
# vi test1.c
#include<stdio.h>

void hello(){
        printf("hello echo~ \n");
}
int main(){
        int arr[4] = {1,2,3,4};
        int i=0;
        for(i=0;i<4;i++){
                printf("%d\n",arr[i]);
        }
        hello();
        return 0;
}
# gcc -g test1.c
# gdb ./a.out
(gdb) list
1       #include<stdio.h>
2
3       void hello(){
4               printf("hello echo~ \n");
5       }
6       int main(){
7               int arr[4] = {1,2,3,4};
8               int i=0;
9               for(i=0;i<4;i++){
10                      printf("%d\n",arr[i]);
(gdb) list
11              }
12              hello();
13              return 0;
14      }
##一次显示不完整，可以继续list
##在第12行打一个断点
(gdb) b 12
Breakpoint 1 at 0x4005e5: file test1.c, line 12.
##打完断点，让程序跑起来
(gdb) r
Starting program: /root/./a.out
1
2
3
4

Breakpoint 1, main () at test1.c:12
12              hello();
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64
##用s命令，进入函数调用
(gdb) s
hello () at test1.c:4
4               printf("hello echo~ \n");
(gdb) n
hello echo~
5       }
(gdb) n
main () at test1.c:13
13              return 0;
(gdb) n
14      }
(gdb) n
0x00007ffff7a2f555 in __libc_start_main () from /lib64/libc.so.6
```

gdb的骚操作

（1）通过shell命令调用系统命令

```
(gdb) shell ls
test1.c test.c a.out
(gdb) shell cat test.c
#include<stdio.h>

int main(){
	int arr[4] = {1,2,3,4};
	int i=0;
	for(i=0;i<4;i++){
		printf("%d\n",arr[i]);
	}
	return 0;
}
```

（2）日志记录功能

```
(gdb) set logging on
Copying output to gdb.txt.
(gdb) list
1	#include<stdio.h>
2
3	void hello(){
4		printf("hello echo~ \n");
5	}
6	int main(){
7		int arr[4] = {1,2,3,4};
8		int i=0;
9		for(i=0;i<4;i++){
10			printf("%d\n",arr[i]);
(gdb) list
11		}
12		hello();
13		return 0;
14	}
(gdb) b 12
Breakpoint 1 at 0x4005e5: file test1.c, line 12.
(gdb) r
Starting program: /root/a.out
1
2
3
4

Breakpoint 1, main () at test1.c:12
12		hello();
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64
(gdb) s
hello () at test1.c:4
4		printf("hello echo~ \n");
(gdb) n
hello echo~
5	}
(gdb) n
main () at test1.c:13
13		return 0;
(gdb) n
14	}
(gdb) n
0x00007ffff7a2f555 in __libc_start_main () from /lib64/libc.so.6
(gdb) quit

# cat gdb.txt
```

（3）watchpoint：观察变量是否发现变化

info watchpoints：查看watchpoint

```
(gdb) list
1       #include<stdio.h>
2
3       void hello(){
4               printf("hello echo~ \n");
5       }
6       int main(){
7               int arr[4] = {1,2,3,4};
8               int i=0;
9               for(i=0;i<4;i++){
10                      printf("%d\n",arr[i]);
(gdb) b 8
Breakpoint 1 at 0x4005b1: file test1.c, line 8.
(gdb) r
Starting program: /root/a.out

Breakpoint 1, main () at test1.c:8
8               int i=0;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64
(gdb) print &i
$2 = (int *) 0x7fffffffe4cc
(gdb) watch *0x7fffffffe4cc
Hardware watchpoint 2: *0x7fffffffe4cc  -->设置了一个观察点
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      *0x7fffffffe4cc
(gdb) n
9               for(i=0;i<4;i++){
(gdb) n
10                      printf("%d\n",arr[i]);
(gdb) n
1
9               for(i=0;i<4;i++){
(gdb) n
Hardware watchpoint 2: *0x7fffffffe4cc    -->发现变量发生了变化

Old value = 0
New value = 1
0x00000000004005df in main () at test1.c:9
9               for(i=0;i<4;i++){
```

gdb分析MySQL core文件

core 文件是进程在崩溃时的内存快照，可以用于故障排查和调试。

分析core文件的方法：gdb 二进制文件 core文件

系统打开coredump

####如果core文件没有生成，那么需要查看ulimit限制

```
###没有生成core文件，要通过ulimit命令进行设置
# ulimit -a
core file size          (blocks, -c) 0   -->设置为0不会生成
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31117
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31117
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
####开启core文件
# ulimit -c unlimited
# ulimit -a
core file size          (blocks, -c) unlimited

####配置core路径
echo "1" > /proc/sys/kernel/core_uses_pid
echo 2 > /proc/sys/fs/suid_dumpable
mkdir  /data/core && chmod 777 /data/core && echo "/data/core/%e.core.%p" > /proc/sys/kerne

###为了下次系统重启时可以生效，写入配置文件
echo "mysql   -   core   unlimited" >> /etc/security/limits.conf
在/etc/sysctl.conf中，增加配置：
kernel.core_uses_pid = 1
fs.suid_dumpable = 2
kernel.core_pattern=/data/core/%e.core.%p

当core产生时，会在/data/core生成名字为： 程序名+core+PID号
```

模拟MySQL进程异常

#mysql配置core，配置完重启生效

```
vi /etc/my.cnf
[mysqld]
core_file
```

#模拟异常

```
[root@mysql80:/data/core]# kill -SEGV `pidof mysqld`

[root@mysql80:/data/core]# ls -lh
total 1.2G
-rw------- 1 mysql mysql 2.5G Mar 12 23:57 mysqld.core.2996
```

解析core文件

gdb加载core文件，使用bt命令查看堆栈回溯，以诊断崩溃原因：

```
[root@mysql80:/data/core]# gdb /usr/local/mysql/bin/mysqld mysqld.core.2996
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.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 /usr/local/mysql-8.0.27-linux-glibc2.12-x86_64/bin/mysqld...Dwarf Erro
(no debugging symbols found)...done.
[New LWP 1721]
[New LWP 1727]
[New LWP 1726]
[New LWP 1728]
[New LWP 1725]
[New LWP 1729]
[New LWP 1730]
[New LWP 1731]
[New LWP 1732]
[New LWP 1733]
[New LWP 1740]
[New LWP 1734]
[New LWP 1735]
[New LWP 1798]
[New LWP 1736]
[New LWP 1797]
[New LWP 1737]
[New LWP 1796]
[New LWP 1738]
[New LWP 1739]
[New LWP 1795]
[New LWP 1794]
[New LWP 1741]
[New LWP 1742]
[New LWP 1793]
[New LWP 1792]
[New LWP 1763]
[New LWP 1791]
[New LWP 1764]
[New LWP 1790]
[New LWP 1769]
[New LWP 1784]
[New LWP 1770]
[New LWP 1783]
[New LWP 1771]
[New LWP 1779]
[New LWP 1773]
[New LWP 1778]
[New LWP 1777]
[New LWP 1774]
[New LWP 1766]
[New LWP 1775]
[New LWP 1765]
[New LWP 1776]
[New LWP 1762]
[New LWP 1785]
[New LWP 1760]
[New LWP 1759]
[New LWP 1786]
[New LWP 1787]
[New LWP 1788]
[New LWP 1799]
[New LWP 1754]
[New LWP 1753]
[New LWP 1756]
[New LWP 1744]
[New LWP 1746]
[New LWP 1747]
```

```
[New LWP 1750]
[New LWP 1751]
[New LWP 1752]
[New LWP 1755]
[New LWP 1757]
[New LWP 1758]
[New LWP 1748]
[New LWP 1745]
[New LWP 1749]
[New LWP 1743]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Dwarf Error: wrong version in compilation unit header (is 0, should be 2, 3, or 4) [in modu
Core was generated by `/usr/local/mysql/bin/mysqld --defaults-file=/data/3306/my.cnf --user
Program terminated with signal 11, Segmentation fault.
#0  0x00007f0c89a42aa1 in pthread_kill () from /lib64/libpthread.so.0
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.3.x86_64 libaio-0.

### 多个命令都可以显示程序的调用轨迹： bt, where, info stack, backtrace

(gdb) bt
#0  0x00007f0c89a42aa1 in pthread_kill () from /lib64/libpthread.so.0
#1  0x000000000102909d in handle_fatal_signal ()
#2  <signal handler called>
#3  0x00007f0c87d4dddd in poll () from /lib64/libc.so.6
#4  0x0000000000101dd38 in Mysqld_socket_listener::listen_for_connection_event() ()
#5  0x0000000000df44a9 in mysqld_main(int, char**) ()
#6  0x00007f0c87c7c555 in __libc_start_main () from /lib64/libc.so.6
#7  0x0000000000dd68a5 in _start ()
或
(gdb) where
#0  0x00007f0c89a42aa1 in pthread_kill () from /lib64/libpthread.so.0
#1  0x000000000102909d in handle_fatal_signal ()
#2  <signal handler called>
#3  0x00007f0c87d4dddd in poll () from /lib64/libc.so.6
#4  0x0000000000101dd38 in Mysqld_socket_listener::listen_for_connection_event() ()
#5  0x0000000000df44a9 in mysqld_main(int, char**) ()
#6  0x00007f0c87c7c555 in __libc_start_main () from /lib64/libc.so.6
#7  0x0000000000dd68a5 in _start ()
```

从下往上查看这些信息，我们可以分析出导致崩溃的函数和具体位置，进而进行更深层次的排查和调试。

有时候，需要查看所有线程，才能排查问题，info threads查看所有线程

```
(gdb) info threads
  Id   Target Id         Frame
  68   Thread 0x7f82fb39b700 (LWP 3003) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  67   Thread 0x7f82f27fc700 (LWP 3007) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  66   Thread 0x7f82f1ffb700 (LWP 3008) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  65   Thread 0x7f82f17fa700 (LWP 3009) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  64   Thread 0x7f82aa276700 (LWP 3012) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  63   Thread 0x7f82a9a75700 (LWP 3013) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  62   Thread 0x7f82aaa77700 (LWP 3011) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  61   Thread 0x7f82a9274700 (LWP 3014) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  60   Thread 0x7f82a8a73700 (LWP 3015) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  59   Thread 0x7f82a8272700 (LWP 3016) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  58   Thread 0x7f82a7a71700 (LWP 3017) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  57   Thread 0x7f82a7270700 (LWP 3018) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  56   Thread 0x7f82a6a6f700 (LWP 3019) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  55   Thread 0x7f82a526c700 (LWP 3022) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  54   Thread 0x7f82a426a700 (LWP 3024) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  53   Thread 0x7f82a3a69700 (LWP 3025) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  52   Thread 0x7f82a2a67700 (LWP 3027) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  51   Thread 0x7f82a2266700 (LWP 3028) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  50   Thread 0x7f82a1a65700 (LWP 3029) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  49   Thread 0x7f82a1264700 (LWP 3030) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  48   Thread 0x7f829f260700 (LWP 3034) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
  47   Thread 0x7f829ea5f700 (LWP 3035) 0x00007f832cc1e644 in __io_getevents_0_4 () from /l
```

gdb用来抽丝剥茧某些疑难case的时候非常有用，甚至gdb可以在紧急情况下救你一命，例如，当MySQL数据库连接打满又没有后台线程可以连接到MySQL的时候，你可以通过gdb来修改MySQL的连接数：
gdb -p $(pidof mysqld) -ex “set max_connections=2000” -batch

总结

通过以上步骤，我们可以有效地分析 MySQL 产生的 Core 文件，找出程序崩溃的根本原因。理解这些步骤对于数据库管理员和开发人员是非常重要的，有助于迅速定位问题。

墨力计划 mysql

最后修改时间：2025-03-13 09:36:27

评论

分享你的看法，一起交流吧~

鲁鲁 LV.6

MySQL突然崩溃？教你用gdb解剖core文件，快速锁定“元凶”!

4月前 点赞 评论

芃芃 LV.4

MySQL突然崩溃？教你用gdb解剖core文件，快速锁定“元凶”!

4月前 点赞 评论

相关阅读

ACDU周度精选 | 本周数据库圈热点 + 技术干货分享（2025/7/25期）
墨天轮小助手 470次阅读 2025-07-25 15:54:18

ACDU周度精选 | 本周数据库圈热点 + 技术干货分享（2025/7/17期）
墨天轮小助手 436次阅读 2025-07-17 15:31:18

墨天轮「实操看我的」数据库主题征文活动启动
墨天轮编辑部 379次阅读 2025-07-22 16:11:27

深度解析MySQL的半连接转换
听见风的声音 205次阅读 2025-07-14 10:23:00

MySQL 9.4.0 正式发布，支持 RHEL 10 和 Oracle Linux 10
严少安 199次阅读 2025-07-23 01:21:32

索引条件下推和分区-一条SQL语句执行计划的分析
听见风的声音 197次阅读 2025-07-23 09:22:58

null和子查询--not in和not exists怎么选择？
听见风的声音 182次阅读 2025-07-21 08:54:19

MySQL数据库SQL优化案例(走错索引)
陈举超 166次阅读 2025-07-17 21:24:40

使用 MySQL Clone 插件为MGR集群添加节点
黄山谷 163次阅读 2025-07-23 22:04:19

MySQL 8.0.40：字符集革命、窗口函数效能与DDL原子性实践
shunwah 141次阅读 2025-07-15 15:27:19