



第九届中国数据库技术大会
DATABASE TECHNOLOGY CONFERENCE CHINA 2018

使用动态跟踪技术SystemTap 监控MySQL、Oracle性能

吕海波 (VAGE)

DTCC
2018

2018.05.10 - 12 北京国际会议中心



IT168.com

ChinaUnix

ITPUB



第九届中国数据库技术大会
DATABASE TECHNOLOGY CONFERENCE CHINA 2018

如何不

《读源码从入门到放弃》

For MySQL 5.7

吕海波 (VAGE)

DTCC
2018

2018.05.10 - 12 北京国际会议中心



IT168.com

ChinaUnix

ITPUB

Systemtap介绍：动态跟踪技术

What was really happening

“Report Regarding Comprehensive Reorganization and Standardization of the Company Export Control Related Matters”

[illegible]

ZTE		Name: <u>Wang Chongsheng</u>	
ZTE Corporation Document Submitted for Ratification (Review) Form			
Title of the Document submitted for ratification (review)	Report Regarding Comprehensive Recognition and Study Situation of the Company Export Control Related Matters		Document type: <input checked="" type="checkbox"/> New document <input type="checkbox"/> Company system <input type="checkbox"/> Other document
Attachment	- Report Regarding Comprehensive Recognition and Study Situation of the Company Export Control Related Matters -		<input type="checkbox"/> Original document <input checked="" type="checkbox"/> Replicate document
Drawn by	Legal Department	Drawn by	GUO, Pengjun
Contact phone	010-11717887	Classification	Department suggestion: For higher authorities to <input checked="" type="checkbox"/> verify <input type="checkbox"/> audit
List of people who can read the document (apply to secret and above document)			
Director of Legal Department, corporate leaders			
Scope of the document:			
Is our company from abroad have greater impact in recent years, so that the company's export control rules following Director TIAN and Director GUO's instructions, our company's export control rules is completely local export control project team to study, handle, and report to the company's current export control rules.			
Need revision stage: <input checked="" type="checkbox"/> Yes <input type="checkbox"/> No		Compliance to apply to documents that need revision stage:	
Responsible department's opinion		<input checked="" type="checkbox"/> Agree. Subject to study verification for approval. <input type="checkbox"/> Disagree. Reasoning (signature) _____ Date: 2012.8.29 For higher authorities to <input checked="" type="checkbox"/> verify <input type="checkbox"/> audit	
Company leader's opinion (Director GUO)		<input checked="" type="checkbox"/> Agree. [GUO's signature] _____ Date: 2012.8.29 For higher authorities to <input checked="" type="checkbox"/> verify <input type="checkbox"/> audit	
Company leader's opinion (Director TIAN)		<input checked="" type="checkbox"/> Agree. TIAN, Wangjun (signature) _____ Date: 2012.8.29 For higher authorities to <input checked="" type="checkbox"/> verify <input type="checkbox"/> audit	
CEO review/approval		<input checked="" type="checkbox"/> Agree. SHI, Liang (signature) _____ Date: 2012.8.29	

©All information is property of ZTE. No external dissemination.
 All Rights reserved. All spreading allowed without permission of ZTE

Page 3

Translation



U.S. DEPARTMENT OF COMMERCE • BUREAU OF INDUSTRY AND SECURITY



数领先机·智赢未来 (9)



DB2

BigDate

100

SQL Serv

Systemtap介绍：动态跟踪技术

近日，美国禁止向中兴技术出口。这将进一步激发国内基础技术研究的高潮。OS、数据库……，都是重要的基础技术。

于是，有朋友跟我说，想研究下MySQL的源码。但是，代码量太大了，不知道从哪里入手，就放弃了。

还有朋友说，我已经编译成功了，就差看了。。。然后，然后就没有下文了。

[illegible]

ZTE		Secret <i>Highly Confidential</i>	
ZTE Corporation Document Submitted for Ratification (Review) Form			
Title of the Document submitted for certification (review)	Report Regarding Comprehensive Investigation and Standardization of the Company Export Control Related Matters		Document type: <input checked="" type="checkbox"/> Bill-level document <input type="checkbox"/> Company system <input type="checkbox"/> Other document
Attachment	<input checked="" type="checkbox"/> Report Regarding Comprehensive Investigation and Standardization of the Company Export Control Related Matters <input type="checkbox"/>		<input checked="" type="checkbox"/> Urgent document <input type="checkbox"/> Regular document
Drafted by	Legal Department 13731147381	Draftsman GONG, Jiaojiao	Date August 25, 2021
Contact phone	Classification Document designated for higher authorities to see early		Document designated for higher authorities to see early <input checked="" type="checkbox"/> Yes <input type="checkbox"/> No
List of people who can read the document (apply to secret and/or designated documents): Director of Legal Department, company leaders Managers of the document On cross-border business has general rapidly in recent years, to help the company control risks, strengthen Chinese TIAN and Director GUC's interaction, our conditions (failed sufficiently), form a company-level export control project team to study, analyze, and respond to legal company's control export control matters.			
Need counter-sign: <input type="checkbox"/> Yes <input checked="" type="checkbox"/> No		Counter-signs to apply to documents that need counter-sign: Agree: Submit to higher authorities for approval	
Responsible department's opinion	GUC (Legal Department) <input checked="" type="checkbox"/> Agree 2021.8.22 For higher authorities to see early <input checked="" type="checkbox"/> read		
Company leader's opinion (Director GUC)	Agree: (GUC)'s signature <input checked="" type="checkbox"/> Agree 2021.8.22 For parties to see early <input checked="" type="checkbox"/> read		
Company leader's opinion (Director TIAN)	Agree: TIAN, Wang (Importance) <input checked="" type="checkbox"/> Agree 2021.8.22 For higher authorities to see early <input checked="" type="checkbox"/> read		
CEO review/approval	Agree: SHI, Long (Importance) <input checked="" type="checkbox"/> Agree 2020/11		

*All information is property of ZTE. No content disseminated.
 All Rights reserved. No Spreading without Permission of ZTE

Page 1

Systemtap介绍：动态跟踪技术

为什么……

因为没有建立一个正向反馈机制。

你花很多精力研究源码，
并没有什么用。

你已经看了一个月代码，

再坚持一个月。

也没什么不同。

于是放弃吧。



不要再说了，
都是有原因的，我懂！

我来过，我看过。”

Systemtap介绍：动态跟踪技术

如果我研究个一、两周，就能有点用处，说不定我也可以坚持下去，不断积累，集小胜为大胜，最终成为技术大牛，走向人生巅峰，迎娶白富美（这是按原来的说法，中兴被美国封锁后，现在改为“最终成为技术大牛，为科技兴国贡献力量”）。

源码看个一、两周，就能有点小用途，这就是正向反馈，它会激励人不断深入下去。

如何建立正向反馈呢？

Systemtap

Systemtap介绍：动态跟踪技术

➤ 什么是动态跟踪技术：

又称：动态调试，Dynamic Tracing。不影响程序的情况下，动态的观察程序运行。

➤ 常见的动态追踪技术：

Dtrace: solaris

SystemTap: Linux

.....

➤ Systemtap基础

基本的开发基础：变量、常量、条件、分枝、循环、子程序或子函数、数组、指针与地址。

Systemtap介绍：动态跟踪技术

➤ 重要概念：探针，probe

动态跟踪技术的基础，相当于数据库的触发器，探针是操作系统的触发器。我们可以利用systemtap，将我们的脚本动态的插入内存。等到一定的条件被解发，脚本就会被操作系统执行。

通常脚本都是显示个内存状态、统计时间等。

Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap
2
3 global tm;
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```



Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap      //定义解释器，类似Shell脚本
2
3 global tm;
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```

Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap    //定义解释器，类似Shell脚本
2
3 global tm; //全局变量，计录时间。
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```

Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap    //定义解释器，类似Shell脚本
2
3 global tm; //全局变量，计录时间。
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```

Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap    //定义解释器，
2
3 global tm; //全局变量，计录时间。
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
```

探针：进入函数
buf_page_get_gen入口点时触发。
相当于before buf_page_get_gen触
发器。

```
9 probe process("/data/db/mysql/lib/bin/mysqld").function
   ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
   ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```

Systemtap介绍：动态跟踪技术

比如，已知MySQL的逻辑读操作，从buf_page_get_gen函数开始，到buf_page_release_latch函数为止。我们来写个统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap      //定义解释器，类似Shell脚本
2
3 global tm; //全局变量，计录时间。
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_get_gen") {
10     tm=gettimeofday_us();
11 }
12
13 probe process("/data/db/mysql/lib/bin/mysqld").function
    ("buf_page_release_latch").return {
14     printf("Time:%d\n", gettimeofday_us()-tm );
15 }
```

探针：退出函数
buf_page_release_latch入口点时触发。相当于after
buf_page_release_latch触发器。

Systemtap介绍：动态跟踪技术

```
[root@VAGE01 ~]#  
[root@VAGE01 ~]# ./lgr1.stp -x 3164  
Begin.  
Time:63
```

```
mysql>  
mysql>  
mysql>  
mysql> select * from t4 where id=1;  
+----+-----+  
| id | c1                |  
+----+-----+  
|  1 | BBBBBBaaaaaaaabbbb |  
+----+-----+  
1 row in set (0.01 sec)
```

在我的测试机上，MySQL完成一次主键索引逻辑读，耗时0.063毫秒。

➤ Oracle逻辑读方式

再来看看Oracle的逻辑读耗时吧。Oracle的逻辑读有好多种，我们试一种最普通的吧，以rowid方式访问读。这种读与MySQL中的逻辑读非常不同，但Oracle的DBA一定耳熟能详，就是先获得CBC Latch，在CBC Latch的保护下，获得块的Buffer Pin Lock。在Buffer Pin Lock保护下，读块中行数据。从获得CBC Latch始，作为逻辑读的开始。到再次获得CBC Latch，释放Buffer Pin Lock为止，为一次逻辑读的结束。

➤ 对应函数：

- kcbgtcr: 获得CBC Latch，在CBC Latch保护下，获得块的Buffer Pin Lock。对应MySQL的buf_page_get_gen函数。
- kcbrls: 再次获得CBC Latch，并在CBC Latch保护下，释放Buffer Pin Lock。对应MySQL的buf_page_release_latch函数。



Systemtap介绍：动态跟踪技术

有了函数名，可以很容易写出Oracle统计逻辑读耗时的systemtap脚本。

```
1 #!/usr/bin/stap
2
3 global tm;
4
5 probe begin {
6     printf("Begin.\n");
7 }
8
9 probe process("/opt/oracle/product/11204/bin/oracle").function("kcbgtcr") {
10     tm = gettimeofday_us();
11 }
12
13 probe process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").return {
14     printf("Time:%d\n", gettimeofday_us() - tm );
15 }
```

Systemtap介绍：动态跟踪技术

```
[root@VAGE01 ~]#  
[root@VAGE01 ~]# ./lgr_o.stp -x  
2630  
Begin.  
Time:31
```

```
SQL>  
SQL> select * from t1 where  
rowid='AAADs6AAEAAAACRAAB';
```

ID	C1	C2
1	aaaaaaaaa	bbbbbbbbbbbbbb

```
SQL>
```

Oracle完成一次Rowid逻辑读，耗时0.031毫秒。

MySQL一次逻辑读用了0.065毫秒，比Oracle慢了一倍，因为MySQL的逻辑读中包含了索引对比。

Systemtap介绍：动态跟踪技术

脚本很简单，可以满足我们一个小好奇心。

问题是，如何得知对应某操作的函数名呢？

如何得知MySQL逻辑读，是从buf_page_get_gen函数开始，到buf_page_release_latch函数为止？

这就需要从源码中发掘了。

按照后面我所介绍的方法，估计找到这两个函数需要个一、两天左右时间（如果是刚开始搞，可要久一点）。

也就是说，一、两天左右的努力，可以写一个小脚本，统计某项性能指标、输出某些状态标志。让几天的努力钻研，有可用武之地。



Systemtap介绍：动态跟踪技术

这就是前面所说的正向反馈。读个几天代码，搞点小脚本，监控一下性能、分析一下消耗等等。再读个几天代码，继续搞小脚本……。而且，很多时候，脚本和源码，可以互相印证、互相促进，这样不断正向反馈，不断进步。

MySQL有那么多需要我们去监控性能的地方吗？

Oracle 11G有1367个等待事件和一千多个性能资料。Oracle共计用近3000个指标，反应性能、状态。其他数据库相比之下弱了很多，这给systemtap这样的动态跟踪技术，留下了巨大的空间。

阅读源码，不再是无用的屠龙之技，你可以不断在源码中发掘，利用动态跟踪技术积累脚本，既能得到成长，没准这些脚本在工作上可以帮上大忙。



时间都去哪儿了

下面来看“时间都去哪儿了”的脚本。

脚本很简单，就是一个统计DML执行时，各阶段时间消耗的脚本。关键还是脚本后面隐藏的一系列方法。

写这个脚本的初衷，是为了统计一个大加载操作的时间消耗。因为前一段时间，有客户的导入数据操作，加载百万级的数据量，每天进行一次，耗时十分钟左右，DBA诊断后，根据他的环境，判断这是一个正常的耗时。感觉慢是因为没有并行，单个线程的吞吐有限。另外，Innodb引擎的事务、恢复机制，也会消耗一定时间（根据应用，表其实并不需要事务）。但客户始终觉得不应该这样慢，但又不知道时间消耗到哪儿了，于是，就有了这个systemtap脚本。

时间都去哪儿了

一条SQL的执行，总是分阶段的，有解析、执行、抓取等等阶段，我们来统计如下阶段的时间消耗：

1. 解析
2. 处理Binlog
3. 处理UNDO
4. 处理Redo
5. 总执行时间

这些阶段当然不够细，只是粗略统计下，以介绍方法为主。

掌握方法后，你可以根据需要，调节脚本，更细分、或只针对某些事件统计。

下面，我们以Binlog为例，讲解一下如何使用systemtap脚本+源码，统计DML执行期间，处理Binlog所消耗的时间。

时间都去哪儿了

如何开始？

Easy！先找到SQL执行过程中，处理Binlog的函数。然后，就能像前面统计逻辑读耗时一样，统计Binlog的时间消耗。这里说的简单，是systemtap脚本简单，寻找binlog函数，就稍有点复杂了。

如何寻找处理Binlog的函数？

重要技能：逆流而上，顺腾摸瓜。

“流”：内存数据流。

以Binlog为例，Binlog数据最终一定要写进磁盘的。写磁盘时候，一定是从某一个Buffer写磁盘，我们称这个Buffer为Buffer Z。那么，Buffer Z的数据，又是从那里来的呢？

假设Buffer Z的数据是从Buffer Y中拷贝过去的，我们称Buffer Z的上线为Buffer Y。那么，Buffer Y的上线呢？

只要顺腾摸瓜，每找到一个上线，观察一下它的代码，看是不是我们要找的Binlog处理函数，如此，即可。

时间都去哪儿了

具体过程:

要先找到写binlog到文件的函数。

这个简单, 使用systemtap, 显示所有名称中包含write字样的函数。脚本如下:

```
1 #!/usr/bin/stap
2
3 probe Ulong_arg(1): 函数的第一参数。
4     pr 131: Binlog文件的文件号。可
5 }      以从/proc/进程号/fd中得到。
6
7 probe process(/data/db/mysql/lib/bin,
8     if ( ulong_arg(1) == 131 ) {
9         printf("%s %x %x %x\n", pp(), ulong_arg(1), ulong_arg(2),
ulong_arg(3));
10         print_ubacktrace();
11         printf("-----")
12     }
13 }
```

write: 跟踪所有名称中包含write字样的函数。

pp(): 跟踪点的信息。
ulong_arg(n): 参数

显示调用堆栈。即跟踪点的上层、上上层、.....函数。

时间都去哪儿了

```
[root@VAGE01 ~]#  
[root@VAGE01 ~]# ./mio.stp -x 3164  
Begin.  
process("/data/db/mysql/lib/bin/mysqld").function("my_write@/home/mysql/mysql-  
5.7.19/mysys/my_write.c:40") 83 38114b0 117  
0x... : my_write+0x0/0x41c  
0x... : inline_mysql_file_write+0x98/0x120  
0x... : my_b_flush_io_cache+0x2f3/0x47f  
0x... :  
_ZN13MYSQL_BIN_LOG19flush_cache_to_fileEPy+0x2  
7/0x8c  
0x...
```

```
mysql>  
mysql> insert into t1 values(43,'43aaaaabbbbb',  
'AAAAAAAAA');  
Query OK, 1 row affected (7 min 37.49  
sec)  
  
mysql> commit;  
Query OK, 0 rows affected (0.01 sec)  
  
mysql>
```

时间都去哪儿了

```
[root@VAGE01 ~]# ./mio .stp -x 3164
```

```
Begin.
```

```
process("/data/db/mysql/lib/bin/mysqld").function("my_write@/home/mysql/mysql-5.7.19/mysys/my_write.c:40") 83 38114b0 117
```

```
0x... : my_write+0x0/0x41c
```

```
0x... : inline_mysql_file_write+0x120
```

```
0x... : my_b_flush_io_cache+0x17c
```

```
0x... : _ZN13MYSQL_BIN_
```

```
0x... : _ZN13MYSQL_BIN_
```

```
0x... : _ZN13MYSQL_BIN_
```

```
o o o o o
```

```
0x... : _Z11mysql_parse
```

```
0x... : _Z10do_commandP
```

```
0x... : handle_connection+0x1e0/0x2d4
```

```
0x... : pfs_spawn_thread+0x170/0x177
```

83: 十六进制的文件号，十进制为131。

38114b0: 前文中的Buffer Z。

117: 十六进制的长度。一共279字节。

my_write函数：
写Binlog的函数。

时间都去哪儿了

my_write.c中的写I/O函数:

```
40 size_t my_write(File Filedes, const uchar *Buffer, size_t Count,
myf MyFlags)
41 {
42     size_t writtenbytes;
.....
57 for (;;)
58     {
59         errno= 0;
60 #ifdef _WIN32
61         writtenbytes= my_win_write(Filedes, Buffer, Count);
62 #else
63         writtenbytes= write(Filedes, Buffer , Count);
64 #endif
.....
128     DBUG_RETURN(sum_written);
129 } /* my_write */
```

从这里，库函数write将Binlog数据写进磁盘。下面，我们开始找它的上线，Buffer Z。

时间都去哪儿了

➤ 准备工作

首先，要准备一条SQL，SQL一定要有标识度：

```
insert into t1 values(42, 'aaaaaaaa', 'AAAAAAAAA');
```

有大量的重复数据，容易辨认。

时间都去哪儿了

```
[root@VAGE01 ~]# gdb -p 3164
GNU gdb (GDB) Red Hat
Enterprise Linux 7.6.1-51.el7
.....
(gdb)
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb) c
```

```
mysql>
mysql>
mysql>
mysql> insert into t1 values(42,
'aaaaaaaaa', 'AAAAAAAAA');
===>HANG住<===
```


时间都去哪儿了

```
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb)C
Continuing.
```

```
mysql>
mysql>
mysql>
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK, 1 row affected (7 min
37.49 sec)

mysql>
```

```
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb)C
Continuing.
Breakpoint 1, my_write
(Filedes=131, Buffer=0x3810a40
"\362\337\275Z\"q\004",
Count=284, MyFlags=52) at
/home/mysql/mysql-
5.7.19/mysys/my_write.c:63
63  writtenbytes= write(Filedes,
Buffer, Count);
(gdb)
```

```
mysql>
mysql>
mysql>
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK, 1 row affected (7 min
37.49 sec)

mysql> commit;
===>HANG住<===
```

```
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb)C
Continuing.
Breakpoint 1, my_write
(Filedes=131, Buffer=0x3810a40
"\362\337\275Z\"d\004",
Count=284, MyFlags=52) at
/home/mysql/mysql-
5.7.19/mysys/my_write.c:63
63 writtenbytes= write(Filedes,
Buffer, Count);
(gdb)
```

```
mysql>
mysql>
mysql>
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK, 1 row affected (7 min
37.49 sec)

mysql> commit;
===>HANG住<===
```

```
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb)C
Continuing.
Breakpoint 1, my_write
(Filedes=131, Buffer=0x3810a40
"\362\337\275Z\"d\004",
Count=284, MyFlags=52) at
/home/mysql/mysql-
5.7.19/mysys/my_write.c:63
63 writtenbytes= write(Filedes,
Buffer, Count);
(gdb)
```

Filedes : 文件号

Buffer : 缓存

Count : 写长度

```
(gdb) b my_write.c:63
Breakpoint 1 at 0x1877865:
file /home/mysql/mysql-
5.7.19/mysys/my_write.c,
line 63.
(gdb)C
Continuing.
Breakpoint 1, my_write
(Filedes=131, Buffer=0x3810a40
"\362\337\275Z\"d\004",
Count=284, MyFlags=52) at
/home/mysql/mysql-
5.7.19/mysys/my_write.c:63
63  writtenbytes= write(Filedes,
Buffer, Count);
(gdb) p Filedes
$1 = 131
```

```
[root@VAGE01 fd]# pwd
/proc/3164/fd
[root@VAGE01 fd]# ls -lFrt|grep 131
l-wx-----. 1 mysql mysql 64 Mar
30 09:27 131 ->
/data/db/mysql/binlog.000048
```

```
(gdb)C
Continuing.
Breakpoint 1, . . . . .
63 writtenbytes= write(Filedes,
Buffer, Count);
(gdb) p Filedes
$1 = 130
(gdb) p Buffer
$2 = (const uchar *) 0x3810a40
(gdb)
```

```
[root@VAGE01 fd]# pwd
/proc/7792/fd
[root@VAGE01 fd]# ls -lFrt|grep 130
l-wx-----. 1 mysql mysql 64 Mar
30 09:27 131 ->
/data/db/mysql/binlog.000048
```

```
(gdb)C
Continuing.
Breakpoint 1, . . . . .
63 writtenbytes = write(fd, buf, count);
```

```
(gdb) p Filedes
```

```
$1 = 130
```

```
(gdb) p Buffer
```

```
$2 = (const uchar *)
```

```
(gdb) x/80x 0x3810a40:
```

```
0x3810a40:
```

```
0x3810ae0:
```

```
0x3810af0:
```

```
0x3810b00:
```

```
0x3810b10:
```

```
0x3810b20:
```

```
0x3810b30:
```

```
0x3810b40:
```

```
0x00000000
```

```
0x00000000
```

```
0xca1e0000
```

```
0x00000000
```

```
0x61080000
```

```
0x41414141
```

```
0xa0ca105a
```

0x61080000

||

0x00000861

```
004100
```

```
0x000d0e00
```

```
00317473
```

```
0x00317402
```

```
0x3c003c04
```

```
0xf3e60700
```

```
0xdbdff85f
```

```
0x3f0004a0
```

```
0xca000000
```

```
0x0000000d
```

```
0x01000000
```

```
0x03000200
```

```
0x002bf8ff
```

```
0x61616161
```

```
0x08616161
```

```
0x41414141
```

```
0xbddff2e8
```

```
0x1ca5dc41
```

```
0xbddff2e8
```

```
0x001f0004
```

```
0x0de90000
```

```
0x00000000
```


(gdb)C
Continuing.
Breakpoint 1,
63 written by

(gdb) p Filed

\$1 = 130

(gdb) p Buffer

\$2 = (const u

(gdb) x/80x 0

0x3810a40:

0x3810ae0:

0x3810af0:

0x3810b00:

0x3810b10:

0x3810b20:

0x3810b30:

0x3810b40:

0x0100

0x0f0f

0xca1e

0x0000

0x6108

0x41414141

0xa0ca105a

内存地址 数 据
0x3810b20: 0x61080000

||

0x3810b20: 0x00000861

+

2

||

0x3810b22: 0x08616161

0x000d0e00

0x00317473

0xf3e60700

0xca000000

0x03000200

0x08616161

0x1ca5dc41

0x0de90000

0x00317402

0xdbdffb85f

0x0000000d

0x002bf8ff

0x41414141

0xbddffb2e8

0x00000000

```
(gdb) watch *0x3810b22
Hardware watchpoint 2:
*0x3810b22
(gdb)
```

Watch: 监察点，观察点。

GDB中的触发器，当进程读或写指定内存地址的数据时被触发。

如前文所述，0x3810b22，是我们的“Buffer Z”，只要监控Buffer Z，看“谁”向Buffer Z中写数据，就能找到Buffer Z的上线：Buffer Y。

```
(gdb) watch *0x3810b22
Hardware watchpoint 2:
*0x3810b22
(gdb) c
Continuing.
```

```
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK, 1 row affected (7 min
37.49 sec)
```

```
mysql> commit;
Query OK.....
```

```
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK .....
mysql>
```

```
(gdb) watch *0x3810b22  
Hardware watchpoint 2:  
*0x3810b22
```

```
(gdb) c  
Continuing.
```

```
Hardware watchpoint 2:  
*0x3810b22
```

```
Old value = 1630745612
```

```
New value = 1647588364
```

```
0x00007ffff6ae6e7b in  
__memcpy_ssse3_back () from  
/lib64/libc.so.6
```

```
(gdb)
```

```
mysql> commit;  
Query OK.....
```

```
mysql> insert into t1  
values(41,'aaaaaaa', 'AAAAAAA');  
Query OK .....
```

```
mysql> commit;  
==>HANG住<==
```

```
(gdb) watch *0x3810b22  
Hardware watchpoint 2:
```

```
*0x3810b22
```

```
(gdb) c
```

```
Continuing.
```

```
Hardware watchpoint 2:
```

```
*0x3810b22
```

```
Old value = 1630745612
```

```
New value = 1647588364
```

```
0x00007ffff6ae6e7b in  
__memcpy_ssse3_back () from  
/lib64/libc.so.6
```

```
(gdb)
```

原值: 6133340C

新值: 61616108

新值是08616161。

```
(gdb) watch *0x3810b22
Hardware watchpoint 2:
*0x3810b22
(gdb) c
Continuing.
Hardware watchpoint 2:
*0x3810b22

Old value = 1630745612
New value = 1647588364
0x00007ffff6ae6e7b in
__memcpy_ssse3_back () from
/lib64/libc.so.6
(gdb)
```

Libc库中的函数，它是memcpy的底层函数。实现内存块拷贝。

由于这个断点还是在commit时被触发的，所以Buffer Y不会是我们要找的源头：SQL执行时处理Binlog的函数。继续找Buffer Y的上线。

首先，确定Buffer Y的准确地址，然后使用同样的方法，使用watch观察点，监控Buffer Y的地址即可。

(gdb) info registers

```
rax      0x38113b1
rbx      0xd6b639c0
rcx      0x7fff89fff0b8
rdx      0x7ffff6ae7062
rsi      0x7fff89fff0dc
rdi      0x3810B45
rbp      0x7fffd5f28
rsp      0x7fffd5f228
.....
rip
```

.....
(gdb)

从rsi寄存器中，复制内存：
memcpy(rdi, rsi, rdx)

这是因为，watch观察点，是在被观察的内存位置被修改后触发，而不是修改前触发。观察因此，此处的rdi, rsi严格上不是memcpy的第一、二个参数值了。Memcpy已经改了它们。但是memcpy虽对它们有改变，但改变不大。

所以我们可以得出：rdi不是Buffer Z，但它在Buffer Z附近。

同理：rsi也不是Buffer Y的准确地址，Buffer Y在rsi附近。

还记得Buffer Z的准确地址吗？
和当前rdi的值相差多少？0x

也就是说：rdi-0x23，等于Buffer Z的地址。

有很大的可能，rsi-0x23，就是Buffer Y的地址。

di、
等，
的四

参
一个

。存。


```
(gdb) x/4x 0x7fff89fff0b9
0x7fff89fff0b9: 0x61616108
0x41410861      0x41414141
(gdb)
```

0x7fff89fff0dc也减去0x23字节，为0x7fff89fff0b9，显示此处的内存值

0x7fff89fff0b9处的值，正是08616161.....。也就是说，它就是我们要找的Buffer Z的上线，Buffer Y。

rsi: 源缓存。
rdi: 目标缓存。

```
(gdb) x/4x 0x7fff89fff0b9
0x7fff89fff0b9: 0x61616108      0x61616161
0x41410861      0x41414141
```

```
(gdb) delete 2
```

```
(gdb) watch *0x7fff89fff0b9
```

```
Hardware watchpoint 3: *0x7fff89fff0b9
```

```
(gdb)
```

删除刚才的watch
监察点，设置新的
监察点为
0x7fff89fff0b9。

```
(gdb) delete 2
(gdb) watch *0x7fff89fff0b9
Hardware watchpoint 3:
*0x7fff89fff0b9
```

```
(gdb) c
```

Continuing.

```
Hardware watchpoint 3:
*0x7fff89fff0b9
```

```
Old value = 1094795528
New value = 1111638536
0x00007ffff6ae75e4 in
__memcpy_ssse3_back () from
/lib64/libc.so.6
```

```
mysql> insert into t1 values(42,
'aaaaaaaa', 'AAAAAAAAA');
Query OK .....
```

```
mysql> commit;
Query OK.....
```

```
mysql> insert into t1 values(43,
'bbbbbbbb', 'BBBBBBBBB');
===>HANG住<===
```

bt, 显示调用栈。

调用栈是函数一层一层的调用关系。比如此处, __memcpy_ssse3_back被 my_b_safe_write调用, my_b_safe_write被 Log_event::wrapper_my_b_safe_write调用, 等等。

```
(gdb) bt
```

```
#0 0x00007ffff6ae75e4 in
```

```
#1 0x000000000185d155  
    at /home/mys
```

```
#2 0x00000000017a11aa
```

```
#3 0x00000000017bda73
```

```
#4 0x00000000017c84c7
```

```
.....
```

```
#8 0x00000000017f04c4 in THD::binlog_query
```

```
#9 0x0000000001761333 in Sql_cmd_insert::mysql_insert
```

```
.....
```

```
#17 0x00007ffff7bc7df3 in start_thread () from
```

```
#18 0x00007ffff6a933dd in clone () from /lib64/libc.so.6
```

```
(gdb)
```

THD::binlog_query的下层函数, 调用memcpy, 将 Buffer X中的数据, 拷贝到Buffer Y。

然后又经Buffer Y拷贝到Buffer Z, 又经Buffer Z写到

但THD::binlog_query不一定是DML执行期间, 唯一的Binlog处理函数。因此, 要继续“顺藤摸瓜”。

行期间, 处理Binlog的

```
(gdb) info registers
```

```
.....
```

rcx	0x7fffb42a4550	140736216057168
rdx	0x7ffff6ae75d6	140737332016598
rsi	0x7fffb4219598	140736216057168
rdi	0x7fff89fff0b4	140735500630010
rbp	0x7fffd6b5f7f0	0x7fffd6b5f7f0
rsp	0x7fffd6b5f7c8	0x7fffd6b5f7c8

```
.....
```

rip	0x7ffff6ae75e4	0x7ffff6ae75e4
-----	----------------	----------------

<__memcpy_ssse3_back+10>

```
.....
```

```
(gdb)
```

```
(gdb) x/100x 0x7fffb4219508
```

```
0x7fffb4219508: 0x0210a108 0x00000000 0x00000000 0x00000000
```

```
.....
```

```
0x7fffb4219578: 0x8f8f8f8f 0x8f8f8f8f 0x000033f8 0x62620800
```

```
0x7fffb4219588: 0x62626262 0x42086262 0x42424242 0x42424200
```

这一次是在rsi之前。

步1:

确认Buffer X的准确地址。

可以从rsi入手，因为rsi是Buffer X的附近。

步2:

在Buffer X处设置观察点。

步3:

再运行一条近似的SQL，等待观察点被触发。

步4:

使用bt命令，观察调用栈中是否有Binlog处理函数。

从当前的调用栈来看，已经出了Binlog的范围。

从函数名上看，当前在“解析”中。

我们的源头，已经找到了：**0x7fffb4219325**

```
(gdb) bt
```

```
#0  in __memcpy_ssse3_back ()
#1  in get_text
#2  in lex_one_token
#3  in MySQLlex
#4  in MySQLparse
#5  in parse_sql
#6  in mysql_parse
#7  in dispatch_command
#8  in do_command
#9  in handle_connection
#10 in pfs_spawn_thread
#11 in start_thread ()
#12 in clone
```

Binlog文件

```
  /\
  |
0x3810b22
  /\
  |
0x7fff89fff0b9
  /\
  |
0x7fffb42a4550
  /\
  |
0x7fffb4219585
  /\
  |
0x7fffb4219325
```

时间都去哪儿了

➤ 顺腾摸瓜结果：

除了THD::binlog_query函数，还有binlog_log_row函数，也是DML期间处理Binlog的函数。

➤ 更进一步：

我们已经知道binlog_query、binlog_log_row函数是处理Binlog的起点，有兴趣的话，你可以看看它的源码。它的代码并不长，使用gdb，单步跟踪一遍它的执行过程，看看在它们的执行过程中，数据是怎么处理的，需要哪些锁，数据最后怎么被复制到IO_CACHE中。

我们也知道，在提交时，Binlog数据用my_write函数写到磁盘文件中，在my_write上设断点，使用bt显示调用堆栈，可以很容易发现，MYSQL_BIN_LOG类的flush_cache_to_file函数，是IO的发起函数。相关代码在binlog.cc的8850行。



我们也知道，在提交时，Binlog数据用my_write函数写到磁盘文件中，在my_write上设断点，使用bt显示调用堆栈，可以很容易发现，MYSQL_BIN_LOG类的flush_cache_to_file函数，是IO的发起函数。相关代码在binlog.cc的8850行。

```
Breakpoint 3, my_write (Filedes=125,
    Buffer=0x380d280 "# Time: 2018-04-18T01:38:54.290419Z\n# User@Host: root[root] @ [192.168.1.100:3306] MySQL: [Query] (root)\n", Count=245, MyFlags=20) at
43 size_t sum_written= 0;
(gdb) c
Continuing.

Breakpoint 3, my_write (Filedes=131, Buffer=0x38112e0 "\264\241\326Z\"d\004", Count=280, MyFlags=52) at
43 size_t sum_written= 0;
(gdb) bt
#0 my_write (Filedes=131, Buffer=0x38112e0 "\264\241\326Z\"d\004", Count=280, MyFlags=52) at /home/mysql/mysql-5.7.19/sql/binlog.cc:8850
#1 0x00000000185a1e9 in inline_mysql_file_write (src_file=0x21eeef0 "/home/mysql/mysql-5.7.19/sql/binlog.cc", buffer=0x38112e0 "\264\241\326Z\"d\004", count=280, flags=52) at /home/mysql/mysql-5.7.19/sql/binlog.cc:8850
#2 0x00000000185d6cc in my_b_flush_io_cache (info=0x2d84ec8 <mysql_bin_log+840>, need_append=0) at /home/mysql/mysql-5.7.19/sql/binlog.cc:8850
#3 0x0000000017ea4e1 in MYSQL_BIN_LOG::flush_cache_to_file (this=0x2d84b80 <mysql_bin_log>, thd=0x7fffbc296060) at /home/mysql/mysql-5.7.19/sql/binlog.cc:9246
#4 0x0000000017eb2b0 in MYSQL_BIN_LOG::ordered_commit (this=0x2d84b80 <mysql_bin_log>, thd=0x7fffbc296060) at /home/mysql/mysql-5.7.19/sql/binlog.cc:9246
#5 0x0000000017e9959 in MYSQL_BIN_LOG::commit (this=0x2d84b80 <mysql_bin_log>, thd=0x7fffbc296060) at /home/mysql/mysql-5.7.19/sql/binlog.cc:9246
#6 0x000000000f1fad6 in ha_commit_trans (thd=0x7fffbc296060, all=true, ignore_global_read_lock=0) at /home/mysql/mysql-5.7.19/sql/ha_commit_trans.cc:100
#7 0x00000000163d42a in trans_commit (thd=0x7fffbc296060) at /home/mysql/mysql-5.7.19/sql/transaction.cc:100
#8 0x00000000153f992 in mysql_execute_command (thd=0x7fffbc296060, first_level=true) at /home/mysql/mysql-5.7.19/sql/execute.cc:100
#9 0x000000001542ed4 in mysql_parse (thd=0x7fffbc296060, parser_state=0x7fffd6b61db0) at /home/mysql/mysql-5.7.19/sql/parse.cc:100
#10 0x0000000015386a4 in dispatch_command (thd=0x7fffbc296060, com_data=0x7fffd6b62df0, command_type=COM_QUERY) at /home/mysql/mysql-5.7.19/sql/sql.cc:100
#11 0x000000001537582 in do_command (thd=0x7fffbc296060) at /home/mysql/mysql-5.7.19/sql/sql.cc:100
#12 0x00000000166a016 in handle_connection (arg=0x39460f0) at /home/mysql/mysql-5.7.19/sql/connection.cc:100
#13 0x000000001cf5690 in pfs_spawn_thread (arg=0x3928730) at /home/mysql/mysql-5.7.19/storage/engine/Perfschema.cc:100
#14 0x00007ffff7bc7df3 in start_thread () from /lib64/libpthread.so.0
#15 0x00007ffff6a933dd in clone () from /lib64/libc.so.6
(gdb) █
```

时间都去哪儿了

➤ 更进一步：

从MySQL_BIN_LOG::flush_cache_to_file，到在my_write中调用libc库函数write，完成写I/O，整个流程也不复杂，使用gdb跟踪一下，留意一下数据的流动、并发控制（锁，或Mutex锁）的申请释放，将使你对这一部分有更深入的了解。

很多朋友像了解下源码，但不知道如何入手，使用这种方法，可以很简单的找到感兴趣的入手点。

另外，刚开始阅读源码的时候，要尽量缩小自己的目标与范围，整个解析、执行、提交等过程是很复杂的。但提交时的I/O处理部分，目标就小多了，也简单很多。再进一步的：“提交时Binlog的I/O处理”，这个目标就更小了，代码量也更少，更容易入手。

而且，利用systemtap脚本，还可以不断写些时间统计、性能监控脚本，既有实际意义、又可以使用自己不断成长。

再而且，如果觉得源码那里不合适了，还可以自己改改。

这就是我所说的正向反馈式的“源码研究”。



时间都去哪儿了

找到了BinLog函数，可以很马上写出统计时间的systemtap脚本：

```
#!/usr/bin/stap
```

```
global tm_binlog, tt_binlog;
```

```
probe begin {  
    printf("Begin.\n");  
}
```

```
probe process("/data/db/mysql/lib/bin/mysqld").function("binlog_log_row") {  
    tm_binlog = gettimeofday_us();  
}
```

```
probe  
process("/data/db/mysql/lib/bin/mysqld").function("binlog_log_row").return {  
    tt_binlog += (gettimeofday_us() - tm_binlog);  
    printf("binlog:%d\n", tt_binlog);  
}
```

时间都去哪儿了

```
[root@VAGE01 stp]#  
[root@VAGE01 stp]# ./exmp1.stp -g 3714  
Begin.  
binlog:80
```

```
mysql>  
mysql>  
mysql> insert into t1  
      -> values(5b,'55bbbbbb', 'AAAAAAAAA');  
Query OK, 1 row affected (0.00 sec)
```

时间都去哪儿了

➤ 各个阶段对应的函数

使用前面所述方法，我们可以找到各个阶段对应的函数，结果如下表：

解析	parse_sql	
Binlog	binlog_log_row、binlog_query	
回滚	trx_undo_report_row_operation（包含trx_undof_page_add_undo_rec_log）	
Redo	UNDO的Redo	trx_undof_page_add_undo_rec_log、
	页的Redo	page_cur_insert_rec_write_log btr_cur_update_in_place_log page_cur_delete_rec_write_log
总执行	dispatch_command	



时间都去哪儿了

函数有了，可以写了如下时间统计脚本：

```
#!/usr/bin/stap

global tm_binlog, tm_undo, tm_u_redo, tm_redo, tm_exec, tm_parse;
global tt_binlog, tt_u_redo, tt_undo, tt_redo, tt_parse, tt_exec;
global cs;

probe begin {
    printf("Begin.\n");
}

probe process("/data/db/mysql/lib/bin/mysqld").function("parse_sql") {
    if ( tid() == $1 )
        tm_parse = gettimeofday_us();
}

probe process("/data/db/mysql/lib/bin/mysqld").function("parse_sql").return {
    if ( tid() == $1 ) {
        tt_parse += (gettimeofday_us() - tm_parse);
    }
}

probe process("/data/db/mysql/lib/bin/mysqld").function("binlog_query") {
    if ( tid() == $1 )
        tm_binlog = gettimeofday_us();
}

probe process("/data/db/mysql/lib/bin/mysqld").function("binlog_query").return {
    if ( tid() == $1 ) {
        tt_binlog += (gettimeofday_us() - tm_binlog);
    }
}

probe process("/data/db/mysql/lib/bin/mysqld").function("binlog_log_row") {
    if ( tid() == $1 )
        tm_binlog = gettimeofday_us();
}

probe process("/data/db/mysql/lib/bin/mysqld").function("binlog_log_row").return {
    if ( tid() == $1 ) {
        tt_binlog += (gettimeofday_us() - tm_binlog);
    }
}

probe process("/data/db/mysql/lib/bin/mysqld").function("trx_undo_report_row_operation") {
    if ( tid() == $1 )
        tm_undo = gettimeofday_us();
}

probe process("/data/db/mysql/lib/bin/mysqld").function("trx_undo_page_add_undo_rec_log") {
    if ( tid() == $1 )
        tm_u_redo = gettimeofday_us();
}

probe process("/data/db/mysql/lib/bin/mysqld").function("trx_undo_page_add_undo_rec_log").return {
    if ( tid() == $1 ) {
```

```
#!/usr/bin/stap
```

```
global tm_binlog, tm_undo, tm_u_redo, tm_parse;  
global tt_binlog, tt_u_redo, tt_parse, tt_exec;  
global
```

在函数开始执行时，记录时间，在函数执行完成时，取得函数的耗时，并累加进全局变量。

```
probe b  
pri  
}
```

只记录某一个线程的操作。

定义全局变量。
Systemtap的全局变量不需初始化，系统将自动赋值为0

```
probe process("/data/db/mysql/lib/bin/mysqld").function("parse_sql")  
if ( tid() == $1 )  
tm_parse = gettimeofday_us();  
}
```

```
probe process("/data/db/mysql/lib/bin/mysqld").function("parse_sql").return {  
if ( tid() == $1 )  
tt_parse += (tm_parse - gettimeofday_us());  
}
```

time.s(1): 定义器探针，每一秒触发一次。
触发后输出当前的累计时间。
效果类似于“vmstat 1”，一秒显示一行。

.....

```
probe timer.s(1) {  
if ( cs % 20 == 0 ) {  
printf("parse\tbin log\tundolog\tredo for undo\tredolog\tTotal  
Time\n");  
printf("-----\n");  
printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", tt_parse, tt_binlog, tt_undo, tt_u_redo,  
tt_redo, tt_exec );  
cs++;  
}  
}
```

.....

时间都去哪儿了

运行脚本需要知道线程号，在performance_schema.threads视图中的PROCESSLIST_ID列、THREAD_OS_ID列，可以得到我们需要的线程ID。
以下语句，得到当前线程的ID：

```
mysql>
mysql> select PROCESSLIST_ID, THREAD_OS_ID from
performance_schema.threads where PROCESSLIST_INFO like 'select
PROCESSLIST_ID, THREAD_OS_ID from performance_schema.threads where
PROCESSLIST_INFO%';
```

PROCESSLIST_ID	THREAD_OS_ID
4	3659

```
1 row in set (0.00 sec)

mysql>
```




```
[root@VAGE01 old]# ./tmstat.stp -x 3047 3659
```

3047: mysqld进程号。
3569: 要跟踪的线程号。

```
Begin.
```

parse	bin log	undolog	redo for undo	redolog	Total Time
-------	---------	---------	---------------	---------	------------

0	0	0	0	0	0
---	---	---	---	---	---

0	0	0	0	0	0
---	---	---	---	---	---

0	0	0	0	0	0
---	---	---	---	---	---

97	25649	38279	6641	9854	0
----	-------	-------	------	------	---

97	41822	88855	15717	23190	502118
----	-------	-------	-------	-------	--------

97	41822	88855	15717	23190	502118
----	-------	-------	-------	-------	--------

```
^C .....
```

```
mysql> insert into t5(c1, c2)
```

```
select c1, c2 from t5;
```

```
Query OK, 4096 rows affected (0.50 sec)
```

```
Records: 4096 Duplicates: 0 Warnings: 0
```

```
[root@VAGE01 old]# ./tmstat.stp -x 3047 3659
Begin.
```

```
parse      bin log undolog redo for undo redolog
```

	bin	log	undolog	redo	for undo	redolog
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
97	25649	38279	6641	9854	0	0
97	41822	88855	15717	23190	502118	502118
97	41822	88855	15717	23190	502118	502118

```
^C .....
```

Redo For Undo的时间，占了总时间的3.1%，页Redo占了总时间的4.6%，Redo共计占7.7%

Undo占了总时间的17.7%

Binlog占了总时间的8.3%

这些为了事务、更高的可靠性，而占的总时间，达到33.7%。

脚本统计的总执行时间，502118微秒，和MySQL自己统计的时间一致，0.50秒。

```
mysql> insert into t5(c1, c2)
      select c1, c2 from t5;
Query OK, 4096 rows affected (0.50 sec)
Records: 4096 Duplicates: 0 Warnings: 0
```

开课了

我在ITPUB平台上，将陆续发布系列Oracle/MySQL调试课程，如果想进一步学习Oracle，或学习更多调试MySQL的技巧，敬请期待。



THANKS





讲师申请

联系电话（微信号）：18612470168

关注“ITPUB”更多
技术干货等你来拿~

与百度外卖、京东、魅族等先后合作系列分享活动



让学习更简单

微学堂是以ChinaUnix、ITPUB所组建的微信群为载体，定期邀请嘉宾对热点话题、技术难题、新产品发布等进行移动端的在线直播活动。

截至目前，累计举办活动期数60+，参与人次40000+。

ITPUB学院

ITPUB学院是盛拓传媒IT168企业事业部（ITPUB）旗下
企业级在线学习咨询平台
历经18年技术社区平台发展
汇聚5000万技术用户
紧随企业一线IT技术需求
打造全方式技术培训与技术咨询咨询服务
提供包括企业应用方案培训咨询（包括企业内训）
个人实战技能培训（包括认证培训）
在内的全方位IT技术培训咨询服务

ITPUB学院讲师均来自于企业
一些工程师、架构师、技术经理和CTO
大会演讲专家1800+
社区版主和博客专家500+

培训特色

无限次免费播放
随时随地在线观看
碎片化时间集中学习
聚焦知识点详细解读
讲师在线答疑
强大的技术人脉圈

八大课程体系

基础架构设计与建设
大数据平台
应用架构设计与开发
系统运维与数据库
传统企业数字化转型
人工智能
区块链
移动开发与SEO



联系我们

联系人：黄老师
电话：010-59127187
邮箱：edu@itpub.net
网址：edu.itpub.net
培训微信号：18500940168