管中窥豹：从内核角度谈运维

吕海波（VAGE）

内容简介

内容简介：

- 管中窥豹的工具：动态跟踪技术介绍
- 发现函数
- 神奇的等待事件
- 基于动态跟踪的运维工具

管中窥豹的工具：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

ProbeVue: AIX

……

- 静态追踪：

ptrace

gdb/mdb

管中窥豹的工具：动态跟踪技术

- 程序开发的基本语法

变量、常量
条件
分枝
循环
子程或子函数
数组

- 探针：probe

动态内核的基础。

相当于Oracle的触发器。

举个例子

计算Oracle逻辑读时间的动态跟踪脚本：

```
[root@VAGE01 old]# cat lgr.stp
#!/usr/bin/stap

global tm;

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

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbgtcr")
{
    tm=gettimeofday_us();
}

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").retu
rn {
    printf("Time:%d\n",gettimeofday_us()-tm );
}
```

举个例子


```
[root@VAGE01 old]# cat lgr.stp
#!/usr/bin/stap    //定义解释器，类似Shell脚本

global tm;

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

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbgtcr")
{
    tm=gettimeofday_us();
}

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").retu
rn {
    printf("Time:%d\n",gettimeofday_us()-tm );
}
```

举个例子


```
[root@VAGE01 old]# cat lgr.stp
#!/usr/bin/stap

global tm;  //全局变量，计录时间。

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

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbgtcr")
{
    tm=gettimeofday_us();
}

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").retu
rn {
    printf("Time:%d\n",gettimeofday_us()-tm );
}
```

举个例子


```
[root@VAGE01 old]# cat lgr.stp
#!/usr/bin/stap

global tm;

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

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbgtcr")
{
    tm=gettimeofday_us();
}

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").retu
rn {
    printf("Time:%d\n",gettimeofday_us()-tm );
}
```

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

举个例子


```
[root@VAGE01 old]# cat lgr.stp
#!/usr/bin/stap

global tm;

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

probe
process("/opt/oracle/product/11204                                    gtcr")
{
    tm=gettimeofday_us();
}

probe
process("/opt/oracle/product/11204/bin/oracle").function("kcbrls").retu
rn {
    printf("Time:%d\n",gettimeofday_us()-tm );
}
```

探针：退出函数kcbrls时触发。相当于after kcbrls触发器。

举个例子

kcbgtcr: 申请块上的cache buffer chains latch，申请块上的buffer pin lock。可以看作大部分种类逻辑读的开始。

kcbrls: 释放块上持有的buffer pin lock，可以看作逻辑读的结束。

举个例子


```
SQL> select c.sid,spid from (select sid from
v$mystat where rownum<=1) c,v$session
a,v$process b where c.sid=a.sid and
a.paddr=b.addr;


       SID SPID
---------- ------------------------
       395 7016
```

```
[root@VAGE old]# ./lgr.stp -x 7016
WARNING: cannot find module。。。
Begin.
```

举个例子


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

        ID C1                   C2
---------- -------------------- ---------------
         1 aaaaaaaa             bbbbbbbbbbbb
```

```
[root@VAGE old]# ./lgr.stp -x 7016
WARNING: cannot find module。。。
Begin.
Time:23
```

➢ 脚本效果

效果还不错，满足一下好奇心，多次测试的均值，Oracle完成一次“标准逻辑读”，耗时25微秒左右。

（注：因为有动态调试的影响，实际耗时会比25微秒略低一点。动态调试对性能的影响，大概在1%到5%范围内。）

那么，MySQL的呢？下面是测量Innodb MySQL的纯逻辑读时间的动态调试脚本：

管中窥豹的工具：动态跟踪技术


```
[root@VAGE01 old]# cat lgr_mysql.stp
#!/usr/bin/stap

global tm;

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

probe
process("/usr/local/mysql/bin/mysqld").statement("buf_page_get_ge
n@buf0buf.cc:2563") {
    tm=gettimeofday_us();
}

probe process("/usr/local/mysql/bin/mysqld").statement(0xb48fce)
{
    printf("Time:%d\n", gettimeofday_us()-tm );
}
```

buf_page_get_gen，相当于Oracle的kcbgtcr。

管中窥豹的工具：动态跟踪技术


```
[root@VAGE01 old]# cat lgr_mysql.stp
#!/usr/bin/stap

global tm;

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

probe
process("/usr/local/mysql/bin/mysqld").statement("buf_p          c_ge
n@buf0buf.cc:2563") {
    tm=gettimeofday_us();
}

probe process("/usr/local/mysql/bin/mysqld").statement(0xb48fce)
{
    printf("Time:%d\n", gettimeofday_us()-tm );
}
```

0xb48fce，这个内存地址，是buf_page_release函数的结束处。相当于Oracle的kcbrls函数return处。

管中窥豹的工具：动态跟踪技术

脚本运行步骤：

```
[root@VAGE01 old]# ps -ef|grep mysqld
。。。。。。
mysql      5798   4912  0 09:26 pts/1    00:00:02 /usr/local/mysql/bin/mysqld –
。。。。。。
。。。。。。
[root@VAGE01 old]# ./lgr_mysql.stp -gx 5798
Begin.
Time:33
Time:39
```

测试SQL：

```
mysql> select * from t1 where id=1;
+----+-----------+
| id | name      |
+----+-----------+
|  1 | XYZXYZXYZ |
+----+-----------+
1 row in set (0.00 sec)
```

管中窥豹的工具：动态跟踪技术

➢ 发现函数

动态跟踪技术其实非常简单，脚本也都不长，重要的是， 如何获得kcbgtcr、kcbrls，MySQL的buf_page_get_gen， buf_page_release这些函数的意义。

MySQL：源码+gdb。

Oracle：DTrace+mdb。

发现函数

是DTrace登场的时候了。

DTrace发源于Solaris系统，虽然现在也移植到Linux下，但Linux下的DTrace缺少一个重要功能，目前我们还不能使用它来“发现函数”。

注意：

虽然DTrace只能在Solaris下，但Oracle在所有OS系统中的函数名基本都是一致的，因此我们可以在Solaris下发现我们要找的函数，在Linux或其他系统中，使用动态跟踪写脚本。

发现函数

先来看看我们的“发现函数”的第一步，也是我们打开Oracle这个黑盒子的第一步。

只需要用下面短短几行代码，就可以得到Oracle在进行某个操作时的所调用的函数名，以及函数相关参数：

```
#!/usr/sbin/dtrace -s -n

dtrace:::BEGIN
{
	i=1;
}
pid$1:::entry
{
	printf("i=%d %s(%x,%x,%x,%x,%x,%x);",i, probefunc,arg0,arg1,arg2,arg3,arg4,arg5);
	i=i+1;
}
```

发现函数

这是跟踪下列测试SQL执行的结果：

```
SQL> select * from vage where rowid='AAADLMAAEAAAACDAAA';
        ID NAME
---------- --------------------
         1 aaaaaa
```

```
bash-3.2# chmod 755 get_func.d
bash-3.2# ./get_func.d 1340
dtrace: script './get_func.d' matched 152978 probes
CPU     ID                    FUNCTION:NAME
  0 201201                      memcpy:entry i=1 memcpy(fffffd7fffdfc8eb,d4bd860,1,1,d4bd9a2,142);
  0  52479                  kslwtectx:entry i=2 kslwtectx(fffffd7fffdfc610,d4bd860,d4bd861,fffffd7fffdfc8ec,ced87ac,11);
  0 203504                  gethrtime:entry i=3 gethrtime(fffffd7fffdfc610,d4bd860,d4bd861,fffffd7fffdfc8ec,0,11);
  0  52554         kslwt_end_snapshot:entry i=4 kslwt_end_snapshot(395625898,395625898,1,1ee029adacf7,ced881c,ced87a8);
  0  72715      kews_update_wait_time:entry i=5 kews_update_wait_time(6,20735b7,1f,159,395ed9f60,afe770c);
  0  52268                  kskthewt:entry i=6 kskthewt(ef5b7e59,0,6,0,0,394d0f7e0);
  0 201201                      memcpy:entry i=7 memcpy(fffffd7fffdfc8f6,d4bd861,2,2010,d4bd863,395623bb0);
  0 120954                   kpuhhmrk:entry i=8 kpuhhmrk(cedd8e0,d4bd861,d4bd863,11,ced8c68,1068);
  0 135665                  kpggGetPG:entry i=9 kpggGetPG(cedd8e0,d4bd861,d4bd863,11,fffffd7ffcb21f90,fffffd7ffcb21f90);
  0 201208                      setjmp:entry i=10 setjmp(fffffd7fffdfc758,d4bd861,d4bd863,11,ceed7f8,bc794f8);
  0 124896                      kghmrk:entry i=11 kghmrk(ceed648,fffffd7ffcb21000,0,1,cedd8e0,fffffd7fffdfc730);
  0 123372                      ttcpip:entry i=12 ttcpip(cedd950,69,fffffd7fffdfc920,0,fffffd7fffdfdaa0,fffffd7fffdfc8fc);
  0 135665                  kpggGetPG:entry i=13 kpggGetPG(cedd950,69,fffffd7fffdfc920,0,fffffd7ffcb21f90,fffffd7fffdfc8fc);
  0 201201                      memcpy:entry i=14 memcpy(fffffd7fffdfc920,d4bd863,10,0,d4bd873,1);
  0 201201                      memcpy:entry i=15 memcpy(fffffd7fffdfc7b0,d4bd873,4,0,d4bd877,1);
  0  77515                      opiodr:entry i=16 opiodr(69,2,fffffd7fffdfc920,0,41e3a80,bc76740);
  0 201202                      memset:entry i=17 memset(fffffd7ffcb4bc58,0,28,0,ced8c30,ced9a70);
  0 201208                      setjmp:entry i=18 setjmp(fffffd7fffdfbec8,0,0,0,bc794f8,ced9a70);
  0  53523                     ksupucg:entry i=19 ksupucg(1,c7267b8,28d,1,380021018,fffffd7fffdfbea0);
  0 127726                       slcpu:entry i=20 slcpu(ceed278,c7267b8,28d,8,ceed2b0,391712210);
  0 203596                       times:entry i=21 times(fffffd7fffdfbaf0,c7267b8,28d,8,ceed2b0,391712210);
  0 203504                  gethrtime:entry i=22 gethrtime(fffffd7fffdfbaf0,c7267b8,0,fffffd7ffcd92caa,2008,64);
  0  52413       ksl_get_shared_latch:entry i=23 ksl_get_shared_latch(395683950,1,395623bb0,56,8,1);
  0 128134                  skgslocas:entry i=24 skgslocas(395683950,0,1,fffffd7fffdfbac0,0,bc7acc8);
  0  52418                     kslfre:entry i=25 kslfre(395683950,1,2,1,38001bc20,ceed648);
  0 128136                 sskgsldecr:entry i=26 sskgsldecr(395683950,1,ffffffffd,10,395683950,e);
  0  59601                    ktcsptg:entry i=27 ktcsptg(39567a4e0,0,0,10,0,ced8cc0);
  0 103289                    kscdnfy:entry i=28 kscdnfy(1,39567a450,0,10,38000af10,395623bb0);
  0  60477                kticallpush:entry i=29 kticallpush(1,39567a450,ff,ff,1,fffffd7ffc9df470);
  0  53171            ksptch_callpush:entry i=30 ksptch_callpush(1,39567a450,ff,ff,fffffd7ffc9df480,0);
```

看到满屏奇奇怪怪的函数名，很多人在这一步放弃了。其实，这才刚刚开始。理解它们很简单。下面我们就以等待事件为例，说一下如何从这里面发现价值。

神奇的等待事件

有没有觉得Oracle的等待事件非常神奇？它是我们DBA的重要工具。它的原理是什么呢？下面，我们就以它为例，找出Oracle登记等待事件时的相关函数。

神奇的等待事件

万事开头难。为了研究等待事件，我还是花了点时间开头的。DTrace中有一个简单的方法，可以统计调用每个函数的次数。我就是从这个次数开始的。

执行测试SQL：“select * from vage where rowid=‘AAADLMAAEAAAACDAAA‘”，当是软软解析、逻辑读时，在没有竞争的情况下，会有四次等待事件：

两次SQL*Net message to client
两次SQL*Net message from client

关于这点可以很容易的从v$session_event中得到，也可以从10046中得知。

神奇的等待事件


```
=====================
PARSING IN CURSOR #139822497182000 len=49 dep=0 uid=35 oct=3 lid=35
select * from t1 where rowid='AAADs6AAEAAAACRAAB'
END OF STMT
PARSE #139822497182000:c=2061,e=38769,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og
EXEC #139822497182000:c=0,e=23,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,p
WAIT #139822497182000: nam='SQL*Net message to client' ela= 1 driver
FETCH #139822497182000:c=0,e=15,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=1,
STAT #139822497182000 id=1 cnt=1 pid=0 pos=1 obj=15162 op='TABLE A
WAIT #139822497182000: nam='SQL*Net message from client' ela= 363 d
FETCH #139822497182000:c=0,e=2,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,p
WAIT #139822497182000: nam='SQL*Net message to client' ela= 0 driver

*** 2017-12-15 10:45:06.210
WAIT #139822497182000: nam='SQL*Net message from client' ela= 3404087
CLOSE #139822497182000:c=12,e=12,dep=0,type=0,tim=1513305906210691
=====================
PARSING IN CURSOR #139822497182000 len=55 dep=0 uid=35 oct=42 lid=35 t
alter session set events '10046 trace name context off'
END OF STMT
```

神奇的等待事件


```
=====================
PARSING IN CURSOR #139822497182000 len=49 dep=0 uid=35 oct=3 lid=35
select * from t1 where rowid='AAADs6AAEAAAACRAAB'
END OF STMT
PARSE #139822497182000:c=2061,e=38769,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og
EXEC #139822497182000:c=0,e=23,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,p
WAIT #139822497182000: nam='SQL*Net message to client' ela= 1 driver
FETCH #139822497182000:c=0,e=15,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=1,
STAT #139822497182000 id=1 cnt=1 pid=0 pos=1 obj=15162 op='TABLE A
WAIT #139822497182000: nam='SQL*Net message from client' ela= 363 d
FETCH #139822497182000:c=0,e=2,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,p
WAIT #139822497182000: nam='SQL*Net message to client' ela= 0 driver

*** 2017-12-15 10:45:06.210
WAIT #139822497182000: nam='SQL*Net message from client' ela= 3404087
CLOSE #139822497182000:c=12,e=12,dep=0,type=0,tim=1513305906210691
=====================
PARSING IN CURSOR #139822497182000 len=55 dep=0 uid=35 oct=42 lid=35 t
alter session set events '10046 trace name context off'
END OF STMT
```

神奇的等待事件

然后，我用如下脚步跟踪测试SQL的执行，观察函数调用的次数：

```
#!/usr/sbin/dtrace -s -n

dtrace:::BEGIN
{
      printf("Start...\n");
}


pid$1:::entry
{
      @counts[probefunc]=count();
}

dtrace:::END
{
      trace("--------------------------");
      printa(@counts);
}
```

神奇的等待事件

因为测试SQL一共会有4次等待事件，所以我只观注调用次数为4的函数，这些函数共有15个：

KGHISPIR	4
_save_nv_regs	4
kews_sqlcol_begin	4
kews_update_wait_time	4
kghxhal	4
kghxhfr	4
kglGetMutex	4
kksGetStats	4
kskthbwt	4
kskthewt	4
kslwt_end_snapshot	4
kslwt_start_snapshot	4
kslwtbctx	4
kslwtectx	4
opikndf2	4

在这些函数里面，一定有一些是关于等待事件的函数？

神奇的等待事件

经过观察，如下几个函数引起我的注意，原因很简单，它们的名字中带有”wt”：

KGHISPIR	4
_save_nv_regs	4
kews_sqlcol_begin	4
kews_update_wait_time	4
kghxhal	4
kghxhfr	4
kglGetMutex	4
kksGetStats	4
kskthbwt	4
kskthewt	4
kslwt_end_snapshot	4
kslwt_start_snapshot	4
kslwtbctx	4
kslwtectx	4
opikndf2	4

神奇的等待事件

这其中**kslwtbctx**是最早被调用的函数:

```
KGHISPIR                    4
 _save_nv_regs                  4
 kews_sqlcol_begin              4
 kews_update_wait_time          4
 kghxhal                        4
 kghxhfr                        4
 kglGetMutex                    4
 kksGetStats                    4
 kskthbwt                       4
 kskthewt                       4
 kslwt_end_snapshot             4
 kslwt_start_snapshot           4
 kslwtbctx                      4
 kslwtectx                      4
 opikndf2                       4
```

下面，我们就从kslwtbctx开始。

神奇的等待事件

我查看了从kslwtbctx开始Oracle会调用的一些函数，它们依次是：

```
i=520 kslwtbctx(fffffd7fffdfb180,1,fffffd7fffdfb3d7,1,c725158,4034);
i=521 gethrtime(21,1,fffffd7fffdfb3d7,1,ceed648,0);
i=522 kskthbwt(0,42beed13,742beed13,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfb1e8,30,7b,c725158,395c42ba8);

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
 0: 55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,1,7b,ced881c,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,1,380009ce8,ceed648);
```

……

Oracle在第520次函数调用时，调用了kslwtbctx，在第521次调用了gethrtime，这是一个获取时间的函数。等待事件的一个重要操作，不就是记录时间吗！

补充一点，观察内存的流动很重要。Memcpy就是完成内存的流动函数。因此观察memcpy拷贝了什么样的值很重要。

神奇的等待事件


```
i=520 kslwtbctx(fffffd7fffdfb180,1,fffffd7fffdfb3d7,1,c725158,4034);
i=521 gethrtime(21,1,fffffd7fffdfb3d7,1,ceed648,0);
i=522 kskthbwt(0,42beed13,742beed13,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfb1e8,30,7b,c725158,395c42ba8);

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
 0: 55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,1,7b,ced881c,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,1,380009ce8,ceed648);
```

它从fffffd7fffdfb1e8处，向395be69b0拷贝0x30（十进制48）个字节。拷贝的内容我也用DTrace把它显示出来了，“55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff ……”。

这其中前两个字节“5501”引起了我的注意，我的测试机是小端，5501真正表示的数据是0155，十进制是341.

神奇的等待事件


```
i=520 kslwtbctx(fffffd7fffdfb180,1,fffffd7fffdfb3d7,1,c725158,4034);
i=521 gethrtime(21,1,fffffd7fffdfb3d7,1,ceed648,0);
i=522 kskthbwt(0,42beed13,742beed13,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfb1e8,30,7b,c725158,395c42ba8);

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
 0: 55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,1,7b,ced881c,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,1,380009ce8,ceed648);
```

Oracle中的每个等待事件，都有一个编号，这个341会不会就是等待事件编号呢？这很容易验证，查询等待事件编号等于341的，看看到底有没有这个编号，如果有的话，等待事件是什么！

神奇的等待事件

查询结果：

```
SQL> select event#, event_id from v$event_name where event#=341;

    EVENT# NAME
---------- --------------------------
       341 SQL*Net message to client
```

编号341的等待事件是SQL*Net message to client。

我的测试SQL，会有两次to client等待事件。因此，综合上面这些因素，我更加怀疑kslwtbctx和等待事件一定有关系。哪么，接下来要怎么确认这个猜想呢？

神奇的等待事件

还记得前面提过的静态跟踪（gdb/mdb等）吗，下面要使用静态跟踪了。

找到了断点又有什么用呢？

使用静态跟踪，我们可以让Oracle的执行流，在某个函数处停下来，然后我们可以慢慢观察Oracle的状态。

下面，我们就演示一下，我们如何利用静态跟踪，挖掘等待事件机制。

静态跟踪，使用Linux/AIX下的gdb，或使用Solaris下的mdb都可以，命令差别不大。

以gdb为例，在kslwtbctx函数处设置断点的方式如下：

```
gdb -p oracle_server进程号
b kslwtbctx
c
```

神奇的等待事件


```
SQL> select event,state from v$session where
sid=19;

EVENT                          STATE
------------------------------ ----------------
---
SQL*Net message from client    WAITED KNOWN
TIME
```

kslwtbctx
gethrtime
kskthbwt
memcpy
kslwt_start_snapshot
nioqsn

我先在kslwtbctx函数入口处设置断点，让CPU执行到kslwtbctx函数入口处。我们这里，需要观察的“状态”，就是Oracle的等待事件。

可以看到，在CPU执行到kslwttctx入口处时，进程什么都不在等。State列值为“WAITED KNOWN TIME”，这说明进程是ON CPU的状态，当前没有等待事件。

神奇的等待事件


```
SQL> select event,state from v$session where
sid=19;

EVENT                          STATE
------------------------------ ----------------
---
SQL*Net message from client    WAITED KNOWN
TIME
```

kslwtbctx

gethrtime

kskthbwt

memcpy

kslwt_start_snapshot

nioqsn

继续，在下一函数gethrtime处设置断点，让CPU执行到gethrtime函数入口处，观察等待事件，没有变化，进程这时还是没有任何等待事件。

神奇的等待事件


```
SQL> select event,state from v$session where
sid=19;

EVENT                          STATE
------------------------------ ----------------
---
SQL*Net message from client    WAITED KNOWN
TIME
```

kslwtbctx
gethrtime
kskthbwt
memcpy
kslwt_start_snapshot
nioqsn

重复上面的操作，在CPU前进到kskthbwt函数入口处时，还是没有任何等待事件。

神奇的等待事件


```
SQL> select event,state from v$session where
sid=19;

EVENT                          STATE
------------------------------ ----------------
---
SQL*Net message from client    WAITING
```

kslwtbctx
gethrtime
kskthbwt
memcpy
kslwt_start_snapshot
nioqsn

在CPU前进到memcpy函数入口处时，State列变为了Waiting。

神奇的等待事件


```
SQL> select event,state from v$session where
sid=19;

EVENT                          STATE
------------------------------ ----------------
---
SQL*Net message to client      WAITING
```

kslwtbctx
gethrtime
kskthbwt
memcpy
kslwt_start_snapshot
nioqsn

在CPU前进到kslwt_start_snapshot函数入口处时，EVENT列变为了”SQL*Net message to client”。到此，Oracle完成了“登记”等待事件这一操作。其始从kslwtbctx处，等待事件就已经开始了。它调用gethrtime得到时间，然后调用kskthbwt修改v$session.STATE列状态。再接着调用memcpy函数，将等待事件编号拷贝到SGA中的v$session中。

神奇的等待事件

我们已经知道了kslwtbctx是Oracle产生等待事件的函数，哪么，等待事件的结束函数是什么呢？

这个很容易找到，仍然从调用次数为4的函数中找，很快就能找到，很快就能找到，等待结束的函数是：kslwtectx。下面这一串，是等待事件结束时的函数调用堆栈：

Kslwtectx→
  gethrtime
  kslwt_end_snapshot
  kslwt_update_stats_int
    → kews_update_wait_time
  kskthewt

使用前面的方法，让CPU一个函数一个函数的向下执行，当执行到kskthewt函数时，v$session中state列，会由WAITING变为WAITED KNOWN TIME，说明等待事件到此已经完全结束了。

神奇的等待事件

现在，让我们总结一下我们的成果：

1、看到kslwtbctx，就是某个等待事件开始了。红色字母“b”，应该是begin。
2、看到Kslwtectx，就是某个等待事件结束了。红色字母“e”，应该是end。
3、在kslwtbctx之下，会有memcpy，它所拷贝内存的前两个字节，是一个整数（短整型, short int），这个数字就是等待事件编号。

有了这三点信息，我们可以非常容易从DTrace的跟踪结果，总结出Oracle等待事件的机制。

曾经有人问过我一个问题，Oracle会在什么情况下记录等待事件，利用我们的成果，我想，可以回答这个问题了。

神奇的等待事件

在v$system_event视图中，有一个TIME_WAITED_MICRO列，以微秒（百万分之一秒）为单位的等待时间，正是这一列误导了我，我一直认为Oracle会记录任何超过一微秒的动作：

EVENT	VARCHAR2(64)	Name of the wait event
TOTAL_WAITS	NUMBER	Total number of waits for the event
TOTAL_TIMEOUTS	NUMBER	Total number of timeouts for the event
TIME_WAITED	NUMBER	Total amount of time waited for the event (in hundredths of a second)
AVERAGE_WAIT	NUMBER	Average amount of time waited for the event (in hundredths of a second)
TIME_WAITED_MICRO	NUMBER	Total amount of time waited for the event (in microseconds)
TOTAL_WAITS_FG	NUMBER	Total number of waits for the event, from foreground sessions
TOTAL_TIMEOUTS_FG	NUMBER	Total number of timeouts for the event, from foreground sessions

当然，这种想法是错误的，记录任何超过一微秒的动作，这样代价太大了。事实上，Oracle分两种情况记录等待事件。

神奇的等待事件

Oracle的等待事件可以分为两大类：

- 主动式
- 被动式

什么是主动式等待事件呢？当Oracle要开始进行一个耗时很久的操作时，无论是否遇到阻塞，都会主动登记一个等待事件，告诉DBA，我要开始某某操作了。

被动式等待事件，发生在有竞争、进程被阻塞时。产生了阻塞，进程被Block住，在进程转入Sleeping状态前，会登记一个等待事件，告诉DBA发生了什么。

简单点说，主动式就相当于，你要出远门了，要很久才能回家，出发前你会通知一下家人，“我要出远门了”。被动式的呢，相当于你下楼买包烟，你可能穿着睡衣、拖鞋，直接就下楼了，通常不用再主动的告诉家里人。但是在你买烟的过程中出现了竞争，楼下小超市正在大促销，很多人排队结帐，你被阻塞了。这个时候，为了避免家人担心，你打个电话通知一下，“我被阻塞了”。

神奇的等待事件

- 主动式

主动式等待事件，Oracle只用于磁盘I/O、网络I/O，或和I/O相关的等待事件 。这两类I/O操作，Oracle无法控制其快慢，它们的进度取决于多种设备（磁盘、交换机、网线、FC卡或网卡等）。有可能由于设备问题，导致进程长时间无法完成操作。每一次I/O操作，都相当于上页PPT中说的“出趟远门”，所以主动登记个等待事件，免得家人担心。

- 被动式

被动式等待事件，如果进程没有遇到阻塞，它不会登记等待事件。因此被动式等待事件，是Oracle遇到竞争的标志。
但是不必担心，只要竞争在合理范围内，也就是说等待事件的响应时间不高、等待次数正常，就不必担心。竞争总是会有的。

也就是说，除非遇到这两种情况：进行I/O操作、遇到竞争，否则，Oracle是不会记录等待事件的。

基于动态跟踪的运维工具

典型的运维性能问题：某操作的最耗时的部分在哪儿。

在Oracle中，可以用“等待时间”、“时间模型”回答这个问题。

但有些时候“等待时间”、“时间模型”的粒度还是有点粗，或者有时候意义不明确。我们可以使用动态跟踪工具，打造一个意义清晰的、粒度可控的展现工具，为我们发现性能瓶颈。

基于动态跟踪的运维工具


```
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Releas CBC: %d \n", df_t );
        prev_ts = t;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("ksulbem").return
{
    if ( start_sql == 1 )
    {
        start_sql = 2;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("nttfpwr")
{
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Start Sent: %d \n", df_t );
        prev_ts = t;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("nttfpwr").return
{
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Finish Sent: %d \n", df_t );
        prev_ts = t;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("nttfprd")
{
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Start recv: %d \n", df_t );
        prev_ts = t;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("nttfprd").return
{
```

基于动态跟踪的运维工具

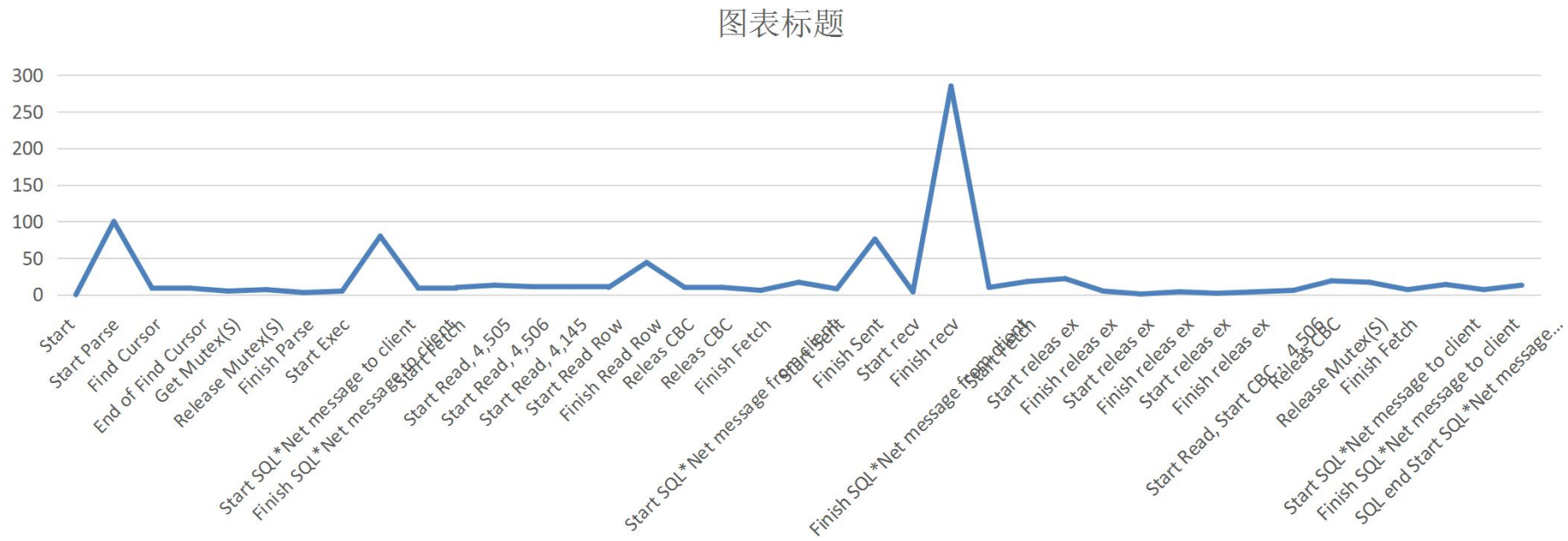

```
probe process("/opt/oracle/product/11204/bin/oracle").function("nttfpwr")
{
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Start Sent: %d \n", df_t );
        prev_ts = t;
    }
}

probe process("/opt/oracle/product/11204/bin/oracle").function("nttfpwr").return
{
    if ( start_sql == 1 )
    {
        t = gettimeofday_us()-ts;
        df_t = ( t - prev_ts );
        print_tm ( df_t / 100 );
        printf("Finish Sent: %d \n", df_t );
        prev_ts = t;
    }
}
```

```
[root@VAGE01 old]# ./sql1.stp -gx 10741
WARNING: cannot find module /opt/oracle/product/1120
Begin...
Start : 0
|
|
Start Parse : 100
|
Find Cursor: 9
|
End of Find Cursor: 9
|
Get Mutex(S): 5
|
Release Mutex(S): 7
|
Finish Parse: 3
|
Start Exec: 5
|
Start Wait Event: 80 , SQL*Net message to client
|
Finish Wait Event: 9 , SQL*Net message to client
|
Start Fetch: 10
|
Start Read, Start get CBC latch on file_id:4, block_id:505 , : 13
```

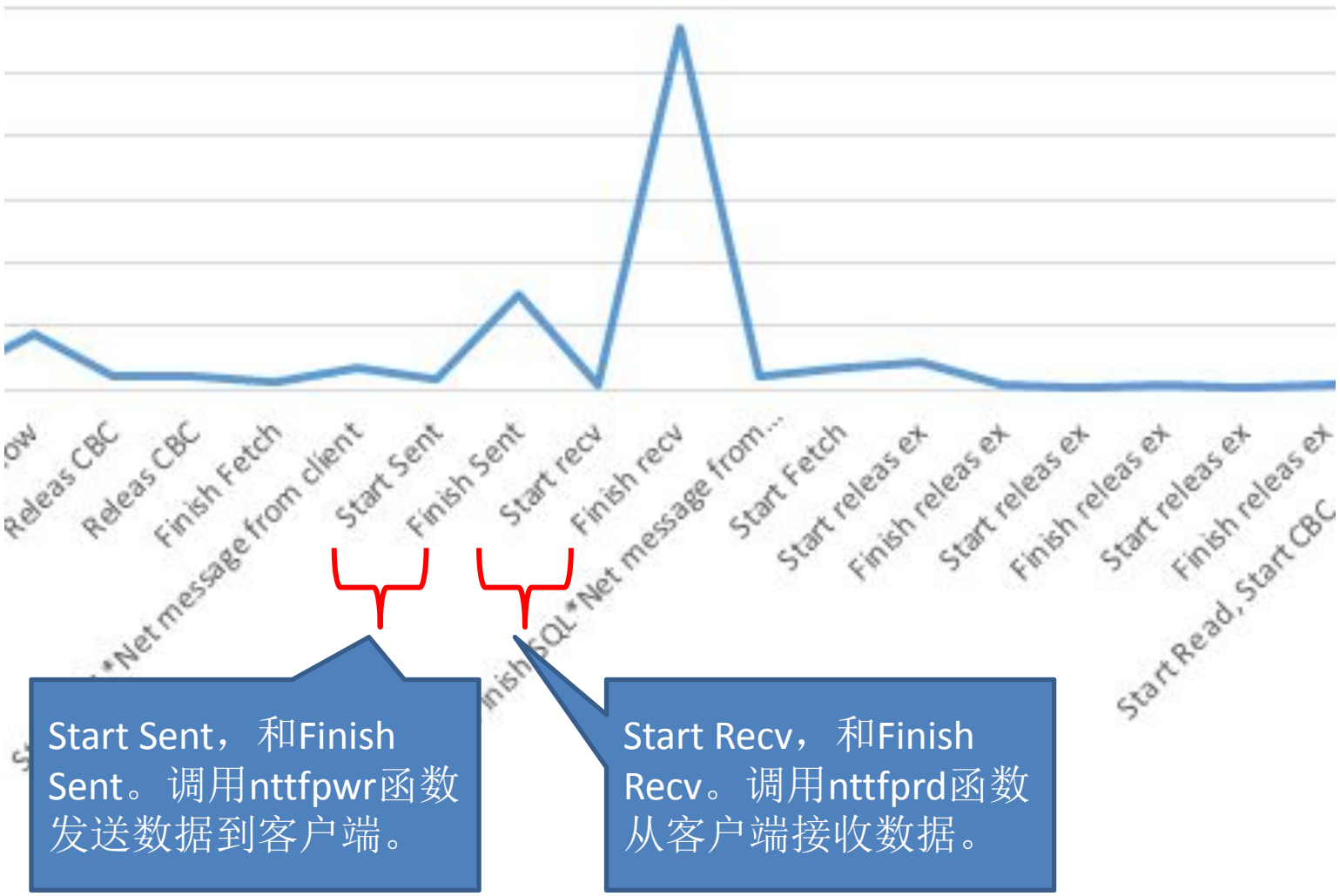
基于动态跟踪的运维工具

示例SQL：select * from t1 where id=1，两层索引，逻辑读，软软解析。最耗时的部分：



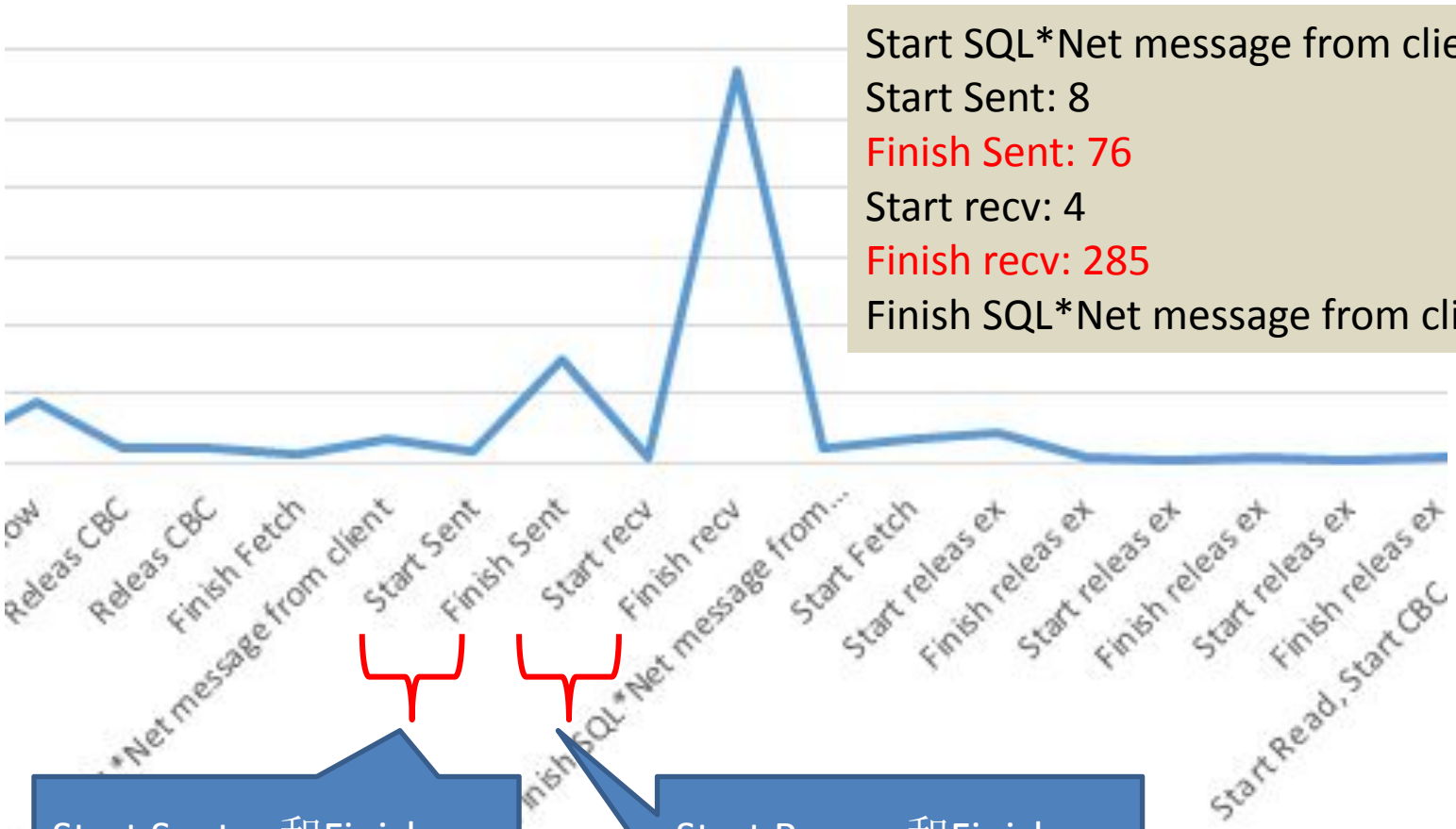
基于动态跟踪的运维工具

最耗时的部分，是从SQL*Net message from client期间的Start Sent到Finish Sent， Start Recv和Finish Recv。



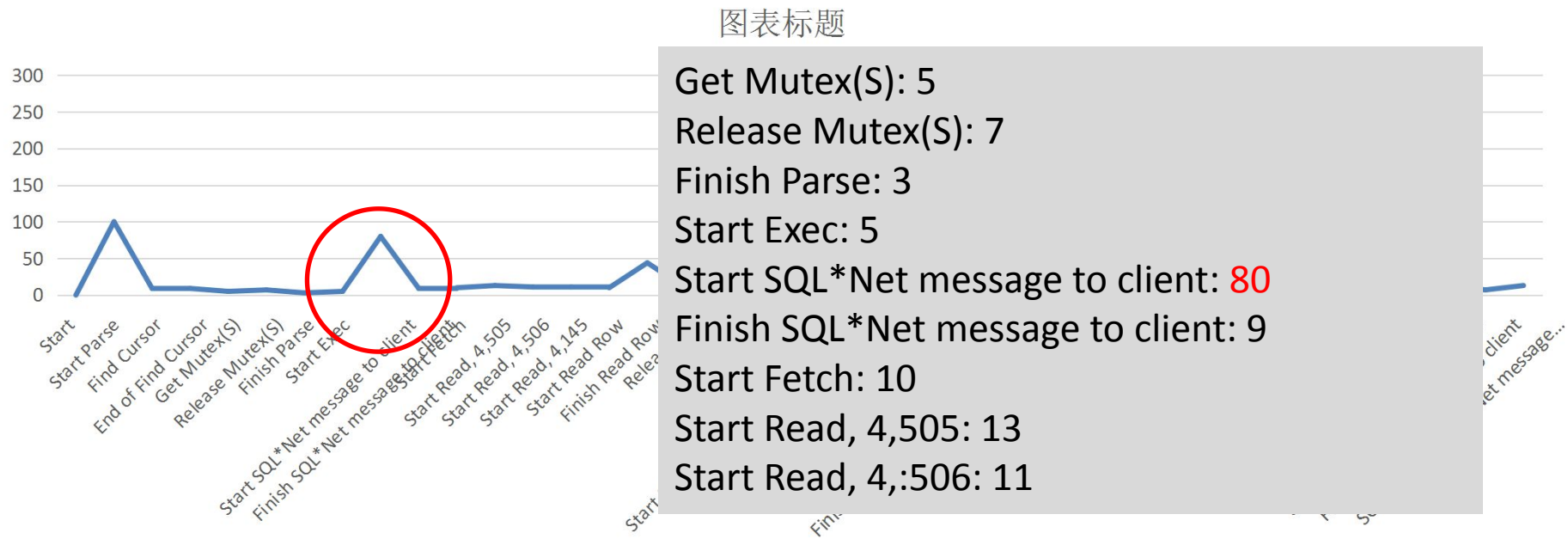
基于动态跟踪的运维工具

最耗时的部分，是从SQL*Net message from client期间的Start Sent到Finish Sent， Start Recv和Finish Recv。



基于动态跟踪的运维工具

示例SQL：select * from t1 where id=1，两层索引，逻辑读，软软解析。最耗时的部分：



基于动态跟踪的运维工具

sqll.stp.txt

包含了13个软解析、软软解析和逻辑读相关的重要的函数。
Kskthbwt：开始等待事件。11.2.0.3后，第三个参数为等待事件编号。
kskthewt ： 结束等待事件。
kksParseCursor ： 开始解析Cursor。
kgscFindCursor ： 开始尝试在共享池中发现Cursor。
kgxSharedExamine ： 开始持有共享Mutex。
opiexe：开始执行Cursor。
opifch2：开始抓取的第二阶段。
kcbgtcr：逻辑读的起始。
kpofdr：读行数据。
kghfrempty_ex：合并共享池中的Chunk。
kcbrls：某些逻辑读的结束。
nttfpwr：发送数据到客户端。
nttfprd：从客户端接收数据。