第 6 章　Buffer Cache 与 Shared Pool 原理

Buffer Cache 与 Shared Pool 是 SGA 中的最重要和最复杂的两个部分，在此我们拿出一点篇幅来对 Buffer Cache 和 Shared Pool 进行一点深入探讨。

6.1　Buffer Cache 原理

Buffer Cache 是 Oracle SGA 中一个重要部分，通常的数据访问和修改都需要通过 Buffer Cache 来完成。当一个进程需要访问数据时，首先需要确定数据在内存中是否存在，如果数据在 Buffer 中存在，则需要根据数据的状态来判断是否可以直接访问还是需要构造一致性读取；如果数据在 Buffer 中不存在，则需要在 Buffer Cache 中寻找足够的空间以装载需要的数据，如果 Buffer Cache 中找不到足够的内存空间，则需要触发 DBWR 去写出脏数据，释放 Buffer 空间。

这样一个过程，描述起来并不复杂，但是在数据库的处理过程中实际上是相当复杂的。在以上的描述中，有几个问题是我们需要考虑的，首先，Oracle 如何才能快速定位到 Buffer 中是否存在需要的数据呢？如果请求的数据在 Cache 中不存在，那么 Oracle 又是如何去 Buffer Cache 中快速寻找内存空间呢？

6.1.1　LRU 与 LRUW List

我们知道，在 Buffer Cache 中，Oracle 通过几个链表进行内存管理，其中最为熟知的是 LRU List 和 LRUW List（也经常被称为 Write/Dirty List），各种 List 上存放的是指向具体的 Buffer 的指针等信息。

从 Oracle8 开始，为了实施增量检查点，Oracle 还引入了检查点队列- Checkpoint Queue 和文件队列 – File Queue；从 Oracle8i 开始，由于异步 DBWn 的引入，现在关于各种 List 以及 Queue 的更为精确的概念是工作集（WS - Working Sets），在每个 WS 中包含几个不同功能的 List，每个 List 都通过 Cache Buffers LRU CHAIN Latch 进行保护，当使用多个 DBWR 进程时（通过 DB_WRITER_PROCESSES 参数可以设置数据库使用多个 DBWR 进程），数据库中会存在多个 WS，同时当使用 Buffer Cache 的多缓冲池技术时，每个独立的缓冲池也会存在各自独立的 WS。

LRU List 用于维护内存中的 Buffer，按照 LRU 算法进行管理（在不同版本中，管理方式有所不同），数据库初始化时，所有的 Buffer 都被 Hash 到 LRU List 上管理。当需要从数据文件上读取数据时，首先要在 LRU List 上寻找 Free 的 Buffer，然后读取数据到 Buffer Cache

中；当数据被修改之后，状态变为 Dirty，就可以被移动至 LRUW List，LRUW List 上的都是候选的可以被 DBWR 写出到数据文件的 Buffer，一个 Buffer 要么在 LRU List 上，要么在 LRUW List 上存在，不能同时在这两个 List 上存在。

图 6-1 是 Buffer Cache 中 LRU 及 LRUW List 的简要示意图。

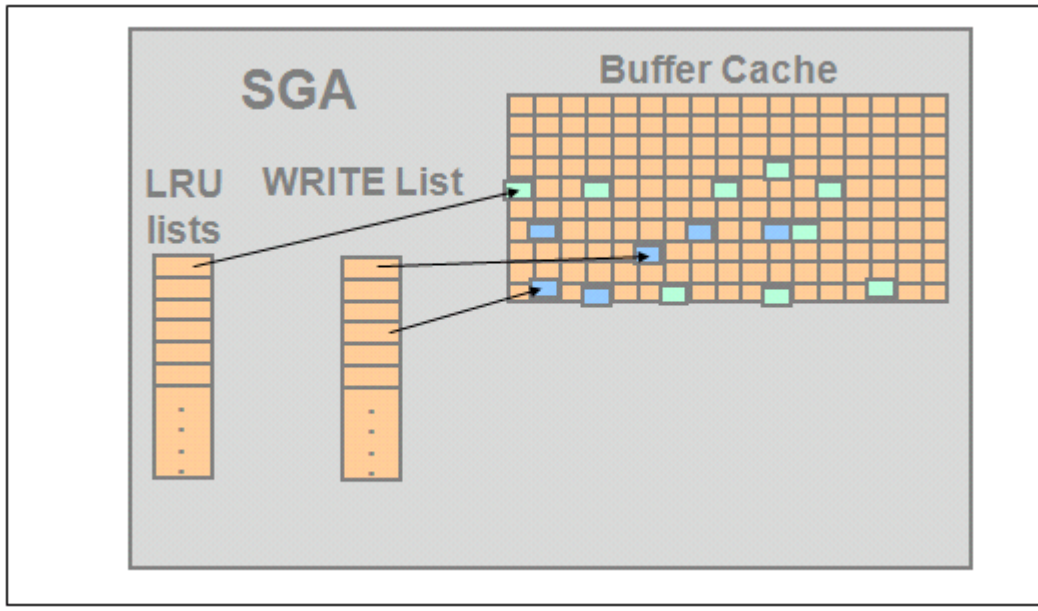


图 6-1　LRU 与 Write List

检查点队列（Checkpoint Queue）则负责按照数据块的修改顺序记录数据块，同时将 RBA 和数据块关联起来，这样在进行增量检查点时，数据库可以按照数据块修改的先后顺序将其写出，从而在进行恢复时，可以根据最后写出数据块及其相关的 RBA 开始进行快速恢复。在检查点触发时 DBWR 根据检查点队列执行写出，在其他条件触发时，DBWR 由 Dirty List 执行写出。检查点队列的内存在 Shared Pool 内存中分配：

```
SQL> select * from v$version where rownum <2;
BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
SQL> select * from v$sgastat where name='Checkpoint queue';
POOL         NAME                            BYTES
------------ -------------------------- ----------
shared pool  Checkpoint queue               128320
```

同样具有相关 Latch 对其进行保护：

```
SQL> col name for a40
SQL> select name ,gets,misses from v$latch where name like '%checkpoint queue%';
NAME                                          GETS     MISSES
---------------------------------------- ---------- ----------
active checkpoint queue latch                 98706          0
```

```
checkpoint queue latch                    1508091          0
```

可以通过图 6-2 来详细介绍一下 Buffer Cache 的原理及使用。

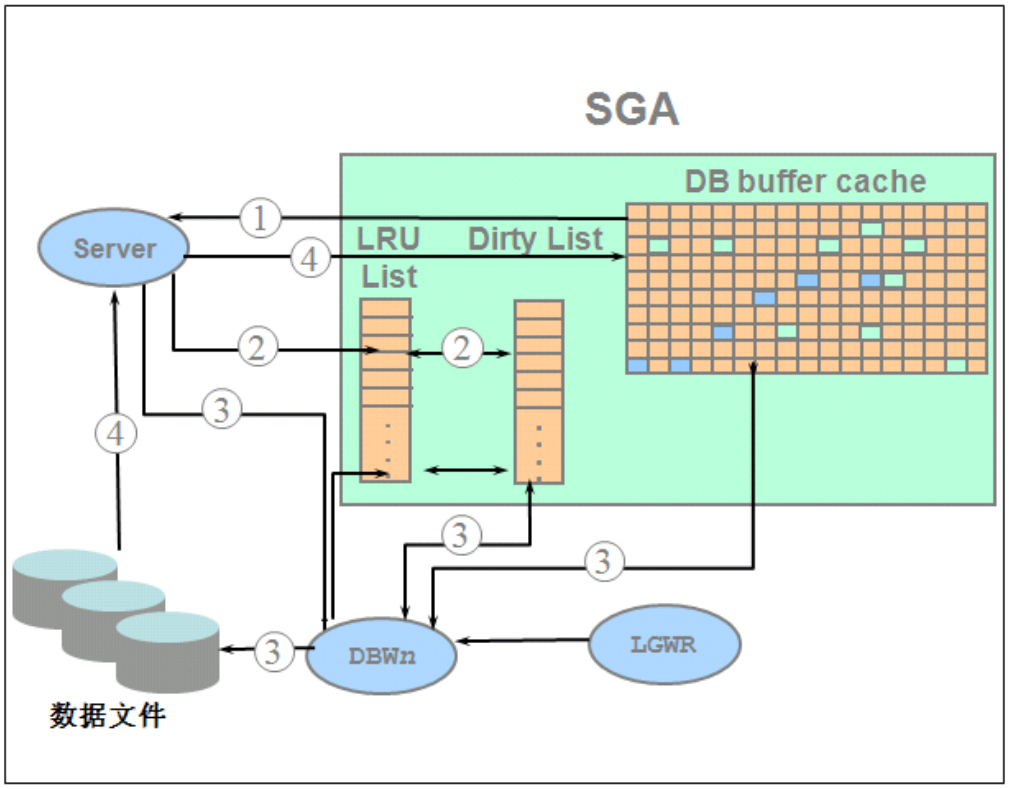


图 6-2　Buffer Cache 的原理及使用

1. 当一个 Server 进程需要读数据到 Buffer Cache 中时，首先必须判断该数据在 Buffer 中是否存在（图中①所示过程），如果存在且可用，则获取该数据，同时根据 LRU 算法增进其访问计数；如果 Buffer 中不存在该数据，则需要从数据文件上进行读取。
2. 在读取数据之前，Server 进程需要扫描 LRU List 寻找 Free 的 Buffer，扫描过程中 Server 进程会把发现的所有已经被修改过的 Buffer 注册到 LRUW List 上（图中②所示过程），这些 Dirty Buffer 随后可以被写出到数据文件
3. 如果 LRUW Queue 超过了阀值，Server 进程就会通知 DBWn 去写出脏数据（图中③所示过程）；

   这也是触发 DBWn 写的一个条件，这个阀值曾经提到是 25%，也就是当 Dirty Queue 超过 25%满就会触发 DBWn 的写操作：

```
SQL> select kvittag,kvitval,kvitdsc from x$kvit
  2  where kvittag='kcbldq';
KVITTAG                KVITVAL KVITDSC
-------------------- ---------- ----------------------------------------------------------
kcbldq                         25 large dirty queue if kcbclw reaches this
```

如果 Server 进程扫描 LRU 超过一个阀值仍然不能找到足够的 Free Buffer，将停

止寻找，转而通知 DBWn 去写出脏数据，释放内存空间。

同样这个阀值可以从以上字典表中查询得到，这个数字是 40%，也就是说当 Server 进程扫描 LRU 超过 40%还没能找到足够的 Free Buffer 就会停止搜索，通知 DBWn 执行写出，这是进程会处于 free buffer wait 等待：

```
SQL> select kvittag,kvitval,kvitdsc from x$kvit
  2  where kvittag='kcbfsp';
KVITTAG                KVITVAL KVITDSC
-------------------- ---------- ----------------------------------------------------------
kcbfsp                         40 Max percentage of LRU list foreground can scan for free
```

同时我们知道，由于增量检查点的引入，DBWn 也会主动扫描 LRU List，将发现的 Dirty Buffer 注册到 Dirty List 以及 Checkpoint Queue，这个扫描也受一个内部约束，在 Oracle9iR2 中，这个比例是 25%：

```
SQL> select kvittag,kvitval,kvitdsc from x$kvit
  2  where kvittag='kcbdsp';
KVITTAG                KVITVAL KVITDSC
-------------------- ---------- ----------------------------------------------------------
kcbdsp                         25 Max percentage of LRU list dbwriter can scan for dirty
```

4. 找到足够的 Buffer 之后，Server 进程就可以将 Buffer 从数据文件读入 Buffer Cache（图中④所示过程）

5. 如果读取的 Block 不满足读一致性需求，则 Server 进程需要通过当前 Block 版本和回滚段构造前镜像返回给用户。

从 Oracle 8i 开始，LRU List 和 LRUW List 又分别增加了辅助 List（AUXILIARY List），用于提高管理效率。引入了辅助 List 之后，当数据库初始化时，Buffer 首先存放在 LRU 的辅助 List 上（AUXILIARY RPL_LST），当被使用后移动到 LRU 主 List 上（MAIN RPL_LST），这样当用户进程搜索 Free Buffer 时就可以从 LRU-AUX List 开始，而 DBWR 搜索 Dirty Buffer 时，则可以从 LRU-Main List 开始，从而提高了搜索效率和数据库性能。

可以通过如下命令转储 Buffer Cache 的内容，从而清晰的看到以上描述的数据结构：

```
alter session set events 'immediate trace name buffers level 4';
```

不同 level 转储的内容详细程度不同，此命令的可用级别主要有 1～10 级，其中各级别的含义如下。

- Level 1：仅包含 Buffer Headers 信息。
- Level 2：包含 Buffer Headers 和 Buffer 概要信息转储。
- Level 3：包含 Buffer Headers 和完整 Buffer 内容转储。
- Level 4：Level 1 + Latch 转储 + LRU 队列。
- Level 5：Level 4 + Buffer 概要信息转储。
- Level 6 和 Level 7：Level 4 + 完整的 Buffer 内容转储。
- Level 8：Level 4 + 显示 users/waiters 信息。
- Level 9：Level 5 + 显示 users/waiters 信息。
- Level 10：Level 6 + 显示 users/waiters 信息。

转储仅限于在测试环境中使用，转储的跟踪文件可能非常巨大，为获取完整的跟踪文件，建议设置初始化参数 max_dump_file_size 为 UNLIMITED。

本章节选取的环境为：

```
SQL> select * from v$version;
BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
PL/SQL Release 11.1.0.6.0 - Production
CORE    11.1.0.6.0      Production
TNS for Linux: Version 11.1.0.6.0 - Production
NLSRTL Version 11.1.0.6.0 - Production
```

主要相关参数设置为：

```
SQL> show parameter max_dump
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
max_dump_file_size                   string      UNLIMITED
SQL> show parameter memory_target
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
memory_target                        big integer 500M
```

从 Level 4 级跟踪文件的开头部分可以获得如下信息，这是记录的不同 List 的 Prev 和 Next 定位信息。其中 WS 就是指 Working Sets，注意 WSID 指不同 WS 的编号：

```
Dump of buffer cache at level 4 for tsn=2147483647, rdba=0
  (WS) size: 0 wsid: 1 state: 0
    (WS_REPL_LIST) main_prev: 0x3e102164 main_next: 0x3e102164 aux_prev: 0x3e10216c aux_next:
0x3e10216ccurnum: 0 auxnum: 0
cold: 3e102164 hbmax: 0 hbufs: 0
    (WS_WRITE_LIST) main_prev: 0x3e102180 main_next: 0x3e102180 aux_prev: 0x3e102188 aux_next:
0x3e102188curnum: 0 auxnum: 0
    (WS_XOBJ_LIST) main_prev: 0x3e10219c main_next: 0x3e10219c aux_prev: 0x3e1021a4 aux_next:
0x3e1021a4curnum: 0 auxnum: 0
    (WS_XRNG_LIST) main_prev: 0x3e1021b8 main_next: 0x3e1021b8 aux_prev: 0x3e1021c0 aux_next:
0x3e1021c0curnum: 0 auxnum: 0
    (WS_REQ_LIST) main_prev: 0x3e1021d4 main_next: 0x3e1021d4 aux_prev: 0x3e1021dc aux_next:
0x3e1021dccurnum: 0 auxnum: 0
  (WS) fbwanted: 0
  (WS) bgotten: 0 sumwrt: 0
  (WS) pwbcnt: 0
```

接下来是具体的 List 链表信息，注意这里存在多条 NULL 列表，这是为 Buffer Cache 不

同部分（Keep 池、Recycle 池以及不同 block_size 大小的内存使用）预分配的 List：

```
MAIN RPL_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN RPL_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY RPL_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY RPL_LST Queue header (PREV_DIRECTION)[NULL]
MAIN WRT_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN WRT_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY WRT_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY WRT_LST Queue header (PREV_DIRECTION)[NULL]
MAIN XOBJ_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN XOBJ_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY XOBJ_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY XOBJ_LST Queue header (PREV_DIRECTION)[NULL]
MAIN XRNG_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN XRNG_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY XRNG_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY XRNG_LST Queue header (PREV_DIRECTION)[NULL]
MAIN REQ_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN REQ_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY REQ_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY REQ_LST Queue header (PREV_DIRECTION)[NULL]
  (WS) size: 12948 wsid: 3 state: 0
    (WS_REPL_LIST) main_prev: 0x2d7e982c main_next: 0x2efeef1c aux_prev: 0x317ef0bc aux_next:
0x2efee48ccurnum: 12948 auxnum:
 3195
cold: 2d7e9f7c hbmax: 6450 hbufs: 6096
    (WS_WRITE_LIST) main_prev: 0x3e102888 main_next: 0x3e102888 aux_prev: 0x3e102890 aux_next:
0x3e102890curnum: 0 auxnum: 0
    (WS_XOBJ_LIST) main_prev: 0x3e1028a4 main_next: 0x3e1028a4 aux_prev: 0x3e1028ac aux_next:
0x3e1028accurnum: 0 auxnum: 0
    (WS_XRNG_LIST) main_prev: 0x3e1028c0 main_next: 0x3e1028c0 aux_prev: 0x3e1028c8 aux_next:
0x3e1028c8curnum: 0 auxnum: 0
    (WS_REQ_LIST) main_prev: 0x3e1028dc main_next: 0x3e1028dc aux_prev: 0x3e1028e4 aux_next:
0x3e1028e4curnum: 0 auxnum: 0
  (WS) fbwanted: 0
  (WS) bgotten: 31531 sumwrt: 38791
  (WS) pwbcnt: 0
```

从以上输出还可以看到，Buffer Cache 中除了 RPL_LST 和 WRT_LST 外还存在其他分类的 List，作用各不相同。Buffer Cache 的多缓冲池以及多 WS 结构如图 6-3 所示.

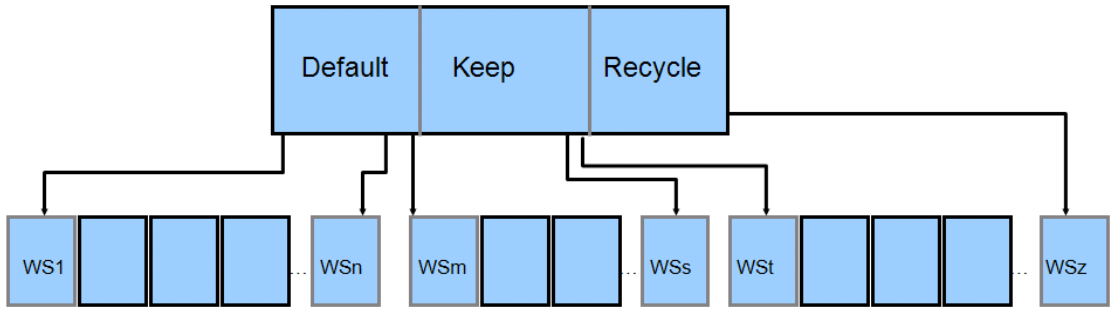


图 6-3　Buffer Cache 的多缓冲池

同时在 Level 4 级的专储中，再向下可以看到主要 RPL_LST 的队列信息，这也是链表的一个最直观表现：

```
MAIN RPL_LST Queue header (NEXT_DIRECTION)[0x2efeef1c,0x2d7e982c]
0x2efeeebc=>0x2f3ebdfc=>0x2f3ebecc=>0x2eff03dc=>0x2f3ebabc=>0x2eff04ac=>0x2f3ee0ec=>0x2eff064c
0x2f3ec6ec=>0x2eff07ec=>0x2f3f3ffc=>0x2eff0ccc=>0x2eff0a5c=>0x2f3ee69c=>0x2f3ee76c=>0x2eff175c
0x2eff100c=>0x2eff10dc=>0x2f3f0bfc=>0x2f3f592c=>0x2f3f4a8c=>0x2f3f48ec=>0x2eff182c=>0x2eff1b6c
0x2eff1a9c=>0x2f3f56bc=>0x2f3f7c1c=>0x2eff1d0c=>0x2f3fa31c=>0x31be670c=>0x317f38ac=>0x2eff2adc
0x2f3f802c=>0x2f3f9fdc=>0x31ff14ec=>0x2cff89ec=>0x31ff15bc=>0x31ff168c=>0x31ff182c=>0x31ff1eac
0x31ff1f7c=>0x2dfea59c=>0x2dfea80c=>0x2dfea9ac=>0x2dfeacec=>0x2dfeb0fc=>0x2dfeb29c=>0x2dfeb43c
```

6.1.2　Cache Buffers LRU Chain 闩锁竞争与解决

当用户进程需要读数据到 Buffer Cache 时或 Cache Buffer 根据 LRU 算法进行管理等，就不可避免的要扫描 LRU List 获取可用 Buffer 或更改 Buffer 状态，我们知道，Oracle 的 Buffer Cache 是共享内存，可以为众多并发进程并发访问，所以在搜索的过程中必须获取 Latch（Latch 是 Oracle 的一种串行锁机制，用于保护共享内存结构），锁定内存结构，防止并发访问损坏内存中的数据（我们必须认识到**对于数据的访问、Buffer 的存取就意味着多次的 Latch 访问，而过于严重的 Latch 竞争常常是系统的瓶颈所在**）。

这个用于锁定 LRU 的 Latch 就是我们经常见到的 Cache Buffers Lru Chain。

```
SQL> col name for a25
SQL> select addr,latch#,name,gets,misses,immediate_gets,immediate_misses
  2  from v$latch where name = 'cache buffers lru chain';
ADDR   LATCH# NAME                          GETS     MISSES IMMEDIATE_GETS IMMEDIATE_MISSES
-------- --- ------------------------ ---------- ---------- -------------- ----------------
1200BFA8  93 cache buffers lru chain     40851181      11972     4608605853          2965271
```

Cache Buffers Lru Chain Latch 存在多个子 Latch，其数量受隐含参数_db_block_lru_latches 控制：

```
SQL> @GetParDescrb.sql
Enter value for par: lru
```

```
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%lru%'
NAME                      VALUE               DESCRIB
------------------------------ -------------------- --------------------------------
_db_block_lru_latches          64                   number of lru latches
```

可以从 v$latch_children 视图察看当前各子 Latch 使用情况：

```
SQL> select addr,child#,name,gets,misses,immediate_gets igets,immediate_misses imisses
  2  from v$latch_children where name = 'cache buffers lru chain';
ADDR     CHILD# NAME                      GETS     MISSES      IGETS    IMISSES
-------- ------ ------------------------- --------- ---------- ---------- ----------
14C7E7F4     64 cache buffers lru chain         174          0          0          0
14C7E328     63 cache buffers lru chain         174          0          0          0
……
14C73678     27 cache buffers lru chain         174          0          0          0
14C731AC     26 cache buffers lru chain         174          0          0          0
14C72CE0     25 cache buffers lru chain         174          0          0          0
14C72814     24 cache buffers lru chain     5168833       2009  570411112     391045
14C72348     23 cache buffers lru chain     5179461       1685  577348879     369675
14C71E7C     22 cache buffers lru chain     5079655       1526  570766961     368414
14C719B0     21 cache buffers lru chain     5082539       1466  579774239     373785
14C714E4     20 cache buffers lru chain     5077378       1288  572272107     360877
14C71018     19 cache buffers lru chain     5084936       1396  586932913     368418
14C70B4C     18 cache buffers lru chain     5120549       1309  572987405     363633
14C70680     17 cache buffers lru chain     5069659       1296  578336549     369442
14C701B4     16 cache buffers lru chain         174          0          0          0
……
14C6C358      3 cache buffers lru chain         174          0          0          0
14C6BE8C      2 cache buffers lru chain         174          0          0          0
14C6B9C0      1 cache buffers lru chain         174          0          0          0

64 rows selected.
```

如果该 Latch 竞争激烈，通常有如下方法可以采用：

（1） 适当增大 Buffer Cache，这样可以减少读数据到 Buffer Cache 的机会，减少扫描 Lru List 的竞争。

（2） 可以适当增加 LRU Latch 的数量，修改_db_block_lru_latches 参数可以实现，但是该参数通常来说是足够的，除非在 Oracle Support 的建议下或确知该参数将带来的影响，否则不推荐修改。

（3） 通过多缓冲池技术，可以减少不希望的数据老化和全表扫描等操作对于 Default 池的冲击，从而可以减少竞争。

（4）　优化 SQL，减少数据读取，从而减少对于 LRU List 的扫描。

6.1.3　Cache Buffer Chain 闩锁竞争与解决

在 LRU 和 Dirty List 这两个内存结构之外，Buffer Cache 的管理还存在另外两个重要的数据结构：Hash Bucket 和 Cache Buffer Chain。

1．Hash Bucket 和 Cache Buffer Chain

我们可以想象，如果所有的 Buffer Cache 中的所有 Buffer 都通过同一个结构管理，当需要确定某个 Block 在 Buffer 中是否存在时，将需要遍历整个结构，性能会相当低下。

为了提高效率，Oracle 引入了 Bucket 的数据结构，Oracle 把管理的所有的 Buffer 通过一个内部的 Hash 算法运算后存放到不同 Hash Bucket 中，这样通过 Hash Bucket 进行分割之后，众多的 Buffer 被分布到一定数量的 Bucket 之中，当用户需要在 Buffer 中定位数据是否存在时，只需要通过同样的算法获得 Hash 值，然后到相应的 Bucket 中查找少量的 Buffer 即可确定。

每个 Buffer 的存放的 Bucket 由 Buffer 的数据块地址（DBA，Data Block Address）运算决定。在 Bucket 内部，通过 Cache Buffer Chain（它是一个双向链表）将所有的 Buffer 通过 Buffer Header 信息联系起来。

Buffer Header 存放的是对应数据块的概要信息，包括数据块的文件号、块地址、状态等。在判断数据块在 Buffer 中是否存在，通过检查 Buffer header 即可确定。

让我们通过一个现实的场景来回顾一下这个过程。

如果大家去过老一点的图书馆，查找过手工索引，你可能记得这样的场景：树立在你面前的是一排柜子（那是相当的壮观），柜子又被分为很多小的抽屉，抽屉上按照不同的分类方法标注了相关信息，比如按开头字母顺序，如果我们要查询 Oracle 相关书籍，就需要找到标记有“O”的抽屉。打开抽屉，我们会看到一系列的卡片，这些卡片通常被一根铁闩串起来（通常就是一个铁丝），每根卡片上会记录相关书籍的信息，可能包括书籍名称、作者、ISBN 号、出版日期等，当然这些卡片上还存储了一个重要的信息，就是书籍存放的书架位置信息，有了这个信息，通过翻阅这些卡片，就可以快速地找到想要的书籍，并且在需要时能够快速从图书馆浩如烟海的图书中找到我们需要的一本。

在这里，图书馆就是我们的 Buffer Cache，这个 Cache 可能因为“图书数量”的增加而不断扩大；每个抽屉都是一个 Bucket，这个 Bucket 中存放了根据一定的分类方式（也就是通过 Hash 运算）归入的图书信息，也就是 Buffer Header；抽屉中的每张卡片就是一个 Buffer Header，这些 Buffer Header 上记录了关于数据块的重要信息，如 DBA 等；这些卡片在 Bucket 中，通过一个铁闩串接起来，这就是 Cache Buffer Chain。

由于每个抽屉只有一根铁闩，如果很多同学都想翻阅这个链上的卡片，那么就产生了 Cache Buffer Chain 的竞争，先来到那个同学持有了 Latch 就能不停的翻阅，其他同学只好不停的来检查，当然如果检查次数多了（超过了_spin_count），也可以去休息室小憩一会，再来和其他同学争夺。

从 Oracle 9i 开始,对于 Cache Buffer Chain 的只读访问，其 Latch 可以被共享。也就是说，如果大家都只是翻一翻卡片，那么大家可以一起读，但是如果有人要借走这本书，那么就只能

独享这个 Latch 了。这就是 Buffer Cache 与 Latch 竞争。

由于 Buffer 根据 Buffer Header 进行散列，最终决定存入那一个 Hash Bucket，那么 Hash Bucket 的数量在一定程度上决定了每个 Bucket 中 Buffer 数量的多少，也就间接影响了搜索的性能。所以在不同版本中，Oracle 一直在修改算法，优化 Hash Bucket 的数量。我们可以想象，Bucket 的数量多一些，那么在同一时间就可以有更多的同学可以拿到不同的抽屉，进行数据访问；但是更多的抽屉，显然需要更多的存放空间，更多的管理成本，所以优化在什么时候都不是简单的一元方程。

Hash Bucket 的设置受一个隐含参数_ DB_BLOCK_HASH_BUCKETS 的影响。在 Oracle 7 和 Oracle 8 中，该参数缺省值为 DB_BLOCK_BUFFERS/4 的下一个素数。

在 Oracle 8i 中，该参数缺省为 db_block_buffers*2。

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
SQL> @GetParDescrb.sql
Enter value for par: hash_buckets
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%hash_buckets%'
NAME                           VALUE                DESCRIB
------------------------------ -------------------- ----------------------------------------
_db_block_hash_buckets         50000                Number of database block hash buckets
SQL> show parameter db_block_buffers
NAME                           TYPE    VALUE
---------------------------------- ------- ------------------------------
db_block_buffers                       integer 25000
```

通过以上的讨论可以知道，对应每个 Bucket，只存在一个 Chain，当用户试图搜索 Cache Buffer Chain 时，必须首先获得 Cache Buffer Chain Latch。

那么 Cache Buffer Chain Latch 的设置就同样值得研究了

在 Oracle 8i 之前，对于每一个 Hash Bucket，Oracle 使用一个独立的 Hash Latch 来维护，其缺省 Bucket 数量为 db_block_buffers/4 的下一个质数。

由于过于严重的热点块竞争，从 Oracle8i 开始，Oracle 改变了这个算法，首先 Bucket 数量开始增加，_db_block_hash_buckets 增加到 2*db_block_buffers，而_db_block_hash_latchs 的数量也发生了变化：

- 当 cache buffers 少于 2052 buffers。
  _db_block_hash_latches = power(2,trunc(log(2, db_block_buffers - 4) - 1))
- 当 cache buffers 多于 131075 buffers。
  _db_block_hash_latches = power(2,trunc(log(2, db_block_buffers - 4) - 6))
- 当 cache buffers 位于 2052～131075 buffers 之间。

_db_block_hash_latches　=　1024

我们看到，从 Oracle 8i 开始，_db_block_hash_buckets 的数量较以前增加了 8 倍，而 _db_block_hash_latchs 的数量增加有限，这意味着，每个 Latch 需要管理多个 Bucket，但是由于 Bucket 数量的多倍增加，每个 Bucket 上的 Block 数量得以减少，从而使少量 Latch 管理更多 Bucket 成为可能。

通过图 6-4 简要描述这个变化（其中省略了一些内容）。

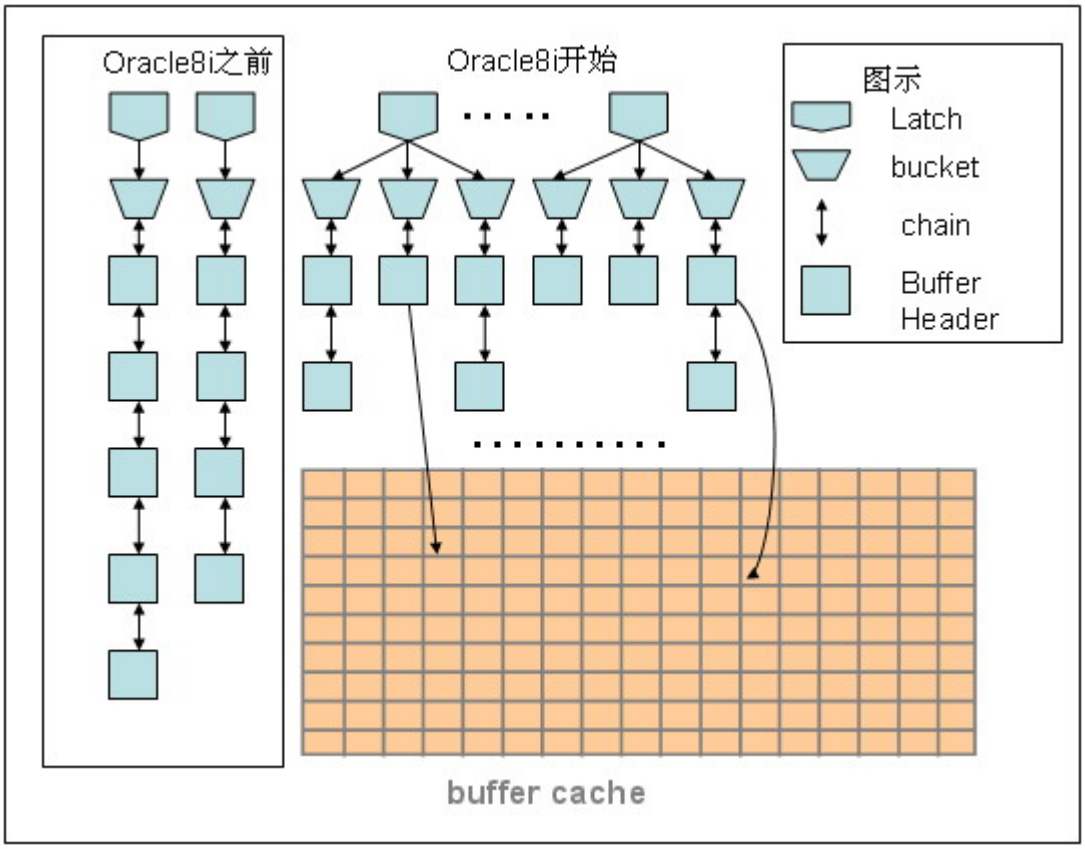


图 6-4　Oracle 8i 的变化

总结一下图 6-4 中所描述的内容。

（1）　从 Oracle 8i 开始，Bucket 的数量比以前大大增加；通过增加的 Bucket 的“稀释”使得每个 Bucket 上的 Buffer 数量大大减少。

（2）　在 Oracle 8i 之前，_db_block_hash_latches 的数量和 Bucket 的数量是一致的，每个 Latch 管理一个 Bucket；从 Oracle 8i 开始每个 Latch 需要管理多个 Bucket，由于每个 Bucket 上的 Buffer 数量大大降低，所以 Latch 的性能反而得到了提高。

（3）　每个 Bucket 存在一条 Cache Buffer Chain。

（4）　Buffer Header 上存在指向具体 Buffer 的指针。

以下是一个 Oracle8i 数据库的查询输出：

```
SQL> @GetHidPar.sql
Enter value for par: hash_latches
NAME                           VALUE                DESCRIB
------------------------------ -------------------- --------------------------------
_db_block_hash_latches         1024                 Number of database block hash latches
SQL> show parameter db_block_buffers
NAME                           TYPE    VALUE
------------------------------ ------- ------------------------------
db_block_buffers               integer   25000
```

让我们通过试验来验证以上的一些说明。首先测试库 db_cache_size 设置为 16M，此时的 _db_block_hash_buckets 为 4001：

```
SQL> show parameter db_cache_size
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
db_cache_size                        big integer 16777216
SQL> @GetHidPar.sql
Enter value for par: bucket
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%bucket%'
NAME                           VALUE                DESCRIB
------------------------------ -------------------- -------------------------------------
_db_block_hash_buckets         4001                 Number of database block hash buckets
```

转储一下 buffer，获得跟踪文件：

```
SQL> alter session set events 'immediate trace name buffers level 10';
Session altered.
SQL> !
[oracle@jumper udump]$ ls
eygle_ora_1197.trc
```

跟踪文件中的 Cache Buffer Chain 的数量正好是 4001：

```
[oracle@jumper udump]$ grep CHAIN eygle_ora_1197.trc |wc -l
   4001
```

某些 CHAIN 上可能没有 Buffer Header 信息（标记为 NULL），这些 Chain 的数据类似如下显示：

```
[oracle@jumper udump]$ grep CHAIN eygle_ora_1197.trc |head -20
CHAIN: 0 LOC: 0x0x532996b0 HEAD: [51ff3fac,51ff3fac]
CHAIN: 1 LOC: 0x0x532997d0 HEAD: [NULL]
CHAIN: 2 LOC: 0x0x532998f0 HEAD: [NULL]
CHAIN: 3 LOC: 0x0x53299a10 HEAD: [NULL]
CHAIN: 4 LOC: 0x0x53299b30 HEAD: [NULL]
```

```
CHAIN: 5 LOC: 0x0x53299c50 HEAD: [NULL]
CHAIN: 6 LOC: 0x0x53299d70 HEAD: [NULL]
CHAIN: 7 LOC: 0x0x53299e90 HEAD: [NULL]
CHAIN: 8 LOC: 0x0x53299fb0 HEAD: [51ff3ef0,51ff3ef0]
CHAIN: 9 LOC: 0x0x5329a0d0 HEAD: [NULL]
CHAIN: 10 LOC: 0x0x5329a1f0 HEAD: [NULL]
CHAIN: 11 LOC: 0x0x5329a310 HEAD: [NULL]
CHAIN: 12 LOC: 0x0x5329a430 HEAD: [NULL]
CHAIN: 13 LOC: 0x0x5329a550 HEAD: [NULL]
CHAIN: 14 LOC: 0x0x5329a670 HEAD: [NULL]
CHAIN: 15 LOC: 0x0x5329a790 HEAD: [NULL]
CHAIN: 16 LOC: 0x0x5329a8b0 HEAD: [517f6174,517e99b4]
CHAIN: 17 LOC: 0x0x5329a9d0 HEAD: [NULL]
CHAIN: 18 LOC: 0x0x5329aaf0 HEAD: [NULL]
CHAIN: 19 LOC: 0x0x5329ac10 HEAD: [NULL]
```

摘录一个 Chain 8 的数据给大家参考：

```
CHAIN: 8 LOC: 0x0x53299fb0 HEAD: [51ff3ef0,51ff3ef0]
    BH (0x0x51ff3ef0) file#: 1 rdba: 0x00400ac1 (1/2753) class 4 ba: 0x0x51e08000
      set: 3 dbwrid: 0 obj: 391 objn: -1
      hash: [53299fb0,53299fb0] lru: [51ff3ff4,51ff3e7c]
      LRU flags:
      ckptq: [NULL] fileq: [NULL]
      st: CR md: NULL rsop: 0x(nil) tch: 1
      cr:[[scn: 0x0000.000bed07],[xid: 0x0000.000.00000000],[uba: 0x00000000.0000.00],[cls:
0x0000.000bed07],[sfl: 0x0]]
      buffer tsn: 0 rdba: 0x00400ac1 (1/2753)
      scn: 0x0000.00000fd4 seq: 0x01 flg: 0x04 tail: 0x0fd41001
      frmt: 0x02 chkval: 0x4d32 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
```

这个 Chain 中存在一个 BH 信息，注意其中包含 **hash: [53299fb0,53299fb0] lru: [51ff3ff4,51ff3e7c]**。

- hash: [53299fb0,53299fb0]中的两个数据分别代表 X$BH 中的 NXT_HASH 和 PRV_HASH，也就是指同一个 Hash Chain 上的下一个 BH 的地址和上一个 Buffer 地址。如果某个 Chain 只包含一个 BH，那么这两个值将同时指向该 Chain 地址。
- lru: [51ff3ff4,51ff3e7c]中的两个数据分别代表 X$BH 中的 NXT_REPL 和 PRV_REPL，也就是 LRU 上的下一个 Buffer 和上一个 Buffer。

注意前面章节曾经提到，CKPTQ 和 FILEQ 也是两个队列，在 Buffer 修改后会记录相应的链接地址信息。

通常所说的 Buffer Header 是一个双向链就是从这里实现的，从 Oracle 8i 开始，由于 Bucket 数量的增加，通常我们不容易见到包含多个 BH 的 Bucket，以下是从我们的一个生产环境中转

储的 Buffer 信息，选取的 Chain 包含两个 Buffer Header 供读者参考：

```
CHAIN: 1598 LOC: 0x407200880 HEAD: [3c4f8bd00,3e9fc6000]
    BH (0x3c4f8bd00) file#: 52 rdba: 0x0d01d126 (52/119078) class 1 ba: 0x3c413a000
      set: 6 dbwrid: 0 obj: 148015 objn: 148015
      hash: [3e9fc6000,407200880] lru: [38bfbd468,3bffca068]
      LRU flags: hot_buffer
      ckptq: [NULL] fileq: [NULL]
      st: XCURRENT md: NULL rsop: 0x0 tch: 5
      LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]
      buffer tsn: 41 rdba: 0x0d01d126 (52/119078)
      scn: 0x0819.184596ea seq: 0x01 flg: 0x06 tail: 0x96ea0601
      frmt: 0x02 chkval: 0x5332 type: 0x06=trans data
...........
    BH (0x3e9fc6000) file#: 51 rdba: 0x0cc1c8ce (51/116942) class 1 ba: 0x3e9880000
      set: 6 dbwrid: 0 obj: 151861 objn: 151861
      hash: [407200880,3c4f8bd00] lru: [3a3fdf068,3fcfde168]
      LRU flags: hot_buffer
      ckptq: [NULL] fileq: [NULL]
      st: XCURRENT md: NULL rsop: 0x0 tch: 0
      LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]
      buffer tsn: 42 rdba: 0x0cc1c8ce (51/116942)
      scn: 0x0819.193f1d34 seq: 0x01 flg: 0x06 tail: 0x1d340601
      frmt: 0x02 chkval: 0xd8d1 type: 0x06=trans data
```

了解了以上管理算法我们很容易想象，如果大量进程对相同的 block 进程进行操作，那么必然引发 Cache Buffer Chain 的竞争，也就是通常所说的热点块的竞争。接下来从 Buffer Header 继续讨论。

2．X$BH 与 Buffer Header

Buffer Header 数据，可以从数据库的字典表中查询得到，这张字典表是：X$BH。X$BH 中的 BH 就是指 Buffer Headers，每个 Buffer 在 x$bh 中都存在一条记录：

```
Connected to:
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
With the Partitioning option
JServer Release 8.1.7.4.0 - 64bit Production
SQL> select count(*) from x$bh;
  COUNT(*)
----------
     25000
SQL> show parameter db_block_buffers;
```

```
NAME                                 TYPE     VALUE
------------------------------------ -------- ------------------------------
db_block_buffers                     integer  25000
```

Buffer Header 中存储每个 Buffer 容纳的数据块的文件号、块地址、状态等重要信息，可以将 Buffer Header 看作是 Buffer 的“名片”，通过这张名片，Buffer 的重要信息得以展现。

图 6-5 包含了 Buffer Header 上记录的信息，Hash Bucket 上的 Chian 连接起这些 BH。

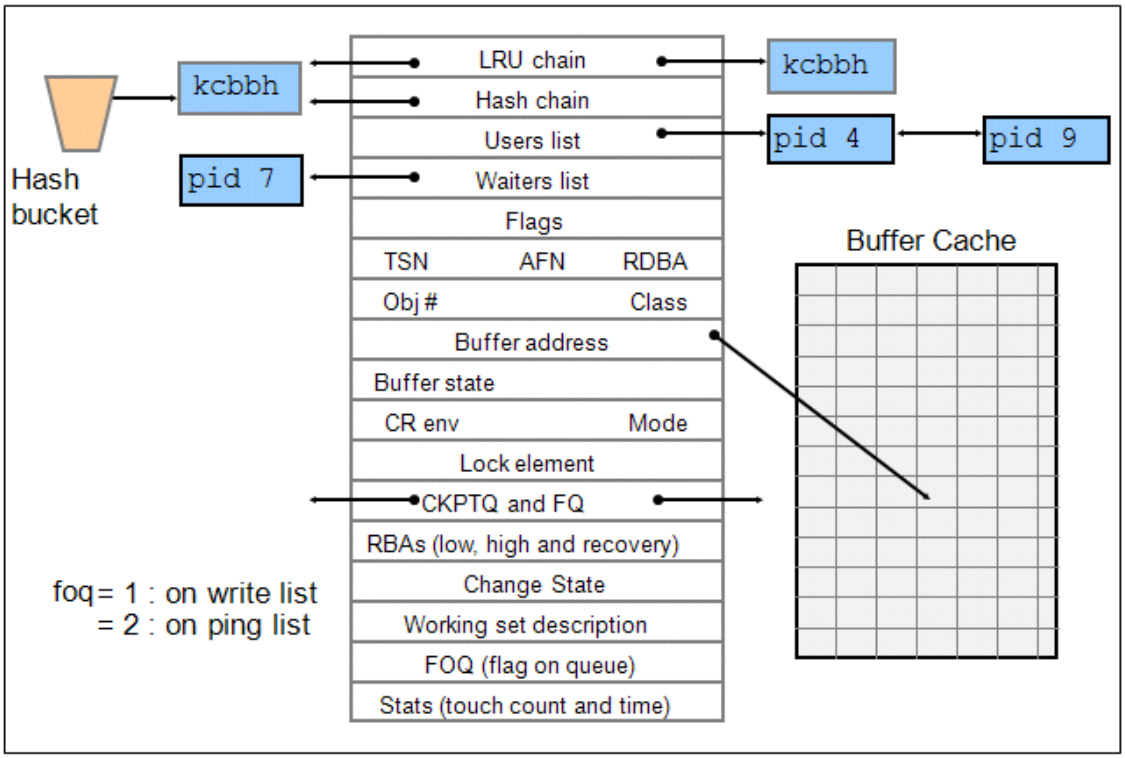


图 6-5　Buffer Header 上记录的信息

根据 Buffer Header 上记录的这些信息，结合 dba_extents 视图，可以很容易地找到每个 Buffer 对应的对象信息：

```
SQL> desc x$bh;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ADDR                                               RAW(8)
。。。。。
 HLADDR                                             RAW(8)
。。。。。。
 TS#                                                NUMBER
 FILE#                                              NUMBER
 DBARFIL                                            NUMBER
 DBABLK                                             NUMBER
```

```
CLASS                                                  NUMBER
STATE                                                  NUMBER
MODE_HELD                                              NUMBER
CHANGES                                                NUMBER
……
TCH                                                    NUMBER
```

X$BH 中还有一个重要字段 TCH，TCH 为 Touch 的缩写，表征一个 Buffer 的访问次数，Buffer 被访问的次数越多，说明该 Buffer 越“抢手”，也就可能存在热点块竞争的问题。

以下通过几个简单的查询供大家理解参考。以下查询用于获得当前数据库最繁忙的 Buffer（以下查询来自一个生产数据库）：

```
SQL> SELECT *
  2    FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
  3              FROM x$bh
  4          ORDER BY tch DESC)
  5   WHERE ROWNUM < 11;
ADDR                    TS#      FILE#    DBARFIL     DBABLK        TCH
---------------- ---------- ---------- ---------- ---------- ----------
FFFFFFFF7AFD1110         32         33         33       4732       1079
FFFFFFFF7AFD1110         32         33         33       4956       1079
FFFFFFFF7AFD1110         32         33         33         66       1078
FFFFFFFF7AFD1110         32         33         33       4492       1078
FFFFFFFF7AFD0E40         28         20         20        348       1078
FFFFFFFF7AFD0E40         28         20         20        355       1078
FFFFFFFF7AFD0E40         28         20         20        351       1078
FFFFFFFF7AFD0E40         28         20         20        349       1078
FFFFFFFF7AFD0E40         28         20         20        364       1078
FFFFFFFF7AFD0E40         28         20         20        360       1078
10 rows selected.
```

再结合 dba_extents 中的信息，可以查询得到这些热点 Buffer 都来自哪些对象：

```
SQL> SELECT e.owner, e.segment_name, e.segment_type
  2             FROM dba_extents e,
  3                  (SELECT *
  4                     FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
  5                                FROM x$bh
  6                            ORDER BY tch DESC)
  7                    WHERE ROWNUM < 11) b
  8            WHERE e.relative_fno = b.dbarfil
  9              AND e.block_id <= b.dbablk
 10              AND e.block_id + e.blocks > b.dbablk;
```

```
OWNER           SEGMENT_NAME                   SEGMENT_TYPE
--------------- ------------------------------ --------------------
BOSSV2          HY_AREA                        TABLE
BOSSV2          HYUIDX_PLATFORMID              INDEX
BOSSV2          HYUIDX_MOBILESEG               INDEX
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         HSOLAP_RPTJOB                  TABLE
HSQUERY         PK_HSOLAP_RPTJOB               INDEX

10 rows selected.
```

除了查询 X$BH 之外，其实也可以从 Buffer Cache 的转储信息中，看到 Buffer Header 的具体内容，以下信息来自 Level 1 级的 Buffer Dump：

```
Dump of buffer cache at level 1
BH (0x0x51fe8000) file#: 0 rdba: 0x00000000 (0/0) class 0 ba: 0x0x51c00000
  set: 3 dbwrid: 0 obj: 0 objn: 0
  hash: [532996b0,532996b0] lru: [51fe8104,532bec4c]
  LRU flags: on_auxiliary_list
  ckptq: [NULL] fileq: [NULL]
  st: FREE md: NULL rsop: 0x(nil) tch: 0
BH (0x0x51be8814) file#: 0 rdba: 0x00000200 (0/512) class 0 ba: 0x0x51816000
  set: 3 dbwrid: 0 obj: 0 objn: 0
  hash: [532996b8,532996b8] lru: [51be8918,51be87a0]
  LRU flags: on_auxiliary_list
  ckptq: [NULL] fileq: [NULL]
  st: FREE md: NULL rsop: 0x(nil) tch: 0
BH (0x0x513f87a4) file#: 1 rdba: 0x00405cde (1/23774) class 1 ba: 0x0x512ce000
  set: 3 dbwrid: 0 obj: 8 objn: 14
  hash: [532996c0,532996c0] lru: [513f8730,513f88a8]
  LRU flags: moved_to_tail
  ckptq: [NULL] fileq: [NULL]
  st: XCURRENT md: NULL rsop: 0x(nil) tch: 0
  flags: only_sequential_access
  LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]
```

在 Oracle 10g 之前，数据库的等待事件中，所有 Latch 等待被归入 Latch Free 等待事件中，在 Statspack 的 report 中，如果在 Top 5 等待事件中看到 Latch Free 这一等待处于较高的位置，

就需要 DBA 介入进行研究和解决。

3．热点块竞争与解决

接下来将通过一个生产环境的实际情况对热点块的问题进行分析和探讨。这是一个典型的性能低下的数据库，Top 5 等待事件都值得关注。注意到这个数据库中 Latch Free 是最严重的竞争。

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                             Wait     % Total
Event                                               Waits    Time (cs)    Wt Time
-------------------------------------------- ------------ ------------ -------
latch free                                       149,015   14,299,942   53.72
db file scattered read                         1,781,670    2,793,591   10.49
buffer busy waits                                 45,001    2,386,174    8.96
log file switch (checkpoint incomplete)           19,527    1,991,844    7.48
enqueue                                            6,523    1,809,849    6.80
                  -------------------------------------------------------------
```

由于 Latch Free 是一个汇总等待事件，我们需要从 v$latch 视图获得具体的 Latch 竞争主要是由哪些 Latch 引起的。在 Statspack Report 中同样存在这样一部分数据：

```
Latch Sleep breakdown for DB: HHCIMS  Instance: hhcims  Snaps: 154 -174
-> ordered by misses desc
                               Get                                  Spin &
Latch Name                    Requests     Misses Sleeps       Sleeps 1->4
-------------------------- -------------- ------- ------ -----------------------
cache buffers chains        5,186,232,773 617,065 80,524  610224/6250/158/433/0
library cache                 108,899,100 144,642 61,327 99020/31173/13558/891/0
row cache objects              83,115,714  46,477  1,610 44892/1565/17/3/0
shared pool                     7,091,076   7,403  3,669 5426/420/1464/93/0
cache buffers lru chain        18,885,002   6,405    400 6024/369/9/3/0
enqueues                       41,504,627   1,559    110 1486/59/6/8/0
redo writing                      370,920   1,524    178 1347/176/1/0/0
redo allocation                 4,170,274   1,184     95 1098/78/7/1/0
```

注意到 cache buffers chains 正是主要的 Latch 竞争。实际上，这个数据库在繁忙的时段，基本上处于停顿状态，大量进程等待 latch free 竞争，这些获得 Session 的等待事件可以很容易地从 v$session_wait 视图中查询得到：

```
       SID       SEQ# EVENT
---------- ---------- ------------------------------------------------
         4      14378 latch free
        43       1854 latch free
       176        977 latch free
```

```
      187       4393 latch free
      111       8715 latch free
      209      48534 latch free
      379       1008 latch free
      455       1974 latch free
      478      24713 latch free
      388        444 latch free
      369        855 latch free
      264        567 enqueue
      438        563 enqueue
      355        563 enqueue
      531        567 enqueue
      513        819 enqueue
      612         55 refresh controlfile command
```

如果需要具体确定热点对象，可以从 v$latch_children 中查询具体的子 Latch 信息，以下是一个生产环境中的部分信息摘录：

```
SQL> SELECT *
  2    FROM (SELECT   addr, child#, gets, misses, sleeps, immediate_gets igets,
  3                   immediate_misses imiss, spin_gets sgets
  4              FROM v$latch_children
  5             WHERE NAME = 'cache buffers chains'
  6          ORDER BY sleeps DESC)
  7   WHERE ROWNUM < 11;
ADDR               CHILD#   GETS     MISSES     SLEEPS      IGETS      IMISS SGETS
---------------- ---------- ---------- ---------- ---------- ---------- ---------- -----
0000000406F24860       1093 1759617932  518883201    2587204    9765719      32224     0
0000000406F180E0       1081  509119236   13183658     623513    9720398      16398     0
0000000406F19180       1082 3264036800    7000899     336205    9669611      10077     0
0000000406E7F500        934 3252332119   40742230     218898   10945581      43083     0
00000004072A4A30       1757 1356574143    8959581     134591    7165568      17482     0
0000000406B6A4D0        175 1252888934     873697     103414   14699959       5887     0
0000000406F1A220       1083 2190392429    2054269      98862    9692515       5444     0
0000000406E462A0        879  840225741    4716843      79799   10789540       5743     0
0000000406BAAF70        237 2697876674    7349818      78180    8168375       3124     0
0000000406B9A470        221 3051410653    6924588      69161    9942446       4118     0

10 rows selected.
```

X$BH 中还存在另外一个关键字段 HLADDR，即 Hash Chain Latch Address，这个字段可以和 v$latch_child.addr 进行关联，这样就可以把具体的 Latch 竞争和数据块关联起来，再结合

dba_extents 视图，就可以找到具体的热点竞争对象。

到具体热点竞争对象之后，可以进一步地结合 v$sqlarea 或者 v$sqltext 视图，找到频繁操作这些对象的 SQL，对其进行优化，就可以缓解或解决热点块竞争的问题。通过以下查询可以实现以上的思想，获取当前持有最热点数据块的 Latch 及 Buffer 信息：

```
SQL> SELECT b.addr, a.ts#, a.dbarfil, a.dbablk, a.tch, b.gets, b.misses, b.sleeps
  2    FROM (SELECT *
  3            FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch, hladdr
  4                      FROM x$bh
  5                  ORDER BY tch DESC)
  6           WHERE ROWNUM < 11) a,
  7         (SELECT addr, gets, misses, sleeps
  8            FROM v$latch_children
  9           WHERE NAME = 'cache buffers chains') b
 10   WHERE a.hladdr = b.addr
 11  /
ADDR             TS#    DBARFIL DBABLK  TCH       GETS     MISSES     SLEEPS
---------------- --- ---------- ---------- ---- ---------- ---------- ----------
0000000406AF3670  42         51     209612  216 2068740950    1396285      18734
0000000406B14C70  42         51     209644  216 1840436663    1671667      20622
0000000406B36270  42         51     209676  216 3447942770    2985525      29424
0000000406B57870  42         51     209708  216 1902183007    1913809      18806
0000000406B78E70  42         51     209740  216 2035257361    2314202      21304
0000000406B9A470  42         51     209772  216 3061630199    6932017      69275
0000000406E962C0  41         50     558579  217  723102299      32807       1112
00000004071BDF70  41         50     190251  217 3409957883     657572       9196
00000004071F9310  41         50     587887  217 1492232461      88013       4418
000000040721A710  32         33       4703  224  756763990      69890       1470
10 rows selected.
```

利用前面提到的 SQL，可以找到这些热点 Buffer 的对象信息：

```
SQL> SELECT distinct e.owner, e.segment_name, e.segment_type
  2             FROM dba_extents e,
  3                  (SELECT *
  4                     FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
  5                                FROM x$bh
  6                            ORDER BY tch DESC)
  7                    WHERE ROWNUM < 11) b
  8            WHERE e.relative_fno = b.dbarfil
  9              AND e.block_id <= b.dbablk
 10              AND e.block_id + e.blocks > b.dbablk;
```

```
OWNER           SEGMENT_NAME                    SEGMENT_TYPE
--------------- ------------------------------- --------------------
BOSSV2          HYUIDX_MOBILESEG                INDEX
BOSSV2          HY_PLATFORM                     TABLE
BOSSV2          MAIDX_BATTASK_USERLIST_DATSEND INDEX PARTITION
BOSSV2          MAIDX_BATTASK_USERLIST_STATUS  INDEX PARTITION
HSQUERY        HSOLAP_RPTJOB                  TABLE
```

结合 v$sqltext 或 v$sqlarea，可以找到操作这些对象的相关 SQL，让我们继续查询：

```
SQL> break on hash_value skip 1
SQL> SELECT /*+ rule */ hash_value,sql_text
  2      FROM v$sqltext
  3     WHERE (hash_value, address) IN (
  4               SELECT a.hash_value, a.address
  5                 FROM v$sqltext a,
  6                      (SELECT DISTINCT a.owner, a.segment_name, a.segment_type
  7                                  FROM dba_extents a,
  8                                       (SELECT dbarfil, dbablk
  9                                          FROM (SELECT   dbarfil, dbablk
 10                                                    FROM x$bh
 11                                                ORDER BY tch DESC)
 12                                         WHERE ROWNUM < 11) b
 13                                WHERE a.relative_fno = b.dbarfil
 14                                  AND a.block_id <= b.dbablk
 15                                  AND a.block_id + a.blocks > b.dbablk) b
 16                WHERE upper(a.sql_text) LIKE '%' || b.segment_name || '%'
 17                  AND b.segment_type = 'TABLE')
 18  ORDER BY hash_value, address, piece
 19  /

HASH_VALUE SQL_TEXT
---------- ------------------------------------------------------------------
 175397557 select sum(snum) from HS_UNISMS_ORDERLOG_XUDING f where f.snum>:
           1 and f.c_id=:2

 180602532 select * from HS_UNISMS_PAYLOG_99DVD t  where status=:1 and t.re
           pterrorcode=9999 order by t.paylog_id

 388500828 select * from HS_UNISMS_PAYLOG_99DVD t  where status=:1 and t.re
           pterrorcode=9999  and t.paytime >=to_date('2006-05-14 00:00:00',
```

```
           'yyyy-mm-dd hh24:mi:ss')  and t.paytime <=to_date('2006-05-16 00
           :00:00','yyyy-mm-dd hh24:mi:ss')  order by t.paylog_id

 724740081 select HS_SCSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

 773054905 select count(*) from HS_UNISMS_PAYLOG_99DVD t where t.paytime>=t
           o_date(:1,'yyyy-mm-dd hh24:mi:ss') and t.paytime<=to_date(:2,'yy
           yy-mm-dd hh24:mi:ss') and t.status=0

1071368535 select count(snum) from HS_UNISMS_ORDERLOG_XUDING f where f.snum
           >:1 and f.c_id=:2

1109655340 insert into UM_PUT_M_ORDER values (UM_M_ORDER_SEQ.NEXTVAL,:1,:2,
           :3,:4,sysdate,:5,:6,:7,:8,:9,:10,null,:11,:12,:13)

1451196467 select HS_ZJSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

1542951187 SELECT * FROM UM_PUT_M_ORDER

1669311552 insert into hm_user_info(USER_ID,PASSWORD,user_add2,user_seq_id)
            values(:1,:2,:3,HM_USER_INFO_SEQ.NEXTVAL)

1669935485 insert into HS_UNISMS_PAYLOG_99DVD values (HS_UNISMS_PAYLOG_99DV
           D_seq.NEXTVAL,:1,sysdate,:2,:3,:4,null,9999,9999,9999,9999,00000
           0,null,null)

1697705163 insert into hm_user_info(USER_ID,M_PHONE,password,user_seq_id) v
           alues(:1,:2,:3,HM_USER_INFO_SEQ.NEXTVAL)

1866895271 select * from UM_PUT_M_ORDER

1979706000 select mobile,c_id,status,snum from HS_UNISMS_ORDERLOG_XUDING t
           where t.status=:1

2152157475 select HS_UNISMS_ORDERLOG_SEQ.NEXTVAL from DUAL

2248706418 select * from UM_PUT_M_ORDER where MOBILEID=:1 and ORDERID=:2 or
           der by datetime desc
```

```
2336420675 select HS_JSSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

2963798180 select * from HS_UNISMS_PAYLOG_99DVD t  where status=:1 and t.re
           pterrorcode=9999  and t.paytime <=to_date('2006-05-15 12:00:00',
           'yyyy-mm-dd hh24:mi:ss')  order by t.paylog_id
。。。。。。。。。
83 rows selected.
```

找到这些 SQL 之后，剩下的问题就简单了，可以通过优化 SQL 减少数据的访问，避免或优化某些容易引起争用的操作（如 connect by 等操作）来减少热点块竞争。

6.2　Shared Pool 的基本原理

Shared Pool 是 Oracle SGA 设置中最复杂也是最重要的一部分内容，Oracle 通过 Shared Pool 来实现 SQL 共享、减少代码硬解析等，从而提高数据库的性能。在某些版本中，如果设置不当，Shared Pool 可能会极大影响数据库的正常运行。

在 Oracle 7 之前，Shared Pool 并不存在，每个 Oracle 连接都有一个独立的 Server 进程与之相关联，Server 进程负责解析和优化所有 SQL 和 PL/SQL 代码。典型的，在 OLTP 环境中，很多代码具有相同或类似的结构，反复的独立解析浪费了大量的时间以及资源，Oracle 最终认识到这个问题，并且从 PL/SQL 开始尝试把这部分可共享的内容进行独立存储和管理，于是 Shared Pool 作为一个独立的 SGA 组件开始被引入，并且其功能和作用被逐渐完善和发展起来。

在这里注意到，Shared Pool 最初被引入的目的，也就是它的本质功能在于**实现共享**。如果用户的系统代码是完全异构的（假设代码从不绑定变量，从不反复执行），那么就会发现，这时候 Shared Pool 完全就成为了一个负担，它在徒劳无功地进行无谓的努力：**保存代码、执行计划等期待重用**，并且客户端要不停的获取 Latch，试图寻找共享代码，却始终一无所获。如果真是如此，那这是我们最不愿看到的情况，Shared Pool 变得有害无益。当然这是极端，可是在性能优化中我们发现，大多数性能低下的系统都存在这样的通病：**代码极少共享**，**缺乏或从不实行变量绑定**。优化这些系统的根本方法就是**优化代码**，**使代码**（在保证性能的前提下）**可以充分共享**，**减少无谓的反复硬/软解析。**

实际上，Oracle 引入 Shared Pool 就是为了帮助我们实现代码的共享和重用。了解了这一点之后，我们在应用开发的过程中，也应该有意识地提高自己的代码水平，以期减少数据库的压力。这应该是对开发人员最基本的要求。

Shared Pool 主要由两部分组成，一部分是库缓存（Library Cahce），另一部分是数据字典缓存（Data Dictionary Cache）。Library Cache 主要用于存储 SQL 语句、SQL 语句相关的解析树、执行计划、PL/SQL 程序块（包括匿名程序块、存储过程、包、函数等）以及它们转换后能够被 Oracle 执行的代码等，这部分信息可以通过 v$librarycache 视图查询；至于 Data Dictionary Cache 主要用于存放数据字典信息，包括表、视图等对象的结构信息，用户以及对象权限信息，这部分信息相对稳定，在 Shared Pool 中通过字典缓存单独存放，字典缓存的内容是按行（Row）存储的（其他数据通常按 Buffer 存储），所以又被称为 Row Cache，其信息

可以通过 V$ROWCACHE 查询。

图 6-6 说明了 Shared Pool 各个部分协同工作以及与 Buffer Cache 的配合。

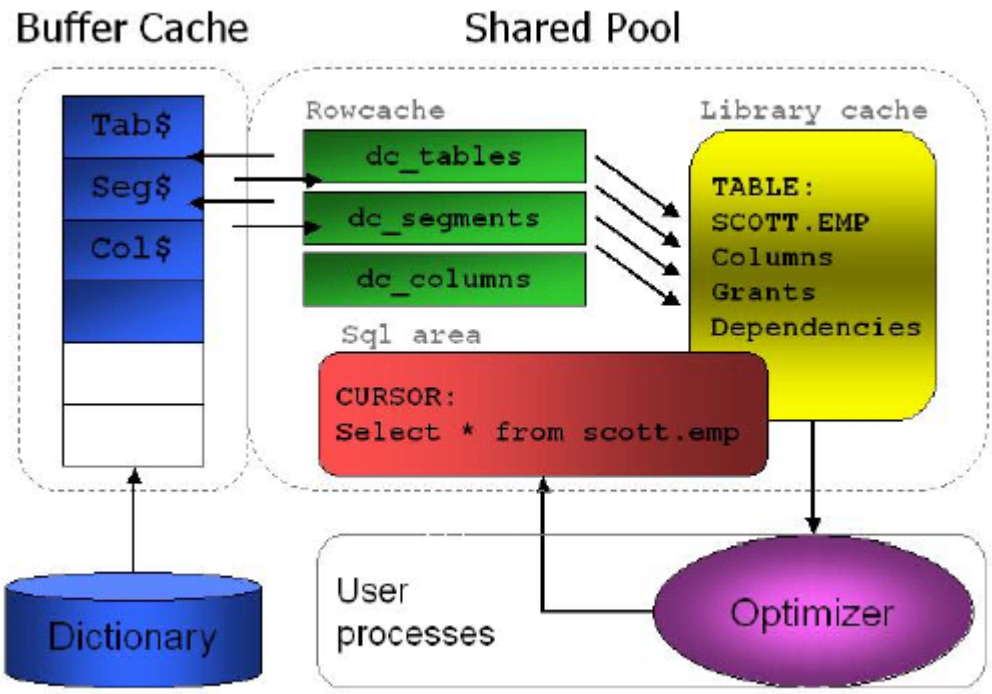


图 6-6　Shared Pool 与 Buffer Cache

除了以上两个部分外，从 Oracle Database 11g 开始，在 Shared Pool 中划出了另外一块内存用于存储 SQL 查询的结果集，称为 Result Cache Memory。以前 Shared Pool 的主要功能是共享 SQL，减少硬解析，从而提高性能，但是 SQL 共享之后，执行查询同样可能消耗大量的时间和资源，现在 Oracle 尝试将查询的结果集缓存起来，如果同一 SQL 或 PL/SQL 函数多次执行（特别是包含复杂运算的 SQL），那么缓存的查询结果可以直接返回给用户，不需要真正去执行运算，这样就又为性能带来了极大的提升。

6.2.1　Oracle 11g 新特性：Result Cache

结果集缓存（Result Cache）是 Oracle Database 11g 新引入的功能，除了可以在服务器端缓存结果集（Server Result Cache）之外，还可以在客户端缓存结果集（Client Result Cache）。下面着重介绍一下服务器端结果集缓存。

服务器端的 Result Cache Memory 由两部分组成。

- SQL Query Result Cache：存储 SQL 查询的结果集。
- PL/SQL Function Result Cache：用于存储 PL/SQL 函数的结果集。

Oracle 通过一个新引入初始化参数 result_cache_max_size 来控制该 Cache 的大小。如果

result_cache_max_size=0 则表示禁用该特性。参数 result_cache_max_result 则控制单个缓存结果可以占总的 Server Result Cache 大小的百分比。

```
SQL> select * from v$version where rownum <2;
BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
SQL> show parameter result_cache
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
client_result_cache_lag              big integer 3000
client_result_cache_size             big integer 0
result_cache_max_result              integer     5
result_cache_max_size                big integer 1280K
result_cache_mode                    string      MANUAL
result_cache_remote_expiration       integer     0
```

上面显示的参数中 result_cache_mode 用于控制 Server result cache 的模式，该参数有 3 个可选设置。

- 设置 auto：则优化器会自动判断是否将查询结果缓存。
- 设置 manual：则需要通过查询提示 result_cache 来告诉优化器是否缓存结果。
- 设置 force ：则尽可能地缓存查询结果（通过提示 no_result_cache 可以拒绝缓存）。

下面通过测试来看一下这一新特性的使用及优势所在，首先创建一张测试表：

```
SQL> connect eygle/eygle
Connected.
SQL> create table eygle as select * from dba_objects;
Table created.
```

在以前版本中，第一次执行该 SQL 可以看到 consistent gets 和 physical reads 大致相同：

```
SQL> set autotrace on
SQL> select count(*) from eygle;
  COUNT(*)
----------
     15993
Statistics
----------------------------------------------------------
         28  recursive calls
          0  db block gets
        282  consistent gets
        217  physical reads
```

再次执行同样查询时，由于数据 Cache 在内存中，physical reads 会减少到 0，但是 consistent gets 很难降低：

```
SQL> select count(*) from eygle;
  COUNT(*)
----------
     15993
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        221  consistent gets
          0  physical reads
```

现在再来看看在 Server Result Cache 下 Oracle 的行为，首先在 result_cache_mode 参数设置为 MANUAL 时：

```
SQL> show parameter result_cache_mode
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
result_cache_mode                     string      MANUAL
```

需要在 SQL 语句中手工指定 Cache，这需要通过加入一个 hints 来实现，这个 hints 是 result_cache：

```
SQL> select /*+ result_cache */ count(*) from eygle;
  COUNT(*)
----------
    15993
Execution Plan
----------------------------------------------------------
Plan hash value: 3602634261
-------------------------------------------------------------------------------------------
| Id  | Operation          | Name                        | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |                              |    1 |    64  (0)| 00:00:01 |
|  1 |  RESULT CACHE      | 76rwwyazv6t6c39f1d8rrqh8rb |      |            |          |
|  2 |  SORT AGGREGATE    |                              |    1 |            |          |
|  3 |    TABLE ACCESS FULL| EYGLE                       | 14489 |    64  (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=1; dependencies=(EYGLE.EYGLE);
  attributes=(single-row); name="select /*+ result_cache */ count(*) from eygle"

Statistics
```

```
----------------------------------------------------------
          4  recursive calls
          0  db block gets
        280  consistent gets
          0  physical reads
```

注意到这个执行计划已经和以往的不同，RESULT CACHE 以 76rwwyazv6t6c39f1d8rrqh8rb 名称创建。那么在接下来的查询中，这个 Result Cache 就可以被利用：

```
SQL> select /*+ result_cache */ count(*) from eygle;
  COUNT(*)
----------
    15993
Execution Plan
----------------------------------------------------------
Plan hash value: 3602634261

-------------------------------------------------------------------------------------------
| Id  | Operation          | Name                       | Rows  | Cost (%CPU)| Time    |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |                             |    1 |    64  (0) | 00:00:01 |
|  1 |  RESULT CACHE       | 76rwwyazv6t6c39f1d8rrqh8rb |      |            |          |
|  2 |  SORT AGGREGATE     |                             |    1 |            |          |
|  3 |    TABLE ACCESS FULL| EYGLE                       | 14489|    64  (0)| 00:00:01  |
-------------------------------------------------------------------------------------------
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=1; dependencies=(EYGLE.EYGLE);
  attributes=(single-row); name="select /*+ result_cache */ count(*) from eygle"

Note
-----
  - dynamic sampling used for this statement

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
```

在这个利用到 Result Cache 的查询中，consistent gets 减少到 0，直接访问结果集，不再需要执行 SQL 查询。这就是 Result Cache 的强大之处。

在以上测试中，当 result_cache_mode 设置为 MANUAL 时，只有使用 hints 的情况下，Oracle 才会利用缓存结果集；而如果将 result_cache_mode 设置为 AUTO，Oracle 如果发现缓冲结果集已经存在，那么就会自动使用。但是如果缓冲结果集不存在，Oracle 并不会自动进行缓冲，只有使用 HINT 的情况下，Oracle 才会将执行的结果集缓存。

可以通过查询 v$result_cache_memory 视图来看 Cache 的使用情况：

```
SQL> select * from V$RESULT_CACHE_MEMORY
 2  where FREE='NO';
      ID      CHUNK    OFFSET FRE  OBJECT_ID  POSITION
--------- ---------- ---------- --- ---------- ----------
       0          0          0 NO          0          0
       1          0          1 NO          1          0
```

通过 V$RESULT_CACHE_STATISTICS 可以查询 Result Cache 的统计信息：

```
SQL> select * from V$RESULT_CACHE_STATISTICS;
       ID NAME                                VALUE
---------- ------------------------------ ----------
       1 Block Size (Bytes)                 1024
       2 Block Count Maximum                 992
       3 Block Count Current                  32
       4 Result Size Maximum (Blocks)         49
```

V$RESULT_CACHE_OBJECTS 记录了 Cache 的对象：

```
SQL> SELECT ID,TYPE,NAME,BLOCK_COUNT,ROW_COUNT FROM V$RESULT_CACHE_OBJECTS;
       ID TYPE      NAME                           BLOCK_COUNT  ROW_COUNT
---------- ---------- ------------------------------ ----------- ----------
       0 Dependency EYGLE.EYGLE                              1          0
       1 Result    select /*+ result_cache */ cou            1          1
                     nt(*) from eygle
```

一个新的系统包被引入，DBMS_RESULT_CACHE 可以用于执行关于 Result Cache 的管理：

```
SQL> set serveroutput on
SQL> exec dbms_result_cache.memory_report
R e s u l t  C a c h e  M e m o r y  R e p o r t
[Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 992K bytes (992 blocks)
Maximum Result Size = 49K bytes (49 blocks)
[Memory]
Total Memory = 100836 bytes [0.059% of the Shared Pool]
```

```
... Fixed Memory = 5132 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 95704 bytes [0.056% of the Shared Pool]
....... Overhead = 62936 bytes
....... Cache Memory = 32K bytes (32 blocks)
.......... Unused Memory = 30 blocks
.......... Used Memory = 2 blocks
.............. Dependencies = 1 blocks (1 count)
.............. Results = 1 blocks
................. SQL    = 1 blocks (1 count)

PL/SQL procedure successfully completed.
```

6.2.2　Shared Pool 的设置说明

Shared Pool 的大小可以通过初始化参数 shared_pool_size 设置。对于 Shared Pool 的设置，一直以来是最具有争议的一部分内容。当然从 Oracle 10g 开始，如果使用自动共享内存管理，则无需单独设置共享池大小，这是 Oracle 10g 的一大进步。

在这里首先来看一下 Oracle 共享池技术的演进。在 Oracle 10g 之前在共享池的设置上存在很多不同声音，一方面很多人建议可以把 Shared Pool 设置得稍大，以充分 Cache 代码和避免 ORA-04031 错误的出现；另一方面又有很多人建议不能把 Shared Pool 设置得过大，因为过大可能会带来管理上的额外负担，从而会影响数据库的性能。

至于哪一种说法更为准确，这个管理上的额外负担究竟指什么，这并不是一句话就能予以定论的，下面我通过一点内部分析，试图从内部原理上回答这个问题。同时探讨一下共享池技术的变迁。

在下面的测试中用到了 Shared Pool 的转储，所以首先需要了解一下相关的命令。可以通过如下命令转储 Shared Pool 共享内存的内容：

```
SQL> alter session set events 'immediate trace name heapdump level 2';
Session altered.
```

本测试中引用的两个 trace 文件：

Oracle9iR2:

```
SQL> @gettrcname
TRACE_FILE_NAME
------------------------------------------------------------------------------
/opt/oracle/admin/hsjf/udump/hsjf_ora_24983.trc
```

Oracle8.1.5:

```
SQL> @gettrcname
TRACE_FILE_NAME
------------------------------------------------------------------------------
/usr/oracle8/admin/guess/udump/guess_ora_22038.trc
```

注意 alter session set events 'immediate trace name heapdump level 2'是一条内部命令，指定 Oracle 把 Shared Pool 的内存结构在 Level 2 级转储出来。

转储的内容被记录在一个 trace 文件中，这个 trace 文件可以在 undmp 目录下找到。为了比较 Oracle 8i 以及 Oracle 9i 的不同，文中引用了两个版本的跟踪文件。

其中 gettrcname.sql 是我用来获取 trace 文件名称的一个脚本，代码如下：

```
SELECT a.VALUE || b.symbol || c.instance_name || '_ora_' || d.spid || '.trc' trace_file_name
  FROM (SELECT VALUE FROM v$parameter WHERE NAME = 'user_dump_dest') a,
       (SELECT SUBSTR (VALUE, -6, 1) symbol FROM v$parameter
         WHERE NAME = 'user_dump_dest') b,
       (SELECT instance_name FROM v$instance) c,
       (SELECT spid FROM v$session s, v$process p, v$mystat m
         WHERE s.paddr = p.addr AND s.SID = m.SID AND m.statistic# = 0) d
/
```

Shared Pool 通过 Free Lists 管理 free 内存块（Chunk），Free 的内存块（Chunk）按不同 size 被划分到不同的部分（Bucket）进行管理。

结合 Dump 文件，可以通过图 6-7 对 Shared Pool 的 Free List 管理进行说明。

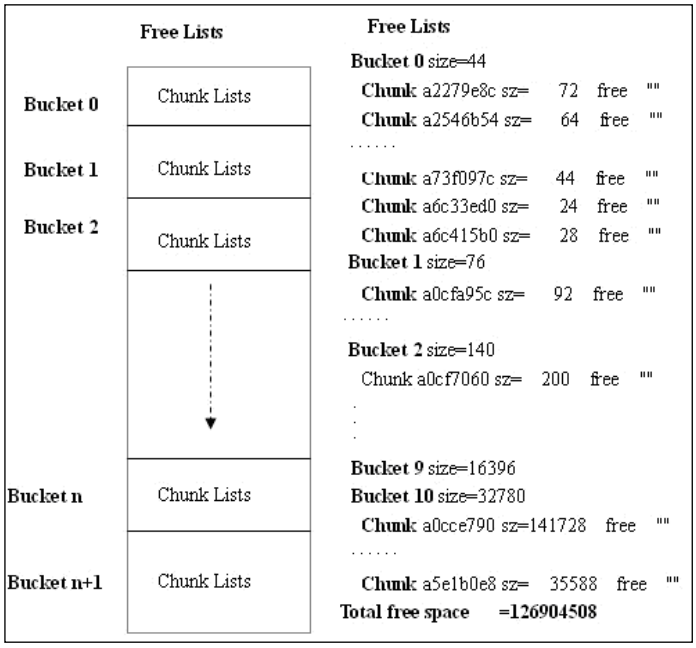


图 6-7　Shared Pool 自由列表

在 Oracle 8.1.5 中，不同 bucket 管理的内存块的 size 范围如下所示（size 显示的是下边界）：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
oracle:/usr/oracle8/admin/guess/udump>cat guess_ora_22038.trc|grep Bucket
```

```
Bucket 0 size=44
Bucket 1 size=76
Bucket 2 size=140
Bucket 3 size=268
Bucket 4 size=524
Bucket 5 size=1036
Bucket 6 size=2060
Bucket 7 size=4108
Bucket 8 size=8204
Bucket 9 size=16396
Bucket 10 size=32780
```

注意观察这个输出结果，在这里，小于 76 bytes 的块都位于 Bucket 0 上；大于 32780 的块，都在 Bucket 10 上。中间的 Bucket 边界值遵循：

```
Buckt Size(N) = 2 * Bucket Size(N-1)
76    — 44  = 32
140   — 76 = 64
268   — 140 = 128
524   — 268 = 256
1036  — 524 = 512
2060  — 1036 = 1024
4108  — 2060 = 2048
8204  — 4108 = 4196
16396 — 8204 = 8192
32780 — 16396 = 16384
```

初始地，数据库启动以后，Shared Pool 多数是连续内存块。但是当空间分配使用以后,内存块开始被分割，碎片开始出现，Bucket 列表开始变长。

Oracle 请求 Shared Pool 空间时，首先进入相应的 Bucket 进行查找。如果找不到，则转向下一个非空的 Bucket，获取第一个 Chunk。分割这个 Chunk，剩余部分会进入相应的 Bucket，进一步增加碎片。

最终的结果是，由于不停分割，每个 Bucket 上的内存块会越来越多，越来越碎小。通常 Bucket 0 的问题会最为显著，在我这个测试的小型数据库上，Bucket 0 上的碎片已经达到 9030 个，而 shared_pool_size 设置仅为 150MB。

通常如果每个 Bucket 上的 Chunk 多于 2000 个，就被认为是 Shared Pool 碎片过多。Shared Pool 的碎片过多，是 Shared Pool 产生性能问题的主要原因。

碎片过多会导致搜索 Free Lists 的时间过长，而我们知道，Free Lists 的管理和搜索都需要获得和持有一个非常重要的 Latch，就是 Shared Pool Latch。Latch 是 Oracle 数据库内部提供的一种低级锁，通过串行机制保护共享内存不被并发更新/修改所损坏。Latch 的持有通常都非常短暂（通常微秒级），但是对于一个繁忙的数据库，这个串行机制往往会成为极大的性能瓶颈。关于 Latch 的机制就不在这里过多介绍，那需要太多的篇幅。

继续前面的话题，如果 Free Lists 链表过长，搜索这个 Free Lists 的时间就会变长，从而可能导致 Shared Pool Latch 被长时间持有，在一个繁忙的系统中，这会引起严重的 Shared Pool Latch 的竞争。在 Oracle 9i 之前，这个重要的 Shared Pool Latch 只有一个，所以长时间持有将会导致严重的性能问题。

而在大多数情况下，我们请求的都是相对小的内存块（Chunk），这样搜索 Bucket 0 往往消耗了大量的时间以及资源，Latch 的争用此时就会成为一个非常严重的问题。

从 Oracle 8.1.7 开始，Oracle 对 Shared Pool 的管理进行了改进，在 Oracle 8.1.7.4 中：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
```

新的 Bucket 划分为：

```
bash-2.03$ grep  Bucket testora8_ora_13684.trc
 Bucket 0 size=32
 Bucket 1 size=40
 Bucket 2 size=48
 Bucket 3 size=56
 Bucket 4 size=64
 Bucket 5 size=72
……
 Bucket 198 size=1616
 Bucket 199 size=1624
 Bucket 200 size=1672
 Bucket 201 size=1720
……
 Bucket 248 size=3976
 Bucket 249 size=4024
 Bucket 250 size=4120
 Bucket 251 size=8216
 Bucket 252 size=16408
 Bucket 253 size=32792
 Bucket 254 size=65560
```

可以看到，初始 Oracle 分配了 255 个 Bucket，Bucket 分配：

- 0～199 以 8 bytes 递增；
- 200～248 以 48 bytes 递增；
- 249～250 递增 96；
- 250～251 递增 4096；
- 251～252 递增 8192；
- 252～253 递增 16384；

- 253～254 递增 32768。

通过进一步细分 Bucket，Oracle 可以强化对于共享池的管理。在 Oracle 9i 中，Oracle 进一步改写了 Shared Pool 管理的算法，下面来看一下 Oracle 9i 中的处理方式。

首先记录一下测试环境：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.3.0 - Production
```

转储 Shared Pool:

```
SQL> alter session set events 'immediate trace name heapdump level 2';
Session altered.
SQL> @gettrcname
TRACE_FILE_NAME
--------------------------------------------------------------------------------
/opt/oracle/admin/hsjf/udump/hsjf_ora_24983.trc
SQL> !
[oracle@jumper oracle]$ cd $admin
[oracle@jumper udump]$ cat hsjf_ora_24983.trc|grep Bucket
Bucket 0 size=16
Bucket 1 size=20
Bucket 2 size=24
Bucket 3 size=28
Bucket 4 size=32
Bucket 5 size=36
Bucket 6 size=40
Bucket 7 size=44
Bucket 8 size=48
Bucket 9 size=52
Bucket 10 size=56
Bucket 11 size=60
……<这里省略了部分内容>….
Bucket 248 size=3948
Bucket 249 size=4012
Bucket 250 size=4108
Bucket 251 size=8204
Bucket 252 size=16396
Bucket 253 size=32780
Bucket 254 size=65548
```

观察以上输出，可以看到在 Oracle 9i 中，Free Lists 被划分为 0～254，共 255 个 Bucket。

每个 Bucket 容纳的 size 范围进一步细分。

- Bucket 0～199 容纳 size 以 4 bytes 递增。
- Bucket 200～249 容纳 size 以 64 bytes 递增。

从 Bucket 249 开始,Oracle 各 Bucket 步长进一步增加。

- Bucket 249：4012～4107 = 96
- Bucket 250：4108～8203 = 4096
- Bucket 251：8204～16395 = 8192
- Bucket 252：16396～32779 = 16384
- Bucket 253：32780～65547 = 32768
- Bucket 254：>=65548

对比 Oracle 8i，在 Oracle9i 中 Shared Pool 管理最为显著的变化就是，对于数量众多的 chunk，Oracle 增加了更多的 Bucket 来管理（这和之前讲过的 Buffer Cache 的增强非常相似）。

0～199 共 200 个 Bucket，size 以 4 为步长递增；200～249 共 50 个 Bucket，size 以 64 递增。这样每个 Bucket 中容纳的 chunk 数量大大减少，查找的效率得以提高。

所以，在 Oracle 9i 之前，如果盲目地增大 shared_pool_size 或设置过大的 shared_pool_size，往往会适得其反。这就是也许你曾经听过的，过大 Shared_Pool 带来的管理上的负担。经过 Oracle 的不断努力，如果是在 Oracle 9i 中，设置较大的 Shared Pool 并不一定会给你带来和 Oracle 8i 同样的麻烦。在论坛上经常看到很多人对于 Shared_Pool 的建议一直就是 200～300MB，而且一直认为这就是 Shared Pool 性能问题的关键，实际上是不确切的。

6.2.3 Oracle 9i 子缓冲池的增强

从 Oracle 9i 开始，Shared Pool 可以被分割为多个子缓冲池（SubPool）进行管理，每个 SubPool 可以被看作是一个 Mini Shared Pool，拥有自己独立的 Free List、内存结构以及 LRU List。同时 Oracle 提供多个 Latch 对各个子缓冲池进行管理，从而避免单个 Latch 的竞争（Shared Pool Reserved Area 同样进行分割管理）。SubPool 最多可以有 7 个，Shared Pool Latch 也从原来的一个增加到现在的 7 个。如果系统有 4 个或 4 个以上的 CPU，并且 SHARED_POOL_SIZE 大于 250MB，Oracle 可以把 Shared Pool 分割为多个子缓冲池（SubPool）进行管理，在 Oracle 9i 中，每个 SubPool 至少为 128MB。

如果你看到过类似如下信息，那就意味着你可能遇到了 SubPool 的问题：

```
Tue Dec 11 17:14:49 2007
Errors in file /oracle/app/admin/ctais2/udump/ctais2_ora_778732.trc:
ORA-04031: unable to allocate 4216 bytes of shared memory
("shared pool","IDX_DJ_NSRXX_P_NSRMCCTAIS2","sga heap(2,0)","library cache")
ORA-04031: unable to allocate 4216 bytes of shared memory
("shared pool","IDX_DJ_NSRXX_P_NSRMCCTAIS2","sga heap(2,0)","library cache")
Tue Dec 11 17:14:51 2007
Errors in file /oracle/app/admin/ctais2/bdump/ctais2_pmon_393248.trc:
ORA-04031: unable to allocate 4216 bytes of shared memory
```

```
("shared pool","unknown object","sga heap(2,0)","library cache")
```

Oracle 9i 中多个子缓冲池的结构示意如图 6-8 所示。

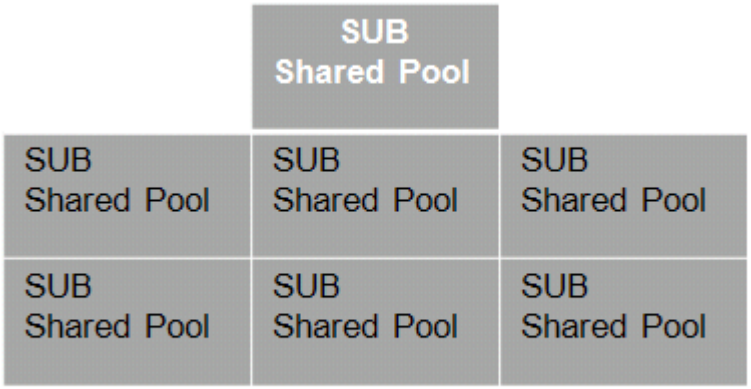


图 6-8　Oracle 9i 多个子缓冲池的结构

以下查询显示的是为管理 SubPool 而新增的子 Latch：

```
SQL> select addr, name, gets, misses, spin_gets
  2  from   v$latch_children where name = 'shared pool';
ADDR             NAME                  GETS     MISSES  SPIN_GETS
---------------- -------------------- ---------- ---------- ----------
0000000380068F38 shared pool                  0          0          0
0000000380068E40 shared pool                  0          0          0
0000000380068D48 shared pool                  0          0          0
0000000380068C50 shared pool                  0          0          0
0000000380068B58 shared pool                  0          0          0
0000000380068A60 shared pool                  0          0          0
0000000380068968 shared pool           13808572       3089       3087


7 rows selected.
```

子缓冲的数量受一个新引入的隐含参数**_KGHDSIDX_COUNT** 影响。可以手工调整该参数（仅限于试验环境研究用），观察共享池管理的变化：

```
SQL> alter system set "_kghdsidx_count"=2 scope=spfile;
System altered.
SQL> startup force;
ORACLE instance started.
…..
SQL> col KSPINM for a20
SQL> col KSPPSTVL for a20
SQL> select a.ksppinm, b.ksppstvl from   x$ksppi a, x$ksppsv b
```

```
  2  where  a.indx = b.indx and a.ksppinm = '_kghdsidx_count';
KSPPINM                                   KSPPSTVL
----------------------------------------- --------------------
_kghdsidx_count                              2
SQL> col name for a20
SQL> select addr, name, gets, misses, spin_gets
  2  from   v$latch_children where name = 'shared pool';
ADDR     NAME                      GETS     MISSES  SPIN_GETS
-------- -------------------- ---------- ---------- ----------
50043078 shared pool                  0          0          0
50042FB0 shared pool                  0          0          0
50042EE8 shared pool                  0          0          0
50042E20 shared pool                  0          0          0
50042D58 shared pool                  0          0          0
50042C90 shared pool               8166          0          0
50042BC8 shared pool                298          0          0
7 rows selected.
```

通过如下步骤转储缺省情况以及修改后的 Shared Pool，再进行观察：

```
alter session set events 'immediate trace name heapdump level 2';
alter system set "_kghdsidx_count"=2 scope=spfile;
startup force;
alter session set events 'immediate trace name heapdump level 2';
```

以下是概要输出，注意在前者的跟踪文件中，sga heap(1,0)指共享池只存在一个子缓冲，后者则存在 sga heap(1,0)以及 sga heap(2,0)两个子缓冲池：

```
[oracle@jumper udump]$ grep "sga heap" eygle_ora_25766.trc
HEAP DUMP heap name="sga heap"  desc=0x5000002c
HEAP DUMP heap name="sga heap(1,0)"  desc=0x5001ef0c
[oracle@jumper udump]$ grep "sga heap" eygle_ora_25786.trc
HEAP DUMP heap name="sga heap"  desc=0x5000002c
HEAP DUMP heap name="sga heap(1,0)"  desc=0x5001ef0c
HEAP DUMP heap name="sga heap(2,0)"  desc=0x50023c04
```

但是需要注意的是，虽然多缓冲池技术使 Oracle 可以管理更大的共享池，但是 SubPool 的划分可能也会导致各分区之间的协调问题，甚至可能因为内存分散而出现 ORA-04031 错误。最常见的问题是某个子缓冲池（SubPool）可能出现过度使用，当新的进程仍然被分配到这个 SubPool 时，可能会导致内存请求失败（而此时其他 SubPool 可能还有很多内存空间）。本节最初引用的信息就是来自一个实际的诊断案例，在具体诊断中可以通过如下 SQL 查询数据库各缓冲池的自由内存空间数量（摘要信息）：

```
SQL> select KSMCHIDX "SubPool", 'sga heap('||KSMCHIDX||',0)'sga_heap,ksmchcom ChunkComment,
  2  decode(round(ksmchsiz/1000),0,'0-1K', 1,'1-2K', 2,'2-3K',3,'3-4K',
```

```
  3  4,'4-5K',5,'5-6k',6,'6-7k',7,'7-8k',8, '8-9k', 9,'9-10k','> 10K') "size",
  4  count(*),ksmchcls Status, sum(ksmchsiz) Bytes
  5  from x$ksmsp where KSMCHCOM = 'free memory' group by ksmchidx, ksmchcls,
  6  'sga heap('||KSMCHIDX||',0)',ksmchcom, ksmchcls,decode(round(ksmchsiz/1000),0,'0-1K',
  7  1,'1-2K', 2,'2-3K', 3,'3-4K',4,'4-5K',5,'5-6k',6,
  8  '6-7k',7,'7-8k',8,'8-9k', 9,'9-10k','> 10K');
SubPool SGA_HEAP        CHUNKCOMMENT     size    COUNT(*) STATUS          BYTES
------- --------------- ---------------- ------- -------- -------- ------------
      1 sga heap(1,0)   free memory      0-1K        4842 free           909400
      1 sga heap(1,0)   free memory      > 10K         77 R-free       75113896
      2 sga heap(2,0)   free memory      0-1K        5948 free           729904
      2 sga heap(2,0)   free memory      1-2K          55 free            52056
      3 sga heap(3,0)   free memory      0-1K        5572 free           914096
      3 sga heap(3,0)   free memory      1-2K         204 free           153120
      3 sga heap(3,0)   free memory      8-9k           3 R-free          24456
      3 sga heap(3,0)   free memory      9-10k          2 R-free          17656
      3 sga heap(3,0)   free memory      > 10K         64 R-free       75655456
      4 sga heap(4,0)   free memory      4-5K        3683 free         14001176
      4 sga heap(4,0)   free memory      9-10k          1 free             8736
      4 sga heap(4,0)   free memory      5-6k           1 R-free           5424
      4 sga heap(4,0)   free memory      6-7k           1 R-free           5504
      5 sga heap(5,0)   free memory      1-2K         771 free           847032
      5 sga heap(5,0)   free memory      > 10K         73 R-free       79633288
      6 sga heap(6,0)   free memory      4-5K           1 R-free           3912
      6 sga heap(6,0)   free memory      6-7k           3 R-free          18840
      6 sga heap(6,0)   free memory      7-8k           1 R-free           7080
      6 sga heap(6,0)   free memory      > 10K         94 R-free       73405856
```

因为子缓冲池存在的种种问题，从 Oracle 10g 开始，Oracle 允许内存请求在不同 SubPool 之间进行切换（Switch），从而提高了请求成功的可能（但是显然切换不可能是无限制的，所以问题仍然可能存在）。

6.2.3　Oracle 9i 子缓冲池的案例

在客户的一个 Oracle9i 系统中，经常出现 ORA-04031 的错误，客户系统的主要配置如下：

```
SQL> select * from v$version;

BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.6.0 - 64bit Production
```

```
PL/SQL Release 9.2.0.6.0 - Production
CORE    9.2.0.6.0       Production
TNS for Solaris: Version 9.2.0.6.0 - Production
NLSRTL Version 9.2.0.6.0 - Production

SQL> show parameter cpu_count

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
cpu_count                            integer     48
SQL> select * from v$sga;

NAME                                     VALUE
---------------------------- ----------------
Fixed Size                           762240
Variable Size                       2600468480
Database Buffers                   18975031296
Redo Buffers                          6578176
```

我们检查其参数设置，缺省的子池设置是 7 个：

```
SQL> col KSPINM for a20
SQL> col KSPPSTVL for a20
SQL> select a.ksppinm, b.ksppstvl from   x$ksppi a, x$ksppsv b
  2  where  a.indx = b.indx and a.ksppinm = '_kghdsidx_count';

KSPPINM                                                          KSPPSTVL
---------------------------------------------------------------- --------------------
_kghdsidx_count                                                  7
```

7 个子池都被使用，其 Latch 使用情况如下：

```
SQL> select child#, gets
  2  from v$latch_children
  3  where name = 'shared pool' order by child#;

    CHILD#       GETS
---------- ----------
         1  333403016
         2  355720323
         3  273944301
         4  197980497
         5  282347697
```

```
         6  354398593
         7  468809111
```

那我们来看一下具体的子池使用及内存情况：

```
SQL> @/tmp/eygle/sgasatx.sql total

-- All allocations:

SUBPOOL                             BYTES         MB
------------------------------ ---------- ----------
shared pool (1):                 352321536        336
shared pool (2):                 335544320        320
shared pool (3):                 335544320        320
shared pool (4):                 335544320        320
shared pool (5):                 335544320        320
shared pool (6):                 335544320        320
shared pool (7):                 335544320        320
shared pool (Total):            2365587456       2256

8 rows selected.
```

进一步的可以来查询一下各个子池的剩余内存：

```
SQL> @/tmp/eygle/sgasatx.sql "free memory"

-- All allocations:

SUBPOOL                             BYTES         MB
------------------------------ ---------- ----------
shared pool (1):                 352321536        336
shared pool (2):                 335544320        320
shared pool (3):                 335544320        320
shared pool (4):                 335544320        320
shared pool (5):                 335544320        320
shared pool (6):                 335544320        320
shared pool (7):                 335544320        320
shared pool (Total):            2365587456       2256

8 rows selected.

SUBPOOL                         NAME                           SUM(BYTES)         MB
```

```
------------------------------ ------------------------------ ---------- ----------
shared pool (1):               free memory                       8158640       7.78
shared pool (2):               free memory                       7414472       7.07
shared pool (3):               free memory                       7831608       7.47
shared pool (4):               free memory                      10690992       10.2
shared pool (5):               free memory                      17201856       16.4
shared pool (6):               free memory                       8239920       7.86
shared pool (7):               free memory                      13925416      13.28

7 rows selected.
```

我们注意到，每个子池的剩余内存都很有限，更具体的，可以查询剩余内存块大小：

```
SQL> SELECT   ksmchidx "SubPool", 'sga heap(' || ksmchidx || ',0)' sga_heap,
  2           ksmchcom chunkcomment,
  3           DECODE (ROUND (ksmchsiz / 1000),
  4                   0, '0-1K',
  5                   1, '1-2K',
  6                   2, '2-3K',
  7                   3, '3-4K',
  8                   4, '4-5K',
  9                   5, '5-6k',
 10                   6, '6-7k',
 11                   7, '7-8k',
 12                   8, '8-9k',
 13                   9, '9-10k',
 14                   '> 10K'
 15                  ) "size",
 16           COUNT (*), ksmchcls status, SUM (ksmchsiz) BYTES
 17      FROM x$ksmsp
 18     WHERE ksmchcom = 'free memory'
 19  GROUP BY ksmchidx,
 20           ksmchcls,
 21           'sga heap(' || ksmchidx || ',0)',
 22           ksmchcom,
 23           ksmchcls,
 24           DECODE (ROUND (ksmchsiz / 1000),
 25                   0, '0-1K',
 26                   1, '1-2K',
 27                   2, '2-3K',
 28                   3, '3-4K',
```

```
29                   4, '4-5K',
30                   5, '5-6k',
31                   6, '6-7k',
32                   7, '7-8k',
33                   8, '8-9k',
34                   9, '9-10k',
35                   '> 10K'
36                  );
```

SUBPOOL	SGA_HEAP	CHUNKCOMMENT	size	COUNT(*)	STATUS	BYTES
1	sga heap(1,0)	free memory	0-1K	5173	free	922568
1	sga heap(1,0)	free memory	1-2K	5422	free	5274920
1	sga heap(1,0)	free memory	2-3K	759	free	1512944
1	sga heap(1,0)	free memory	3-4K	367	free	1177400
1	sga heap(1,0)	free memory	4-5K	412	free	1714432
1	sga heap(1,0)	free memory	5-6k	78	free	384232
1	sga heap(1,0)	free memory	6-7k	14	free	81384
1	sga heap(1,0)	free memory	7-8k	2	free	14048
1	sga heap(1,0)	free memory	8-9k	8	free	65960
1	sga heap(1,0)	free memory	9-10k	2	free	17464
1	sga heap(1,0)	free memory	0-1K	17	R-free	4336
1	sga heap(1,0)	free memory	1-2K	20	R-free	19904
1	sga heap(1,0)	free memory	2-3K	16	R-free	31704
1	sga heap(1,0)	free memory	3-4K	36	R-free	112176
1	sga heap(1,0)	free memory	4-5K	25	R-free	100392
1	sga heap(1,0)	free memory	5-6k	29	R-free	142176
1	sga heap(1,0)	free memory	6-7k	2	R-free	11968
1	sga heap(1,0)	free memory	7-8k	9	R-free	62096
1	sga heap(1,0)	free memory	8-9k	12	R-free	95480
1	sga heap(1,0)	free memory	9-10k	11	R-free	99192
1	sga heap(1,0)	free memory	> 10K	25	R-free	434272
2	sga heap(2,0)	free memory	0-1K	4919	free	848864
2	sga heap(2,0)	free memory	1-2K	1883	free	1605952
2	sga heap(2,0)	free memory	2-3K	1399	free	2812184
2	sga heap(2,0)	free memory	3-4K	210	free	637920
2	sga heap(2,0)	free memory	4-5K	496	free	2056256
2	sga heap(2,0)	free memory	5-6k	48	free	231232
2	sga heap(2,0)	free memory	6-7k	15	free	88392

```
  2 sga heap(2,0)       free memory      7-8k          10 free          68064
  2 sga heap(2,0)       free memory      8-9k           6 free          49584
  2 sga heap(2,0)       free memory      9-10k          2 free          17744
  2 sga heap(2,0)       free memory      0-1K          12 R-free         3088
  2 sga heap(2,0)       free memory      1-2K          26 R-free        21528
  2 sga heap(2,0)       free memory      2-3K          16 R-free        31800
  2 sga heap(2,0)       free memory      3-4K          28 R-free        85232
  2 sga heap(2,0)       free memory      4-5K          24 R-free        97152
  2 sga heap(2,0)       free memory      5-6k          28 R-free       137592
  2 sga heap(2,0)       free memory      6-7k          11 R-free        66744
  2 sga heap(2,0)       free memory      7-8k           6 R-free        41496
  2 sga heap(2,0)       free memory      8-9k           8 R-free        65200
  2 sga heap(2,0)       free memory      9-10k          5 R-free        46056
  2 sga heap(2,0)       free memory      > 10K         43 R-free       769144
  3 sga heap(3,0)       free memory      0-1K        6921 free        1058264
  3 sga heap(3,0)       free memory      1-2K          12 free          11920
  3 sga heap(3,0)       free memory      2-3K          25 free          52608
  3 sga heap(3,0)       free memory      3-4K         472 free        1408720
  3 sga heap(3,0)       free memory      4-5K         161 free         610504
  3 sga heap(3,0)       free memory      0-1K          12 R-free         2832
  3 sga heap(3,0)       free memory      1-2K          15 R-free        14976
  3 sga heap(3,0)       free memory      2-3K          13 R-free        27192
  3 sga heap(3,0)       free memory      3-4K          20 R-free        62696
  3 sga heap(3,0)       free memory      4-5K          19 R-free        78136
  3 sga heap(3,0)       free memory      5-6k          21 R-free       101688
  3 sga heap(3,0)       free memory      6-7k           8 R-free        47472
  3 sga heap(3,0)       free memory      7-8k           3 R-free        21648
  3 sga heap(3,0)       free memory      8-9k           6 R-free        49016
  3 sga heap(3,0)       free memory      9-10k          9 R-free        81344
  3 sga heap(3,0)       free memory      > 10K         64 R-free      1212424
  4 sga heap(4,0)       free memory      0-1K        6430 free         928688
  4 sga heap(4,0)       free memory      1-2K           1 free           1472
  4 sga heap(4,0)       free memory      2-3K           3 free           6696
  4 sga heap(4,0)       free memory      3-4K           2 free           6112
  4 sga heap(4,0)       free memory      4-5K         169 free         625528
  4 sga heap(4,0)       free memory      0-1K          14 R-free         3056
  4 sga heap(4,0)       free memory      1-2K          34 R-free        30880
  4 sga heap(4,0)       free memory      2-3K          20 R-free        39704
  4 sga heap(4,0)       free memory      3-4K          24 R-free        75184
```

```
      4 sga heap(4,0)       free memory      4-5K           35 R-free       138808
      4 sga heap(4,0)       free memory      5-6k           23 R-free       109912
      4 sga heap(4,0)       free memory      6-7k            7 R-free        42496
      4 sga heap(4,0)       free memory      7-8k            2 R-free        14152
      4 sga heap(4,0)       free memory      8-9k            4 R-free        31464
      4 sga heap(4,0)       free memory      9-10k           9 R-free        80464
      4 sga heap(4,0)       free memory      > 10K          34 R-free       689640
      5 sga heap(5,0)       free memory      0-1K         4416 free         779096
      5 sga heap(5,0)       free memory      1-2K            1 free          1352
      5 sga heap(5,0)       free memory      2-3K          138 free         243720
      5 sga heap(5,0)       free memory      3-4K            1 free          2824
      5 sga heap(5,0)       free memory      0-1K           11 R-free         2104
      5 sga heap(5,0)       free memory      1-2K           17 R-free        16584
      5 sga heap(5,0)       free memory      2-3K           19 R-free        36888
      5 sga heap(5,0)       free memory      3-4K           16 R-free        48400
      5 sga heap(5,0)       free memory      4-5K           33 R-free       134480
      5 sga heap(5,0)       free memory      5-6k           17 R-free        83688
      5 sga heap(5,0)       free memory      6-7k           10 R-free        59544
      5 sga heap(5,0)       free memory      8-9k            3 R-free        23840
      5 sga heap(5,0)       free memory      9-10k           4 R-free        36344
      5 sga heap(5,0)       free memory      > 10K          40 R-free      1669384
      6 sga heap(6,0)       free memory      0-1K         6203 free         863104
      6 sga heap(6,0)       free memory      1-2K           59 free          66608
      6 sga heap(6,0)       free memory      2-3K          231 free         469040
      6 sga heap(6,0)       free memory      3-4K            4 free          12632
      6 sga heap(6,0)       free memory      4-5K            7 free          29352
      6 sga heap(6,0)       free memory      0-1K           17 R-free         4248
      6 sga heap(6,0)       free memory      1-2K           22 R-free        22112
      6 sga heap(6,0)       free memory      2-3K            9 R-free        16576

SUBPOOL SGA_HEAP            CHUNKCOMMENT     size    COUNT(*) STATUS        BYTES
------- ------------------- ---------------- ----- ---------- -------- ----------
      6 sga heap(6,0)       free memory      3-4K           15 R-free        46144
      6 sga heap(6,0)       free memory      4-5K           30 R-free       121880
      6 sga heap(6,0)       free memory      5-6k           15 R-free        70808
      6 sga heap(6,0)       free memory      6-7k            4 R-free        24304
      6 sga heap(6,0)       free memory      7-8k            6 R-free        41144
      6 sga heap(6,0)       free memory      8-9k            7 R-free        56328
      6 sga heap(6,0)       free memory      9-10k          11 R-free        99464
```

```
      6 sga heap(6,0)       free memory      > 10K         56 R-free      1758912
      7 sga heap(7,0)       free memory      0-1K        3814 free         607616
      7 sga heap(7,0)       free memory      1-2K         205 free         164592
      7 sga heap(7,0)       free memory      2-3K          30 free          61008
      7 sga heap(7,0)       free memory      3-4K           4 free          11456
      7 sga heap(7,0)       free memory      4-5K           5 free          19448
      7 sga heap(7,0)       free memory      0-1K           6 R-free         1984
      7 sga heap(7,0)       free memory      1-2K          22 R-free        22072
      7 sga heap(7,0)       free memory      2-3K          10 R-free        20744
      7 sga heap(7,0)       free memory      3-4K           9 R-free        28312
      7 sga heap(7,0)       free memory      4-5K          10 R-free        40728
      7 sga heap(7,0)       free memory      5-6k          21 R-free       102824
      7 sga heap(7,0)       free memory      6-7k           5 R-free        29288
      7 sga heap(7,0)       free memory      7-8k           3 R-free        20104
      7 sga heap(7,0)       free memory      8-9k           5 R-free        40344
      7 sga heap(7,0)       free memory      9-10k          6 R-free        54432
      7 sga heap(7,0)       free memory      > 10K         52 R-free      2816480


120 rows selected.
```

仔细观察以上输出，可以注意到，大块内存

6.2.4 Oracle 10g 共享池管理的增强

子缓冲池的分配的算法很简单：

- 每个子缓冲池必须满足一定的内存约束；
- 每 4 颗 CPU 可以分配一个子缓冲池，最多 7 个。

在 Oracle 9i 中，每个 SubPool 至少 128MB，在 Oracle 10g 中，每个子缓冲池至少为 256MB。如前所述，SubPool 的数量可以通过_kghdsidx_count 参数来控制，但是没有参数可以显示地控制 SubPool 的大小。

根据以上规则，在一个 12 颗 CPU 的系统中，如果分配 300MB Shared Pool，Oracle 9i 将创建 2 个 SubPool，每个大约 150MB，如果共享池增加到 500MB，Oracle 将创建 3 个 SubPool，每个大约 166MB。

不管 Oracle 9i 中的 128MB 以及 Oracle10g 中的 256MB，这样的 SubPool 在许多复杂的系统中，都可能是过小的，在这些情况下，可能需要增加 SubPool 的大小。可以通过控制 Shared Pool 大小以及 SubPool 的数量来改变 SubPool 的大小。一些 Bug 以及内部测试表明 500MB 的 SubPool 可能会带来更好的性能，所以从 Oracle 11g 开始，每个 SubPool 至少为 512MB。

除大小控制之外，在 Oracle 10g 中，Oracle 仍然对共享池的管理做出了进一步改进，那就是对单个子缓冲池进行进一步的细分。现在缺省地，Oracle 10g 会将单个缓冲池分割为会 4 个子分区进行管理（这可能是因为通常 4 颗 CPU 才分配一个 SubPool），使用类似如上的方法在

Oracle 10gR2 中进行测试：

```
alter session set events 'immediate trace name heapdump level 2';
alter system set "_kghdsidx_count"=2 scope=spfile;
startup force;
alter session set events 'immediate trace name heapdump level 2';
```

分析得到的日志，当仅有一个子缓冲时，Shared Pool 被划分为 sga heap(1,0)～sga heap(1,3)共 4 个子分区：

```
[oracle@eygle udump]$ grep "sga heap" eygle_ora_13577.trc
HEAP DUMP heap name="sga heap"  desc=0x2000002c
HEAP DUMP heap name="sga heap(1,0)"  desc=0x2001b550
HEAP DUMP heap name="sga heap(1,1)"  desc=0x2001c188
HEAP DUMP heap name="sga heap(1,2)"  desc=0x2001cdc0
HEAP DUMP heap name="sga heap(1,3)"  desc=0x2001d9f8
```

当使用两个子缓冲时，Shared Pool 则被划分为 8 个子分区进行管理：

```
[oracle@eygle udump]$ grep "sga heap" eygle_ora_13618.trc
HEAP DUMP heap name="sga heap"  desc=0x2000002c
HEAP DUMP heap name="sga heap(1,0)"  desc=0x2001b550
HEAP DUMP heap name="sga heap(1,1)"  desc=0x2001c188
HEAP DUMP heap name="sga heap(1,2)"  desc=0x2001cdc0
HEAP DUMP heap name="sga heap(1,3)"  desc=0x2001d9f8
HEAP DUMP heap name="sga heap(2,0)"  desc=0x20020640
HEAP DUMP heap name="sga heap(2,1)"  desc=0x20021278
HEAP DUMP heap name="sga heap(2,2)"  desc=0x20021eb0
HEAP DUMP heap name="sga heap(2,3)"  desc=0x20022ae8
```

Oracle 10g 中多缓冲池结构示意图如图 6-9 所示。

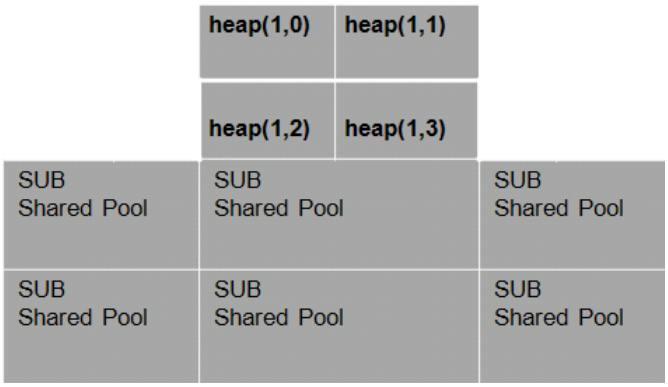


图 6-9　Oracle 10g 的多缓冲池结构

通过一个内部表 X$KGHLU（[K]ernel [G]eneric memory [H]eap manager State of [L]R[U] Of Unpinned Recreatable chunks）可以查询这些子缓冲池的分配：

```
SQL> select addr,indx,kghluidx,kghludur,kghluops,kghlurcr from x$kghlu;
```

```
ADDR     INDX   KGHLUIDX   KGHLUDUR   KGHLUOPS   KGHLURCR
-------- ---- ---------- ---------- ---------- ----------
B5F4C5B4    0          2          3      12773        257
B5F4C1AC    1          2          2      43675       1042
B5F4D9C8    2          2          1      18831       1518
B5F4D5C0    3          2          0          0          0
B5F4D1B8    4          1          3     144697        327
B5F4E9E4    5          1          2     483428       1462
B5F4E5DC    6          1          1       6558        982
B5F4E1D4    7          1          0          0          0
8 rows selected.
```

通过这一系列的算法改进，Oracle 中 Shared Pool 管理得以不断增强，较好的解决了大 Shared Pool 的性能问题；Oracle 8i 中，过大 Shared Pool 设置可能带来的栓锁争用等性能问题在某种程度上得以解决。从 Oracle 10g 开始，Oracle 开始提供自动共享内存管理，使用该特性，用户可以不必显示设置共享内存参数，Oracle 会自动进行分配和调整，虽然 Oracle 给我们提供了极大的便利，但是了解自动化后面的原理对于理解 Oracle 的运行机制仍然是十分重要的。

6.2.5 了解 X$KSMSP 视图

Shared Pool 的空间分配和使用情况，可以通过一个内部视图来观察，这个视图就是 X$KSMSP。

X$KSMSP 的名称含义为: **[K]**ernal **[S]**torage **[M]**emory Management **[S]**GA Hea**[P]**

其中每一行都代表着 Shared Pool 中的一个 Chunk。以下是 x$ksmsp 的结构:

```
SQL> desc x$ksmsp
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ADDR                                               RAW(4)
 INDX                                               NUMBER
 INST_ID                                            NUMBER
 KSMCHIDX                                           NUMBER
 KSMCHDUR                                           NUMBER
 KSMCHCOM                                           VARCHAR2(16)
 KSMCHPTR                                           RAW(4)
 KSMCHSIZ                                           NUMBER
 KSMCHCLS                                           VARCHAR2(8)
 KSMCHTYP                                           NUMBER
 KSMCHPAR                                           RAW(4)
```

这里需要关注一下以下几个字段。

（1）x$ksmsp.ksmchcom 是注释字段，每个内存块被分配以后，注释会添加在该字段中。

（2）x$ksmsp.ksmchsiz 代表块大小。

（3）x$ksmsp.ksmchcls 列代表类型，主要有 4 类，具体说明如下。

- **free**：即 Free Chunks，不包含任何对象的 Chunk，可以不受限制的被自由分配。
- **recr**：即 Recreatable Chunks，包含可以被临时移出内存的对象，在需要的时候，这个对象可以被重新创建。例如，许多存储共享 SQL 代码的内存都是可以重建的。
- **freeable**：即 Freeable Chunks，包含 session 周期或调用的对象，随后可以被释放。这部分内存有时候可以全部或部分提前释放。但是注意，由于某些对象是中间过程产生的，这些对象不能临时被移出内存（因为不可重建）。
- **perm**：即 Permanent Memory Chunks，包含永久对象，通常不能独立释放。

从以上引用的 trace 文件中，摘出开头一段，可以清楚地看到 Oracle 对这部分 Chunk 的记录情况：

```
HEAP DUMP heap name="sga heap"  desc=0x80000030
 extent sz=0xfc4 alt=48 het=32767 rec=9 flg=2 opc=0
 parent=0 owner=0 nex=0 xsz=0x1
EXTENT 0
  Chunk a7412000 sz= 23801020    perm      "perm           "  alo=23801020
  Chunk a8ac4cbc sz=       68    free      "               "
  Chunk a8ac4d00 sz=      560    freeable  "library cache  "  ds=a3e4fd24
  Chunk a8ac4f30 sz=      588    freeable  "sql area       "  ds=a6bcc328
  Chunk a8ac517c sz=      448    freeable  "library cache  "  ds=a57b6a38
  Chunk a8ac533c sz=     1072    freeable  "partitioning d "  ds=a4002688
  Chunk a8ac576c sz=     2036    freeable  "library cache  "  ds=a6c29e3c
  Chunk a8ac5f60 sz=      560    freeable  "library cache  "  ds=a2decda8
  Chunk a8ac6190 sz=       96    freeable  "library cache  "
  Chunk a8ac61f0 sz=       20    free      "               "
  Chunk a8ac6204 sz=      176    recreate  "KGL handles    "  latch=0
  Chunk a8ac62b4 sz=      560    recreate  "library cache  "  latch=8000cc38
     ds a6c33f54 sz=     1680 ct=        3
        a400a0dc sz=      560
        a24aee88 sz=      560
```

可以通过查询 x$ksmsp 视图来考察 Shared Pool 中存在的内存片的数量，不过注意，Oracle 的某些版本（如 10.1.0.2）在某些平台上（如 HP-UX PA-RISC 64-bit）查询该视图可能导致过度的 CPU 耗用，这是由于 Bug 引起的。

看一下测试，在这个测试数据库中，初始启动数据库，在 x$ksmsp 视图中存在 2259 个 Chunk：

```
SQL> select count(*) from x$ksmsp;
  COUNT(*)
----------
      2259
```

执行查询:

```
SQL> select count(*) from dba_objects;
  COUNT(*)
----------
     10491
此时 shared pool 中的 chunk 数量增加
SQL> select count(*) from x$ksmsp;
  COUNT(*)
----------
      2358
```

这就是由于 Shared Pool 中进行 SQL 解析，请求空间，进而导致请求 free 空间分配、分割，从而产生了更多、更细碎的内存 Chunk。

由此可以看出，如果数据库系统中存在大量的硬解析，不停请求分配 free 的 Shared Pool 内存，除了必需的 Shared Pool Latch 等竞争外，还不可避免地会导致 Shared Pool 中产生更多的内存碎片（当然，在内存回收时，你可能看到 Chunk 数量减少的情况）。

继续进行一点深入研究，首先重新启动数据库：

```
SQL> startup force;
```

创建一张临时表用以保存之前 x$ksmsp 的状态：

```
SQL> CREATE GLOBAL TEMPORARY TABLE e$ksmsp ON COMMIT PRESERVE ROWS AS
  2  SELECT      a.ksmchcom,
  3           SUM (a.CHUNK) CHUNK,
  4           SUM (a.recr) recr,
  5           SUM (a.freeabl) freeabl,
  6           SUM (a.SUM) SUM
  7      FROM (SELECT   ksmchcom, COUNT (ksmchcom) CHUNK,
  8                     DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
  9                     DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
 10                     SUM (ksmchsiz) SUM
 11                FROM x$ksmsp GROUP BY ksmchcom, ksmchcls) a
 12  where 1 = 0
 13  GROUP BY a.ksmchcom;
Table created.
```

保存当前 Shared Pool 状态：

```
SQL> INSERT INTO E$KSMSP
  2  SELECT      a.ksmchcom,
  3           SUM (a.CHUNK) CHUNK,
  4           SUM (a.recr) recr,
  5           SUM (a.freeabl) freeabl,
  6           SUM (a.SUM) SUM
  7      FROM (SELECT   ksmchcom, COUNT (ksmchcom) CHUNK,
```

```
  8                     DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
  9                     DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
 10                     SUM (ksmchsiz) SUM
 11                FROM x$ksmsp
 12            GROUP BY ksmchcom, ksmchcls) a
 13  GROUP BY a.ksmchcom
 14  /
41 rows created.
```

执行查询：

```
SQL> select count(*) from dba_objects;
  COUNT(*)
----------
     10492
```

比较前后 Shared Pool 内存分配的变化：

```
SQL> select a.ksmchcom,a.chunk,a.sum,b.chunk,b.sum,(a.chunk - b.chunk) c_diff,(a.sum -b.sum)
s_diff
  2  from
  3  (SELECT   a.ksmchcom,
  4           SUM (a.CHUNK) CHUNK,
  5           SUM (a.recr) recr,
  6           SUM (a.freeabl) freeabl,
  7           SUM (a.SUM) SUM
  8      FROM (SELECT   ksmchcom, COUNT (ksmchcom) CHUNK,
  9                     DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
 10                     DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
 11                     SUM (ksmchsiz) SUM
 12                FROM x$ksmsp
 13            GROUP BY ksmchcom, ksmchcls) a
 14  GROUP BY a.ksmchcom) a,e$ksmsp b
 15  where a.ksmchcom = b.ksmchcom and (a.chunk - b.chunk) <>0
 16  /
KSMCHCOM              CHUNK        SUM      CHUNK        SUM     C_DIFF     S_DIFF
---------------- ---------- ---------- ---------- ---------- ---------- ----------
KGL handles              313     102080        302      98416         11       3664
KGLS heap                274     365752        270     360424          4       5328
KQR PO                   389     198548        377     192580         12       5968
free memory               93    2292076         90    2381304          3     -89228
library cache           1005     398284        965     381416         40      16868
sql area                 287     547452        269     490052         18      57400
```

```
6 rows selected.
```

简单分析一下以上结果：首先 free memory 的大小减少了 89228（增加到另外 5 个组件中），这说明 SQL 解析存储占用了一定的内存空间；而 Chunk 从 90 增加为 93，这说明内存碎片增加了，碎片增加是共享池性能下降的开始。

6.2.6 Shared Pool 的转储与分析

使用如下命令可以对共享池 Library Cache 信息进行转储分析：

```
ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level LL';
```

其中 LL 代表 Level 级别,对于 9.2.0 及以后版本,不同 Level 含义如下：

- Level =1，转储 Library Cache 统计信息；
- Level =2，转储 Hash Table 概要；
- Level =4，转储 Library Cache 对象，只包含基本信息；
- Level =8，转储 Library Cache 对象，包含详细信息（如 child references、pin waiters 等）；
- Level =16，增加 heap sizes 信息；
- Level =32，增加 heap 信息。

Library Cache 由一个 Hash 表组成，而 Hash 表是一个由 Hash Buckets 组成的数组，每个 hash Bucket 都是包含 Library Cache Handle 的一个双向链表。Library Cache Handle 指向 Library Cache Object 和一个引用列表。Library Cache 对象进一步分为依赖表、子表和授权表等。

首先通过以下命令对 Library Cache 进行转储：

```
ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 4'
```

获得以下输出（以下输出来自 Oracle8i），第一部分(等价于 Level 1)信息:

```
LIBRARY CACHE STATISTICS:
      gets hit ratio       pins hit ratio    reloads   invalids namespace
---------- --------- ---------- --------- ---------- ---------- ---------
 619658171 0.9999160 2193292112 0.9999511       9404        380 CRSR
  79698558 0.9998832  424614847 0.9999108      13589          0 TABL/PRCD/TYPE
    163399 0.9979926     163402 0.9978948         16          0 BODY/TYBD
         0 0.0000000          0 0.0000000          0          0 TRGR
        34 0.0294118         35 0.0571429          0          0 INDX
     18948 0.9968862      24488 0.9953855          0          0 CLST
         0 0.0000000          0 0.0000000          0          0 OBJE
    115071 0.9992179     115071 0.9930999        704          0 EVNT
……
 699654181 0.9999117 2618209955 0.9999440      23713        380 CUMULATIVE
```

这部分信息也就是 v$librarycache 中显示的相关内容。

第 2 部分信息如下所示（等价于 Level 2 中的输出）：

```
LIBRARY CACHE HASH TABLE: size=509 count=354
```

```
BUCKET    0:
BUCKET    1:
BUCKET    2: *
BUCKET    3:
BUCKET    4:
。。。。。。。。。。。。。。。。。。。。
BUCKET  506: *
BUCKET  507:
BUCKET  508:
BUCKET  509:
BUCKET  510:
BUCKET  511:
```

在 Oracle 8i 中，Oracle 以一个很长的 Library Cache Hash Table 来记录 Library Cache 的使用情况，“*”代表该 Bucket 中包含的对象的个数，在以上输出中看到 Bucket 198 中包含 4 个对象。在第 3 部分中可以找到 Bucket 198，每个 NAME 部分包含了一个对象，可以是 SQL 或者 PACKAGE 等。

```
BUCKET 198:
  LIBRARY OBJECT HANDLE: handle=2c2b4ac4
  name=SELECT a.statement_id, a.timestamp, a.remarks, a.operation, a.options,
       a.object_node, a.object_owner, a.object_name, a.object_instance,
       a.object_type, a.optimizer, a.search_columns, a.id, a.parent_id,
       a.position, a.cost, a.cardinality, a.bytes, a.other_tag,
       a.partition_start, a.partition_stop, a.partition_id, a.other,
       a.distribution, ROWID FROM plan_table a
  hash=60dd47a1 timestamp=08-27-2004 10:19:28
  namespace=CRSR flags=RON/TIM/PN0/LRG/[10010001]
  kkkk-dddd-llll=0000-0001-0001 lock=0 pin=0 latch=0
  lwt=2c2b4adc[2c2b4adc,2c2b4adc] ltm=2c2b4ae4[2c2b4ae4,2c2b4ae4]
  pwt=2c2b4af4[2c2b4af4,2c2b4af4] ptm=2c2b4b4c[2c2b4b4c,2c2b4b4c]
  ref=2c2b4acc[2c2b4acc,2c2b4acc]
    LIBRARY OBJECT: object=2c0b1430
    type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
    CHILDREN: size=16
    child#    table reference   handle
    ------ -------- --------- --------
         0 2c0b15ec  2c0b15b4 2c2c0d50
    DATA BLOCKS:
    data#     heap  pointer status pins change
    ----- -------- -------- ------ ---- ------
```

```
      0 2c362290 2c0b14b4 I/-/A     0 NONE
LIBRARY OBJECT HANDLE: handle=2c3675d4
name=SYS.DBMS_STANDARD
hash=50748ddb timestamp=NULL
namespace=BODY/TYBD flags=TIM/SML/[02000000]
kkkk-dddd-llll=0000-0011-0011 lock=0 pin=0 latch=0
lwt=2c3675ec[2c3675ec,2c3675ec] ltm=2c3675f4[2c3675f4,2c3675f4]
pwt=2c367604[2c367604,2c367604] ptm=2c36765c[2c36765c,2c36765c]
ref=2c3675dc[2c3675dc,2c3675dc]
  LIBRARY OBJECT: object=2c1528e8
  flags=NEX[0002] pflags= [00] status=VALD load=0
  DATA BLOCKS:
  data#     heap  pointer status pins change
  ----- -------- -------- ------ ---- ------
      0 2c367564 2c1529cc I/-/A     0 NONE
      4 2c15297c        0 -/P/-     0 NONE
LIBRARY OBJECT HANDLE: handle=2c347dd8
name=select pos#,intcol#,col#,spare1 from icol$ where obj#=:1
hash=fa15ebe3 timestamp=07-28-2004 18:04:43
namespace=CRSR flags=RON/TIM/PN0/SML/[12010000]
kkkk-dddd-llll=0000-0001-0001 lock=0 pin=0 latch=0
lwt=2c347df0[2c347df0,2c347df0] ltm=2c347df8[2c347df8,2c347df8]
pwt=2c347e08[2c347e08,2c347e08] ptm=2c347e60[2c347e60,2c347e60]
ref=2c347de0[2c347de0,2c347de0]
  LIBRARY OBJECT: object=2c1cd1a0
  type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
  CHILDREN: size=16
  child#    table reference   handle
  ------ -------- --------- --------
       0 2c1cd35c  2c1cd324 2c281678
       1 2c1cd35c  2c352c50 2c0eeb8c
       2 2c1cd35c  2c352c6c 2c2bb05c
  DATA BLOCKS:
  data#     heap  pointer status pins change
  ----- -------- -------- ------ ---- ------
      0 2c2e8c58 2c1cd224 I/-/A     0 NONE
LIBRARY OBJECT HANDLE: handle=2c3a6484
name=SYS.TS$
hash=bb42852e timestamp=04-24-2002 00:04:15
```

```
  namespace=TABL/PRCD/TYPE flags=PKP/TIM/KEP/SML/[02900000]
  kkkk-dddd-llll=0111-0111-0119 lock=0 pin=0 latch=0
  lwt=2c3a649c[2c3a649c,2c3a649c] ltm=2c3a64a4[2c3a64a4,2c3a64a4]
  pwt=2c3a64b4[2c3a64b4,2c3a64b4] ptm=2c3a650c[2c3a650c,2c3a650c]
  ref=2c3a648c[2c0d4b14,2c09353c]
    LIBRARY OBJECT: object=2c3a626c
    type=TABL flags=EXS/LOC[0005] pflags= [00] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer status pins change
    ----- -------- -------- ------ ---- ------
        0 2c3a8ea4 2c3a63b0 I/P/A     0 NONE
        3 2c3a5828        0 -/P/-     0 NONE
        4 2c3a6300 2c3a5960 I/P/A     0 NONE
        8 2c3a6360 2c3a4f00 I/P/A     0 NONE
```

再来看看 Oracle 9i 中的情况，以下是 Library Cache Hash Table 的转储输出：

```
LIBRARY CACHE HASH TABLE: size=131072 count=217
Buckets with more than 20 objects:
NONE
Hash Chain Size     Number of Buckets
---------------     -----------------
              0                130855
              1                   217
              2                     0
              3                     0
              4                     0
              5                     0
              6                     0
              7                     0
              8                     0
              9                     0
             10                     0
             11                     0
             12                     0
             13                     0
             14                     0
             15                     0
             16                     0
             17                     0
             18                     0
```

```
            19                    0
            20                    0
           >20                    0
```

Oracle 9i 中通过新的方式记录 Library Cache 的使用状况。按不同的 Hash Chain Size 代表 Library Cache 中包含不同对象的个数。0 表示 Free 的 Bucket，>20 表示包含超过 20 个对象的 Bucket 的个数。从以上列表中看到,包含一个对象的 Buckets 有 217 个，包含 0 个对象的 Buckets 有 130855 个。

来验证一下以上理论：

```
[oracle@jumper udump]$ cat hsjf_ora_15800.trc |grep BUCKET|more
BUCKET 12:
BUCKET 12 total object count=1
BUCKET 385:
BUCKET 385 total object count=1
BUCKET 865:
BUCKET 865 total object count=1
...
[oracle@jumper udump]$ cat hsjf_ora_15800.trc |grep BUCKET|wc -l
434
```

434/2 = 217，证实了我们的猜想。通过 HASH TABLE 算法的改进,Oracle Library Cache 管理的效率得以提高。从 Oracle 9i 至 Oracle 11g，HASH TABLE 的管理算法一直如此。

接下来进一步讨论一下 Shared Pool 的内容存储。先进行相应查询，获得测试数据：

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.3.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production
SQL> connect eygle/eygle
Connected.
SQL> create table emp as select * from scott.emp;
Table created.
SQL> connect / as sysdba
Connected.
SQL> startup force;
SQL> set linesize 120
SQL> connect scott/tiger
Connected.
SQL> select * from emp;
SQL> connect eygle/eygle
Connected.
SQL> select * from emp;
```

```
SQL> select SQL_TEXT,VERSION_COUNT,HASH_VALUE,to_char(HASH_VALUE,'xxxxxxxxxx') HEX,ADDRESS
  2  from v$sqlarea where sql_text like 'select * from emp%';
SQL_TEXT             VERSION_COUNT HASH_VALUE HEX         ADDRESS
-------------------- ------------- ---------- ----------- --------
select * from emp                2 2648707557    9de011e5 52D9EA28
SQL> select sql_text,username,ADDRESS,HASH_VALUE,
  2  to_char(HASH_VALUE,'xxxxxxxxxx') HEX_HASH_VALUE,CHILD_NUMBER,CHILD_LATCH
  3  from v$sql a,dba_users b
  4  where a.PARSING_USER_ID = b.user_id and  sql_text like 'select * from emp%';
SQL_TEXT             USERNAME   ADDRESS  HASH_VALUE HEX_HASH_VA CHILD_NUMBER CHILD_LATCH
-------------------- ---------- -------- ---------- ----------- ------------ -----------
select * from emp    SCOTT      52D9EA28 2648707557    9de011e5            0           1
select * from emp    EYGLE      52D9EA28 2648707557    9de011e5            1           1
```

这里可以看出 V$SQLAREA 和 V$SQL 两个视图的不同之处，V$SQL 中为每一条 SQL 保留一个条目，而 V$SQLAREA 中根据 SQL_TEXT 进行 GROUP BY，通过 version_count 计算子指针的个数。

在以上两次查询中，两条 SQL 语句因为其代码完全相同，所以其 ADDRESS、HASH_VALUE 也完全相同。这就意味着，这两条 SQL 语句在共享池中的存储位置是相同的（尽管其执行计划可能不同），代码得以共享。在 SQL 解析过程中，Oracle 将 SQL 文本转换为相应的 ASCII 数值，然后根据数值通过 Hash 函数计算其 HASH_VALUE，再通过 HASH_VALUE 在 Shared Pool 中寻找是否存在相同的 SQL 语句，如果存在则进入下一步骤；如果不存在则尝试获取 Shared Pool Latch，请求内存，存储该 SQL 代码。

在这里有一个问题需要说明一下，因为大小写字母的 ASCII 值是不同的，所以 Oracle 会把大小写不同的代码作为不同的 SQL 来处理。看一下测试：

```
SQL> select * from scott.dept;
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
SQL> select * from scott.DEPT;
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
SQL> col sql_text for a30
```

```
SQL> select sql_text,hash_value from v$sql where sql_text like 'select * from scott%';

SQL_TEXT                       HASH_VALUE
------------------------------ ----------
select * from scott.DEPT       4096614922
select * from scott.dept       2089404358
```

注意到以上输出，仅仅是大小写的不同使得原本相同的 SQL 语句变成了两条“不同的代码”，所以从这里可以看出，SQL 的规范编写非常重要。

SQL 解析首先要进行的是语法解析，语法无误后进入下一个步骤，进行语义分析，在此步骤中，Oracle 需要验证对象是否存在、相关用户是否具有权限、引用的是否是相同的对象。

对于先前的查询，实际上 emp 表来自不同的用户，那么 SQL 的执行计划也就不同了（当然影响 SQL 执行计划的因素还有很多，包括优化器模式等），通过对象依赖关系可以看到这个不同：

```
SQL> select a.*,to_char(to_hash,'xxxxxxxxxx') Hex_HASH_VALUE
  2  from V$OBJECT_DEPENDENCY  a where to_name='EMP';
FROM_ADD  FROM_HASH TO_OWNER  TO_NAME TO_ADDRE    TO_HASH TO_TYPE HEX_HASH_VA
-------- ---------- --------- ------- -------- ---------- ------- -----------
52D9EA28 2648707557 SCOTT     EMP     52D9DEBC  828228010       2    315dc1aa
52D9EA28 2648707557 EYGLE     EMP     52D82E58 1930491453       2    7310f63d
```

接下来分别在不同级别对 Library Cache 进行转储，收集 trace 文件用于分析：

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 1';
Session altered.
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 2';
Session altered.
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 4';
Session altered.
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 8';
Session altered.
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 16';
Session altered.
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 32';
Session altered.
```

在转向 TRACE 文件分析之前，通过图 6-10 来看一下 Library Cache 的结构。

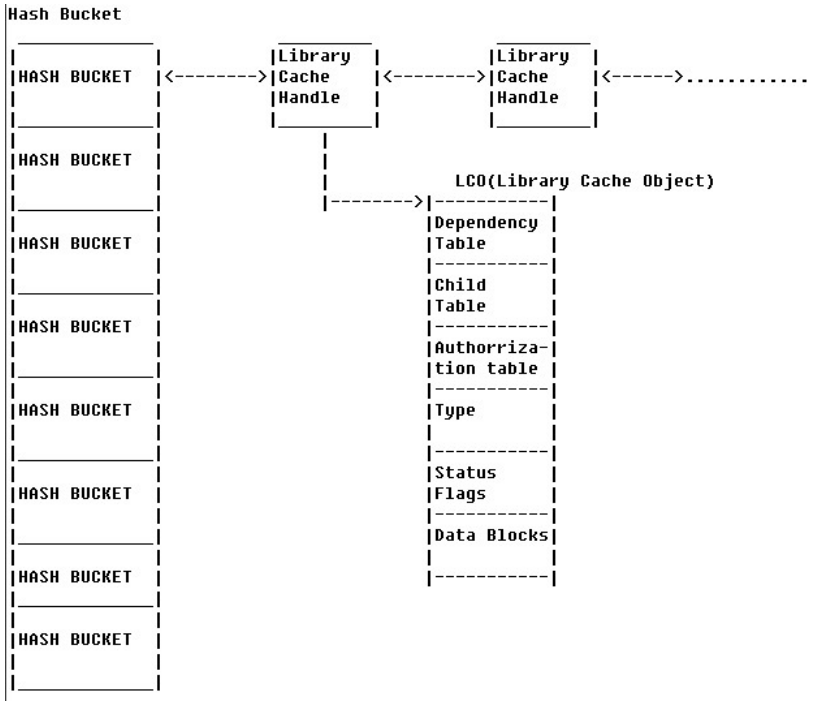


图 6-10　Library Cache 的结构

回忆一下前面介绍过的 Buffer Cache 的管理，其中 Bucket→BH→Buffer 的管理方式与以上 Library Cache 的管理原理完全类似。

Library Cache Handle 可以被看作库缓存对象的概要信息，Handle 上存有指针指向 Library Cache Object，Handle 中还包含对象名、namespace、时间戳、引用列表、锁定对象及 pin 对象列表等信息。这里还需要说明的是 Handle 上的指针指向的是 Library Cache Object 的 Heap 0，库缓存对象可能占用多个内存 Heap，Heap 0 则记录了控制信息，包括对象类型、对象依赖表、指向其他 Heap 的指针等，图 6-11 列举了主要 Shared Pool 对象的具体内存结构组成。

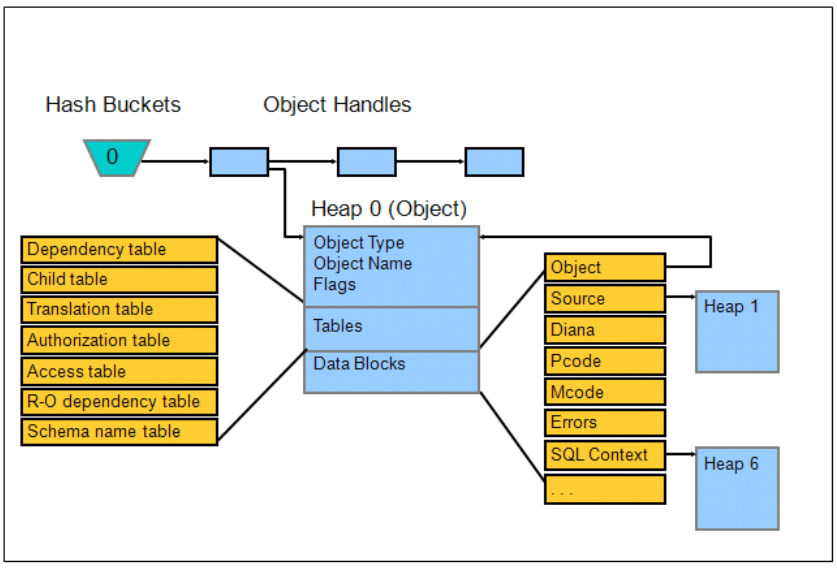


图 6-11　Shared Pool 对象的内存结构

接下来通过 Trace 文件看看以上提到的具体信息，由以上 v$sql 视图得到以上查询的 hash_value 为 9de011e5，ADDRESS 为 52D9EA28。在 Bucket 4851 中，找到了 select * from emp 这条 SQL 语句：

```
BUCKET 4581:
  LIBRARY OBJECT HANDLE: handle=52d9ea28
  name=select * from emp
  hash=9de011e5 timestamp=08-26-2004 10:24:43
==>这个 hash 正是 v$sql 中该 sql 语句的 hash_value 值
  namespace=CRSR flags=RON/TIM/PN0/SML/[12010000]
  kkkk-dddd-llll=0000-0001-0001 lock=0 pin=0 latch#=1
  lwt=0x52d9ea40[0x52d9ea40,0x52d9ea40] ltm=0x52d9ea48[0x52d9ea48,0x52d9ea48]
  pwt=0x52d9ea58[0x52d9ea58,0x52d9ea58] ptm=0x52d9eab0[0x52d9eab0,0x52d9eab0]
  ref=0x52d9ea30[0x52d9ea30, 0x52d9ea30] lnd=0x52d9eabc[0x52d9eabc,0x52d9eabc]
    LIBRARY OBJECT: object=52d9e7b0
    type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
    CHILDREN: size=16
    child#    table reference   handle
    ------ -------- --------- --------
         0 52d9e96c  52d9e6cc 52d9e4ac
         1 52d9e96c  52d9e70c 52d885cc
==>这就是我们前边提到过的子指针,每个都指向了一个不同的 handle
    DATA BLOCKS:
    data#     heap  pointer status pins change    alloc(K)  size(K)
    ----- -------- -------- ------ ---- ------     -------- --------
        0 52d9e9b8 52d9e838 I/-/A     0 NONE       0.86     1.09
==>此处的 heap 就是指内存地址.
==>这里存放的就是 SQL 代码及用户连接信息
    HEAP DUMP OF DATA BLOCK 0:
*******************************************************
HEAP DUMP heap name="library cache"  desc=0x52d9e9b8
 extent sz=0x224 alt=32767 het=16 rec=9 flg=2 opc=0
 parent=0x5000002c owner=0x52d9e7b0 nex=(nil) xsz=0x224
EXTENT 0 addr=0x52d9e558
  Chunk 52d9e560 sz=      540    perm      "perm           "  alo=448
EXTENT 1 addr=0x52d9e798
  Chunk 52d9e7a0 sz=      360    perm      "perm           "  alo=360
  Chunk 52d9e908 sz=       88    free      "               "
  Chunk 52d9e960 sz=       76    freeable  "kgltbtab       "
```

```
Total heap size    =     1064
FREE LISTS:
 Bucket 0 size=0
  Chunk 52d9e908 sz=       88    free      "               "
Total free space   =       88
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d9e560 sz=      540    perm      "perm           "  alo=448
  Chunk 52d9e7a0 sz=      360    perm      "perm           "  alo=360
Permanent space    =      900
*******************************************************
  BUCKET 4581 total object count=1
```

继续以 handle:52d885cc 为例看一下 Library Cache Object 的结构：

```
*******************************************************
  LIBRARY OBJECT HANDLE: handle=52d885cc
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0041-0041 lock=0 pin=0 latch#=1
  lwt=0x52d885e4[0x52d885e4,0x52d885e4] ltm=0x52d885ec[0x52d885ec,0x52d885ec]
  pwt=0x52d885fc[0x52d885fc,0x52d885fc] ptm=0x52d88654[0x52d88654,0x52d88654]
  ref=0x52d885d4[0x52d9e70c, 0x52d9e70c] lnd=0x52d88660[0x52d88660,0x52d88660]
    LIBRARY OBJECT: object=52d82a24
    type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
    DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle position flags
    ----------- -------- --------- -------- -------- ------------------
              0 52d82be0  52d82b20 52d82e58       14 DEP[01]
```

在 dependency 部分看到，这个 Cursor 依赖的对象 handle:**52d82e58**，这个 handle 指向的就是 EYGLE.EMP 表，如果以上两个 CRSR 访问的是同一个对象，那么这两个 SQL 才会是真的共享。这里 SQL 虽然是相同的，访问的却是不同用户的数据表，子指针的概念就体现出来了。

这里看到 52d82e58 指向的是 EYGLE.EMP 这个对象，也就是 EYGLE 所查询的数据表。

```
    ACCESSES: count=1 size=16
    dependency# types
    ----------- -----
              0 0009
    TRANSLATIONS: count=1 size=16
    original    final
    -------- --------
    52d82e58 52d82e58
    DATA BLOCKS:
```

```
   data#     heap  pointer status pins change    alloc(K)  size(K)
   ----- -------- -------- ------ ---- ------     -------- --------
       0 52d8c244 52d827e4 I/-/A     0 NONE       1.09     1.64
       6 52d82ac0 52d817c4 I/-/A     0 NONE       3.70     3.73
```

接下来的 Data Blocks 是个重要的部分，每个控制块包含一个 heap descriptor，指向相应的 heap memory，这个 heap memory 包含的就是 Diana Tree、P-Code、Source Code、Shared Cursor Context Area 等重要数据，也就是通常所说的，解析过的 SQL 及执行计划树，真正到这里以后 SQL 才得以共享，也就真正地避免了硬解析。

```
 HEAP DUMP OF DATA BLOCK 0:
****************************************************
HEAP DUMP heap name="library cache"  desc=0x52d8c244
 extent sz=0x224 alt=32767 het=16 rec=9 flg=2 opc=0
 parent=0x5000002c owner=0x52d82a24 nex=(nil) xsz=0x224
EXTENT 0 addr=0x52d80ff0
  Chunk 52d80ff8 sz=      464    free      "               "
  Chunk 52d811c8 sz=       76    freeable  "kgltbtab       "
EXTENT 1 addr=0x52d827cc
  Chunk 52d827d4 sz=      540    perm      "perm           "  alo=532
EXTENT 2 addr=0x52d82a0c
  Chunk 52d82a14 sz=      252    perm      "perm           "  alo=252
  Chunk 52d82b10 sz=      120    perm      "perm           "  alo=104
  Chunk 52d82b88 sz=       76    freeable  "kgltbtab       "
  Chunk 52d82bd4 sz=       76    freeable  "kgltbtab       "
Total heap size    =     1604
FREE LISTS:
 Bucket 0 size=0
  Chunk 52d80ff8 sz=      464    free      "               "
Total free space   =      464
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d82b10 sz=      120    perm      "perm           "  alo=104
  Chunk 52d827d4 sz=      540    perm      "perm           "  alo=532
  Chunk 52d82a14 sz=      252    perm      "perm           "  alo=252
Permanent space    =      912
****************************************************
    HEAP DUMP OF DATA BLOCK 6:
****************************************************
HEAP DUMP heap name="sql area"  desc=0x52d82ac0
 extent sz=0x1024 alt=32767 het=16 rec=0 flg=2 opc=5
```

```
 parent=0x5000002c owner=0x52d82a24 nex=(nil) xsz=0x0
EXTENT 0 addr=0x52d817ac
  Chunk 52d817b4 sz=     3784    perm      "perm           "  alo=3784
Total heap size    =     3784
FREE LISTS:
 Bucket 0 size=0
Total free space   =        0
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d817b4 sz=     3784    perm      "perm           "  alo=3784
Permanent space    =     3784
MARKS:
  Mark 0x52d8237c
*******************************************************
```

接下来的信息包括了 handle=52d82e58，也就是 SQL 依赖的对象信息，记录的是 EYGLE.EMP 表信息：

```
BUCKET 63037:
  LIBRARY OBJECT HANDLE: handle=52d82e58
  name=EYGLE.EMP
  hash=7310f63d timestamp=08-26-2004 10:23:40
  namespace=TABL/PRCD/TYPE flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0501-0501 lock=0 pin=0 latch#=1
  lwt=0x52d82e70[0x52d82e70,0x52d82e70] ltm=0x52d82e78[0x52d82e78,0x52d82e78]
  pwt=0x52d82e88[0x52d82e88,0x52d82e88] ptm=0x52d82ee0[0x52d82ee0,0x52d82ee0]
  ref=0x52d82e60[0x52d82e60, 0x52d82e60] lnd=0x52d82eec[0x52d7dcf0,0x52d89fc8]
    LIBRARY OBJECT: object=52d81594
    type=TABL flags=EXS/LOC[0005] pflags= [00] status=VALD load=0
==>Type:对象类型,这里是一张表
==>flags:代表对象状态
    DATA BLOCKS:
    data#     heap  pointer status pins change    alloc(K)  size(K)
    ----- -------- -------- ------ ---- ------     -------- --------
        0 52d8c1e4 52d8161c I/-/A     0 NONE       0.66     1.09
        8 52d81238 52d80a18 I/-/A     0 NONE       1.10     1.13
       10 52d8129c 52d80ea0 I/-/A     0 NONE       0.12     0.37

    HEAP DUMP OF DATA BLOCK 0:
*******************************************************
HEAP DUMP heap name="library cache"  desc=0x52d8c1e4
```

```
 extent sz=0x224 alt=32767 het=16 rec=9 flg=2 opc=0
 parent=0x5000002c owner=0x52d81594 nex=(nil) xsz=0x224
```

==>每个 heap descriptor 都包含一个 owner 部分，指向所有者，这里的 52d81594 也就是 EYGLE.EMP

==>指向的 Library 对象： LIBRARY OBJECT: object=52d81594

```
EXTENT 0 addr=0x52d81220
  Chunk 52d81228 sz=      540    perm      "perm           "  alo=196
EXTENT 1 addr=0x52d8157c
  Chunk 52d81584 sz=      484    perm      "perm           "  alo=484
  Chunk 52d81768 sz=       40    free      "               "
Total heap size    =     1064
FREE LISTS:
 Bucket 0 size=0
  Chunk 52d81768 sz=       40    free      "               "
Total free space   =       40
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d81228 sz=      540    perm      "perm           "  alo=196
  Chunk 52d81584 sz=      484    perm      "perm           "  alo=484
Permanent space    =     1024
*******************************************************
    HEAP DUMP OF DATA BLOCK 8:
*******************************************************
HEAP DUMP heap name="KGLS heap"  desc=0x52d81238
 extent sz=0x824 alt=32767 het=16 rec=0 flg=2 opc=5
 parent=0x5000002c owner=0x52d81594 nex=(nil) xsz=0x0
EXTENT 0 addr=0x52d80a00
  Chunk 52d80a08 sz=     1124    perm      "perm           "  alo=1124
Total heap size    =     1124
FREE LISTS:
 Bucket 0 size=0
Total free space   =        0
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d80a08 sz=     1124    perm      "perm           "  alo=1124
Permanent space    =     1124
*******************************************************
    HEAP DUMP OF DATA BLOCK 10:
*******************************************************
```

```
HEAP DUMP heap name="KGLS heap"  desc=0x52d8129c
 extent sz=0x824 alt=32767 het=16 rec=0 flg=2 opc=5
 parent=0x5000002c owner=0x52d81594 nex=(nil) xsz=0x15c
EXTENT 0 addr=0x52d80e88
  Chunk 52d80e90 sz=      340    perm      "perm           "  alo=120
Total heap size    =      340
FREE LISTS:
 Bucket 0 size=0
Total free space   =        0
UNPINNED RECREATABLE CHUNKS (lru first):
PERMANENT CHUNKS:
  Chunk 52d80e90 sz=      340    perm      "perm           "  alo=120
Permanent space    =      340
********************************************************
  BUCKET 63037 total object count=1
```

这就是 Shared Pool 中 Library Cache 的结构与原理。

至于 Dictionary Cache 信息则可以通过如下命令进行转储：

```
ALTER SESSION SET EVENTS 'immediate trace name row_cache level N';
```

这里的 N 可以取的值如下：

- 1，转储 dictionary cache 的统计信息；
- 2，转储 hash 表的汇总信息；
- 8，转储 dictionary cache 中的对象的结构信息。

使用 Level 1 进行转储，转储出来的内容就是 V$ROWCACHE 中的统计信息：

```
ROW CACHE STATISTICS:
cache                          size     gets  misses  hit ratio
-------------------------  -------  -------  ------  ---------
dc_tablespaces                  280   437782       7      1.000
dc_free_extents                 168        0       0      0.000
dc_segments                     236    77380    2312      0.971
dc_rollback_segments            232   167148      11      1.000
dc_used_extents                 180        0       0      0.000
dc_tablespace_quotas            176        2       3      0.400
dc_files                        192    13657       5      1.000
dc_objects                      468   576155    7884      0.987
```

使用 Level 2 级转储，则可以获得 ROW CACHE HASH TABLE 信息，输出会显示大量 HASH TABLE 信息，对应于每一种字典信息，都会有一个 HASH TABLE 与之对应，在 Oracle 11gR1 中，共有 50 个 HASH 表，每个表信息类似以下：

```
ROW CACHE HASH TABLE: cid=49 ht=0x3cdb8f30 size=65536
Buckets with more than 20 objects:
```

```
NONE
Hash Chain Size     Number of Buckets
---------------     -----------------
              0                 65536
              1                     0
              2                     0
              3                     0
              4                     0
              5                     0
              6                     0
              7                     0
              8                     0
              9                     0
             10                     0
             11                     0
             12                     0
             13                     0
             14                     0
             15                     0
             16                     0
             17                     0
             18                     0
             19                     0
             20                     0
            >20                     0
[oracle@localhost trace]$ grep "ROW CACHE HASH TABLE" 11gtest_ora_18720.trc|wc -l
50
```

使用Level 8级转储则可以将Dictionary Cache中的数据转储出来，摘录一个对象输出（Row Cache的内容就是由一连串这样的Bucket信息组成）：

```
BUCKET 3:
  row cache parent object: address=0x3eae4484 cid=0(dc_tablespaces)
  hash=a6decc2 typ=9 transaction=(nil) flags=00000002
  own=0x3eae44f0[0x3eae44f0,0x3eae44f0] wat=0x3eae44f8[0x3eae44f8,0x3eae44f8] mode=N
  status=VALID/-/-/-/-/-/-/-/-
  data=
  00000003 45540004 0000504d 00000000 00000000 00000000 00000000 00000000
  00000000 00000000 00000000 00002000 7fffffd 7fffffd 00000080 00000080
  00000000 00000101 00000001 00000080 00000000 00000000 00000000 00000000
  00000a52 00000000 00000080 00000001 00001002 00000000 00000000 00000000
```

```
00000000 0a6decc2 3eae4484 3d3fe0a4 3d3fe0a4
BUCKET 3 total object count=1
```

6.2.7　诊断和解决 ORA-04031 错误

关于 ORA-04031 错误，网络上的相关文档很多，在这里只做一下简要说明。Shared Pool 的主要问题在根本上只有一个，就是**碎片过多带来的性能影响**。前面讲到了 Oracle 8i 和 Oracle 9i 在 Shared Pool 管理上的不同。

1．什么是 ORA-04031 错误

当尝试在共享池分配大块的连续内存失败（很多时候是由于碎片过多，而并非真是内存不足）时，Oracle 首先清除共享池中当前没使用的所有对象，使空闲内存块合并。如果仍然没有足够大的单块内存可以满足需要，就会产生 ORA-04031 错误。

Shared Pool 的内存分配算法相当复杂，ORA-04031 错误的出现原因也有多种，经过简化，可以通过如下一段伪代码来描述 04031 错误的产生：

```
Scan free lists                              --扫描 Free Lists
if (request size of RESERVED Pool size)      --如果请求 RESERVED POOL 空间
     scan reserved list                      --扫描保留列表
if (chunk found)                             --如果发现满足条件的内存块
     check chunk size and perhaps truncate   --检查大小，可能需要分割
     return                                  --返回
do LRU operation for n objects               --如果并非请求 RESERVED POOL 或不能发现足够内存
     scan free lists                         --则转而执行 LRU 操作，释放内存，重新扫描
     if (request sizes exceeds reserved pool min alloc) - 如果请求大于
_shared_pool_reserved_min_alloc
       scan reserved list                          --扫描保留列表
     if (chunk found)                              --如果发现满足条件的内存块
        check chunk size and perhaps truncate      --检查大小，可能需要分割
      return                                      --在 Freelist 或 reserved list 找到则成功返回

signal ORA-4031 error                              --否则报告 ORA-04031 错误。
```

Oracle 关于 04031 错误的解释及建议如下：

```
04031, 00000, "unable to allocate %s bytes of shared memory (\"%s\",\"%s\",\"%s\",\"%s\")"
// *Cause: More shared memory is needed than was allocated in the shared
// pool.
// *Action: If the shared pool is out of memory, either use the
// dbms_shared_pool package to pin large packages,
// reduce your use of shared memory, or increase the amount of
// available shared memory by increasing the value of the
```

```
// INIT.ORA parameters "shared_pool_reserved_size" and
// "shared_pool_size".
// If the large pool is out of memory, increase the INIT.ORA
// parameter "large_pool_size".
```

2．关注相关 Bug 信息

在 Oracle 9iR2 之前的很多 ORA-04031 的错误都和内存泄漏的 Bug 有关，所以是否及时应用相关 Patch 是非常重要的。在 Oracle 8.1.7 中，几乎每个人都曾经遇到 04031 的问题，这同样是因为 Bug 导致的。

表 6-1 是 Oracle 发布的不同版本和 Shared Pool 以及 04031 错误相关的 Bug 列表，摘录在这里供大家参考，如果数据库遇到共享池方面的错误（首先需要确认，你的 shared_pool_size 不是设置得非常小），你应该确认是否符合下列 Bug 的特征；即使目前你并未遇到相关问题，但是阅读一下这些 Bug 描述也可以使你对这些问题有所了解。

表 6-1　Bug 列表

Bug 号	报告版本	修正版本	Bug 描述
2934402	9.2.x	10.x	The variable component of the 'show sga' command will show more memory than expected when SGA_MAX_SIZE is much larger than the SGA size required and you set parameters like PROCESSES, DB_FILES, OPEN_CURSORS to a high value
3490108 (4171368)	9.2.x	10.2.x	Seeing ORA-4031 on 'BAMIMA: Bam Buffer', however lots of free memory on hand in the shared pool. The problem appears to stem from very small chunks used in the Shared Pool (Reserved Area specifically)
4992466 (3046725)	9.2.x	10.2.x	Depending on application work loads, some components in a RAC environment can temporarily consume a lot of SGA memory for "ges enqueues" and this can fragment the shared pool sufficiently to cause ORA-04031 type errors
3090397	9.2.0.3	9.2.0.5	In situations when using ASYNC or SYNC=PARALLEL, the LGWR trace file can indicate ORA-04031 errors and indicate large memory request failures
3854318	9.2.0.5	10.2.x	The problem shows up as heavy 'perm' allocations in the 'sql area' in a heapdump trace. The root issue is allocations related to ansi-join processing and query transformation routines that allocate memory and do not get cleaned up properly
4375655	9.2.0.5	9.2.0.8 and 10g	A large in-list can consume excessive memory if there is a zero length string in the inlist. Do not use '' in an inlist
5246688/ (4184298)	10.1.0.3	10.1.0.5 / 10.2.0.1	If the system is configured with multiple shared pool subpools, and if many sessions are active, it is possible for an allocation imbalance to occur between the subpools. It extreme cases, this could lead to ORA-04031 due to a large number of "session parameters" allocations using up space from one subpool. Workaround: increase the size of the Shared Pool or revert back to one large Shared Pool instead of using subpools
4367986	10.1.0.3	10.2.0.4 / 11.x	Parallel execution cursors are not shared when bind peeking is used. Excessive child cursors created can lead to fragmentation and ORA-4031. Watch for nonshared code showing BIND_PEEKED_PQ_MISMATCH as the problem
3513427	10.1.0.3 / 10.1.0.2	10.1.0.3 / 10.2.0.1	When an ORA-4031 occurs SGA heapdumps can occur every minute or two. This can lead to contention for the shared pool latch
4733833/ 5473945	10.1.0.3 / 10.2.x	10.2.0.3 / 11.x	Full outer join queries can be transformed into union all code with left outer joins and anti joins. The transformation causes multiple entries in the Library Cache

			and high parse counts and can cause additional fragmentation leading to ORA-04031 errors
4467058	10.2.x	11	With ORA-04031 dumping events set, DBWR and LGWR can generate spurious dumps
4237613	10.2.x	10.2.0.3, 11g	Increases to 'ASM extent pointer array' in V$SGASTAT over time, while no decreases are accounted for. This appears to be a leak, but is simply an accounting problem in v$SGASTAT
5045507/ (4507532)	10.2.0.1	10.2.0.2	Seeing aggressive growth of entries in V$SGASTAT for "KGH: NOACCESS"
5918642	10.2.0.3	10.2.0.4	With very large Buffer Cache, cache statistics can cause excessive latch contention. The latch issues can lead to ORA-4031 errors in some cases. Workaround is to set DB_CACHE_ADVICE=OFF. However, this will impact ASMM management in the SGA
5618049	10.2.0.2	10.2.0.4 and 11	4031 during partitioning ddl's and there are a number of allocations with the comment "mvobj part des". This can show up with automated jobs that perform operations on partitions as well
5552515	10.2.0.2	11.x	Commiting DML on table with MView log causes an increase in heap stats for the memory structure "ktcmvcb". No workaround
5548510	10.2.0.2	10.2.0.4 and 11.x	CBO leak. The allocation in the SGA for fix control will grow with each new session. You will see this running: select * from v$sgastat where name like 'qksbg%'
5705795	10.2.0.3	10.2.0.4/ 11.x	This problem is introduced in 10.2.0.3 on Windows 32bit and Linux 32bit only. 10.2.0.3 on other platforms include the fix for this bug# and so are not affected. SQL using bind variables with different bind sizes can lead to a large number of child cursors being created leading to excess shared pool usage and latch contention
6271680	10.2.0.2/ 11g	no fix yet	Free memory reported in the default 4031 trace can reflect large negative values in cases where the memory exceeds 2GB

3．绑定变量和 cursor_sharing

如果 SHARED_POOL_SIZE 设置得足够大，又可以排除 Bug 的因素，那么大多数的 ORA-04031 错误都是由共享池中的大量的 SQL 代码等导致过多内存碎片引起的。

可能的主要原因有：

- SQL 没有足够的共享；
- 大量不必要的解析调用；
- 没有使用绑定变量。

实际上说，应用的编写和调整始终是最重要的内容，Shared Pool 的调整根本上要从应用入手。根本上，使用绑定变量可以充分降低 Shared Pool 和 Library Cache 的 Latch 竞争，从而提高性能。

反复的 SQL 硬解析不仅会消耗大量的 CPU 资源，也会占用更多的内存，严重影响数据库的性能，而使用绑定变量则可以使 SQL 充分共享，实现 SQL 的软解析，提高系统性能。以下是 Oracle 10g 中一个关于绑定变量和非绑定变量的测试对比，由此可以略窥绑定性能影响之一斑。

首先创建测试表并记录解析统计数据：

```
SQL> select * from v$version where rownum <2;
BANNER
```

```
-----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
SQL> select user from dual;
USER
------------------------------
EYGLE
SQL> create table eygle (id number);
Table created.
SQL> SELECT NAME,VALUE FROM V$MYSTAT A,V$STATNAME B WHERE A.STATISTIC#=B.STATISTIC#
  2  AND NAME LIKE 'parse%';
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
parse time cpu                                                           111
parse time elapsed                                                       181
parse count (total)                                                     1688
parse count (hard)                                                       283
parse count (failures)                                                     1
```

进行循环插入数据，以下代码并未使用绑定变量：

```
SQL> begin
  2  for i in 1..10 loop
  3  execute immediate 'insert into eygle values('||i||')';
  4  end loop;
  5  commit;
  6  end;
  7  /
PL/SQL procedure successfully completed.
```

完成之后检查统计信息，注意硬解析次数增加了 10 次，也就是说每次 INSERT 操作都需要进行一次独立的解析：

```
SQL> SELECT NAME,VALUE FROM V$MYSTAT A,V$STATNAME B WHERE A.STATISTIC#=B.STATISTIC#
  2  AND NAME LIKE 'parse%';
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
parse time cpu                                                           111
parse time elapsed                                                       181
parse count (total)                                                     1702
parse count (hard)                                                       293
parse count (failures)                                                     1
```

查询 V$SQLAREA 视图，可以找到这些不能共享的 SQL，注意每条 SQL 都只执行了一次，这些 SQL 不仅解析要消耗密集的 SQL 资源，也要占用共享内存存储这些不同的 SQL 代码：

```
SQL> col sql_text for a60
SQL> SELECT sql_text, version_count, parse_calls, executions
  2    FROM v$sqlarea
  3   WHERE sql_text LIKE 'insert into eygle%';
SQL_TEXT                       VERSION_COUNT PARSE_CALLS EXECUTIONS
------------------------------ ------------- ----------- ----------
insert into eygle values(9)                1           1          1
insert into eygle values(5)                1           1          1
insert into eygle values(8)                1           1          1
insert into eygle values(1)                1           1          1
insert into eygle values(4)                1           1          1
insert into eygle values(6)                1           1          1
insert into eygle values(3)                1           1          1
insert into eygle values(7)                1           1          1
insert into eygle values(2)                1           1          1
insert into eygle values(10)               1           1          1
10 rows selected.
```

重构测试表，进行第二次测试：

```
SQL> drop table eygle;
Table dropped.
SQL> create table eygle (id number);
Table created.
SQL> SELECT NAME,VALUE FROM V$MYSTAT A,V$STATNAME B WHERE A.STATISTIC#=B.STATISTIC#
  2  AND NAME LIKE 'parse%';
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
parse time cpu                                                           112
parse time elapsed                                                       183
parse count (total)                                                     1760
parse count (hard)                                                       296
parse count (failures)                                                     1
```

这一次使用绑定变量，同样 10 次数据插入：

```
SQL> begin
  2  for i in 1..10 loop
  3  execute immediate 'insert into eygle values(:v1)' using i;
  4  end loop;
  5  commit;
  6  end;
  7  /
```

```
PL/SQL procedure successfully completed.
```

现在看一下 SQL 解析的统计数据，硬解析由原来的 296 增加到 298：

```
SQL>
SQL> SELECT NAME,VALUE FROM V$MYSTAT A,V$STATNAME B WHERE A.STATISTIC#=B.STATISTIC#
  2  AND NAME LIKE 'parse%';

NAME                                                                  VALUE
---------------------------------------------------------------- ----------
parse time cpu                                                          112
parse time elapsed                                                      184
parse count (total)                                                    1765
parse count (hard)                                                      298
parse count (failures)                                                    1
```

对于该 SQL，在共享池中只存在一份，解析一次，执行 10 次，这就是绑定变量的优势所在：

```
SQL> col sql_text for a60
SQL> SELECT sql_text, version_count, parse_calls, executions
  2    FROM v$sqlarea
  3   WHERE sql_text LIKE 'insert into eygle%';
SQL_TEXT                            VERSION_COUNT PARSE_CALLS EXECUTIONS
----------------------------------- ------------- ----------- ----------
insert into eygle values(:v1)                   1           1         10
```

在应用程序开发的过程中，都应该优先考虑使用绑定变量（在 JAVA 应用中可以使用 PreparedStatement 进行变量绑定），但是如果应用没有很好地使用绑定变量，那么 Oracle 从 8.1.6 开始提供了一个新的初始化参数用以在 Server 端进行强制变量绑定，这个参数就是 cursor_sharing。最初这个参数有两个可选设置：exact 和 force。

缺省值是 exact，表示精确匹配；force 表示在 Server 端执行强制绑定。在 8i 的版本里使用这个参数对某些应用可以带来极大的性能提高，但是同时也存在一些副作用，比如优化器无法生成精确的执行计划，SQL 执行计划发生改变等（所以如果启用 cursor_sharing 参数时，一定确认用户的应用在此模式下经过充分的测试）。

从 Oracle 9i 开始，Oracle 引入了绑定变量 Peeking 的机制，SQL 在第一次执行时，首先在 Session 的 PGA 中使用具体值生成精确的执行计划，以期可以提高执行计划的准确性，然而 Peeking 的方式只在第一次硬解析时生效，所以仍然可能存在问题，导致后续的 SQL 错误的执行；同时，在 Oracle 9i 中，cursor_sharing 参数有了第 3 个选项：similar。该参数指定 Oracle 在存在柱状图信息时，对于不同的变量值，重新解析，从而可以利用柱状图更为精确地制定 SQL 执行计划。也即当存在柱状图信息时，similar 的表现和 exact 相同；当柱状图信息不存在时，similar 的表现和 force 相同。

但是需要注意的是，在某些版本中（如 Oracle 9.2.0.5），设置 cursor_sharing 为 similar 可能导致 SQL 的 version_count 过高的 Bug，该选项在不同版本中都可能存在问题，是需要斟酌

使用的一个参数，设置该参数不过是一个临时的解决办法，根本的性能提升仍然需要通过优化 SQL 来解决。

除了 Bug 之外，**在正常情况下，由于 Similar 的判断机制，可能也会导致 SQL 无法共享**。在收集了柱状图（Hisogram）信息之后，如果 SQL 未使用绑定变量，当 SQL 使用具备柱状图信息的 Column 时，数据库会认为 SQL 传递过来的每个常量都是不可靠的，需要为每个 SQL 生成一个 Cursor，这种情况被称为 UNSAFE BINDS。大量的 Version_Count 可能会导致数据库产生大量的 cursor: pin S wait on X 等待。解决这类问题，可以设置 CURSOR_SHARING 为 Force 或者删除相应字段上的柱状图信息。

4．使用 Flush Shared Pool 缓解共享池问题

前面提到，本质上 ORA-04031 多数是由于 SQL 编写不当引起，所以如果能够修改应用绑定变量是最好的解决之道。当然如果应用不能修改，或者不能强制变量绑定，那么 Oracle 还提供一种应急处理方法，强制刷新共享池。

```
alter system flush shared_pool;
```

刷新共享池可以帮助合并碎片（small chunks），强制老化 SQL，释放共享池，但是这通常是不推荐的做法，这是因为：

（1）Flush Shared Pool 会导致当前未使用的 cursor 被清除出共享池，如果这些 SQL 随后需要执行，那么数据库将经历大量的硬解析，系统将会经历严重的 CPU 争用，数据库将会产生激烈的 Latch 竞争。

（2）如果应用没有使用绑定变量，大量类似 SQL 不停执行，那么 Flush Shared Pool 可能只能带来短暂的改善，数据库很快就会回到原来的状态。

（3）如果 Shared Pool 很大，并且系统非常繁忙，刷新 Shared Pool 可能会导致系统挂起，对于类似系统尽量在系统空闲时进行。

从 Oracle 9i 开始，Oracle 的共享池算法发生了改变，Flush Shared Pool 的方法已经不再推荐使用。

5．SHARED_POOL_RESERVED_SIZE 参数的设置及作用

还有一个参数是需要提及的：shared_pool_reserved_size。该参数指定了保留的共享池空间，用于满足将来的大的连续的共享池空间请求。当共享池出现过多碎片，请求大块空间会导致 Oracle 大范围的查找并释放共享池内存来满足请求，由此可能会带来较为严重的性能下降，设置合适的 shared_pool_reserved_size 参数，结合 shared_pool_reserved_min_alloc 参数可以用来避免由此导致的性能下降。

这个参数理想值应该大到足以满足任何对 RESERVED LIST 的内存请求，而无需数据库从共享池中刷新对象。这个参数的缺省值是 shared_pool_size 的 5%，通常这个参数的建议值为 shared_pool_size 参数的 10%～20%大小，最大不得超过 shared_pool_size 的 50%。

同样地，在 trace 文件中，可以找到关于保留列表（RESERVED LIST）的内存信息：

```
RESERVED FREE LIST:
  Chunk a6c6d778 sz=  7864320  R-free      "               "
Total reserved free space   =  7864320
```

shared_pool_reserved_min_alloc 这个参数的值控制保留内存的使用和分配。如果一个足够尺寸的大块内存请求在共享池空闲列表中没能找到，内存就从保留列表（RESERVED LIST）中分配一块比这个值大的空间。

在不同版本中，该参数的缺省值一直是 4400，以下输出来自 Oracle 11g 版本：

```
SQL> @GetHidPar
Enter value for name: shared_pool_reserved_min_alloc
old   4:    AND x.ksppinm LIKE '%&name%'
new   4:    AND x.ksppinm LIKE '%shared_pool_reserved_min_alloc%'
NAME                                     VALUE
---------------------------------------- --------------------
_shared_pool_reserved_min_alloc          4400
```

这个参数默认的值对于大多数系统来说都足够了。如果你的系统经常出现的 ORA-04031 错误都是请求大于 4400 的内存块，那么就可能需要增加 shared_pool_reserved_size 参数设置。

而如果主要的引发 LRU 合并、老化并出现 04031 错误的内存请求在 4100～4400 byte 之间，那么降低_shared_pool_reserved_min_alloc 同时适当增大 SHARED_POOL_RESERVED_SIZE 参数值通常会有所帮助。设置_shared_pool_reserved_min_alloc=4100 可以增加 Shared Pool 成功满足请求的概率。需要注意的是，这个参数的修改应当结合 Shared Pool Size 和 Shared Pool Reserved Size 的修改。设置_shared_pool_reserved_min_alloc=4100 是经过证明的可靠方式，不建议设置更低。

查询 v$shared_pool_reserved 视图可以用于判断共享池问题的引发原因，以下查询来自一个业务系统，注意系统出现过 76 次的请求失败，最后一次请求失败的内存块大小是 4215 Bytes，在这种情况下，就可以考虑降低_shared_pool_reserved_min_alloc 参数设置：

```
SQL> SELECT free_space, avg_free_size,used_space,
  2  avg_used_size,request_failures, last_failure_size
  3  FROM v$shared_pool_reserved;
FREE_SPACE AVG_FREE_SIZE USED_SPACE AVG_USED_SIZE REQUEST_FAILURES LAST_FAILURE_SIZE
---------- ------------- ---------- ------------- ---------------- -----------------
  23068672   41161.7398    2062076    25457.7284               76              4215
```

如果 request_failures>0 并且 last_failure_size>shared_pool_reserved_min_alloc，那么 ORA-04031 错误就可能是因为共享池保留空间缺少连续空间所致。要解决这个问题，可以考虑加大 shared_pool_reserved_min_alloc 来降低缓冲进共享池保留空间的对象数目，并增大 shared_pool_reserved_size 和 shared_pool_size 来加大共享池保留空间的可用内存。

如果 request_failures>0 并且 last_failure_size<shared_pool_reserved_min_alloc 或者 request_failures＝0 并且 last_failure_size<shared_pool_reserved_min_alloc，那么是因为在库高速缓冲缺少连续空间导致 ORA-04031 错误。

对于这一类情况应该考虑降低 shared_pool_reserved_min_alloc 以放入更多的对象到共享池保留空间中并且加大 shared_pool_size。

6．Oracle 10.2.0.5 04031 错误诊断一则

在客户的 10.2.0.5 数据库中，出现 ORA-04031 错误案例，以下是用户的数据库环境：

```
Dump file d:\oracle\admin\ndc\udump\ndc_ora_7892.trc
Sun Mar 11 07:36:54 2012
ORACLE V10.2.0.5.0 - Production vsnsta=0
vsnsql=14 vsnxtr=3
Oracle Database 10g Enterprise Edition Release 10.2.0.5.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
Windows Server 2003 Version V5.2 Service Pack 1
CPU                 : 4 - type 586, 2 Physical Cores
Process Affinity    : 0x00000000
Memory (Avail/Total): Ph:2029M/3583M, Ph+PgF:3895M/5475M, VA:884M/2047M
Instance name: ndc
```

以下是错误跟踪文件中的 4031 诊断信息，其中显示 SQL 请求 4064 字节的内存不能获得，出现 ORA-04031 错误：

```
*** 2012-03-11 07:36:54.753
================================
Begin 4031 Diagnostic Information
================================
The following information assists Oracle in diagnosing
causes of ORA-4031 errors.  This trace may be disabled
by setting the init.ora _4031_dump_bitvec = 0
====================================
Allocation Request Summary Informaton
====================================
Current information setting:  04014fff
  SGA Heap Dump Interval=3600 seconds
  Dump Interval=300 seconds
  Last Dump Time=03/11/2012 07:36:53
  Dump Count=1
Allocation request for:      kglsim heap
  Heap: 06A136F0, size: 4064
******************************************************
HEAP DUMP heap name="sga heap(1,0)"  desc=06A136F0
 extent sz=0xfc4 alt=108 het=32767 rec=9 flg=-126 opc=0
 parent=00000000 owner=00000000 nex=00000000 xsz=0x400000
 latch set 1 of 1
 durations enabled for this heap
```

```
 reserved granules for root 0 (granule size 4194304)
====================
```

以上错误显示，SGA 内存请求的是“kglsim heap”内存，KGLSIM 是指 Library Cache SIMulator,用于进行 Shared Pool/Library Cache advisor 的评估。

在下图查询输出中，隐含参数_library_cache_advice 用于对库缓存进行评估建议，如果我们不需要参考这个建议指标，可以考虑关闭这个参数；隐含参数_kglsim_maxmem_percent 用于设置用于进行评估运算的共享内存百分比，缺省使用 5%的共享池内存用于评估：

```
SQL> SELECT x.ksppinm NAME, y.ksppstvl VALUE, x.ksppdesc describ
  2    FROM SYS.x$ksppi x, SYS.x$ksppcv y
  3   WHERE x.indx = y.indx AND x.ksppdesc LIKE '%&par%'
  4  /
Enter value for par: advice
old   3:  WHERE x.indx = y.indx AND x.ksppdesc LIKE '%&par%'
new   3:  WHERE x.indx = y.indx AND x.ksppdesc LIKE '%advice%'

NAME                           VALUE                DESCRIB
------------------------------ -------------------- ------------------------------------------------------------
_library_cache_advice          TRUE                 whether KGL advice should be turned on
_kglsim_maxmem_percent         5                    max percentage of shared pool size to be used for KGL advice
_smm_advice_log_size           0                    overwrites default size of the PGA advice workarea history l
                                                    og

_smm_advice_enabled            TRUE                 if TRUE, enable v$pga_advice
```

从转储中的堆栈信息可以看到 4031 出现的内核位置：

```
----- Call Stack Trace -----
calling              call     entry              argument values in hex
location             type     point              (? means dubious value)
-------------------- -------- ------------------ ----------------------------
_ksedst+38           CALLrel  _ksedst1+0         0 0
_ksm_4031_dump+1178  CALLrel  _ksedst+0          0
_ksmasg+286          CALLrel  _ksm_4031_dump+0   4014FFF 4042D90 6A136F0
                                                 21071000 FE0 45D2D78
_kghnospc+849        CALLreg  00000000           893C7D0 4042D90 FE0 FE0
                                                 4042DB4 1036B2C8 21071000 0
                                                 45D2D78
_kghalo+1631         CALLrel  _kghnospc+0        893C7D0 6A136F0 FE0 45D2D78
                                                 1071000
_kglsim_chk_heaplis  CALLrel  _kghalo+0
t+350
_kglsim_upd_newhp+1  CALLrel  _kglsim_chk_heaplis 893C7D0 7
481                           t+0
_kksUpdateSimulator  CALLrel  _kglsim_upd_newhp+0 893C7D0 37BDDBD8 4 176 19 0 0
+122                                             0 2B05C16C
_kksEndOfCompile+62  CALLrel  _kksUpdateSimulator 893C7D0 A345484
7                             +0
_opitca+4456         CALLrel  _kksEndOfCompile+0 893C7D0 A345484 1
__PGOSF285__kksFull  CALLrel  _opitca+0          A345484 29CF0064
TypeCheck+15
_rpiswu2+426         CALLreg  00000000           1036BB90
_kksLoadChild+24945  CALLrel  _rpiswu2+0         40DE1E80 1B 2B05C1BC C
                                                 3079B768 1B 2B05C1E8 0 5E9C00
                                                 0 1036BB90 0
_kxsGetRuntimeLock+  CALLrel  _kksLoadChild+0    893C7D0 2C3DCA2C 1036C8D4
1669
_kksfbc+10697        CALLrel  _kxsGetRuntimeLock+ 893C7D0 A345484 1036C8D4 3 1
                              0
_opiexe+2635         CALLrel  _kksfbc+0          A345484 3 102 0 0 1036CB70
                                                 1036CE24 0
```

在日志文件中可以看到空间分配失败信息 4064 字节：

```
Sun Mar 11 09:45:27 2012
ORA-12012: error on auto execute of job 85
ORA-12008: error in materialized view refresh path
```

```
ORA-04031: unable to allocate 4064 bytes of shared memory ("shared pool","SELECT /*+ */
DISTINCT "A1"....","sga heap(1,0)","kglsim heap")
ORA-02063: preceding line from CSLNDC
ORA-06512: at "SYS.DBMS_SNAPSHOT", line 1883
ORA-06512: at "SYS.DBMS_SNAPSHOT", line 2089
ORA-06512: at "SYS.DBMS_IREFRESH", line 683
ORA-06512: at "SYS.DBMS_REFRESH", line 195
ORA-06512: at line 1
```

在这个案例中，设置 _shared_pool_reserved_min_alloc 为 4000 字节即可消除 4064 字节的请求失败。在客户的案例中，66 次 4031 错误中，其中 23 次请求 4064 字节，43 次请求 3936 字节，缩减 _shared_pool_reserved_min_alloc 设置至 3900 将极大帮助这个数据库实例大大降低 04031 错误的发生概率：

```
MacBook-Pro:4031 eygle$ grep Heap: *|grep 4064|wc -l
      23
MacBook-Pro:4031 eygle$ grep Heap: *|grep 3936|wc -l
      43
MacBook-Pro:4031 eygle$ grep Heap: *|wc -l
      66
MacBook-Pro:4031 eygle$ grep Heap: *|awk '{print $5}'|sort -u
3936
4064
```

在客户系统中，还多次出现 DBMS_STATS 包统计信息收集的 04031 错误，这个后台任务通常在夜里 22：00 调用执行，需要较高的内存：

```
Wed Feb 29 22:00:36 中国标准时间 2012
Errors in file d:\oracle\admin\cslndc\bdump\cslndc_j001_7584.trc:
ORA-12012: 自动执行作业 8896 出错
ORA-04031: 无法分配 ORA-04031: 无法分配 4064 字节的共享内存 ("shared pool","SELECT CASE WHEN BITAND(FLAG...","sga heap(1,0)","kglsim heap")
ORA-06512: 在 "SYS.DBMS_STATS", line 21496
 字节的共享内存 ("","","","")

Wed Feb 29 22:00:36 中国标准时间 2012
Errors in file d:\oracle\admin\cslndc\bdump\cslndc_j001_7584.trc:
ORA-04031: 无法分配  字节的共享内存 ("","","","")
ORA-04031: 无法分配 ORA-04031: 无法分配 3936 字节的共享内存 ("shared pool","update sys.scheduler$_job se...","sga heap(1,0)","kglsim object batch")
ORA-04031: 无法分配 ORA-04031: 无法分配 4064 字节的共享内存 ("shared pool","SELECT CASE WHEN BITAND(FLAG...","sga heap(1,0)","kglsim heap")
ORA-06512: 在 "SYS.DBMS_STATS", line 21496
 字节的共享内存 ("","","","")
 字节的共享内存 ("","","","")
ORA-04031: 无法分配 ORA-04031: 无法分配 4064 字节的共享内存 ("shared pool","SELECT CASE WHEN BITAND(FLAG...","sga heap(1,0)","kglsim heap")
ORA-06512: 在 "SYS.DBMS_STATS", line 21496
 字节的共享内存 ("","","","")

Wed Feb 29 22:01:47 中国标准时间 2012
Errors in file d:\oracle\admin\cslndc\bdump\cslndc_j002_1848.trc:
ORA-12012: 自动执行作业 8897 出错
ORA-04031: 无法分配 ORA-04031: 无法分配 3936 字节的共享内存 ("shared pool","UPDATE SYS.WRI$_ADV_USAGE SE...","sga heap(1,0)","kglsim object batch")
ORA-06512: 在 "SYS.PRVT_ADVISOR", line 1709
ORA-06512: 在 "SYS.DBMS_ADVISOR", line 186
ORA-06512: 在 "SYS.DBMS_SPACE", line 1500
ORA-06512: 在 "SYS.DBMS_SPACE", line 1566
 字节的共享内存 ("","","","")
```

所以，设置足够的共享内存，并且根据错误信息及时做出调整，对于数据库的正常运行是必不可少的。

7. 其他

此外，某些特定的 SQL，较大的指针或者大的 Package 都可能导致 ORA-04031 错误。在很多 ERP 软件中，这样的情况非常常见。在这种情况下，可以考虑把这个大的对象 Pin 到共享池中，减少其动态请求、分配所带来的负担。

使用 dbms_shared_pool.keep 系统包可以把这些对象 pin 到内存中，最常见的 SYS.STANDARD、SYS.DBMS_STANDARD 等都是常见的候选对象。

> 注意：要使用 DBMS_SHARED_POOL 系统包，首先需要运行 dbmspool.sql 脚本，该脚本会自动调用 prvtpool.plb 脚本创建所需对象。

引发 ORA-04031 错误的因素还有很多，通过设置相关参数如 session_cached_cursors、cursor_space_for_time 等也可以解决一些性能问题并带来针对性的性能改善，这里不再过多讨论。

8．模拟 ORA-04031 错误

Oracle 9iR2 开始引入了段级统计信息（segment statistics）收集的新特性，其中一个新引入的视图是 v$segstat，查询该视图会引发 Shared Pool 的内存泄露（在 9201～9206 版本中都存在此问题，本案例测试来自 Windows 平台 9206），我们可以利用这一问题来模拟 ORA-00431 错误（该 Bug 在 Oracle 10g 中已经修正，所以以下方法并不适用于 Oracle 10g）。

以下是一段测试代码：

```
set heading off
column what format a40
column value format a30

select 'db instance' what, user || '@' || global_name value from global_name
UNION
select '# rows in v$segstat', to_char(count(*)) from v$segstat;

set linesize 200
set time on
set serveroutput on size 300000

declare
  l_temp        char(1);
  l_before      number;
  l_after       number := 0;
  l_loop_times  pls_integer := 1000;    -- try 1000
  l_sleep       number   := 0.00;    -- makes no difference

  cursor c_seg is select * from v$segstat;
```

```
  r_seg  c_seg%ROWTYPE;

   function get_mem return number is
     cursor c_mem is select bytes from v$sgastat
        where name = 'free memory' and pool = 'shared pool';
     r_mem   c_mem%ROWTYPE;
     begin
       open c_mem; fetch c_mem into r_mem; close c_mem;
       return r_mem.bytes;
     end get_mem;
begin
  l_after := get_mem();

  for x in 1..l_loop_times loop
    l_before := l_after;

    OPEN c_seg; FETCH c_seg INTO r_seg; CLOSE c_seg;

    l_after := get_mem();
    dbms_output.put_line ('Loop ' || x || ': (' ||
       to_char(sysdate,'hh24:mi:ss') || ') from ' ||
       to_char(l_before,'999,999,999') || ' to ' ||
       to_char(l_after,'999,999,999') || ' (loss of ' ||
       to_char((l_before-l_after),'9,999,999') || ')');
    dbms_lock.sleep(l_sleep);
  end loop;
end;
/
```

首先来看看之前的状态（测试环境，已经把 Shared Pool 调整降低）：

```
SQL> select * from v$sgastat
  2  where name in('miscellaneous','free memory') and pool='shared pool';
POOL        NAME                    BYTES
----------- ------------------------ ----------
shared pool free memory                3927884
shared pool miscellaneous               5752896
```

执行以上代码：

```
20:49:57 SQL> @d:\mem.leak.sql

# rows in v$segstat                    1034
```

```
db instance                                SYS@EYGLE

Loop 1: (20:50:00) from      782,072 to      769,336 (loss of     12,736)
Loop 2: (20:50:00) from      769,336 to      949,276 (loss of   -179,940)
Loop 3: (20:50:00) from      949,276 to      978,872 (loss of    -29,596)
Loop 4: (20:50:00) from      978,872 to      970,436 (loss of      8,436)
Loop 5: (20:50:00) from      970,436 to      962,012 (loss of      8,424)
Loop 6: (20:50:00) from      962,012 to      949,364 (loss of     12,648)
Loop 7: (20:50:00) from      949,364 to      946,280 (loss of      3,084)
Loop 8: (20:50:00) from      946,280 to      993,064 (loss of    -46,784)
Loop 9: (20:50:00) from      993,064 to      984,640 (loss of      8,424)
Loop 10: (20:50:00) from      984,640 to    1,092,628 (loss of   -107,988)
declare
*
ERROR at line 1:
ORA-04031: unable to allocate 4212 bytes of shared memory ("shared pool","unknown object","sga
heap(1,0)","obj stat memor")
ORA-06512: at line 26

20:50:10 SQL>  alter session set events 'immediate trace name heapdump level 2';

Session altered.
20:50:20 SQL> select * from v$sgastat
20:50:20   2  where name in('miscellaneous','free memory') and pool='shared pool';

shared pool free memory                  888268
shared pool miscellaneous              10144656
```

转储共享内存，可以看到：

```
……
Bucket 248 size=3948
  Chunk 7b69b5a0 sz=     3956    free      "               "
  Chunk 7b6d16b4 sz=     3956    free      "               "
 Bucket 249 size=4012
  Chunk 7b5da178 sz=     4064    free      "               "
  Chunk 7b66360c sz=     4060    free      "               "
  Chunk 7b554204 sz=     4060    free      "               "
  Chunk 7b6e2af4 sz=      4060    free      "               "
  Chunk 7b76b8b4 sz=     4052    free      "               "
 Bucket 250 size=4108
```

```
 Bucket 251 size=8204
 Bucket 252 size=16396
 Bucket 253 size=32780
 Bucket 254 size=65548
Total free space   =   180396
```

当前最大的 Chunk size 是 4052，所以请求 4212 时出现了 04031 错误。

如果系统的 ORA-04031 错误通常都是在 4200 左右出现，如前文提到的，可以通过修改_shared_pool_reserved_min_alloc 参数设置减少 04031 错误的出现，可以比较一下。

（1）缺省情况下，_shared_pool_reserved_min_alloc = 4400，在 04031 错误情况下，来看一下保留池的使用：

```
RESERVED FREE LIST:
  Chunk 7a000038 sz=    85940  R-free      "               "
  Chunk 7a400038 sz=    85940  R-free      "               "
  Chunk 7a800038 sz=    85940  R-free      "               "
  Chunk 7ac00038 sz=    85940  R-free      "               "
  Chunk 7b000038 sz=    85940  R-free      "               "
  Chunk 7b400038 sz=    85940  R-free      "               "
Total reserved free space   =   515640
```

保留池保留了 85940 的 Chunk Size，未被使用。

（2）修改_shared_pool_reserved_min_alloc = 4100，在 04031 错误情况下，来看一下保留池的使用。

修改方式如下，修改参数后需要重新启动数据库：

```
21:01:43 SQL> alter system set "_shared_pool_reserved_min_alloc"=4100 scope=spfile;
System altered.
```

修改后保留池的使用情况：

```
RESERVED FREE LIST:
  Chunk 7b414994 sz=     1624  R-free      "               "
  Chunk 7b014994 sz=     1624  R-free      "               "
  Chunk 7ac14988 sz=     1636  R-free      "               "
  Chunk 7a814994 sz=     1624  R-free      "               "
  Chunk 7a414988 sz=     1636  R-free      "               "
  Chunk 7a014994 sz=     1624  R-free      "               "
Total reserved free space   =     9768
```

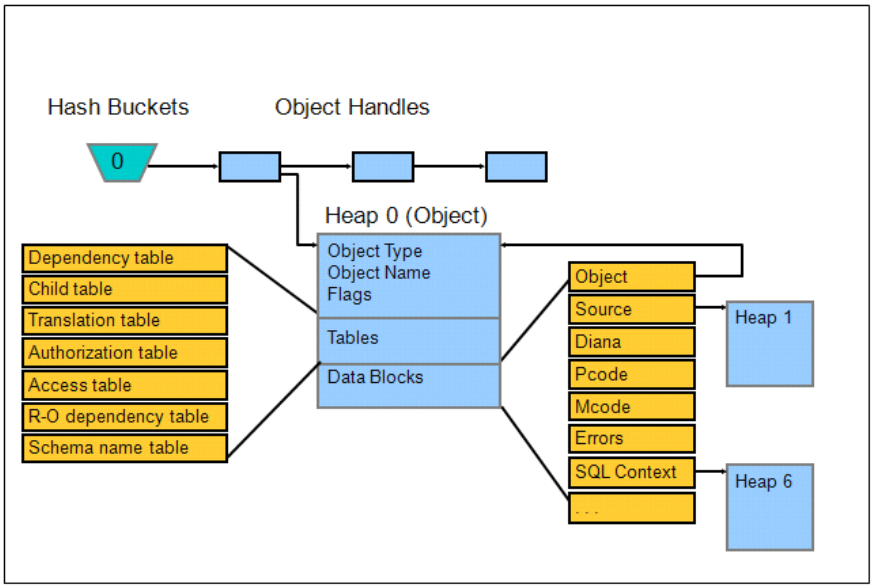
可以看到，在修改了_shared_pool_reserved_min_alloc 参数以后，保留池的使用更为充分，从而使得 ORA-04031 错误的出现得以延迟。

> 提示：本例提供一种方法模拟 ORA-04031 错误，你可以在测试环境中模拟和研究 04031 问题，但是严禁在生产环境中使用。

6.2.8 Library Cache Pin 及 Library Cache Lock 分析

Oracle 使用两种数据结构来进行 Library Cache 的并发访问控制：lock 和 pin。Lock 可以被认为是解析锁，而 Pin 则可以被认为是以读取或改变对象内容为目的所加的短时锁。之所以将 Library Cache Object 对象分开，使用多个锁定来保护，其中的一个重要目的就是为了提高并发。

Lock 比 Pin 具有更高的级别。Lock 在对象 handle 上获得，在 pin 一个对象之前，必须首先获得该 handle 的锁定。Handle 可以理解为 Libray Cache 对象的 Buffer Header，其中包含了库缓存对象的名称、标记、指向具体对象的内存地址指针等信息。再次引用一下前文曾经提到的图表，通过下图我们可以清晰的看到 Object Handles 和 Heaps 的关系：



锁定主要有三种模式: Null，share，Exclusive。在读取访问对象时,通常需要获取 Null(空)模式以及 share(共享)模式的锁定。在修改对象时,需要获得 Exclusive(排他)锁定。Library Cache Lock 根本作用就是控制多个 Oracle 客户端对同一个 Library Cache 对象的并发访问，通过对 Library Cache Object Hadle 上加锁来防止非兼容的访问。常见的使用或保护包括：

1. 一个客户端防止其他客户端访问同一对象
2. 一个客户端可以通过锁定维持相对长时间的依赖性（例如，防止其他客户端修改对象）
3. 当在 Library Cache 中定位对象时也需要获得这个锁定

在锁定了 Library Cache 对象以后，一个进程在访问之前必须 pin 该对象。同样 pin 有三种模式，Null,shared 和 exclusive。只读模式时获得共享 pin,修改模式获得排他 pin。通常我们访问、执行过程、Package 时获得的都是共享 pin，如果排他 pin 被持有,那么数据库此时就要产生等待。

为了实现更好的性能，从 Oracle10gR2 开始，Library Cache Pin 已经逐渐被互斥机制（Mutex）所取代，在 Oracle Database 11g 中，这个变化就更为明显，关于这部分内容请参考第九章“Oracle

10g/11g Latch 机制的变化”一节。

在很多 statspack 的 report 中,我们可能看到以下等待事件:

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                             Wait     % Total
Event                                        Waits      Time (cs)    Wt Time
-------------------------------------------- ---------- ---------- -------
library cache lock                               75,884  1,409,500   48.44
latch free                                   34,297,906  1,205,636   41.43
library cache pin                                   563    142,491    4.90
db file scattered read                          146,283     75,871    2.61
enqueue                                           2,211     13,003     .45
          -------------------------------------------------------------
```

这里的 library cache lock 和 library cache pin 都是我们关心的.接下来就研究一下这几个等待事件。

1．LIBRARY CACHE PIN 等待事件

Oracle 文档上这样介绍这个等待事件：library cache pin 是用来管理 library cache 的并发访问的，pin 一个 Object 会引起相应的 heap 被载入内存中（如果此前没有被加载），pins 可以在 Null、Share、Exclusive 这 3 个模式下获得，可以认为 pin 是一种特定形式的锁。

当 library cache pin 等待事件出现时,通常说明该 pin 被其他用户已非兼容模式持有。library cache pin 的等待时间为 3 秒钟，其中有 1 秒钟用于 PMON 后台进程，即在取得 pin 之前最多等待 3 秒钟，否则就超时。

library cache pin 的参数有 P1（KGL Handle Address）、P2（Pin Address）和 P3（Encoded Mode & Namespace），常用的主要是 P1 和 P2。

library cache pin 通常是发生在编译或重新编译 PL/SQL、VIEW、TYPES 等 Object 时。编译通常都是显性的,如安装应用程序、升级、安装补丁程序等,另外 ALTER、GRANT、REVOKE 等操作也会使 Object 变得无效，可以通过 Object 的 LAST_DDL_TIME 观察这些变化。

当 Object 变得无效时，Oracle 会在第一次访问此 Object 时试图去重新编译它，如果此时其他 session 已经把此 Object pin 到 library cache 中，就会出现问题，特别时当有大量的活动 session 并且存在较复杂的 dependence 时。在某种情况下，重新编译 Object 可能会花几个小时时间，从而阻塞其他试图去访问此 Object 的进程。

下面通过一个例子来模拟及解释这个等待，以下测试来自 Oracle 9ir2 数据库：

（1）创建测试用存储过程。

```
SQL> create or replace PROCEDURE pining
  2  IS
  3  BEGIN
  4          NULL;
```

```
  5  END;
  6  /
Procedure created.
SQL> create or replace procedure calling
  2  is
  3  begin
  4          pining;
  5          dbms_lock.sleep(3000);
  6  end;
  7  /
Procedure created.
```

（2）模拟竞争。

首先执行 calling 过程，在 calling 过程中调用 pining 过程。此时 pining 过程上获得共享 pin，如果此时尝试对 pining 进行授权或重新编译，将产生 library cache pin 等待，直到 calling 执行完毕。

Session 1：

```
SQL> exec calling
```

此时 calling 开始执行。

Session 2：

```
SQL> grant execute on pining to eygle;
```

此时 Session 2 挂起。

如果在 Oracle 10g 或 Oracle 11g 环境中测试，可以将以上命令更改为：

```
revoke execute on pining from eygle;
```

则同样的现象将会再现。现在开始研究，从 v$session_wait 入手可以得到哪些 session 正在经历 library cache pin 的等待：

```
SQL> select sid,seq#,event,p1,p1raw,p2,p2raw,p3,wait_time wt,seconds_in_wait sw,state
  2  from v$session_wait where event like 'library%';
 SID SEQ# EVENT                 P1         P1RAW         P2     P2RAW    P3 WT SW STATE
---- ---- ------------------- ---------- -------- ---------- -------- --- -- -- --------
   8  268 library cache pin   1389785868 52D6730C 1387439312 52B2A4D0 301  0  2 WAITING
```

等待 3 秒就超时，seq#会发生变化：

```
SQL> /
SID  SEQ# EVENT                       P1 P1RAW            P2 P2RAW     P3  WT SW STATE
---- ----- ------------------- ---------- -------- ---------- -------- --- --- -- -------
   8   269 library cache pin   1389785868 52D6730C 1387439312 52B2A4D0 301   0  2 WAITING
SQL> /
 SID  SEQ# EVENT                       P1 P1RAW            P2 P2RAW     P3  ME IT STATE
---- ----- ------------------- ---------- -------- ---------- -------- --- --- -- -------
   8   270 library cache pin   1389785868 52D6730C 1387439312 52B2A4D0 301   0  0 WAITING
```

在这个输出中，P1 列是 Library Cache Handle Address，Pn 字段是十进制表示，PnRaw 字段是十六进制表示。可以看到，library cache pin 等待的对象的 handle 地址为 **52D6730C**。通过这个地址，查询 X$KGLOB（[K]ernel [G]eneric [L]ibrary Cache Manager [OB]ject）视图就可以得到对象的具体信息。

```
col KGLNAOWN for a10
col KGLNAOBJ for a20
select ADDR,KGLHDADR,KGLHDPAR,KGLNAOWN,KGLNAOBJ,KGLNAHSH,KGLHDOBJ
from X$KGLOB
where KGLHDADR ='52D6730C'
/
ADDR     KGLHDADR   KGLHDPAR KGLNAOWN   KGLNAOBJ         KGLNAHSH   KGLHDOBJ
-------- -------- -------- ---------- -------------------- ---------- --------
404F9FF0 52D6730C   52D6730C   SYS        PINING         2300250318  52D65BA4
```

这里 KGLNAHSH 代表该对象的 Hash Value，由此知道，在 PINING 对象上正经历 library cache pin 的等待。然后引入另外一个内部视图 X$KGLPN（[K]ernel [G]eneric [L]ibrary Cache Manager object [P]i[N]s）。

```
select a.sid,a.username,a.program,b.addr,b.KGLPNADR,b.KGLPNUSE,b.KGLPNSES,b.KGLPNHDL,
b.kGLPNLCK, b.KGLPNMOD, b.KGLPNREQ
from v$session a,x$kglpn b
where a.saddr=b.kglpnuse and b.kglpnhdl = '52D6730C' and b.KGLPNMOD<>0
/
  SID USERNAME     PROGRAM                                          ADDR      KGLPNADR KGLPNUSE
KGLPNSES KGLPNHDL KGLPNLCK   KGLPNMOD   KGLPNREQ
-----  -----------  ---------------------------------------  --------  --------  --------
--------- --------- --------- ---------- ----------
   13 SYS          sqlplus@eygle.com (TNS V1-V3) 404FA034 52B2A518 51E2013C 51E2013C 52D6730C
52B294C8          2          0
```

通过联合 v$session，可以获得当前持有该 handle 的用户信息，对于测试 sid=13 的用户正持有该 handle。那么这个用户正在等什么呢?

```
SQL> select * from v$session_wait where sid=13;

       SID       SEQ# EVENT               P1TEXT            P1 P1RAW    P2TEXT          P2
P2RAW    P3TEXT          P3 P3RAW     WAIT_TIME SECONDS_IN_WAIT STATE
---------- ---------- ------------------- --------- ---------- -------- ------- ----------
-------- ------- ---------- -------- ---------- --------------- -------
        13         25 PL/SQL lock timer   duration      120000 0001D4C0                  0
00                        0 00                0            1200 WAITING
```

现在可以看到，这个用户正在等待一次 PL/SQL lock timer 计时。

得到了 sid，就可以通过 v$session.SQL_HASH_VALUE、v$session.SQL_ADDRESS 等字

段关联 v$sqltext、v$sqlarea 等视图获得当前 Session 正在执行的操作。

```
SQL> select sql_text from v$sqlarea where v$sqlarea.hash_value='3045375777';
SQL_TEXT
--------------------------------------------------------------------------------
BEGIN calling; END;
```

这里得到这个用户正在执行 calling 这个存储过程，接下来的工作就应该去检查 calling 在做什么了。这个 calling 做的工作是 dbms_lock.sleep(3000)，也就是 PL/SQL lock timer 正在等待的原因。至此就找到了 library cache pin 的原因。

简化一下以上查询过程。

（1）获得 library cache pin 等待的对象：

```
SELECT addr, kglhdadr, kglhdpar, kglnaown, kglnaobj, kglnahsh, kglhdobj
  FROM x$kglob
 WHERE kglhdadr IN (SELECT p1raw
                      FROM v$session_wait
                     WHERE event LIKE 'library%')
/
ADDR     KGLHDADR KGLHDPAR KGLNAOWN   KGLNAOBJ                KGLNAHSH KGLHDOBJ
-------- -------- -------- ---------- -------------------- ---------- --------
404F2178 52D6730C 52D6730C SYS        PINING                2300250318 52D65BA4
```

（2）获得持有等待对象的 Session 信息。

```
SELECT a.SID, a.username, a.program, b.addr, b.kglpnadr, b.kglpnuse,
       b.kglpnses, b.kglpnhdl, b.kglpnlck, b.kglpnmod, b.kglpnreq
  FROM v$session a, x$kglpn b
 WHERE a.saddr = b.kglpnuse
   AND b.kglpnmod <> 0
   AND b.kglpnhdl IN (SELECT p1raw
                        FROM v$session_wait
                       WHERE event LIKE 'library%')
/
SQL>
       SID USERNAME   PROGRAM                                          ADDR     KGLPNADR
KGLPNUSE KGLPNSES KGLPNHDL KGLPNLCK   KGLPNMOD   KGLPNREQ
---------- ---------- ------------------------------------------------ -------- --------
-------- -------- -------- -------- ---------- ----------
        13 SYS         sqlplus@eygle.com (TNS V1-V3)          404F6CA4 52B2A518 51E2013C
51E2013C 52D6730C 52B294C8          2          0
```

（3）获得持有对象用户执行的代码：

```
SELECT sql_text
  FROM v$sqlarea
```

```
 WHERE (v$sqlarea.address, v$sqlarea.hash_value) IN (
          SELECT sql_address, sql_hash_value
            FROM v$session
           WHERE SID IN (
                    SELECT SID
                      FROM v$session a, x$kglpn b
                     WHERE a.saddr = b.kglpnuse
                       AND b.kglpnmod <> 0
                       AND b.kglpnhdl IN (SELECT p1raw
                                            FROM v$session_wait
                                           WHERE event LIKE 'library%')))
/
SQL_TEXT
--------------------------------------------------------------------------------
BEGIN calling; END;
```

在 grant 之前和之后，可以转储一下 Shared Pool 的内容观察比较一下：

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 32';
Session altered.
```

在 grant 之前，从前面的查询获得 pining 的 Handle 是 52D6730C：

```
**************************************************
BUCKET 67790:
  LIBRARY OBJECT HANDLE: handle=52d6730c
  name=SYS.PINING
  hash=891b08ce timestamp=09-06-2004 16:43:51
  namespace=TABL/PRCD/TYPE flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0011-0011 lock=N pin=S latch#=1
--在 Object 上存在共享 pin
--在 handle 上存在 Null 模式锁定,此模式允许其他用户继续以 Null/shared 模式锁定该对象
  lwt=0x52d67324[0x52d67324,0x52d67324] ltm=0x52d6732c[0x52d6732c,0x52d6732c]
  pwt=0x52d6733c[0x52b2a4e8,0x52b2a4e8] ptm=0x52d67394[0x52d67394,0x52d67394]
  ref=0x52d67314[0x52d67314, 0x52d67314] lnd=0x52d673a0[0x52d67040,0x52d6afcc]
    LIBRARY OBJECT: object=52d65ba4
    type=PRCD flags=EXS/LOC[0005] pflags=NST [01] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer status pins change    alloc(K)  size(K)
    ----- -------- -------- ------ ---- ------     -------- --------
        0 52d65dac 52d65c90 I/P/A     0 NONE        0.30     0.55
        4 52d65c40 52d67c08 I/P/A     1 NONE        0.44     0.48
```

在发出 grant 命令后：

```
**************************************************
BUCKET 67790:
  LIBRARY OBJECT HANDLE: handle=52d6730c
  name=SYS.PINING
  hash=891b08ce timestamp=09-06-2004 16:43:51
  namespace=TABL/PRCD/TYPE flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0011-0011 lock=X pin=S latch#=1
--由于 calling 执行未完成,在 object 上仍让保持共享 pin
--由于 grant 会导致重新编译该对象,所以在 handle 上的排他锁已经被持有
--进一步的需要获得 object 上的 Exclusive pin,由于 shared pin 被 calling 持有,所以
library cache pin 等待出现.
  lwt=0x52d67324[0x52d67324,0x52d67324] ltm=0x52d6732c[0x52d6732c,0x52d6732c]
  pwt=0x52d6733c[0x52b2a4e8,0x52b2a4e8] ptm=0x52d67394[0x52d67394,0x52d67394]
  ref=0x52d67314[0x52d67314, 0x52d67314] lnd=0x52d673a0[0x52d67040,0x52d6afcc]
    LIBRARY OBJECT: object=52d65ba4
    type=PRCD flags=EXS/LOC[0005] pflags=NST [01] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer status pins change    alloc(K)  size(K)
    ----- -------- -------- ------ ---- ------     -------- --------
        0 52d65dac 52d65c90 I/P/A     0 NONE       0.30     0.55
        4 52d65c40 52d67c08 I/P/A     1 NONE       0.44     0.48
```

实际上 recompile 过程包含以下步骤，同时来看一下 lock 和 pin 是如何交替发挥作用的。

- 存储过程的 library cache object 以排他模式被锁定，这个锁定是在 handle 上获得的。Exclusive 锁定可以防止其他用户执行同样的操作，同时防止其他用户创建新的引用此过程的对象。
- 以 Shared 模式 pin 该对象，以执行安全和错误检查。
- 共享 pin 被释放，重新以排他模式 pin 该对象，执行重编译。
- 使所有依赖该过程的对象失效。
- 释放 Exclusive Lock 和 Exclusive Pin。

2．LIBRARY CACHE LOCK 等待事件

如果此时再发出一条 grant 或 compile 的命令，那么 library cache lock 等待事件将会出现。

Session 3：

```
SQL> alter procedure pining compile;
```

此进程挂起，查询 v$session_wait 视图可以获得以下信息：

```
SQL> select * from v$session_wait;
 SID SEQ# EVENT               P1TEXT                  P1 P1RAW    P2TEXT               P2
P2RAW    P3TEXT              P3 P3RAW     WAIT_TIME SECONDS STATE
---- ---- ------------------- --------------- ---------- -------- ------------ ----------
```

```
-------- ---------------- -------------- ---------- ------  ---
  11  143 library cache pin   handle address  1390239716 52DD5FE4 pin address  1387617456
52B55CB0 100*mode+namespace 301 0000012D          0       6 WAITING
  13   18 library cache lock  handle address  1390239716 52DD5FE4 lock address 1387433984
52B29000 100*mode+namespace 301 0000012D          0       3 WAITING
   8  415 PL/SQL lock timer   duration        120000     0001D4C0                        0
00                            0 00               0      63 WAITING
……


13 rows selected
```

由于 handle 上的 lock 已经被 Session 2 以 exclusive 模式持有，所以 Session 3 产生了等待。可以看到，在生产数据库中权限的授予、对象的重新编译都可能会导致 library cache pin 等待的出现，所以应该尽量避免在高峰期进行以上操作。

另外，测试的案例本身就说明：如果 Package 或过程中存在复杂、交互的依赖关系极易导致 library cache pin 的出现，所以在应用开发的过程中，也应该注意这方面的内容。

3．Oracle 10g 的增强

从 Oracle 10g 开始，以上测试将不会看到同样的效果，这是因为 Oracle 10g 对于对象编译与重建做出了增强。注意以下测试（来自 Oracle10gR2 环境）：

```
SQL> SELECT OBJECT_NAME, LAST_DDL_TIME FROM USER_OBJECTS
  2  WHERE OBJECT_NAME='PINING';
OBJECT_NAME                    LAST_DDL_TIME
------------------------------ -------------------
PINING                         2006-07-04 14:15:34
SQL> create or replace PROCEDURE pining
  2    IS
  3    BEGIN
  4            NULL;
  5    END;
  6  /
Procedure created.
SQL> SELECT OBJECT_NAME, LAST_DDL_TIME FROM USER_OBJECTS
  2  WHERE OBJECT_NAME='PINING';
OBJECT_NAME                    LAST_DDL_TIME
------------------------------ -------------------
PINING                         2006-07-04 14:15:34
```

注意当重新 replace 一个过程时，Oracle 会首先执行检查，如果代码前后完全相同，则 replace 工作并不会真正进行（因为没有变化），对象的 LAST_DDL_TIME 不会改变，这就意味着 Latch 的竞争可以减少。

再来看一下此前的测试在 Oracle 10g 中的情况如何。首先在 Session 1 中执行：

```
SQL> create or replace PROCEDURE pining
  2 IS
  3 BEGIN
  4 NULL;
  5 END;
  6 /
Procedure created.
SQL> alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss';
Session altered.
SQL> create or replace procedure calling
 2 is
 3 begin
 4 pining;
 5 dbms_lock.sleep(60);
 6 end;
 7 /
Procedure created.
SQL> col object_name for a30
SQL> select object_name,last_ddl_time from dba_objects
  2  where object_name in ('PINING','CALLING');
OBJECT_NAME                    LAST_DDL_TIME
------------------------------ -------------------
CALLING                         2007-04-02 09:40:18
PINING                          2007-04-02 09:40:18
SQL> exec calling;
```

然后在 Session 2 执行授权：

```
SQL> set time on
09:40:22 SQL> grant execute on pining to sys;
Grant succeeded.
```

可以看到 Session 2 的授权顺利通过，再转到 Session 1：

```
SQL> exec calling;
PL/SQL procedure successfully completed.
SQL> select object_name,last_ddl_time from dba_objects
  2  where object_name in ('PINING','CALLING');
OBJECT_NAME                    LAST_DDL_TIME
------------------------------ -------------------
CALLING                         2007-04-02 09:40:18
PINING                          2007-04-02 09:40:22
```

注意到对象 pining 的 LAST_DDL_TIME 已经变化。grant 授权已经能够绕过了 library cache pin 的竞争，这是 Oracle 10g 的增强。

6.2.9 诊断案例一：version_count 过高造成的 Latch 竞争解决

本节是关于 Shared Pool 的一个诊断案例，案例本身可能并不重要，重要的是给大家一个解决问题的思路，并且通过这个案例可以进一步了解 Oracle 的工作原理。

问题起因是公司要进行短信群发，群发的时候每隔一段时间就会发生一次消息队列拥堵的情况。在数据库内部实际上是向一个数据表中记录发送日志，数据库版本是 Oracle 8.1.5。

在一个拥堵时段开始诊断，检查数据库端可能存在的问题，首先查询 V$SESSION_WAIT 视图，列出当前发生的等待：

```
SQL> select sid,event,p1,p1raw from v$session_wait;
       SID EVENT                                         P1 P1RAW
---------- ---------------------------------------------------------------
        76 latch free                            2147535824 8000CBD0
        83 latch free                            2147535824 8000CBD0
       148 latch free                            3415346832 CB920E90
       288 latch free                            2147535824 8000CBD0
       285 latch free                            2147535824 8000CBD0
       196 latch free                            2147535824 8000CBD0
       317 latch free                            2147535824 8000CBD0
         3 log file parallel write                      1 00000001
        13 log file sync                             2705 00000A91
        60 SQL*Net message to client              1413697536 54435000
       239 SQL*Net message to client              1413697536 54435000
...ignore some idle waiting here...
        11 SQL*Net message from client             675562835 28444553
        12 SQL*Net message from client            1413697536 54435000
170 rows selected.
```

在这次查询中，发现存在大量的 latch free 等待，再次查询时这些等待消失，应用也恢复了正常：

```
 SQL> select sid,event,p1,p1raw from v$session_wait where event not like 'SQL*Net%';
       SID EVENT                                    P1 P1RAW
---------- ---------------------------------------------------
         2 pmon timer                              300 0000012C
         1 rdbms ipc message                       300 0000012C
         4 rdbms ipc message                       300 0000012C
         6 rdbms ipc message                    180000 0002BF20
        18 rdbms ipc message                      6000 00001770
```

```
       102 rdbms ipc message                 6000 00001770
       178 rdbms ipc message                 6000 00001770
       194 rdbms ipc message                 6000 00001770
       311 rdbms ipc message                 6000 00001770
         3 log file parallel write              1 00000001
       148 log file sync                     2547 000009F3
       273 log file sync                     2544 000009F0
       190 log file sync                     2545 000009F1
         5 smon timer                         300 0000012C
14 rows selected.
```

接下来通过 V$LATCH 视图判断是哪些 Latch 竞争消耗了最多的等待时间：

```
SQL> select addr,latch#,name,gets,spin_gets from v$latch order by spin_gets;
ADDR         LATCH# NAME                                GETS  SPIN_GETS
-------- ---------- ---------------------------------------------------------------
80001398          3 session switching                    111937          0
80002010          6 longop free list                      37214          0
.....
80001330          2 session allocation                 261826230     428312
800063E0         64 multiblock read objects           1380614923    1366278
800026B8         11 messages                           207935758    1372606
80001218          0 latch wait list                    203479569    1445342
80006310         62 cache buffers chains              3.8472E+10    2521699
8000A17C         92 row cache objects                 1257586714    2555872
80007F80         74 redo writing                       264722932    4458044
80006700         67 cache buffers lru chain           5664313769   30046921
8000CBD0         98 shared pool                        122433688   59070585
8000CC38         99 library cache                     4414533796   1037032730

142 rows selected.
```

通过查询输出注意到，在当前数据库中竞争最严重的两个 Latch 是 Shared Pool 和 Library Cache。这两个 Latch 是 Shared Pool 管理中最重要也是最常见的 Latch 竞争。

Shared Pool Latch 用于共享池中内存空间的分配和回收，如果 SQL 没有充分共享，反复解析，那么将会不断请求 Shared Pool Latch 在共享池中分配空间，由此可能造成非常严重的 CPU 消耗，前面曾经论述过这个问题。

而 Library Cache Latch 用于保护 Cache 在内存中的 SQL 以及执行计划等，当需要向 Library Cache 中增加新的 SQL 时，Library Cache Latch 必须被获得。在解析 SQL 过程中，Oracle 需要搜索 Library Cache 查找匹配的 SQL，如果没有可共享的 SQL 代码，Oracle 将全新解析 SQL，获得 Library Cache Latch 向 Library Cache 中插入新的 SQL 代码。Library Cache Latch 的数量受一个隐含参数_kgl_latch_count 控制，其缺省值为大于或等于 CPU_COUNT 的最小素数，最大

值不能超过 67。

可以简化一下 SQL 的执行过程，以说明这两个 Latch 在 SQL 解析过程中所起的作用。

（1）首先需要获得 Library Cache Latch，根据 SQL 的 HASH_VALUE 值在 Library Cache 中寻找是否存在可共享代码。如果找到则为软解析，Server 进程获得该 SQL 执行计划，转向第（4）步；如果找不到共享代码则执行硬解析。

（2）释放 Library Cache Latch，获取 Shared Pool Latch，查找并锁定自由空间。

（3）释放 Shared Pool Latch，重新获得 Library Cache Latch，将 SQL 及执行计划插入到 Library Cache 中。

（4）释放 Library Cache Latch，保持 Null 模式的 Library Cache Pin/Lock。

（5）开始执行。

通过以上过程可以看到，如果系统中存在过度的硬解析，系统的性能必然受到反复解析、Latch 争用的折磨。通过查询 V$SYSSTAT 视图获得关于数据库解析的详细信息：

```
SQL> select name,value from v$sysstat where name like 'parse%';
NAME                              VALUE
------------------------------ ------
parse time cpu                    66
parse time elapsed                505
parse count (total)               801
parse count (hard)                193
parse count (failures)             0
```

通过（parse count (total)－parse count (hard)）/parse count (total)得出的**软解析率**经常被用作衡量数据库性能的一个重要指标。

现在回到我们这个问题上来，过多的 Shared Pool 和 library Cache 竞争，显然极有可能是 SQL 的过度解析造成的。进一步检查 v$sqlarea，来查找可能存在问题的 SQL 语句，发现了以下异常 SQL：

```
SQL> select sql_text,VERSION_COUNT,INVALIDATIONS,PARSE_CALLS,
  2  OPTIMIZER_MODE,PARSING_USER_ID PUI,PARSING_SCHEMA_ID PSI,ADDRESS,HASH_VALUE
  3  from v$sqlarea where version_count >1000;
SQL_TEXT
--------------------------------------------------------------------------------
VERSION_COUNT INVALIDATIONS PARSE_CALLS OPTIMIZER_MODE            PUI PSI ADDRESS   HASH_VALUE
------------- ------------- ----------- ------------------------ --- --- ------- -------
insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,MSGSTATUS,
AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,PLANID,FEETYPE,FEEV
ALUE,DATACODING,FLAGS,SMLEN,SMCONT) values :b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,
:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b21,:b22)
       7023             0         1596 MULTIPLE CHILDREN PRESENT   36   36 C82AF1C8 3974744754
```

这就是写日志记录的代码，这段代码使用了绑定变量，但是 version_count 却有 7023 个。也就是说这个 SQL 有 7023 个子指针，这是不可想象的。

通过前面几节的研究可以知道，如果这个 SQL 有 7023 个子指针，那么意味着这些子指针都将存在于同一个 Bucket 的链表上。这也就意味着，如果同样 SQL 再次执行，Oracle 将不得不搜索这个链表以寻找可以共享的 SQL。这将导致大量的 Library Cache Latch 的竞争。

应该注意数据库中 version_count 过多的 SQL 语句，version_count 过高通常会导致 Library Cache Latch 的长时间持有，从而影响性能，所以很多时候我们应该尽量避免这种情况的出现。

继续问题研究，这时候我们开始推测原因，可能的情况有：

- 可能代码存在问题，在每次执行之前程序修改某些 Session 参数，导致 SQL 不能共享；
- 可能是 Oracle 8i 的 v$sqlarea 记录存在问题，看到的结果是假象；
- ◆ 由于 Oracle 的 Bug 导致了 SQL 无法共享。

根据以上判断，继续诊断，最直接的可以将 Shared Pool 转储出来，看一看真实的内存结构：

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 4';
```

查看 trace 文件得到如下结果（摘录包含该段代码的片断），注意在 BUCKET 21049 上记录了这条 SQL 语句，而且的确存在 7023 个子指针：

```
BUCKET 21049:
  LIBRARY OBJECT HANDLE: handle=c82af1c8
  name=insert                                   into                                   sms_log
(MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,
MSGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICEC
ODE,PLANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,SMLEN,SMCONT)                values
(:b0,:b1,:b2,:b3,:b4,:b5,:b6,
:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b21,:b22
)
  hash=ece9cab2 timestamp=09-09-2004 12:51:29
  namespace=CRSR flags=RON/TIM/PN0/LRG/[10010001]
  kkkk-dddd-llll=0000-0001-0001 lock=N pin=S latch=5
  lwt=c82af1e0[c82af1e0,c82af1e0] ltm=c82af1e8[c82af1e8,c82af1e8]
  pwt=c82af1f8[c82af1f8,c82af1f8] ptm=c82af250[c82af250,c82af250]
  ref=c82af1d0[c82af1d0,c82af1d0]
    LIBRARY OBJECT: object=c1588e84
    type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0
    CHILDREN: size=7024
    child#    table reference   handle
    ------ -------- --------- --------
         0 c1589040  c1589008 c668c2bc
         1 c1589040  bfd179c4 c6ec9ee8
         2 c1589040  bfd179e0 c2dd9b3c
         3 c1589040  bfd179fc c5a46614
.....................................
```

```
.....ignore losts of child cursor here.....
.........................................
      7016 c641fcb8  c0ae4f2c c60196d0
      7017 c641fcb8  c0ae4f48 c4675d2c
      7018 c641fcb8  c0ae4f64 bd5e2750
      7019 c641fcb8  c0ae4f80 c09b1bb0
      7020 c641fcb8  c0ae4f9c bf2d6044
      7021 c641fcb8  c0ae4fb8 c332c1c4
      7022 c641fcb8  c0ae4fd4 cbdde0f8
    DATA BLOCKS:
    data#     heap  pointer status pins change
    ----- -------- -------- ------ ---- ------
        0 c3ef2c50 c1588f08 I/P/A     0 NONE
```

转储的数据与查询 V$SQL 得到的结果相同：

```
SQL> select CHILD_NUMBER,EXECUTIONS,OPTIMIZER_MODE OM,OPTIMIZER_COST OC,
   2 PARSING_USER_ID PUID,PARSING_SCHEMA_ID,ADDRESS,HASH_VALUE
   3 from v$sql where HASH_VALUE='3974744754';
CHILD_NUMBER EXECUTIONS OM             OC PUID PARSING_SCHEMA_ID ADDRESS  HASH_VALUE
------------ ---------- ---------- ------ --- ----------------- -------- ----------
           0      12966 CHOOSE     238150  36                36 C82AF1C8 3974744754
           1       7111 CHOOSE     238150  36                36 C82AF1C8 3974744754
.....

        7020        625 CHOOSE     237913  36                36 C82AF1C8 3974744754
        7021      10101 CHOOSE     237913  36                36 C82AF1C8 3974744754
        7022       7859 CHOOSE     237913  36                36 C82AF1C8 3974744754
7023 rows selected.
```

这里确实存在 7023 个子指针，第 2 种猜测被否定了，同时研发发过来的代码也不存在第 1 种情况。那么只能是第 3 种情况了，Oracle 的 Bug，搜索 Oracle 的官方支持站点 Metalink，发现 Bug 1210242。该 Bug 描述如下：

```
On certain SQL statements cursors are not shared when TIMED_STATISTICS is enabled.
```

碰巧这个数据库的 TIMED_STATISTICS 设置为 True。修改 TIMED_STATISTICS 为 False 以后，观察 v$sql，发现有效子指针很快下降到 2 个：

```
SQL> select CHILD_NUMBER,OPTIMIZER_COST,OPTIMIZER_MODE,EXECUTIONS,ADDRESS from v$sql where
hash_value=3974744754 and OPTIMIZER_MODE='CHOOSE';
CHILD_NUMBER OPTIMIZER_COST OPTIMIZER_ EXECUTIONS ADDRESS
------------ -------------- ---------- ---------- --------
           0         238167 CHOOSE          63943 C82AF1C8
           1         238300 CHOOSE          28915 C82AF1C8
```

第 2 天下降到只有 1 个：

```
SQL> select CHILD_NUMBER,OPTIMIZER_COST,OPTIMIZER_MODE,EXECUTIONS,ADDRESS from v$sql where
hash_value=3974744754 and OPTIMIZER_MODE='CHOOSE';
CHILD_NUMBER OPTIMIZER_COST OPTIMIZER_ EXECUTIONS ADDRESS
------------ -------------- ---------- ---------- --------
           0         238702 CHOOSE         578124 C82AF1C8
```

此时，相关业务恢复正常。

对于这个问题，另外一个可选的方法是设置一个隐含参数_sqlexec_progression_cost = 0，这个参数的具体含义为 SQL execution progression monitoring cost threshold，即 SQL 执行进度监控成本阀值。

这个参数根据 COST 来决定需要监控的 SQL，执行进度监控会引入额外的函数调用和 Row Sources，这可能导致 SQL 的执行计划或成本发生改变，从而产生不同的子指针。_sqlexec_progression_cost 的缺省值为 1000，成本大于 1000 的所有 SQL 都会被监控。如果该参数设置为 0,那么 SQL 的执行进度将不会被跟踪。

执行进度监控信息会被记录到 V$SESSION_LONGOPS 视图中，如果 time_statistics 参数设置为 False，那么这个信息就不会被记录。所以 time_statistics 参数和_sqlexec_progression_cost 是解决问题的两个途径。

通过查询也可以看到，在这个数据库中，OPTIMIZER_COST >1000 的 SQL 主要有以下 5 个：

```
SQL> select distinct(sql_text) from v$sql where OPTIMIZER_COST >1000;
SQL_TEXT
--------------------------------------------------------------------------------
insert into sms_detail_error (msgdate,addruser,msgid,areaid,reason,spnumber,msgt
ime,ifiddest,msqkey,servicecode,planid,feetype,feevalue,smcont,submittimes,submi
tdate,submittime,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept) valu
es (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:
b17,:b18,:b19,:b20)
insert into sms_detail_success (msgdate,addruser,msgid,areaid,spnumber,msgtime,i
fiddest,servicecode,planid,feetype,feevalue,smcont,submittimes,submitdate,submit
time,respdate,resptime,reptdate,repttime,msqkey) values (:b0,:b1,:b2,:b3,:b4,:b5
,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19)
insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,M
SGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,P
LANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,SMLEN,SMCONT) values (:b0,:b1,:b2,:b3,:b
4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b2
1,:b22)
insert into sms_resprept_error (msgdate,areaid,addruser,msgid,submittimes,submit
date,submittime,msgid_gw,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_re
pt,servicecode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12)
```

```
insert into sms_statusrept (reptdate,addruser,msgid_gw,repttime,statustype,msgid
_stus,msgstate,errorcode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7)
```

而这 5 个 SQL 中，在 v$sqlarea 中有 4 个 version_count 都在 10 以上：

```
SQL>  select sql_text,version_count from v$sqlarea where version_count>10;
SQL_TEXT
--------------------------------------------------------------------------------
VERSION_COUNT
-------------
insert into sms_detail_error (msgdate,addruser,msgid,areaid,reason,spnumber,msgt
ime,ifiddest,msqkey,servicecode,planid,feetype,feevalue,smcont,submittimes,submi
tdate,submittime,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept) valu
es (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:
b17,:b18,:b19,:b20)
           42
insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,M
SGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,P
LANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,SMLEN,SMCONT) values (:b0,:b1,:b2,:b3,:b
4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b2
1,:b22)
         7026
insert into sms_resprept_error (msgdate,areaid,addruser,msgid,submittimes,submit
date,submittime,msgid_gw,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_re
pt,servicecode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12)
          301
insert into sms_statusrept (reptdate,addruser,msgid_gw,repttime,statustype,msgid
_stus,msgstate,errorcode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7)
           41
```

具体可以参考 Metalink: Note 62143，至此这个关于 Shared Pool 的问题找到了原因，并得以及时解决。

最近在 Oracle 9i 中遇到过另外一个类似案例，同样是 Library Cache Latch 和 Shared Pool Latch 竞争，类似的在 v$sqlarea 中发现大量高 version_count 的 SQL。

对于 version_count 过高的问题，可以查询 V$SQL_SHARED_CURSOR 视图，这个视图会给出 SQL 不能共享的具体原因，如果是正常因素导致的，相应的字段会被标记为“Y”；对于异常的情况（如本案例），查询结果可能显示的都是“N”，这就表明 Oracle 认为这种行为是正常的，在当前系统设置下，这些 SQL 不应该被共享，那么可以判断是某个参数设置引起的。和 SQL 共享关系最大的一个初始化参数就是 cursor_sharing，在这个案例中 cursor_sharing 参数被设置为 similar，正是这个设置导致了大量子指针不能共享。

搜索 Metalink 可以获得相关说明，当 cursor_sharing 参数设置为 similar，并且数据库存在

相关柱状图（histograms）信息时，对于每一条新执行的 SQL，Oracle 都通过硬解析以获得更为精确的执行计划，这最终导致了 version_count 过高，这是 cursor_sharing = similar 的正常行为，并非 Bug。

了解了这个行为之后，解决这个问题也就并不复杂了，可以将 cursor_sharing 设置为 Exact 或者 Force 以避免此问题，或者通过删除柱状图信息（Histograms）来防止不必要的硬解析，实际上，如果数据不存在失衡分布，我们也不必要收集柱状图信息。

这两个案例给我们的另外一个启示就是，**当需要设置某些特殊的参数来影响数据库的行为时，必须了解这些设置会给数据库带来的影响**，这样一方面可以避免问题的出现，另一方面在问题出现时，我们可以快速地发现问题根源并解决问题。

6.2.10 V$SQL 与 V$SQLAREA 视图

在前面提到过一个经常被问及的问题：V$SQL 与 V$SQLAREA 两个视图有什么不同？所以有这样一个问题是因为这两个视图在结构上非常相似。

上文提到，**V$SQLAREA 和 V$SQL 两个视图的不同之处在于，V$SQL 中为每一条 SQL 保留一个条目，而 V$SQLAREA 中根据 SQL_TEXT 进行 GROUP BY，通过 version_count 计算子指针的个数。**下面对这个问题进行一点延伸探讨。

1. V$SQL 视图与 V$SQLAREA 视图

首先介绍一下 V$SQL 视图，V$SQL 视图列举了共享 SQL 区（Shared SQL Area）中的 SQL 统计信息，这个视图中的信息未经分组，每个 SQL 指针都包含一条独立的记录。这个视图的主要字段如表 6-2 所示。

表 6-2　V$SQL 视图的主要字段

Column	Datatype	Description
SQL_TEXT	VARCHAR2(1000)	当前 SQL 指针的前 1000 个字符（也就是说这里记录的 SQL 是不完整的）
EXECUTIONS	NUMBER	执行次数
DISK_READS	NUMBER	这个子指针 Disk Read 的次数
BUFFER_GETS	NUMBER	这个子指针的 buffer Gets 数量
OPTIMIZER_MODE	VARCHAR2(10)	SQL 执行的优化器模式
OPTIMIZER_COST	NUMBER	SQL 执行成本
HASH_VALUE	NUMBER	在 Library Cache 中父指针的 Hash Value 值

用前文应用的例子进行进一步说明，假定数据库中存在一个用户 EYGLE，用户下存在一张 EMP 表（以下测试来自 Oracle 9i 9.2.0.4 数据库环境）：

```
SQL> create table emp as select * from scott.emp;
Table created.
```

进行一个查询测试，通过 Autotrace 观察一下执行计划和统计数据：

```
SQL> set autotrace on
```

```
SQL> select count(*) from emp;
  COUNT(*)
----------
        14
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'EMP'
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          1  physical reads
```

这个查询的统计信息显示，执行了 1 个物理读，3 个 Consistent Gets，来看一下 v$sql 中记录的统计数据：

```
SQL> select sql_text,executions,disk_reads,optimizer_mode,buffer_gets,hash_value
  2  from v$sql where sql_text='select count(*) from emp';
SQL_TEXT                  EXECUTIONS DISK_READS OPTIMIZER_ BUFFER_GETS HASH_VALUE
------------------------- ---------- ---------- ---------- ----------- ----------
select count(*) from emp           1          1 CHOOSE               3 4085390015
```

记录的信息和 AUTOTRACE 显示的信息完全一致。在第一次执行时，这个 SQL 的 HASH_VALUE 被计算出来为 4085390015，并且随之，这个 SQL 的父指针（Parent Cursor）在内存中被创建，一个子指针同时创建。父指针可以被认为是 Hash Value 的相关信息，子指针可以被认为是 SQL 的元数据。

再次执行这个查询，统计信息中物理读（DISK_READS）不再增加，因为数据已经在 Buffer 中存在，而 BUFFER_GETS 继续增加。执行次数也变为 2 次：

```
SQL> select count(*) from emp;
  COUNT(*)
----------
        14
SQL> select sql_text,executions,disk_reads,optimizer_mode, buffer_gets,hash_value
  2  from v$sql where sql_text='select count(*) from emp';
SQL_TEXT                  EXECUTIONS DISK_READS OPTIMIZER_ BUFFER_GETS HASH_VALUE
------------------------- ---------- ---------- ---------- ----------- ----------
select count(*) from emp           2          1 CHOOSE               6 4085390015
```

V$SQLAREA 视图也是非常重要的一个视图，在 Oracle 9iR2 的文档中，Oracle 这样定义这个视图：V$SQLAREA 列出了共享 SQL 区（Shared SQL Area）中的 SQL 统计信息，这些

SQL 按照 SQL 文本的不同，每条会记录一行统计数据。注意这里所说的是“按照 SQL 文本”来进行区分，也就是说这个视图的信息可以看作是根据 SQL_TEXT 进行的一次汇总统计。

V$SQLAREA 视图的主要字段如图 6-3 所示。

表 6-3　V$SQLAREA 视图的主要字段

Column	Datatype	Description
SQL_TEXT	VARCHAR2(1000)	当前指针的前 1000 个字符
VERSION_COUNT	NUMBER	Cache 中这个父指针下存在的子指针的数量
EXECUTIONS	NUMBER	总的执行次数，包含所有子指针执行次数的汇总
DISK_READS	NUMBER	所有子指针的 Disk Reads 总和
BUFFER_GETS	NUMBER	所有子指针 Buffer Gets 的总和
OPTIMIZER_MODE	VARCHAR2(10)	SQL 执行的优化器模式
HASH_VALUE	NUMBER	父指针的 Hash value

通过前文可以知道，文本完全相同的 SQL 语句，在数据库中的意义可能完全不同。比如数据库中存在两个用户 EYGLE 和 JULIA，两个用户各拥有一张数据表 EMP。

那么当两个用户发出一个查询 select count(*) from emp 时，这个查询访问的对象，返回的结果可能完全不同，EYGLE 的查询访问的是 EYGLE.EMP 表，而 JULIA 用户访问的则是 JULIA.EMP 表。但是单从 SQL_TEXT 上来说，这两个 SQL 没有任何区别。

继续前面的测试，再来简单看一下以下的输出：

```
SQL> connect julia/julia
Connected.
SQL> create table emp as select * from scott.emp where rownum <9;
Table created.
SQL> select count(*) from emp;
  COUNT(*)
----------
         8
```

现在 V$SQL 中应该有了两条完全一样的 SQL，但是各自查询的物理对象却是截然不同：

```
SQL> select sql_text,executions,disk_reads,optimizer_mode,buffer_gets,hash_value
  2  from v$sql where sql_text='select count(*) from emp';
SQL_TEXT                  EXECUTIONS DISK_READS OPTIMIZER_ BUFFER_GETS HASH_VALUE
------------------------- ---------- ---------- ---------- ----------- ----------
select count(*) from emp           2          1 CHOOSE               6 4085390015
select count(*) from emp           1          1 CHOOSE               3 4085390015
```

现在再来查询 v$sqlarea 视图，就可以看到着两个视图的不同：

```
SQL> select sql_text,executions,disk_reads,buffer_gets,hash_value,version_count
  2  from v$sqlarea where sql_text='select count(*) from emp';
SQL_TEXT                  EXECUTIONS DISK_READS BUFFER_GETS HASH_VALUE VERSION_COUNT
------------------------- ---------- ---------- ----------- ---------- -------------
```

```
select count(*) from emp          3          2          9 4085390015             2
```

在这个视图中，Oracle 将 v$sql 中 sql_text 相同的 2 个子指针合并起来，执行次数等信息也都进行了累计，version_count 也显示为 2，这就是 v$sqlarea 的聚合作用。

2．v$sql 视图和 v$sqlarea 的构建

通过 v$fixed_view_definition 视图，可以查询得到 v$sql 视图和 v$sqlarea 视图的构建语句：

```
select view_definition from v$fixed_view_definition where view_name='GV$SQL';
select view_definition from v$fixed_view_definition where view_name='GV$SQLAREA';
```

GV$SQL 的定义结构如下：

```
SELECT inst_id, kglnaobj,
       kglobhs0 + kglobhs1 + kglobhs2 + kglobhs3 + kglobhs4 + kglobhs5 + kglobhs6 + kglobt16,
       kglobt08 + kglobt11, kglobt10, kglobt01, DECODE (kglobhs6, 0, 0, 1),
       DECODE (kglhdlmd, 0, 0, 1), kglhdlkc, kglobt04, kglobt05, kglobpc6,
       kglhdldc, SUBSTR (TO_CHAR (kglnatim, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19),
       kglhdivc, kglobt12, kglobt13, kglobt14, kglobt15, kglobt02,
       DECODE (kglobt32, 0, 'NONE', 1, 'ALL_ROWS', 2, 'FIRST_ROWS', 3, 'RULE', 4, 'CHOOSE',
               'UNKNOWN'),
       kglobtn0, kglobt17, kglobt18, kglhdkmk, kglhdpar, kglobtp0, kglnahsh,
       kglobt30, kglobt09, kglobts0, kglobt19, kglobts1, kglobt20, kglobt21,
       kglobts2, kglobt06, kglobt07, kglobt28, kglhdadr, kglobt29,
       DECODE (BITAND (kglobt00, 64), 64, 'Y', 'N'),
       DECODE (kglobsta, 1, 'VALID', 2, 'VALID_AUTH_ERROR', 3, 'VALID_COMPILE_ERROR',
               4, 'VALID_UNAUTH', 5, 'INVALID_UNAUTH', 6, 'INVALID'),
       kglobt31, SUBSTR (TO_CHAR (kglobtt0, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19),
       DECODE (kglobt33, 1, 'Y', 'N'), kglhdclt
  FROM x$kglcursor
 WHERE kglhdadr != kglhdpar AND kglobt02 != 0
```

而 GV$SQLAREA 的视图结构如下：

```
SELECT   inst_id, kglnaobj,
        SUM (kglobhs0 + kglobhs1 + kglobhs2 + kglobhs3 + kglobhs4 + kglobhs5 + kglobhs6),
        SUM (kglobt08 + kglobt11), SUM (kglobt10), SUM (kglobt01),
        COUNT (*) - 1, SUM (DECODE (kglobhs6, 0, 0, 1)),
        DECODE (SUM (DECODE (kglhdlmd, 0, 0, 1)),0, 0, SUM (DECODE (kglhdlmd, 0, 0, 1)) - 1 ),
      SUM (kglhdlkc) / 2, SUM (kglobt04), SUM (kglobt05), SUM (kglobpc6),SUM (kglhdldc) - 1,
        SUBSTR (TO_CHAR (kglnatim, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19),
        SUM (kglhdivc), SUM (kglobt12), SUM (kglobt13), SUM (kglobt14),
        SUM (kglobt15), SUM (DECODE (kglobt09, 0, kglobt02, 0)),
        DECODE (COUNT (*) - 1, 1, DECODE (SUM (DECODE (kglobt09, 0, kglobt32, 0)),
                               0, 'NONE', 1, 'ALL_ROWS', 2, 'FIRST_ROWS', 3, 'RULE',
```

```
                          4, 'CHOOSE','UNKNOWN'), 'MULTIPLE CHILDREN PRESENT'),
       SUM (DECODE (kglobt09, 0, kglobt17, 0)),SUM (DECODE (kglobt09, 0, kglobt18, 0)),
       DECODE (SUM (DECODE (kglhdkmk, 0, 0, 1)), 0, 0,SUM (DECODE (kglhdkmk, 0, 0, 1)) - 1 ),
       kglhdpar, kglnahsh, kglobts0, kglobt19, kglobts1, kglobt20,
      SUM (kglobt21), SUM (kglobt06), SUM (kglobt07),DECODE (kglobt33, 1, 'Y', 'N'), kglhdclt
   FROM x$kglcursor
GROUP                                                                    BY
inst_id,kglnaobj,kglhdpar,kglnahsh,kglnatim,kglobts0,kglobt19,kglobts1,
        kglobt20,DECODE (kglobt33, 1, 'Y', 'N'),kglhdclt
  HAVING SUM (DECODE (kglobt09, 0, kglobt02, 0)) != 0
```

这两个视图都是来自底层表 x$kglcursor，而 v$sqlarea 较 v$sql 视图多出了一个主要的 Group By 子句，两个视图显示的数据由此不同。

X$KGLCURSOR 是数据库的一个内部表，具体的含义为[K]ernel [G]eneric [L]ibrary Cache Manager [CURSOR]s，记录的也就是 Library Cache 中记录的 Cursor。

更进一步地，可以看一下查询 v$sql 已经 v$sqlarea 的执行计划：

```
SQL> set autotrace trace explain
SQL> select count(*) from v$sql;
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     FIXED TABLE (FULL) OF 'X$KGLOB'
SQL> select count(*) from v$sqlarea;
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     VIEW OF 'GV$SQLAREA'
   3    2       FILTER
   4    3         SORT (GROUP BY)
   5    4           FIXED TABLE (FULL) OF 'X$KGLOB'
```

通过执行计划可以看到 Group By 在 v$sqlarea 视图查询中所起的作用，不过同前面的视图定义有一点差别的是，执行计划显示 v$sql 和 v$sqlarea 是构建在 x$kglob 视图上，而不是 x$kglcursor。如果观察一下 x$kglob 和 x$kglcursor 表的结构完全相同。X$KGLOB 代表[K]ernel [G]eneric [L]ibrary Cache Manager [OB]ject 。实际上，x$kglcursor 是构建于 x$kglob 之上的又一个视图，当进行查询时，执行计划再次解析为底层视图。

按照以上的讨论，同样的 SQL，存在多个不同子指针的情况应该较为常见，在以下的一个来自生产系统的查询中，两个 SQL 指针拥有 35 个子指针：

```
SQL> select substr(sql_text,1,40) sql_text,hash_value,buffer_gets,executions,version_count
```

```
  2  from v$sqlarea where version_count >30;
SQL_TEXT                                 HASH_VALUE BUFFER_GETS EXECUTIONS VERSION_COUNT
---------------------------------------- ---------- ----------- ---------- -------------
update HS_passport set image=:1, gender= 3155639030      570650     124787            35
insert into HS_passport (image, gender,  1887468591      503082      56328            35
```

那么在 v$sql 中，应该存在 35 个完全独立的子指针：

```
SQL> select substr(sql_text,1,40) sql_text,hash_value,buffer_gets,
  2  executions,parsing_user_id from v$sql where hash_value=1887468591;
SQL_TEXT                                 HASH_VALUE BUFFER_GETS EXECUTIONS PARSING_USER_ID
---------------------------------------- ---------- ----------- ---------- ---------------
insert into HS_passport (image, gender,  1887468591      432098      52313              63
insert into HS_passport (image, gender,  1887468591           0          0               0
insert into HS_passport (image, gender,  1887468591           0          0               0
……
insert into HS_passport (image, gender,  1887468591           0          0               0
insert into HS_passport (image, gender,  1887468591        6638        843              51
insert into HS_passport (image, gender,  1887468591        3795        509             136
insert into HS_passport (image, gender,  1887468591           0          0               0
insert into HS_passport (image, gender,  1887468591         181         25              26
insert into HS_passport (image, gender,  1887468591       59532       2627              60
insert into HS_passport (image, gender,  1887468591         110          4             121
insert into HS_passport (image, gender,  1887468591           0          0               0
insert into HS_passport (image, gender,  1887468591          26          2              59
insert into HS_passport (image, gender,  1887468591         186          2              23
insert into HS_passport (image, gender,  1887468591         524          4             129
35 rows selected.
```

注意到部分 SQL 并未真正执行，而真正执行的 SQL 是由不同的用户发起的。

6.2.11 Oracle 10g 中 version_count 过高的诊断

理解了 V$SQL 与 V$SQLAREA 视图的区别，接下来来分析一个 Oracle10g 中 VERSION_COUNT 过高的案例。以下是网友提出的一个问题，他的一条 SQL 的 version_count 很高，达到了 1000 左右，数据库版本是 10.2.0.3。

表 6-4 是来自 AWR 诊断报告中的一段数据。

表 6-4　AWR 诊断报告分析

Version Count	Executions	SQL Id	SQL Module	SQL Text
1,000	138	**c86jftrtw542z**	w3wp.exe	insert into S_JOBSTMP (JOB...
345	42	**4cynkyw1jdd87**	w3wp.exe	Update S_Company Set CName=:C...
239	66	**7w27m78ucba8t**	w3wp.exe	insert into S_JOBSTMP (JOB...

188	146	**6pqxgscpt16n8**	w3wp.exe	update S_JOBS set CTMID=:CTM...
164	18	**djuyzucd13wyn**	w3wp.exe	update S_JOBSTMP set CTMID=:...
125	1,670	**cb3pz3mxcmcb1**	w3wp.exe	INSERT INTO S_RUBBISH (MySeqID...
59	10	**dcvns3atvwnk5**	w3wp.exe	INSERT INTO S_SearchEngine(ID,...
52	13	**9thwhd2qu60hz**	w3wp.exe	UPDATE S_SearchEngine SET CT...
42	77	**0q03ugc3nmydq**	w3wp.exe	INSERT INTO S_CompanyHr (MySeq...
29	4,368	**3233uytt5g2ur**	w3wp.exe	select reFreshdate, SetReFr...

在查询 V$SQL 视图时，却发现实际上并没有那么多子指针存在：

```
SQL>  select CHILD_NUMBER,EXECUTIONS,OPTIMIZER_MODE,PARSING_USER_ID,ADDRESS,HASH_VALUE
  2  from v$sql where sql_id='4cynkyw1jdd87';
CHILD_NUMBER EXECUTIONS OPTIMIZER_ PARSING_USER_ID  ADDRESS          HASH_VALUE
------------ ---------- ---------- ---------------  ---------------- ----------
         332          1 ALL_ROWS                69  C0000000E6615060   51819783
         333         23 ALL_ROWS                69  C0000000E6615060   51819783
         334         44 ALL_ROWS                69  C0000000E6615060   51819783
         335          2 ALL_ROWS                67  C0000000E6615060   51819783
         337         10 ALL_ROWS                67  C0000000E6615060   51819783
         338          1 ALL_ROWS                69  C0000000E6615060   51819783
         344         20 ALL_ROWS                68  C0000000E6615060   51819783
```

于是一个问题被提出来，为什么 version_count 的数量会大于子指针的数量呢？是不是 Bug 呢？接到这个问题后，我开始思考。首先我不认为是 Bug，因为根据前面的讨论，version_count 的数据来自 v$sqlarea 视图，v$sqlarea 视图和 v$sql 视图是同源的，只不过增加了一个 Group By 的分组。如果说这两者之间存在了差异，那么有可能是 Oracle 10g 中的视图定义发生了改变。

在 Oracle 10g 环境中执行一下查询，马上发现了改变，现在 v$sqlarea 视图来自 X$KGLCURSOR_CHILD_SQLID 底层表 ：

```
SQL> select count(*) from v$sqlarea;
Execution Plan
----------------------------------------------------------
Plan hash value: 1686081275
-------------------------------------------------------------------------------------------
| Id  | Operation         | Name                    |Rows |Bytes|Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                         |    1|   26|    0   (0)| 00:00:01|
|   1 |  SORT AGGREGATE   |                         |    1|   26|           |          |
|*  2 |   FIXED TABLE FULL| X$KGLCURSOR_CHILD_SQLID |    1|   26|    0   (0)| 00:00:01|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("KGLOBTO2"<>0 AND "INST_ID"=USERENV('INSTANCE'))
```

而 v$sql 视图则来自 X$KGLCURSOR_CHILD：

```
SQL> select count(*) from v$sql;
Execution Plan
----------------------------------------------------------
Plan hash value: 2618101788
-------------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |    13 |     0   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |                   |     1 |    13 |            |          |
|*  2 |   FIXED TABLE FULL| X$KGLCURSOR_CHILD |     1 |    13 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("INST_ID"=USERENV('INSTANCE'))
```

通过 v$fixed_view_definition 视图获得着两个视图的定义语句。v$sql 视图的定义如下：

```
SELECT inst_id, kglnaobj, kglfnobj, kglobt03,
      kglobhs0 + kglobhs1 + kglobhs2 + kglobhs3 + kglobhs4 + kglobhs5 + kglobhs6 + kglobt16,
       kglobt08 + kglobt11, kglobt10, kglobt01, DECODE (kglobhs6, 0, 0, 1),
       DECODE (kglhdlmd, 0, 0, 1), kglhdlkc, kglobt04, kglobt05, kglobt48,
       kglobt35, kglobpc6, kglhdldc,
       SUBSTR (TO_CHAR (kglnatim, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19), kglhdivc,
       kglobt12, kglobt13, kglobwdw, kglobt14, kglobwap, kglobwcc, kglobwcl,
       kglobwui, kglobt42, kglobt43, kglobt15, kglobt02,
       DECODE (kglobt32, 0, 'NONE', 1, 'ALL_ROWS', 2, 'FIRST_ROWS', 3, 'RULE',
               4, 'CHOOSE', 'UNKNOWN'),
       kglobtn0, kglobcce, kglobcceh, kglobt17, kglobt18, kglobts4, kglhdkmk,
       kglhdpar, kglobtp0, kglnahsh, kglobt46, kglobt30, kglobt09, kglobts5,
       kglobt48, kglobts0, kglobt19, kglobts1, kglobt20, kglobt21, kglobts2,
       kglobt06, kglobt07, DECODE (kglobt28, 0, TO_NUMBER (NULL), kglobt28),
       kglhdadr, kglobt29, DECODE (BITAND (kglobt00, 64), 64, 'Y', 'N'),
       DECODE (kglobsta, 1, 'VALID',2, 'VALID_AUTH_ERROR',3, 'VALID_COMPILE_ERROR',
               4, 'VALID_UNAUTH', 5, 'INVALID_UNAUTH', 6, 'INVALID' ),
       kglobt31, SUBSTR (TO_CHAR (kglobtt0, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19),
       DECODE (kglobt33, 1, 'Y', 'N'), kglhdclt, kglobts3, kglobt44, kglobt45,
       kglobt47, kglobt49, kglobcla, kglobcbca
  FROM x$kglcursor_child
```

而 v$sqlarea 视图的定义为：

```
SELECT inst_id, kglnaobj, kglfnobj, kglobt03,
       kglobhs0 + kglobhs1 + kglobhs2 + kglobhs3 + kglobhs4 + kglobhs5  + kglobhs6,
```

```
      kglobt08 + kglobt11, kglobt10, kglobt01, kglobccc, kglobclc, kglhdlmd,
      kglhdlkc, kglobt04, kglobt05, kglobt48, kglobt35, kglobpc6, kglhdldc,
      SUBSTR (TO_CHAR (kglnatim, 'YYYY-MM-DD/HH24:MI:SS'), 1, 19), kglhdivc,
      kglobt12, kglobt13, kglobwdw, kglobt14, kglobwap, kglobwcc, kglobwcl,
      kglobwui, kglobt42, kglobt43, kglobt15, kglobt02,
      DECODE (kglobt32, 0, 'NONE', 1, 'ALL_ROWS', 2, 'FIRST_ROWS', 3, 'RULE',
              4, 'CHOOSE', 'UNKNOWN'),
      kglobtn0, kglobcce, kglobcceh, kglobt17, kglobt18, kglobts4, kglhdkmk,
      kglhdpar, kglnahsh, kglobt46, kglobt30, kglobts0, kglobt19, kglobts1,
      kglobt20, kglobt21, kglobts2, kglobt06, kglobt07,
      DECODE (kglobt28, 0, NULL, kglobt28), kglhdadr,
      DECODE (BITAND (kglobt00, 64), 64, 'Y', 'N'),
      DECODE (kglobsta,1, 'VALID',2, 'VALID_AUTH_ERROR',3, 'VALID_COMPILE_ERROR',
              4, 'VALID_UNAUTH',5, 'INVALID_UNAUTH',6, 'INVALID'),
      kglobt31, kglobtt0, DECODE (kglobt33, 1, 'Y', 'N'), kglhdclt, kglobts3,
      kglobt44, kglobt45, kglobt47, kglobt49, kglobcla, kglobcbca
 FROM x$kglcursor_child_sqlid
WHERE kglobt02 != 0
```

现在这两个视图来自两个独立的底层 X$表，v$sqlarea 视图也不再包含 Group By 子句，这使得以前版本中查询 v$sqlarea 的性能问题得以缓解。那么 version_count 和 v$sql 的差异应该就来自底层 x$表的变更，在 V$SQLAREA 中增加了一个“!=0”的条件过滤（kglobt02 对应视图中的 COMMAND_TYPE 定义）。

于是我请网友进行以下查询：

```
SQL> select hash_value,version_count from v$sqlarea where version_count > 300;
HASH_VALUE VERSION_COUNT
---------- -------------
4089614431          1014
  51819783           359
```

再查询 x$kglob 表：

```
SQL> select count(*)  from x$kglob where KGLNAHSH=4089614431;
  COUNT(*)
----------
      1015
```

能够发现，所有的子指针在 x$kglob 表中全部存在，也就是说，观察到的变化是由于底层表的变更导致不同过滤算法导致的。在我的一个产品环境中，这样的情况也可以看到：

```
SQL> select a.sql_id,a.version_count,a.hash_value,count(*)
  2  from v$sqlarea a,v$sql b
  3  where a.version_count >100 and a.hash_value=b.hash_value
  4  group by a.sql_id,a.version_count,a.hash_value
```

```
  5  order by 4;
SQL_ID        VERSION_COUNT HASH_VALUE   COUNT(*)
------------- ------------- ---------- ----------
4qu48ka4pr6mh           103 2304481904         26
2n54c9mrg0amk           126 4008716914         30
fvp6ty9ass9w4           102 1435248516         99
b5yyh2vc9g8tm           102 3633816371        102
31a13pnjps7j3           102  593239587        102
gvynt9bqh451z           102 3976336447        102
0vwa9n600yvgm           103 2148494835        103
5an8d9ctcysja           106  852451882        106
bdv0rkkssq2jm           107 2978679347        107
fy9ta5zgkwqmp           107 3744356981        107
10 rows selected.
```

以其中一个 SQL 为例进行进一步分析：

```
SQL> SELECT   sql_id, hash_value, buffer_gets, executions, parsing_user_id
  2      FROM v$sql
  3     WHERE hash_value = 4008716914
4  ORDER BY buffer_gets DESC;
SQL_ID        HASH_VALUE BUFFER_GETS EXECUTIONS PARSING_USER_ID
------------- ---------- ----------- ---------- ---------------
2n54c9mrg0amk 4008716914       17013       5517              55
2n54c9mrg0amk 4008716914        4030        887              55
2n54c9mrg0amk 4008716914        3935        560              55
2n54c9mrg0amk 4008716914        3783        876              55
2n54c9mrg0amk 4008716914        3341        474              55
2n54c9mrg0amk 4008716914        2831        626              55
2n54c9mrg0amk 4008716914        2610        372              55
2n54c9mrg0amk 4008716914        2597        361              55
2n54c9mrg0amk 4008716914        1857        265              55
2n54c9mrg0amk 4008716914        1844        430              55
2n54c9mrg0amk 4008716914        1587        226              55
2n54c9mrg0amk 4008716914        1423        203              55
2n54c9mrg0amk 4008716914        1244        313              55
2n54c9mrg0amk 4008716914        1115        254              55
2n54c9mrg0amk 4008716914         975        139              55
2n54c9mrg0amk 4008716914         920        131              55
2n54c9mrg0amk 4008716914         831        176              55
2n54c9mrg0amk 4008716914         825        117              55
```

```
2n54c9mrg0amk 4008716914         737        173              55
2n54c9mrg0amk 4008716914         717        162              55
2n54c9mrg0amk 4008716914         694         99              55
2n54c9mrg0amk 4008716914         512         73              55
2n54c9mrg0amk 4008716914         498         71              55
2n54c9mrg0amk 4008716914         364         52              55
2n54c9mrg0amk 4008716914         266         38              55
2n54c9mrg0amk 4008716914         161         23              55
2n54c9mrg0amk 4008716914          42          6              55
2n54c9mrg0amk 4008716914          28          4              55
2n54c9mrg0amk 4008716914          21          3              55
2n54c9mrg0amk 4008716914           7          1              55
30 rows selected.
```

注意到，所有的 SQL 都是执行过的。而从 x$kglob 中查询得到的 SQL 或者 Oracle 9i 的 v$sql 视图中查询得到的 SQL 包含未执行或者 buffer_gets 为 0 的 SQL 指针，这部分在 Oracle 10g 中被从 v$sqlarea 中过滤了出去：

```
SQL> SELECT kglobt03, kglnahsh, kglobt14, kglhdexc, kglobt17
  2    FROM x$kglob
  3   WHERE kglnahsh = 4008716914 order by kglobt14 desc,4;
SQL_ID        HASH_VALUE BUFFER_GETS EXECUTIONS PARSING_USER_ID
------------- ---------- ----------- ---------- ---------------
°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°
2n54c9mrg0amk 4008716914          7          1              55
2n54c9mrg0amk 4008716914          0          0      2147483644
2n54c9mrg0amk 4008716914          0          1               0
2n54c9mrg0amk 4008716914          0          1               0
2n54c9mrg0amk 4008716914          0          1               0
……
2n54c9mrg0amk 4008716914          0        232               0
2n54c9mrg0amk 4008716914          0        274               0
2n54c9mrg0amk 4008716914          0        289               0
2n54c9mrg0amk 4008716914          0        320               0
2n54c9mrg0amk 4008716914          0        341               0
2n54c9mrg0amk 4008716914          0        399               0
2n54c9mrg0amk 4008716914          0        420               0
2n54c9mrg0amk 4008716914          0        461               0
2n54c9mrg0amk 4008716914          0       1305               0
127 rows selected.
```

6.2.12 诊断案例二：临时表引发的竞争

这是帮助一个网友解决的一个问题，通过 MSN 交流，以下是问题的解决过程及思路，供大家参考。

问：如果一个 DB 里面的几个存储过程总是跑不完，同样的存储过程在其他的 6 个省都很正常，数据库里没有锁，数据库和 Server 上面的空间足够。正常的情况几分钟就能运行完，现在都 n 多小时了还没有运行完，会是什么原因呢?

答：检查 v$session_wait，看系统在等什么？

> 提示：如果你的系统慢，通常是存在等待，v$session_wait 是你应该优先检查的视图。

下面是这位网友发过来的查询结果，这里截取了主要的部分：

```
SID   SEQ#   EVENT                P1TEXT          P1         P1RAW      P2TEXT          P2
13    7210   library cache pin    handle address  3172526924 BD18EB4C   pin address 3205742908
33    16179 library cache pin     handle address  3172526924 BD18EB4C   pin address 3206485860
32    14721 library cache pin     handle address  3172526924 BD18EB4C   pin address 3206555324
27    54913 library cache pin     handle address  3172526924 BD18EB4C   pin address 3205741540
30    16169 library cache lock    handle address  3174604528 BD389EF0   lock address3206478252
```

通过以上信息注意到，数据库目前正在经历 Library Cache Pin 和 Library Cache Lock 的等待和竞争。我要求她执行本章上文中讲到过的 SQL，并提供结果：

```
SQL> select ADDR,KGLHDADR,KGLHDPAR,KGLNAOWN,KGLNAOBJ,KGLNAHSH,KGLHDOBJ
  2  from X$KGLOB
  3  where KGLHDADR ='BD18EB4C';
ADDR     KGLHDADR KGLHDPAR  KGLNAOWN  KGLNAOBJ  KGLNAHSH KGLHDOBJ
---------- ---------------- ------------------------- ------------------------- ------
01920880   BD18EB4C  BD18EB4C         truncate table iptt_pm_all   653109544 BD18E8D4
SQL> SELECT a.SID, a.username, a.program, b.addr, b.kglpnadr, b.kglpnuse,
  2         b.kglpnses, b.kglpnhdl, b.kglpnlck, b.kglpnmod, b.kglpnreq
  3 FROM v$session a, x$kglpn b WHERE a.saddr = b.kglpnuse AND b.kglpnmod <> 0
  4 AND b.kglpnhdl IN (SELECT p1raw FROM v$session_wait WHERE event LIKE 'library%') ;
SID USERNAME PROGRAM       ADDR     KGLPNADR KGLPNUSE KGLPNSES KGLPNHDL KGLPNLCK KGLPNMOD
--- -------- ------------- -------- -------- -------- -------- -------- -------- --------
 30 IPNMS     sqlplus@gs-db 0191BEC0 BF2024C4 BE0AE940  BE0AE940 BD18EB4C BF1FA208   3
 54 IPNMS      sqlplus@gs-db 0191BEC0 BF13814C BE0BB360 BE0BB360 BD389EF0 00          3
SQL> SELECT sql_text FROM v$sqlarea
  3 WHERE (v$sqlarea.address, v$sqlarea.hash_value) IN (
  4     SELECT sql_address, sql_hash_value FROM v$session
  5         WHERE SID IN (SELECT SID FROM v$session a, x$kglpn b
  6                   WHERE a.saddr = b.kglpnuse AND b.kglpnmod <> 0
  7               AND b.kglpnhdl IN (SELECT p1raw FROM v$session_wait
```

```
  8                                                WHERE event LIKE 'library%')));
SQL_TEXT
--------------------------------------------------------------------------------
truncate table iptt_pm_all
```

至此，发现了导致问题的关键所在，持有 pin 的用户在执行 truncate table iptt_pm_all 的操作。

问：这个 truncate 是嵌在过程里面的？

答：是的，在一个 loop 中间的。每半个小时调用一次，类似的怎么也有 10 个程序吧，公用 iptt_pm_all 临时表。

我请求查看网友的代码，在一个 Procedure 中发现了大量如下语句（做了适当简化）：

```
update iptt_pm_all p
     set n27 = (SELECT count(*)
                  FROM iptca_interface b
                 WHERE p.int_id = b.related_node
                   AND b.iftype = 18
                   AND b.IFOPERSTATUS IN (1,5));

 insert into iptt_pm_all (col_time, int_id, ipaddr,
                n1, c1, n2, c2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
                n17, n18, n19, n20, c3, n21, c4, n22, c5, n23, n24, n25, n26, n27)
         select compress_day, int_id, object_ip_addr, ……..
           from iptaws_gwgj_hour
          where compress_day = v_time.col_time;

 SELECT col_time,n21,c4,n22,
         c5,n1,c1,n2,c2,
         sum(n3),sum(n4),sum(n5),sum(n7),sum(n8),sum(n9),
         sum(n10),sum(n11),sum(n12),sum(n13),sum(n14),
         sum(n16),sum(n17),sum(n18),sum(n19),sum(n20)
    FROM iptt_pm_all
   GROUP BY col_time,n21,c4,n22,c5,n1,c1,n2,c2;

  v_dsql := 'truncate table iptt_pm_all';
  EXECUTE IMMEDIATE v_dsql;
```

类似的存储过程还有很多。我请求获取 Shared Pool 的转储文件用于分析，Level 32 级。

```
ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 32';
```

限于篇幅，这里不再列举 dump 文件内容，需要注意的是，在生产环境上使用以上命令应该十分慎重。如果 Shared Pool 很大，转储文件可能非常巨大，而且可能引发性能问题和 Bug。

根据 trace 文件及 MetaLink 说明，最终发现问题是由于 truncate 临时表时不适当地请求了

排他锁所致，理论上 truncate 临时表无需排他锁定，但是 Oracle 使用了与处理常规表同样的方式处理临时表的锁定，从而导致了 Library Cache Pin 和 Library Cache Lock 的竞争，而且该问题并未作为 Bug 修正。由于该问题主要当多户交叉访问时引起，所以建议对于不同用户改用独立的临时表，此问题就可得以避免。

6.2.13 小结

Shared Pool 的管理是 Oracle 内存管理中相对复杂的一部分内容，在性能调整时也是非常重要的内容，深刻理解 Shared Pool 的实现，有助于进一步了解 Oracle 的实现及内部机制。本章就这一方面进行了一点探索，由于个人能力及认知有限，错漏之错在所难免，期待大家指正。

限于篇幅，本文作了适当简化，更完整的内容可以在我的网站（www.eygle.com）上找到。

参考信息及建议阅读

（1）Troubleshooting and Diagnosing ORA-4031 Error　Metalink Note:396940.1

（2）Understanding Shared Pool Memory Structures　By　Russell Green

（3）Shared Pool Internals　By　Lawrence

（4）Oracle 8.1 Buffer Cache Management White Paper　By　Nitin Vengurlekar

（5）LIBRARY CACHE LOCK, PIN AND LOAD LOCK　Metalink Note: 444561.1