**Oracle内核揭秘学习笔记**

云和恩墨(北京)信息技术有限公司

技术顾问 燕鑫

http://www.enmotech.com

**文档控制：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序** | **版本号** | **更改人** | **日期** | **备注** |
| 1 | 1.0版 | 燕鑫 | 2019-03-18 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 调优方法论 - 4 -](#_Toc3818993)

# 存储结构

## 区

# 调优排故方法论

# Buffer Cache内部原理与I/O

## HASH链表

Oracle在buffer cache中找一个block使用的仍然是hash算法。

### hash链表与逻辑读

HASH链表中非常重要的一个概念就是bucket，buffer cache中的hash bucket的数量由\_db\_block\_hash\_buckets决定：

SQL> show parameter \_db\_block\_hash\_buckets

NAME TYPE VALUE

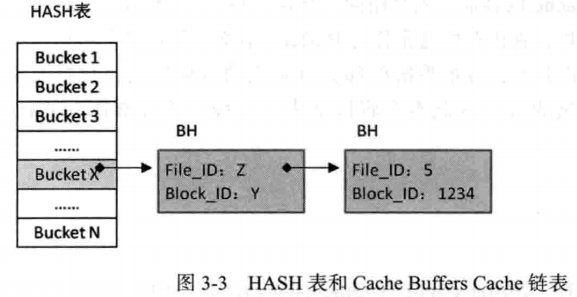
------------------------------------ ----------- ------------------------------

\_db\_block\_hash\_buckets big integer 524288

当进程要读取一个块时，它会先根据文件号和块号计算hash值。

我们假设现在一个进程要读5号文件的1234号块，计算出来的hash值为X，那么进程就会直接定位到Bucket X，定位到Bucket X就可以读取里面的内容，Bucket中记录的就是第一个BH的地址，也就是该bucket对应的CBC（Cache Buffers Chains）的链表头地址。

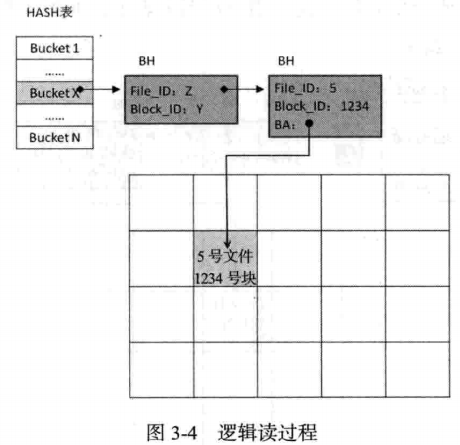
如下图：



在buffer cache中，每一个buffer都有一个buffer header。

我们看当进程从定位到bucket X的时候，也就确定它所需的buffer所在的CBC，从bucket X找到CBC的链表头，然后依次对BH进行查找，在本例中，进程查到第二个BH的时候，就找到了目标buffer的buffer header。

接着就是根据BH中记录的buffer的地址（也就是下图中BA）找到对应的buffer，整个过程如下：



最后总结一下整个逻辑读的过程：

1. process根据要访问的块的file#和block#计算HASH值
2. 根据HASH值找到HASH Bucket
3. 搜索Bucket后的CBC，找到对应buffer的BH
4. 根据BH中的BA找到对应的buffer进行访问。

显然，如果没有找到对应的BH，那么就需要进行物理读。

### Cache Buffers Chains Latch与Buffer Pin锁

SGA是公共内存，哪怕只访问公共内存中的一个字节，都一定要有某种锁机制进行保护，Oracle使用的就是latch和mutex。

在找到对应的bucket之后，开始进行CBC搜索之前，Oracle会在CBC链表头加一把锁，叫做CBC Latch。也就是说访问CBC之前必须要先申请CBC Latch。

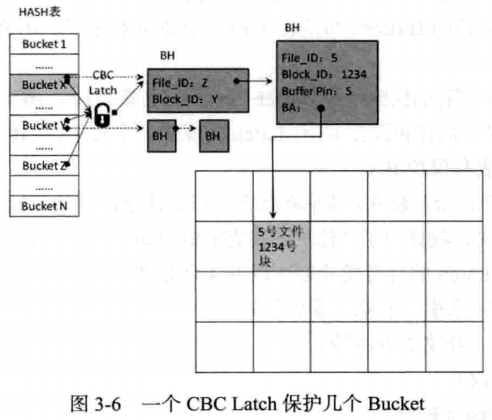
CBC Latch的作用主要有俩个：

1. 保护CBC的访问
2. 修改BH中buffer pin锁的状态。

在修改完BH中的buffer pin锁的状态后，CBC Latch就可以被释放了，接下来，进程将在Buffer Pin锁的保护下访问buffer。

这里稍微提一下，Buffer Pin锁常见的模式有俩种：S和X。在没有加锁的时候，buffer pin的值为0；在逻辑读时是S；在DML时是X。详细都后面再说。

一个CBC Latch要同时保护几个bucket，如下图：



Oracle这样做的目的就是为了节约内存。我们可以通过如下方式，统计CBC Latch所占的内存量。

select latch\_size

from (select to\_number(a.addr, 'xxxxxxxxxxxxxxxxxxxxx') -

to\_number(lag(addr) over(partition by name order by addr),

'xxxxxxxxxxxxxxxxxxxxx') latch\_size

from V$LATCH\_CHILDREN a

where name = 'cache buffers chains')

where latch\_size is not null and rownum=1;

LATCH\_SIZE

----------

200

我们看到，每个CBC Latch占用200字节内存。如果每个bucket一个latch的话，那么将会占用很大一块内存。

我们再来说Buffer Pin。Buffer Pin实际上就是BH中的几个字节，但它同时也是一个锁，只有将它修改为S或X才能进一步访问buffer。

那么当访问完buffer，还要释放Buffer Pin，也就是再一次修改buffer pin的状态，那么就又需要在CBC Latch的保护下修改BH。

所以，再次强调，CBC Latch就俩个作用：保护CBC和BH。

CBC Latch有俩种持有模式，共享和独占。CBC Latch的模式不是通过简单的读或写来确定的。CBC Latch的持有模式取决于下列4个要素：

1. 对象类型（唯一索引、非唯一索引等）
2. 块类型（根块、叶块或表块等）
3. 操作（读、修改）
4. 访问路径（Access Path）

除了表有唯一索引外，无论读写，大多数情况下，访问表块都是以X模式获得CBC Latch。所以前面我们所说的流程都是持有的X模式的CBC Latch。

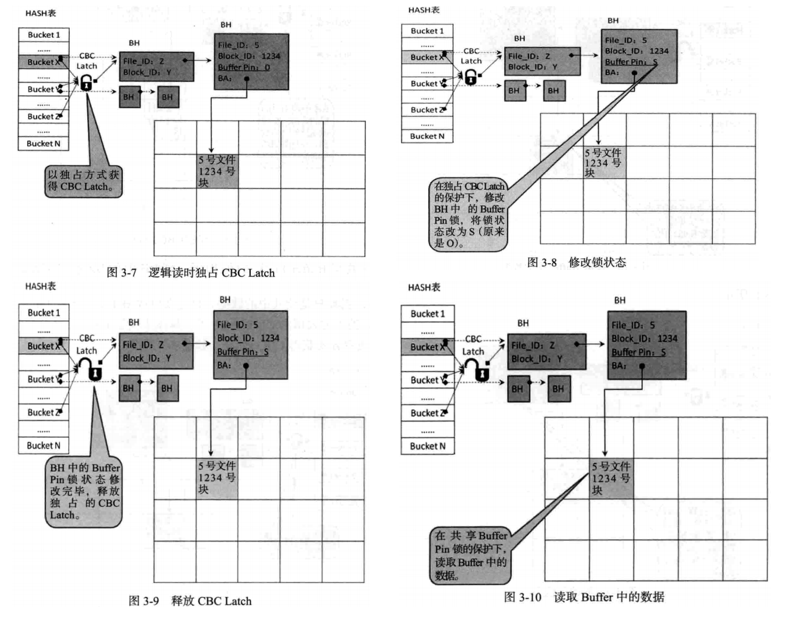
另外，索引的根块、枝块只要不修改，都是获得的S模式的CBC Latch。

#### 对表块的访问

其实访问表块时以X模式持有CBC Latch很好理解，因为进程要对BH进行修改。

下面是访问一个表块更详尽的过程：

1. 以独占模式在CBC链表头加上CBC Latch
2. 在Latch的保护下将BH中的Buffer Pin从0修改为S
3. 释放加在链表头的CBC Latch
4. 根据BA，在S模式的buffer pin的保护下，找到并访问buffer。

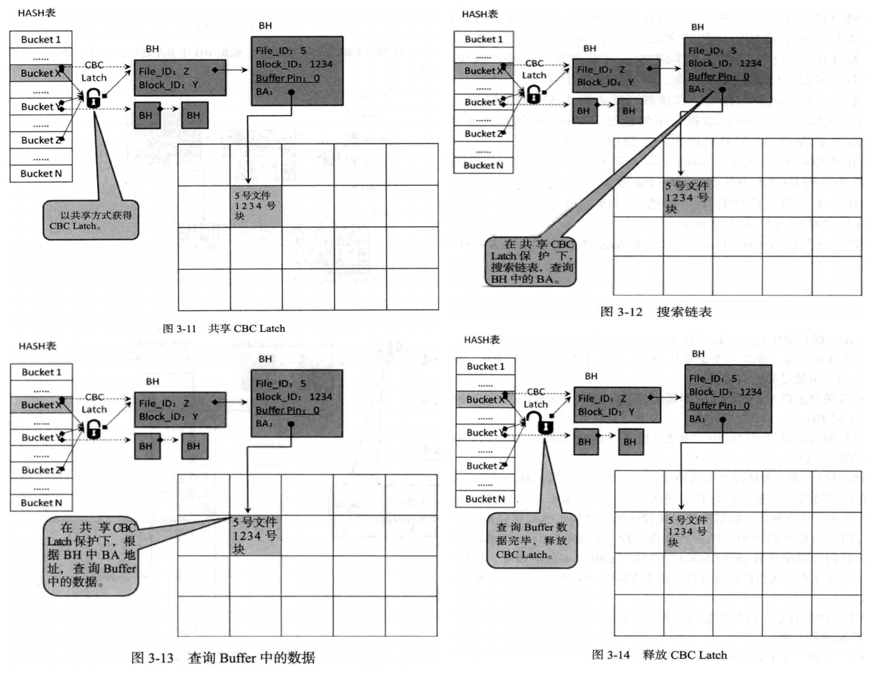


#### 对根块、枝块的访问

由于对根块枝块的访问频率比叶块和表块高的多，所以如果依然使用上述的访问方式，则会造成大量CBC Latch的争用。

具体步骤如下：

1. 以共享模式获得CBC Latch
2. 在共享模式的CBC Latch的保护下直接找到对应的BH，并读取其中的BA，但不修改Buffer Pin。
3. 在共享模式的CBC Latch的保护下根据BA找到并访问buffer
4. buffer访问完毕后，释放共享模式的CBC Latch



除了普通索引的根块和枝块，在INDEX UNIQUE SCAN时，需要访问的根块、枝块、叶块以及表块都是在共享模式的CBC Latch的保护下进行访问的。

但是对于TABLE ACCESS BY USER ROWID，及直接使用rowid进行对表块的访问，仍然时在X模式的CBC Latch的保护下进行访问的。

### CBC Latch和Buffer Pin的竞争

CBC Latch的竞争是不同的进程以不兼容的模式访问同一个CBC Latch导致的。

Buffer Pin的竞争是不同的进程以不兼容的模式获取Buffer Pin产生的，主要就是试图将已为X的buffer pin修改为S。

主要有俩种情况：

1. 热链竞争：访问不同bucket下的CBC或者同一CBC中的不同BH。
2. 热块竞争：访问相同BH，在Buffer Pin上产生的竞争。

#### 热链竞争

SQL> select hladdr,count(1) from x$bh group by hladdr having count(1)>5;

HLADDR COUNT(1)

---------------- ----------

0000000076B01DF0 6

select c.rfile#, dbablk, owner, object\_name, object\_id, object\_type

from x$bh a, dba\_objects b, v$datafile c

where hladdr = '0000000076B01DF0'

and a.obj = b.data\_object\_id

and object\_type = 'TABLE'

and c.file# = a.file#;

RFILE# DBABLK OWNER OBJECT\_NAME OBJECT\_ID OBJECT\_TYPE

---------- ---------- ---------- --------------------- ---------- -----------------------

1 30259 SYS DEPENDENCY$ 83 TABLE

1 19313 SYS OPTSTAT\_HIST\_CONTROL$ 694 TABLE

1 19313 SYS OPTSTAT\_HIST\_CONTROL$ 694 TABLE

1 19313 SYS OPTSTAT\_HIST\_CONTROL$ 694 TABLE

1 19313 SYS OPTSTAT\_HIST\_CONTROL$ 694 TABLE

1 19313 SYS OPTSTAT\_HIST\_CONTROL$ 694 TABLE

SQL> select dbms\_rowid.ROWID\_CREATE(1,83,1,30259,0) from dual;

DBMS\_ROWID.ROWID\_C

------------------

AAAABTAABAAAHYzAAA

SQL> select dbms\_rowid.ROWID\_CREATE(1,694,1,19313,0) from dual;

DBMS\_ROWID.ROWID\_C

------------------

AAAAK2AABAAAEtxAAA

declare

r number;

begin

for i in 1..10000000 loop

select count(\*) into r from DEPENDENCY$ where rowid='AAAABTAABAAAHYzAAA';

end loop;

end;

/

declare

r number;

begin

for i in 1..10000000 loop

select count(\*) into r from OPTSTAT\_HIST\_CONTROL$ where rowid='AAAAK2AABAAAEtxAAA';

end loop;

end;

/

一定要注意PDB下的RFILE#和FILE#不相同，构建rowid需要使用rfile#。

#### 热块竞争

##### 原理

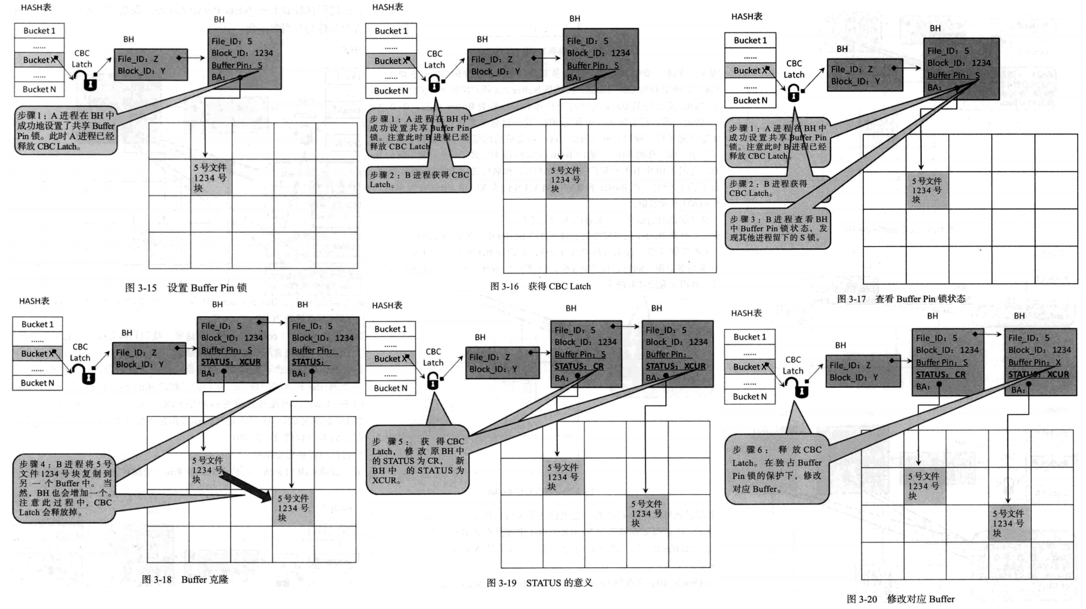
如果争用发生在Buffer Pin上，那么就是著名的buffer busy wait。

Buffer Pin有俩种模式，共享和独占：

1. 读一个buffer需要先获得共享buffer pin
2. 写一个buffer 需要先活动独占buffer pin

这俩种模式显然是不兼容的，那么如何做到读不阻塞写呢，步骤如下：

1. 首先A进程成功将BH的buffer pin修改为S，然后释放了CBC Latch
2. B进程此时要修改buffer，正是因为之前的A进程已经释放了CBC Latch，此时B进程才能获取CBC Latch。否则，就成CBC Latch的争用了。
3. B进程想要将BH的buffer pin修改为X，但是发现有其他进程留下的S锁，虽然不兼容，但是B进程不会等。
4. B进程也留下一个共享的buffer pin，然后释放CBC Latch，接着在共享buffer pin的保护下将当前buffer克隆一个出来，这就是buffer clone。  
   但是一个buffer就要有一个BH，所以B进程又必须获得CBC Latch，创建一个新的BH，然后释放CBC Latch。现在buffer cache中多了一个新的BH和新的buffer。  
   我们注意，这时候旧的BH的status的属性仍然是XCUR，而新的BH的status的属性是空的。
5. B进程再次获得CBC Latch，在CBC Latch的保护下，B进程要做俩件事，首先把原BH的status由XCUR改为CUR、将新BH的status改为XCUR，然后将新BH的buffer pin改为X。   
   至此buffer clone才算真正完成！  
   我们看到B进程事实上也不会影响A进程的读，因为A读的是原buffer，而B修改的是原BH。
6. 最后B进程释放CBC Latch，在X模式的Buffer Pin的保护下修改新buffer。



这时，如果C进程对也要访问（无论目的是读还是写）5号文件1234号block，那么它会发现此时状态为XCUR的BH的buffer pin为X，那么C进程就要进入等待，这就是buffer busy wait。

##### 什么叫做buffer正在被修改

但是我们一定要注意区分正在修改的buffer和正在执行的事务这俩个概念：

1. buffer pin为X的时候，是buffer正在被修改，比如本来要把1234写入buffer，才写到12，那么这叫正在被修改。一旦buffer被修改完，buffer pin就会被释放。
2. 而事务提不提交只是决定buffer被修改的内容能不能永久化，发起事务会同时发起对buffer的修改，但是不代表事务结束才叫buffer修改结束。这就是为什么我们能进行一致读，根据xcur构建CR block的原因。换句话说，当buffer正在被修改的时候，是不能进行CR block的构建的。

##### 实验一

如下是对上述过程的实验：

开启俩个会话，132会话读，624会话写：

declare

r number;

begin

for i in 1..10000 loop

select count(\*) into r from t1 where rowid='AAARlTAAEAAAJFNAAA';

end loop;

end;

/

declare

r number;

begin

for i in 1..10000 loop

update t1 set id=id+0 where rowid='AAARlTAAEAAAJFNAAA';

end loop;

end;

/

SQL> select SID,TOTAL\_WAITS,EVENT from v$session\_event where sid in (132,624) and event like '%buffer%';

SID TOTAL\_WAITS EVENT

---------- ----------- ------------------------------

132 19 latch: cache buffers chains

132 208 buffer busy waits

624 20 latch: cache buffers chains

624 1 buffer busy waits

624 134 log buffer space

可以看到，进行读的会话被进行写的会话严重阻塞。

##### 实验二（事务提交频次的影响）

我们在上一个实验看到写会话也有一次buffer busy wait，吕海波老师的解释是，当132会话构建CR block时会在对应的undo block上加更高级别的Buffer Pin，导致624会话在写undo block的时候遭遇阻塞。这个有待进一步验证。

所以吕老师说缩短事务执行时间会降低buffer busy wait没问题。但是吕老师说，通过提高提交频次来缩短事务的方式，我暂时持不同观点：

1. 调高提交频次意味着提高了事务量
2. 频繁的提交也意味对undo段头进行频繁的修改。

我们接下来再做一个实验：

505会话不停的读：

declare

r number;

begin

for i in 1..10000000 loop

select count(\*) into r from t1 where rowid='AAARlTAAEAAAJFNAAA';

end loop;

end;

/

628会话分别执行三次3个存储过程，我们看每次的效果：

第一次，循环3次提交1次：

SQL> begin

2 for i in 1..10000 loop

3 update t1 set id=id+0 where rowid='AAARlTAAEAAAJFNAAA';

4 if mod(i,3)=0 then

5 commit;

6 end if;

7 end loop;

8 end;

9 /

PL/SQL procedure successfully completed.

SQL> select SID,TOTAL\_WAITS,EVENT from v$session\_event where sid in (505,628) and event like '%buffer%';

SID TOTAL\_WAITS EVENT

---------- ----------- ------------------------------

505 91 latch: cache buffers chains

505 1620 buffer busy waits

628 9 latch: cache buffers chains

628 35 buffer busy waits

第二次，最终提交一次：

SQL> begin

2 for i in 1..10000 loop

3 update t1 set id=id+0 where rowid='AAARlTAAEAAAJFNAAA';

4 end loop;

5 commit;

6 end;

7 /

PL/SQL procedure successfully completed.

SQL> select SID,TOTAL\_WAITS,EVENT from v$session\_event where sid in (505,628) and event like '%buffer%';

SID TOTAL\_WAITS EVENT

---------- ----------- ------------------------------

505 93 latch: cache buffers chains

505 1825 buffer busy waits

628 15 latch: cache buffers chains

628 69 buffer busy waits

第三次，1次循环1次提交：

SQL> begin

2 for i in 1..10000 loop

3 update t1 set id=id+0 where rowid='AAARlTAAEAAAJFNAAA';

4 commit;

5 end loop;

6 end;

7 /

PL/SQL procedure successfully completed.

SQL> select SID,TOTAL\_WAITS,EVENT from v$session\_event where sid in (505,628) and event like '%buffer%';

SID TOTAL\_WAITS EVENT

---------- ----------- ------------------------------

505 506 latch: cache buffers chains

505 4998 buffer busy waits

628 25 latch: cache buffers chains

628 128 buffer busy waits

我们可以看到随着事务提交频次的上升，争用将愈加严重，读和写的会话的等待都会变得严重，而且，执行读操作的会话的等待随事务提交频次上升，可以说是成倍的加剧，无论是CBC Latch的争用还是Buffer Pin的争用。

#### 解决竞争

我们再来捋一遍整个buffer进行修改的过程：

1. 申请CBC链头加CBC Latch
2. 修改BH中的Buffer Pin为X
3. 释放CBC Latch
4. 生成Redo、Undo
5. 修改buffer中的内容
6. 申请持有CBC Latch
7. 修改BH的Buffer Pin为0
8. 释放CBC Latch

对于除了第4步的解决竞争的办法有：

1. 优化sql：无论是读还是写，高效的sql意味着更少的竞争。
2. 离散数据：例如增大pctfree、使用更小的block size、对表进行HASH分区
3. 抬高HWM：就是让段可以支持更高的并发

这里注意，HASH分区虽然会减少热块争用，但是会加大聚簇因子，从而导致加大range scan的代价。

而对于第4步，undo方面之前说过了。而redo方面需要简单的捋一下redo的流程：现在PGA生成redo record，然后传到shared pool中的private redo log buffer（IMU）或者直接传到log buffer（非IMU），然后才能修改要修改的buffer。

那么这个过程中，如果log buffer暂时没有空间，那么进程就需要等待LGWR将log buffer中的内容刷到redo上来腾出空间，此时会有一个等待事件叫log buffer space，我们在实验一中其实看到了。

所以如果第4步慢，那么也会导致更多的buffer busy wait，当最显著增多的不是等待次数，而是每次等待的时间。

所以跟根据上述原理就存在如下几种种情况：

1. log buffer 小了
2. 事务量异常增多，lgwr将log buffer中的内容写入到磁盘上的速度跟不上redo产生的速度。
3. 归档满了，redo无法完成切换
4. 主备的日志传输模式为lgwr sync

所以第一种，我们会看到buffer busy wait伴随log buffer space，我们只需要增大log buffer就可以了。

那么第二种，除了上述俩种等待事件，我们可能还会看到log file sync的等待时间。事务量异常增多是从业务、sql上优化，而从硬件角度上，应当使用读写性能更好的存储来存放redo log。

还有一种情况，如果我们开了归档，而归档没空间了，导致无法进行日志切换，也会产生前俩种的连锁反应。

注意log file sync并不是直接导致buffer busy wait的原因，只是如果这俩并存，我们可以推断是否是存储性能出了问题。

## 检查点队列链表

### 检查点队列（CKPT-Q）

#### 产生脏块

ORACLE会把脏块“串”成一条链，好让DBWR知道该把哪些写入磁盘。

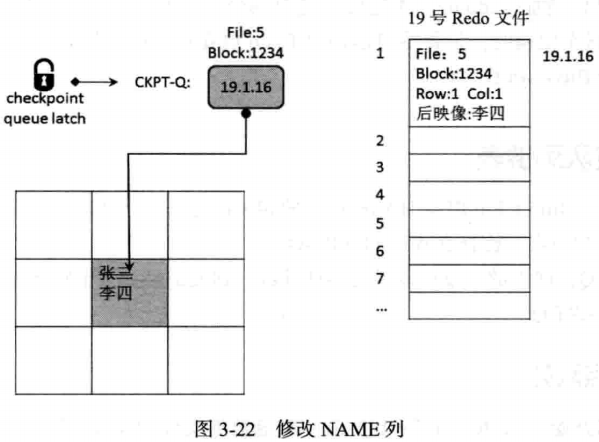
这样的链在buffer cache中有俩条，CKPT-Q和LRUW。本节主要说CKPT-Q。

假设现在没有任何脏块。此时5号文件1234号块被一个进程修改，那么该块就要变成脏块，而它是不是脏的，这个状态是记录在BH中的，当然就是前面说过的，持有CBC Latch对BH进行修改。

然后进程会首先持有checkpoint queue latch，然后将5号文件的1234号block加入到CKPT-Q。

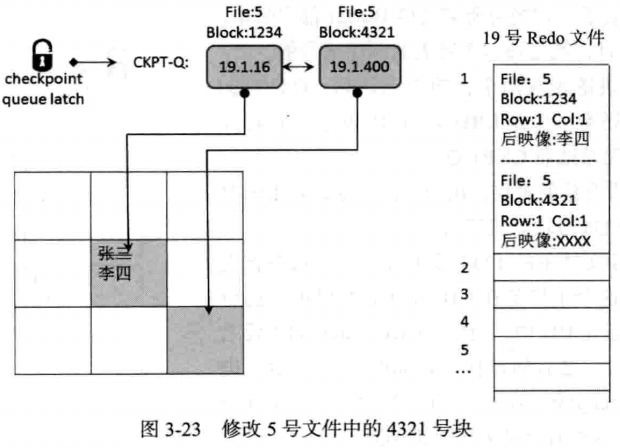
当然块被修改前还要产生对应的redo，假设将NAME列从张三修改为李四，并且当前的redo是19号（sequence号）redo，那么oracle将该redo record记录在redo文件的1号block的第16个字节处，那么这个Redo Block Address就是我们常说的RBA。

那么所有过程下来就如下图：



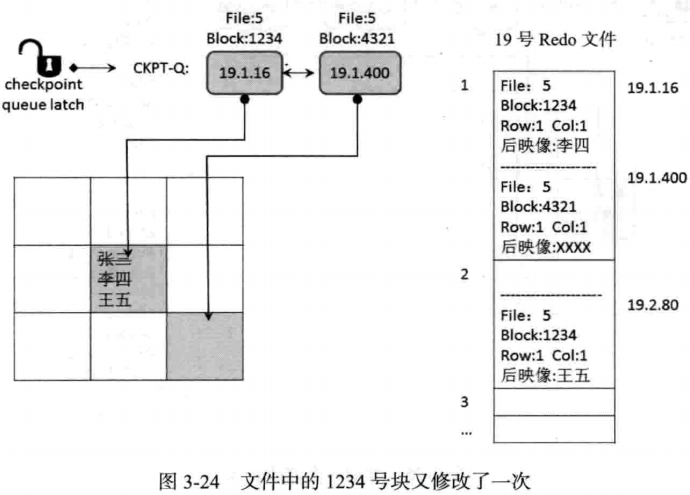
图中的19.1.16就是RBA。

那现在又一个块被修改了，它的RBA是19.1.400，因为它比之前的5号文件1234号块晚修改，那么它在CKPT-Q中就排在5号文件1234号块之后。如下图：



那如果再次修改5号文件的1234号块会怎么样呢？CKPT-Q不会变化，Redo新添一条record。

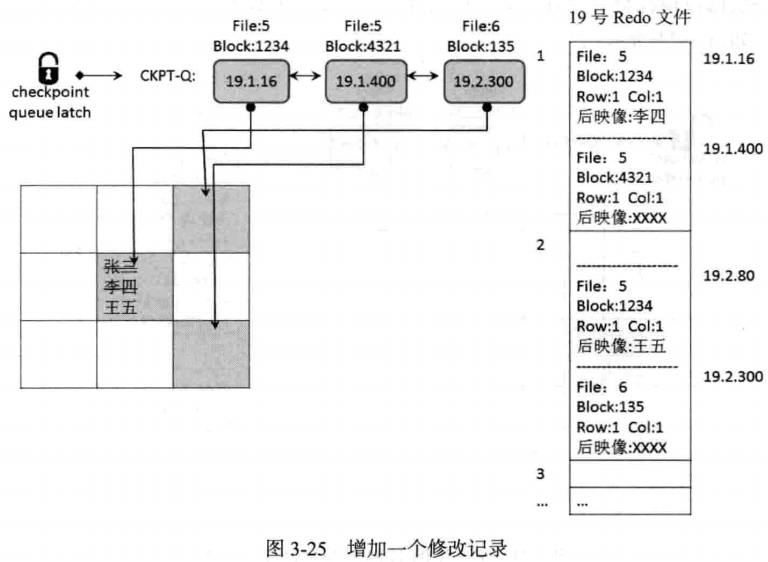
也就是说如果一个block被多次修改，只有第一次被修改时才会记录在CKPT-Q，而且它在CKPT-Q中的顺序直到被flush到磁盘之前都不会变化。如下图：



这里需要注意，正是由于这样的机制，所以一个脏块在进入CKPT-Q中时才需要持有checkpoint queue latch，而之后，在改脏块被flush到磁盘之前，无论对其进行多少次修改，都不再需要申请及持有checkpoint queue latch，因为CKPT-Q并不需要为此进行改变。所以我们也很难看到checkpoint queue latch的竞争。

那么对于block 1234，第一次修改记录在redo中的位置叫LRBA（19.1.16），最后一次被修改记录在redo的位置叫HRBA（19.2.80）。

说HRBA没啥用。我们来看如果再有另外一个block被修改，如下图：



我们看到，其实LRBA的顺序，就是CKPT-Q的顺序。

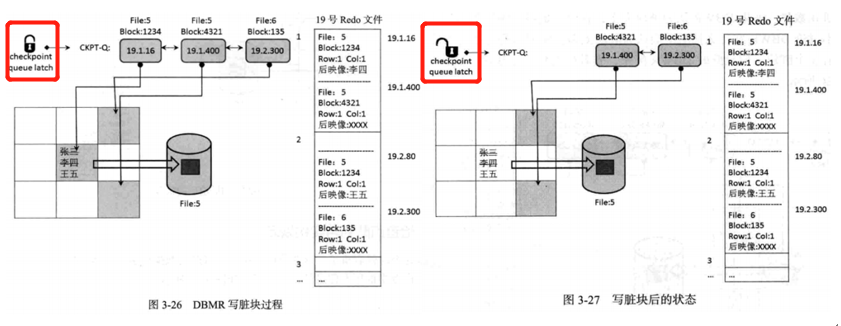
千万注意！将脏块加入到CKPT-Q中也是块被修改过程中的一步。

#### 写脏块

那现在我们来说些脏块。DBWR每3秒被唤醒一次查看CKPT-Q的长度，如果DBWR认为脏块太多了，那么它将CKPT-Q链头的一部分脏块写入磁盘中。

现在假设DBWR觉得需要些脏块了，那么它会先持有checkpoint queue latch扫描CKPT-Q，确定了要写的脏块后，DBWR会把脏块从CKPT-Q移到OBJ-Q（对象队列）中。移动完毕后会马上释放checkpoint queue latch。

假设本次DBWR只决定写一个脏块，全部流程如下图：



### 检查点队列与实例恢复

实例恢复其实本质就是把本来是该利用CKPT-Q完成的工作，用redo来进行完成。

那么能够实现这一需求的根本原因就是前面提到的，LRBA的顺序就是检查点队列的顺序。说白了就是redo记录块变化的顺序和CKPT-Q的顺序是一致的。

#### RBA

这里我从网上摘录了一段总结，我觉得很简明扼要，所以直接拿来用。

rba包括哪些内容？

1. 条目所在redo文件的sequence号
2. 条目所在日志文件的块号
3. 条目距离日志文件开始位置的偏移量

rba的分类：

1. low rba ：在buffer cache中的数据块第一次数据改变所对应的RAB。 脏数据块在检查点 队列里面按照low rba排列。
2. high rba ：在buffer cache中的数据块最近一次数据改变时所对应的RAB。
3. checkpoint rba：在checkpint queue中（每次checkpoint queue被clean以后）第一个脏数据块第一次被修改对应的RAB,这个RBA之前的脏数据已经被全部写入磁盘。
4. on-disk rba：是 lgwr 写日志文件的最末位置的地址。

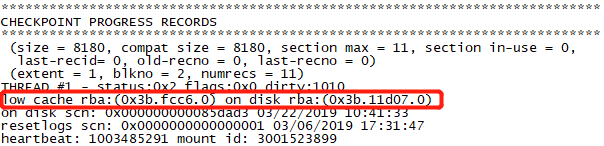
所以我个人理解，严格的说LRBA和HRBA是针对一个特定的block的。而控制文件中记录的low cache RBA就是上面所说的checkpoint RBA，也就是CKPT-Q的链表头对应的RBA。

#### 实例恢复过程

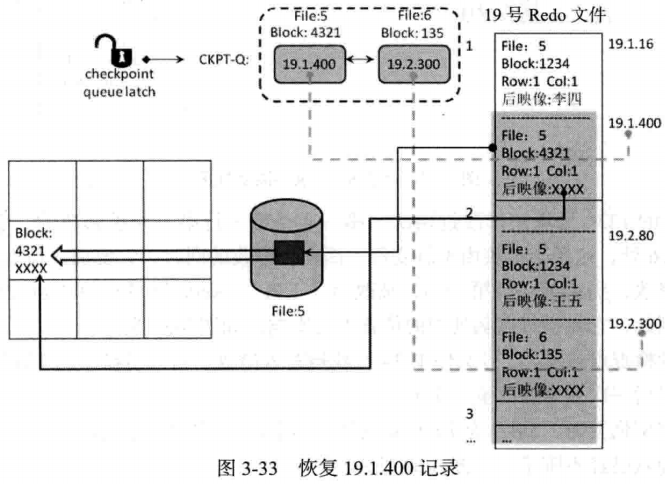
首先要确定从哪儿恢复到哪儿，这个虽然吕老师讲了很多，总结起来就是：从控制文件中的记录的low cache RBA恢复到on disk RBA。

就是说如果存在某个block的HRBA大于on disk RBA，那也只会恢复到on disk RBA，因为这个RBA之前的所有redo record都是确保已经从log buffer写到logfile中的。

如下是一个控制文件dump的checkpoint部分的截图：



那么这里我们着重记录下实例恢复的时候，到底是怎样将redo应用到对应的block的。



我们看到，是恢复进程将要恢复的block先读到buffer cache中，再将redo应用到该buffer上。

至于吕老师提到的重复恢复的情况，比如5号文件的1234号block的第二次修改其实已经写入到磁盘了，那么恢复的时候又要重新恢复一遍，因为它的RBA在low cache rba之后。

那么吕老师说ORACLE有机制可以减少这种情况发生，其实这个机制就是：BWR（Block Writren Record）和two-phase构建恢复集。这里就不详述了。

#### CKPT

CKPT-Q是存在与内存中的，所以它的链表头对应的RBA叫low cache RBA，而这个链表头就是我们常说的checkpoint position。

CKPT会每3秒一次，将low cache RBA，也就是checkpoint position对应的RBA写入到控制文件。

这就是我们常说的增量检查点。

### DBWR如何写脏块

DBWR每3秒会醒来一次，查看CKPT-Q是否过长，来决定是否进行写脏块。

#### 评判标准

对于上述的“过长”的评判标准取决于很多因素。比较被人熟知的就是fast\_start\_mttr\_target这个参数，但这个参数一般是没有人设置的。

比如说fast\_start\_mttr\_target设置为300，ORACLE会根据硬件写性能以及redo的产生量来估算一个恢复一个脏块需要的时间，比如估算出来需要20ms，那么当脏块超过15000个，DBWR就会进行写脏块，以保证实例能在300秒内完成启动。

10g以后，该值默认为0，及ORACLE会使用自调节检查点。

也就是说，Oracle会根据实际的硬件写性能、日志产生量等来计算写脏块的阈值。

#### 脏块合并

之前我们也提到过，DBWR会把要写的脏块从CKPT-Q移到OBJ-Q。

移到OBJ-Q就是为了对脏块进行整合。

首先说俩个概念：

1. 相邻，比如5号文件的101号块和102号块
2. 对象链表，每个对象（表、索引等）在SGA中对有各自的对象链表。  
   说白了，一个对象链表上的buffer都是属于这一个对象的。

虽然吕老师对于合并算法解释的非常详细，但是总结下来就四步：

1. 把要写的脏buffer分batch
2. 以每个batch为单位，将batch中的buffer分对象，及移到各自的对象链表上。
3. 各自对象链表上进行相邻脏buffer的合并，比如原本三个8k的block，合完就是一个24k的大内存块。  
   也就是把原本需要3次I/O的操作缩减成了1次I/O。

上面就是脏块的合并算法，那么由于在buffer cache中，即使是相邻的buffer，其内存的位置并不一定相邻，所以，ORACLE为该算法在shared pool中分配了一处I/O合并缓存区。相邻buffer的合并操作放在这里做。

该缓存区的大小由下面这个隐藏参数决定：

SQL> @yx/par

Enter value for par: \_db\_writer\_coalesce\_area\_size

old 6: AND x.ksppinm LIKE '%&par%'

new 6: AND x.ksppinm LIKE '%\_db\_writer\_coalesce\_area\_size%'

NAME VALUE DESCRIB

-------------------------------------------------- ---------- ----------------------------------------------------------------------------------------------------

\_db\_writer\_coalesce\_area\_size 4194304 Size of memory allocated to dbwriter for coalescing writes

那么它到底是不是shared pool里面的呢？

SQL> select pool,name,bytes from v$sgastat where name like '%coalesce%';

POOL NAME BYTES

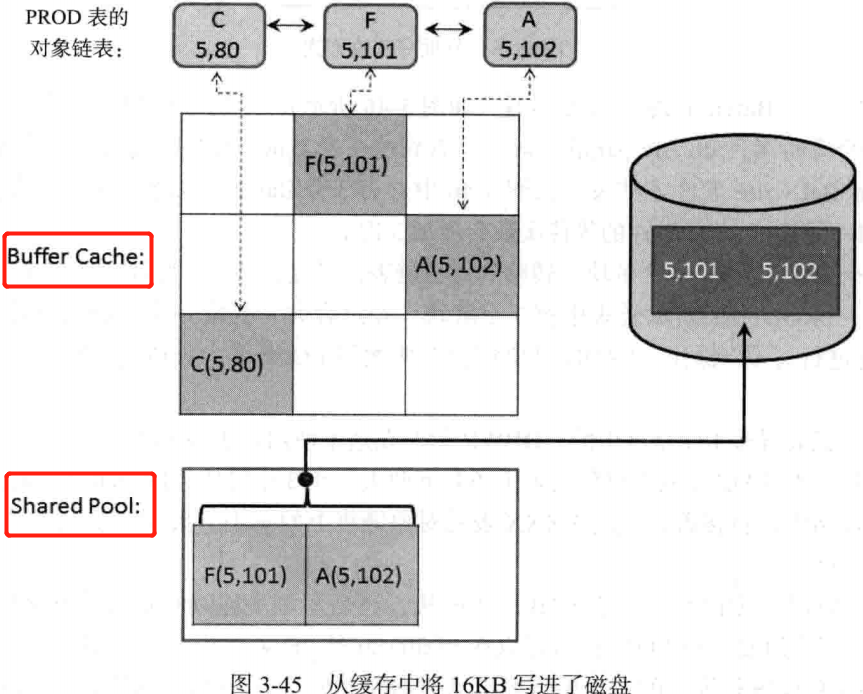
-------------- ------------------------------ ----------

shared pool dbwriter coalesce buffer 4198400

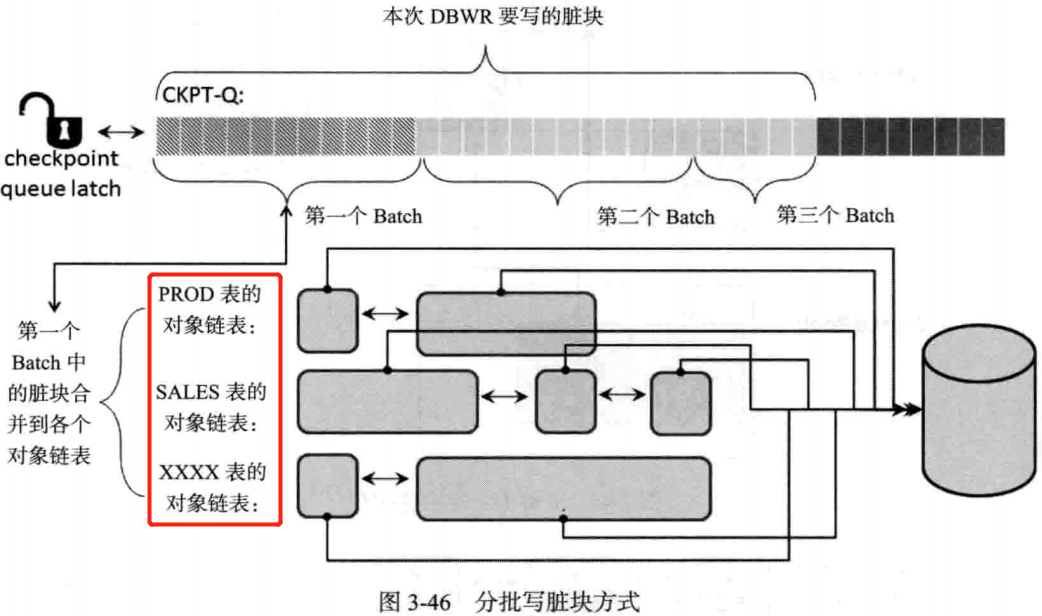
shared pool dbwriter coalesce struct 48

shared pool dbwriter coalesce bitmap 128

所以使用shared pool分配出来一块内存进行buffer合并图示如下：



完整的DBWR写的流程如下：



#### db file parallel write

前面说到DBWR从CKPT-Q把脏块移到OBJ-Q要分Batch，那么什么情况下才分batch呢？

显而易见，当DBWR觉得一次处理不完这么多脏块时，就要分batch，那怎么判断一次能处理完多少呢？其中一个重要因素就是shared pool给的dbwriter coalesce buffer的大小。

dbwriter coalesce buffer越大，那么需要分的batch就越少。

而每完成一个batch的处理，就是一次db file parallel write。

那么对于等待事件，我们往往不仅要关注次数，还要关注平均等待时间。

这里先说一些事实:

1. batch与batch之间的I/O串行的，也就是说DBWR处理完一个batch才能处理下一个batch。
2. batch内部的I/O是异步的（加入异步I/O）可用。

所以，batch大了，不代表处理就慢，因为大的batch可能由于大量可合并的相连的buffer，经过整合只需要很少的I/O，而小的batch可能大部分的buffer都是离散的，反而需要较多的I/O。

说白了，归根结底DBWR写的快不快还要看写物理I/O的次数和性能。

#### 判断DBWR的写I/O的性能

v$filestat的PHYWRTS可以查看对于每个文件的物理写I/O的次数，求sum就可以求出总的物理写I/O。

v$sysstat的physical write IO requests统计值也可以得到物理写I/O的总量。

SQL> select \* from v$sysstat where name='physical write IO requests';

STATISTIC# NAME CLASS VALUE STAT\_ID CON\_ID

---------- ------------------------------ ---------- ---------- ---------- ----------

186 physical write IO requests 8 83300 2904164198 0

SQL> select sum(PHYWRTS) from v$filestat;

SUM(PHYWRTS)

------------

83300

如果想了解ORACLE一共写了多少块，可以通过v$sysstat的physical writes的统计值获得。

SQL> select \* from v$sysstat where name='physical writes';

STATISTIC# NAME CLASS VALUE STAT\_ID CON\_ID

---------- ------------------------------ ---------- ---------- ---------- ----------

183 physical writes 8 126863 1190468109 0

那么我们判断DBWR的写I/O性能就依据如下3个指标：

1. 写I/O的总块数
2. 写I/O的总次数
3. db file parallel write的次数，也就是batch的个数。

我们通过：

平均每次写IO写入的块数=写IO的总块数/写IO的总次数

再结合db file parallel write的次数，可以推断出脏块合并的效率高不高。

显然，脏块合并的效率越高，DBWR写的性能就越好。

那么吕老师说，因为一个batch越大，存在连续block的可能性越高，而且也意味着batches的数量越少，所以对于DBWR的效率提升也会很有效。也就是说增大隐藏参数\_db\_writer\_coalesce\_area\_size会对DBWR的效率有一定的提升。

因为写块数取决于应用，我们无法掌控，而按照上面的说法，次数受批数的间接影响，所以调整\_db\_writer\_coalesce\_area\_size成了我们最好掌控的一个因素。

#### \_db\_writer\_coalesce\_area\_size实验

## LRU队列

一个workset有4条链表，分别是：

1. 主LRU链表、辅助LRU链表
2. 主LRUW链表、辅助LRUW链表

### 主LRU链表、辅助LRU链表

#### 基本概念

当要向buffer cache中读一个buffer时，决定应该覆盖那个已有的buffer，就要依赖LRU链表来做决策。

而所有LRU算法的本质就是覆盖最不常用的buffer。

在ORACLE中，LRU链表回将所有buffer都连接在一起，换句话说就是buffer cache中所有的buffer不是在主LRU上，就是在辅助LRU上。

我们知道，LRU链表有一个热端、冷端的概念，但是一定注意，4条链表中，只有主LRU才有热端、冷端的区分！

那么说一个buffer常不常用，就需要用一个计数器来记录。TCH以3秒为一个阶段记录buffer的访问次数，也就是说，3秒内访问多少次都只算一次。

我们这里做个小实验：

第一次select \* from t1，我们buffer 被读到buffer cache中，BH的状态是1，表示是xcur当前块，然后TCH是1：

SQL> select state,tch from x$bh where file#=11 and dbablk=135;

STATE TCH

---------- ----------

1 1

过3秒之后，在读一次就是TCH就变成了2：

SQL> select state,tch from x$bh where file#=11 and dbablk=135;

STATE TCH

---------- ----------

1 2

然后再过3秒，我在3秒内连续执行了5次select \* from t1，我们看到TCH也只增加了1：

SQL> select state,tch from x$bh where file#=11 and dbablk=135;

STATE TCH

---------- ----------

1 3

那么LRU是怎么根据TCH来选在可覆盖的buffer呢？接下来就是一个非常重要的概念，什么样的buffer可覆盖：

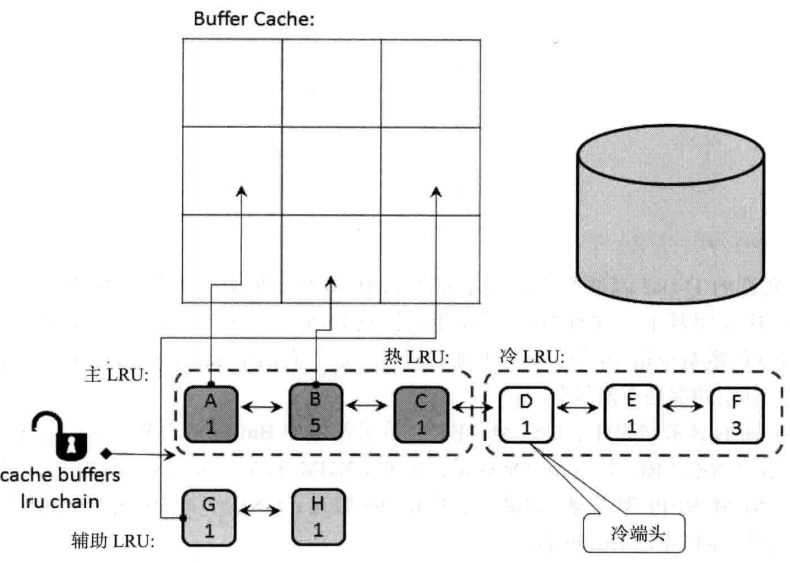
1. 非脏块
2. TCH小于2

那么什么样的buffer不可覆盖：

1. 脏块
2. TCH大于2
3. 正在被pin住的buffer

#### LRU和LATCH

如下就是一个workset中的LRU的概图，这里吕老师为了方式图画的乱，所以就只画了3个箭头表示BH与真实buffer的对应关系，另外每个BH里的数字就是TCH：



我们最开始就说了一个workset有4个链表，那么有链表就要有latch保护，所以每个workset由一把叫做：cache buffers lru chain latch的Latch保护。

主辅LRU上buffer数量的比例可以通过如下查询看到：

SQL> select cnum\_set,cnum\_repl,ANUM\_REPL from x$kcbwds;

CNUM\_SET CNUM\_REPL ANUM\_REPL

---------- ---------- ----------

26428 26428 23705

26436 26436 23749

26436 26436 23760

26430 26430 23662

26430 26430 23762

26438 26438 23300

26438 26438 23663

26428 26428 23762

72 rows selected.

这里我把是0的都去掉了，可以看到，目前有72个workset，在使用的是8个workset。

那么怎么证明1个workset由1把cache buffers lru latch保护呢：

SQL> select count(1) from v$latch\_children where name='cache buffers lru chain' and gets<>0;

COUNT(1)

----------

72

但实际上在被使用的也就只有8个latch，其他都是空闲的：

SQL> select gets,misses,immediate\_gets,SPIN\_GETS from v$latch\_children where name='cache buffers lru chain' and gets<>0;

GETS MISSES IMMEDIATE\_GETS SPIN\_GETS

---------- ---------- -------------- ----------

…

26 0 0 0

41143 3 47653 2

41181 2 47633 1

40743 4 47592 4

40271 3 47224 3

40511 5 47431 5

40354 3 47367 2

40416 2 47420 1

40416 5 47432 4

26 0 0 0

…

72 rows selected.

我们在第一个查询中的三列，CNUM\_SET，CNUM\_REPL，ANUM\_REPL的意思分别为：该workset中的buffer总数、该workset上所有LRU上的buffer总数、该workset的辅助LRU的buffer总数。

一般来说CNUM\_SET和CNUM\_REPL是相等的，而主LRU上buffer的数量就是CNUM\_REL-ANUM\_REL。

我们可以通过如下sql得到辅助LRU的buffer数量比例：

SQL> select round(sum(ANUM\_REPL)/sum(CNUM\_SET)\*100,2) rate from x$kcbwds;

RATE

----------

89.54

我们看到，我的实验环境中辅助LRU中buffer的占比是90%，是因为我的系统很闲，并且buffer最开始是在辅助LRU链上的，这个后面细说。

吕老师说，一般来说正常辅助LRU的buffer占比是25%，但这个也不尽然。

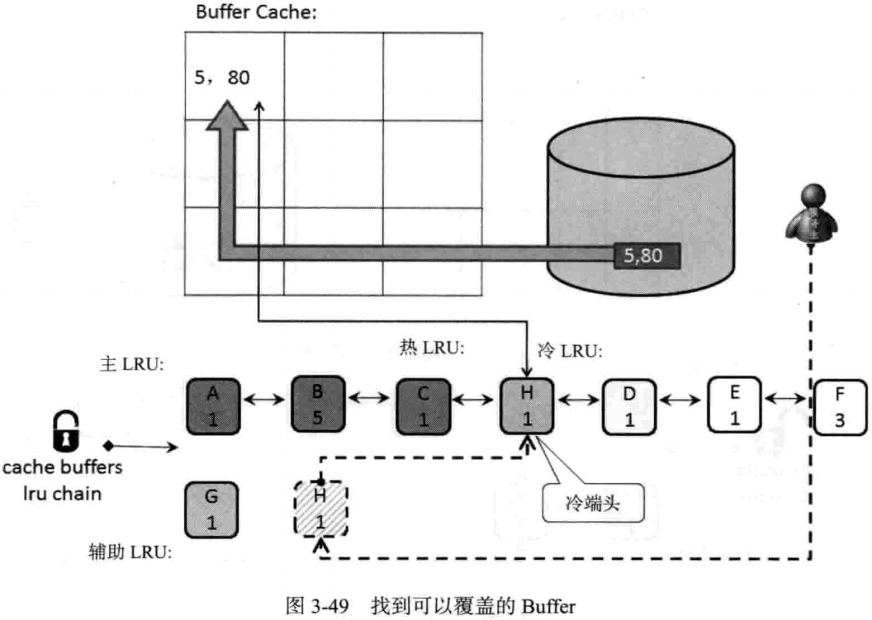
我们还是说上面那个图，吕老师做的真的很细致，图中又9个buffer，但是我们只看到8个BH，原因是吕老师想表达，虽然所有的buffer都在LRU中，但不一定都在主辅LRU上，也可能在主辅LRUW上。

#### 物理读

下面我们来看整个物理读的流程：

1. 在buffer cache中搜索CBC chain没找到需要的buffer，这个过程在3.1有详细记录。
2. 首先获得cache buffer lru chain latch
3. 从辅助LRU的尾端开始搜索可被覆盖的buffer。
4. 发现H符合可覆盖的条件（非脏块+TCH小于2）
5. 将H从辅助LRU移到主LRU的冷端头
6. 释放cache buffer lru chain latch
7. LRU链相关操作完成，进程发起一次真正的物理读，将块读入buffer cache。

这个流程如下图：

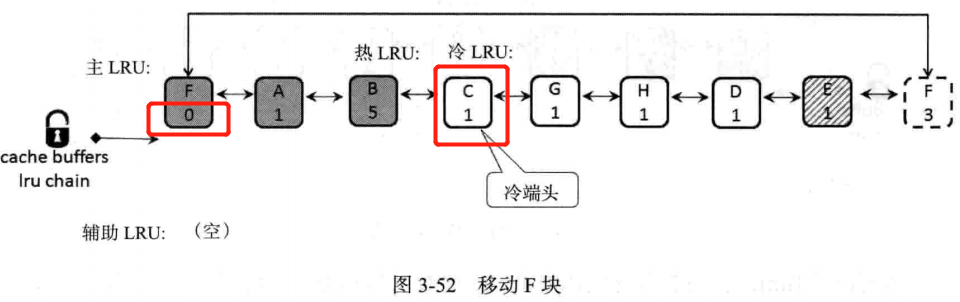


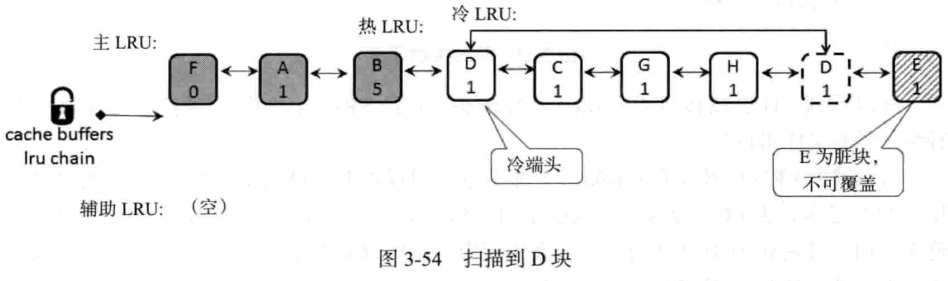
我们看到，现在辅助LRU中就只剩一个G了，那么再发生一次物理读，G也将移到主LRU的冷端头。那么这时候一种极端的情况就出现了，辅助LRU没有buffer了，那么再次发生物理读，就会从LRU的冷端尾开始找可覆盖的buffer。

那么这第三次物理读，LRU动作流程如下：

1. 先获得cache buffer lru chain latch
2. 扫描过程遇到TCH大于等于2的buffer，移动到热端头，并将其TCH置为0
3. 热端尾变成了冷端头
4. 遇到脏buffer跳过
5. 遇到可覆盖的buffer，将其移动到冷端头
6. 释放cache buffer lru chain latch

整个过程如下面俩张图：





#### 辅助LRU的作用

那么像现在这样，辅助LRU空了或者很少，怎么办？

SMON会每3秒醒来一次，在cache buffer lru chain latch的保护下，检查主、辅LRU链表的长度，如果辅助LRU的buffer少于25%，那么就从主LRU冷端的尾端开始扫描buffer，将TCH小于2的非脏块移到辅助LRU中。

可以看到，辅助LRU中的buffer都是TCH小于2+非脏的，也就是说都是可覆盖的，那么这对于搜索可覆盖的buffer的效率可以说是极大的提升。

另外，当数据库启动，或者刚flush过buffer cache后，所有buffer都在辅助LRU链表中。这就是为啥我自己的实验环境，辅助LRU链表的buffer占90%。我们可以做个小实验：

SQL> alter system flush buffer\_cache;

System altered.

SQL> select cnum\_set,cnum\_repl,ANUM\_REPL from x$kcbwds where CNUM\_SET<>0;

CNUM\_SET CNUM\_REPL ANUM\_REPL

---------- ---------- ----------

26428 26428 24428

26436 26436 24469

26436 26436 24485

26430 26430 24389

26430 26430 24490

26438 26438 24019

26438 26438 24387

26428 26428 24489

8 rows selected.

SQL> select round(sum(ANUM\_REPL)/sum(CNUM\_SET)\*100,2) rate from x$kcbwds;

RATE

----------

92.28

虽然不是百分之百，不过差不多吧，可以看到在做了flush buffer cache后，辅助LRU的buffer占比比之前变高了。

#### 判断buffer所在的链

我们可以通过X$BH的LRU\_FLAG来确定buffer的所在链以及是冷端还是热端：

LRU\_FLAG

0==>LRU-主链冷端的头部，这个比较特殊他在DUMP没有显示LRU\_FLAG

2==>LRU-主链冷端的尾部，lru-flags: moved\_to\_tail

4==>LRU-辅助链,lru-flags:on\_auxiliary\_list

6==>LRU-辅助链的尾部，lru-flags: moved\_to\_tail on\_auxiliary\_list

8==>LUR-主链热端

这个使用x$BH和buffer的dump来进行核对确认：

alter session set events 'immediate trace name buffers level 1';

举个例子，比如我先找t1表的一个块：

SQL> select lower(BA),LRU\_FLAG,TCH from X$BH where file#=11 and dbablk=135;

LOWER(BA) LRU\_FLAG TCH

---------------- ---------- ----------

00000000caeb2000 0 3

00000000e26a8000 2 2

00000000dbbec000 4 0

00000000e3878000 6 0

00000000b9912000 4 0

00000000b989e000 6 0

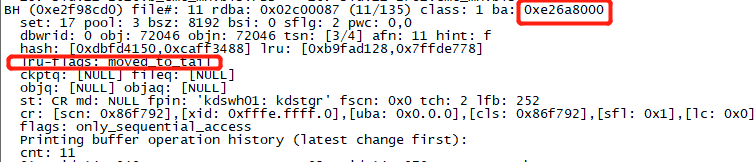
00000000ba63c000 6 0

00000000bc360000 6 0

8 rows selected.

我们知道BA是buffer的在内存的真实地址。所以我们可以就从buffer的dump中来对BH的很多信息进行确认，lru\_flag就是一例。

我们在dump出来trace中搜索上面的这几个BA，比如搜第二个，对应的LRU\_FLAG是2的BA，注意一定要转化成小写，在dump中搜索e26a8000：



从这个，我们就知道2代表的就是主LRU的冷端尾。

其他的都是这样一个个确认。

#### cache buffers lru chain latch的争用

我们之前看到，只有8个workset在工作，原因是因为我的实验环境cpu\_count是8：

SQL> show parameter cpu\_count

NAME TYPE VALUE

------------------------------------ ----------- ------------------------------

cpu\_count integer 8

不过这也不一定，客户生产环境160个cpu\_count，但是有320个workset被用过。但用的最多的仍然是160个。

我们前面也证明了，一个workset只有一个cache buffers lru chain latch保护。而该latch只有持有与不持有俩种状态，不存在共享一说，所以当频繁访问LRU链表时，就会产生该latch的竞争。

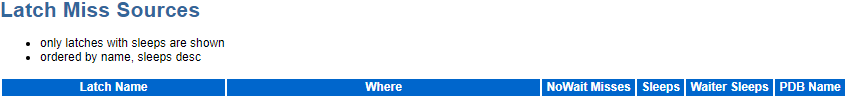
有俩种情况会需要频繁访问LRU：

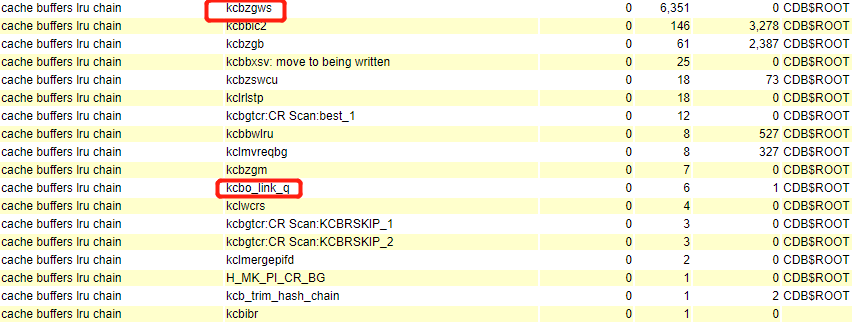
1. 前面说过的物理读
2. 构建CR block

在申请LATCH时，Oracle会将一个附加信息写到内存，来说明申请latch的原因。

我们可以在v$latch\_misses的where列获得该信息，AWR中Latch Miss Sources的信息就是来源于该视图：







那么物理读和构建CR的区别就是：

1. 物理会需要申请俩次该Latch，where中对应的是kcbo\_link\_q和kcbzgws。
2. 而CR只需要申请一次该Latch，对应的是kcbzgws。

所以AWR中如果看到只有kcbzgws有争用的话，那说明就是构建CR导致的Latch争用。

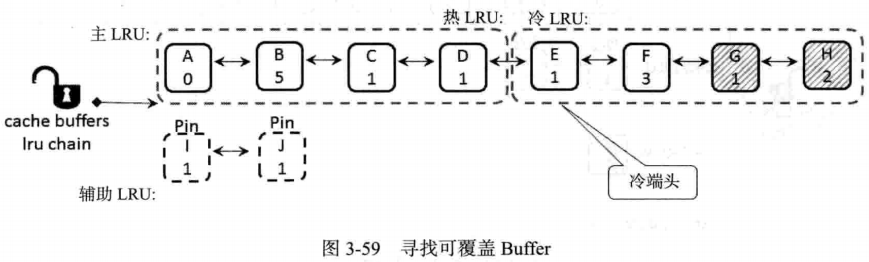
### 脏链表LRUW

LRUW可以说是对Oracle写脏块机制的一个完善。

与CKPT-Q不同，脏buffer不会第一时间放到LRUW中，而是当进程搜索可覆盖的buffer时，在主LRU中发现TCH小于2的脏buffer时，才会将该buffer放入LRUW中。

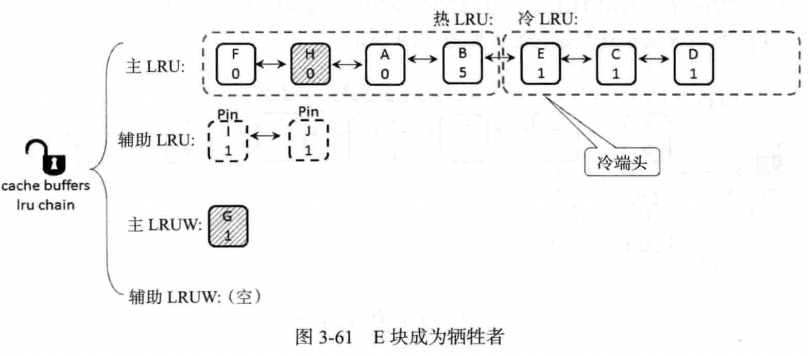
注意，该过程一定是发生在搜索主LRU时，因为辅助LRU中的buffer一定不脏。

我们看如下这个图例：



现在假设辅助LRU中的俩个buffer都被pin住了，那么现在要到主LRU中找可覆盖的buffer。找到H发现虽然它脏，但是TCH不小于2，所以把H移到热端头并将TCH置0，然后继续找，发现G是TCH小于2的脏块，那么就把G移到LRUW中，然后继续找，后面的过程就不详述了，最终会覆盖E。

最终搜索结果如下图：



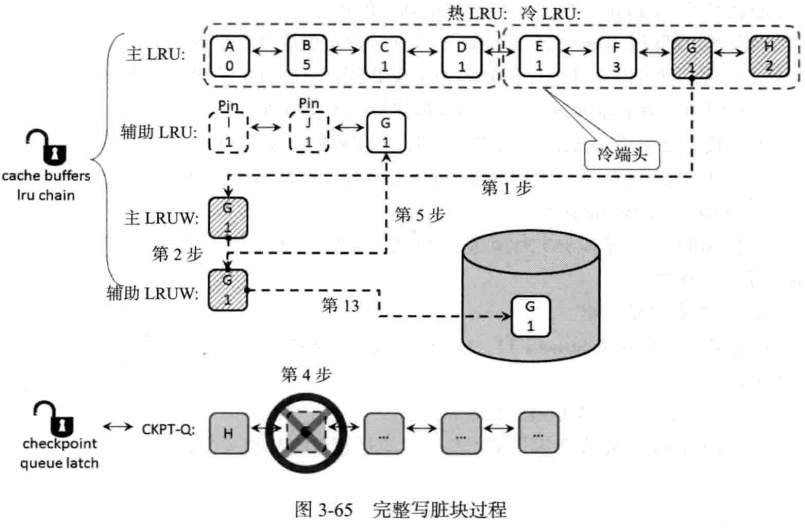
那么什么时候写LRUW中的脏buffer呢？

DBWR每3秒醒来检查主LRUW，如果有脏块就把它移到辅助LRUW，然后写脏块，并且把该buffer从CKPT-Q中去掉，最后把该buffer移到辅助LRU中等待被覆盖。如果是写多个脏块的时候，也会使用和之前介绍的一样的合并算法。

总结一下上面这句话就是：

1. DBWR写LRUW中的buffer要先移到辅助LRUW才写
2. DBWR醒来的时候，只要发现主LRUW中有buffer，就要进行写
3. 写脏buffer依然使用合并算法来减少写I/O
4. DBWR写完，还要把这些buffer从CKPT-Q中去掉。
5. 最后把这些buffer放到辅助LRU中。

如下是上述过程的完整图示：



LRUW中的buffer可以通过如下方式查询：

SQL> select cnum\_set,cnum\_repl,ANUM\_REPL,cnum\_write,anum\_write from x$kcbwds where cnum\_set<>0;

CNUM\_SET CNUM\_REPL ANUM\_REPL CNUM\_WRITE ANUM\_WRITE

---------- ---------- ---------- ---------- ----------

25465 25465 8366 0 0

25469 25469 8390 0 0

25468 25468 8389 0 0

25465 25465 8420 0 0

25465 25465 8430 0 0

25469 25469 8440 0 0

25470 25470 8469 0 0

25465 25465 8446 0 0

前3个列就不说了，CNUM\_WRITE是LRUW上的总buffer，ANUM\_WRITE是辅助LRUW上的buffer，所以主LRUW上的buffer就是这俩的差值。

#### 验证LRUW的工作机制

在linux上，Oracle写数据文件调用的函数：

1. 串行：pwrite64
2. 异步：io\_submit

虽然我是文件系统，但是我把异步IO打开了：

SQL> show parameter FILESYSTEMIO\_OPTIONS

NAME TYPE VALUE

------------------------------------ ----------- ------------------------------

filesystemio\_options string ASYNCH

SQL> show parameter disk\_async

NAME TYPE VALUE

------------------------------------ ----------- ------------------------------

disk\_asynch\_io boolean TRUE

所以我们这里用gdb设置断点要使用io\_submit：

[oracle@host01 ~]$ ps -ef|grep dbw

oracle 18682 1 0 11:54 ? 00:00:00 ora\_dbw0\_PRODCDB

[oracle@host01 ~]$ gdb -p 18682

(gdb) b io\_submit

Breakpoint 1 at 0x7f499f820690

(gdb) c

Continuing.

我们先来看下LRU的情况：

SQL> select cnum\_set,cnum\_repl,ANUM\_REPL,cnum\_write,anum\_write from x$kcbwds where cnum\_set<>0;

CNUM\_SET CNUM\_REPL ANUM\_REPL CNUM\_WRITE ANUM\_WRITE

---------- ---------- ---------- ---------- ----------

25710 25710 6388 0 0

25713 25713 6391 0 0

25713 25713 6391 0 0

25710 25710 6406 0 0

25710 25710 6435 0 0

25714 25714 6423 0 0

25715 25715 6422 0 0

25710 25710 6359 0 0

8 rows selected.

执行一个大的更新操作：

SQL> select count(1) from t1;

COUNT(1)

----------

2331872

SQL> update t1 set object\_id=object\_id+0;

然后观察：

gdb中出现：

Breakpoint 1, 0x00007fb96c791690 in io\_submit () from /lib64/libaio.so.1

然后我们来看LRUW的情况：

SQL> select cnum\_set,cnum\_repl,ANUM\_REPL,cnum\_write,anum\_write from x$kcbwds where cnum\_set<>0;

CNUM\_SET CNUM\_REPL ANUM\_REPL CNUM\_WRITE ANUM\_WRITE

---------- ---------- ---------- ---------- ----------

25710 25485 6193 225 207

25713 25482 6193 231 214

25713 25525 6240 188 133

25710 25578 6221 132 1

25710 25612 6277 98 18

25714 25612 6273 102 24

25715 25549 6207 166 23

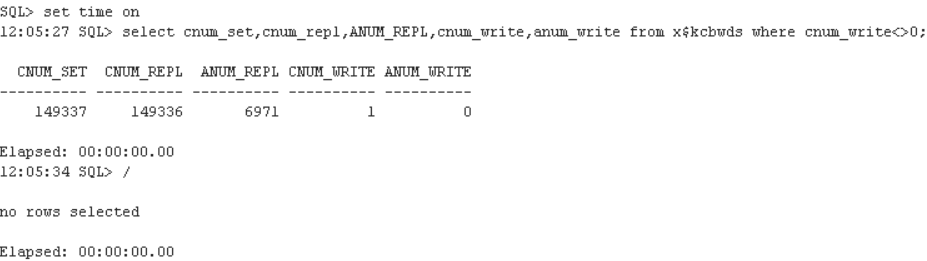
25710 25568 6256 142 46

8 rows selected.

我们看到这里堆积了很多在LRUW中的buffer，但是我们因为gdb设置了断点，所以dbwr在要进行写的时候停住了。

那么其实怎么证明LRUW中有脏buffer就dbwr就会写呢，其实上面这个是一种方式，只是要更新的表需要小一点。

还有就是找个繁忙的系统查一下就知道了：



### Free Buffer Waits

我们先来看俩个隐藏参数：

NAME VALUE DESCRIB

----------------------- ---------- ----------------------------------------------------------

\_db\_large\_dirty\_queue 25 Number of buffers which force dirty queue to be written

NAME VALUE DESCRIB

------------------------ ---------- ----------------------------------------------------------

\_db\_block\_max\_scan\_pct 40 Percentage of buffers to inspect when looking for free

虽然第一个隐藏参数说的是25个，但是吕老师已经验证过，这是25%。

\_db\_large\_dirty\_queue就是当CKPT-Q上的脏buffer的数量达到百分之25时，就要直接唤醒DBWR进行脏块写。

\_db\_block\_max\_scan\_pct就是当server process扫描LRU链表时，找了40%的buffer都还没有找到可覆盖的buffer，则停止扫描，唤醒DBWR写脏块，同时server process进入睡眠，开始等待free buffer waits，当再次醒来并找到可覆盖的buffer，本次free buffer waits结束。

这就是free buffer waits的来历。

那么这里面有一个关键点，扫描这40%不能被覆盖的buffer的过程中会发生些什么呢？其实就是对前面的一个总结，就是对三类不能覆盖的buffer的处理：

1. 正在被pin住的buffer将被跳过
2. TCH小于2的脏buffer将被移到主LRUW
3. TCH大于2的不论是不是脏buffer，都将被移到主LRU热端

那么跟\_db\_large\_dirty\_queue有什么关系呢？

这个参数就是为了减轻free buffer waits等待的，这个参数的意图很明显，就是不想让buffer cache中有太多的脏buffer，而导致在server process扫描LRU的时候，找了40%了都还没找到可覆盖的buffer（free buffer）。

还有一点需要注意，就是DBWR被唤醒后，一定会写LRUW里的脏buffer，而对于CKPT-Q的其余脏buffer是否要写还是按照常规判断，看CKPT-Q是否够长。所以说白了，在上述的来个过程中，DBWR不论是被强制唤醒，还是按时（每3秒）自己醒来，所做的工作都是一样的。

### DBWR的机制

我们现在来总结一下，DBWR在哪些时候被唤醒：

1. 自身3秒超时，超时后做俩件事情：  
   1)写LRUW的脏buffer；2)检查CKPT-Q的长度和redo的生成量，确定是否写脏块。
2. server process扫描LRU发现脏buffer数超过25%，会唤醒DBWR
3. server process扫描LRU，扫了40%的buffer依然没找到可覆盖的buffer，唤醒DBWR写脏块，等待free buffer waits
4. 还有下列其他情况：  
   alter system checkpoint;  
   正常关闭数据库（非abort）  
   日志切换  
   表空间或数据文件offline  
   Drop、truncate  
   直接路径读某个对象（为了保证数据的一致性）

在v$sysstat中有几个指标需要我们关注：

1. physical writes：  
   这是所有的物理写的块数，awr中该值也会放在最前面的load profile中，  
   这个物理写甚至包括：日志切换时LGWR对数据文件头的改写
2. physical writes from cache：  
   这个就是纯粹从buffer cache中写入磁盘的脏块数
3. DBWR checkpoint buffers written  
   这个是DBWR从检查点队列写出的所有脏块数

这3个值的查询如下：

SQL> select name,VALUE from v$sysstat where name in('physical writes','physical writes from cache','DBWR checkpoint buffers written');

NAME VALUE

---------------------------------------------------------------- ----------

physical writes 69406

physical writes from cache 68073

DBWR checkpoint buffers written 67363

那么我们知道，DBWR无非就从俩个地方往磁盘上写脏块，所以：

从LRUW写到磁盘的脏块=physical writes from cache- DBWR checkpoint buffers written

### 日志切换与增量检查点

#### 几个参数

下面是几个跟增量检查点相关的参数。

SQL> show parameter checkpoint

NAME TYPE VALUE

------------------------------------ ----------- ------------------------------

log\_checkpoint\_interval integer 0

log\_checkpoint\_timeout integer 1800

log\_checkpoints\_to\_alert boolean TRUE

log\_checkpoint\_interval不说了，都默认值为0。

log\_checkpoint\_timeout就是增量检查点的时间，默认是30分钟，也就是说每隔半小时oracle将自动发起一次增量检查点。

log\_checkpoints\_to\_alert就是把检查点做事情的信息记录到alert日志中。

还有一个参数：

NAME VALUE DESCRIB

--------------------------------- ---------- -------------------------------

\_disable\_selftune\_checkpointing FALSE Disable self-tune checkpointing

#### 原理

就一个问题：日志切换的检查点是否一定是增量检查点！

答案是：不一定。

吕老师的这句话我觉得说的非常精辟：

日志切换和增量检查点的最大区别就是：没有等待的日志切换，并不会触发DBWR立即写脏块。

##### 无等待的日志切换

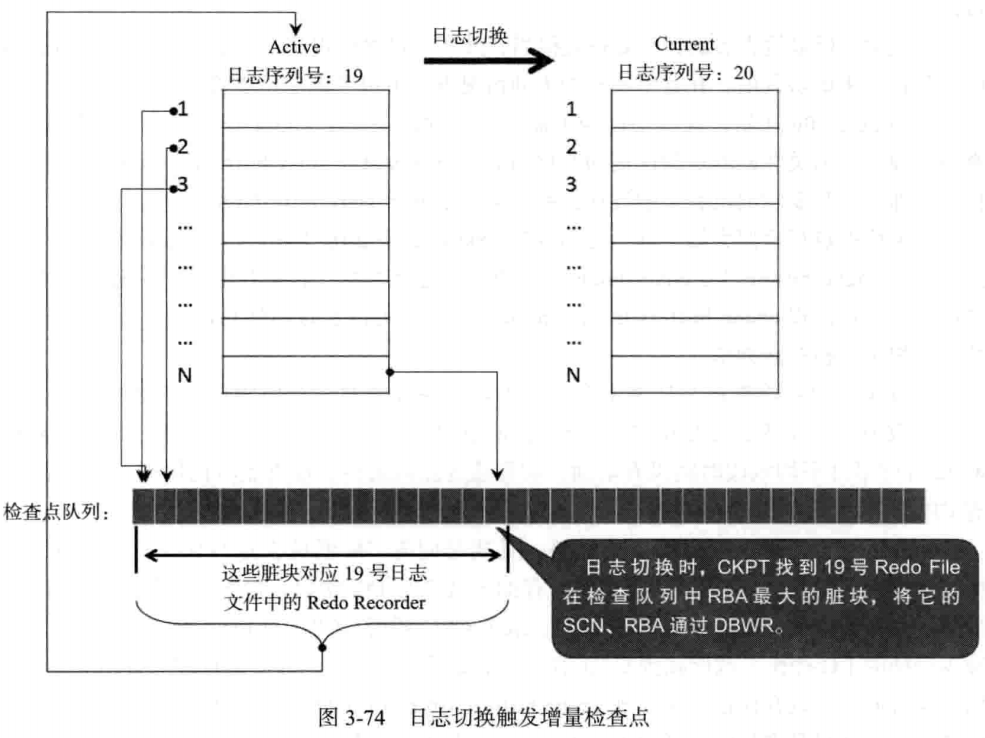
没有等待的日志切换，只会唤醒DBWR，但是DBWR写不写脏块，策略还是跟之前一致：LRUW啥也没有，CKPT-Q也不够长，那DBWR啥都不做再次睡眠。

另外，日志切换还会触发CKPT进程写数据文件头，但这个过程也不是立即执行，要等到日志文件中记录的脏buffer都写入到磁盘，才会执行。也就是说当日志文件从active变成inactive时，CKPT才会去修改数据文件头（RBA，SCN，chkpt cnt，ctl cnt）。

当发生日志切换时，LGWR通知CKPT和DBWR，然后马上开始向下一组redo写redo record，在此期间，所有的DDL、DML都会被阻塞，所以如果切换慢的话，就会产生log file switch系列等待。

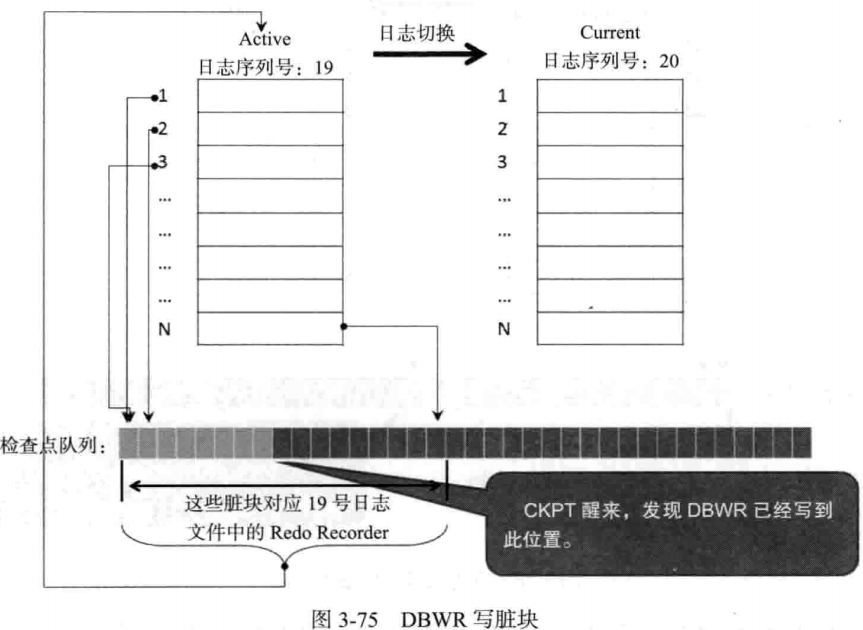
之后的事情就交给CKPT和DBWR了。

CKPT会将该redo中的RBA最大的脏buffer的SCN和RBA通知给DBWR。



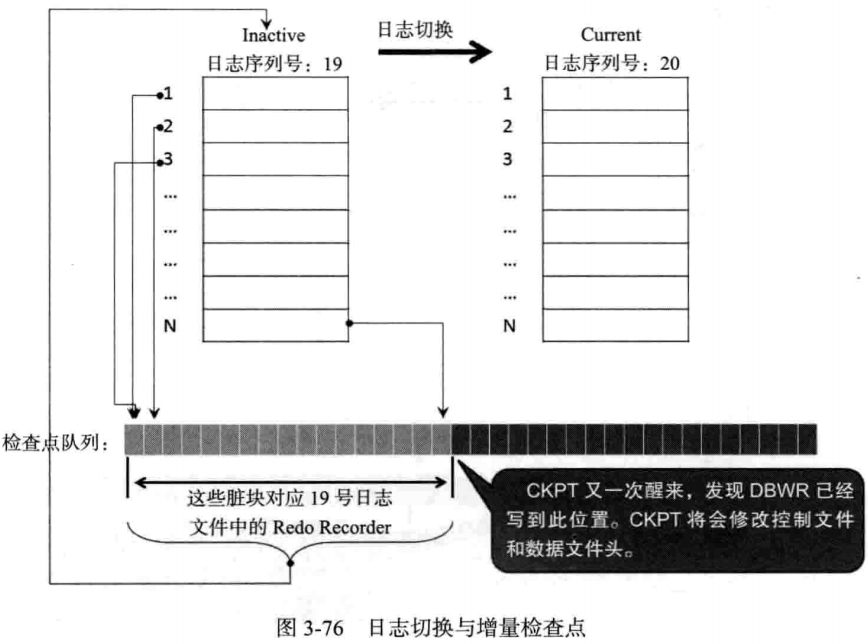
然后，DBWR醒来按正常规则写脏buffer。

CKPT会每3秒醒来一次看看DBWR写到哪儿了，如果发现DBWR还没写到之前告诉它的最大的RBA的buffer，那么CKPT就继续睡，并且redo依然保持为ACTIVE。



等到CKPT自己醒来，发现19号redo中对应的脏buffer都已经写完了，那么它会做如下3件事：

1. 修改控制文件中19号redo的状态为inactive
2. 修改控制文件中数据文件的SCN。
3. 修改数据文件头的SCN和RBA。



##### 有等待的日志切换

那么前面一直在说没有等待的日志切换，那什么叫做有等待的日志切换呢？

就是当切换日志时，发现所有的redo都是active的。

因为active的redo是不能被覆盖的，所以此时DBWR被唤醒，不管CKPT-Q有几个脏块都要写，最终效果就是所有的active redo都变成了inactive，接着才能继续进行正常日志切换。

那么在等待DBWR写完脏块的这个过程，对应的等待事件就是log file switch(checkpoint incomplete)。

出现这种等待说明：要么就是检查点间隔事件太长了，要么就是脏块产生的速度太快了。

脏块产生的速度快慢这是一个相对的说法，比如说redo很小或者redo组很少，那可能并不需要大量的脏块变化都可能会导致该等待发生。

所以合适大小的redo，合适的redo组数以及合适的增量检查点间隔🡪非常关键！

#### 实验

##### 日志切换实验

先展示一下我的参数设置：

SQL> show parameter checkpoint

NAME TYPE VALUE

------------------------------------ ----------- ------------------------------

log\_checkpoint\_interval integer 0

log\_checkpoint\_timeout integer 1800

log\_checkpoints\_to\_alert boolean TRUE

打开了对增量检查点的日志记录，增量检查点的时间间隔是默认的1800秒。

我们先看日志组状态：

11:33:32 SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 202 CURRENT

2 200 INACTIVE

3 201 INACTIVE

再查看当前物理写的一个统计信息：

NAME VALUE

---------------------------------------------------------------- ----------

physical writes 72494

physical writes from cache 70944

DBWR checkpoint buffers written 70216

然后我另个会话做个事务：

SQL> update emp set empno=empno+0;

14 rows updated.

这样这些脏buffer的变更记录就记在了202日志。

然后我们开始切日志，第一次切会很快，我们切完查看日志状态以及物理写情况：

SQL> alter system switch logfile;

System altered.

SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 202 ACTIVE

2 203 CURRENT

3 201 INACTIVE

NAME VALUE

---------------------------------------------------------------- ----------

physical writes 72494

physical writes from cache 70944

DBWR checkpoint buffers written 70216

我们看到没有发生物理写。

我们看alert日志输出：

2019-03-28T11:36:08.470685+08:00

Beginning log switch checkpoint up to RBA [0xcb.2.10], SCN: 11704476

2019-03-28T11:36:08.470810+08:00

Thread 1 advanced to log sequence 203 (LGWR switch)

Current log# 2 seq# 203 mem# 0: /u01/app/oracle/oradata/PRODCDB/redo02.log

2019-03-28T11:36:08.474701+08:00

Archived Log entry 189 added for T-1.S-202 ID 0xb09a21a0 LAD:1

然后进行第二次切换，结果如下：

SQL> alter system switch logfile;

System altered.

SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 202 ACTIVE

2 203 ACTIVE

3 204 CURRENT

NAME VALUE

---------------------------------------------------------------- ----------

physical writes 72494

physical writes from cache 70944

DBWR checkpoint buffers written 70216

2019-03-28T11:36:26.289945+08:00

Beginning log switch checkpoint up to RBA [0xcc.2.10], SCN: 11704515

2019-03-28T11:36:26.290073+08:00

Thread 1 advanced to log sequence 204 (LGWR switch)

Current log# 3 seq# 204 mem# 0: /u01/app/oracle/oradata/PRODCDB/redo03.log

2019-03-28T11:36:26.293367+08:00

Archived Log entry 190 added for T-1.S-203 ID 0xb09a21a0 LAD:1

我们看到现在没有inactive的日志组了，而且到目前为止仍没发生物理写。

那么我们进行第三次切换：

SQL> alter system switch logfile;

System altered.

SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 205 CURRENT

2 203 INACTIVE

3 204 INACTIVE

NAME VALUE

---------------------------------------------------------------- ----------

physical writes 72735

physical writes from cache 71139

DBWR checkpoint buffers written 70411

这一次切换会卡一会儿，为什么呢？

我们看告警：

2019-03-28T11:36:45.136530+08:00

Thread 1 cannot allocate new log, sequence 205

Checkpoint not complete

Current log# 3 seq# 204 mem# 0: /u01/app/oracle/oradata/PRODCDB/redo03.log

2019-03-28T11:36:47.696392+08:00

Completed checkpoint up to RBA [0xcc.2.10], SCN: 11704515

Completed checkpoint up to RBA [0xcb.2.10], SCN: 11704476

2019-03-28T11:36:47.753224+08:00

Beginning log switch checkpoint up to RBA [0xcd.2.10], SCN: 11704580

2019-03-28T11:36:47.753358+08:00

Thread 1 advanced to log sequence 205 (LGWR switch)

Current log# 1 seq# 205 mem# 0: /u01/app/oracle/oradata/PRODCDB/redo01.log

2019-03-28T11:36:47.762753+08:00

Completed checkpoint up to RBA [0xcd.2.10], SCN: 11704580

2019-03-28T11:36:47.765800+08:00

Archived Log entry 191 added for T-1.S-204 ID 0xb09a21a0 LAD:1

我们来看这个过程：

1. 发现没有inactive的redo了，所以不能分配新的序列号205。
2. 报出Checkpoint not complete，其实对应的等待事件就是：log file switch(checkpoint incomplete)
3. 然后完成0xcc、0xcb的检查点，也就是203、204 redo的检查点，说白了就是把上面记录的脏buffer都写干净了。
4. 这次开始正常分配新的sequence 205，并开始写redo。

所以这和前面概念部分讲的是完全一致的。我们也看到，确确实实DBWR checkpoint buffers written统计值增加了。

##### 增量检查点频率实验

那么为了验证，增量检查点在有脏buffer的情况下，是不是一定会写脏buffer，或者说如果把增量检查点的频率加大，active的redo很快就会自己变成inactive。

SQL> alter system switch logfile;

System altered.

SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 205 ACTIVE

2 206 CURRENT

3 204 INACTIVE

SQL> alter system set log\_checkpoint\_timeout=5;

System altered.

SQL> select group#,sequence#,status from v$log;

GROUP# SEQUENCE# STATUS

---------- ---------- ----------------

1 205 INACTIVE

2 206 CURRENT

3 204 INACTIVE

SQL> alter system set log\_checkpoint\_timeout=1800;

System altered.

我们主要看，但我们把增量检查点的时间间隔设为5秒时，alert日志中：

2019-03-28T13:21:27.375305+08:00

Incremental checkpoint up to RBA [0xcd.689a.0], current log tail at RBA [0xce.20.0]

Completed checkpoint up to RBA [0xce.2.10], SCN: 11723481

可以看到第一行增量检查点发生时记录的RBA的位置还是在205号redo，很快RBA就到了206号redo，当RBA推进到206号文件时，原来时active状态的205号redo也就变成了inactive。因为205号redo上记录的脏buffer的改变已经全部写到磁盘了。

这里我们也记录一下，如果是执行完全检查点，日志是怎么显示的呢？

SQL> alter system checkpoint;

System altered.

2019-03-28T13:29:35.266462+08:00

Beginning global checkpoint up to RBA [0xce.28d.10], SCN: 11724730

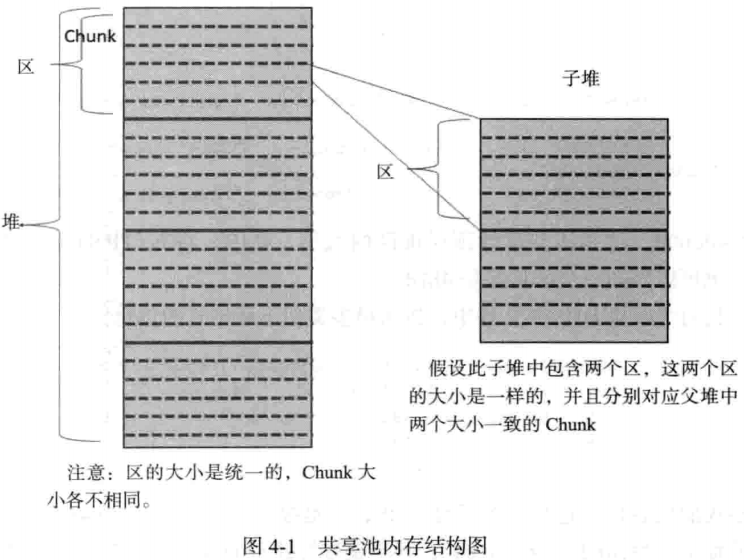
Completed checkpoint up to RBA [0xce.28d.10], SCN: 11724730

## I/O总结

# 共享池揭秘

## 共享池内存结构

### 堆、区、Chunk与子堆



如图所见，shared pool的内存中最上面一层是堆，叫做Top Heap。一个Top Heap下可以分若干个大小相等的extent，每一个extent由若干大小不等的Chunk组成。而每个Chunk又可以做为一个子堆，子堆下面依然可分若干大小相等的extent，每个extent由又若干大小不等的chunk构成，依次类推。

由上可见Chunk是共享池中最基本的内存分配单元。

当进程需要从共享池中分配内存时，有以下俩种方式：

1. 从Top Heap中分配
2. 从某个子堆中分配

这里着重说一种情况，当进程需要以第二种方式获得内存时，及从子堆A（为了方便描述，给该子堆取名为A）中分配chunk，但是子堆A中所有区的内存都用完了，那么就会触发第一种方式，及从父堆中的一个区获得一个chunk来做为子堆A的一个新区，然后再从这个新区分配进程所需要大小的chunk。

当然，如果向Top Heap索取内存时，发现不能获得足够大的内存时，就会报ORA-4031的错误。

我们来dump一下shared pool来看看它的结构：

SQL> alter session set events 'immediate trace name heapdump level 2';

Session altered.

如下时dump的一小部分结果：

…

HEAP DUMP heap name="sga heap(1,0)" desc=0x60171dc8

extent sz=0xfe0 alt=336 het=32767 rec=9 flg=0x82 opc=0

parent=(nil) owner=(nil) nex=(nil) xsz=0x1000000 heap=(nil)

fl2=0x24, nex=(nil), idx=1, dsxvers=1, dsxflg=0x0

dsx first ext=0x7c000000

dsx empty ext bytes=0 subheap rc link=0x7c0000d0,0x7c0000d0

dsx heap size=385873576, dsx heap depth=0

pdb id=1, src pdb id=1

latch set 1 of 1

durations enabled for this heap

reserved granules for root 0 (granule size 16777216)

EXTENT 0 addr=0x62000000

Chunk 062000068 sz= 48 R-stopper "reserved stoppe"

Chunk 062000098 sz= 2920 R-free " "

Chunk 062000c00 sz= 13264 R-freeable "KTSL subheap " ds=0x710d45a8

Chunk 062003fd0 sz= 48 R-stopper "reserved stoppe"

…

Chunk 062ffffd8 sz= 40 lastchk "last-type chunk"

EXTENT 1 addr=0x65000000

Chunk 065000068 sz= 48 R-stopper "reserved stoppe"

Chunk 065000098 sz= 109128 R-perm "perm " alo=109128

Chunk 06501aae0 sz= 409904 R-freeable "SO private sga " ds=0x60006c98

我们可以看到xsz=0x1000000，意思就是Heap：sga heap(1,0)中的extent的size是16M。

那么我们会注意到，sga heap(1,0)的俩个连续的extent在地址上并不连续。事实上，与该heap的EXTENT 0连续的下一个extent是sga heap(1,3)的EXTENT 1（可以通过搜索0x63000000得知）。

我们还可以dump文件中找到很多类似下面格式的信息（这里的格式跟老师书中记录的格式有一点出入，是因为我使用的版本是19c）：

Chunk 0610d32a0 sz= 4096 recrUC001 "SQLA^9196c55f " C02:51:01

ds 065636b98 sz= 12288 ct= 3

0610cd2a0 sz= 4096

0610ce2a0 sz= 4096

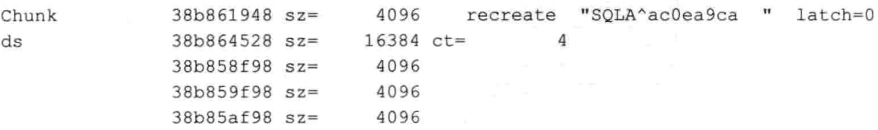
这种格式的信息，说明这是一个子堆！

DS是堆描述符，ds 065636b98是这个堆描述符所在的地址，后面的sz=12288不是堆描述的大小，而是子堆的总大小。再后面的ct=3意思就是该子堆有3个extent，我们看到子堆总大小是12k，每个extent（也就是从父堆分配的一个chunk）是4k。

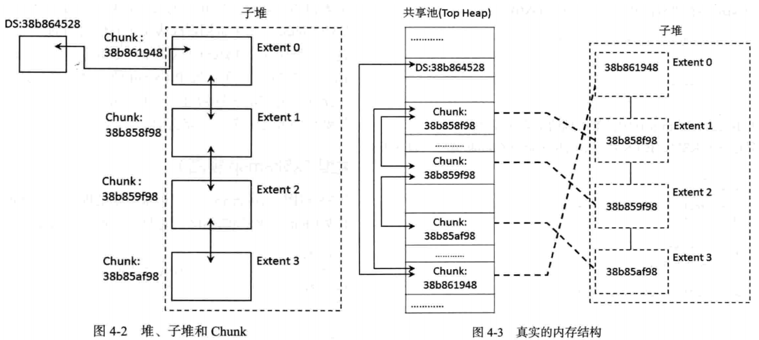
DS只是记录子堆的第一个chunk的地址，只占几百字节，并不算在子堆的空间内，说白了，DS就是子堆的一个索引，并不是子堆的一部分。

为了更好的说明他们的关系，我把吕老师的配套原图都贴一下：

吕老师的示例子堆如下：



然后根据这个子堆，吕老师的画图如下：



可以看到，子堆只是一个虚拟概念，子堆中每个extent来自父堆中的一个chunk，而这些chunk不仅在地址上不连续，在顺序上也与子堆中extent的顺序不一致。

这4个chunk互相指向，且第一个chunk与DS也互相指向，加在一起构成一个双向链表。

### Chunk类型（x$ksmsp视图）

x$ksmsp是专门记录共享池中Chunk的视图，该视图有多少行，就代表共享池中有多少Chunk。

查这个视图一定要在CDB下查！

SQL> desc x$ksmsp;

Name Type

--------- -------------

ADDR RAW(8)

INDX NUMBER

INST\_ID NUMBER

CON\_ID NUMBER

KSMCHIDX NUMBER

KSMCHDUR NUMBER

KSMCHCOM VARCHAR2(16)

KSMCHPTR RAW(8)

KSMCHSIZ NUMBER 每个Chunk的大小

KSMCHCLS VARCHAR2(8) Chunk的类型

KSMCHTYP NUMBER

KSMCHPAR RAW(8)

我们来做几个查询：

SQL> select count(\*),round(sum(KSMCHSIZ)/1024/1024,0) total\_size\_MB,round(avg(KSMCHSIZ/1024),0) avg\_size from x$ksmsp;

COUNT(\*) TOTAL\_SIZE\_MB AVG\_SIZE

---------- ------------- ----------

60948 448 8

我们看到共享池中目前有60948个chunk，总chunk大小为448M，我们来看看shared pool是不是也是448M：

SQL> select sum(bytes)/1024/1024 shared\_pool\_size from v$sgastat where pool='shared pool';

SHARED\_POOL\_SIZE

----------------

448

确实大小上是吻合的。

我们接着看chunk的种类：

SQL> select KSMCHCLS,count(\*),round(sum(KSMCHSIZ)/1024/1024,0) total\_size\_MB,round(avg(KSMCHSIZ/1024),0) avg\_size\_KB from x$ksmsp group by KSMCHCLS order by KSMCHCLS;

KSMCHCLS COUNT(\*) TOTAL\_SIZE\_MB AVG\_SIZE\_KB

-------- ---------- ------------- -----------

R-free 59 20 341

R-freea 231 3 12

R-perm 6 1 99

R-recr 4 0 7

free 6211 24 4

freeabl 27122 97 4

perm 717 255 365

recr 26382 49 2

8 rows selected.

虽然这里我们看到有8种Chunk，但是前面带R-的表示是保留池中的Chunk，所以从大的分类上说，其实就4中chunk：

1. free：空闲的Chunk  
   这种Chunk可以直接被覆盖，进程从共享池分配内存时，第一找的就是free类型的Chunk。  
   所有free的Chunk都会被放入freelist链表中。
2. perm：永久的Chunk  
   这种Chunk在示例运行期间将不会被释放或覆盖，Oracle中众多的内部内存结构以及很多X$表，占用的都是这个类型的Chunk。
3. recr：即Recreatable，可重建的Chunk
4. freeabl：可空闲、可释放、可覆盖的Chunk

下面着重理解recr和freeabl类型的chunk。

### freeabl、recr与LRU链表

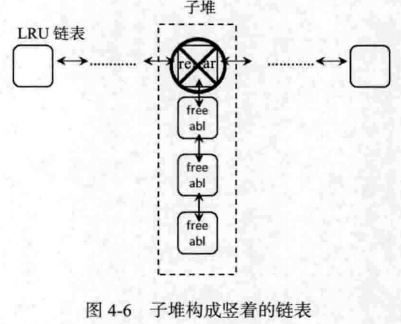
这俩种类型的Chunk，经常用与SQL。

recr型Chunk和LRU链表关系巨大，shared pool中的LRU和buffer cache中的LRU工作原理有很大不同。

在shared pool中并不是所有的chunk都在LRU上（buffer cache中所有的buffer都在LRU上），只有recr类型的chunk才在LRU上。

这里先说一个概念，老化。老化就是chunk被移到了freelist链表，也叫做被释放了。

这里我们直接来看recr、freeabl和LRU的关系：



然后我们来解释这张图：一个SQL会有多个子堆，我们单拿一个子堆A来说，进程申请了4个chunk构成了一个子堆A，当SQL在被解析和执行完成之前，子堆A的4个Chunk是不能被老化的。也就是说子堆A的这4个chunk是共进退的，那么ORACLE就只从这个子堆中选出一个chunk来作为代表放入LRU链中。

这样做的好处显而易见，LRU链表的长度将大大缩短。那么被放入LRU的Chunk就是recr类型的，其余的就是freeabl类型的。被放入LRU的这个Chunk就是子堆向父堆申请的第一个Chunk，也就是子堆的第一个extent。

当需要释放空间时，Oracle只需要将LRU链表上的这个recr的chunk老化，那么该子堆中的剩余freeabl的chunk就也可以被老化了。

freeabl类型的chunk大部分跟某个子堆的一部分，但注意recr的chunk不一定跟子堆有关系，这取决于chunk的用途。

我们可以随便找个子堆验证一下：

Chunk 0699433f8 sz= 4096 recrPC001 "KGLH0^9c998ec4 "

ds 069944460 sz= 27808 ct= 6

066c10b98 sz= 4160

066c156a0 sz= 4160

06993c198 sz= 4096

06993d198 sz= 5648

0699409c0 sz= 5648

然后我们看一下第一个chunk是不是recr：

SQL> select ksmchcls,ksmchptr from x$ksmsp where lower(ksmchptr) like '%0699433f8';

KSMCHCLS KSMCHPTR

-------- ----------------

recr 00000000699433F8

然后随便挑俩个下面的chunk再验证是不是freeabl的：

SQL> select ksmchcls,ksmchptr from x$ksmsp where lower(ksmchptr) like '%066c10b98';

KSMCHCLS KSMCHPTR

-------- ----------------

freeabl 0000000066C10B98

SQL> select ksmchcls,ksmchptr from x$ksmsp where lower(ksmchptr) like '%06993d198';

KSMCHCLS KSMCHPTR

-------- ----------------

freeabl 000000006993D198

### Free List链表

注意，只有free类型的Chunk才会再Free List中。

我们从共享池的dump中看看Free List：

FREE LISTS:

Bucket 0 size=32

Chunk 07c000090 sz= 0 kghdsx

Bucket 1 size=40

Bucket 2 size=48

Chunk 066428758 sz= 48 free " "

…

Chunk 0657c4530 sz= 48 free " "

Bucket 3 size=56

Chunk 065691e50 sz= 56 free " "

…

Chunk 065f9c520 sz= 56 free " "

Bucket 4 size=64

Chunk 0659d62f0 sz= 64 free " "

…

Bucket 253 size=32792

Chunk 067c32d98 sz= 36864 free " "

…

Bucket 254 size=65560

Chunk 067c14d98 sz= 98304 free " "

Chunk 067be3d98 sz= 122880 free " "

…

Total free space = 26431088

RESERVED FREE LISTS:

Reserved bucket 0 size=40

Reserved bucket 1 size=4400

…

Reserved bucket 13 size=65560

Chunk 061000098 sz= 689048 R-free " "

Chunk 063000098 sz= 839480 R-free " "

…

Reserved bucket 14 size=7968784

Total of this reserved free space = 4046848

UNPINNED RECREATABLE CHUNKS (lru first):

SEPARATOR

Unpinned space = 0 rcr=0 trn=0

PERMANENT CHUNKS:

Chunk 06a000068 sz= 120 perm "perm " alo=120

06a000068 sz= 120 cpmlst "kghdsx "

Permanent space = 120

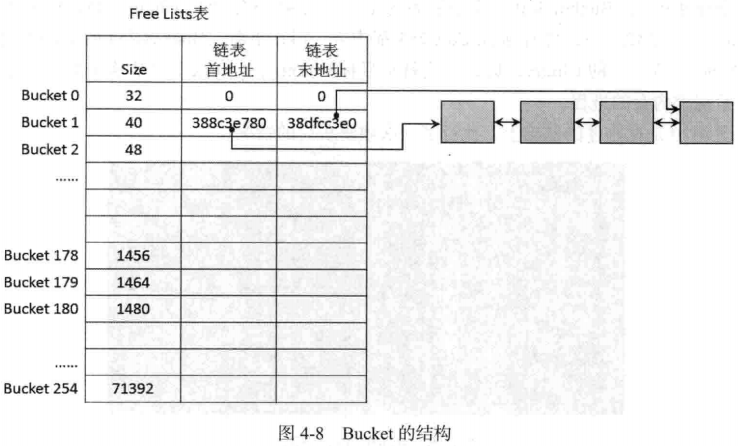
我们可以很明显的看到，Free List一共有255个bucket（0-254），每个bucket保存一个链表，用于将bucket对应大小的chunk“串”在一起。

bucket后面的size就是这个bucket中的chunk的下限，比如bucket 253里放的就是大等32792且小于65560的chunk。

我们可以看到，开始是8字节一个范围的bucket，到后面一个bucket包含的chunk size越来越来大。

Freelist的bucket在64位系统中占192个字节，也就是192/64=3个字，一个字保存大小，另俩个字保存链表的首尾俩个node的地址。

具体如下：



Freelist占用的内存并不多，固定的255个bucket，共占用255\*192=48960字节的内存，且这部分内存在SGA的Fixed Area中分配。

下面来说FreeList的工作原理。

假如进程需要4096字节的chunk，那么它会到Bucket 240里面去找，发现正好有4096字节大小的chunk，那么就直接拿来用，同时把这个chunk从Free List中移除。

但是如果Bucket 240为空，如下：

Bucket 240 size=4096

Bucket 241 size=4104

Bucket 242 size=4120

Chunk 067bc8d98 sz= 8192 free " "

Chunk 063356040 sz= 8192 free " "

那么就再往下找，241仍然没有，发现242有，那么就把这个8192字节的chunk切一个4096的chunk出来给请求内存的进程。

那么切剩下的就重新挂到对应的bucket中，比如这个8192的chunk被切的就剩4096了，那么它会被重新挂到240号bucket。

这样的切分chunk当然有可能产生很多碎片，那么俩个相邻的Chunk会发生合并么？答案是不会！因为Oracle并不知道FreeList中哪些Chunk是相邻的。

### 保留池

保留池相关的参数如下：

NAME VALUE DESCRIB

------------------------ ---------- ------------------------------------------------------------

shared\_pool\_reserved\_size 23488102 size in bytes of reserved area of shared pool

\_shared\_pool\_reserved\_pct 5 percentage memory of the shared pool allocated for the reserved area

\_shared\_pool\_reserved\_min\_alloc 4400 minimum allocation size in bytes for reserved area of shared pool

头俩个参数的意思很明显，就是用来规定保留池大小的，19c中，我们看到我的实验环境默认值是shared pool的百分之5，我们前面已知我们的shared pool大小为448M：

SQL> select 448\*0.05 from dual;

448\*0.05

----------

22.4

SQL> select 23488102/1024/1024 from dual;

23488102/1024/1024

------------------

22.3999996

验证一下，是这样的。

前面类型开头是R-的chunk就是保留池中的Chunk。保留池的目的就是为了当进程申请大的chunk时，可以从保留池中直接获得。

\_shared\_pool\_reserved\_min\_alloc就是定义从保留池最小分配的chunk的大小，默认时4400字节。

我们把前面的dump再粘一遍：

RESERVED FREE LISTS:

Reserved bucket 0 size=40

Reserved bucket 1 size=4400

…

Reserved bucket 13 size=65560

Chunk 061000098 sz= 689048 R-free " "

Chunk 063000098 sz= 839480 R-free " "

…

Reserved bucket 14 size=7968784

Total of this reserved free space = 4046848

我们看到bucket 0是空的，bucket 1就是从4400开始的。

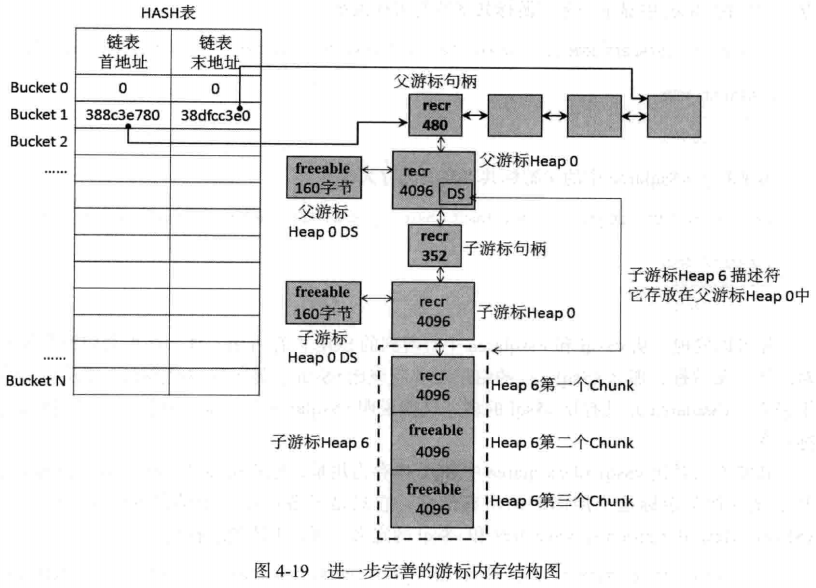
但是并不是只要申请大于4400字节的chunk就直接到保留池中找，Oracle依然会先从free list中找是否能给申请进程要求的大小的chunk，如果没有，才从保留池中找。所以\_shared\_pool\_reserved\_min\_alloc只是定义了保留池中最小的chunk从多少字节开始。

### SQL的内存结构

#### 父游标、子游标

父子游表的设计跟recr和freeabl的设计类似，就是为了缩短链表的长度，提高检索的效率。

我直接把终极结构图先放上：



#### 父游标句柄

父游标句柄对应一个独立的chunk，其类型是recr，它的地址可以在x$kglob中KGLHDADR中找到。

父游标句柄对应一个独立的chunk，它的类型为recr，我们来做实验找一下父游标。

我们先执行一条sql：

select \* from emp where empno=7788;

然后在x$kglob中找到它：

SQL> select kglhdadr,kglnaobj from x$kglob where kglnaobj like 'select \* from emp where empno=7788' and kglhdadr=kglhdpar;

KGLHDADR KGLNAOBJ

---------------- ----------------------------------------

0000000065C84700 select \* from emp where empno=7788

然后，我们来做一个2050级的dump：

SQL> alter session set events 'immediate trace name heapdump level 2050';

Session altered.

这个导出文件就大了，我的实验环境导出是1.4g的trace。

然后我们在文件里搜065C84，只搜这么点，是因为more一个文件的时候，会把中间的都skip，搜的太细（比如065C84700，而且也不一定就有这个地址）就不方便阅读了。

Chunk 065c846d0 sz= 816 recrPT003 "KGLHD "

Dump of memory from 0x0000000065C846D0 to 0x0000000065C84A00

065C846D0 00000331 80B38F00 65C844A0 00000000 [1........D.e....]

065C846E0 00000000 00000000 00000000 00000000 [................]

065C846F0 00000000 00000000 00000003 00080050 [............P...]

065C84700 6ED5EF70 00000000 6ED5EF70 00000000 [p..n....p..n....]

…

065C848C0 656C6573 2A207463 6F726620 6D65206D [select \* from em]

065C848D0 68772070 20657265 6E706D65 37373D6F [p where empno=77]

065C848E0 00003838 00000000 00000000 00000000 [88..............]

065C848F0 00000000 00000000 00000000 00000000 [................]

Repeat 16 times

我们看到65C84700这个地址后16个字节的内容，虽然看不出具体是啥。

我们可以看最前面的内容，就是说我们这条sql的父游标句柄在65C846D0（一定要转成大写方便在x$ksmsp中查询）这个Chunk，大小为816字节。

我们知道每一个chunk都在x$ksmsp中可以查到，所以我们查一下：

SQL> select KSMCHCLS,KSMCHPTR,KSMCHSIZ from x$ksmsp where KSMCHPTR like '%65C846D0';

KSMCHCLS KSMCHPTR KSMCHSIZ

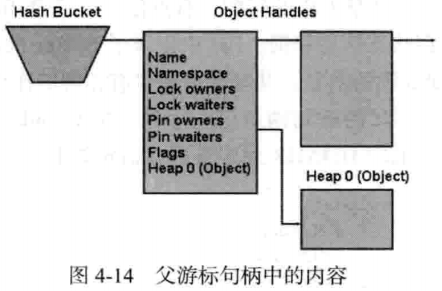
-------- ---------------- ----------

recr 0000000065C846D0 816

确实找到了。

当然我们知道，父游标句柄里有很多内容，包括我们从上面的dump中确实看到，sql文本确实是保存在父游标句柄中的。

简单的概括如下这副名图：

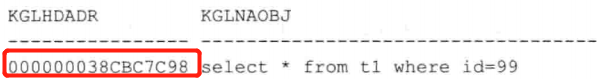


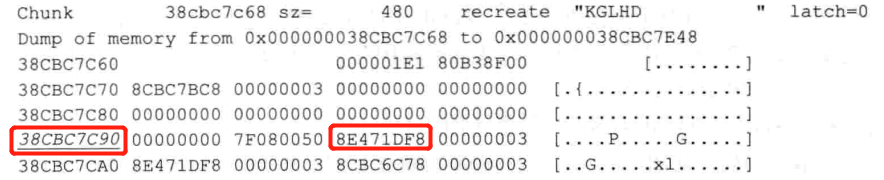
其实我们想要了解父游标句柄中有那些东西，应该做如下查询：

select \* from x$kglob where kglnaobj like 'select \* from emp where empno=7788' and kglhdadr=kglhdpar;

因为x$kglob的信息就来自于句柄。当然也并不是包含了句柄的所有信息，比如Library Cache Lock/Pin的信息。

这里为了方便贴图，所以把吕老师的示例也贴出来：

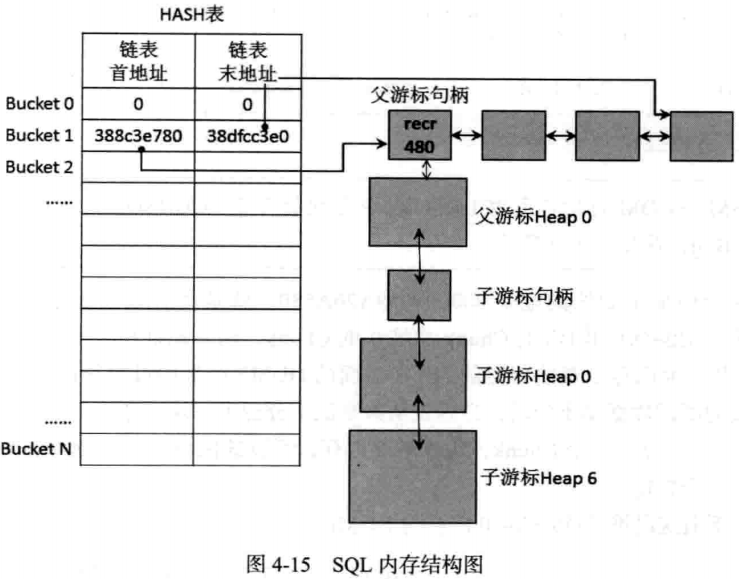




吕老师这个示例中，如果直接搜7C98就搜不到了，这里显然对应的地址应该是第四行（38CBC7C90）的第一个字节算起，向后偏移8个字节。

也可以看到，吕老师这个示例的sql的父游标句柄是一个480字节的Chunk。

那么因此得到第一个简图：



我们接着一步一步把该图补充完整（当然吕老师的图中这个地址也没对应上他自己的示例）。

#### SQL的Chunk：父游标堆0和DS

父游标句柄一个非常重要的内容就是指向父游标堆0的地址。

通过如下方式从x$kglob中得到父游标堆0的“地址”：

SQL> select kglhdadr,kglnaobj,kglobhd0 from x$kglob where kglnaobj like 'select \* from emp where empno=7788' and kglhdadr=kglhdpar;

KGLHDADR KGLNAOBJ KGLOBHD0

---------------- ---------------------------------------- ----------------

0000000065C84700 select \* from emp where empno=7788 0000000065F85510

事实上，这个查询结果只是父游标堆0的DS的地址。而非真正堆0的地址。

需要通过如下查询再查一手，也就是到x$ksmsp中查：

SQL> select KSMCHPTR,KSMCHPAR,KSMCHCOM,KSMCHCLS,KSMCHSIZ from x$ksmsp where KSMCHPAR='0000000065F85510';

KSMCHPTR KSMCHPAR KSMCHCOM KSMCHCLS KSMCHSIZ

---------------- ---------------- ---------------- -------- ----------

00000000982F3E18 0000000065F85510 KGLH0^4015a79a recr 4096

这个982F3E18就是父游标堆0真正的chunk了。

我们从dump中找一下堆0的Chunk和它的DS所在的chunk.

这是堆0的，可以看到这是个recr类型的Chunk，大小为4KB（这个大小应该是固定的）：

Chunk 0982f3e18 sz= 4096 recrPC003 "KGLH0^4015a79a "

ds 065f85510 sz= 4096 ct= 1

Dump of memory from 0x00000000982F3E18 to 0x00000000982F4E18

0982F3E10 00001001 80B38F00 [........]

0982F3E20 982F3AE8 00000000 00000000 00000000 [.:/.............]

0982F3E30 00000000 00000000 00000000 00000000 [................]

0982F3E40 00000003 000A0FFF 65F85510 00000000 [.........U.e....]

…

这是DS所在的Chunk，这里要搜065F85：

Chunk 065f854a8 sz= 504 freeable "KGLDA "

Dump of memory from 0x0000000065F854A8 to 0x0000000065F856A0

065F854A0 000001F9 00B38F00 [........]

…

065F85510 60170470 00000000 00000000 00000000 [p..`............]

065F85520 65F854C0 00000000 982F3E48 00000000 [.T.e....H>/.....]

065F85530 982F3ED0 00000000 00000000 00000000 [.>/.............]

065F85540 00000000 00000000 000C0209 0000FFFF [................]

065F85550 00000000 00000000 00000FD0 484C474B [............KGLH]

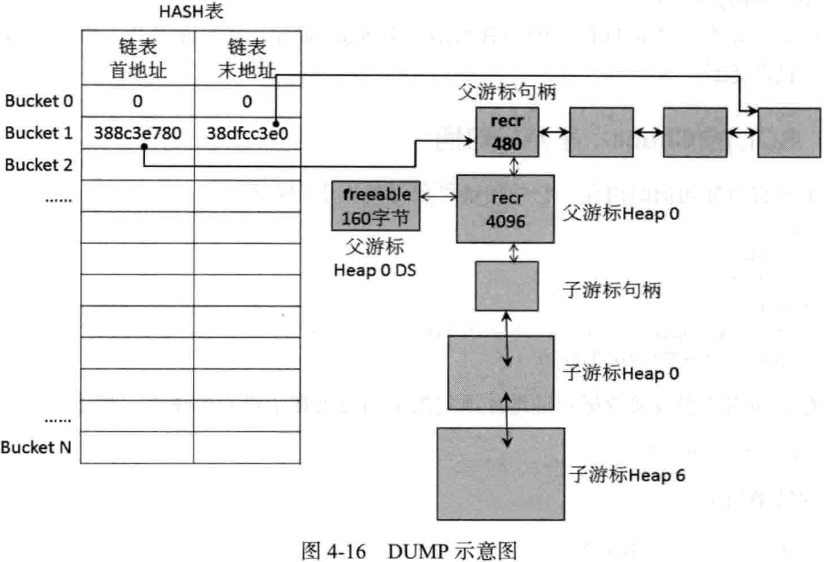
065F85560 30345E30 37613531 26006139 00387FFF [0^4015a79a.&..8.]

…

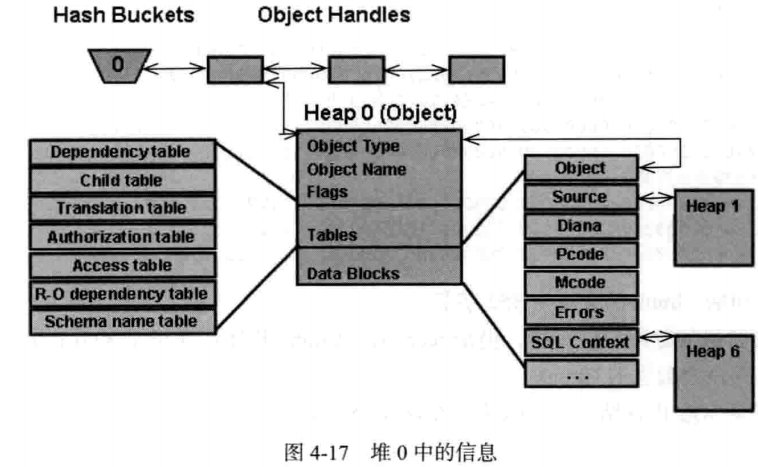
我们可以看到DS所在的Chunk和父游标堆0的Chunk互相指向。而且，DS所在的Chunk是一个freeabl类型的chunk。

我们注意下划线那个982f3e48，如果它是个地址的话，我们可以看一下，堆0的Chunk的内容里的982f3e48那个字节开始往后的内容，正是堆0中记录DS地址的地址。（这个是我个人猜测，因为我并没有像吕老师的示例那样在DS的chunk中找到堆0的地址）。

那么我们继续补完上一个图，依然数据是吕老师的数据，比如我们的DS所在的chunk就是504字节，而不是160字节：



再看一副名画：



虽然该图很全面，但是对于父游标的堆0来说，有一个内容它没有展示（因为这是所有堆0的一个总概），那就是子游标的句柄地址。

#### SQL的Chunk：子游标句柄

我们看如下这个查询：

SQL> select kglhdadr,kglhdpar,kglobhd0 from x$kglob where kglnaobj = 'select \* from emp where empno=7788';

KGLHDADR KGLHDPAR KGLOBHD0

---------------- ---------------- ----------------

0000000098C4F168 0000000065C84700 0000000069C39220

0000000065C84700 0000000065C84700 0000000065F85510

前面我们通过ADR=PAR得到父游标句柄的地址，那么另一条ADR<>PAR的ADR就是子游标句柄的地址。

还是一样的办法，我们来搜098C4F1，然后慢慢找：

Chunk 098c4f138 sz= 560 recrPT003 "KGLHD "

Dump of memory from 0x0000000098C4F138 to 0x0000000098C4F368

098C4F130 00000231 80B38F00 [1.......]

098C4F140 98C4F038 00000000 00000000 00000000 [8...............]

098C4F150 00000000 00000000 00000000 00000000 [................]

098C4F160 00000003 00080050 98C4F168 00000000 [....P...h.......]

我们从98C4F160往后偏移8个字节就是98C4F168，这里注意不要混淆，第一列是第二列第一个字节的地址，后面4大列（也就是16个字节，都是真实的内存中的内容）。

我们看到，子游标句柄所在的Chunk也是recr型的，Chunk描述仍然是KGLHD，大小为560字节。

它是一个独立的Chunk，不属于任何子堆。

我们拿个这个地址去父游标堆0里面找一找，确实找到了：

Chunk 0982f3e18 sz= 4096 recrPC003 "KGLH0^4015a79a "

ds 065f85510 sz= 4096 ct= 1

Dump of memory from 0x00000000982F3E18 to 0x00000000982F4E18

0982F3E10 00001001 80B38F00 [........]

0982F3E20 982F3AE8 00000000 00000000 00000000 [.:/.............]

0982F3E30 00000000 00000000 00000000 00000000 [................]

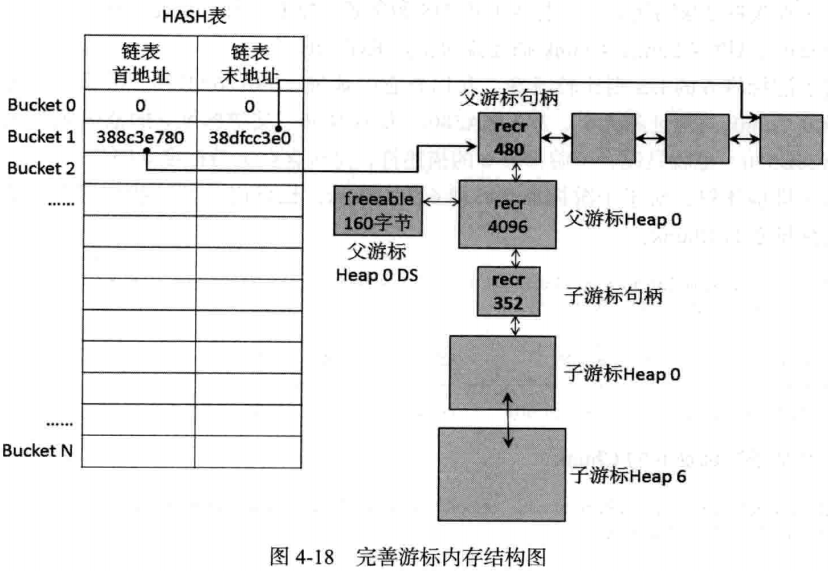
0982F3E40 00000003 000A0FFF 65F85510 00000000 [.........U.e....]

…

0982F4860 98C4F168 00000000 65C84700 00000000 [h........G.e....]

…

继续补充之前的图：



#### SQL的Chunk：子游标堆0和堆6

再接下来就是子游标的堆0和堆6。我们先用sql找到它们：

SQL> select kglobhd0,kglobhd6 from x$kglob where kglnaobj = 'select \* from emp where empno=7788' and kglhdadr<>kglhdpar;

KGLOBHD0 KGLOBHD6

---------------- ----------------

0000000069C39220 00000000982F4600

这依然是堆0和堆6的DS地址。

那么它们真实的地址同理：

SQL> select KSMCHPTR,KSMCHPAR,KSMCHCOM,KSMCHCLS,KSMCHSIZ from x$ksmsp where KSMCHPAR='0000000069C39220';

KSMCHPTR KSMCHPAR KSMCHCOM KSMCHCLS KSMCHSIZ

---------------- ---------------- ---------------- -------- ----------

0000000098C744E0 0000000069C39220 KGLH0^4015a79a recr 4096

SQL> select KSMCHPTR,KSMCHPAR,KSMCHCOM,KSMCHCLS,KSMCHSIZ from x$ksmsp where KSMCHPAR='00000000982F4600';

KSMCHPTR KSMCHPAR KSMCHCOM KSMCHCLS KSMCHSIZ

---------------- ---------------- ---------------- -------- ----------

000000006AB179D8 00000000982F4600 SQLA^4015a79a recr 4096

000000006AB169D8 00000000982F4600 SQLA^4015a79a freeabl 4096

000000006AAE50F0 00000000982F4600 SQLA^4015a79a freeabl 4096

000000006AAE40F0 00000000982F4600 SQLA^4015a79a freeabl 4096

000000006AAE30F0 00000000982F4600 SQLA^4015a79a freeabl 4096

我们看到，子游标堆6有5个chunk，每个chunk大小为4KB。

我们到dump里面找一找，先找堆0的chunk：

Chunk 098c744e0 sz= 4096 recrPC003 "KGLH0^4015a79a "

ds 069c39220 sz= 4096 ct= 1

Dump of memory from 0x0000000098C744E0 to 0x0000000098C754E0

098C744E0 00001001 80B38F00 98C740B8 00000000 [.........@......]

098C744F0 00000000 00000000 00000000 00000000 [................]

098C74500 00000000 00000000 00000003 000A0FFF [................]

098C74510 69C39220 00000000 00000000 00000000 [ ..i............]

…

098C746D0 00000008 00000017 00000017 544F4353 [............SCOT]

098C746E0 00000054 00000000 00000000 00000000 [T...............]

我们注意到子游标堆0里我们看到了用户名。

堆0的DS所在的Chunk：

Chunk 069c391b8 sz= 504 freeable "KGLDA "

Dump of memory from 0x0000000069C391B8 to 0x0000000069C393B0

069C391B0 000001F9 00B38F00 [........]

069C391C0 69C39080 00000000 12B1AF44 00000000 [...i....D.......]

…

069C39220 60170470 00000000 00000000 00000000 [p..`............]

069C39230 69C391D0 00000000 98C74510 00000000 [...i.....E......]

…

069C39260 00000000 00000000 00000FD0 484C474B [............KGLH]

069C39270 30345E30 37613531 26006139 00387FFF [0^4015a79a.&..8.]

然后我们看堆6的chunk：

Chunk 06ab179d8 sz= 4096 recrUC003 "SQLA^4015a79a " A01:31:30

ds 0982f4600 sz= 20480 ct= 5

06aae30f0 sz= 4096

06aae40f0 sz= 4096

06aae50f0 sz= 4096

06ab169d8 sz= 4096

Dump of memory from 0x000000006AB179D8 to 0x000000006AB189D8

06AB179D0 00001001 80B38F00 [........]

06AB179E0 6AB169D8 00000000 982F3AF8 00000000 [.i.j.....:/.....]

06AB179F0 6017A6B0 00000000 6825BE40 00000000 [...`....@.%h....]

06AB17A00 00000003 00020FFF 982F4600 00000000 [.........F/.....]

…

06AB17E00 00000000 4D450003 00000050 00000000 [......EMP.......]

06AB17E10 00000029 00B38F00 6AB17DE8 00000000 [)........}.j....]

06AB17E20 14B3CE44 00000000 00000000 43530005 [D.............SC]

06AB17E30 0054544F 00000000 00000029 00B38F00 [OTT.....).......]

…

Chunk 06aae30f0 sz= 4096 freeableU "SQLA^4015a79a " ds=0x982f4600

Dump of memory from 0x000000006AAE30F0 to 0x000000006AAE40F0

06AAE30F0 00001001 00B38F00 6AAE20F0 00000000 [......... .j....]

06AAE3100 601766D0 00000000 982F4600 00000000 [.f.`.....F/.....]

…

06AAE3170 14AB9738 00000000 5F4B5006 05504D45 [8........PK\_EMP.]

06AAE3180 4E504D45 4553054F 0331244C 05504D45 [EMPNO.SEL$1.EMP.]

06AAE3190 544F4353 00000054 00000029 00B38F00 [SCOTT...).......]

…

Chunk 06aae40f0 sz= 4096 freeableU "SQLA^4015a79a " ds=0x982f4600

Dump of memory from 0x000000006AAE40F0 to 0x000000006AAE50F0

06AAE40F0 00001001 00B38F00 6AAE30F0 00000000 [.........0.j....]

06AAE4100 601766D0 00000000 982F4600 00000000 [.f.`.....F/.....]

…

06AAE4A50 14B3CE44 00000000 00000007 45440006 [D.............DE]

06AAE4A60 4F4E5450 00000000 00000039 00B38F00 [PTNO....9.......]

…

06AAE4B30 14B3CE44 00000000 00000007 4F430004 [D.............CO]

06AAE4B40 00004D4D 00000000 00000039 00B38F00 [MM......9.......]

…

06AAE4CF0 14B3CE44 00000000 00000007 49480008 [D.............HI]

06AAE4D00 41444552 00004554 00000039 00B38F00 [REDATE..9.......]

…

Chunk 06aae50f0 sz= 4096 freeableU "SQLA^4015a79a " ds=0x982f4600

Dump of memory from 0x000000006AAE50F0 to 0x000000006AAE60F0

06AAE50F0 00001001 00B38F00 6AAE40F0 00000000 [.........@.j....]

06AAE5100 601766D0 00000000 982F4600 00000000 [.f.`.....F/.....]

…

Chunk 06ab169d8 sz= 4096 freeableU "SQLA^4015a79a " ds=0x982f4600

Dump of memory from 0x000000006AB169D8 to 0x000000006AB179D8

06AB169D0 00001001 00B38F00 [........]

06AB169E0 6AB15BD8 00000000 601766D0 00000000 [.[.j.....f.`....]

06AB169F0 982F4600 00000000 6AB17A08 00000000 [.F/......z.j....]

…

从上面的dump我们可以看到：

1. 出了第一个extent是recr的chunk，其余都是freeabl的chunk
2. freeabl的chunk也记录着ds的地址，都是在第24字节的偏移量处
3. 第二个extent和第三个extent可以明显看出跟执行计划有关，分别是索引扫描和回表（我估计这个回表是project information那部分）。
4. 我们看到子游标堆0和堆6中都会有用户名，而父游标中则不会有，所以不同用户下的相同sqltext的sql会在一个父游标下的不同子游标中。
5. 那么子游标堆6中，只有第一个exten/chunk会被链接到LRU中。

下面我们来看子游标堆6的DS所在的chunk，它是有点儿特殊的：

Chunk 0982f3e18 sz= 4096 recrPC003 "KGLH0^4015a79a "

ds 065f85510 sz= 4096 ct= 1

Dump of memory from 0x00000000982F3E18 to 0x00000000982F4E18

0982F3E10 00001001 80B38F00 [........]

…

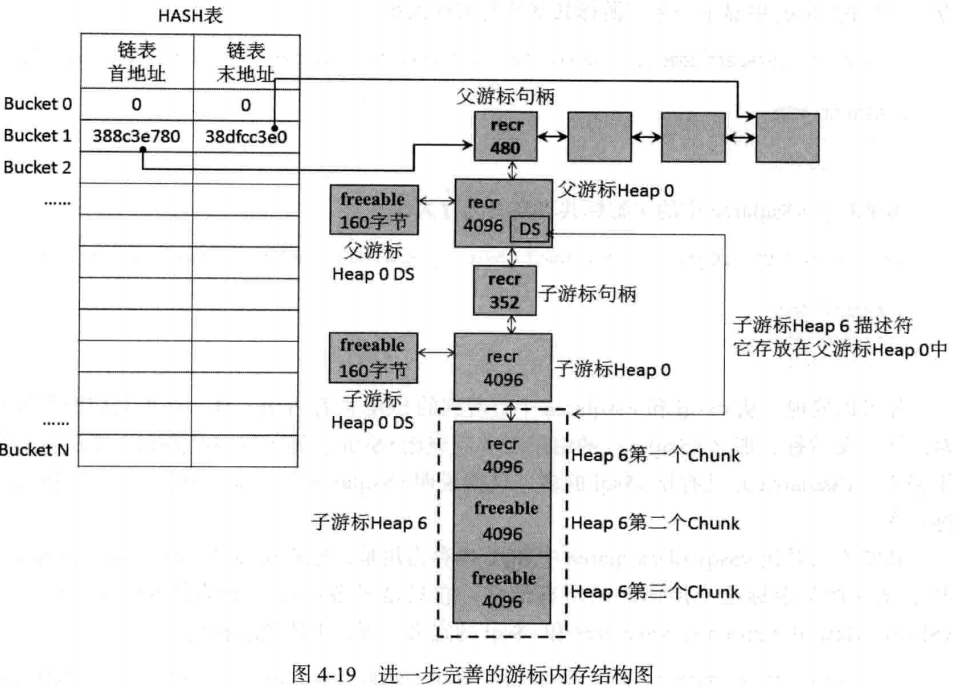
0982F4600 60170470 00000000 00000000 00000000 [p..`............]

…

我们看这个Chunk是不是有点儿眼熟，没错，就是父游标的堆0。

所以子游标堆6的DS所在的chunk正是父游标的堆0，而不会单独占一个chunk。

最后，我们把这个图完成：



这里注意，我并没有找到父/子游标句柄指向父/子游标堆0的指针，这个后面空了可以再仔细研究下。

#### SQL所占共享池内存

我们看，我们的实验sql一共有11个Chunk，父游标：816+4096+504=5416字节，子游标：560+4096+504+4096\*5=25640字节，一共31056字节。

我们来看下v$sql和v$sqlarea的SHARABLE\_MEM：

SQL> select SHARABLE\_MEM from v$sqlarea where sql\_text='select \* from emp where empno=7788';

SHARABLE\_MEM

------------

27499

SQL> select SHARABLE\_MEM from v$sql where sql\_text='select \* from emp where empno=7788';

SHARABLE\_MEM

------------

27499

我们看到，这俩值一样，都比我们实际算的小。因为这个值其实是子游标占的内存大小。

v$sqlarea中这个值是所有子游标占内存的总大小，而v$sql中当然就是各个子游标各自占的内存大小。

比如我们在另一个用户下仍然执行这条sql，来看下：

SQL> select SHARABLE\_MEM from v$sqlarea where sql\_text='select \* from emp where empno=7788';

SHARABLE\_MEM

------------

48651

SQL> select SHARABLE\_MEM from v$sql where sql\_text='select \* from emp where empno=7788';

SHARABLE\_MEM

------------

28419

24363

结果一目了然就不多说了。

其实想要知道父游标和子游标占的内存大小，都可以从x$kglob中获得：

SQL> select KGLOBHS0+KGLOBHS6+KGLOBT16 from x$kglob where kglnaobj = 'select \* from emp where empno=7788' and kglhdadr<>kglhdpar;

KGLOBHS0+KGLOBHS6+KGLOBT16

--------------------------

28419

24363

SQL> select KGLOBHS0+KGLOBHS6+KGLOBT16 from x$kglob where kglnaobj = 'select \* from emp where empno=7788' and kglhdadr=kglhdpar;

KGLOBHS0+KGLOBHS6+KGLOBT16

--------------------------

4072

可以看到，子游标算出来的大小跟v$sql中是一样的。所以想算总的一条sql（父+子）总共占的内存大小，把最后的where条件去掉就行。

### LRU链表：我的共享池大了还是小了

#### 判断共享池过大

共享池的LRU链从总体上分为俩条：瞬时LRU、周期LRU。

一个Chunk第一次被操作后会被放入瞬时LRU，如果又一次被用到了，就会被放到周期LRU。

我执行一条实验sql，是硬解析，然后我们定位它的堆0的chunk地址：

SQL> select kglobhd0,kglobhd6 from x$kglob where kglnaobj = 'select \* from dept' and kglhdadr<>kglhdpar;

KGLOBHD0 KGLOBHD6

---------------- ----------------

00000000626F28F0 0000000062B37ED8

SQL> select KSMCHPTR,KSMCHPAR,KSMCHCOM,KSMCHCLS,KSMCHSIZ from x$ksmsp where KSMCHPAR='00000000626F28F0';

KSMCHPTR KSMCHPAR KSMCHCOM KSMCHCLS KSMCHSIZ

---------------- ---------------- ---------------- -------- ----------

0000000069706D30 00000000626F28F0 KGLH0^d68ee9be recr 4096

然后我们做一个level 2的heapdump：

alter session set events 'immediate trace name heapdump level 2';

观察dump，用vi看，使用:set number打开行数：

62561 Chunk 069706d30 sz= 4096 recrPC003 "KGLH0^d68ee9be "

62562 ds 0626f28f0 sz= 4096 ct= 1

…

100180 SEPARATOR

100181 Chunk 0669e9580 sz= 4096 recrUR003 "KGLH0^a9720066 " B04:21:48

100182 hits= 99

100183 Chunk 069648c68 sz= 4096 recrUR001 "KGLH0^d19000c0 " B04:06:12

dump中的第一个SEPARATOR就是瞬时和周期LRU的分界线。

可以看到现在chunk在瞬时LRU上。

我们再执行一遍sql：

62380 Chunk 069706d30 sz= 4096 recrPR003 "KGLH0^d68ee9be " B04:32:01

62381 hits= 2

…

99803 SEPARATOR

…

118207 Chunk 069706d30 sz= 4096 recrPR003 "KGLH0^d68ee9be " B04:32:01

118208 hits= 2

我们看到，周期LRU上有该Chunk了，但是注意，瞬时LRU上该Chunk还在。（实验环境19c）。我们可以看到它有个hits，经过测试，这个不是简单的执行过的次数，可能原理跟buffer header的TCH有点儿类似。这里暂不做深究。

Oracle有一个视图可以查看瞬时LRU和周期LRU中的Chunk数—x$KGHLU：

SQL> select KGHLUTRN,KGHLURCR from x$kghlu;

KGHLUTRN KGHLURCR

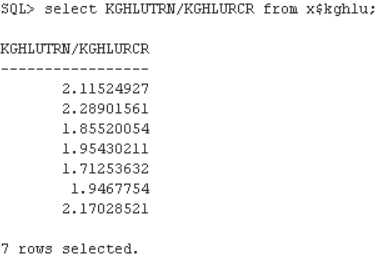
---------- ----------

10169 9260

KGHLUTRN是瞬时LRU，KGHLURCR是周期LRU。

KGHLUTRN如果超过KGHLURCR的3倍，说明大量Chunk是只用过一次的，所以说共享池太大了。

如下是某人社生产库的数据：



#### 判断共享池太小

SQL> select KGHLUFSH,KGHLUOPS from x$kghlu;

KGHLUFSH KGHLUOPS

---------- ----------

273184 861316

SQL> /

KGHLUFSH KGHLUOPS

---------- ----------

273184 861367

SQL> alter system flush shared\_pool;

System altered.

SQL> select KGHLUFSH,KGHLUOPS from x$kghlu;

KGHLUFSH KGHLUOPS

---------- ----------

298820 909055

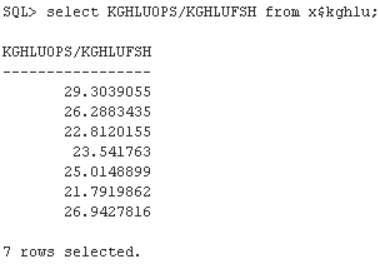
注意上面查询数据的变化。

KGHLUFSH是进程flush shared pool的次数。Oracle觉得共享池内存紧张时，会自己flush shared pool。

当Pin Chunk或者释放Chunk Pin时，KGHLUOPS就会增加，所以可以看作进程是操作chunk的次数。

如果KGHLUOPS/ KGHLUFSH<20，那说明共享池小了，也就是说操作不到20次chunk，就要进行一次flush是不合适的。

如下是某人社生产库的数据：



### ORA-4031：错误的报错信息

#### 判断共享池设置大小是否合适

参考4.1.7。

#### 判断共享池对象是否过多

使用如下查询：

col KGLHDNSD for a30

col KGLOBTYD for a30

select kglhdnsd,kglobtyd,count(\*),round(sum(kglobhs0+kglobhs1+kglobhs2+kglobhs3+kglobhs4+kglobhs5+kglobhs6+kglobt16)/1024) M from x$kglob group by kglhdnsd,kglobtyd order by count(\*) desc;

KGLHDNSD KGLOBTYD COUNT(\*) M

------------------------------ ------------------------------ ---------- ----------

SQL AREA CURSOR 1773 21421

SQL AREA STATS CURSOR STATS 565 2247

TABLE/PROCEDURE TABLE 455 1423

SQL AREA BUILD CURSOR 424 0

TABLE/PROCEDURE VIEW 169 231

INDEX INDEX 128 91

TABLE/PROCEDURE PACKAGE 108 4189

RUNTIME STATISTICS RUNTIME STATISTICS 60 974

BODY PACKAGE BODY 51 3087

TABLE/PROCEDURE TYPE 38 8

OPTIMIZER EXPRESSION HEADER Optimizer Expression Header 33 130

OPTIMIZER DIRECTIVE OWNER CURSOR 30 0

CLUSTER CLUSTER 22 87

TABLE/PROCEDURE SEQUENCE 17 8

TABLE/PROCEDURE SYNONYM 15 8

TABLE/PROCEDURE CURSOR 9 0

TABLE/PROCEDURE LIBRARY 9 8

MULTI-VERSION OBJECT FOR TABLE MULTI-VERSIONED OBJECT 8 57

SCHEMA NONE 6 0

MULTI-VERSION OBJECT FOR INDEX MULTI-VERSIONED OBJECT 5 8

BODY TYPE BODY 5 0

OPTIMIZER DIRECTIVE OWNER Optimizer Directive Owner 4 16

OPTIMIZER FINDING Optimizer Finding 4 16

TABLE/PROCEDURE SCHEDULER JOB 3 12

TABLE/PROCEDURE FUNCTION 3 0

DBINSTANCE CURSOR 3 0

PUB SUB INTERNAL INFORMATION PUB SUB INTERNAL INFORMATION 3 8

OPTIMIZER EXPRESSION OBJECT Optimizer Expresion Object 2 8

SCHEDULER GLOBAL ATTRIBUTE SCHEDULER GLOBAL ATTRIBUTE 2 8

AUDIT POLICY AUDIT POLICY 2 8

QUEUE QUEUE 2 0

TABLE/PROCEDURE OPERATOR 2 0

EDITION EDITION 1 4

FED APP FED APP 1 4

DBLINK CURSOR 1 0

RULESET RULESET 1 0

OPTIMIZER EXPRESSION OBJECT CURSOR 1 0

SCHEDULER GLOBAL ATTRIBUTE CURSOR 1 0

TRIGGER TRIGGER 1 6

BODY CURSOR 1 0

TABLE/PROCEDURE SCHEDULER CLASS 1 4

41 rows selected.

#### 判断碎片是否太多

使用如下查询：

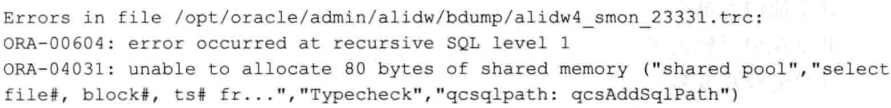
select ksmchsiz,count(\*) from x$ksmsp where KSMCHCLS='free' and KSMCHSIZ<=150 group by KSMCHSIZ order by KSMCHSIZ;

该查询可以根据实际情况改变KSMCHSIZ的输入值。

使用该sql一定要注意，会持有shared pool latch。如果是繁忙的数据库，切不可用此方法进行查询。

当然通过4031的报错信息也能看出端倪，但这个报错信息很有可能是个错误的报错信息，就如标题所说。

这里直接拿吕老师的示例解释该问题：



我们看到报错是进程想要80字节的chunk，但是内存不足无法获得。

加入这是个子堆，比如子堆有俩个4096字节的extent，但是都满了，进程需要的80字节无法从子堆现有的extent中获得，那么就需要从父堆获得，父堆要首先分配给子堆一个4096字节的chunk来作为子堆的一个新区（因为堆里的extent大小必须一致），然后再从这个新区分配80字节给进程。

在这种情况下，其实是无法获得4096字节的chunk，但是报错依然会报无法获得80字节。

##### 实验

下面做个实验：

首先把open\_cursor改成最大65535：

SQL> alter system set open\_cursors=65535;

System altered.

然后开一个会话，执行下面匿名块：

declare

msql varchar2(500);

mcur number;

mstat number;

begin

for i in 1..60000 loop

mcur:=dbms\_sql.open\_cursor;

msql:='select loc from dept where deptno='||to\_char(i);

dbms\_sql.parse(mcur,msql,dbms\_sql.native);

mstat:=dbms\_sql.execute(mcur);

end loop;

end;

/

观察alert日志：

不一会儿就要报错了：

2019-04-11T15:29:33.234374+08:00

PDBPROD1(3):Errors in file /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/incident/incdir\_36329/PRODCDB\_ora\_77893\_i36329.trc:

ORA-01013: user requested cancel of current operation

ORA-04031: unable to allocate 56 bytes of shared memory ("shared pool","select loc from dept where d...","SQLA^2ab5b062","qosdInitExprCtx: currExprInfo\_r")

2019-04-11T15:29:35.351956+08:00

Errors in file /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/trace/PRODCDB\_smon\_111130.trc:

ORA-00604: error occurred at recursive SQL level 1

ORA-04031: unable to allocate 40 bytes of shared memory ("shared pool","select f.file#, f.block#, f....","SQLA","tmp")

当发生报错的时候，我们做一个level 2的heapdump。观察dump：

FREE LISTS:

Bucket 0 size=32

Chunk 07c000090 sz= 0 kghdsx

Bucket 1 size=40

Bucket 2 size=48

Chunk 083696498 sz= 48 free " "

Chunk 08376fab8 sz= 48 free " "

…

Bucket 3 size=56

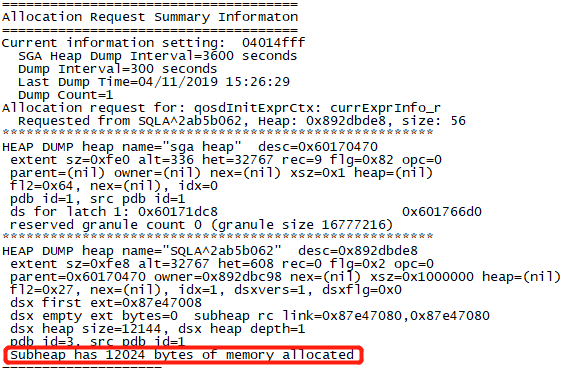
Chunk 086ffada8 sz= 56 free " "

Chunk 085ffcda8 sz= 56 free " "

…

可以看到，当时有大量的48字节和56字节的free类型的chunk。所以显然真正缺少的并不是这样的chunk。

我们看下alert中的trace文件：



我们可以很清楚的从trace里面看到，subheap需要12024个字节，那么freelist到底有没有这么大的chunk呢？

…

Bucket 249 size=9392

Bucket 250 size=12368

Bucket 251 size=12376

Bucket 252 size=16408

Bucket 253 size=32792

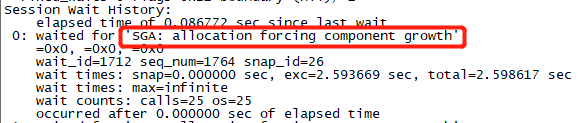
Bucket 254 size=65560

Chunk 0892a9e88 sz= 169936 free " "

Total free space = 9425632

我们看，只有一个169936字节的chunk可以切分，虽然按道理依然可以完成分配，但是这里我推断，应该是oracle忙不过来了，所以很多要求分配内存的请求得不到满足就报4031了。

我之所以如此推断，是因为保留池其实还有更多的chunk，同时当时的等待事件全部都是SGA: allocation forcing component growth：



但是这不重要了，重要的是，我们证明了4031的报错的欺骗性，所以一定要到trace里面去进行进一步的确认。

#### 4031的危害

容易引起宕机，当LGWR、DBWR、PMON、SMON、CKPT无法获得shared pool内存时，数据库会崩溃。

但错误出现时及时处理，仍然可以挽救。

不过我在实验的过程中，由于这次实验做的有点儿猛，导致完全救不过来，进程杀了数据库都缓不过来，就只能重启了，所以实验做的并不细致。

### 解决ORA-4031之道：如何正确释放内存

### session cached cursor与内存占用

## 语句解析和执行

### SQL执行流程

我们通过PLSQL最原始的“动态游标”（其实就是dbms\_sql包），来了解SQL语句的执行流程：

准备实验表：

SQL> create table vage(id number,name varchar2(40));

Table created.

SQL> insert into vage select rownum,object\_name from dba\_objects where rownum<=100;

100 rows created.

SQL> commit;

Commit complete.

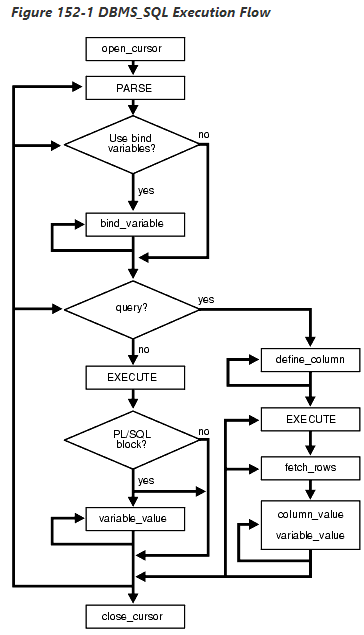
SQL> select name from vage where id=1;

NAME

----------------------------------------

TS$

然后我们看官文给出的sql执行流程图：



可以说写的非常明确了：

打开游标🡪解析sql🡪向绑定变量传值🡪判断是否为select语句，这里我们的实验是“是”🡪然后将select list和预定义的变量进行对应🡪执行🡪取出需要的行🡪将行中每一列的值传递给前面difine\_column定义好的列🡪关闭游标

下面就是对应上述步骤的匿名block：

declare

mcur number;

mstat number;

v\_name varchar2(40);

begin

mcur:=dbms\_sql.open\_cursor;

dbms\_sql.parse(mcur,'select name from vage where id=:x',dbms\_sql.native);

dbms\_output.put\_line(mcur);

dbms\_sql.bind\_variable(mcur,':x',1);

dbms\_sql.define\_column(mcur,1,v\_name,10);

mstat:=dbms\_sql.execute(mcur);

dbms\_output.put\_line('Number of rows processed: '||mstat);

mstat:=dbms\_sql.fetch\_rows(mcur);

dbms\_output.put\_line('Number of rows processed: '||mstat);

dbms\_sql.column\_value(mcur,1,v\_name);

dbms\_output.put\_line ('Query result:'||v\_name);

dbms\_sql.close\_cursor(mcur);

end;

/

注意，execute和fetch\_rows返回的是处理了的行数，我们看下面的执行结果，会注意到，execute过程并没有处理行，而是在fetch过程才会真的获取行：

879571609

Number of rows processed: 0

Number of rows processed: 1

Query result:TS$

PL/SQL procedure successfully completed.

这里额外说明一下，吕老师的书中define\_column是在execute之后的，这个顺序至少在12c中是不对的，所以如果按吕老师的顺序，将无法执行出结构。

另外，从图中可以看出，当然吕老师也说了这一点：如果不是select语句，比如dml，ddl在bind\_variable之后就会直接进入执行阶段，并且没有fetch阶段。

### 内存锁原理

#### 内存锁分类

Oracle针对SGA，设置了5种内存锁：

1. Library Cache Lock/Pin
2. Buffer Pin
3. Row Cache Lock
4. 各种Latch
5. Mutex

前3种锁级别稍高，通常这3种锁的获取和释放，需要在相应的latch或mutex的保护下进行。

Latch和Mutex是低级别锁，各有利弊，所以并不是Mutex就可以完全取代Latch。

这5类锁有一个主要区别：

1. Library Cache Lock/Pin、Row Cache Lock和一部分Latch有等待者队列和持有者队列
2. Buffer Pin、小部分的Latch和所有Mutex都没有任何队列。

有队列的锁出现竞争时按先来后到，没有队列的锁靠运气竞争。

#### 内存锁的加锁过程（持有者队列&等待者队列）

我们先来说有队列的内存锁的加锁机制。

我们先执行一个sql：

SQL> select \* from vage where id=10;

ID NAME

---------- ----------------------------------------

10 I\_CDEF4

然后进行如下查询（查询X$KGLOB）：

SQL> select KGLHDADR,KGLHDLMD,KGLHDPMD,KGLOBHD0,KGLOBHD6 from x$kglob where KGLNAOBJ='select \* from vage where id=10';

KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------- ---------- ---------------- ----------------

00000000A5408750 1 0 00000000631A7C70 00000000C22AAD28

00000000A5C95A38 1 0 00000000B54ECCA8 00

KGLHDLMD是父子游标句柄持有的锁模式，我们看到都是1号锁，这里我sql只执行了一次（12.2环境中），依然可以看到父子游标上有1号library cache lock，而不是非要将该sql cache到PGA中才会持有1号library cache lock。

KGLHDPMD是持有的library cache pin的模式，我们看到，父子游标的heap都未持有任何模式的library cache pin。

接着进行如下查询（查询X$KGLLK）：

SQL> select KGLLKADR,KGLLKHDL,KGLLKMOD,KGLLKREQ,KGLLKCNT,KGLLKSNM from x$kgllk where KGLNAOBJ='select \* from vage where id=10';

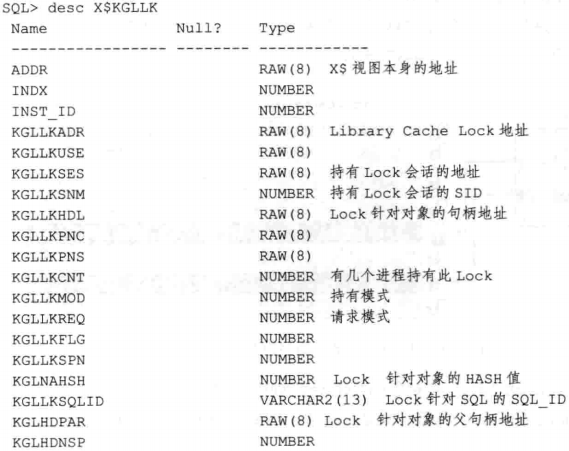
KGLLKADR KGLLKHDL KGLLKMOD KGLLKREQ KGLLKCNT KGLLKSNM

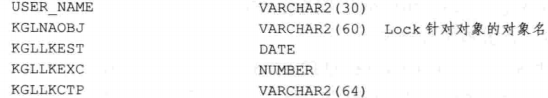
---------------- ---------------- ---------- ---------- ---------- ----------

00000000A58AF1A0 00000000A5408750 1 0 1 131

00000000BE2976B0 00000000A5C95A38 1 0 1 131

我们先看下X$KGLLK的结构：





KGLLKHDL是已经持有library cache lock的对象句柄地址，这和我们前面查到的x$kglob中的父子游标的句柄地址完全呼应。

KGLLKMOD列依然是持有library cache lock的模式。

KGLLKREQ列是请求以多少级别的模式持有library cache lock，显然，这一列如果有值，那么说明有阻塞了（跟v$lock中的req是一样的）。

KGLLKCNT列是持有者的数量。KGLLKSNM是持有者的sid。

最后注意：KGLLKADR就是对应的锁的内存地址。也就是除了非队列锁，锁都是一小块内存，存着：持有者的sid、持有模式、针对的对象。

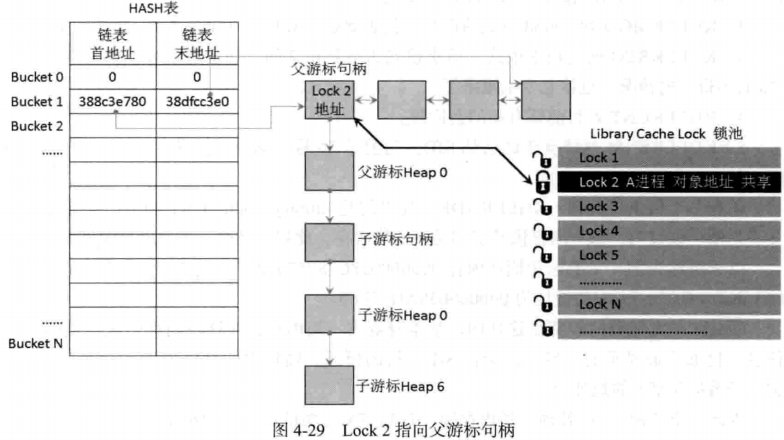
下面来说整个上锁的流程：

假如A进程想在一个sql的父游标句柄上获得library cache lock：

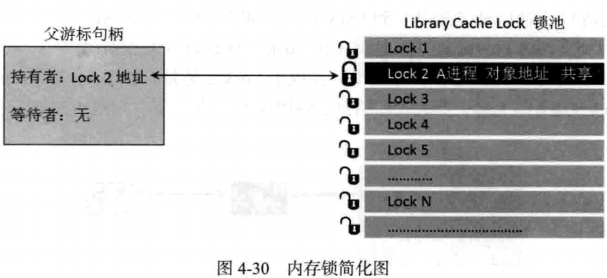
首先，A进程到library cache lock的锁池中找一把空闲的锁（lock2），把自己的进程号和锁模式等信息写入其中：



然后，将要加锁的对象，及父游标的句柄地址记录到lock 2所在的内存中，同时也将lock2的内存地址记录到父游标的句柄中（这一步才是真正的加锁过程！）：



简化一下这张图就是：



现在我们可以清楚的看到，lock 2的地址实际上是记录在了父游标的持有者地址中（注意这里只能放一个地址，它是持有者队列头，但这里不是持有者队列）的。

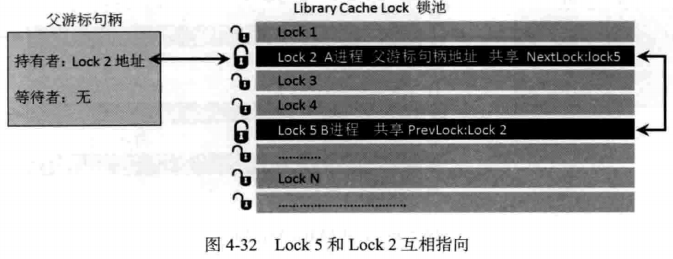
接着B进程也要以共享模式申请该父游标句柄上的锁，第一步一样，假如B进程使用的是lock 5，则B进程把自己的信息和要申请的模式记录到lock 5中。

拿到锁之后，B进程到对应的父游标句柄中的持有者地址中看是否已有进程加锁在该父游标句柄上。

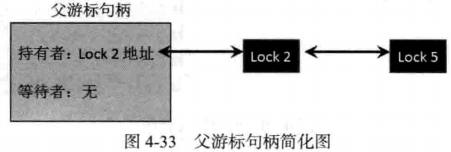
B进程发现有个lock 2在里面，则根据lock 2的地址，到lock 2中获得实际的信息，发现是A进程加的共享锁。

B进程要加的也是共享锁，所以并不冲突，可以加锁，但是并不能像A进程那样，把自己的锁（lock 5）的地址也写到父游标句柄中，如果是这样，那父游标句柄占用的内存将会变得很大。

所以B进程将自己的lock 5的地址写到lock 2的NextLock中，同时也将lock 2的地址写到自己的lock 5的PrevLock中。



简化一下该图就是：



我们可以看到，持有者队列形成了：



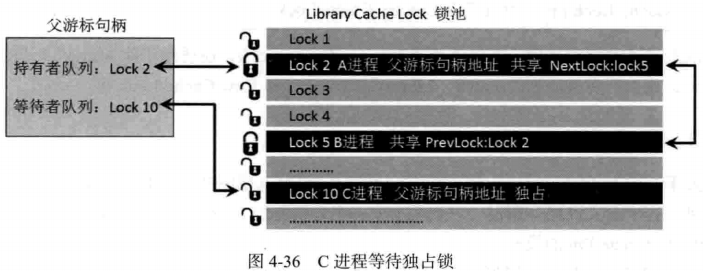
现在假如C进程要在该父游标句柄上持有独占锁。

那么第一步还是一样，找个lock 10，把自己的信息和想干的事儿写进去。

然后到对应父游标句柄中查看持有者队列，发现A和B进程都在以共享模式持有锁。那么C要申请的锁模式与之不兼容，所以C进程要进入等待，则C进程进入睡眠。

所以lock 10的地址会被写入到父游标句柄的等待者地址中。

那么后面，不管D进程以何种模式（共享或者独占）申请锁，都会进入等待！（体现出了先来后到！）



那么当A进程释放了锁之后，B进程持有的lock 5就成了新的持有者队列头，只有等B进程也释放了锁，B进程唤醒C进程。C进程才能以独占模式访问该父游标句柄，C进程的锁的相关等待事件到此结束。

虽然事实上，对于SQL语句，无论父子游标句柄，都不会加独占的library cache lock，但是这里老师之所以这么举例，主要目的是想说明当遇到不兼容模式的锁申请时，Oracle是如何处理的。

所以上述的加锁、等待过程适用于Oracle中所有有持有者队列的内存锁！

### Library Cache Lock/Pin

我们在前面看到x$kglob中对象对应的library cache lock和pin的模式（KGLHDLMD和KGLHDPMD）。

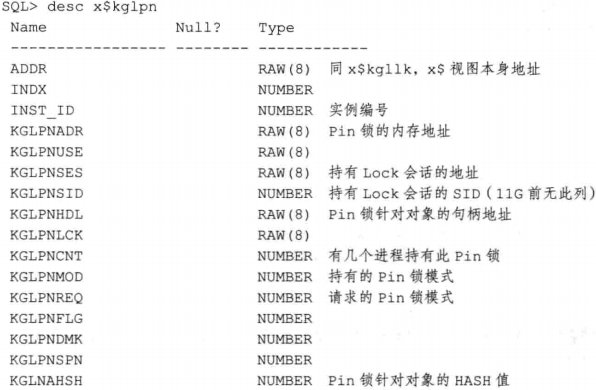
注意library cache lock有3中模式：1（NULL），2（Shared），3（Exclusive）。

而pin只有2和3。

模式为1的library cache lock意思就是该对象重要的chunk不会被链到LRU链上，保证其不会被覆盖。

但是x$kglob中是每一个对象一行记录，也就是说该表是从对象的角度来进行统计的。

所以更详细的library cache lock的信息需要查看x$kgllk，而library cache pin的信息需要查看x$kglpn。



该视图11g前需要使用KGLPNSES同v$session中的addr关联获得sid。

### Library Cache Lock/Pin和硬解析

这里我用吕老师的dtrace脚本进行观察，但是没有输出结果。

所以将就使用gdb来进行探究。

开3个会话：

第一个会话：

SQL> @mysp

SID SPID

---------- ------------------------

15 64471

用gdb进行跟踪，并设置俩个断点，按吕老师的讲解，kgllkal是跟library cache lock相关（句柄），而kglpnal是跟library cache pin相关（heap），其实这俩个都是分配（allocate）锁的函数：

[oracle@host01 ~]$ gdb -p 64471

(gdb) b kgllkal

Breakpoint 1 at 0x10ff9a80

(gdb) b kglpnal

Breakpoint 2 at 0x10ffd9e0

然后我们在会话1中做一个硬解析：

SQL> select \* from dept1 where dname='BB';

在gdb中输入c：

(gdb) c

Continuing.

Breakpoint 1, 0x0000000010ff9a80 in kgllkal ()

这时候我们在第二个会话，同样执行这条sql：

SQL> @mysp

SID SPID

---------- ------------------------

257 64503

SQL> select \* from dept1 where dname='BB';

我们会看到sql hang住。

当进入函数kgllkal时，x$kglob是无法进行查询的！

然后我们到会话3，查v$session:

SQL> @yx/mysp

SID SPID

---------- ------------------------

383 65386

SQL> select event from v$session where sid in (15,257);

我们会发现，这里会hang住，当然，如果查其他sid就可以正常查询！

v$session\_event也无法进行查询，甚至不能desc。

在最初的几次调用kgllkal时，都是上述情况。

然后我们一直摁c，到v$session能出结果：

SQL> select event from v$session where sid in (15,257);

EVENT

------------------------------

SQL\*Net message from client

library cache lock

最后一路c到最后，期间会多次调用kgllkal和kglpnal，比吕老师说的次数要多一些。

当会话1的硬解析完成后，会话2的sql才能执行成功。

然后我们来看下v$session\_event:

SQL> select sid,event,TOTAL\_WAITS from v$session\_event where sid in(15,257,383) and event not like 'SQL\*Net%';

SID EVENT TOTAL\_WAITS

---------- ------------------------------ -----------

15 events in waitclass Other 38

257 log file sync 1

257 library cache lock 1

257 library cache load lock 2

257 library cache: mutex X 1

257 events in waitclass Other 39

383 Disk file operations I/O 30

383 library cache: mutex X 2

383 library cache: bucket mutex X 4

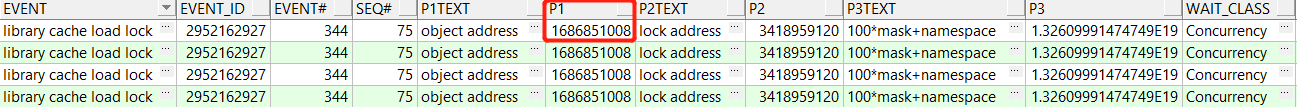
383 events in waitclass Other 17

首先我们看到，我们查询v$session时的等待其实是等待library cache: mutex X，那么这个等待在12c中被细化成了3个等待事件：

In 12, the events have been split further into three separate events:

1. library cache: mutex X -- for handle objects
2. library cache: bucket mutex X -- for library cache hash buckets
3. library cache: dependency mutex X -- for dependencies

然后我们来看会话2等待硬解析的等待，有俩次library cache load lock的等待，我们具体可以在v$active\_session\_history里面找一下：



再然后我们换算并且看一下，这个object到底是哪个？

SQL> select to\_char(1686851008,'xxxxxxxxxxxxxxxxxxxx') from dual;

TO\_CHAR(1686851008,'X

---------------------

648b4dc0

SQL> select KGLHDPAR,KGLHDADR,KGLHDLMD,KGLHDPMD,KGLOBHD0,KGLOBHD6 from x$kglob where kglnaobj like 'select \* from dept1 where dname=%BB%';

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

00000000648B4DC0 00000000A558E770 1 0 00000000692B67B0 00000000B1B016D8

00000000648B4DC0 00000000648B4DC0 1 0 00000000A5C4D9D0 00

我们发现，正是该sql的父游表的句柄。

这里我只把我的实验结果和思路大体记录一下，具体很多细节我还没有捋清楚。

#### 吕老师的实验

最后我们做一下吕老师的实验：

新开俩个会话，619和752：

SQL> select sid,event,TOTAL\_WAITS from v$session\_event where sid in(619,752) and event not like 'SQL\*Net%';

SID EVENT TOTAL\_WAITS

---------- ------------------------------ -----------

619 log file sync 1

619 events in waitclass Other 38

752 log file sync 1

752 events in waitclass Other 39

先在619中单独执行如下代码：

declare

mcur number;

mstat number;

v\_name varchar2(40);

begin

mcur:=dbms\_sql.open\_cursor;

for i in 1..100000 loop

dbms\_sql.parse(mcur,'select /\*+SESS\_619\*/name from vage where id='||i,dbms\_sql.native);

mstat:=dbms\_sql.execute(mcur);

end loop;

dbms\_sql.close\_cursor(mcur);

end;

/

相当于做10万次硬解析，然后我们看等待事件：

SID EVENT TOTAL\_WAITS

---------- ------------------------------ -----------

619 log file sync 1

619 latch: shared pool 167

619 library cache: mutex X 1

619 events in waitclass Other 38

752 log file sync 1

752 events in waitclass Other 39

发现有一次library cache: mutex X，更多的是latch: shared pool。

现在我们在俩个会话分别执行如下代码：

会话619：

for i in 100001..200000 loop

dbms\_sql.parse(mcur,'select /\*+SESS\_619\*/name from vage where id='||i,dbms\_sql.native);

mstat:=dbms\_sql.execute(mcur)

会话752：

for i in 1..100000 loop

dbms\_sql.parse(mcur,'select /\*+SESS\_752\*/name from vage where id='||i,dbms\_sql.native);

mstat:=dbms\_sql.execute(mcur)

这里我只把与之前代码不同的部分展示出来。

俩段代码执行完，我们可以看到等待事件（已去掉无关的等待事件）：

SID EVENT TOTAL\_WAITS

---------- ------------------------------ -----------

619 row cache mutex 43

619 latch: shared pool 44014

619 library cache: mutex X 285

619 library cache: bucket mutex X 10

752 row cache mutex 24

752 latch: shared pool 42646

752 library cache: mutex X 305

752 library cache: bucket mutex X 4

实验效果与吕老师的实验基本一致。

可以看到latch: shared pool仍然是最严重的等待，而且相比之前可以说是急速加剧。

同时我们也看到了许多library cache: mutex X和个别row cache mutex等待，那么这里产生library cache: mutex X和library cache: bucket mutex X的原因就是因为俩个会话同时都在访问vage表，说白了就是涉及了相同的对象。

当然对于同实例下的实验，这里就给个实验结果：

SID EVENT TOTAL\_WAITS

---------- ------------------------------ -----------

619 row cache mutex 88

619 latch: shared pool 97628

619 library cache: mutex X 309

619 library cache: bucket mutex X 16

752 row cache mutex 66

752 latch: shared pool 93973

752 library cache: mutex X 330

752 library cache: bucket mutex X 8

结论就是library cache: mutex X和row cache mutex比之前涉及同一个对象要好一些，但是还是有。

### Library Cache Lock/Pin和软解析、软软解析

#### 软解析

依然用gdb进行跟踪，会话2中先执行一遍sql做硬解析，然后会话1中（被trace会话）进行第二次执行就是软解析：

SQL> select \* from dept1 where dname='DD';

(gdb) b kgllkal

Breakpoint 1 at 0x10ff9a80

(gdb) b kglpnal

Breakpoint 2 at 0x10ffd9e0

Breakpoint 1, 0x0000000010ff9a80 in kgllkal ()

Breakpoint 1, 0x0000000010ff9a80 in kgllkal ()

Breakpoint 1, 0x0000000010ff9a80 in kgllkal ()

Breakpoint 1, 0x0000000010ff9a80 in kgllkal ()

Breakpoint 2, 0x0000000010ffd9e0 in kglpnal ()

我们看到，这俩个函数的调用次数明显少于硬解析。

然后当第一次调用kgllkal时，我们在会话2再次执行该sql，然后一直到最后一次调用kglpnal之后，会话2都将被阻塞，阻塞原因如下：

SQL> select event from v$session where sid in (15,257);

EVENT

------------------------------

SQL\*Net message from client

library cache: mutex X

所以软解析不会遇到library cache lock/pin的阻塞，但是会遇到library cache: mutex X。

#### 软软解析

接着我们看软软解析。

(gdb) b kgllkal

Breakpoint 1 at 0x10ff9a80

(gdb) b kglpnal

Breakpoint 2 at 0x10ffd9e0

(gdb) c

Continuing.

我们看到，软软解析就不会再调用这俩个函数了，其实在第二次软解析的时候，就不会再调用这俩函数了。

至于大量软软解析会产生的Cursor: Pin S的等待，我们在后面mutex部分细说，这里先简单做个实验。

吕老师说软软解析要加一次Cursor: Pin S的mutex。

加锁使用的是kgxSharedExamine，它将调用sskgslcas来获取mutex，获取之后使用kgxEndExamine来完成获取流程。

kgxSharedExamine sharedexamine mode

kgxEndExamine mutex clear E mode

我们实验一下：

(gdb) b kgxSharedExamine

Breakpoint 1 at 0x11027d70

(gdb) b kgxEndExamine

Breakpoint 2 at 0x11029280

Breakpoint 1, 0x0000000011027d70 in kgxSharedExamine ()

(gdb)

Continuing.

Breakpoint 2, 0x0000000011029280 in kgxEndExamine ()

(gdb)

Continuing.

Breakpoint 2, 0x0000000011029280 in kgxEndExamine ()

(gdb)

Continuing.

在第一次调用kgxEndExamine ()时我们可以看到：

SQL> select KGLHDPAR,KGLHDADR,KGLHDLMD,KGLHDPMD,KGLOBHD0,KGLOBHD6 from x$kglob where kglnaobj like 'select \* from dept1 where dname=%DD%';

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

00000000AD8E9620 00000000AD795ED0 1 2 00000000AB273A68 00000000BF27CD20

00000000AD8E9620 00000000AD8E9620 1 0 00000000A978E800 00

子游标上加的是2级的pin锁，其实这里是cursor pin，而不是library cache pin，如下查询可以证明这不是library cache pin：

SQL> select \* from x$kglpn;

no rows selected

第二次调用kgxEndExamine ()时，加的就是3级的cursor pin锁：

SQL> select KGLHDPAR,KGLHDADR,KGLHDLMD,KGLHDPMD,KGLOBHD0,KGLOBHD6 from x$kglob where kglnaobj like 'select \* from dept1 where dname=%DD%';

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

00000000AD8E9620 00000000AD795ED0 1 3 00000000AB273A68 00000000BF27CD20

00000000AD8E9620 00000000AD8E9620 1 0 00000000A978E800 00

那么我们看到这里，有独占的情况出现，我们就试试在另一个会话再次执行该sql，也就是说也做软软解析会不会等待呢（在此之前，也就是持有2级cursor pin的时候，另一个会话执行该sql并不会阻塞）？

确实会发生阻塞，而且我们来看看这个阻塞：

SQL> select sid,event from v$session where sid in (15,257);

SID EVENT

---------- ------------------------------

15 SQL\*Net message to client

257 cursor: pin S

正是cursor: pin S，所以我们也就知道，到底软软解析会在哪一步发生争用了。

虽然说cursor pin确实一定程度上取代了library cache pin，但是并不是完全取代。有一个实验可以说明这一点，就是执行一个存储过程的同时，另开一个会话对其进行编译。

### Null模式Library Cache Lock和依赖链

Null模式的Library Cache Lock的主要作用就是标志对象是有效的，相关内存不能被覆盖。当对象不在有效时，null模式的锁将会被去掉。

我们来做个小实验：

开一个会话执行如下sql：

SQL> select dname from dept1 where deptno=30;

DNAME

--------------

SALES

我们来看该sql的library cache lock的情况：

SQL> select KGLHDPAR,KGLHDADR,KGLHDLMD,KGLHDPMD,KGLOBHD0,KGLOBHD6 from x$kglob where kglnaobj like 'select dname from dept1 where deptno=30%';

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

0000000061ED6858 00000000632AB678 0 0 000000009E4CB3E8 000000006693B598

0000000061ED6858 0000000061ED6858 0 0 0000000066805798 00

我们看到父子游标句柄都没有锁。

再执行一次sql再看：

SQL> /

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

0000000061ED6858 00000000632AB678 1 0 000000009E4CB3E8 000000006693B598

0000000061ED6858 0000000061ED6858 1 0 0000000066805798 00

现在有null锁了。

我们重命名dept1：

SQL> alter table dept1 rename to dept2;

Table altered.

SQL> alter table dept2 rename to dept1;

Table altered.

再看：

SQL> /

KGLHDPAR KGLHDADR KGLHDLMD KGLHDPMD KGLOBHD0 KGLOBHD6

---------------- ---------------- ---------- ---------- ---------------- ----------------

0000000061ED6858 00000000632AB678 0 0 000000009E4CB3E8 000000006693B598

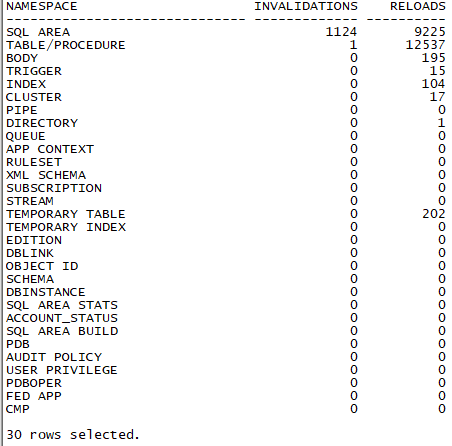
0000000061ED6858 0000000061ED6858 0 0 0000000066805798 00

确实是，当对象失效时，null锁释放。

除了sql，表、索引、视图这些都可以失效。

v$librarycache中可以看到某一种类型的对象失效和重载的次数：

SQL> select NAMESPACE,INVALIDATIONS,RELOADS from v$librarycache;



#### 依赖链

上述的失效过程是通过依赖链来实现的，依赖链是由各个对象堆0中的ReferenceList组成，依赖链是一条单向链。

比如上面那个示例，dept1对象的referencelist就会记录select dname from dept1 where deptno=30的父游标句柄。

ReferenceList的信息可以在v$object\_dependency（来源就是heap 0中的ReferenceList）中获得：



我们看到后面的to\_type就是对象的type#（obj$中的type#），具体namespace以及type#的翻译见如下文本：



比如存储过程中包含了某个表对象，也可以在这个视图里找到他们的依赖关系。

当dept1表做了ddl，Oracle就会根据referenclist找到select dname from dept1 where deptno=30这条sql，然后释放其句柄上的null模式的library cache lock。

### 存储过程与Library Cache Lock/Pin

开三个会话：16、634、619，再另起一个会话做观察。

先写一个存储过程：

SQL> create or replace procedure dummy is

2 a number;

3 begin

4 select deptno into a from dept1 where deptno=10;

5 dbms\_lock.sleep(5000);

6 end;

7 /

Procedure created.

然后我们在16会话中执行它：

SQL> begin

2 dummy;

3 end;

4 /

然后我们给dept1改名再改回来：

SQL> alter table scott.dept1 rename to dept10;

Table altered.

SQL> alter table scott.dept10 rename to dept1;

Table altered.

SQL> select object\_type,status from dba\_objects where object\_name='DUMMY';

OBJECT\_TYPE STATUS

----------------------- -------

PROCEDURE INVALID

我们看到procedure已经失效了。

然后我们在634会话也执行dummy：

SQL> @yx/mysp

SID SPID

---------- ------------------------

634 81526

SQL> begin

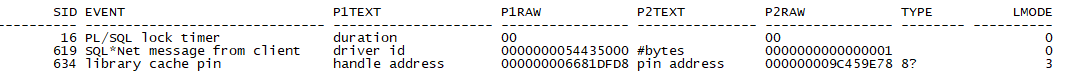
2 dummy;

3 end;

4 /

然后我们来看等待：

select s.sid,s.event,s.p1text,s.p1raw,s.p2text,s.p2raw,chr(bitand(p3,-16777216)/16777215)||chr(bitand(p3, 16711680)/65535) Type, mod(p3,16) lmode from v$session s where sid in(16,634,619) ;



我们从session中看到634号会话再申请3号（独占）library cache pin，但求而不得。

然后我们根据P1 RAW去找对应library cache pin的具体信息：

select KGLPNADR,KGLPNSID,KGLPNHDL,KGLPNMOD,KGLPNREQ,KGLNAOBJ from x$kglpn a,x$kglob b where a.KGLPNHDL=b.KGLHDADR and b.KGLHDADR='000000006681DFD8';

KGLPNADR KGLPNSID KGLPNHDL KGLPNMOD KGLPNREQ KGLNAOBJ

---------------- ---------- ---------------- ---------- ---------- --------------------

000000009C459E78 634 000000006681DFD8 0 3 DUMMY

00000000A01FF248 16 000000006681DFD8 2 0 DUMMY

我们看到，是会话16持有2号（共享）library cache pin（只有2和3俩种模式），但是634会话在执行dummy前需要重新编译dummy（因为失效了），此时需要持有3号library cache pin，由于锁模式不兼容，所以进入等待。

我们可以看到634号会话对应的KGLPNADR就是P2 RAW。

当然这里由于原本会话16持有的只是1号library cache lock，所以634号可以成功获取了3号library cache lock。这一点在这次实验没有体现出来，一会儿补一个实验我们来看。

现在我们继续在619号会话执行存储过程：

SQL> @yx/mysp

SID SPID

---------- ------------------------

619 81551

SQL> begin

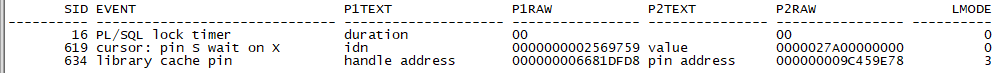
2 dummy;

3 end;

4 /

现在634是持有3号library cache lock的，那么619号会话的申请将会遇到阻塞，但是会是library cache lock等待么？10g会是，而11g以后就不是了：

select s.sid,s.event,s.p1text,s.p1raw,s.p2text,s.p2raw,mod(p3,16) lmode from v$session s where sid in(16,634,619) ;



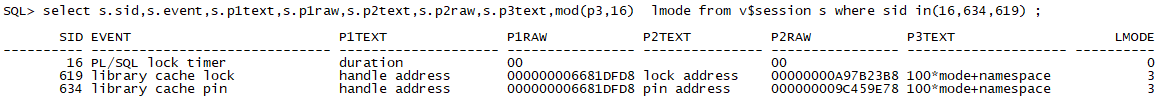
我们看到，等待事件是cursor:Pin S wait on X的等待，说明634会话在重新编译dummy的时候，不仅持有3号library cache lock，还持有了独占mutext🡪Cursor:Pin X。

所以当然看cursor:Pin S wait on X和library cache pin同时出现时，很有可能就是存储过程出了问题。

现在我们把619 ctrl C掉，然后我们在619中编译存储过程，又会是什么情形呢？

SQL> alter procedure dummy compile;

这次等待将会是library cache lock：



那么我们发现619和634会话的P1RAW是相同的，这是必然的，因为他们是在同一个句柄上上的锁嘛。

我们来追踪这个library cache lock：

SQL> select KGLLKADR,KGLLKSNM,KGLLKHDL,KGLLKMOD,KGLLKREQ,KGLNAOBJ from x$kgllk where KGLLKHDL='000000006681DFD8';

KGLLKADR KGLLKSNM KGLLKHDL KGLLKMOD KGLLKREQ KGLNAOBJ

---------------- ---------- ---------------- ---------- ---------- --------------------

00000000A97B23B8 619 000000006681DFD8 0 3 DUMMY

00000000A595FF70 16 000000006681DFD8 1 0 DUMMY

000000009C459D50 634 000000006681DFD8 3 0 DUMMY

这里我们就能看到，会话16原本持有的是1号模式的library cache lock，而会话634要重新编译dummy，成功获取了3号library cache lock，因为null模式的锁跟任何模式都兼容。

我们将执行存储过程的会话都终止掉，然后存储过程完成重新编译后：

SQL> select KGLLKADR,KGLLKSNM,KGLLKHDL,KGLLKMOD,KGLLKREQ,KGLNAOBJ from x$kgllk where KGLLKHDL='000000006681DFD8';

KGLLKADR KGLLKSNM KGLLKHDL KGLLKMOD KGLLKREQ KGLNAOBJ

---------------- ---------- ---------------- ---------- ---------- --------------------

000000009C459538 634 000000006681DFD8 1 0 DUMMY

00000000A595FF70 16 000000006681DFD8 1 0 DUMMY

我们看到，俩个会话持有的library cache lock的锁模式都变成了null模式。

### 断开依赖链

## MUTEX

# REDO调优与备份恢复原理

## 非IMU与IMU redo格式的不同