**游标和绑定变量**

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 | 2018-06-10 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. ORACLE里的cursor - 4 -](#_Toc518066735)

[1.1 ORACLE体系结构 - 4 -](#_Toc518066736)

[1.1.1 体系结构总概 - 4 -](#_Toc518066737)

[1.1.2 Library Cache - 4 -](#_Toc518066738)

[1.2 SHARED CURSOR - 8 -](#_Toc518066739)

[1.2.1 存在哪儿 - 8 -](#_Toc518066740)

[1.2.2 存些啥 - 8 -](#_Toc518066741)

[1.2.3 咋分类的 - 9 -](#_Toc518066742)

[1.2.4 咋干活的 - 10 -](#_Toc518066743)

[1.3 SESSION CURSOR - 10 -](#_Toc518066744)

[1.3.1 SESSION CURSOR的特点 - 10 -](#_Toc518066745)

[1.3.2 SESSION DUMP - 11 -](#_Toc518066746)

[1.3.3 SESSION CURSOR的参数 - 14 -](#_Toc518066747)

[1.4 三种解析方式 - 18 -](#_Toc518066748)

[1.4.1 硬解析 - 18 -](#_Toc518066749)

[1.4.2 软解析 - 19 -](#_Toc518066750)

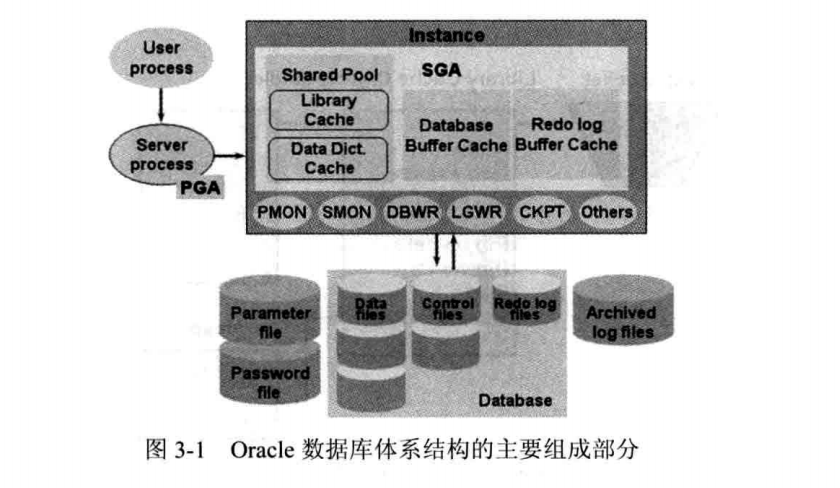
[1.4.3 软软解析 - 19 -](#_Toc518066751)

[1.4.4 实验 - 19 -](#_Toc518066752)

# ORACLE里的cursor

## ORACLE体系结构

### 体系结构总概



### Library Cache

#### 库缓存的结构

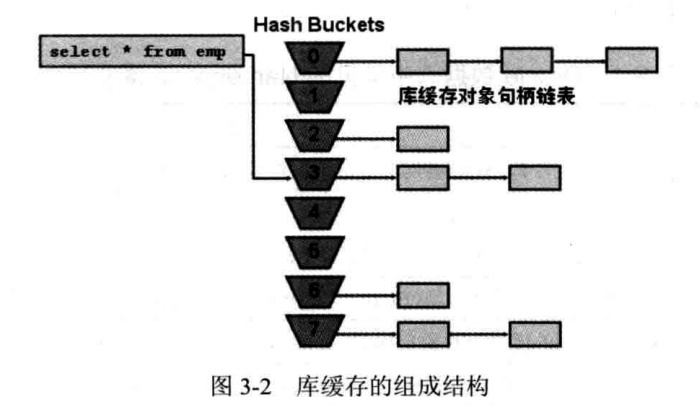
库缓存的主要作用是缓存刚刚执行过的sql语句和pl/sql语句所对应的执行计划、解析树（Parse tree）等对象。

这些缓存再库缓存中的对象叫做库缓存对象（Library Cache Object）,所有库缓存对象都以库缓存对象句柄（Library Cache Object Handle）的结构存储在库缓存中。

ORACLE通过访问相关的library cache object handle来访问对应的library cache object。

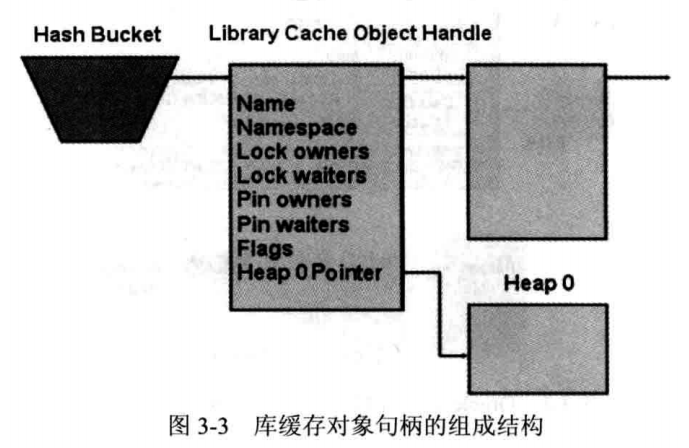
Library cache object handle是以hash table的方式存储在library cache中。

从hash table的角度看，库缓存如下图：



这里注意，一个hash buckets放一个hash值（本文说的就是SQL文本的hash值），也就是说同一个hash bucket下的library cache object handle的hash值是相同的，这些同一个hash bucket的不同库缓存对象句柄之间用指针连接，形成一条库缓存对象句柄链表（Library Cache Object Handles）。

#### 库缓存对象句柄的结构

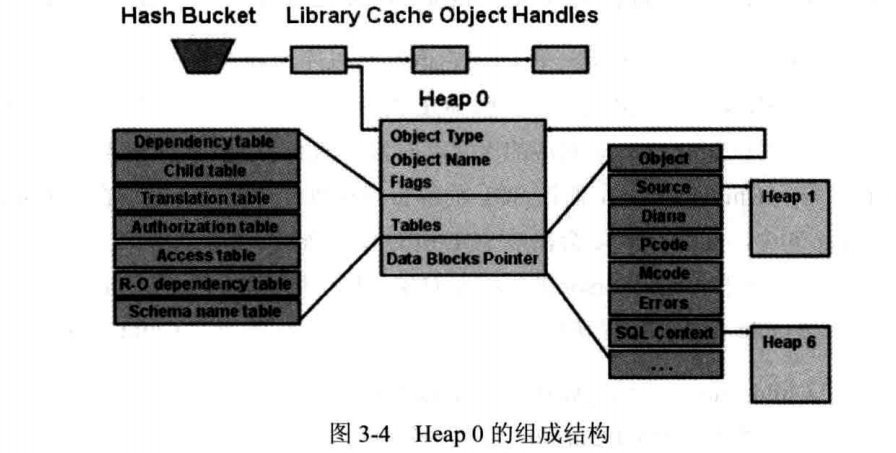


库缓存对象句柄中有很多属性，对于sql来说，我们先主要了解3个：Name、Namespace、Heap 0 Pointer。

NAME：sql就是SQL文本，table就是tablename。

NAMESPACE：CRSR（SQL和匿名PL/SQL语句所对应的namespace），TABL（表）

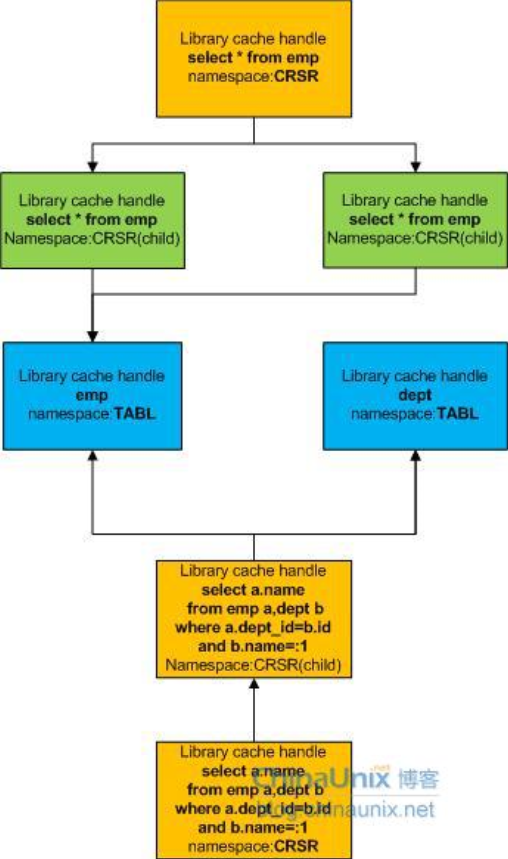
HEAP 0 POINTER：



Heap 0 pointer中的重要属性是tables和data blocks pointer。

Tables放的是所有与本对象关联的库缓存对象句柄的地址的集合。也就是这里记得都是地址，且根据这些地址可以找到相关对象的句柄。

其中比较重要的dependency table和child table如下：

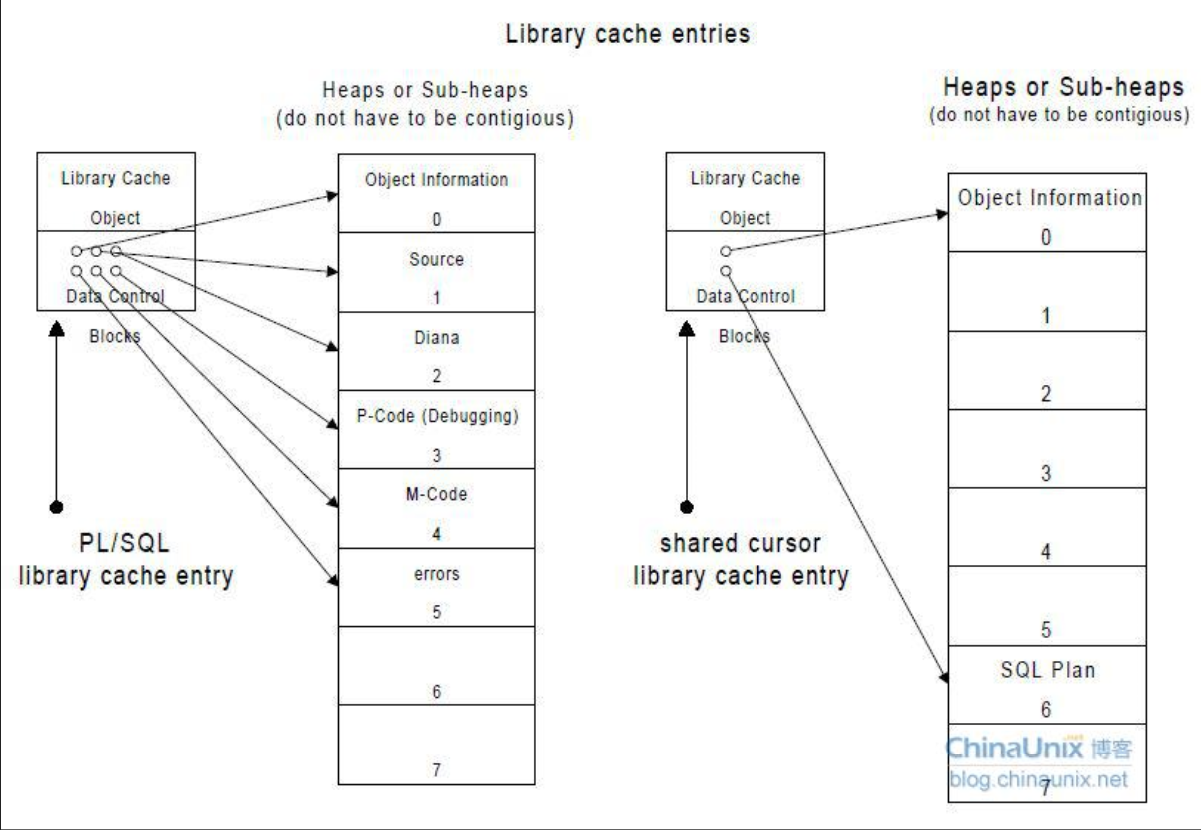


首尾的俩个方块是俩个不同的父对象句柄（也就是sql被hash以后首先找到的句柄）；

然后就是通过child table去找子对象的句柄（下面的select只有一个child，上面select的有俩个child）；

接着根据子游标中的heap0的table中的dependency table找到涉及的table（下面的select涉及到俩个表对象，上面的select只涉及一个表对象）。

现在我们再来看下data blocks pointer：



这个图里的object information就是放在heap 0中的，之后的叫做heap1…heapn，这些都叫做data heap。各个data heap是独立的，存放着真实数据，是库缓存中一块连续的内存区域。

这些data heap都需要通过heap 0来找到。

### Library Cache Latch的演进

#### Mutex替代latch

因为latch一次要管理多个对象，这样即使俩个访问访问的对象不一样，也有可能发生争用和等待，所以在11g以后，使用mutex替换掉了之前很大一部分latch，library cache latch就是其中一个。

以下是摘自盖老师的讲义，暂时摘抄，以后一定搞清楚，包括library cache dump。

在 Oracle 11g中 Library cache mutex 替代了 Library Cache Latch，Library cache Mutex \* 用于管理 Hash Bucket ：

1. Library Cache: Mutex S：  
   试图在 library cache hash bucket 上获得 S 模式的共享锁，  
   请求的 mutex 已经被其他进程以 incompatible 模式持有，或者 "in flux”
2. Library cache: mutex X：  
   试图在 library cache hash bucket 上获得 X 模式的排它锁，  
   请求的 mutex 已经被其他进程以 incompatible 模式持有，或者 "in flux”

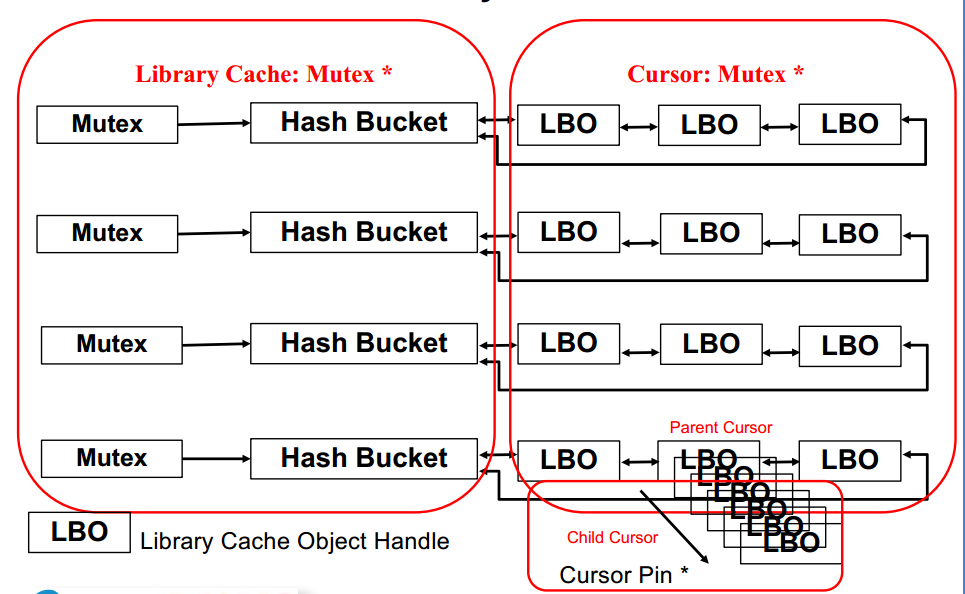
等待事件 cursor: mutex \* 被用于保护父游标和游标统计信息操作。

1. cursor: mutex S：  
   试图在 Parent 游标 或者 V$SQLSTATS bucket 上获得共享 Mutex，Mutex “in flux” 必须等其他进程完成其 shared get。
2. cursor: mutex X：  
   试图在 Parent 游标 或者 V$SQLSTATS bucket 上获得排他 Mutex，请求的Mutex 已经被其他进程以 X 或 S 模式持有，在加载新的 Child Cursor ，修改 V$SQLSTATS bucket，更新绑定变量数据时被持有；

等待事件 cursor:pin \* 用于针对游标的 pin 操作：

1. cursor: pin S  
   试图以 S 模式 Pin 某个 Cursor，但是该Cursor正在被 Pining ，也就是 “in flux” ，必须等待这个过程完成，才能被共享Pin。
2. cursor: pin X  
   试图以 X 模式 Pin 某个 Cursor ，但是该Cursor已经被其他会话以 X 模式或者 S 模式 Pinned。必须等待其他会话释放 Pin，才能继续。
3. cursor: pin S wait on X  
   试图以 S 模式 Pin 某个 Cursor ，但是某个会话已经以 X 模式 Pinned，正在执行 Loading，也就是 Parsing。

下图完美的将上述3类等待事件与library cache objects融合整理：



#### Library Latch的转储

Library Cache Dump是通过转储 Library Cache 的信息，可以分析其结构，了解其原理，但是需要谨慎，不能在生产系统中轻易使用。

ALTER SESSION 方式，以下命令以 10 级别转储

alter session set events 'immediate trace name library\_cache level 10'

使用 oradebug 方式

oradebug setmypid

oradebug unlimit

oradebug dump library\_cache <level>

不同级别转储的信息不同，以下是级别和信息的列表

Level 1 Dumps library cache statistics

Level 2 Dumps hash table summary

Level 4 Dumps library cache objects with basic information

Level 8 Dumps objects with detailed information (including child references, pin waiters, etc.)

Level 16 Dumps heap sizes (can be latch intensive )

Level 32 Dumps heap information

以上级别可以结合使用，比如 level 11 就包含了 8+2+1 几个级别的信息。

可以用如下sql转储一个特定的heap：

alter session set events 'immediate trace name heapdump\_addr level 2,addr 0x89d8a918';

## SHARED CURSOR

### 存在哪儿

Shared cursor是存在SGA的Library cache当中的一种库缓存对象（Library Cache Object）。上面我们其实已经提到了，SQL和匿名PL/SQL的namespace都是CRSR。

### 存些啥

Shared cursor中会存储SQL文本、解析树、执行计划、涉及的对象定义、使用的绑定变量的类型和长度。

### 咋分类的

不管是父游标还是子游标，namespace都是CRSR。

#### Parent Cursor

父游标的信息可以在v$sqlarea中可以查到，该视图我们可以先暂时关注如下几个列：

1.sql\_id：父游标在库缓存中的sql标识，所以sql\_id就是父游标的标识。这个sqlid是通过dbms\_sqltune\_util0.sqltext\_to\_sqlid（）对sql文本||char(0)处理得来的：

select dbms\_sqltune\_util0.sqltext\_to\_sqlid('sql\_text'||chr(0)) sql\_id from dual;

我们验证一下：

SQL> select sql\_id,dbms\_sqltune\_util0.sqltext\_to\_sqlid(sql\_fulltext||chr(0)) sql\_id\_2 from v$sqlarea where rownum<5;

SQL\_ID SQL\_ID\_2

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

2z0udr4rc402m 2z0udr4rc402m

gngtvs38t0060 gngtvs38t0060

06gfrprr7w0r2 06gfrprr7w0r2

0m167zdnfn2sf 0m167zdnfn2sf

2.sql\_text/sql\_fulltext：sql文本，父游标中的name部分。

3.version\_count：拥有的子游标的个数。

4.adress：父游标对象句柄的地址

5.hash\_value:这个是sql\_id的hash值，就是说这个值是根据sql\_id算出来的而不是sql文本。

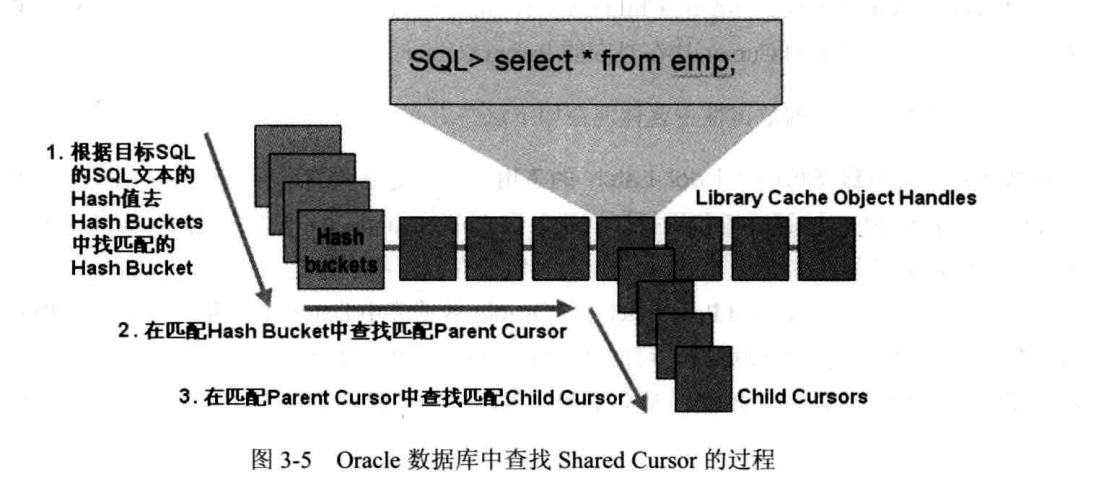
这个视图是一个sql\_id一条记录，所以这张视图是查父游标信息的。

#### Child Cursor

注意到上面提到父游标的name属性就是sql文本。那子游标的name属性是什么呢？是空。

子游标的信息可以在v$sql中查到，该视图为啥说是查子游标的呢，因为这个视图一个child cursor一行（这个视图sql跑完更新，sql没跑完就5秒一更新，这个是adptive sql plan的特性）。

### 咋干活的



崔华老师这张图说的很清楚。然后我们说一些细节。

当找了到对应sql的父游标之后，根据父游标的heap 0中的child table找到子游标库缓存对象句柄地址，然后在子游标的heap 0中找到子游标的heap 6，子游标的heap 6中放的就是执行计划和解析树。

所以说，父游表主要就是用来找子游标的，而具体相关的执行计划、涉及的对象都存储子在子游标中。

## SESSION CURSOR

Session cursor是用来在当前session中解析和执行sql。

既然是session级别的，显然它应当存在于PGA中。

### SESSION CURSOR的特点

Session cursor有如下几个特点：

1. session cursor与session一一对应，不同的session之间是不能共享session cursor的，这与shared cursor不同；
2. session cursor是有生命周期的，在其使用过程中都要至少经历一次open、parse、bind、execute、fetch和close中的几个或全部阶段。用过的session cursor可能会缓存在对应session的pga中；
3. session cursor也是以hash表的方式缓存在PGA中，所以访问方式与shared curor雷同；
4. session cursor会把目标sql所涉及的数据从buffer cache对应的数据块读到PGA中，在PGA中进行排序、hash join等处理，最后将最终处理结果返回给用户；
5. 一个session cursor只能对应一个shared cursor，而一个shared cursor可以对应多个session cursor。

### SESSION DUMP

首先打开一个会话，看下本会话的sid：

SQL> select sid from v$mystat where rownum=1;

SID

---------

142

然后执行一个查询：

SQL> select ename from emp where empno='7788';

ENAME

----------

SCOTT

我们查看v$sqlarea视图：

select sql\_text,

sql\_id,

first\_load\_time,

parsing\_schema\_id,

parsing\_schema\_name,

address,

hash\_value,

module,

last\_active\_child\_address,

to\_char(last\_load\_time,'yyyy-mm-dd/hh24:mi:ss') last\_load\_time

from v$sqlarea

where sql\_text like 'select ename from emp%';





然后执行：

SQL> alter session set events 'immediate trace name ERRORSTACK level 3';

Session altered.

SQL> alter session set events 'immediate trace name ERRORSTACK off';

Session altered.

这时候alert日志显示：



然后我们more这个文件：

先搜索sql\_id：62hk5046kqb12



这张截图我根据我的理解简单的一个红圈一个红圈分析一下，确实是不会errorstack。

首先我们看到sql\_id旁边的xsc=0x7fc1bd10d478，这个xsc我也不知道具体是啥，但是先记下，因为后面会看到，它很有用。

之后我们看到LibraryHandle: Address=0xb7b26fd0 Hash=d2b2c22，注意这个adress是不是跟我们上面查询的adress一模一样，而且这个hash，我们可以算一下：

SQL> select to\_number('d2b2c22','xxxxxxxxxxx') hash\_value from dual;

HASH\_VALUE

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

220933154

跟上面查询出来的hash value一模一样。也就是说这个库缓存句柄正如我们之前所说的，就是parent cursor。

我们接着可以看到，ObjectName: Name=select ename from emp where empno='7788'，这个主要注意高亮部分。我们前面说到父游标的name属性是sql文本，果然是这样。

同时我们往后看，可以看到Namespace=SQL Type=CURSOR ContainerId=1，这里namespace记录的跟之前说的有出入，我不知道是不是版本问题（我是12.2），我们关键看到containerid是1，也就是说这个父游标是cdb级别的，这很好理解，也就是说不同pdb，甚至cdb下的相同的sql文本都可以找到这个父游标这里来。

再接着走，对于lock这里先不深究，再后面我们看到Timestamp: Current=06-16-2018 14:53:34，这个我没深究，应该是对应着上面的last load time，或者是last active time。

继续往下看，我们注意到了ChildTable，这个应该就是之前说过的父游标中的heap0中的table属性中的childtable属性。我们看到这个属性里面记录了一条Handle=0xb7b26ba8。这样接着继续往下看，才知道这个handle是什么。

接着就来到了child cursor，我们看到：LibraryHandle: Address=0xb7b26ba8，child cursor的库缓存句柄地址就是上面parent cursor的child table中记录的handle。

接着又看到，这一次，child cursor的name是没有的，其他的跟父游标一样，但是containerid是3（是我的第一个pdb），也就是说到了子游标才开始判断更详细的内容。

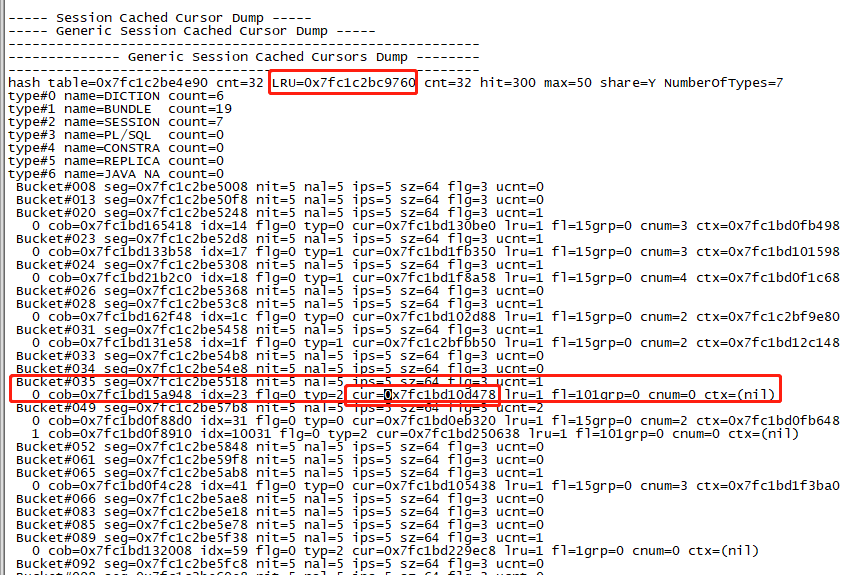
接着是namespace dump，这里我有点儿蒙，难道这才是真正的namespace？

这里首先就可以看到Child Cursor: Heap0=0xb23c4888 Heap6=0xc1717fc8，这里也就是我们前面说的，再通过child cursor的heap0找到child cursor的heap6。

接下来的的parent cursor的内容，就非常简明的概括，包括有几个子游标、子游标地址、module等等，这些基本都可以再v$sqlarea中查到。

最后这俩红圈，第一个我们看到cursor instantiation=0x7fc1bd10d478，游标实例化后是这个“代号”，和我们最开始看到的xsc一模一样。后面子游标出现的ctx=0xc1717fc8又和heap 6一模一样，这个是真不知道，这个后面研究。

我们这里重点关注这个游标实例化后的“代号”，我们搜索它：



首先我们看到session cursor的存在还是清除也是基于LRU算法的，然后我们看重点，这部分是session cached curors的dump，也就是说这部分是真正的session cursor部分。

前面我们说过，session cursor的访问方式也是按hash表的访问方式，所以这里我们发现在35号bucket中cur=0x7fc1bd10d478正是父游标记录的那个“代号”，而父游标在SGA中，也就是说Oracle可以通过在PGA中的session cursor找到SGA中parent cursor。

这一部分暂时就研究这么多，更多更深入的东西，后续再慢慢研究。

### SESSION CURSOR的参数

#### OPEN\_CURSORS

参数open\_cursors用于设定单个session中可以同时以open状态并存的session cursor的最大数量。

SQL> show parameter open\_cursors

NAME TYPE VALUE

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

open\_cursors integer 300

通过v$OPEN\_CURSOR视图可以查看到当前已open的和已缓存的cursor。

SQL> select sid from v$mystat where rownum=1;

SID

----------

758

SQL> select count(\*) from v$open\_cursor where sid=758;

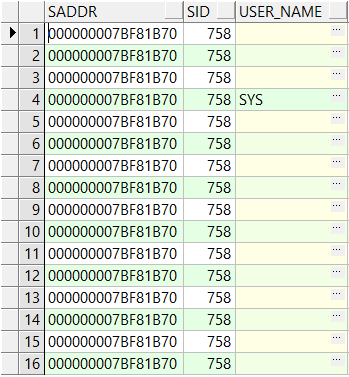
COUNT(\*)

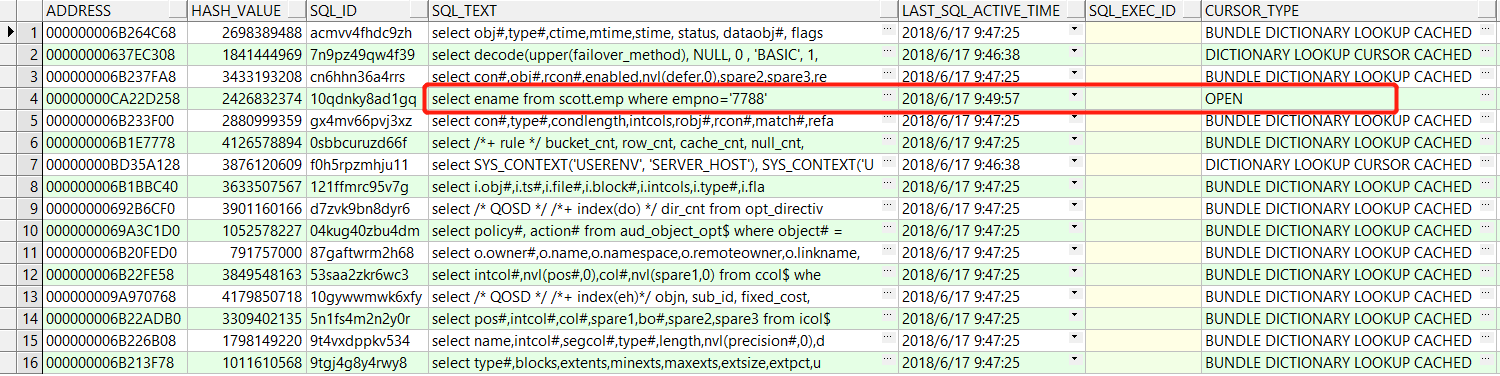
----------

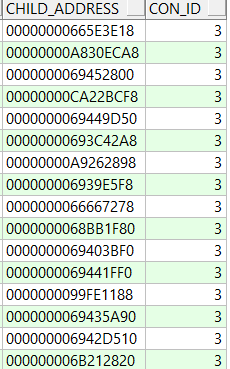
16

然后看下这个视图里有点儿啥：

select \* from v$open\_cursor where sid=758;







可以看到某个session中open或缓存的sql的sqlid及sql文本，包括父游标地址等等。

也可以这样查：

SQL> select statistic#,name,value,con\_id from v$sysstat where name='opened cursors current';

STATISTIC# NAME VALUE CON\_ID

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

5 opened cursors current 19 3

#### SESSION\_CACHED\_CURSORS

SQL> show parameter session\_cached\_cursors

NAME TYPE VALUE

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

session\_cached\_cursors integer 50

这个参数就是控制单个session的PGA中最多可以同时缓存多少session cursor。

如果这个值是0，那session cursor在sql执行完就会被ORACLE正常close。

如果这个值不为0，且sql被ORACLE解析超过4次，那sql在第5次执行完成后，ORACLE会把它标记为soft closed，同时将其缓存在当前的PGA中。

如果这个值不为0，且满足缓存条件的sql超过了这个数，如我们在前面session cached cursors dump中看到的，ORACLE同样会采用LRU算法对不常用的cursor进行清理，来给新的session cursor腾位置。

下面我们对第二种情况进行一下实验，先开一个session准备执行实验sql：

SQL> select sid from v$mystat where rownum=1;

SID

----------

628

再开一个session进行实验查询：

SQL> select sql\_text,cursor\_type from v$open\_cursor where sid=628 and sql\_text ='select count(1) from emp';

no rows selected

现在我们可以看到，还没执行，这条sql显然不会出现在v$open\_cursor中。

现在我们来第1次执行实验sql：

SQL> select count(1) from emp;

COUNT(1)

----------

14

然后进行观察：

SQL> select sql\_text,cursor\_type from v$open\_cursor where sid=628 and sql\_text ='select count(1) from emp';

SQL\_TEXT CURSOR\_TYPE

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

select count(1) from emp OPEN

我们看到现在有记录了，而且cursor\_type为OPEN。

第2次执行，再次查询：

SQL> /

SQL\_TEXT CURSOR\_TYPE

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

select count(1) from emp OPEN

第3次执行，接着进行查询：

SQL> /

SQL\_TEXT CURSOR\_TYPE

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

select count(1) from emp OPEN

还是OPEN，第4次执行：

SQL> /

SQL\_TEXT CURSOR\_TYPE

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

select count(1) from emp DICTIONARY LOOK UP CURSOR CACHED

我们看到cursor type发生了改变，然后我们第5次执行：

SQL> /

SQL\_TEXT CURSOR\_TYPE

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

select count(1) from emp SESSION CURSOR CACHED

现在我们看到cusor type为SESSION CURSOR CACHED，说明此时该session cursor已被缓存到PGA了。

## 三种解析方式

在进行后续陈述之前，先进行俩个申明：

1. 虽然我的实验版本是12.2，但是这里原理还是按照崔老师书上老的原理来记，因为大同小异。也就是说下面依然会通过latch来解释，而不是mutex。
2. 下面的解释是按照正常情况下session\_cached\_cursors这个参数不为0作为前提的。

### 硬解析

一个sql执行，首先在自己会话的PGA中找session curor，硬解析是肯定找不到的，这里我们后面的实验会很清楚的表明这一点。

然后ORACLE先持有Library Cache Latch，去找对应的parent cursor（这里说白了就是用sql文本去找），这里要分俩种情况。

如果找到了，那也肯定找不到对应的child cursor，不然就不叫硬解析了。这种情况下，ORACLE先释放刚才持有的Library Cache Latch，因为这个latch就是为了找对应cursor才持有的，找不到就释放喽。

但紧接着，ORACLE会再次持有一个Library Cache Latch，这一次是为了创建一个新child cursor而持有的。

但是创建child cursor，就要由SGA分配一块内存出来放这个cursor，那么就要在持有Library Cache Latch的同时，再申请一个Shared Pool Latch。

再然后才开始把child cursor创建出来，同时把session cursor也建出来，然后先把Shared Pool Latch释放了，再把Library Cache Latch释放了。

SQL执行完之后，因为session\_cached\_cursors不为0，所以这里PGA不会急着把session cursor关闭掉，可能该sql接着被多次执行，则该session cursor就被缓存再了PGA中，这一过程前面是有详细解释的。

那如果直接就没找到对应的parent cursor，那比上面的步骤就多了个建parent cursor的过程，这个也是要先释放查找cursor用的Library Cache Latch，再持有创建cursor用的Library Cache Latch，再持有Shared Pool Latch，建完parent cursor、child cursor（一对儿），然后把session cursor也生成出来，再释放Shared Pool Latch，再释放Library Cache Latch。

### 软解析

### 软软解析

### 实验

首先，还是老办法，开一个会话做实验，另一个会话开效果，下面这个是实验会话：

SQL> select sid from v$mystat where rownum=1;

SID

----------

511

先建个实验表，这个表没有任何索引：

SQL> create table emp1 as select \* from emp;

Table created.

再另一个会话中先查看v$open\_cursor视图：

SQL> select \* from v$open\_cursor where sid=627 and sql\_text like 'select ename from emp1 where empno=%';

no rows selected

然后实验会话执行实验sql：

SQL> select ename from emp1 where empno='7900';

ENAME

----------

JAMES

观察窗口看实验效果：

SQL> select address,hash\_value,sql\_id,sql\_text,cursor\_type,child\_address from v$open\_cursor where sid=627 and sql\_text like 'select ename from emp1 where empno=%';

ADDRESS HASH\_VALUE SQL\_ID SQL\_TEXT

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

0000000061386348 152036724 9gbstun4hztbn select ename from emp1 where empno='7900'

CURSOR\_TYPE CHILD\_ADDRESS

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

OPEN 00000000B5BE13E0

现在游标类型是open，我们不多说，直接再执行它4次，让它缓存到PGA中：

ADDRESS HASH\_VALUE SQL\_ID SQL\_TEXT CURSOR\_TYPE CHILD\_ADDRESS

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

0000000061386348 152036724 9gbstun4hztbn select ename from emp1 where empno='7900 SESSION CURSOR CACHED 00000000B5BE13E0

此时我们看到这个游标已经缓存在PGA中了。

#### 修改表结构或增删索引

现在，如果我们这里给emp1加个索引，执行计划一定会由原来的全表扫描变为索引扫描，同时，由于sql文本并没有改变，所以该父游标应当还可以找到，那么这个session cursor还能不能被重用：

SQL> create index idx\_emp1 on emp1(empno);

Index created.

我们看下观察窗口：

SQL> /

no rows selected

没了！为什么？我们去查一下v$sqlarea：

SQL> select sql\_id,version\_count,open\_versions,object\_status from v$sqlarea where sql\_id='9gbstun4hztbn';

SQL\_ID VERSION\_COUNT OPEN\_VERSIONS OBJECT\_STATUS

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

9gbstun4hztbn 1 0 INVALID\_UNAUTH

原来是这样，其实表的索引也算是表的一部分结构，只要对这个表在结构上动了手脚，无论是添加索引还是修改表结构，都会是父游标状态变为INVALID\_UNAUTH。同时，会导致对应的session cursor被从PGA中踢除。

修改表结构的也是完全一样的效果，这里就不演示了。

#### 制造子游标不共享

前面的实验可以看到，表结构修改了，对应session cursor都会被踢出PGA，更别说重用了。那么如果是子游标不共享，那这个session cursor会因为父游标没有发生变化而被重用么？

我们在实验窗口继续执行实验sql（这时候sql是走索引的，虽然这跟本次实验关系不大）：

SQL> select ename from emp1 where empno='7900';

ENAME

----------

JAMES

这里我们还是承上个实验，看一下父游标的变化：

SQL\_ID VERSION\_COUNT OPEN\_VERSIONS OBJECT\_STATUS

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

9gbstun4hztbn 1 1 VALID

我们看到现在对象状态变为了有效，同时open的version也变成了1。

我们这里还是再执行4次，让它缓存到PGA中：



然后我们制造子游标不共享，我这里通过修改statistics\_level的值，来制造子游标不共享（顺带说一下，这属于optimizer mismatch(12)）：

SQL> alter session set statistics\_level=ALL;

Session altered.

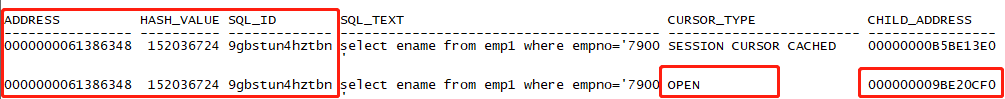
然后我们执行实验sql，先看下父游标的变化：

SQL\_ID VERSION\_COUNT OPEN\_VERSIONS OBJECT\_STATUS

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

9gbstun4hztbn 2 2 VALID

注意看，此时version count变为2，open versions也是2。说明我们确实达到了目的，现在我们再来看PGA中session cursor的情况：



现在我们看的很清楚，虽然父游标完全一样，但是因为子游标不能共享，ORACLE便会生成一个新的session cursor。

所以说，软软解析的前提一定是可以进行软解析。