Oracle11g RAC 数据库中的 Master 实例、Owner 实例和 Past Image 的概念

Botang 唐波

摘要

本文探索 Oracle RAC 数据库的内部原理。探索 Oracle RAC 数据库的调优源头，弄清“Master 实例”、“Owner 实例”和“Past Image”的概念。

目录

正文

1. Oracle RAC 数据库服务器实例间的网络通信

下图描述了 Oracle RAC 数据库服务器实例间内连网通信体系结构（见图 1）：

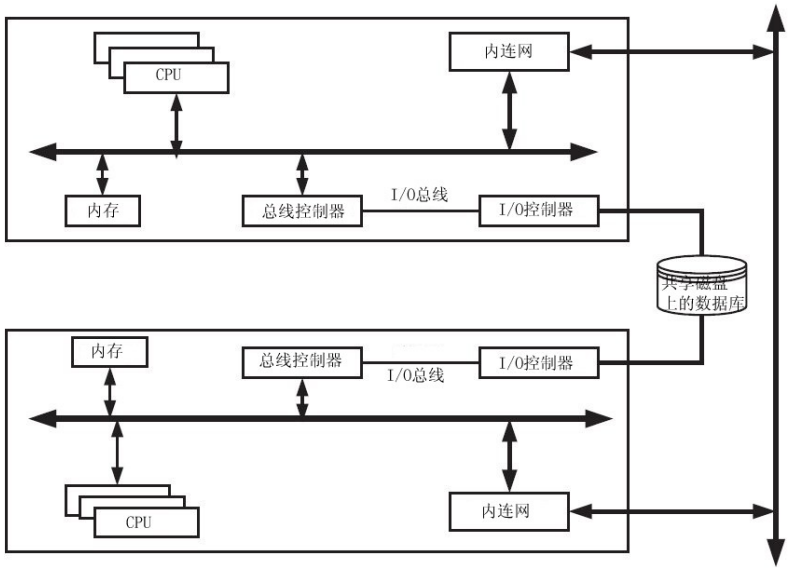

图 1：Oracle RAC 数据库服务器实例间内连网通信体系结构示意图

Oracle RAC 数据库服务器调优的重要目标就是尽量减少实例间没有必要的内连网通信量。下节所讨论的“Oracle RAC 数据库服务器中最重要的内存结构：GRD 和 GRD 中的资源”，都是通过内连网通信来维护。

2．Oracle RAC 数据库服务器中最重要的内存结构：GRD 和 GRD 中的资源

在 RAC 的体系结构中（见图 2）,全局资源目录（Global Resource Directory 简称 GRD）是 Oracle RAC 数据库服务器中最重要的内存结构。它是一套哈希分布于各个实例间由被称为“gcs mastership buckets”、“gcs res hash bucket”和“gcs resources”等内存结构组成的的元数据集（见表 1）。这个哈希分布元数据集用以描述 Oracle RAC 数据库服务器中数据块的状态、属主信息以及数据块内部和数据块自身的锁信息。GRD 分布在所有实例的共享池中，每个实例维护 GRD 的一部份。所有实例维护的 GRD 合起来形成哈希分布式的整体集。GRD 内部包含“转换队列”和“写队列”，这两个队列被 GCS（Global Cache Service）和 GES（Global Enqueue Service）维护。所有维护信息通过前面介绍过的内连网传输。

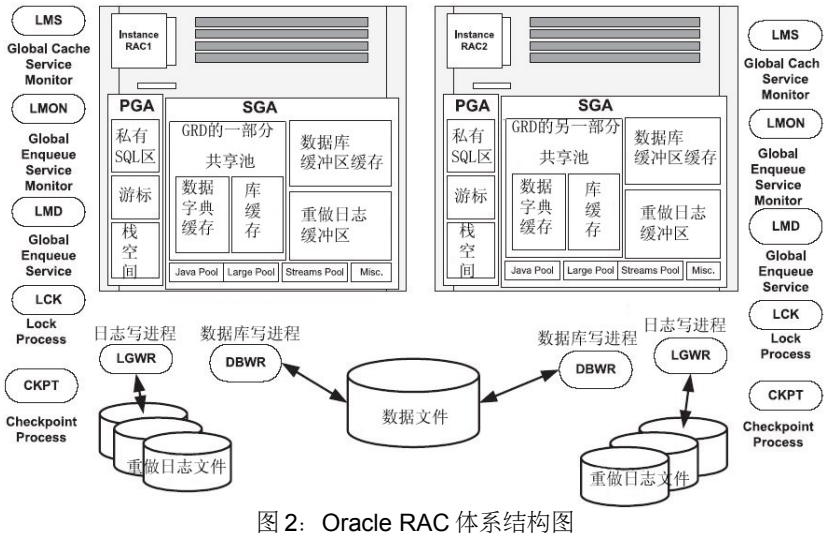

图 2：Oracle RAC 体系结构图

	POOL	NAME	BYTES
1	shared pool	KCL buffer header	9664
2	shared pool	KCL instance cache transf	262144
3	shared pool	KCL lock contexts	14420
4	shared pool	KCL lock state	3584
5	shared pool	KCL name table	262144
6	shared pool	KCL offline scn array	1608
7	shared pool	KCL partition table	720896
8	shared pool	KCL region array	8
9	shared pool	gcs I/O statistics struct	32
10	shared pool	gcs affinity	4108
11	shared pool	gcs close obj	4104
12	shared pool	gcs commit sga state	67596
13	shared pool	gcs mastership buckets	4608
14	shared pool	gcs opaque in	4028

15	shared pool	gcs res hash bucket	32768
16	shared pool	gcs res latch table	15360
17	shared pool	gcs resource freelist arr	272
18	shared pool	gcs resource freelist dyn	32
19	shared pool	gcs resources	4812504
20	shared pool	gcs scan queue array	216
21	shared pool	gcs shadow locks dyn seg	32
22	shared pool	gcs shadow locks freelist	272
23	shared pool	gcs shadows	3418328

表 1：共享池中的与 RAC 相关内存结构，数据来源于：V$SGASTAT

理论上每个数据块都有对应于它的 GRD 信息：无论它当前存在于某个实例的数据库缓冲区缓存中还是已经被写回硬盘数据文件中。这些信息加上管理这些信息的锁合在一起称为“资源”。

既然 GRD 是分布式的整体，对于一个数据块而言，管理该数据块的状态和属主信息以及数据块内部和数据块自身的锁信息的实例只有一个。这个实例就被称作为该数据块（或更准确地说：资源）的 Master 实例。Oracle 这样做是为了将数据块状态和属主等信息均衡地哈希分布在不同实例上。从 10g 以后版本开始，数据块的状态和属主等信息被存储成每 128 个块的信息一个 master 单元，即 128 个数据块的状态和属主等信息构成一个“gcs mastership bucket”。但是要说明以下：一个“gcs mastership bucket”不一定要存满 128 个块的状态和属主等信息。这样就能理解：超过 128 个块的表的数据块可以被多个实例分布式地分段 master。如果发生自动 Remaster（Object Affinity and Dynamic Remastering 引起）或手工 Remaster（oradebug 命令），整个对象将作为 master 单元而不进行多个实例分布式地分段 master：即不管表多大，它的数据块都由同一个实例 master。另外，任何时候 undo 段整段必需由同一个实例 master。

数据库缓冲区缓存中拥有某个数据块内存拷贝的实例被称作该数据块的 Owner 实例。Owner 实例的个数可以是 0（最小）到集群节点总数（最大）中的任何数值。如果该数据块内存拷贝在多个实例的数据库缓冲区缓存中同时被找到，也就是说该数据块有多个 Owner 实例，那么证明在近期有多个实例先后修改或访问过它。在所有这些该数据块的内存拷贝中，SCN 最大的那个内存拷贝被称为 Current Image（XI），SCN 不是最大的那些内存拷贝就都被统统称为该数据块的 Past Image（PI）。PI 的存在主要是为了在实例恢复过程中能被利用来减少实例交叉恢复的时间。如果由于检查点事件 XI 被写回硬盘，那么所有它所对应的所有 PI 都将被从内存中直接 flush 掉。还有一点值得注意的是：XI 和 PI 都可以包含各自的 new value 和 old value。通常 old value 在 Oracle 文档中又被称为 Before Image 即：BI。

最后说一下：对于回滚段上的 BI 而言，激活了一个回滚段的实例立刻成为该段的 master 实例。因为该回滚段将会被打开这个回滚段的实例所使用，用以写入事务产生的 BI。因为回滚段没有真正的 object_id，所以在 X$KJBL 表中 Oracle 使用“4294950912+回滚段号”作为该回滚段的 object_id。

以下用一幅图来形象描绘了一下数据库服务器其中一个实例的 GRD 构成（见图 3）：

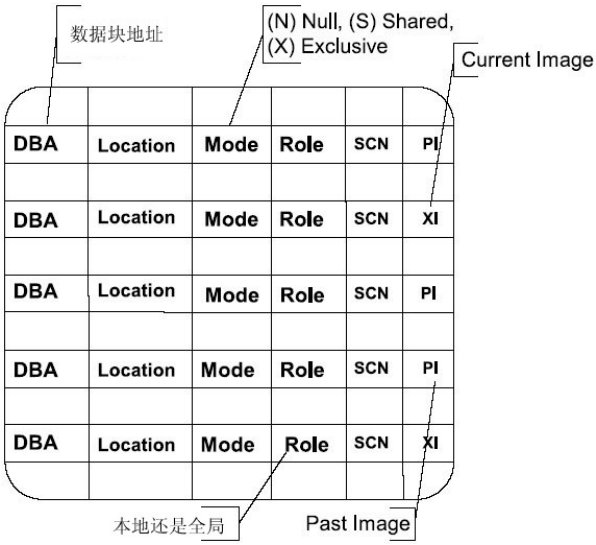

图 3：某个实例的 GRD 构成示意图（其中：Null 表示空锁，Shared 表示共享锁，Exclusive 表示独占锁）

3．数据块上的 BL 锁和 PR 授权

对于 RAC，Buffer cache 中的 Buffer 以及硬盘数据文件上的块，都是锁管理的对象之一。这种锁就被称为“BL 锁”，单实例的数据库上没有这种类型的锁。对 Buffer 和硬盘数据文件上的块的修改或访问都要先得到它的 master 实例的“Protected Read”授权，简称 PR 授权（锁状态为：KJUSERPR）。PR 授权就是获得 BL 锁的过程。下面列出查看 BL 锁和 PR 请求的 SQL 语句（见表 2），同时形象展现 GRD 中的 BL 锁和 PR 请求的动态过程（见图 4）。如前所述 BL 锁和被其锁住的信息合在一起就是一个 BL 锁资源。

```
select  * from v$ges_resource  where resource_name like '%BL%';
select  * from v$ges_enqueue  where  resource_name2 like '%BL%';
```

注：v$ges_resource 和 v$dlm_ress 是相同的。

表 2：查看 BL 锁资源和 PR 请求的 SQL 语句

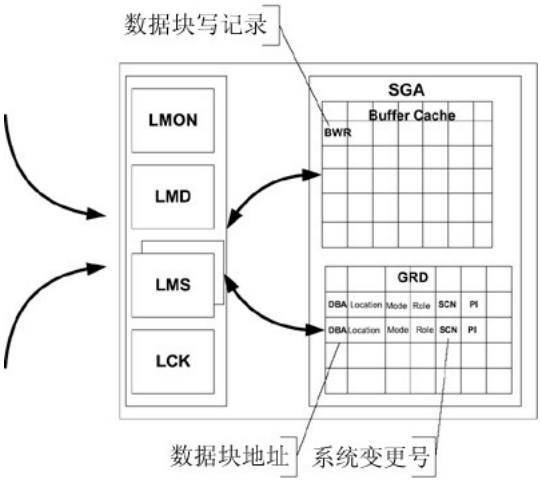

图 4：Buffer Cache 中的数据块变化前必需询问 GRD

某个实例是否有申请了 BL 锁资源，关键是理解该实例对数据块 Open 和 Buffer Access 之间的区别。如果缓存中的数据块已经处于适当的模式，就没有必要在数据块上再打开 BL 锁。所以如果会话反复存取同一个数据块而不请求

额外的 BL 锁，那么在 X$OBJECT_POLICY_STATISTICS 中 BL 锁的 Open 计数就不会增加。这个计数可以从以下的查询语句中查出（见表 3）。

实验第 1 步：
查看实验用表 t04209_uname 的定义
在任一个实例上执行:

```
select t.TABLE_NAME,t.COLUMN_NAME,t.DATA_LENGTH,t.DATA_PRECISION
  from dba_tab_columns t where t.owner='HR' and t.table_name='T04209_UNAME';
```

	TABLE_NAME	COLUMN_NAME	DATA_LENGTH	DATA_PRECISION
1	T04209_UNAME	UNAME	60	
2	T04209_UNAME	UVALUE	22	9

实验第 2 步：
查看实验用表 t04209_uname 的内容
在任一个实例上执行:

```
select  * from hr.t04209_uname;
```

	UNAME	UVALUE
1	a1	1
2	a2	2
3	a3	3
4	a4	4
5	a5	5
6	a6	6
7	a7	7
8	a8	8
9	a9	9
10	a10	10
……	……	……

共 10 万行

实验第 3 步：
查看实验用表 t04209_uname 的数据块个数
在任一个实例上执行:

```
select s.segment_name,s.blocks from dba_segments s where s.owner='HR' and s.segment_name='T04209_UNAME';
```

	SEGMENT_NAME	BLOCKS
1	T04209_UNAME	256 ←该表共有 256 个块

实验第 4 步：
查看实验用表 t04209_uname 的 object_id
在任一个实例上执行:

```
select o.object_name, o.object_id from dba_objects o where o.owner='HR' and o.object_name='T04209_UNAME';
```

	OBJECT_NAME	OBJECT_ID
1	T04209_UNAME	52533

实验第 5 步：
在任一个实例上执行:
select * from x$OBJECT_POLICY_STATISTICS where object=52533;
无输出

实验第 6 步：
在实例 1 上 update 这个表 t04209_uname：
update hr.t04209_uname set uvalue=2 where uname='a1';
update hr.t04209_uname set uvalue=3 where uname='a2';
update hr.t04209_uname set uvalue=4 where uname='a3';
update hr.t04209_uname set uvalue=5 where uname='a4';
update hr.t04209_uname set uvalue=6 where uname='a5';
update hr.t04209_uname set uvalue=7 where uname='a6';
update hr.t04209_uname set uvalue=8 where uname='a7';
……
update hr.t04209_uname set uvalue=99996 where uname='a99995';
update hr.t04209_uname set uvalue=99997 where uname='a99996';
update hr.t04209_uname set uvalue=99998 where uname='a99997';
update hr.t04209_uname set uvalue=99999 where uname='a99998';
update hr.t04209_uname set uvalue=100000 where uname='a99999';
update hr.t04209_uname set uvalue=100001 where uname='a100000';
共 10 万行

实验第 7 步：
在任一个实例上执行:
select * from x$OBJECT_POLICY_STATISTICS where object=52533;

ADDR	INDX	INST_ID	OBJECT	NODE	OPENS
1 B7F4C4C8	7	2 ↖ INST_ID 为 2 代表 Mater 实例为实例 2	52533	1 ↖NODE 为 1 代表 Owner 实例为实例 1	245 ←BL Open 次数，这个表共有 256 个块几乎占全部。说明 update 大部分的行。

实验第 8 步：
等实例 1update 结束后，再查 x$OBJECT_POLICY_STATISTICS 没有输出。
重新在实例 1 上做 update，x$OBJECT_POLICY_STATISTICS 仍然没有输出。
这就说明：如果缓存中的数据块已经处于适当的模式，就没有必要在数据块上再打开 BL 锁。所以如果会话反复存取同一个数据块而不请求额外的 BL 锁，那么在 x$OBJECT_POLICY_STATISTICS 中 BL 锁的 Open 计数就不会增加。

实验第 9 步：
在实例 1 上 Alter system flush buffer_cache 后，再做同样的 update。

实验第 10 步：
在任一个实例上执行:
select * from x$OBJECT_POLICY_STATISTICS where object=52533;

	ADDR	INDX	INST_ID	OBJECT	NODE	OPENS1
1	B688A1D0	7	2↖INST_ID 为 2 代表 Mater 实例为实例 2	52533	1↖NODE 为 1 代表 Owner 实例为实例 1	248 ←BL Open 次数

实验第 11 步：
为了使情况更复杂点,在实例 2 同时进行（保证实例 1 还在执行刚才的 update）：

```
update hr.t04209_uname set uvalue=9 where uname='a8';
update hr.t04209_uname set uvalue=99 where uname='a98';
update hr.t04209_uname set uvalue=999 where uname='a998';
update hr.t04209_uname set uvalue=9999 where uname='a9998';
update hr.t04209_uname set uvalue=99999 where uname='a99998';
update hr.t04209_uname set uvalue=99998 where uname='a99997';
update hr.t04209_uname set uvalue=99997 where uname='a99996';
update hr.t04209_uname set uvalue=99996 where uname='a99995';
update hr.t04209_uname set uvalue=99995 where uname='a99994';
……
update hr.t04209_uname set uvalue=10006 where uname='a10005';
update hr.t04209_uname set uvalue=10005 where uname='a10004';
update hr.t04209_uname set uvalue=10004 where uname='a10003';
update hr.t04209_uname set uvalue=10003 where uname='a10002';
update hr.t04209_uname set uvalue=10002 where uname='a10001';
update hr.t04209_uname set uvalue=10001 where uname='a10000';
update hr.t04209_uname set uvalue=10000 where uname='a9999';
update hr.t04209_uname set uvalue=100001 where uname='a100000';
update hr.t04209_uname set uvalue=100000 where uname='a99999';
```

共 10 万行，这 10 万行 update 是不按顺序打乱后的组合。

实验第 12 步：
在任一个实例上执行:

```
select  * from x$OBJECT_POLICY_STATISTICS where object=52533;
```

	ADDR	INDX	INST_ID	OBJECT	NODE	OPENS
1	B688A1D0	20	2	52533	1	2562← 由于两个实例同时更改，产生了大量的 BL Open
2	B688A1D0	21	2	52533	2	1053 ↙BL Open 的增加是为了进行 gc cr 读

实验第 13 步：
在任一个实例上执行:

```
select  to_char(52533,'xxxxxxxx')  from dual;
```

	TO_CHAR(52533,'XXXXXXXX')
1	cd35

实验第 14 步：
在任一个实例上执行: 查看两个实例此时的资源状态

```
select  inst_id ,  resource_name,  on_convert_q, on_grant_q,master_node, next_cvt_level  from gv$ges_resource  where  resource_name like  '[0xcd35]%';
```

	INST_ID	RESOURCE_NAME	ON_CONVERT_Q	ON_GRANT_Q	MASTER_NODE	NEXT_CVT_LEVEL
1	2	[0xcd35][0x0],[TM]	0	1	0	KJUSERNL
2	2	[0xcd35][0xc09281d],[IV]	0	1	0	KJUSERNL
3	1	[0xcd35][0x0],[TM]	0	1	0	KJUSERNL
4	1	[0xcd35][0xc09281d],[IV]	0	1	0	KJUSERNL

实验第 15 步：

在任一个实例上执行: 查看两个实例此时的锁状态

```
select inst_id,grant_level,request_level,resource_name2 ,PID , TRANSACTION_ID0 , TRANSACTION_ID1 from gv$ges_enqueue where resource_name2 like '52533%';
```

	INST_ID	GRANT_LEVEL	REQUEST_LEVEL	RESOURCE_NAME2	PID	TRANSACTION_ID0	TRANSACTION_ID1
1	1	KJUSERCW	KJUSERCW	52533,0,TM	27730	1835009	33
2	1	KJUSERCW	KJUSERCW	52533,0,TM	0	0	0
3	1	KJUSERPR	KJUSERPR	52533,201926685,IV	7283	0	0
4	1	KJUSERPR	KJUSERPR	52533,201926685,IV	0	0	0
5	2	KJUSERCW	KJUSERCW	52533,0,TM	9231	1703938	111
6	2	KJUSERPR	KJUSERPR	52533,201926685,IV	7200	0	0

KJUSERPR 是 BL 锁，KJUSERCW 是表级共享锁，KJUSERTM 是行级独占锁，KJUSERNL 是空锁，KJUSEREX 是 lock table 之类命令产生的独占锁

表 3：Buffer Cache 中的数据块变化前必需询问 GRD 的完整实验过程

由于表 3 中的实验后半段在两个实例上都对表 t04209_name 做了更改。对于两个实例各自回滚段上的 BI 而言，激活了一个回滚段的实例立刻成为该段的 master 实例。如上一节所述回滚段没有真正的 object_id，所以使用 4294950912+回滚段号作为该回滚段的 object_id。到此我们自然就会得出一个有意思的推论：一个事务产生的 new value 和 old value（BI）可以被两个不同的实例 master：因为实例 1 可以 update 一个实例 2master 的数据块，new value 的 master 自然是实例 2；old value 由于在实例 1 的回滚段上所以归实例 1master。这样推导下去诸如“gc [current/cr] [multiblock] request”、“gc [current/cr] [2/3]-way“、”gc [current/cr] block busy“、”gc [current/cr] grant 2-way“、”gc [current/cr] [block/grant] congested“和“gc [current/cr] [failure/retry]“等待事件中的 current（new value）和 cr（old value）分别对应的 master 实例有可能不是同一个。可以顺便验证一下（见表 4）：

实验第 16 步：
查看回滚段信息
在任一个实例上执行:

```
select rs.instance_num, rs.segment_name,rs.segment_id,rs.segment_id+4294950912 from dba_rollback_segs rs;
```

	INSTANCE_NUM	SEGMENT_NAME	SEGMENT_ID	RS.SEGMENT_ID+4294950912
1	1	SYSTEM	0	4294950912
2	1	_SYSSMU1$	1	4294950913
3	1	_SYSSMU2$	2	4294950914
4	1	_SYSSMU3$	3	4294950915
5	1	_SYSSMU4$	4	4294950916
6	1	_SYSSMU5$	5	4294950917
7	1	_SYSSMU6$	6	4294950918
8	1	_SYSSMU7$	7	4294950919
9	1	_SYSSMU8$	8	4294950920
10	1	_SYSSMU9$	9	4294950921
11	1	_SYSSMU10$	10	4294950922
12	2	_SYSSMU11$	11	4294950923
13	2	_SYSSMU12$	12	4294950924
14	2	_SYSSMU13$	13	4294950925
15	2	_SYSSMU14$	14	4294950926
16	2	_SYSSMU15$	15	4294950927
17	2	_SYSSMU16$	16	4294950928
18	2	_SYSSMU17$	17	4294950929
19	2	_SYSSMU18$	18	4294950930
20	2	_SYSSMU19$	19	4294950931
21	2	_SYSSMU20$	20	4294950932

实验第 17 步：
查看回滚段正被哪个实例 Remaster：
在任一个实例上执行:

```
select * from v$gcspfmaster_info;
```

	FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
1	0↖10.1 版本以前，是以文件作为 Remaster 单元，现在以对象作为单元，这里都是 0	4294950913	0	32767↖表示之前没有发生过 Remaster 事件	0
2	0	4294950914	0	32767	0
3	0	4294950915	0	32767	0
4	0	4294950916	0	32767	0
5	0	4294950917	0	32767	0
6	0	4294950918	0	32767	0
7	0	4294950919	0	32767	0
8	0	4294950920	0	32767	0
9	0	4294950921	0	32767	0
10	0	4294950922	0	32767	0
11	0	4294950923	1	32767	0
12	0	4294950924	1	32767	0
13	0	4294950925	1	32767	0
14	0	4294950926	1	32767	0
15	0	4294950927	1	32767	0

16	0	4294950928	1	32767	0
17	0	4294950929	1	32767	0
18	0	4294950930	1	32767	0
19	0	4294950931	1	32767	0
20	0	4294950932	1	32767	0

表 4：表 3 实验过程中伴随的 undo 段 GRD 信息

4．能够查出某个块 Master 实例和 Owner 实例的 X$表

X$KJBL 描述哪一个实例是某个表的带 BL 锁的块的 master 实例（见表 5）。

让表 3 中所描述的两个实例继续 update，不要停接着做以下的实验（如果两个实例，或其中一个结束了 update 重新 update）

实验第 18 步：

在任一个实例上执行:

创建一个基于 X$KJBL 的视图，后续实验需要它

```
 create or replace view myview as

      select  kj.block#, kjblname, kjblname2, kjblowner+1 "OWNER_Instance", kjblmaster+1
"MASTER_Instance", kjbllockp  from

 (

   select  kjblname, kjblname2, kjblowner, kjblmaster, kjbllockp,

    (substr ( kjblname2,  instr(kjblname2,',')+1,   instr(kjblname2,',',1,2)-instr(kjblname2,',',1,1)-1))/65536
file#,

     substr ( kjblname2, 1, instr(kjblname2,',')-1) block# from x$kjbl

  ) kj,

 (

    select   block_id   block#_begin,  block_id+blocks-1  block#_end, e.file_id file#

    from dba_extents e where e.owner='HR' and e.segment_name='T04209_UNAME'

  ) e

where kj.block# between e.block#_begin and e.block#_end   order by block# ;
```

实验第 19 步：

在任一个实例上执行:

```
select * from myview;
```

	FILE#	BLOCK#	KJBLNAME	KJBLNAME2	OWNER_Instance	MASTER_Instance	KJBLLOCKP
1	4	387	[0x183][0x40000],[BL]	387,262144,BL	1	1	237EC730
2	4	387	[0x183][0x40000],[BL]	387,262144,BL	2	1	49A35F58

3	4	388	[0x184][0x40000],[BL]	388,262144,BL	2	1	49A31B60
4	4	388	[0x184][0x40000],[BL]	388,262144,BL	1	1	237F38A0
5	4	389	[0x185][0x40000],[BL]	389,262144,BL	1	1	237FA490
6	4	389	[0x185][0x40000],[BL]	389,262144,BL	2	1	49A321D0
7	4	390	[0x186][0x40000],[BL]	390,262144,BL	2	1	49A32368
8	4	390	[0x186][0x40000],[BL]	390,262144,BL	1	1	23BF2560
9	4	391	[0x187][0x40000],[BL]	391,262144,BL	2	1	49A31BE8
10	4	391	[0x187][0x40000],[BL]	391,262144,BL	1	1	237F16F0
……	……	……	……	……	……	……	……

从以上的查询结果可以看出：每个数据块内存拷贝在两个实例的数据库缓冲区缓存中都可以被找到，也就是说每个数据块都有两个个 Owner 实例，那么证明在近期有两个实例先后修改或访问过它。

实验第 20 步：

在任一个实例上执行:

```
select  distinct  block# ,  "MASTER_Instance"    from myview  order by 1 ;
```

	BLOCK#	MASTER_Instance
1	387	1
2	388	1
3	389	1
4	390	1
5	391	1
6	392	1
7	393	1
8	394	1
9	395	1
10	396	1
……	……	……
109	502	1
110	503	1
111	504	1
112	505	1
113	506	1
114	507	1
115	508	1
116	509	1
117	510	1
118	511	1
119	512	2←从这变成实例 2
120	523	2
121	524	2
122	525	2
123	526	2
124	527	2
125	528	2
126	529	2
127	530	2
128	531	2
129	532	2
130	533	2
……	……	……
220	628	2
221	629	2
222	630	2
223	631	2
224	632	2
225	633	2
226	635	2
227	636	2
228	637	2
229	638	2
230	639	2

231	640	1
232	641	1
233	642	1
234	643	1
235	644	1
236	645	1
237	646	1
238	647	1
239	648	1

这说明：从 10g 以后版本开始，数据块的状态和属主等信息被存储成每 128 个块的信息一个 master 单元，即 128 个数据块的状态和属主等信息构成一个“gcs mastership bucket”。但是要说明以下：一个“gcs mastership bucket”不一定要存满 128 个块的状态和属主等信息。这样就能理解：超过 128 个块的表的数据块可以被多个实例分布式地分段 master。

实验第 21 步：

在任一个实例上执行:

```
select  count(*)  from myview  where "OWNER_Instance" <> "MASTER_Instance" ;
```

	COUNT(*)
1	239

从以上结果可以看到，存在大量的数据块的 master 实例和 owner 实例不是同一个实例的情况。这是可以预料到的：我们在两个实例上同时密集地 OLTP 同一张表！这样会有大量的实例间的通信。以下内容来自此刻的 AWR 报告：

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
gc cr block busy	85,181	181	2	44.5	Cluster ←对方 CR 块请求收到，但是无法立即发送
CPU time		79		19.4	
gcs log flush sync	37,349	35	1	8.6	Other ←日志系统太满 flush 不过来
gc cr multi block request	1,623	27	16	6.5	Cluster ← placeholder 请求多
gc current grant busy	13,162	20	2	5.0	Cluster ← 由于 master 实例和 owner 实例不是同一个实例导致 master 实例发送磁盘授权信息延迟或接收延迟。

RAC Statistics

	Begin	End
Number of Instances:	2	2

Global Cache Load Profile

	Per Second	Per Transaction
Global Cache blocks received:	264.26	1,108.00
Global Cache blocks served:	356.69	1,495.53
GCS/GES messages received:	452.07	1,895.44
GCS/GES messages sent:	462.24	1,938.10
DBWR Fusion writes:	0.16	0.67
Estd Interconnect traffic (KB)	5,146.16	

Global Cache Efficiency Percentages (Target local+remote 100%)

Buffer access - local cache %: 99.69
Buffer access - remote cache %: 0.30
Buffer access - disk %: 0.00

Global Cache and Enqueue Services - Workload Characteristics

Avg global enqueue get time (ms): 0.3
Avg global cache cr block receive time (ms): 2.0
Avg global cache current block receive time (ms): 1.4
Avg global cache cr block build time (ms): 0.1
Avg global cache cr block send time (ms): 0.0
Global cache log flushes for cr blocks served %: 23.9
Avg global cache cr block flush time (ms): 1.5
Avg global cache current block pin time (ms): 0.0
Avg global cache current block send time (ms): 0.0
Global cache log flushes for current blocks served %: 0.2
Avg global cache current block flush time (ms): 1.9

Global Cache and Enqueue Services - Messaging Statistics

Avg message sent queue time (ms): 0.2
Avg message sent queue time on ksxp (ms): 0.8
Avg message received queue time (ms): 0.0
Avg GCS message process time (ms): 0.1
Avg GES message process time (ms): 0.0
% of direct sent messages: 73.86
% of indirect sent messages: 13.11
% of flow controlled messages: 13.03

实验第 22 步：
现在让问题回到简单的状态，我们把第 2 个实例上的 update 结束。
在任一个实例上执行:

```
select  count(*)  from myview  where "OWNER_Instance" <> "MASTER_Instance"  and
"OWNER_Instance"   =1;

        COUNT(*)
1       118

 select  count(*)  from myview  where "OWNER_Instance" <> "MASTER_Instance"  and
"OWNER_Instance"   =2;

        COUNT(*)
1       110
```

实验第 23 步：
过一会在任一个实例上执行，再查:

```
select  count(*)  from myview  where "OWNER_Instance" <> "MASTER_Instance"  and
"OWNER_Instance"   =1;

        COUNT(*)
1       118

select  count(*)  from myview  where "OWNER_Instance" <> "MASTER_Instance"  and
"OWNER_Instance"   =2;
```

	COUNT(*)
1	20

从以上结果可以看到，当我们把第 2 个实例上的 update 结束后，第 2 个实例的 owner 计数在不断下降，但是仍然存在不少数据块的 master 实例和 owner 实例不是同一个实例的情况，这样在我们的密集 OLTP 应用模型下仍然存在实例间不必要的通信量。

实验第 24 步：
在任一个实例上执行:
select * from v$gcspfmaster_info where object_id=52533;
没有输出

说明此刻没有发生自动 Remaster（Object Affinity and Dynamic Remastering 引起）。

实验第 25 步：
在任一个实例上执行（可执行多次）：
select drms from X$KJDRMAFNSTATS;值始终为 2 也验证了此刻没有发生自动 Remaster。

表 5：能够查出某个块 Master 实例和 Owner 实例的 X$表

5．自动 Remaster（Object Affinity and Dynamic Remastering 引起）和手工 Remaster（oradebug 命令）

实验第 26 步：
在任一个实例上执行:
select x.ksppinm name,y.ksppstvl value
from sys.x$ksppi x,sys.x$ksppcv y
where x.indx=y.indx and substr(x.ksppinm,0,1)='_' and x.ksppinm like '_gc_policy%';

	NAME	VALUE
1	_gc_policy_time	10←每 10 秒查看一下是否要做 Remastering（不是分钟）
2	_gc_policy_minimum	1500←每分钟最少要有的 dynamic affinity 活动个数，才会发 Remastering

下面我们来做自动 remaster（Object Affinity and Dynamic Remastering 引起）的实验。
实验第 27 步：
在任一个实例上执行:
select count(*) from myview where "OWNER_Instance" <> "MASTER_Instance" ;

	COUNT(*)
1	138

在任一个实例上执行:
select * from x$OBJECT_POLICY_STATISTICS where object=52533;
无输出，这是正常的。 x$OBJECT_POLICY_STATISTICS 统计信息会定时清空。

实验第 28 步：
在实例 1 上：连续做以下两条语句，直到 t04209_uname 表中有 51200000 行记录为止。

```
insert /*+ append */ into t04209_uname select * from t04209_uname;
commit;
```

实验第 29 步：
然后，再在实例 1 上执行：

```
select  count(*) from t04209_uname;
select  sum(uvalue) from t04209_uname;
select  avg(uvalue) from t04209_uname;
select  max(uvalue) from t04209_uname;
select  min(uvalue) from t04209_uname;
```

多做一些组函数查询，以进一步增大 BL 的 open 计数。

在任一个实例上执行：

```
select  * from x$OBJECT_POLICY_STATISTICS where object=52533;
```

	ADDR	INDX	INST_ID	OBJECT	NODE	OPENS
1	B7EAB520	1	2↖INST_ID 为 2 代表 Mater 实例为实例 2,还没发生 Remaster 事件	52533	1←实例 1 有大量的 BL Open 计数	42317

稍等片刻.....
在任一个实例上执行:

```
select  * from v$gcspfmaster_info where object_id=52533;
```

	FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
1	0	52533	0↖代表 master 实例是 1	32767↖代表之前没发生过 remaster	0

```
select drms from X$KJDRMAFNSTATS;
```

	DRMS
1	3

DRM 为 2+1=3,验证了此刻发生了自动 Remaster（第 1 次 Remaster）。
再验证：
进入实例 1 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：

```
./rdba1_lmd0_7002.trc:  Transfer pkey 52533 to node 0
./rdba1_lmd0_7002.trc:Begin DRM(5) - transfer pkey 52533 to 0 oscan 0.0
```

进入实例 2 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：

```
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc:   pkey 52533
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc:   pkey 52533
./rdba2_lmd0_7032.trc:Rcvd DRM(5) Transfer pkey 52533 to 0 oscan 1.1
```

以上日志都证明发生了对象 52533 的所有块的 remaster，对象 52533 的所有块的 master 实例现在都是实例 1。也就是说：如果发生自动 Remaster（Object Affinity and Dynamic Remastering 引起）或手工 Remaster（oradebug 命令），整个对象将作为 master 单元而不进行多个实例分布式地分段 master：即不管表多大，它的数据块都由同一个实例 master。
在任一个实例上执行:

```
select  *  from myview  where  "MASTER_Instance"=2 ;
```

没有输出，这就对了，因为都被实例 1master 了。

下面我们继续实验：
实验第 30 步：
回到实例 2 依次执行:
select count(*) from t04209_uname;
select sum(uvalue) from t04209_uname;
select avg(uvalue) from t04209_uname;
select max(uvalue) from t04209_uname;
select min(uvalue) from t04209_uname;
还可以多做一些组函数查询，以进一步增大 BL 的 open 计数。

实验第 31 步：
在任一个实例上执行:
select * from v$gcspfmaster_info where object_id=52533;

	FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
1	0	52533	1↖代表 master 实例是 2	0↖代表上一任 master 是实例 1	0

select drms from X$KJDRMAFNSTATS;

	DRMS
1	4

DRM 为 3+1=4,验证了此刻又发生了一次自动 Remaster（第 2 次 Remaster）。
再验证：
进入实例 1 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：
./rdba1_lmd0_7002.trc: Transfer pkey 52533 to node 0
./rdba1_lmd0_7002.trc:Begin DRM(5) - transfer pkey 52533 to 0 oscan 0.0
./rdba1_lmd0_7002.trc:Rcvd DRM(6) Transfer pkey 52533 from 0 to 1 oscan 0.0
进入实例 2 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc: pkey 52533
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc: pkey 52533
./rdba2_lmd0_7032.trc:Rcvd DRM(5) Transfer pkey 52533 to 0 oscan 1.1
./rdba2_lmd0_7032.trc: Transfer pkey 52533 to node 1
./rdba2_lmd0_7032.trc:Begin DRM(6) - transfer pkey 52533 to 1 oscan 0.0

以上日志都证明发生了对象 52533 的所有块的 remaster，对象 52533 的所有块的 master 实例现在都是实例 2。也就是说：如果发生自动 Remaster（Object Affinity and Dynamic Remastering 引起）或手工 Remaster（oradebug 命令），整个对象将作为 master 单元而不进行多个实例分布式地分段 master：即不管表多大，它的数据块都由同一个实例 master。
select * from myview where "MASTER_Instance"=1 ;
没有输出。这就对了，因为都被实例 2master 了。

下面接着研究手工 remaster：
实验第 32 步：
在实例 1 上:
[oracle@node1 ~]$ sqlplus /nolog
SQL*Plus: Release 10.2.0.1.0 - Production on Mon Jan 13 16:18:27 2014
Copyright (c) 1982, 2005, Oracle. All rights reserved.
SQL> conn / as sysdba
Connected.
SQL> oradebug setmypid;
Statement processed.
SQL> oradebug lkdebug -a hashcount
Statement processed.
SQL> oradebug lkdebug -k
Statement processed.
SQL> oradebug lkdebug -m pkey 52533
Statement processed.
SQL> oradebug lkdebug -k
Statement processed.
SQL> oradebug tracefile_name;
/u01/app/oracle/admin/RDBA/udump/rdba1_ora_18378.trc
以上操作的目的是手工强迫使实例 1 重新夺回对象 52533 所有的块的 mastership。

实验第 33 步：
下面来验证：
节选实例 1 上的/u01/app/oracle/admin/RDBA/udump/rdba1_ora_18378.trc：
/u01/app/oracle/admin/RDBA/udump/rdba1_ora_18378.trc
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Real Application Clusters, OLAP and Data Mining options
ORACLE_HOME = /u01/app/oracle/product/10.2.0/db_1
System name: Linux
Node name: node1.example.com
Release: 2.6.18-164.el5xen
Version: #1 SMP Tue Aug 18 16:06:30 EDT 2009
Machine: i686
Instance name: RDBA1
Redo thread mounted by this instance: 1
Oracle process number: 25
Unix process pid: 18378, image: oracle@node1.example.com (TNS V1-V3)

*** 2014-01-13 16:22:07.764
*** SERVICE NAME:(SYS$USERS) 2014-01-13 16:22:07.763
*** SESSION ID:(125.1496) 2014-01-13 16:22:07.763
****
GLOBAL ENQUEUE SERVICE DEBUG INFORMATION

Resource hash bucket   count
0   4
1   1
2   4
3   11
4   6
5   6
6   1
7   3
8   4

```
……
2039              2
2040              2
2041              6
2042              5
2043              2
2044              9
2045              2
2046              2
2047              3
Total resource count in hash buckets: 8213
***************** End of lkdebug output *************************
*** 2014-01-13 16:26:26.465
*******************************************************************
GLOBAL ENQUEUE SERVICE DEBUG INFORMATION
----------------------------------------
node# 0, #nodes 2, state 4, msgver 4, rcvver 0 validver 4
valid_domain 1
sync acks 0x00000000000000000000000000000000
Resource freelist #0 len 28410 lwm 2893 add 241108 rem 212698
Resource freelist #1 len 28471 lwm 3306 add 241942 rem 213471
LMS0:
Hash buckets log2(11)
Bucket# 0 #res 0
Bucket# 1 #res 0
Bucket# 2 #res 0
Bucket# 3 #res 0
Bucket# 4 #res 0
Bucket# 5 #res 0
Bucket# 6 #res 0
Bucket# 7 #res 0
……
atch buckets log2(6)
GCS shadow freelist #0 len 29067 lwm 7451 add 88332 rem 59265
GCS shadow freelist #1 len 29097 lwm 7257 add 88862 rem 59765
 files in affinity vector:

* >> PT table contents ---:
 pt table bucket = 1
pkey 4294950913, stat 0, masters[32767, 0->0], reminc 2, RM# 1 flg 0x0
 pt table bucket = 2
pkey 4294950914, stat 0, masters[32767, 0->0], reminc 2, RM# 1 flg 0x0
 pt table bucket = 3
pkey 4294950915, stat 0, masters[32767, 0->0], reminc 2, RM# 1 flg 0x0
pkey 52533, stat 0, masters[0, 1->1], reminc 4, RM# 6 flg 0x0      ←手工 remaster 之前， oradebug lkdebug –k 的输出，代表 master 实例是 2（[0, 1->1]中的 1->1 表示实例 2），上一任 master 是实例 1（[0, 1->1]中的 0 表示上一任 master 实例 1）
* kjilpkey = 0
***************** End of lkdebug output *************************
*** 2014-01-13 16:27:14.981
*******************************************************************
GLOBAL ENQUEUE SERVICE DEBUG INFORMATION
----------------------------------------
***************** End of lkdebug output *************************
Latch buckets log2(6)
GCS shadow freelist #0 len 7487 lwm 7451 add 88336 rem 80849
GCS shadow freelist #1 len 7569 lwm 7257 add 88863 rem 81294
 files in affinity vector:
```

```
* >> PT table contents ---:
pkey 4294950932, stat 0, masters[32767, 1->1], reminc 4, RM# 4 flg 0x0
 pt table bucket = 3381
pkey 52533, stat 0, masters[1, 0->0], reminc 4, RM# 7 flg 0x0    ←手工 remaster 之后， oradebug lkdebug –k 的输出，代表 master 实例是 1（[1, 0->0]中的 0->0 表示实例 1），上一任 master 是实例 1（[1, 0->0]中的 1 表示上一任 master 实例 2）
* kjilpkey = 1
****************** End of lkdebug output *************************
```

trace 文件已经说明：实例 1 重新夺回了对象 52533 所有的块的 mastership。
select * from myview where "MASTER_Instance"=2 ;
无输出。这就对了，因为都被实例 1master 了。

select * from v$gcspfmaster_info where object_id=52533;

	FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
1	0	52533	0↖代表 master 实例是 1	1↖代表上一任 master 是实例 2	0

select drms from X$KJDRMAFNSTATS;

	DRMS
1	5

DRM 为 4+1=5，验证了此刻又发生了一次 Remaster（第 3 次 Remaster）。
再验证：
进入实例 1 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：
./rdba1_lmd0_7002.trc: Transfer pkey 52533 to node 0
./rdba1_lmd0_7002.trc:Begin DRM(5) - transfer pkey 52533 to 0 oscan 0.0
./rdba1_lmd0_7002.trc:Rcvd DRM(6) Transfer pkey 52533 from 0 to 1 oscan 0.0
./rdba1_lmd0_7002.trc: Transfer pkey 52533 to node 0
./rdba1_lmd0_7002.trc:Begin DRM(7) - transfer pkey 52533 to 0 oscan 0.0
进入实例 2 的/u01/app/oracle/admin/RDBA/bdump:
执行：grep -r "pkey 52533" ./
输出：
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc: pkey 52533
./rdba2_lms0_7034.trc: GCS CLIENT 0x233f71b0,2 sq[(nil),(nil)] resp[(nil),0x185.40000] pkey 52533
./rdba2_lms0_7034.trc: pkey 52533
./rdba2_lmd0_7032.trc:Rcvd DRM(5) Transfer pkey 52533 to 0 oscan 1.1
./rdba2_lmd0_7032.trc: Transfer pkey 52533 to node 1
./rdba2_lmd0_7032.trc:Begin DRM(6) - transfer pkey 52533 to 1 oscan 0.0
./rdba2_lmd0_7032.trc:Rcvd DRM(7) Transfer pkey 52533 from 1 to 0 oscan 0.0

表 6：自动 remaster（Object Affinity and Dynamic Remastering 引起）和手工改变 Master 实例的 oradebug 命令

6．实例恢复过程中的 Remaster（Dynamic Reconfiguration）

图 5 描述了实例 1 发生故障，实例 2 恢复的情况：

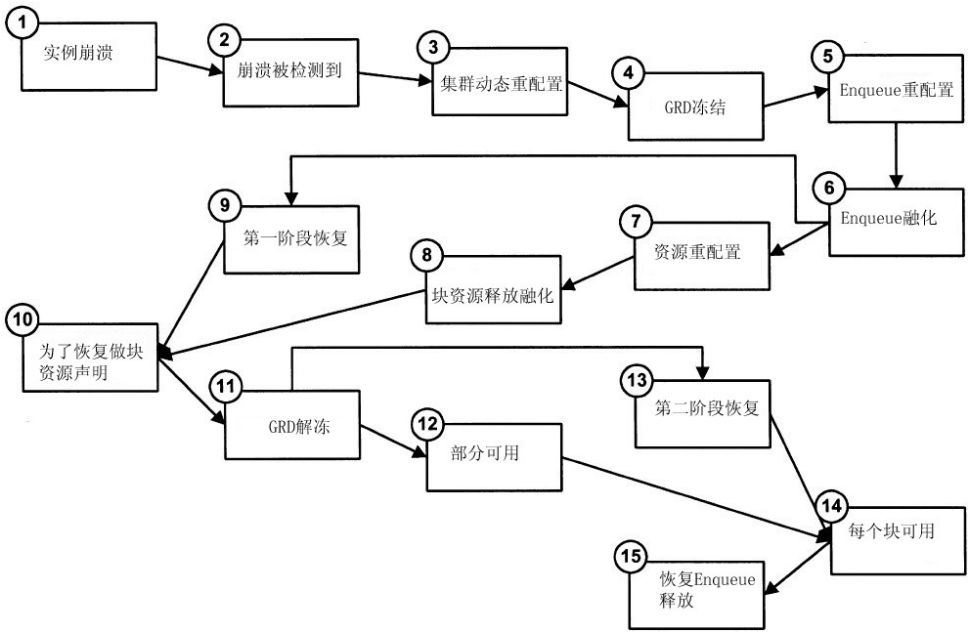

图 5：RAC 实例恢复过程

实验第 34 步：
在实例 1 上杀死 ora_smon 进程：

```
[oracle@node1 bdump]$ ps aux | grep ora_smon
oracle    7057  0.0  3.5 808624 96816 ?        Ss   10:51   0:01 ora_smon_RDBA1
oracle   16557  0.0  0.0   3932   716 pts/0    S+   17:17   0:00 grep ora_smon

[oracle@node1 bdump]$ kill -9 7057
```

之后不久实例 1 得到恢复，重新启动：

```
select  count(*)  from myview  where  "MASTER_Instance"=1;

        COUNT(*)
1       21011

select  count(*)  from myview  where  "MASTER_Instance"=2;

        COUNT(*)
1       20981
```

显然 Oracle 采用了“ lazy remastering”算法，实例 1 在恢复后仅仅重新 remaster 了原来的一部分资源。原因是实例 2 在为实例 1 做交叉恢复时所 Master 到的资源就由实例 2Master 保管了。

此时 select * from v$gcspfmaster_info where object_id=52533;
无输出。

```
select drms from X$KJDRMAFNSTATS;

         DRMS
1        8
```

DRM 为 5+3=5，验证了此刻又发生了三次 lazy remaster。

实验第 34 步：

```
进一步：把实例 1 所在的操作系统关闭。
稍等片刻……
select  count(*)  from myview  where  "MASTER_Instance"=1;

        COUNT(*)
1       11009

稍等片刻……
select  count(*)  from myview  where  "MASTER_Instance"=1;

        COUNT(*)
1       0

select  count(*)  from myview  where  "MASTER_Instance"=2;

        COUNT(*)
1       41789

全部由实例 2 接管了。
```

表 7：实例恢复过程中的 remaster（**Dynamic Reconfiguration**）

总结

在 RAC 的体系结构中,全局资源目录（Global Resource Directory 简称 GRD）是 Oracle RAC 数据库中最重要的内存结构。对于一个数据块而言，管理该数据块的状态和属主信息以及数据块内部和数据块自身的锁信息的实例只有一个。这个实例就被称作为该数据块（或更准确地说：资源）的 Master 实例。从 10g 以后版本开始，数据块的状态和属主等信息被存储成每 128 个块的信息一个 master 单元，即 128 个数据块的状态和属主等信息构成一个“gcs mastership bucket”。但是要说明以下：一个“gcs mastership bucket”不一定要存满 128 个块的状态和属主等信息。这样就能理解：超过 128 个块的表的数据块可以被多个实例分布式地分段 master。如果发生自动 Remaster（Object Affinity and Dynamic Remastering 引起）或手工 Remaster（oradebug 命令），整个对象将作为 master 单元而不进行多个实例分布式地分段 master：即不管表多大，它的数据块都由同一个实例 master。另外，任何时候 undo 段整段必需由同一个实例 master。