第 7 章　重做（redo）

重做（redo）和撤消（Undo）是 Oracle 的重要特性，用以保证事务的可恢复性和可回退性。本章对 Oracle 的重做机制进行说明。

7.1Redo 的作用

Oracle 通过 redo 来保证数据库的事务可以被重演，从而使得在故障之后，数据可以被恢复。Redo 对于 Oracle 数据库来说至关重要。

在数据库中，Redo 的功能主要通过三个组件来实现：Redo Log Buffer 、LGWR 后台进程和 Redo Log File(在归档模式下，Redo Log File 最终会经由 ARCn 进程写出为归档日志文件)。

图 7-1 是 Oracle 的数据库实例示意图，Redo Log Buffer 以及 LGWR 进程在图中皆有体现：

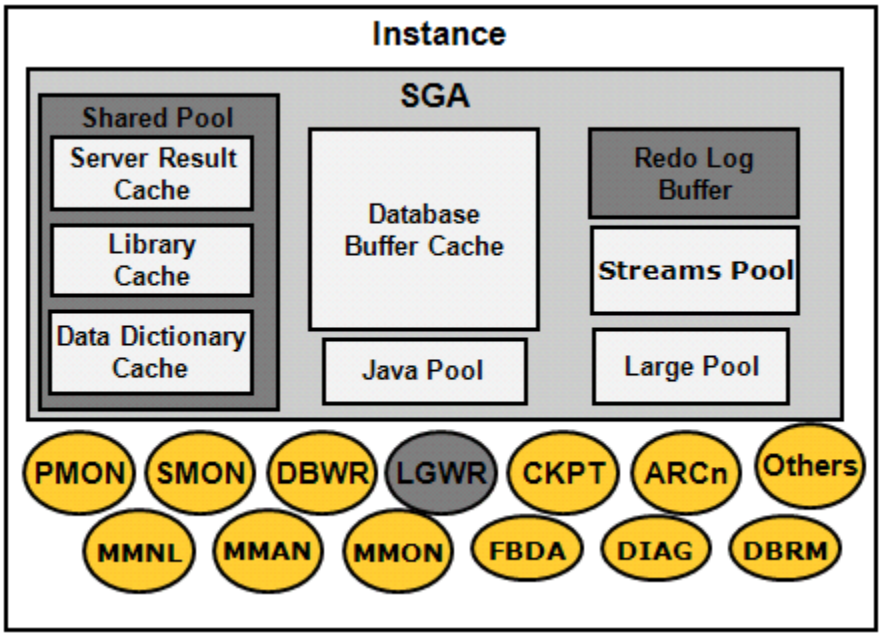


图 7-1

Redo Log Buffer 位于 SGA 之中，是一块循环使用的内存区域。其中保存数据库变更的相关信息。这些信息以重做条目（Redo Entries）形式存储（Redo Entries 也经常被称为 Redo records ）。Redo Entries 包含重构、重做数据库变更的重要信息，这些变更包括 INSERT,UPDATE,DELETE,CREATE,ALTER 或者 DROP 等。在必要的时候 Redo Entries 被用于数据库恢复。

Redo Entries 的内容被 Oracle 数据库进程从用户的内存空间(PGA)复制到 SGA 中的 Redo

Log Buffer 之中。Redo Entries 在内存中占用连续的顺序空间，由于 Redo Log Buffer 是循环使用的，Oracle 通过一个后台进程 LGWR 不断的把 Redo Log Buffer 的内容写出到 Redo Log File 中，Redo Log File 同样是循环使用的。图 7-2 说明了 Redo Log Buffer、LGWR 以及 Redo Log File 三者之间的关系。

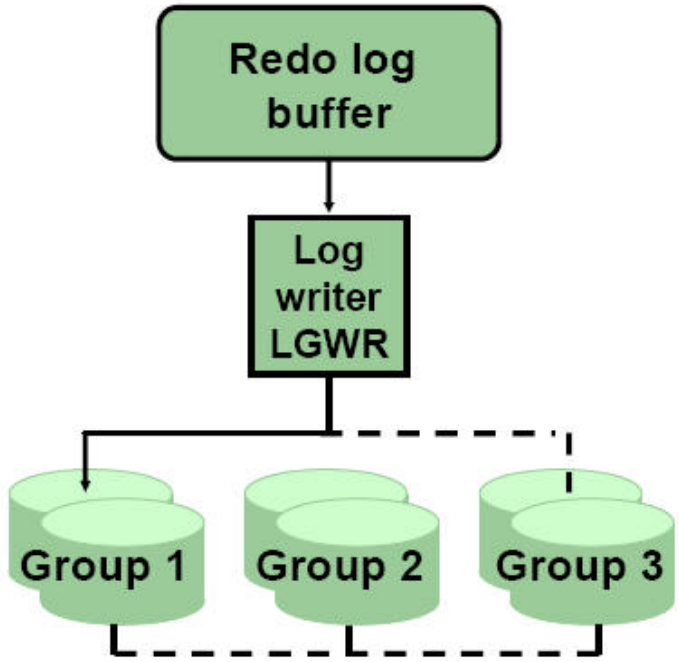


图 7-2

7.2Redo 的原理

通过前面的章节，我们已经知道，用户数据通常在 Buffer Cache 中修改，Oracle 通过高速缓存来提高数据操作的性能。当用户在 Buffer Cache 中修改数据时，Oracle 并不会立即将变更数据写出到数据文件上，因为独立的离散写出效率会很低。到目前为止，计算机系统中最容易出现瓶颈的仍然是磁盘的 I/O 操作，Oracle 这样做的目的是为了减少 IO 的压力，当修改过的数据达到一定数量之后，可以进行高效地批量写出。

大部分传统数据库(当然包括 Oracle)在处理数据修改时都遵循 **no-force-at-commit** 策略。也就是说，在提交时并不强制写。那么为了保证数据在数据库发生故障时(例如断电)可以恢复，Oracle 引入了 Redo 机制，通过连续的、顺序的日志条目的写出将随机的、分散的数据块的写出推延。这个推延使得数据的写出可以获得批量效应等性能提升。

同 Redo Log Buffer 类似，Redo Log File 也是循环使用的，Oracle 允许使用最少两个日志组。缺省的，数据库创建时会建立 3 个日志组。

```
SQL> select group#,members,status from v$log;
    GROUP#    MEMBERS STATUS
```

```
---------- ---------- ----------------
         1          1 INACTIVE
         2          1 CURRENT
         3          1 INACTIVE
```

当一个日志文件写满之后，会切换到另外一个日志文件，这个切换过程称为 Log Switch。Log Switch 会触发一个检查点，促使 DBWR 进程将写满的日志文件保护的变更数据写回到数据库。在检查点完成之前，日志文件是不能够被重用的。

由于 Redo 机制对于数据的保护，当数据库发生故障时，Oracle 就可以通过 Redo 重演进行数据恢复。那么一个非常重要的问题是，恢复应该从何处开始呢？

如果读取的 redo 过多，那么必然导致恢复的时间过长，在生产环境中，我们必需保证恢复时间尽量的短。

Oracle 通过检查点（Checkpoint）来缩减恢复时间。

回顾一下我们在第一章中所提到的：**检查点只是一个数据库事件，它存在的根本意义在于减少恢复时间**。

当检查点发生时（此时的 SCN 被称为 Checkpoint SCN）Oracle 会通知 DBWR 进程，把修改过的数据，也就是此 Checkpoint SCN 之前的脏数据（Dirty Buffer）从 Buffer Cache 写入磁盘，在检查点完成后 CKPT 进程会相应的更新控制文件和数据文件头，记录检查点信息，标识变更。

在检查点完成之后，此检查点之前修改过的数据都已经写回磁盘，重做日志文件中的相应重做记录对于崩溃/实例恢复不再有用。

如果此后数据库崩溃，那么恢复只需要从最后一次完成的检查点开始恢复即可。

如果数据库运行在归档模式（所有生产数据库，都建议运行在归档模式），日志文件在重用之前必须写出到归档日志文件，归档日志在介质恢复时可以用来恢复数据库故障。

7.3Redo 与 Latch

在一个多用户的并发系统中，大量用户进程都需要向 Redo Log Buffer 写入重做数据，Oracle 通过 Latch 来保护和协调 Redo Log Buffer 的工作。同 Redo 相关的 Latch 主要有 Redo Copy Latch、Redo Allocation Latch 等，Redo Allocation Latch 用于管理 Log Buffer 内存空间的分配，Redo Copy Latch 则用于写 Redo 内容到 Redo Log Buffer 过程的保护。

一个进程在修改数据时产生 Redo，Redo 首先在 PGA 中保存，当进程需要将 Redo 信息 Copy 进入 Redo Log Buffer 时需要获得 redo copy latch，获得了该 latch 以后才能把 redo 拷贝到 Log Buffer 中。Redo copy latch 表明进程正在把 redo 拷贝入 log buffer 中，在此过程中，LGWR 应该等待直到进程拷贝完成才能把目标 Log buffer Block 写入磁盘。

初始化参数_LOG_SIMULTANEOUS_COPIES,定义允许同时写 redo 的 redo copy latch 的数量。在 Oracle7 和 Oracle8 里，_LOG_SIMULTANEOUS_COPIES 缺省的等于 CPU 的数量。

从 Oracle8.1.3 开始，缺省的_LOG_SIMULTANEOUS_COPIES 变成 2 倍的 CPU 数量，并且成为了一个隐含参数：

```
SQL> @GetHidPar
Enter value for par: copies
NAME                    VALUE  DESCRIB
----------------------- ----- ------------------------------------------------------------
_log_simultaneous_copies  8  number of simultaneous copies into redo buffer(# of copy latches)
```

从 v$latch 视图中我们可以得到关于 redo copy latch 的汇总信息：

```
SQL> select name,GETS,IMMEDIATE_GETS,IMMEDIATE_MISSES,SPIN_GETS
  2  from v$latch where name='redo copy';
NAME              GETS IMMEDIATE_GETS IMMEDIATE_MISSES  SPIN_GETS
--------------- ---------- ------------- ---------------- ----------
redo copy              112      247810971             7184          0
```

对于 redo copy latch 的多个子 Latch，可以从 v$latch_children 视图获得更为详细的信息：

```
SQL> select addr,latch#,child#,name,gets,immediate_gets,immediate_misses
  2  from v$latch_children where name = 'redo copy';
ADDR         LATCH#     CHILD# NAME                 GETS IMMEDIATE_GETS IMMEDIATE_MISSES
-------- ---------- ---------- --------------- ---------- ------------- ----------------
57187FB4        147          1 redo copy               14     118062595             4199
57188030        147          2 redo copy               14      40781669              912
571880AC        147          3 redo copy               14      68951827             1807
57188128        147          4 redo copy               14       1557929              133
571881A4        147          5 redo copy               14      13341465              105
57188220        147          6 redo copy               14       5115486               28
5718829C        147          7 redo copy               14             0                0
57188318        147          8 redo copy               14             0                0
8 rows selected.
```

Redo copy latch 获取以后，进程紧接着需要获取 redo allocation latch ，分配 redo 空间，空间分配完成以后，redo allocation latch 即被释放，进程把 PGA 里临时存放的 redo 信息 COPY 入 redo log buffer，COPY 完成以后,redo copy latch 释放。

在完成 redo copy 以后，进程可能需要通知 LGWR 去执行写出（如果 redo copy 是 commit 等因素触发的）。为了避免 LGWR 被不必要的通知，进程需要先获取 redo writing latch 去检查 LGWR 是否已经激活或者已经被通知。如果 LGWR 已经激活或被 Post，redo writing latch 将被释放。

```
SQL> col name for a20
SQL> select addr,latch#,name,gets,misses,immediate_gets,immediate_misses
  2  from v$latch where name='redo writing';
ADDR         LATCH# NAME     GETS     MISSES IMMEDIATE_GETS IMMEDIATE_MISSES
--------     ---------- -------------------- ---------- ---------- ------------- -----------
```

```
0217ECC8       113 redo writing        2265           0              0                0
```

如果 redo writing latch 竞争过多,可能意味着你的提交过于频繁。通过系统统计信息或 Statspack 可以获得这些信息，具体参考 "log file sync" 等待事件一节。

在执行 redo copy 的过程中，进程以 log file sync 事件处于等待。当进程从 log file sync 中等待中醒来以后，进程需要重新获得 redo allocation latch 检查是否相应的 redo 已经被写入 redo log file，如果尚未写入，进程必须继续等待。这是一个大致简化的 Latch 处理过程，用以说明 Latch 的处理机制。

和 Redo 相关的另外一个常见 Latch 是 redo allocation latch，当进程需要向 Redo Log Buffer 写入 Redo 信息时需要获得此 Latch，分配 Redo Log Buffer 空间。所以，如果对于一个繁忙的数据库系统，该 Latch 通常也是竞争激烈的 Latch 之一。在以上取样的数据库中，Redo allocation latch 和 Redo Copy Latch 同属 Top 5 请求的 Latch 之一：

```
NAME                             IMMEDIATE_GETS IMMEDIATE_MISSES  SPIN_GETS
-------------------------------- -------------- ---------------- ----------
cache buffers lru chain                259891274           209819     213249
cache buffers chains                   258525736             1470      18065
redo copy                              247810939             7184          0
redo allocation                        247808297             9909        926
checkpoint queue latch                  56443129             4945       3825
```

7.4Oracle9i Redo 的增强

在 Oracle9iR2 中，Oracle 通过 LOG_PARALLELISM 定义 Oracle 中 redo allocation 的并发级别。如果定义 LOG_PARALLELISM 大于 1，那么数据库将分配多个共享的 Redo Log Buffer 区域，每个共享 Buffer 都有独立的 Redo Allocation Latch 进行保护，从而提高了 Redo 的并发性能，多个 Redo Log Buffer 可以被看作是 Redo Log Buffer 的子池，与 Shared Pool 的多 Subpool 原理类似。多 Redo Log Buffer 机制又被称为 Public Redolog Strands。图 7-3 是新机制下 Redo Log Buffer 原理示意图：

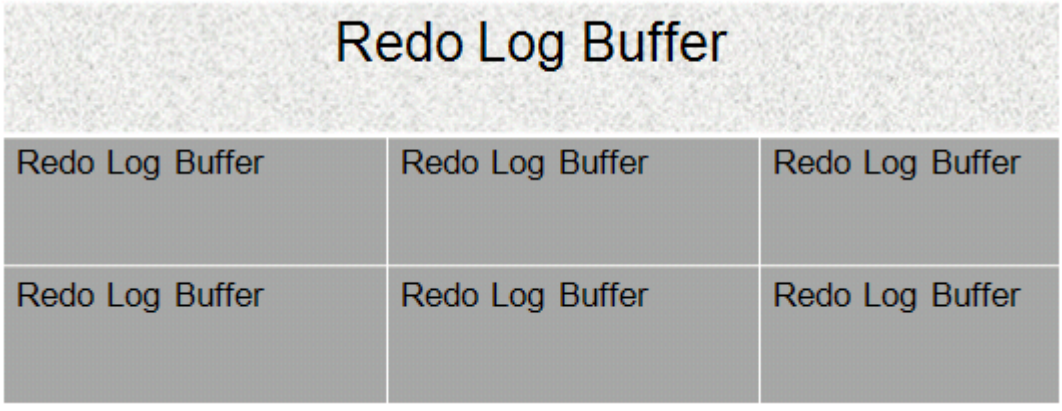


图 7-3

如果你使用的是高端服务器，有超过 16 个处理器，并且经历非常高的 redo allocation latch 竞争，那么可以考虑启用并行 redo。允许并行 redo 生成能够增加更新密集型数据库的吞吐量，可以通过考察 V$LATCH 视图观察 redo allocation latch 竞争的累计等待时间。

通过如下查询可以获得相关 Latch 信息：

```
SELECT  substr(ln.name, 1, 20), gets, misses, immediate_gets, immediate_misses
FROM v$latch l, v$latchname ln
WHERE   ln.name in ('redo allocation', 'redo copy')  and ln.latch# = l.latch#;
```

如果 MISSES 对 GETS 比率超过 1%，或者 IMMEDIATE_MISSES 对 (IMMEDIATE_GETS + IMMEDIATE_MISSES) 比率超过 1%，那么通常认为存在 Latch 竞争。

当主机拥有 16~64 个 CPU 时，Oracle 公司推荐设置 LOG_PARALLELISM 在 2~8 之间。你可以从低值(例如 2)开始，以 1 为步长增进直到 redo allocation latch 竞争不再激烈，这个参数的设置可以提高应用的性能。大于 8 的 LOG_PARALLELISM 设置通常不被推荐。

在 Oracle9iR2 中，该参数的缺省值为 1：

```
SQL> show parameter log_p
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
log_parallelism                      integer     1
```

缺省的 Redo Allocation Latch 也仅有一个：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
SQL> select addr,latch#,child#,name,gets,immediate_gets,immediate_misses
  2  from v$latch_children where name = 'redo allocation';
ADDR         LATCH#     CHILD# NAME                  GETS IMMEDIATE_GETS IMMEDIATE_MISSES
-------- ---------- ---------- --------------- ---------- -------------- ----------------
5343A908        115          1 redo allocation     183268              0                0
```

在 Oracle10gR2 中，这一切又有所改变。

7.5Oracle10g Redo 的增强

在 Oracle0g 中，log_parallelism 参数变为隐含参数，并且 Oracle 引入了另外两个参数，允许 log_parallelism 进行动态调整。缺省的_log_parallelism_dynamic 参数被设置为 True，如果_log_parallelism_max 被设置为不同于_log_parallelism 的参数值，那么 Oracle 会动态的选择并行度，当然不超过最大允许值，这是 Oracle10g 中动态 SGA 的另外一个提高。

```
SQL> @GetHparDes.sql
Enter value for par: log_parallelism
```

```
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%log_parallelism%'
NAME                           VALUE                DESCRIB
------------------------------ -------------------- -----------------------------------
_log_parallelism               1                    Number of log buffer strands
_log_parallelism_max           2                    Maximum number of log buffer strands
_log_parallelism_dynamic       TRUE                 Enable dynamic strands
```

但是需要注意的是当日志并行度被设置大于 1 之后，LogMiner 将不能解析日志文件，ORA-01374 号错误提示说明了这个问题：

```
ORA-01374: _log_parallelism_max greater than 1 not supported in this release
Cause: LogMiner does not mine redo records generated with log_parallelism set to a value greater
than 1
Action: none
```

相较 Oracle9i 中的 Public Redolog Strands(缩写为 PBRS)，Oracle10gR2 中更引入了 Private Redolog Strands 的概念（缩写为 PVRS），在 PVRS 机制下，数据库可以在共享池中分配大量的小的私有内存，通常每个大小在 64～128K 左右，被独立的 Redo Allocation Latch 所保护，当数据库中特定类型的小事务开始时会被绑定到独立且空闲 PVRS，每个 Buffer 绑定一个活动事务。在新的机制下，Redo 产生后可以直接存入 PVRS，而不再保存在 PGA 中，这样就不再需要额外的内存拷贝过程，Redo Copy Latch 也就不再需要（PVRS 也因此被称为 ZERO-COPY Redo），而 Redo Copy 正是引发 Redo Allocation Latch 竞争的根源。

图 7-4 说明了在不断改进后的 Redo 以及 Log Buffer 原理：

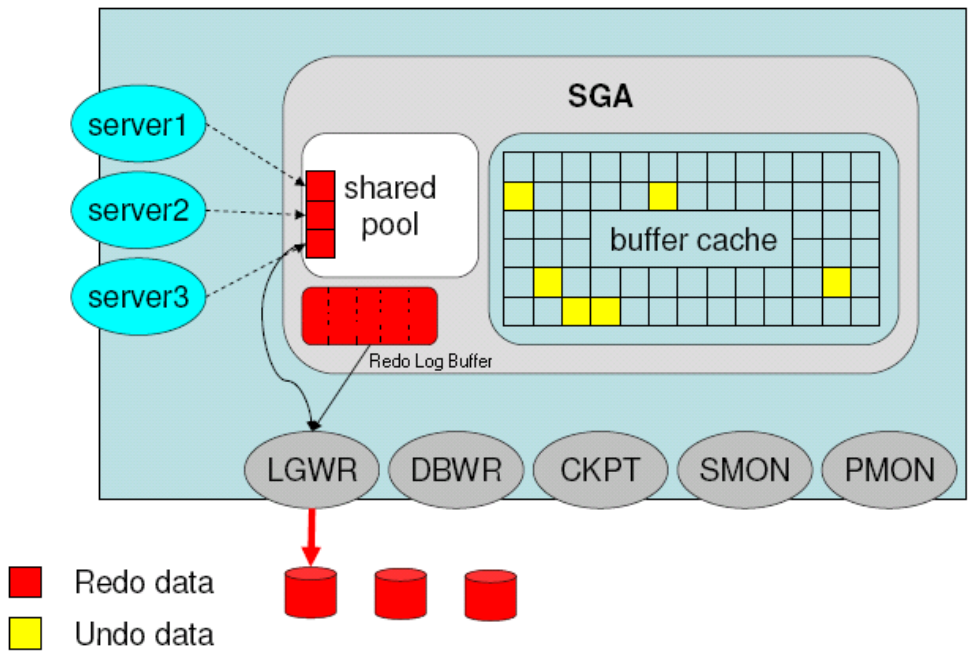


图 7-4

新的机制下，在进行 Redo 写出工作时，LGWR 需要做的工作就是将 PBRS 和 PVRS 中的内容写出，当 Redo Flush 发生时，所有的 Public Redo Allocation Latch 需要被获取，所有 Public Strands 的 Redo Copy Latch 需要被检查，所有包含活动事务的 Private Strands 需要被持有。

以下是 PVRS 在共享池中的内存分配信息（注意在 RAC 环境中不适用 PVRS）：

```
SQL> select banner from x$version where indx in (0,3);
BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
TNS for Linux: Version 10.2.0.1.0 - Production
SQL> select * from v$sgastat where name='private strands';
POOL         NAME                            BYTES
------------ -------------------------- ----------
shared pool  private strands               4060160
```

在 Oracle10gR2 中，Oracle 通过使用多个 Redo Allocation Latch 来提高并发性能：

```
SQL> select addr,latch#,child#,name,gets,immediate_gets,immediate_misses
  2  from v$latch_children where name = 'redo allocation';
ADDR     LATCH# CHILD# NAME             GETS IMMEDIATE_GETS IMMEDIATE_MISSES
-------- ------ ------ ------------------- ---------- -------------- ----------------
573EC7BC    148      1 redo allocation          569765      203606724             9329
573EC820    148      2 redo allocation          373552       44202744              580
573EC884    148      3 redo allocation          109793              0                0
573EC8E8    148      4 redo allocation           56855              0                0
573EC94C    148      5 redo allocation           24753              0                0
573EC9B0    148      6 redo allocation          203959              0                0
573ECA14    148      7 redo allocation            6352              0                0
573ECA78    148      8 redo allocation            6352              0                0
573ECADC    148      9 redo allocation            6352              0                0
573ECB40    148     10 redo allocation            6352              0                0
573ECBA4    148     11 redo allocation            6352              0                0
573ECC08    148     12 redo allocation            6352              0                0
573ECC6C    148     13 redo allocation            6352              0                0
573ECCD0    148     14 redo allocation            6352              0                0
573ECD34    148     15 redo allocation            6352              0                0
573ECD98    148     16 redo allocation            6352              0                0
573ECDFC    148     17 redo allocation            6352              0                0
573ECE60    148     18 redo allocation            6352              0                0
573ECEC4    148     19 redo allocation            6352              0                0
573ECF28    148     20 redo allocation            6352              0                0
20 rows selected.
```

以下是与 PVRS 相关的几个隐含参数，缺省的_log_private_parallelism 被设置为 FALSE：

```
SQL> @GetHidPar
Enter value for par: log_private
NAME                           VALUE PDESC
------------------------------ ----- ----------------------------------------------------
_log_private_parallelism       FALSE Number of private log buffer strands for zero-copy redo
_log_private_parallelism_mul 10      Active sessions multiplier to deduce number of private
                                     strands
_log_private_mul               5     Private strand multiplier for log space preallocation
```

PRVS 的统计数据可以从 V$SYSSTAT 视图查询得到，由于 IMU（In Memory Undo）与 PRVS 紧密相关，所以两者的信息具有相关性（IMU 在下一章中介绍）：

```
SQL>  select name,value from v$sysstat where upper(name) like '%IMU%';
NAME                                                             VALUE
---------------------------------------- ----------------------------
doubling up with imu segment                                         0
IMU commits                                                      26346
IMU Flushes                                                      11391
IMU contention                                                      12
IMU recursive-transaction flush                                      9
IMU undo retention flush                                             0
IMU ktichg flush                                                     0
IMU bind flushes                                                     0
IMU mbu flush                                                        0
IMU pool not allocated                                            4504
IMU CR rollbacks                                                   222
IMU undo allocation size                                      78339248
IMU Redo allocation size                                      8012716
IMU- failed to get a private strand                              4504
OS Maximum resident set size                                         0
```

在实施了 PRVS 之后，在告警日志信息中可能看到如下信息，这里的 Private_strands 就是新特性引入的新的提示：

```
Mon Sep 26 13:29:25 2005
Private_strands 3 at log switch
Thread 1 advanced to log sequence 58
  Current log# 3 seq# 58 mem# 0: /u01/oradata/onlinelog/o1_mf_3_1m7lp0ht_.log
  Current log# 3 seq# 58 mem# 1: /u01/oradata/onlinelog/o1_mf_3_1m7lp0rp_.log
Thread 1 cannot allocate new log, sequence 59
Checkpoint not complete
  Current log# 3 seq# 58 mem# 0: /u01/oradata/onlinelog/o1_mf_3_1m7lp0ht_.log
```

```
  Current log# 3 seq# 58 mem# 1: /u01/oradata/onlinelog/o1_mf_3_1m7lp0rp_.log
```

更进一步的如果收到提示 Private strand flush not complete，这是指从内存到 Redo Log File 的写出尚未完成：

```
Fri May 19 12:47:29 2006
Thread 1 cannot allocate new log, sequence 18358
Private strand flush not complete
Current log# 7 seq# 18357 mem# 0: /u03/oradata/bitst/redo07.log
Thread 1 advanced to log sequence 18358
Current log# 8 seq# 18358 mem# 0: /u03/oradata/bitst/redo08.log
```

通过 V$EVENT_NAME 可以找到关于这个新等待的说明：

```
SQL> select name from v$event_name where upper(name) like '%STRAND%';
NAME
----------------------------------------------------------------
log file switch (private strand flush incomplete)
```

7.6Redo 的内容

大家知道，Oracle 通过 Redo 来实现快速提交，一方面是因为 Redo Log File 可以连续、顺序的快速写出，另外一个方面也和 Redo 记录的精简内容有关。

为了了解 Redo 的内容，需要了解两个概念：

1．改变向量（Change Vector）

改变向量表示对数据库内某一个数据块所做的一次变更。改变向量（Change Vector）中包含了变更的数据块的版本号、事务操作代码、变更从属数据块的地址（DBA）以及更新后的数据。

例如一个 Update 事务包含一系列的改变向量，对于数据块的修改是一个向量，对于回滚段的修改又是一个向量。

2．重做记录（Redo Record）

重做记录通常由一组改变向量组成，是一个改变向量的集合，代表一个数据库的变更（insert、Update、Delete 等操作），构成数据库变更的最小恢复单位。

例如一个 Update 的重做记录包括相应的回滚段的改变向量和相应的数据块的改变向量等。

下面以一个更新（Update）操作为例介绍一下这个过程，如图 7-5 所示。

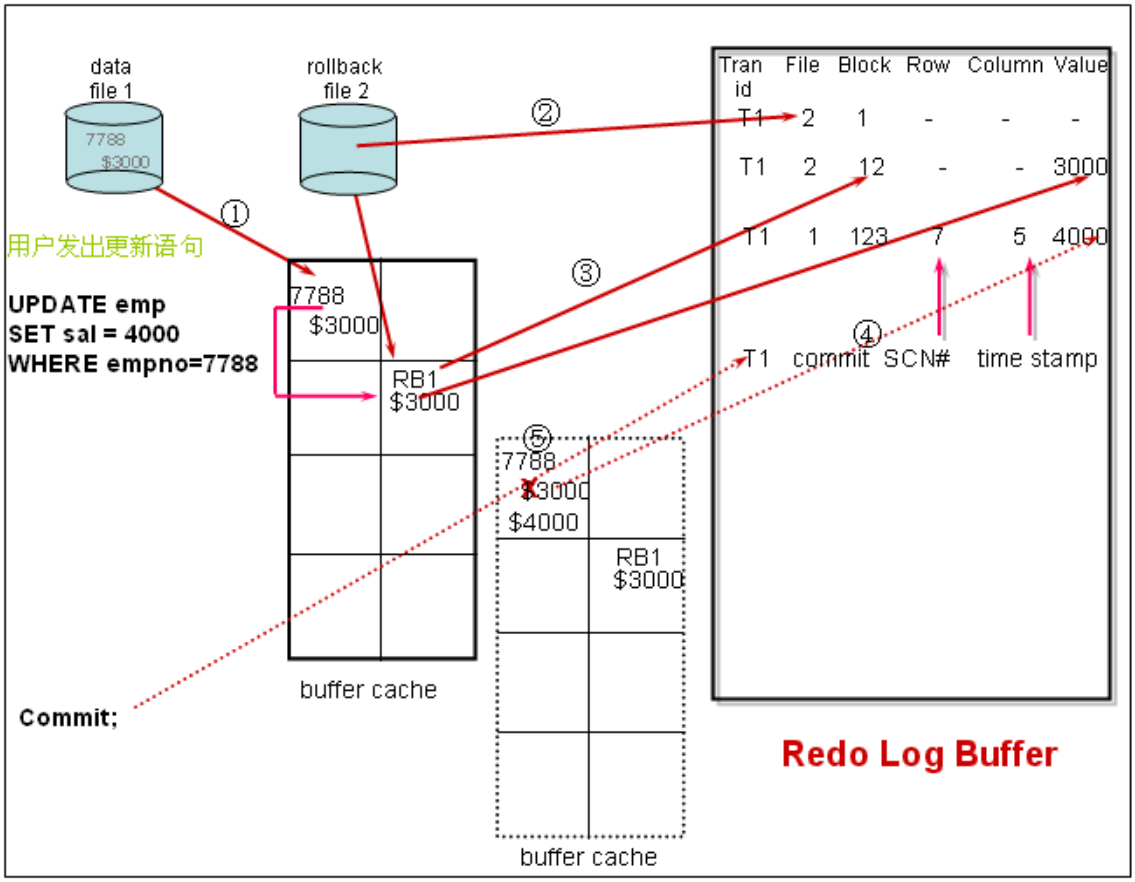


图 7-5

假定我们发出了一个更新语句：

```
UPDATE emp SET sal = 4000 Where empno= 7788;
```

看一下这个语句是怎样执行的（为了简化描述，我们尽量简化了情况）：

1. 检查 empno=7788 记录在 Buffer Cache 中是否存在，如果不存在则读取到 Buffer Cache 中
2. 在回滚表空间的相应回滚段事务表上分配事务槽，这个操作需要记录 Redo 信息
3. 从回滚段读入或者在 Buffer Cache 中创建 sal=3000 的前镜像，这需要产生 Redo 信息并记入 Redo Log Buffer
4. 修改 Sal=4000，这是 update 的数据变更，需要记入 Redo Log Buffer。
5. 当用户提交时，会在 Redo Log Buffer 记录提交信息，并在回滚段标记该事务为非激活（inactive）。

让我们通过一个具体的试验来再现这个过程。

1. 首先通过 switch logfile 切换日志

   使用 sys 用户进行日志切换，使得接下来的更新可以使用新的日志。

```
SQL> alter system switch logfile;
```

```
System altered.
SQL> select * from v$log;
GROUP#    THREAD#  SEQUENCE#      BYTES    MEMBERS ARC STATUS   FIRST_CHANGE# FIRST_TIM
------ ---------- ---------- ---------- ---------- --- -------- ------------- ---------
     1          1        310   10485760          1 NO  ACTIVE      8.9035E+12 26-MAR-06
     2          1        309   10485760          1 NO  INACTIVE    8.9035E+12 19-MAR-06
     3          1        311   10485760          1 NO  CURRENT     8.9035E+12 26-MAR-06
     4          1        308    1048576          1 NO  INACTIVE    8.9035E+12 19-MAR-06
```

2. 更新并提交事务

```
SQL> select * from emp where empno=7788;
     EMPNO ENAME   JOB    MGR    HIREDATE   SAL       COMM     DEPTNO
---------- ------ ------- ------ --------- ---------- ------- ----------
     7788  SCOTT ANALYST 7566    19-APR-87  3000                 20
SQL> update emp set sal=4000 where empno=7788;
1 row updated.
SQL> commit;
Commit complete.
```

3. 确认 Session 信息

```
SQL> select sid,serial#,username from v$session
  2  where username='SCOTT';
       SID    SERIAL# USERNAME
---------- ---------- ------------------------------
        13        405 SCOTT
```

4. 使用 sys 用户在另外 Session 转储日志文件

```
SQL> ALTER SYSTEM DUMP LOGFILE '/opt/oracle/oradata/conner/redo03.log';
System altered.
SQL> @gettrcname
TRACE_FILE_NAME
--------------------------------------------------------------------------------
/opt/oracle/admin/conner/udump/conner_ora_31885.trc
```

5. 获取 Trace 文件

从日志文件的转储信息中，我们可以很容易的找到这个事务（sid= 15,serial#=43870）的信息，为了方便说明，我将这段日志分开讲解：

a）改变向量 1，这是对于回滚段头的修改，分配事务表，从绝对文件号为 2（AFN:2）我们可以知道这是 UNDO 表空间，通过 UBA 机 DBA 的换算我们能够找到相应的 Block

```
REDO RECORD - Thread:1 RBA: 0x000137.00000005.0010 LEN: 0x0198 VLD: 0x01
SCN: 0x0819.0036f14d SUBSCN:  1 03/26/2006 12:01:44
CHANGE #1 TYP:0 CLS:19 AFN:2 DBA:0x00800009 SCN:0x0819.0036f03c SEQ:  1 OP:5.2
ktudh redo: slt: 0x001d sqn: 0x000038ea flg: 0x0012 siz: 108 fbi: 0
```

```
            uba: 0x008000c3.04b1.0c    pxid:  0x0000.000.00000000
```

b)改变向量 2

这里记录的是前镜像信息，注意到，**col　5: [2]　c2 1f** 记录的就是对于 COL 5 的修改，修改前的数值是 3000 （c2 1f，数值及存储转换方式参考本章附录）

```
CHANGE #2 TYP:0 CLS:20 AFN:2 DBA:0x008000c3 SCN:0x0819.0036f03b SEQ:  1 OP:5.1
ktudb redo: siz: 108 spc: 6740 flg: 0x0012 seq: 0x04b1 rec: 0x0c
            xid:  0x0002.01d.000038ea
ktubl redo: slt: 29 rci: 0 opc: 11.1 objn: 7961 objd: 7961 tsn: 0
Undo type:  Regular undo        Begin trans    Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
             0x00000000  prev ctl uba: 0x008000c3.04b1.0b
prev ctl max cmt scn:  0x0819.00364c81  prev tx cmt scn:  0x0819.00365073
KDO undo record:
KTB Redo
op: 0x03  ver: 0x01
op: Z
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x00405c5a  hdba: 0x00405c59
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 7(0x7) flag: 0x2c lock: 0 ckix: 0
ncol: 8 nnew: 1 size: 0
col  5: [ 2]  c2 1f
```

c)改变向量 3

这里记录的是对于数据块的修改，**col　5: [2]　c2 29** 记录的是对于 COL 5 的修改，修改后的值为 4000（c2 29）.

```
CHANGE #3 TYP:2 CLS: 1 AFN:1 DBA:0x00405c5a SCN:0x0819.0036efb1 SEQ:  1 OP:11.5
KTB Redo
op: 0x11  ver: 0x01
op: F  xid:  0x0002.01d.000038ea    uba: 0x008000c3.04b1.0c
Block cleanout record, scn:  0x0819.0036f14d ver: 0x01 opt: 0x02, entries follow...
  itli: 1  flg: 2  scn: 0x0819.0036efb1
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x00405c5a  hdba: 0x00405c59
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 7(0x7) flag: 0x2c lock: 2 ckix: 0
ncol: 8 nnew: 1 size: 0
col  5: [ 2]  c2 29
```

d)改变向量 4

当事务提交之后，记录的 SCN 信息，注意这里标记为“MEDIA RECOVERY MARKER SCN”,也就是说，这是一个可以恢复的时间点，事务的恢复，必须以 Redo Record 为最小单位。

```
CHANGE #4 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:5.20
```

e)session 信息

最后部分记录的是产生这些 Redo 的 session 信息。

```
session number   = 13
serial  number   = 405
transaction name =
```

从以上的分析中可以看到，对于数据块的修改，如果执行写出，那么通常需要写出 8K 的 Block，而对于 Redo 日志来说，重做信息却相当精简，Oracle 只需要记录那些重构事务必须的信息（事务号，文件号，块号，行号，字段等）即可，这个数据量大大减少。

3．操作代码(OP Code)

在以上的改变向量分析中,存在一个重要的操作代码(OP=Operation Code),该代码标志了操作的类型.下表列出了不同代码代表的操作级别,进一步的,操作级别和具体编号联合就可以详细的标示一个 Change Vector 的操作:

Level	Description
4	Block Cleanout
5	Transaction Layer(Undo)
10	Index Operation
11	Table Operation(DML)
13	Block Allocation
14	Extent Allocation
17	Backup Management
18	Online Backup
19	Direct Load
20	Transaction Metadata(LogMiner)
22	Space Management(ASSM)
23	Block Write(DBWR)
24	DDL Statement

例如对于 DML 事务,其 Level 是 11,相应的操作代码如下表所示:

OP Code	Description
11.2	Insert Row Piece
11.3	Drop Row Piece
11.4	Lock Row Piece
11.5	Update Row Piece
11.6	Overflow Row Piece
11.11	Insert Row Array
11.12	Delete Row Array

对于前面的更新示范,其 OP 代码正是 11.5 ．对于 UNDO 的操作,其代码如下表所示,其中 5.1 表示对 UNDO 块或者 UNDO Header 的操作,5.2 表示对于回滚段头的更新操作:

OP Code	Description
5.1	Undo block or undo segment header
5.2	Update rollback segment header
5.4	Commit transaction
5.11	Rollback DBA in transaction table entry
5.19	Transaction start audit log record
5.20	Transaction continue audit log record

4．日志分析获取篡改信息案例一则

在某客户数据库系统中,遇到一则数据被恶意篡改的案例.某用户账户余额为 0 元,被修改为 40000 元.这一篡改说明数据库存在极大的安全隐患,如果不能及时找和消除这一隐患,则数据库可能随时处在危险之中,如果遭遇大规模篡改或数据截断,则损失将会极其巨大.

用户遭遇的具体问题为:

login_name=7847254的balance字段数据被恶意修改,客户反映数据被修改的时间段为7.5-7.6日9:00，balance字段被修改为40000。客户于7.6日上午11:00将此值重置为0.

根据这一信息,我们开始分析日志,进行追踪.

首先获取该行记录的 ROWID 等信息:

```
SQL> select rowid from "CINMS"."BROAD_SUBSCRB" where login_name='7847254';
ROWID
------------------
AAAQXKAAvAAAB2wAAn
```

根据 ROWID 可疑找到该记录的块号等信息:

```
SQL> select get_rowid('AAAQXKAAvAAAB2wAAn') from dual;
```

```
GET_ROWID('AAAQXKAAVAAAB2WAAN')
--------------------------------------------------------------------------------
Object# is      :67018
Relative_fno is :47
Block number is :7600
Row number is   :39
```

get_rowid 是根据 dbms_rowid 包编写的一个函数,用于简化 ROWID 拆分,其代码如下:

```
create or replace function get_rowid
(l_rowid in varchar2)
return varchar2
is
ls_my_rowid  varchar2(200);
rowid_type  number;
object_number  number;
relative_fno  number;
block_number  number;
row_number  number;
begin
 dbms_rowid.rowid_info(l_rowid,rowid_type,object_number,relative_fno,        block_number,
row_number);
 ls_my_rowid := 'Object# is      :'||to_char(object_number)||chr(10)||
  'Relative_fno is :'||to_char(relative_fno)||chr(10)||
  'Block number is :'||to_char(block_number)||chr(10)||
  'Row number is   :'||to_char(row_number);
 return ls_my_rowid ;
end;
/
```

我们可疑看到,被修改的数据行位于文件 47 的 Block 号 7600 上.通过 Logminer 解析日志,可疑找到所有对于这个数据块的修改.

解析日志的步骤如下:

```
SQL>   EXECUTE   DBMS_LOGMNR.ADD_LOGFILE(LogFileName   =>   '/ora_arch_backup/201107arch/1_12446.dbf',Options   =>
DBMS_LOGMNR.NEW);


PL/SQL procedure successfully completed.


SQL>        EXECUTE        DBMS_LOGMNR.START_LOGMNR(OPTIONS        =>        DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG
+ DBMS_LOGMNR.COMMITTED_DATA_ONLY);


PL/SQL procedure successfully completed.
```

解析完成之后,可疑通过 v$logmnr_contents 视图来查找相关的修改操作.在该日志中,解析后共得到约 18 万行数据:

```
SQL> select count(*) from v$logmnr_contents;

  COUNT(*)
----------
    182001
```

对数据块 7600 的修改只有一次:

```
SQL> select count(*) from v$logmnr_contents where DATA_BLK#=7600;

  COUNT(*)
----------
         1
```

该修改正是位于 BROAD_SUBSCRB 表,但是 SQL_REDO 和 SQL_UNDO 信息不可用:

```
SQL> select scn,seg_name,sql_redo,sql_undo,rs_id from v$logmnr_contents where scn=5568491262;
        SCN SEG_NAME           SQL_REDO       SQL_UNDO       RS_ID
---------- ---------------- ------------ ------------ -------------------------
5568491262 BROAD_SUBSCRB    Unsupported  Unsupported  0x00309e.00028a0a.0010
SQL>  select  scn,session#,sql_redo,sql_undo,rs_id  from  v$logmnr_contents  where
DATA_BLK#=7600;
         SCN    SESSION# SQL_REDO      SQL_UNDO      RS_ID
---------- ---------- ------------ ------------ ------------------------------
5568491262          90 Unsupported  Unsupported   0x00309e.00028a0a.0010
```

根据 session#,可以将这个会话的所有操作查询出来,注意第一条信息的绝对文件号是 2,也就是 UNDO 表空间,说明该条目记录的是事务相关信息;第二条信息记录的是对于绝对文件号 47 块号 7602 号的操作:

```
SQL> select ABS_FILE#,REL_FILE#,DATA_BLK#,DATA_OBJ#,SEG_NAME ,rs_id from v$logmnr_contents
where session#=90 and seg_name='BROAD_SUBSCRB';

 ABS_FILE#  REL_FILE#  DATA_BLK#  DATA_OBJ# SEG_NAME        RS_ID
---------- ---------- ---------- ---------- --------------- ------------------------------
         2         47       7600      66237 BROAD_SUBSCRB    0x00309e.00028a0a.0010
        47         47       7602      66237 BROAD_SUBSCRB    0x00309e.00028b4d.0010
SQL>      select      TIMESTAMP,ABS_FILE#,REL_FILE#,DATA_BLK#,DATA_OBJ#,sql_redo      from
v$logmnr_contents where session#=90 and seg_name='BROAD_SUBSCRB';

TIMESTAMP            ABS_FILE#  REL_FILE#  DATA_BLK#  DATA_OBJ# SQL_REDO
-------------------- ---------- ---------- ---------- ---------- -------------------------
2011-07-05 16:41:38          2         47       7600      66237 Unsupported
```

```
2011-07-05 16:41:54         47         47       7602      66237 Unsupported
```

根据前面查询到的 scn 等信息,可以再次搜索,获取用户的连接信息等,根据这些连接信息,已经可以确定连接的客户端:

```
SQL> select session_info from v$logmnr_contents where scn=5568491262;
SESSION_INFO
--------------------------------------------------------------------------------
----------------------------------------
login_username=CINMS client_info= OS_username=Administrator Machine_name=WORKGROUP\ANY
login_username=CINMS client_info= OS_username=Administrator Machine_name=WORKGROUP\ANY
```

但是我们还需要进一步确认,该次操作是否就是那一次恶意的修改.

我们可以将日志文件转储出来,然后根据查询到的 RS_ID(RS_ID 记录的是 REDO Record SET的ID号,可以识别 REDO 中的 REDO 记录条目,实际上也就是 RBA 信息)去搜索,找到相关的 UNDO 信息,转出日志可以用类似如下命令:

alter system dump logfile '/ora_backup/arch/1_12606.dbf'

根据前面的查询相关的事务和修改信息的 RS_ID 分别为: 0x00309e.00028a0a.0010 和 0x00309e.00028b4d.0010.

在跟踪文件中可以找到如下第一个 RS_ID 详细的 REDO 信息,这是事务开始之前,对于 UNDO 的修改和锁定:

```
REDO RECORD - Thread:1 RBA: 0x00309e.00028a0a.0010 LEN: 0x01a4 VLD: 0x01
>>>>这里的 RBA 信息
SCN: 0x0001.4be86efe SUBSCN:  1 07/05/2011 16:41:38
CHANGE #1 TYP:0 CLS:21 AFN:2 DBA:0x00800029 SCN:0x0001.4be86ed9 SEQ:  1 OP:5.2
>>>> OP:5.2 这是更新 UNDO 段 Header,创建一个事务
ktudh redo: slt: 0x0022 sqn: 0x0019482c flg: 0x0012 siz: 108 fbi: 0
            uba: 0x00801542.af17.04    pxid:  0x0000.000.00000000
CHANGE #2 TYP:0 CLS:22 AFN:2 DBA:0x00801542 SCN:0x0001.4be86ed8 SEQ:  1 OP:5.1
>>>>OP:5.1 这是 UNDO BLOCK 和 UNDO HEADER 的修改,
ktudb redo: siz: 108 spc: 7640 flg: 0x0012 seq: 0xaf17 rec: 0x04
            xid:  0x0003.022.0019482c
ktubl redo: slt: 34 rci: 0 opc: 11.1 objn: 66237 objd: 67018 tsn: 8
Undo type:  Regular undo        Begin trans    Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
             0x00000000  prev ctl uba: 0x00801542.af17.03
prev ctl max cmt scn:  0x0001.4be86985  prev tx cmt scn:  0x0001.4be869b3
KDO undo record:
KTB Redo
```

```
op: 0x04  ver: 0x01
op: L  itl: xid:  0x0005.023.0019a945 uba: 0x00800c54.b1a9.0b
                      flg: C---    lkc:  0     scn: 0x0001.4be11d6c
KDO Op code: LKR row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db0  hdba: 0x0900e509
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 39 to: 0
CHANGE #3 TYP:2 CLS: 1 AFN:47 DBA:0x0bc01db0 SCN:0x0001.4be21203 SEQ:  1 OP:11.4
>>>>OP:11.4 是锁定行,实际上也就是锁定 ITL
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0003.022.0019482c    uba: 0x00801542.af17.04
KDO Op code: LKR row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db0  hdba: 0x0900e509
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 39 to: 2
CHANGE #4 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:5.19
session number   = 90
serial  number   = 14480
current username = CINMS
login   username = CINMS
client info      =
OS username      = Administrator
Machine name     = WORKGROUP\ANY
OS terminal      = ANY
OS process id    = 680:2868
OS program name  =
transaction name =
```

接下来找到第二个 RS_ID 的详细信息,这里就是真正的用户数据更新信息:

```
REDO RECORD - Thread:1 RBA: 0x00309e.00028b4d.0010 LEN: 0x0114 VLD: 0x01
>>>>RBA 信息,也即 RS_ID
SCN: 0x0001.4be86fb9 SUBSCN:  1 07/05/2011 16:41:54
CHANGE #1 TYP:0 CLS:22 AFN:2 DBA:0x00801542 SCN:0x0001.4be86efe SEQ:  1 OP:5.1
>>>>OP:5.1 这里记录的是前镜像数据
ktudb redo: siz: 116 spc: 7530 flg: 0x0022 seq: 0xaf17 rec: 0x05
            xid:  0x0003.022.0019482c
ktubu redo: slt: 34 rci: 4 opc: 11.1 objn: 66237 objd: 67018 tsn: 8
Undo type:  Regular undo       Undo type:  Last buffer split:  No
```

```
Tablespace Undo:  No
            0x00000000
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
op: L  itl: xid:  0x000a.01e.001a0c96 uba: 0x00800c0a.b193.29
                     flg: C---    lkc:  0     scn: 0x0001.4bb7744a
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db2  hdba: 0x0900e509
itli: 3  ispac: 0  maxfr: 4863
tabn: 0 slot: 47(0x2f) flag: 0x0c lock: 0 ckix: 0
ncol: 33 nnew: 2 size: -1
col  7: [ 1]  35
col 15: [ 1]  80
>>>>这里记录了修改前的值,分别修改了第 7 和第 15 列信息(由 0 编号,等于第 8 和 16 列)
>>>>80 就是十进制的 0
CHANGE #2 TYP:2 CLS: 1 AFN:47 DBA:0x0bc01db2 SCN:0x0001.4be1efdb SEQ:  1 OP:11.5
>>>>OP:11.5 指更新行记录信息
KTB Redo
op: 0x01  ver: 0x01
op: F  xid:  0x0003.022.0019482c    uba: 0x00801542.af17.05
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0bc01db2  hdba: 0x0900e509
itli: 3  ispac: 0  maxfr: 4863
tabn: 0 slot: 47(0x2f) flag: 0x0c lock: 3 ckix: 0
ncol: 33 nnew: 2 size: 1
col  7: [ 1]  31
col 15: [ 2]  c3 05
>>>>这里是更新后的值,c3 05 就是 40000
```

最后整理确认时间:

```
SQL> select TIMESTAMP,ABS_FILE#,REL_FILE#,DATA_BLK#,DATA_OBJ#,rs_id
  from v$logmnr_contents where session#=90 and seg_name='BROAD_SUBSCRB';
TIMESTAMP            ABS_FILE#  REL_FILE#  DATA_BLK#  DATA_OBJ# RS_ID
-------------------        ----------        ----------        ----------        ----------
------------------------------
2011-07-05 16:41:38          2         47       7600      66237  0x00309e.00028a0a.0010
2011-07-05 16:41:54         47         47       7602      66237  0x00309e.00028b4d.0010
```

通过对监听日志的详细分析，发现了机器名为 ANY 的机器于 7 月 5 日下午登陆服务器的

记录,整个过程被彻底解析:

```
05-JUL-2011 16:41:22 * (CONNECT_DATA=(SID=cinms)(CID=(PROGRAM=C:\Program
Files\PLSQL Developer\plsqldev.exe)(HOST=ANY)(USER=Administrator))) *
(ADDRESS=(PROTOCOL=tcp)(HOST=100.40.0.165)(PORT=1940)) * establish * cinms * 0
```

剩下的就是要看如何去和相关人员交流沟通了.

作为一个数据库从业人员,我们需要从中了解到:**接触数据,意味着责任与义务,必须具备严格的职业道德操守,做数据的保护人而不是威胁者**.

7.7产生多少 Redo?

我们知道，对于数据库的修改操作都会记录 redo，那么不同的操作会产生多少 Redo 呢？可以通过以下一些方式来查询。

1．在 SQL*plus 中使用 autotrace 的功能时

当我们在 SQL*plus 中启用 autotrace 跟踪后，在执行了特定的 DML 语句时，Oracle 会显示该语句的统计信息，其中，redo size 一栏表示的就是该操作产生的 Redo 的数量，其单位为 Bytes：

```
SQL> set autotrace trace stat
SQL> insert into eygle
  2  select * from eygle;
28 rows created.
Statistics
----------------------------------------------------------
          4  consistent gets
          0  physical reads
        776  redo size
```

2．通过 v$mystat 查询

Oracle 通过 v$mystat 视图记录当前 Session 的统计信息，我们也可以从该视图中查询得到 Session 的 Redo 生成情况：

```
SQL> col name for a30
SQL> select a.name,b.value
  2  from v$statname a,v$mystat b
  3  where a.STATISTIC# = b.STATISTIC# and a.name = 'redo size';
NAME                           VALUE
-----------------------    ----------
redo size                      56540
SQL> insert into eygle select * from eygle;
56 rows created.
```

```
SQL> select a.name,b.value
  2  from v$statname a,v$mystat b
  3  where a.STATISTIC# = b.STATISTIC# and a.name = 'redo size';
NAME                                 VALUE
---------------------------- ----------
redo size                            57784
SQL> select 57784 -56540 from dual;
57784-56540
-----------
       1244
```

3．通过 v$sysstat 查询

对于数据库全局 Redo 的生成量，我们可以通过 v$sysstat 视图来查询得到：

```
SQL> col value for 99999999999999
SQL> select name,value
  2  from v$sysstat where name='redo size';
NAME                                      VALUE
---------------------------- ---------------
redo size                          2065603825384
```

从 v$sysstat 视图中得到的是自数据库实例启动以来的累积日志生成量，我们可以根据实例启动时间来大致估算每天数据库的日志生成量：

```
SQL> alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss';
Session altered.
SQL> select startup_time from v$instance;
STARTUP_TIME
-------------------
2005-08-25 03:01:04
SQL>  select (select value/1024/1024/1024 from v$sysstat where name='redo size')/
  2    (select round(sysdate - ( select startup_time from v$instance)) from dual)
REDO_GB_PER_DAY
  3  from dual;
REDO_GB_PER_DAY
---------------
     7.42880595
```

如果数据库运行在归档模式下，由于其他因素的影响，以上 Redo 生成量并不代表归档日志的大小，但是可以通过一定的加权提供参考。

至于归档日志的生成量，可以通过 v$archived_log 视图，根据一段时间的归档日志量进行估算得到。该视图中记录了归档日志的主要信息：

```
SQL> select name,COMPLETION_TIME,BLOCKS*BLOCK_SIZE/1024/1024 Mb
```

```
from v$archived_log where rownum <11
and COMPLETION_TIME between trunc(sysdate) -2 and trunc(sysdate) -1;
NAME                              COMPLETION_TIME  MB
----------------------------------- ------------------- ----------
/bsarch/oracle/1_171913.dbf             2006-05-09 00:00:09  19.996582
/bsarch/oracle/1_171914.dbf             2006-05-09 01:01:13 19.9990234
/bsarch/oracle/1_171915.dbf             2006-05-09 01:35:57 19.9931641
/bsarch/oracle/1_171917.dbf             2006-05-09 02:02:46 19.9990234
/bsarch/oracle/1_171917.dbf             2006-05-09 02:06:23 19.9990234
/bsarch/oracle/1_171918.dbf             2006-05-09 02:09:56 19.9990234
/bsarch/oracle/1_171919.dbf             2006-05-09 02:13:36 19.9990234
/bsarch/oracle/1_171920.dbf             2006-05-09 02:16:21 19.9990234
/bsarch/oracle/1_171921.dbf             2006-05-09 03:00:28 19.9990234
/bsarch/oracle/1_171922.dbf             2006-05-09 03:01:03 4.03271484

10 rows selected.
```

某日全天的日志生成可以通过如下查询计算：

```
SQL> select trunc(COMPLETION_TIME),sum(Mb)/1024 DAY_GB from
  2  (select name,COMPLETION_TIME,BLOCKS*BLOCK_SIZE/1024/1024 Mb from v$archived_log
  3  where COMPLETION_TIME between trunc(sysdate) -2 and trunc(sysdate) -1)
  4  group by trunc(COMPLETION_TIME)
  5 /
TRUNC(COM     DAY_GB
---------  ----------
09-MAY-06  17.8974366
```

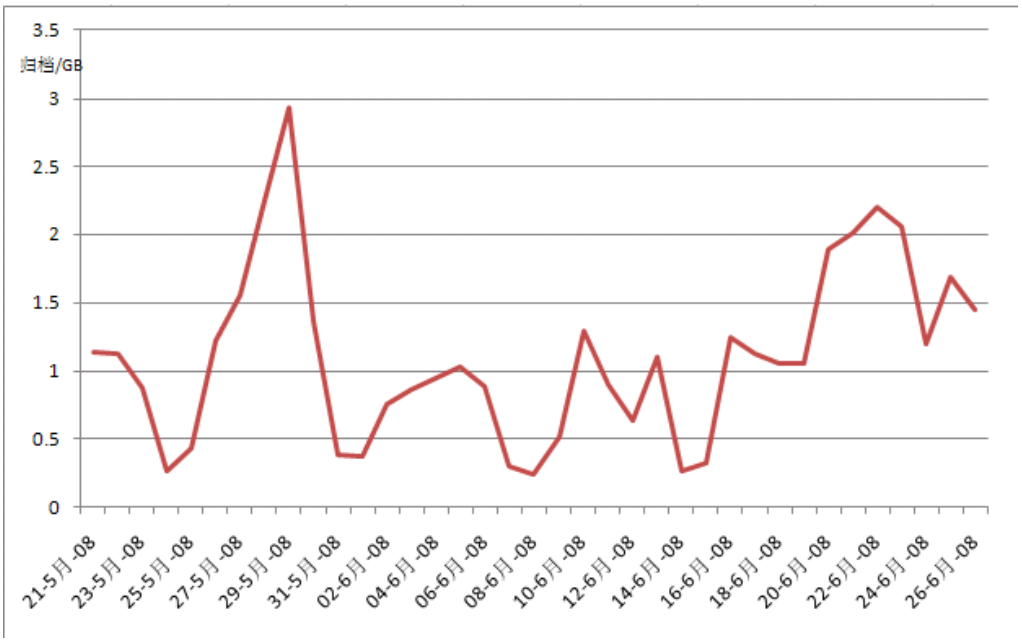
最近日期的日志生成统计：

```
SQL> SELECT   TRUNC (completion_time), SUM (mb) / 1024 day_gb
  2      FROM (SELECT NAME, completion_time, blocks * block_size / 1024 / 1024 mb
  3              FROM v$archived_log)
  4  GROUP BY TRUNC (completion_time);
TRUNC(COM     DAY_GB
--------- ----------
28-APR-06 8.63226318
29-APR-06 11.6235332
30-APR-06 17.7366991
01-MAY-06 35.7830167
02-MAY-06 11.0832992
03-MAY-06 11.6479049
04-MAY-06 8.76808453
```

```
05-MAY-06 9.68909311
06-MAY-06 14.186295
07-MAY-06 10.4164033
08-MAY-06 19.9013429
09-MAY-06 17.8974366
10-MAY-06 19.4107008
11-MAY-06 11.2606988

14 rows selected.
```

根据每日归档的生成量，我们也可以反过来估计每日的数据库活动性及周期性，并决定空间分配等问题。将这些数据导入 Excel，可以很容易的获得直观的曲线图，进行辅助分析与报告（以下图表来自不同数据）：



7.8redo 写的触发条件

为了保证用户可以快速提交，LGWR 的写出必须非常活跃，实际上也确实如此，我们非常熟悉的 LGWR 写触发条件就有：

7.8.1 每 3 秒钟超时（Timeout）

当 LGWR 处于空闲状态时，它依赖于 rdbms ipc message 等待，处于休眠状态，直到 3 秒超时时间到。如果 LGWR 发现有 redo 需要写出，那么 LGWR 将执行写出操作，log file parallel

write 等待事件将会出现。

启用 10046 事件，从 LGWR 跟踪日志中可以清楚的观察到这些事件:

```
WAIT #0: nam='rdbms ipc message' ela= 2999554 p1=300 p2=0 p3=0
WAIT #0: nam='rdbms ipc message' ela= 2999470 p1=300 p2=0 p3=0
WAIT #0: nam='rdbms ipc message' ela= 566819 p1=300 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 115 p1=1 p2=2 p3=1
WAIT #0: nam='rdbms ipc message' ela= 45752 p1=213 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 94 p1=1 p2=3 p3=1
WAIT #0: nam='rdbms ipc message' ela= 51762 p1=208 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 91 p1=1 p2=1 p3=1
WAIT #0: nam='rdbms ipc message' ela= 29033 p1=200 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 99 p1=1 p2=2 p3=1
WAIT #0: nam='rdbms ipc message' ela= 40293 p1=197 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 87 p1=1 p2=1 p3=1
```

7.8.2 阈值达到

我们在各种文档上经常会看到的 2 个触发日志写的条件是：

Redo Log Buffer 1/3 满

Redo Log Buffer 具有 1M 脏数据

这两者都是限制条件，在触发时是协同生效的。

我们知道，只要有进程（Process）在 log buffer 中分配和使用空间，已经使用的 Log buffer 的数量将被计算。如果使用的块的数量大于或等于一个隐含参数_log_io_size 的设置，那么将会触发 LGWR 写操作。

如果此时 LGWR 未处于活动状态，那么 LGWR 将被通知去执行后台写操作。

缺省的_log_io_size 等于 1/3 log buffer 大小，上限值为 1M，此参数在 X$KSPPSV 中显示的 0 值，意为缺省值。

X$KSPPSV　的含义为:　[K]ernel [S]ervice [P]arameter Component [S]ystem [V]alues

也就是，LGWR 将在 Min(1M,1/3 log buffer size)时触发。注意此处的 log buffer size 是以 log block 来衡量的。

```
SQL> @D:\GetHiddenParameter.sql
Enter value for par: log_io
old  14:   x.ksppinm like '%_&par%'
new  14:   x.ksppinm like '%_log_io%'
NAME                           VALUE                ISDEFAULT ISMOD      ISADJ
------------------------------ -------------------- --------- ---------- -----
_log_io_size                   0                       TRUE      FALSE      FALSE
```

一个常见的经验推荐是将 Log Buffer 设置为 3M 大小，就是因为当 Redo Log Buffer 为 3M

时，以上 2 个条件可能同时达到，从而可以在某种程度上避免 LGWR 的过度激活。

7.8.3 用户提交

当一个事物提交时，在 redo stream 中将记录一个提交标志。

在这些 redo 被写到磁盘上之前，这个事物是不可恢复的。所以，在事务返回成功标志给用户前，必须等待 LGWR 写完成。进程通知 LGWR 写，并且以 log file sync 事件开始休眠，超时时间为 1 秒。

Oracle 的隐含参数_wait_for_sync 参数可以设置为 false 避免 redo file sync 的等待，但是就将无法保证事务的恢复性。

```
SQL> @D:\GetHiddenParameter.sql
Enter value for par: wait_for
NAME                          VALUE              ISDEFAULT ISMOD      ISADJ
-----------------         --------------------  -----      --------- -----
_wait_for_sync              TRUE                TRUE      FALSE      FALSE
```

注意,在递归调用(recursive calls)中的提交(比如过程中的提交)不需要同步 redo 直到需要返回响应给用户。因此递归调用仅需要同步返回给用户调用之前的最后一次 Commit 操作的 RBA。

存在一个 SGA 变量用以记录 redo 线程需要同步的 log block number。如果多个提交在唤醒 LGWR 之前发生，此变量记录最高的 log block number，在此之前的所有 redo 都将被写入磁盘。这有时候被称为组提交(group commit).

7.8.4 在 DBWn 写之前

如果 DBWR 将要写出的数据的 High RBA 超过 LGWR 的 On-Disk RBA，DBWR 将通知 LGWR 去执行写出（否则这部分数据在 Recovery 时将无法恢复）。在 Oracle8i 之前，此时 DBWR 将等待 log file sync 事件；从 Oracle8i 开始，DBWR 把这些 Block 放入一个延迟队列，同时通知 LGWR 执行 redo 写出，DBWR 可以继续执行无需等待的数据写出。

在生产环境中，通常用户提交的频率是很高的，下面是来自 Statspack 的一段报告：

```
Statistic                                     Total     per Second    per Trans
--------------------------------              ------------------    --------------
.....
redo blocks written                         432,214          238.4         10.2
redo buffer allocation retries                    4            0.0          0.0
redo entries                                224,270          123.7          5.3
redo log space requests                           4            0.0          0.0
redo log space wait time                          8            0.0          0.0
redo size                               207,176,400      114,272.7      4,905.3
redo synch time                             573,356          317.3         13.6
redo synch writes                            45,230           25.0          1.1
redo wastage                              7,261,484        4,005.2        171.9
```

```
redo write time                              145,896          80.5          3.5
redo writer latching time                         37           0.0          0.0
redo writes                                   29,608          17.3          0.7
.....
user calls                                   876,983         483.7         20.8
user commits                                  42,235          23.3          1.0
.............
```

注意到这个数据库中，平均每秒用户就提交了 23.3 次。

7.9Redo Log Buffer 的大小设置

Redo Log Buffer 的大小由初始化参数 LOG_BUFFER 定义。

该参数的缺省值：Max(512 KB , 128 KB * CPU_COUNT)

通常这一缺省值是足够的，从上一节我们可以知道，Redo Log Buffer 的写出操作是相当频繁的，所以过大的 Log Buffer 设置通常是没有必要的；如果缺省值不能满足要求，根据我们前面的介绍，一般来说 3M 是一个较为合理的调整开端。

Log_buffer 参数的设置是否需要调整，可以从数据库的等待事件来判断：

```
SQL> select event#,name from v$event_name where name='log buffer space';
  EVENT#  NAME
---------- ----------------------------------------------------------------
     196   log buffer space
```

当 Log Buffer Space 等待事件出现并且较为显著时，我们可以考虑增大 Log Buffer 以缩减竞争。

从 Oracle10g 开始，LOG_BUFFER 的分配算法有所改变，由于 Granule 的引入，LOG_BUFFER 的分配也需要以 Granule 为基础，缺省的 SGA 中的 Fixed SGA Size 将和 LOG_BUFFER 共享整数倍的 Granule：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
SQL> show parameter log_buffer
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
log_buffer                           integer     7053312
SQL> select * from v$sgainfo
  2  where name in ('Fixed SGA Size','Redo Buffers','Granule Size');
NAME                                BYTES RES
------------------------------ ---------- ---
```

```
Fixed SGA Size                    1223488 No
Redo Buffers                      7163904 No
Granule Size                      4194304 No
SQL> select (1223488 +7163904)/4194304 from dual;
(1223488+7163904)/4194304
-------------------------
               1.99971008
```

如果不是有明显的性能问题，一般缺省的设置是足够的。

7.10 commit 做了什么?

当完成事务操作，发出 commit 命令之后，随后会收到一个反馈：Commit complete.

```
SQL> insert into t select * from t;
26 rows created.
SQL> commit;
Commit complete.
```

提交完成，这个提示意味着 Oracle 已经将此时间点之前的 redo 写入了重做日志文件中，这个日志写完成之后，Oracle 可以释放用户去执行其他任务。如果此后发生数据库崩溃，那么 Oracle 可以从重做日志文件中恢复这些提交过的数据，从而保证提交成功的数据不会丢失。

那么我们应该记住的 Oracle 的一个原则是：**确保提交成功的数据不丢失**。这个保证正是通过 redo 来实现的。由此可以看到日志文件对于 Oracle 的重要，为了保证日志文件的安全，Oracle 允许对重做日志文件进行镜像。

从 Oracle10g 开始，如果设置了闪回恢复区，则 Oracle 缺省的就会对日志文件进行镜像，以下输出来自一个 Oracle10g RAC 环境，每组日志包含了 2 个成员，分别位于闪回恢复区及数据区：

```
SQL> select * from v$logfile;
    GROUP# STATUS  TYPE    MEMBER                                            IS_
---------- ------- ------- ------------------------------------------------- ---
         1         ONLINE  +ORADG/smsdb/onlinelog/group_1.257.642339925       NO
         1         ONLINE  +FLHDG/smsdb/onlinelog/group_1.257.642339925       YES
         2         ONLINE  +ORADG/smsdb/onlinelog/group_2.258.642339925       NO
         2         ONLINE  +FLHDG/smsdb/onlinelog/group_2.258.642339927       YES
         3         ONLINE  +ORADG/smsdb/onlinelog/group_3.259.642339929       NO
         3         ONLINE  +FLHDG/smsdb/onlinelog/group_3.259.642339931       YES
```

镜像的好处是当某一个日志出现问题，另外一个日志仍然可用，可以保证数据不丢失，而且通常镜像存储于不同的硬盘，当某个存储出现故障时，另外的存储可以用于保证镜像日志的安全。

7.11　Log File Sync 的含义

在 Oracle 数据库的等待事件中，经常可以看到 Log file sync 处于显著的位置，类似以下 AWR 报告信息显示，Log file sync 事件占据数据库 55.1%的 DB Time：

Top 5 Timed Events

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
log file sync	947,617	2,939	3	55.1	Commit
CPU time		1,144		21.5	
latch: In memory undo latch	64,869	108	2	2.0	Concurrency
buffer busy waits	40,633	90	2	1.7	Concurrency
enq: TX - index contention	14,879	54	4	1.0	Concurrency

上一节提到 Commit 的流程，Log File Sync 就是提交的后台体现之一，这个后台流程的主要步骤是：

用户进程 Post LGWR 进程 –> LGWR 执行写出，用户进程处于 Log File Sync 等待 -> LGWR 进程发出日志写动作 -> LGWR 写完成 –> LGWR 通知用户进程 –> 用户进程标记提交完成

通过操作系统的跟踪工具，可以清晰的观察到这个流程，以下是使用 truss 命令跟踪 LGWR 进程：

```
solaris*orcl-/home/oracle$ ps -ef|grep lgwr
    grid  1479     1   0   6 月 27 ?           0:03 asm_lgwr_+ASM
  oracle  2104     1   0   6 月 27 ?           0:07 ora_lgwr_orcl
solaris*orcl-/home/oracle$ truss -p 2104
```

以下重点框住的部分就是 LGWR 在写出时执行的一个操作流程（跟踪信息来自 Solaris 下的 Oracle Database 11gR2）：

```
/1:     times(0xFFFFFD7FFFDFDF00)                          = 17525016
/1:     times(0xFFFFFD7FFFDFDF00)                          = 17525016
/1:     semtimedop(7, 0xFFFFFD7FFFDFD9F8, 1, 0xFFFFFD7FFFDFDA00) = 0
/1:     lwp_unpark(4)                                      = 0
/4:     lwp_park(0x00000000, 0)                            = 0
/4:     fcntl(260, F_GETFL)                                = 8258
/4:     pwrite(260, "01 "\0\0 bB101\0EA\0\0\0"., 512, 0x0362C400) = 512
/4:     kaio(AIONOTIFY)                                    = 0
/1:     kaio(AIOWAIT, 0xFFFFFD7FFFDFD800, 0)               = 1
/1:     semctl(7, 55, SETVAL, 1)                           = 0
/4:     lwp_park(0x00000000, 0)          (sleeping...)
/1:     semtimedop(7, 0xFFFFFD7FFFDFD9F8, 1, 0xFFFFFD7FFFDFDA00) (sleeping...)
```

这里显示的堆栈内容包含了几个过程，其中 LWP 代表轻量级进程 (Light Weight process)，指由内核支持的用户线程，是基于内核线程的高级抽象，只有先支持内核线程，才能有 LWP 。每一个进程有一个或多个 LWPs ，每个 LWP 由一个内核线程支持。

LWP PARK 和 UNPARK 是一对内核调用，前者可以中断 LWP 进程进入等待，后者唤醒和重用之前的 LWP 进程.

接下来的 fcntl 为 文件控制（file control),260 为日志文件,260 为文件描述符编号，在 LGWR 进程的 fd 目录下，可以找到这个文件：

solaris*orcl-/home/oracle$ cd /proc/2104/fd

solaris*orcl-/proc/2104/fd$ ls -l 260

-rw-r----- 1 oracle dba 838861312 6 月 29 日 16:22 260

简单的通过 strings 可以看到这个 800M 左右的文件正是日志文件，其序号为 234，

solaris*orcl-/proc/2104/fd$ strings 260|more

}|{z

}wMORCL

Thread 0001, Seq# 0000000234, SCN 0x000001b0191d-0xffffffffffff

在数据库中可以找到这个日志的详细信息：

```
SQL> select * from v$log where SEQUENCE#=234;

    GROUP#    THREAD#  SEQUENCE#      BYTES  BLOCKSIZE    MEMBERS ARC STATUS           FIRST_CHANGE# FIRST_TIM NEXT_CHANGE#
---------- ---------- ---------- ---------- ---------- ---------- --- ---------------- ------------- --------- ------------
         4          1        234  838860800        512          1 NO  CURRENT               28317981 29-JUN-12   2.8147E+14
```

pwrite 为执行写出，写日志文件，该函数为标准的系统调用，在指定文件特定偏移量开始写入指定数据：

ssize_t pwrite(int fildes, const void *buf, size_t nbyte,

off_t offset);

在以上输出中，写出 512 Bytes 数据，也就是 1 个日志块。

接下来的写出以 Kernal AIO 方式执行。然后通过 semctl 进行信号量置位，其中 7 号为 Oracle 数据库的信号量集，通过 ipcs 可以看到正是这一输出：

```
solaris*orcl-/home/oracle$ ipcs -sa
IPC status from <running system> as of 2012年 06月 29日 星期五 19时 07分 34秒 CST
T         ID      KEY        MODE        OWNER    GROUP  CREATOR   CGROUP NSEMS   OTIME    CTIME
Semaphores:
s          7   0x5c23a1bc --ra-r-----   oracle      dba   oracle      dba   204 19:07:32 15:31:55
s          3   0x8d579290 --ra-r-----     grid      dba     grid      dba   204 18:19:31 15:19:10
```

至此一个日志写操作完成，在这整个过程中，进程处于 Log File Sync 等待，前后各包含一次 semtimedop 的信号量操作。

所以我们可以看出：

1.如果磁盘性能有限，则 Log File Sync 等待则可能时间超长，所以通常 RAID10 是最好的日志磁盘组合；

2.这个等待是累计值，也就是说，当仅有的一个 LGWR 执行写出时，可能有多个进程处于同样的等待，如果有 100 个进程等待，则 1 毫秒的 LGWR 写，会引致 100 毫秒的等待，所以这个事件有时并不一定如显示的那样影响深重；

只有深入理解这一过程，才能在具体案例分析时做出正确的判断。

7.12　日志的状态

可以通过 V$LOG 视图来查看日志文件的状态：

```
SQL> select group#,status,first_change# from v$log;
 GROUP# STATUS       FIRST_CHANGE#
------- -------- -----------------
      1 INACTIVE     8903469794507
      2 CURRENT      8903469794526
      3 INACTIVE     8903469794518
      4 INACTIVE     8903469794521
```

最常见的几种日志状态有：

1．CURRENT

指的是当前的日志文件，该日志文件是活动的,当前正在被使用的，在进行崩溃恢复时 Current 的日志文件是必须的。

2．ACTIVE

ACTIVE 的日志是活动的非当前日志，该日志可能已经完成归档也可能没有归档，活动的日志文件在 Crash 恢复时会被用到。

Active 状态意味着，检查点尚未完成，如果日志文件循环使用再次到达该文件，数据库将处于等待的停顿状态，此时在 alert 文件中，我们可以看到类似如下记录：

```
Fri Nov 18 14:26:57 2005
Thread 1 cannot allocate new log, sequence 7239
Checkpoint not complete
  Current log# 5 seq# 7238 mem# 0: /opt/oracle/oradata/hsmkt/redo05.log
```

当这种问题出现时，我们可以从数据库内部通过 V$SESSION_WAIT 来观察，该视图会显示数据库当前哪些 Session 正处于这种等待。Checkpoint not complete 在数据库中体现为等待事件 log file switch (checkpoint incomplete)：

```
SQL> select sid,event,state from v$session_wait;
       SID EVENT                                    STATE
----------  -------------------------------       ----------------
         1 pmon timer                                WAITING
         3 rdbms ipc message                         WAITING
         4 rdbms ipc message                         WAITING
         6 rdbms ipc message                         WAITING
         8 rdbms ipc message                         WAITING
         7 rdbms ipc message                         WAITING
        10 log file switch (checkpoint incomplete)    WAITING
         2 db file parallel write                     WAITED KNOWN TIME
         5 smon timer                                 WAITING
         9 SQL*Net message to client                  WAITED KNOWN TIME
```

```
10 rows selected.
```

同时注意到 DBWR 进程（sid=2）正在进行 db file parallel write,日志文件必须等待 DBWR 完成检查点触发的写操作之后才能被覆盖。如果设置了参数 log_checkpoints_to_alert 为 True 的话，还可以在 alert 文件中清晰的看到检查点的增进和完成情况：

```
Sat Mar 18 21:47:15 2006
Thread 1 advanced to log sequence 292
  Current log# 4 seq# 292 mem# 0: /opt/oracle/oradata/conner/redo04.log
Thread 1 cannot allocate new log, sequence 293
Checkpoint not complete
  Current log# 4 seq# 292 mem# 0: /opt/oracle/oradata/conner/redo04.log
Sat Mar 18 21:47:19 2006
Completed checkpoint up to RBA [0x123.2.10], SCN: 0x0819.0032c424
Completed checkpoint up to RBA [0x122.2.10], SCN: 0x0819.0032c3e9
Sat Mar 18 21:47:20 2006
Beginning log switch checkpoint up to RBA [0x125.2.10], SCN: 0x0819.0032eda9
Thread 1 advanced to log sequence 293
  Current log# 2 seq# 293 mem# 0: /opt/oracle/oradata/conner/redo02.log
```

向上检查 alert 文件，可以找到这些检查点的触发时间：

```
Sat Mar 18 21:47:12 2006
Beginning log switch checkpoint up to RBA [0x122.2.10], SCN: 0x0819.0032c3e9
Sat Mar 18 21:47:12 2006
ARC0: Media recovery disabled
Sat Mar 18 21:47:12 2006
Thread 1 advanced to log sequence 290
  Current log# 1 seq# 290 mem# 0: /opt/oracle/oradata/conner/redo01.log
Beginning log switch checkpoint up to RBA [0x123.2.10], SCN: 0x0819.0032c424
Sat Mar 18 21:47:13 2006
```

通过这些对比和观察，可以使我们更好的了解 Oracle 的运行机制。

我们可以对这个问题做一下简单的分析：

Checkpoint incomplete 有多种可能原因：

1. 日志文件过小，切换过于频繁
2. 日志组太少，不能满足正常事务量的需要
3. 日志文件所在磁盘 I/O 存在瓶颈，导致写出缓慢，阻塞数据库正常运行
4. 由于数据文件磁盘 I/O 瓶颈，DBWR 写出过于缓慢
5. 由于事务量具大，DBWR 负荷过高，不堪重负

……

针对不同的原因，我们可以从不同角度着手解决问题：

1．适当增加日志文件大小

2．适当增加日志组数

3．使用更快速磁盘存储日志文件（如采用更高转速磁盘；使用 Raid10 而不是 Raid5 等方式）

4．改善磁盘 I/O 性能

5．使用多个 DBWR 进程或使用异步 I/O 等

……..

总之，只要我们能够发现数据库的问题所在，就能够从各个角度分析问题并寻找恰当的解决方法。

我们必须知道的是，这是一类严重的等待，它意味着数据库不能再产生日志，所有数据库修改操作将全部挂起。

3．INACTIVE

非活动日志，该日志在实例恢复时不再需要，但是在介质恢复时可能会用到。INACTIVE 状态的日志也可能没有被归档。

如果数据库启动在归档模式,在未完成归档之前,日志文件也不允许被覆盖,这时候,活动进程会处于 log file switch (archiving needed) 等待之中。

日志是否完成归档,可以根据 V$LOG.ARCHIVED 字段进行判断,以下案例日志文件 ARCHIVED 状态为 NO,也就是尚未归档:

```
SQL> archive log list;
Database log mode              Archive Mode
Automatic archival             Enabled
Archive destination            /opt/oracle/oradata/conner/archive
Oldest online log sequence     308
Next log sequence to archive   308
Current log sequence           311
SQL> select * from v$log;

GROUP# THREAD#  SEQUENCE#     BYTES MEMBERS ARCHIVED  STATUS    FIRST_CHANGE# FIRST_TIM
-----  ------ ---------- ---------- ------- --------- --------- ------------- ---------
    1       1        310   10485760       1 NO        INACTIVE     8.9035E+12 26-MAR-06
    2       1        309   10485760       1 NO        INACTIVE     8.9035E+12 19-MAR-06
    3       1        311   10485760       1 NO        CURRENT      8.9035E+12 26-MAR-06
    4       1        308    1048576       1 NO        INACTIVE     8.9035E+12 19-MAR-06
```

注意到此时所以日志组都没有完成归档，所有 DML 事务都将挂起,用户处于 log file switch (archiving needed)等待:

```
SQL> select sid,event from v$session_wait;
       SID EVENT
```

```
---------- ----------------------------------------------------------------
         1 pmon timer
         2 rdbms ipc message
         3 rdbms ipc message
         6 rdbms ipc message
         7 rdbms ipc message
        10 rdbms ipc message
         4 rdbms ipc message
         5 log file switch (archiving needed)
         9 log file switch (archiving needed)
        13 SQL*Net message to client
10 rows selected.
```

这种情况,需要 DBA 介入进行紧急处理,普通用户将无法连接:

```
SQL> connect eygle/eygle
ERROR:
ORA-00257: archiver error. Connect internal only, until freed.
```

这种情况通常是由数据库异常引起的,可能是因为I/O缓慢,也可能是因为事务量过大,在特殊情况下,有可能是因为日志损坏。对于本案例就属于后者，通常我们可以通过检查告警日志文件(alert_<sid>.log)发现问题所在:

```
Sun Apr  9 17:42:11 2006
Errors in file /opt/oracle/admin/conner/bdump/conner_arc0_4475.trc:
ORA-16038: log 4 sequence# 308 cannot be archived
ORA-00354: corrupt redo log block header
ORA-00312: online log 4 thread 1: '/opt/oracle/oradata/conner/redo04.log'
Sun Apr  9 17:42:11 2006
ARC1: Evaluating archive   log 4 thread 1 sequence 308
ARC1: Beginning to archive log 4 thread 1 sequence 308
Creating              archive              destination              LOG_ARCHIVE_DEST_1:
'/opt/oracle/oradata/conner/archive/1_308.dbf'
ARC1: Log corruption near block 2 change 0 time ?
ARC1: All Archive destinations made inactive due to error 354
```

注意到在以上日志中,数据库提示日志损坏(Log Corruption),这就需要 DBA 进行判断并进行恢复等操作了。

4. UNUSED

指该日志从未被写入，这类日志可能是刚被添加到数据库或者在 RESETLOGS 之后被重置。被使用之后，该状态会被改变。

7.13　日志的块大小

初始化参数 LOG_BUFFER 决定了 Redo Log Buffer 的大小，虽然 LOG_BUFFER 中的 Redo Entries 的大小是以 bytes 为单位，但是 LGWR 仍然以 Block 为单位把 redo 写入磁盘，redo block size 是 Oracle 源代码中固定的，与操作系统相关。

通常的操作系统都是以 512 bytes 为单位，如:Solaris, AIX, Windows NT/2000, Linux 等。这个 Log size 可以从 Oracle 的内部视图中获得:

```
SQL> select max(lebsz) from x$kccle;
MAX(LEBSZ)
----------
       512
```

也可以从 v$sysstat 中的统计信息中通过计算粗略得到，以下几个统计信息如:

redo size----------redo 信息的大小

redo wastage---------浪费的 redo 的大小

redo blocks written--LGWR 写出的 redo block 的数量

额外的信息，每个 redo block header 需要占用 16 bytes，由此可以粗略的计算 redo block size 如下：

```
SQL> select name,value from v$sysstat
  2  where name in ('redo size','redo wastage','redo blocks written');
NAME                                                                  VALUE
---------------------------------------------------------------- ----------
redo size                                                            2242628
redo wastage                                                           63904
redo blocks written                                                     4657
SQL> select ceil(16 + (2242628 + 63904)/4657) rbsize from dual;
    RBSIZE
----------
       512
```

在 Linux/Unix 下，Oracle 还提供另外一个命令行工具可以用于检查文件的 block size 大小：

```
[oracle@jumper conner]$ dbfsize redo01.log
Database file: redo01.log
Database file type: file system
Database file size: 20480 512 byte blocks
[oracle@jumper conner]$ dbfsize system01.dbf
Database file: system01.dbf
Database file type: file system
Database file size: 51200 8192 byte blocks
[oracle@jumper conner]$ which dbfsize
~/product/9.2.0/bin/dbfsize
```

从以上的输出中可以看到，日志文件的 Block Size 是 512 bytes，而数据文件的 Block Size 为 8192 bytes。

当然,也可以通过转储日志文件的方式来获取日志文件块大小，转储日志文件头可以通过如下命令实现:

ALTER SESSION SET EVENTS 'immediate trace name redohdr level 10';

查看跟踪文件可以得到类似如下信息，输出中的 bsz 就是指 Redo Blog Size 为 512 Bytes：

```
LOG FILE #1:
 (name #3) /opt/oracle/oradata/conner/redo01.log
Thread 1 redo log links: forward: 2 backward: 0
siz: 0x5000 seq: 0x0000011e hws: 0x1 bsz: 512 nab: 0xffffffff flg: 0x8 dup: 1
Archive links: fwrd: 0 back: 0 Prev scn: 0x0819.002784de
Low scn: 0x0819.00310b39 03/15/2006 21:39:05
Next scn: 0xffff.ffffffff 01/01/1988 00:00:00
```

有时候当数据库出现归档错误时，也会给出提示信息，告知 Blocksize=512，以下信息来自 Oracle10gR2 环境：

```
Fri Aug  1 18:20:58 2008
Errors in file /opt/oracle/admin/xcorder/bdump/xcorder_arc0_6350.trc:
ORA-19502: write error on file "/data1/oradata/XCORDER/archive/1_30816_592917188.dbf",
blockno 24577 (blocksize=512)
ORA-27072: File I/O error
Linux Error: 2: No such file or directory
Additional information: 4
Additional information: 24577
Additional information: 1023488
ORA-19502: write error on file "/data1/oradata/XCORDER/archive/1_30816_592917188.dbf",
blockno 24577 (blocksize=512)
```

提示: 本文提供的多种方法,旨在开阔大家的视野以及思路,对 Oracle 了解的越多,我们学习研究起来就能够更加得心应手.

7.14 日志文件的大小

前面我们提到，当日志文件发生切换时（Log Switch），会触发一个检查点，那么日志文件的大小就和检查点的触发频率相关。更为频繁的检查点可以缩短数据库的恢复时间，但是过于频繁的检查点却会带来性能负担。所以如何合理的设置日志文件的大小也是数据库优化的一个重要内容。

而且必须考虑到的是，如果日志文件意外损坏会丢失，那么我们会丢失数据，所以更大的日志文件可能意味着更多的数据损失风险。数据库的任何一个调整都是需要慎重的。下面让

我们从 Oracle 在不同版本的变化中，揣摩一下 Oracle 的心思。

在 Oracle8i 之中，缺省的 Redo Log File 大小是 1M：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
SQL> select group#,thread#,sequence#,bytes/1024/1024 "M bytes" from v$log;
    GROUP#    THREAD#  SEQUENCE#     M bytes
----------    -------- ----------    ----------
         1          1        364         1
         2          1        365         1
         3          1        363         1
```

而在 Oracle9iR2 中，这个缺省值更改为 100M：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
SQL> select group#,thread#,sequence#,bytes/1024/1024 "M bytes" from v$log;
    GROUP#    THREAD#  SEQUENCE#    M bytes
---------- ---------- ---------- ----------
         1          1        130        100
         2          1        128        100
         3          1        129        100
```

而在 Oracle10gR1 中，这个 Redo Log File 的缺省值再次改变为 10M 大小：

```
SYS AS SYSDBA>select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - 64bi
SYS AS SYSDBA> select group#,thread#,sequence#,bytes/1024/1024 "M bytes" from v$log;
    GROUP#    THREAD#  SEQUENCE#    M bytes
---------- ---------- ---------- ----------
         1          1        173         10
         2          1        174         10
         3          1        172         10
```

通过观察注意到，即使是 Oracle 公司，在对于日志的设置上，也是在不断调整，在大小、切换、恢复时间、数据损失等等问题上，Oracle 也在试图找到一个平衡点。

但是显然，通过 Oracle 的缺省设置来满足所有用户的需求是不现实的，我们在优化调整过程中，实际上需要考虑的更多。一般来说，我们在生产环境中，把 Log Switch 的时间控制在半小时左右即可；需要知道的是，对于通常的操作系统来说日志文件的最大大小为 2G，在

非常繁忙的业务系统中，受限于日志大小，可能将日志控制在 10 分钟左右也已经不错。总之，理解了原理和影响之后，调整、优化都只是一个选择而已。

> 注意：根据经验和调查表明，绝大部分的数据库实际上的确是在缺省设置下运行，这导致很多设置不合理和资源浪费。

7.15 如何调整日志文件大小

很多时候我们需要调整日志文件的大小，可以通过如下的步骤进行调整。

首先查看一下当前日志文件的信息:

```
SQL> select * from v$log;
GROUP# THREAD#  SEQUENCE#   BYTES  MEMBERS ARC STATUS   FIRST_CHANGE# FIRST_TIM
------ ------- ---------- --------- -------- --- ---------------------- ---------
     1       1         79  10485760        1 YES INACTIVE       3848742 19-SEP-06
     2       1         78  10485760        1 YES INACTIVE       3848629 19-SEP-06
     3       1         80  10485760        1 NO  CURRENT      3848837 19-SEP-06
SQL> select * from v$logfile;
    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- ----------------------------------------
         2         ONLINE  /opt/oracle/oradata/eygle/redo02.log
         3         ONLINE  /opt/oracle/oradata/eygle/redo03.log
         1         ONLINE  /opt/oracle/oradata/eygle/redo01.log
```

可以使用如下命令增加新的日志组，在创建新的日志组时，可以定义期望的日志大小：

```
SQL> alter database add logfile group 4  '/opt/oracle/oradata/eygle/redo04.dbf' size 1M;
Database altered.
SQL> alter database add logfile group 5 '/opt/oracle/oradata/eygle/redo05.dbf' size 1M;
Database altered.
```

查看此时的日志组信息：

```
SQL> select * from v$log;
 GROUP#  THREAD# SEQUENCE#    BYTES    MEMBERS ARC STATUS     FIRST_CHANGE#
---------- ---------- ---------- ---------- ---------- --- ---------- -------------
         1          1         79   10485760          1 YES INACTIVE         3848742
         2          1         78   10485760          1 YES INACTIVE         3848629
         3          1         80   10485760          1 NO  CURRENT          3848837
         4          1          0    1048576          1 YES UNUSED                 0
         5          1          0    1048576          1 YES UNUSED                 0
```

我们可以强制切换日志，使数据库使用新创建的日志组：

```
SQL> alter system switch logfile;
System altered.
```

```
SQL> alter system switch logfile;
System altered.
SQL> select * from v$log;

GROUP#  THREAD#  SEQUENCE#  BYTES    MEMBERS ARC STATUS   FIRST_CHANGE#
------ ---------- ---------- ---------- ---------- --- -------- -------------
     1          1         79   10485760          1 YES INACTIVE       3848742
     2          1         78   10485760          1 YES INACTIVE       3848629
     3          1         80   10485760          1 NO  INACTIVE       3848837
     4          1         81    1048576          1 NO  INACTIVE       4374446
     5          1         82    1048576          1 NO  CURRENT        4374469
```

然后可以将当前 STATUS 为 INACTIVE 的日志组删除，保留新创建的日志组：

```
SQL> alter database drop logfile group 1;
Database altered.
SQL> alter database drop logfile group 2;
Database altered.
```

注意如果在归档模式下，INACTIVE 的日志组尚未完成归档，那么日志组不能被删除：

```
SQL> alter database drop logfile group 3;
alter database drop logfile group 3
*
ERROR at line 1:
ORA-00350: log 3 of thread 1 needs to be archived
ORA-00312: online log 3 thread 1: '/opt/oracle/oradata/eygle/redo03.log'
```

可以等待系统归档完成，如果系统出现问题，可以通过手工归档：

```
SQL> alter system archive log sequence 80;
System altered.
SQL> select * from v$log;
GROUP# THREAD#  SEQUENCE#      BYTES  MEMBERS ARC STATUS   FIRST_CHANGE#
------ ------- ---------- ---------- -------- --- -------- -------------
     3       1         80   10485760        1 YES INACTIVE       3848837
     4       1         81    1048576        1 NO  INACTIVE       4374446
     5       1         82    1048576        1 NO  CURRENT        4374469
```

此时可以删除日志组：

```
SQL> alter database drop logfile group 3;
Database altered.
```

最后一步的清理，有时候需要手工去删除操作系统上的日志文件，以释放存储空间：

```
SQL> ! rm /opt/oracle/oradata/eygle/redo02.log
```

当然，如果需要使用原有的日志组号，日志文件可以被重新初始化使用：

```
SQL> alter database add logfile group 3
  2  '/opt/oracle/oradata/eygle/redo03.log' size 1m reuse;
Database altered.
SQL> select * from v$log;
GROUP# THREAD#  SEQUENCE# BYTES MEMBERS ARC STATUS    FIRST_CHANGE#
------ ------- ---------- ---------- ------- --- --------- -------------
     3       1          0    1048576       1 YES UNUSED                0
     4       1         81    1048576       1 NO  INACTIVE        4374446
     5       1         82    1048576       1 NO  CURRENT         4374469
```

7.16 为什么热备份期间产生的 redo 要比正常得多

我们还要知道的是，在数据库处于热备份（使用 Begin Backup 进行备份时）状态时，会产生了比平常更多的日志。

这是因为在热备份期间，Oracle 为了解决 SPLIT BLOCK 的问题，需要在日志文件中记录修改的行所在的**数据块的前镜像**（image），而不仅仅是修改信息。为了理解这段话，我们还需要简单介绍一下 SPLIT BLOCK 的概念：

我们知道，oracle 的数据块是由多个操作系统块组成。通常 Unix 文件系统使用 512bytes 的数据块，而 oracle 使用 8k 的 db_block_size。 当热备份数据文件的时候，我们使用文件系统的命令工具（cp ）拷贝文件，并且使用文件系统的 blocksize 读取数据文件。

在这种情况下，可能出现如下状况：

当我们拷贝数据文件的同时，数据库正好向数据文件写数据。这就使得拷贝的文件中包含这样的 database block，它的一部分 OS block 来自于数据库向数据文件(这个 db block)写操作之前，另一部分来自于写操作之后。对于数据库来说，这个 database block 本身并不一致，而是一个分裂块（SPLIT BLOCK）。

这样的分裂块在恢复时并不可用(会提示 corrupted block)。

所以，在热备状态下，对于变更的数据，Oracle 需要在日志中记录整个变化的数据块的前镜像。这样如果在恢复的过程中，数据文件中出现分裂块，Oracle 就可以通过日志文件中的数据块的前镜像覆盖备份，以完成恢复。

我们看一下测试，首先通过 SYS 用户连接数据库，确认 SCOTT 用户连接信息及日志信息：

```
SQL> alter system switch logfile;
System altered.
SQL> select * from v$log;
GROUP# THREAD#  SEQUENCE#      BYTES MEMBERS ARC STATUS   FIRST_CHANGE# FIRST_TIM
------ ------- ---------- ---------- ------- --- -------- ------------- ---------
     1       1        133  104857600       1 NO  CURRENT      260777420 19-MAR-06
```

```
     2       1        131  104857600       1 YES INACTIVE     260776886 19-MAR-06
     3       1        132  104857600       1 YES ACTIVE       260777060 19-MAR-06
SQL> select sid,serial#,username from v$session;
       SID    SERIAL# USERNAME
---------- ---------- ------------------------------
     ……….
        12         28 SCOTT
12 rows selected.
```

然后我们看一下正常情况下的日志生成：

```
SQL> select * from redo_size;
     VALUE
----------
     11972
SQL> update emp set sal=10 where empno=7788;
1 row updated.
SQL> commit;
Commit complete.
SQL> select * from redo_size;
     VALUE
----------
     12484
SQL> select 12484 -11972 from dual;
12484-11972
-----------
        512
```

正常的更新操作，大约生成了 512 byte 的日志。然后我们用 SYS 用户，将 EMP 表所在的 SYSTEM 表空间置于热备份模式：

```
SQL> alter tablespace system begin backup;
Tablespace altered.
```

使用 Scott 用户进行同样操作：

```
SQL> select * from redo_size;
     VALUE
----------
      8788
SQL> update emp set sal=10 where empno=7788;
1 row updated.
SQL> commit;
Commit complete.
```

```
SQL> select * from redo_size;
     VALUE
----------
     17516
SQL> select 17516 - 8788 from dual;
17516-8788
----------
      8728
```

我们注意到这一次，生成了 8728 byte 的重做，这个日志量较上次正常模式下大约多出了一个 Block 的数量。然后可以用 SYS 用户转储日志文件，看一下多出来的日志到底是什么内容：

```
SQL> @gettrcname
TRACE_FILE_NAME
------------------------------------------------------------------------------
/opt/oracle9/admin/testora9/udump/testora9_ora_4692.trc
SQL> ALTER SYSTEM DUMP LOGFILE '/opt/oracle9/oradata/testora9/redo01.log';
System altered.
```

检查相关信息,我们注意到较常规状态下，增加了”Log block image redo entry”部分，这一部分就是在热备份时产生的额外日志信息：

```
REDO RECORD - Thread:1 RBA: 0x000085.00000020.0010 LEN: 0x2018 VLD: 0x01
SCN: 0x0000.0f8b260c SUBSCN:  1 03/19/2006 20:24:14
CHANGE #1 TYP:3 CLS: 1 AFN:1 DBA:0x0040a482 SCN:0x0000.0f8b2471 SEQ:  1 OP:18.1
Log block image redo entry
Dump of memory from 0x0000000103DB7020 to 0x0000000103DB9008
103DB7020 01080015 000070F3 0F8B2470 00004000  [......p...$p..@.]
103DB7030 1F020300 00000000 00020001 00009A58  [...............X]
103DB7040 0080091B 06D40300 80000000 0F8B23CF  [..............#.]
103DB7050 00090009 0000967D 008002D2 03222300  [.......}......"#.]
103DB7060 20010000 0F8B2471 00010010 FFFF0032  [ .....$q.......2]
103DB7070 1D311CFF 1CFF0000 00101F7A 1F4F1F24  [.1.........z.O.$]
103DB7080 1EFB1ECE 1EA51E7C 1E541E2E 1E031DDD  [.......|.T......]
103DB7090 1DB71D90 1D691D4D 1D310EA4 0E380E04  [.....i.M.1...8..]
103DB70A0 0DD00D7C 0D280CBC 0C880C54 0C000BAC  [...|.(.....T....]
103DB70B0 0B400B0C 0AD80A84 0A3009C4 0990095C  [.@.......0.....\]
°°°°°°°
°°°°°°°
103DB8FE0 C11F2C00 0803C24A 4605534D 49544805  [..,....JF.SMITH.]
103DB8FF0 434C4552 4B03C250 030777B4 0C110101  [CLERK..P..w.....]
103DB9000 0102C209 FF02C115                    [........]
```

```
Dump of memory from 0x0000000103DB9008 to 0x0000000103DB9009
103DB9000                    06401F3A                    [.@.:]

REDO RECORD - Thread:1 RBA: 0x000085.00000030.0128 LEN: 0x01b0 VLD: 0x01
SCN: 0x0000.0f8b260c SUBSCN:  1 03/19/2006 20:24:14
CHANGE #1 TYP:0 CLS:33 AFN:2 DBA:0x00800089 SCN:0x0000.0f8b25c9 SEQ:  1 OP:5.2
ktudh redo: slt: 0x002c sqn: 0x0000967c flg: 0x0012 siz: 132 fbi: 0
            uba: 0x008002d2.0322.3d    pxid:  0x0000.000.00000000
CHANGE #2 TYP:0 CLS:34 AFN:2 DBA:0x008002d2 SCN:0x0000.0f8b25c8 SEQ:  1 OP:5.1
ktudb redo: siz: 132 spc: 3066 flg: 0x0012 seq: 0x0322 rec: 0x3d
            xid:  0x0009.02c.0000967c
ktubl redo: slt: 44 rci: 0 opc: 11.1 objn: 28915 objd: 28915 tsn: 0
Undo type:  Regular undo        Begin trans    Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
             0x00000000  prev ctl uba: 0x008002d2.0322.3c
prev ctl max cmt scn:  0x0000.0f8b2105  prev tx cmt scn:  0x0000.0f8b2126
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
op: L  itl: xid:  0x0002.001.00009a58 uba: 0x0080091b.06d4.03
                      flg: C---    lkc:  0     scn: 0x0000.0f8b23cf
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0040a482  hdba: 0x0040a481
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 7(0x7) flag: 0x2c lock: 0 ckix: 0
ncol: 8 nnew: 1 size: 0
col  5: [ 2]  c1 0b
CHANGE #3 TYP:2 CLS: 1 AFN:1 DBA:0x0040a482 SCN:0x0000.0f8b2471 SEQ:  1 OP:11.5
KTB Redo
op: 0x11  ver: 0x01
op: F  xid:  0x0009.02c.0000967c    uba: 0x008002d2.0322.3d
Block cleanout record, scn:  0x0000.0f8b260c ver: 0x01 opt: 0x02, entries follow...
  itli: 2  flg: 2  scn: 0x0000.0f8b2471
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x0040a482  hdba: 0x0040a481
itli: 1  ispac: 0  maxfr: 4863
tabn: 0 slot: 7(0x7) flag: 0x2c lock: 1 ckix: 0
ncol: 8 nnew: 1 size: 0
```

```
col  5: [ 2]  c1 0b
CHANGE #4 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ:  0 OP:5.20
session number   = 12
serial  number   = 28
transaction name =
```

在 Oracle 数据库内部，存在一个隐含参数控制这个行为：

```
SQL> @GetParDescrb.sql
Enter value for par: blocks
old   6:    AND x.ksppinm LIKE '%&par%'
new   6:    AND x.ksppinm LIKE '%blocks%'
NAME                           VALUE            DESCRIB
------------------------------ ---------------- -------------------------------------
_log_blocks_during_backup      TRUE             log block images when changed during backup
```

这个参数缺省值为 TRUE，设置在热备份期间允许在 redo 中记录数据块信息，如果数据库块大小等于操作系统块大小，则可以设置该参数为 False，减少热备期间数据库的负担（这种情况极为少见）。

分裂块产生的根本原因在于备份过程中引入了操作系统工具（如 cp 工具等），操作系统工具无法保证 Oracle 数据块的一致性。如果使用 RMAN 备份，由于 Rman 可以通过反复读取获得一致的 Blok，从而可以避免 SPLIT Block 的生成，所以不会产生额外的 REDO。

所以我们建议，在备份时（特别是繁忙的数据库），应该尽量采用 RMAN 备份。

7.17 能否不生成 redo

正常的数据库必须生成 Redo，这是数据库的机制，否则数据库在遇到故障或 Crash 时则无法恢复。但是 Oracle 为了增强某些特殊操作的性能，对于一些 SQL 语句，Oracle 允许使用 NOLOGGING 子句，NOLOGGING 可以使得日志生成大幅降低，但是必要日志（比如对于字典表的修改）仍然会被记录。

可以使用 NOLOGGING 的环境非常有限，在以下操作中，可以增加 NOLOGGING 子句：

1. 创建索引或重建索引时
2. 通过/*+ APPEND */提示，使用直接路径（Direct Path）批量 INSERT 操作或 SQL*Loader 直接路径加载数据。
3. CTAS 方式创建数据表时
4. 大对象（LOB）的操作
5. 一些 ALTER TABLE 操作，例如 MOVE，SPLIT 等

NOLOGGING 和表模式（LOGGING/NOLOGING），插入模式（APPEND/NO APPEND）及数据库运行模式（归档/非归档）都有关系。具体可以归纳如下图所示：

数据库模式	表模式	插入模式	REDO 生成
ARCHIVE LOG	LOGGING	APPEND	有 REDO
		NO APPEND	有 REDO
	NOLOGGING	APPEND	无 REDO
		NO APPEND	有 REDO
NOARCHIVE LOG	LOGGING	APPEND	无 REDO
		NO APPEND	有 REDO
	NO LOGGING	APPEND	无 REDO
		NO APPEND	有 REDO

由于大多数生产数据库运行在归档模式下，所以让我们以归档模式为例，简要介绍一下 NOLOGGING 对于 REDO 生成的影响。

首先创建一个视图用于方便 REDO 的查询：

```
CREATE OR REPLACE VIEW redo_size
AS
   SELECT VALUE
     FROM v$mystat, v$statname
    WHERE v$mystat.statistic# = v$statname.statistic#
      AND v$statname.NAME = 'redo size'
```

通过如下步骤将数据库启动在归档模式：

```
SQL> shutdown immediate
SQL> startup mount
SQL> alter database archivelog;
SQL> alter database open;
```

测试对于常规表，REDO 的生成，我们可以看到在归档模式下，APPEND 操作对于常规表是无效的：

```
SQL> create table test as select * from dba_objects where 1=0;
Table created.
SQL> select table_name,logging
  2  from dba_tables where table_name='TEST';
TABLE_NAME       LOGGING
---------------- --------
TEST             YES
SQL> select * from redo_size;
     VALUE
----------
     56288
SQL> insert into test select * from dba_objects;
10470 rows created.
SQL> select * from redo_size;
```

```
     VALUE
----------
   1143948
SQL> insert /*+ append */ into test select * from dba_objects;
10470 rows created.
SQL> select * from redo_size;
     VALUE
----------
   2227712
SQL> select (2227712 -1143948) redo_append,(1143948 -56288) redo from dual;
REDO_APPEND       REDO
----------- ----------
    1083764    1087660
SQL> drop table test;
Table dropped.
```

再来看对于 NOLOGGING 表的 APPEND 操作：

```
SQL> create table test nologging as select * from dba_objects where 1=0;
Table created.
SQL> select table_name,logging
  2  from dba_tables where table_name='TEST';
TABLE_NAME       LOGGING
------------------------------ --------
TEST               NO
SQL> select * from redo_size;
     VALUE
----------
   2270284
SQL> insert into test select * from dba_objects;
10470 rows created.
SQL> select * from redo_size;
     VALUE
----------
   3357644
SQL> insert /*+ append */ into test select * from dba_objects;
10470 rows created.
SQL> select * from redo_size;
     VALUE
----------
   3359024
```

```
SQL> select (3359024 -3357644) redo_append,(3357644 - 2270284) redo from dual;
REDO_APPEND    REDO
-----------    ----------
       1380    1087360
SQL> drop table test;
Table dropped.
```

但是需要注意的是，由于 NOLOGGING 操作会导致对于数据的操作不记录日志，如果数据库崩溃，这部分数据是无法恢复的，所以通常的建议是，在进行了 NOLOGGING 操作之后，需要对数据库进行备份，以避免数据因数据库失效而丢失。

让我们通过一个简短的测试来看一下 NOLOGGING 对于数据恢复的影响。本测试的环境为:

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

首先对表空间 eygle 进行热备份：

```
SQL> alter tablespace eygle begin backup;
Tablespace altered.
SQL> ! cp /opt/oracle/oradata/conner/eygle01.dbf /opt/oracle/oradata/conner/eygle01.dbf.bak
SQL> alter tablespace eygle end backup;
Tablespace altered.
```

在 eygle 表空间创建 NOLOGGING 测试表并 APPEND 追加测试数据：

```
SQL> connect eygle/eygle
Connected.
SQL> create table test nologging as select * from dba_objects where 1=0;
Table created.
SQL> insert /*+ append */ into test select * from dba_objects;
6338 rows created.
SQL> commit;
Commit complete.
```

移除 eygle 表空间的数据文件，模拟故障：

```
SQL> connect / as sysdba
Connected.
SQL> alter tablespace eygle offline;
Tablespace altered.
SQL> ! mv /opt/oracle/oradata/conner/eygle01.dbf /opt/oracle/oradata/conner/eygle01.dbf.del
```

恢复先前热备份文件并进行恢复：

```
SQL> ! cp /opt/oracle/oradata/conner/eygle01.dbf.bak /opt/oracle/oradata/conner/eygle01.dbf
```

```
SQL> alter tablespace eygle online;
alter tablespace eygle online
*
ERROR at line 1:
ORA-01113: file 3 needs media recovery
ORA-01110: data file 3: '/opt/oracle/oradata/conner/eygle01.dbf'
SQL> recover tablespace eygle;
Media recovery complete.
SQL> alter tablespace eygle online;
Tablespace altered.
```

查询该表，发现数据库出现错误：

```
SQL> select count(*) from eygle.test;
select count(*) from eygle.test
                          *
ERROR at line 1:
ORA-01578: ORACLE data block corrupted (file # 3, block # 9098)
ORA-01110: data file 3: '/opt/oracle/oradata/conner/eygle01.dbf'
ORA-26040: Data block was loaded using the NOLOGGING option
```

由此我们可见 NOLOGGING 对于数据库的影响。而如果我们在 NOLOGGING 之后立即对数据库进行过备份，那么这些数据已经写出到数据文件上，自然就是可以恢复的，那么这个恢复过程可能类似：

```
SQL> alter tablespace eygle offline;
Tablespace altered.
SQL> ! cp /opt/oracle/oradata/conner/eygle01.dbf.del /opt/oracle/oradata/conner/eygle01.dbf
SQL> recover tablespace eygle;
Media recovery complete.
SQL> alter tablespace eygle online;
Tablespace altered.
SQL> select count(*) from eygle.test;
  COUNT(*)
----------
      6338
```

当我们使用 DataGuard 作为数据库的备份或容灾高可用性手段时，通常日志就变得不可缺少。在 Oracle9iR2 中，我们可以将数据库置于强制日志模式（FORCE LOGGING MODE）。

在强制日志模式下，所有操作都将记录日志：

```
SQL> SELECT FORCE_LOGGING FROM V$DATABASE;
FOR
```

```
---
NO
SQL> ALTER DATABASE FORCE LOGGING;
Database altered.
SQL> SELECT FORCE_LOGGING FROM V$DATABASE;
FOR
---
YES
```

7.18　Redo 故障的恢复

我们已经知道日志文件对于数据库来说非常重要，在实际使用过程中，可能会遇到各种各样的问题。

接下来我们介绍一些在日常数据库维护中经常会遇到的情况。

7.18.1　丢失非活动日志组的故障恢复

如果数据库丢失的是非活动（INACTIVE）日志组，由于非活动日志组已经完成检查点，数据库不会发生数据损失，此时只需要通过 Clear 重建该日志组即可恢复。

首先删除一个非活动日志组，模拟一次故障损失：

```
SQL> ! rm /opt/oracle/oradata/eygle/redo02.log
```

如果数据库日志切换，使用到该日志组，则数据库可能马上崩溃：

```
SQL> alter system switch logfile;
alter system switch logfile
*
ERROR at line 1:
ORA-03113: end-of-file on communication channel
```

我们可以从警告日志中获得部分详细信息：

```
Wed May 10 11:06:49 2006
Errors in file /opt/oracle/admin/eygle/bdump/eygle_lgwr_31539.trc:
ORA-00313: open failed for members of log group 2 of thread 1
ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/eygle/redo02.log'
ORA-27037: unable to obtain file status
Linux Error: 2: No such file or directory
Additional information: 3
```

此时启动数据库，数据库会提示日志丢失：

```
SQL> startup
ORACLE instance started.

Total System Global Area  252777592 bytes
Fixed Size                   451704 bytes
Variable Size             134217728 bytes
Database Buffers          117440512 bytes
Redo Buffers                 667648 bytes
Database mounted.
ORA-00313: open failed for members of log group 2 of thread 1
ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/eygle/redo02.log'
```

此时在 Mount 状态我们可以查看各日志组及日志文件的状态：

```
SQL> select * from v$log;
GROUP#  THREAD#  SEQUENCE#  BYTES  MEMBERS ARC STATUS       FIRST_CHANGE#
------ ---------- --------- ---------- ---------- --- ---------------- -------------
     1          1         8   10485760          1 NO  INACTIVE               485702
     2          1         0   10485760          1 NO  UNUSED
0
     3          1         9   10485760          1 NO  INVALIDATED            485719
SQL> select * from v$logfile;
    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- ----------------------------------------
         1         ONLINE  /opt/oracle/oradata/eygle/redo01.log
         2         ONLINE  /opt/oracle/oradata/eygle/redo02.log
         3 STALE   ONLINE  /opt/oracle/oradata/eygle/redo03.log
```

我们注意到，由于日志组 2 已经损失，在日志切换过程中，数据库 Crash，所以日志组 3 的状态变为 INVALIDATED，日志文件 redo03.log 的状态变为 STALE（STALE 通常出现在上一次操作失败之后，在下一次成功操作后状态会恢复正常）。

我们清除该日志组后即可启动数据库：

```
SQL> alter database clear logfile group 2;
Database altered.
SQL> alter database open;
Database altered.
```

注意，如果数据库处于归档模式下，并且该日志组未完成归档则需要使用如下命令强制清除（关于此种情况，请参考本章后面相关诊断案例）：

```
alter database clear unarchived logfile group 2;
```

打开数据库之后，状态为 STALE 的日志文件，在下次正常写入后，状态即可恢复正常：

```
SQL> select * from v$log;
   GROUP#  THREAD# SEQUENCE#    BYTES    MEMBERS ARC  STATUS           FIRST_CHANGE#
```

```
------ ---------- --------- ---------- ---------- --- ---------------- -------------
     1          1         8   10485760          1 NO  INACTIVE               485702
     2          1        10   10485760          1 NO  CURRENT                505721
     3          1         9   10485760          1 NO  INACTIVE               485719
SQL> select * from v$logfile;
GROUP# STATUS  TYPE    MEMBER
------ ------- ------- ----------------------------------------
     1         ONLINE  /opt/oracle/oradata/eygle/redo01.log
     2         ONLINE  /opt/oracle/oradata/eygle/redo02.log
     3 STALE   ONLINE  /opt/oracle/oradata/eygle/redo03.log
SQL> alter system switch logfile;
System altered.
SQL> alter system switch logfile;
System altered.
SQL> select * from v$logfile;
GROUP# STATUS  TYPE    MEMBER
------ ------- ------- ----------------------------------------
     1         ONLINE  /opt/oracle/oradata/eygle/redo01.log
     2         ONLINE  /opt/oracle/oradata/eygle/redo02.log
     3         ONLINE  /opt/oracle/oradata/eygle/redo03.log
```

7.18.2 丢失活动或当前日志文件的恢复

我们知道 Oracle 通过日志文件保证提交成功的数据不丢失。可是我们注意，在故障中，我们可能损失了当前的（CURRENT）日志文件。

这又分为两种情况：

1．如果数据库是正常关闭的

由于关闭数据库前，Oracle 会执行全面检查点，当前日志在实例恢复中可以不再需要。在 Oracle8i 中我们可以通过与 4.9.1 中描述类似方法进行解决，收录简要测试步骤如下（非归档模式下测试过程）：

```
SQL> select * from v$version where rownum <2;
BANNER
----------------------------------------------------------------
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
SQL> select * from v$logfile;
    GROUP# STATUS  MEMBER
---------- ------- ----------------------------------------
         1         /opt/oracle/oradata/testora8/redo01.log
         2         /opt/oracle/oradata/testora8/redo02.log
         3         /opt/oracle/oradata/testora8/redo03.log
```

```
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> ! mv /opt/oracle/oradata/testora8/redo* /tmp
SQL> startup
ORACLE instance started.

Total System Global Area  338390716 bytes
Fixed Size                   102076 bytes
Variable Size             133308416 bytes
Database Buffers          204800000 bytes
Redo Buffers                 180224 bytes
Database mounted.
ORA-00313: open failed for members of log group 1 of thread 1
ORA-00312: online log 1 thread 1: '/opt/oracle/oradata/testora8/redo01.log'

SQL> alter database clear logfile group 1;
Database altered.
SQL> alter database clear logfile group 2;
Database altered.
SQL> alter database clear logfile group 3;
Database altered.
SQL> alter database open;
Database altered.
```

在 Oracle9i 中，可能无法对当前日志进行 Clear，需要通过 Until Cancel 恢复后，Resetlogs 打开，以下是一个简单的测试过程：

```
SQL> ! rm /opt/oracle/oradata/eygle/redo0*
SQL> startup
ORACLE instance started.

Total System Global Area  252777592 bytes
Fixed Size                   451704 bytes
Variable Size             134217728 bytes
Database Buffers          117440512 bytes
Redo Buffers                 667648 bytes
Database mounted.
ORA-00313: open failed for members of log group 1 of thread 1
ORA-00312: online log 1 thread 1: '/opt/oracle/oradata/eygle/redo01.log'
```

```
SQL> alter database clear logfile group 1;
Database altered.
SQL> alter database clear unarchived logfile group 2;
alter database clear unarchived logfile group 2
*
ERROR at line 1:
ORA-00313: open failed for members of log group 2 of thread 1
ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/eygle/redo02.log'
ORA-27037: unable to obtain file status
Linux Error: 2: No such file or directory
Additional information: 3
SQL> recover database until cancel;
Media recovery complete.
SQL> alter database open resetlogs;
Database altered.
```

2．在损失当前日志时，如果数据库是异常关闭的

那么 Oracle 在进行实例恢复时必须要求当前日志，否则 Oracle 将无法保证提交成功的数据不丢失（也就意味着，Oracle 会丢失数据），在这种情况下，Oracle 数据库将无法启动。

对于这种情况，我们通常需要从备份中恢复数据文件，通过应用归档日志文件向前推演，直到最后一个完好的日志文件，然后可以通过 resetlogs 启动数据库完成恢复。丢失的数据就是损坏的日志文件中的数据。

这种恢复方式与以上介绍的方法类似，不再赘述。

可是不幸的是，很多数据库是从不备份的，那么在面对这种情况时，Oracle 提供给我们一种内部手段可以用于强制性数据库打开，忽略一致性等问题，在打开数据库之后，Oracle 建议导出（exp）数据，然后重建数据库，再导入（imp）数据，完成灾难恢复。

在继续之前，我愿意提及一下我经常强调的 DBA 四大守则之一：

备份重于一切

要知道系统总是要崩溃的，没有有效的备份只是等哪一天死而已。

如果说有什么事可以让 DBA 在深夜惊醒的话，那就是“没有备份”。

如果回忆一下，我们在第三章曾经提到过 Oracle 有一类具有特殊作用的隐含参数，其中一个参数是：_allow_resetlogs_corruption。

我们看一下这个参数的说明：

```
SQL> select ksppinm,ksppdesc from x$ksppi
  2  where ksppinm like '%resetlogs_%';
KSPPINM                        KSPPDESC
```

```
------------------------------ ----------------------------------------------------
_allow_resetlogs_corruption    allow resetlogs even if it will cause corruption
```

该参数的含义是，允许在破坏一致性的情况下强制重置日志，打开数据库。_ALLOW_RESETLOGS_CORRUPTION 将使用所有数据文件最旧的 SCN 打开数据库，所以通常需要保证 SYSTEM 表空间拥有最旧的 SCN。

在强制打开数据库之后，可能因为各种原因伴随出现 ORA-600 错误，有些可以依据常规途径解决。我们看一下下面的一个案例。

在数据库启动时出现错误，提示日志文件损坏：

```
SQL> startup ;
ORACLE instance started.

Total System Global Area   97588504 bytes
Fixed Size                   451864 bytes
Variable Size              33554432 bytes
Database Buffers           62914560 bytes
Redo Buffers                 667648 bytes
Database mounted.
ORA-00354: corrupt redo log block header
ORA-00353: log corruption near block 3 change 897612314 time 10/19/2005 14:19:34
ORA-00312: online log 3 thread 1: '/opt/oracle/oradata/conner/redo03.log'
```

在 Mount 状态，可以查询 v$log 视图，发现此处损坏的是 active 的日志文件:

```
SQL> select * from v$log;
GROUP#THREAD#  SEQUENCE#     BYTES    MEMBERS ARC STATUS FIRST_CHANGE# FIRST_TIM
----- ------- ---------- ---------- ---------- --- ------ ------------- ---------
    1       1        159   10485760        1 NO  INACTIVE   897592312 19-OCT-05
    2       1        158   10485760        1 NO  INACTIVE   897572310 19-OCT-05
    3       1        160   10485760        1 NO  ACTIVE     897612314 19-OCT-05
    4       1        161    1048576        1 NO  CURRENT    897612440 19-OCT-05
```

由于 Active 日志未完成检查点，在恢复中需要用到，丢失 Active 日志和 Current 日志情况类似，如果没有备份，我们只好使用隐含参数_allow_resetlogs_corruption 强制启动数据库，设置此参数之后，在数据库 Open 过程中，Oracle 会跳过某些一致性检查，从而使数据库可能跳过不一致状态，Open 打开:

```
SQL> alter system set "_allow_resetlogs_corruption"=true scope=spfile;
System altered.
SQL> shutdown immediate;
ORA-01109: database not open

Database dismounted.
ORACLE instance shut down.
```

```
SQL> startup mount;
SQL> recover database using backup controlfile until cancel;
ORA-00279: change 897612315 generated at 10/19/2005 16:54:18 needed for thread 1
ORA-00289: suggestion : /opt/oracle/oradata/conner/archive/1_160.dbf
ORA-00280: change 897612315 for thread 1 is in sequence #160

Specify log: {=suggested | filename | AUTO | CANCEL}
cancel
ORA-01547: warning: RECOVER succeeded but OPEN RESETLOGS would get error below
ORA-01194: file 1 needs more recovery to be consistent
ORA-01110: data file 1: '/opt/oracle/oradata/conner/system01.dbf'

ORA-01112: media recovery not started
SQL> alter database open resetlogs;
Database altered.
SQL> shutdown immediate;
SQL> startup
ORACLE instance started.

Total System Global Area   97588504 bytes
Fixed Size                   451864 bytes
Variable Size              33554432 bytes
Database Buffers           62914560 bytes
Redo Buffers                 667648 bytes
Database mounted.
Database opened.
```

幸运的时候数据库就可以成功 Open，如果不幸可能会遇到一系列的 Ora-600 错误（最常见的是 2662 错误）此时就需要使用多种手段继续进行调整恢复。

如果注意观察 alert 日志，我们可能会发现类似以下日志:

```
Fri Jun 10 16:30:25 2005
alter database open resetlogs
Fri Jun 10 16:30:25 2005
RESETLOGS is being done without consistancy checks. This may result
in a corrupted database. The database should be recreated.
RESETLOGS after incomplete recovery UNTIL CHANGE 240677200
Resetting resetlogs activation ID 3171937922 (0xbd0fee82)
```

不一致恢复最后恢复到的 Change 号是：240677200

Oracle 告诉我们，强制 resetlogs 跳过了一致性检查，可能导致数据库损坏，数据库应当重

建。而且此方法应该在 Oracle 技术支持的指导之下进行，否则 Oracle 将不对采用此类方式进行恢复的数据库进行支持。

注意：**DBA 需要时刻铭记的一个工作习惯是，在重要操作或故障处理前，保留现场**。也就是说在进行以上类似恢复等工作前，我们应当对数据库进行冷备份，这样在恢复尝试失败后，也仍然可以回退到之前的状态。

7.19 通过 Clear 日志恢复数据库故障一例

收到此次故障时，同事给出的描述是数据库不能归档，做过基本的检查，磁盘空间不存在问题，磁盘状态也都是正常的，并附上部分警报日志文件内容：

```
Wed May 17 10:43:42 2006
ARC1: Evaluating archive   log 1 thread 1 sequence 202
ARC1: Archiving not possible: No primary destinations
ARC1: Failed to archive log 1 thread 1 sequence 202
Wed May 17 10:43:42 2006
Errors in file /oracle/admin/jshs/bdump/jshs_arc1_17874.trc:
ORA-16014: log 1 sequence# 202 not archived, no available destinations
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
```

从日志来看，的确是数据库不能归档，并且提示归档路径错误。

登陆数据库进行检查，首先查询参数设置及归档路径状态：

```
SQL> select dest_id,dest_name,status from v$archive_dest;
   DEST_ID DEST_NAME            STATUS
---------- -------------------- ---------
         1 LOG_ARCHIVE_DEST_1   ERROR
         2 LOG_ARCHIVE_DEST_2   INACTIVE
         3 LOG_ARCHIVE_DEST_3   INACTIVE
         4 LOG_ARCHIVE_DEST_4   INACTIVE
         5 LOG_ARCHIVE_DEST_5   INACTIVE
         6 LOG_ARCHIVE_DEST_6   INACTIVE
         7 LOG_ARCHIVE_DEST_7   INACTIVE
         8 LOG_ARCHIVE_DEST_8   INACTIVE
         9 LOG_ARCHIVE_DEST_9   INACTIVE
        10 LOG_ARCHIVE_DEST_10  INACTIVE
已选择 10 行。
SQL> show parameter log_archive_dest
NAME                                 TYPE        VALUE
```

```
---------------------------------- ----------- ---------------------------------
log_archive_dest                   string
log_archive_dest_1                 string      LOCATION=/u04/oradata/jshs/archive
.....
log_archive_dest_state_1           string      enable
```

发现当前归档路径的状态的确是错误（error）的。检查警报日志文件，找到第一次出现错误的部分：

```
Wed May 17 10:32:31 2006
Errors in file /oracle/admin/jshs/bdump/jshs_arc1_17874.trc:
ORA-00354: corrupt redo log block header
ORA-00353: log corruption near block 92256 change 0 time 05/17/2006 01:57:27
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
ARC1: Archiving not possible: error count exceeded
ARC1: Failed to archive log 1 thread 1 sequence 202
ARCH: Archival stopped, error occurred. Will continue retrying
Wed May 17 10:32:31 2006
ORACLE Instance jshs - Archival Error
ARCH: Connecting to console port...
Wed May 17 10:32:31 2006
ORA-16038: log 1 sequence# 202 cannot be archived
ORA-00354: corrupt redo log block header
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
ARCH: Connecting to console port...
ARCH:
Wed May 17 10:32:31 2006
ORA-16038: log 1 sequence# 202 cannot be archived
ORA-00354: corrupt redo log block header
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
Wed May 17 10:32:31 2006
Errors in file /oracle/admin/jshs/bdump/jshs_arc1_17874.trc:
ORA-16038: log 1 sequence# 202 cannot be archived
ORA-00354: corrupt redo log block header
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
```

我们注意，这里发现了问题的根本原因，归档失败的原因在于日志损坏。检查跟踪文件 jshs_arc1_17874.trc ，由于多次归档不能成功，导致数据库将归档路径标记为 Error，使得后续正常的日志同样无法归档：

```
*** 2006-05-17 10:32:31.621
kcrrfail: dest:1 err:354 force:0
ORA-00354: corrupt redo log block header
```

```
ORA-00353: log corruption near block 92256 change 0 time 05/17/2006 01:57:27
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
*** 2006-05-17 10:32:31.662
ARC1: Archiving not possible: error count exceeded
ORA-16038: log 1 sequence# 202 cannot be archived
ORA-00354: corrupt redo log block header
ORA-00312: online log 1 thread 1: '/u01/oradata/jshs/redo01.log'
ORA-16014: log 1 sequence# 202 not archived, no available destinations
```

查询数据库：

```
SQL> select * from v$logfile;
    GROUP# STATUS  TYPE    MEMBER
---------- ------- ------- ----------------------------------------
         1         ONLINE  /u01/oradata/jshs/redo01.log
         2         ONLINE  /u01/oradata/jshs/redo02.log
         3         ONLINE  /u01/oradata/jshs/redo03.log
SQL> select * from v$log;
    GROUP#    THREAD#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- ---------- --- ----------------
         1          1        202  104857600          1 NO  INACTIVE
         2          1        313  104857600          1 NO  CURRENT
         3          1        312  104857600          1 YES INACTIVE
```

我们看到在其他人进行的多次重起切换过程中，日志组 2 和组 3 的 SEQUENCE#都已经增进，只有日志组 1 的 SEQUENCE#仍然是 202。

由于日志组 1 并非 Current 日志组，所以我们可以通过 Clear 方式清除该日志内容，从而使该日志恢复正常状态：

```
SQL> alter database clear unarchived logfile group 1;
数据库已更改。
SQL> select * from v$log;
    GROUP#    THREAD#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- ---------- --- ----------------
         1          1          0  104857600          1 YES UNUSED
         2          1        313  104857600          1 YES INACTIVE
         3          1        314  104857600          1 NO  CURRENT
```

注意，由于该日志未归档，所以之前的热备份用于恢复时将不能跨越这个缺口，Oracle 建议重新进行全库备份，从警告日志中也可以看到如下提示：

```
Wed May 17 11:17:32 2006
alter database clear unarchived logfile group 1
Wed May 17 11:17:35 2006
WARNING! CLEARING REDO LOG WHICH HAS NOT BEEN ARCHIVED. BACKUPS TAKEN
```

```
    BEFORE 05/17/2006 01:58:01 (CHANGE 338217516) CANNOT BE USED FOR RECOVERY.
Clearing online log 1 of thread 1 sequence number 202
Completed: alter database clear unarchived logfile group 1
Wed May 17 11:18:11 2006
Archiver process freed from errors. No longer stopped
```

并且注意到归档进程从错误中被释放出来，数据库恢复了正常。我们手工进行日志切换：

```
SQL> alter system switch logfile;
系统已更改。
SQL> /
系统已更改。
```

再检查日志归档情况，确认日志组 1 已经被成功归档：

```
SQL> select * from v$log;
    GROUP#    THREAD#  SEQUENCE#      BYTES    MEMBERS ARC STATUS
---------- ---------- ---------- ---------- ---------- --- ----------------
         1          1        315  104857600          1 YES ACTIVE
         2          1        316  104857600          1 NO  CURRENT
         3          1        314  104857600          1 YES INACTIVE
```

检查归档路径的状态，发现已经恢复正常：

```
SQL> select dest_name,status from v$archive_dest where rownum <2;
 DEST_NAME                      STATUS
------------------------------ ---------
LOG_ARCHIVE_DEST_1             VALID
```

至此问题解决完毕，后续的工作是需要对数据库进行备份。

7.19 日志组过度激活的诊断案例

这是一个和 Redo 相关的诊断案例：

```
平台:SunOS 5.8 Generic_108528-23 sun4u sparc SUNW,Ultra-Enterprise
数据库:8.1.5.0.0
症状:响应缓慢，应用请求已经无法返回
```

这时候登陆数据库检查,发现 redo 日志组除 current 外都处于 active 状态：

```
SQL> select * from v$log;
 GROUP# THREAD#  SEQUENCE#      BYTES    MEMBERS ARC STATUS  FIRST_CHANGE# FIRST_TIM
------- ------- ---------- ---------- ---------- --- ---------------------- ---------
      1       1     520403   31457280          1 NO  ACTIVE     1.3861E+10 23-JUN-05
      2       1     520404   31457280          1 NO  ACTIVE     1.3861E+10 23-JUN-05
      3       1     520405   31457280          1 NO  ACTIVE     1.3861E+10 23-JUN-05
```

```
     4       1     520406   31457280          1 NO  CURRENT    1.3861E+10 23-JUN-05
     5       1     520398   31457280          1 NO  ACTIVE     1.3860E+10 23-JUN-05
     6       1     520399   31457280          1 NO  ACTIVE     1.3860E+10 23-JUN-05
     7       1     520400  104857600          1 NO  ACTIVE     1.3860E+10 23-JUN-05
     8       1     520401  104857600          1 NO  ACTIVE     1.3860E+10 23-JUN-05
     9       1     520402  104857600          1 NO  ACTIVE     1.3861E+10 23-JUN-05
9 rows selected.
```

如果日志都处于 active 状态，那么显然 DBWR 的写已经无法跟上 Log Switch 触发的检查点。接下来让我们检查一下 DBWR 的繁忙程度:

```
oracle:/oracle/oracle8>ps -ef|grep ora_dbw
  oracle  2266     1  0   Mar 31 ?       811:42 ora_dbw0_hysms02
  oracle 21023 21012  0 18:52:59 pts/65   0:00 grep ora_dbw
```

DBWR 的进程号是 2266。使用 Top 命令观察一下:

```
oracle:/oracle/oracle8>top


last pid: 21145;  load averages:  3.38,  3.45,  3.67               18:53:38
725 processes: 711 sleeping, 1 running, 10 zombie, 3 on cpu
CPU states: 35.2% idle, 40.1% user,  9.4% kernel, 15.4% iowait,  0.0% swap
Memory: 3072M real, 286M free, 3120M swap in use, 1146M swap free

   PID USERNAME THR PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
 11855 smspf      1  59    0 1355M 1321M cpu/0   19:32 17.52% oracle
  2264 oracle     1   0    0 1358M 1316M run    283.3H 17.36% oracle
 11280 oracle     1  13    0 1356M 1321M sleep   79.8H  0.77% oracle
  6957 smspf     15  29   10   63M   14M sleep  107.7H  0.76% java
 17393 smspf      1  30    0 1356M 1322M cpu/1  833:05  0.58% oracle
 29299 smspf      5  58    0 8688K 5088K sleep   18.5H  0.38% fee_ftp_get
 21043 oracle     1  43    0 3264K 2056K cpu/9    0:01  0.31% top
  8086 smspf      5  23    0   21M   13M sleep   41.1H  0.24% fee_file_in
 16009 root       1  35    0 4920K 3160K sleep    0:03  0.21% sshd2
 25126 smspf      1  58    0 1355M 1321M sleep    0:26  0.20% oracle
  2266 oracle     1  60    0 1357M 1317M sleep  811:42  0.18% oracle
 11628 smspf      7  59    0 3440K 2088K sleep    0:39  0.16% sgip_client_ltz
 26257 smspf     82  59    0  447M  178M sleep  533:04  0.15% java
```

我们注意到，2266 号进程消耗的 CPU 不过 0.18%,显然并不繁忙,那么瓶颈就很可能在 IO 上。使用 IOSTAT 工具检查 IO 状况。

```
gqgai:/home/gqgai>iostat -xn 3
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
```

```
.....
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t6d0
    1.8   38.4   32.4  281.0  0.0  0.7    0.0   17.4   0  29 c0t10d0
    1.8   38.4   32.4  281.0  0.0  0.5    0.0   13.5   0  27 c0t11d0
   24.8   61.3 1432.4  880.1  0.0  0.5    0.0    5.4   0  26 c1t1d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    9.1   0   0 hurraysms02:vold(pid238)
                  extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
.....
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t6d0
    0.3    8.3    0.3   47.0  0.0  0.1    0.0    9.2   0   8 c0t10d0
    0.0    8.3    0.0   47.0  0.0  0.1    0.0    8.0   0   7 c0t11d0
   11.7   65.3  197.2  522.2  0.0  1.6    0.0   20.5   0 100 c1t1d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 hurraysms02:vold(pid238)
                  extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
.....
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t6d0
    0.3   13.7    2.7   68.2  0.0  0.2    0.0   10.9   0  12 c0t10d0
    0.0   13.7    0.0   68.2  0.0  0.1    0.0    9.6   0  11 c0t11d0
   11.3   65.3   90.7  522.7  0.0  1.5    0.0   19.5   0  99 c1t1d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 hurraysms02:vold(pid238)
                  extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
.....
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t6d0
    0.0    8.0    0.0   42.7  0.0  0.1    0.0    9.3   0   7 c0t10d0
    0.0    8.0    0.0   42.7  0.0  0.1    0.0    9.1   0   7 c0t11d0
   11.0   65.7  978.7  525.3  0.0  1.4    0.0   17.7   0  99 c1t1d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 hurraysms02:vold(pid238)
                  extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
.....
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 c0t6d0
    0.3   87.7    2.7  433.7  0.0  2.2    0.0   24.9   0  90 c0t10d0
    0.0   88.3    0.0  437.5  0.0  1.8    0.0   19.9   0  81 c0t11d0
   89.0   54.0  725.4  432.0  0.0  2.1    0.0   14.8   0 100 c1t1d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    0.0   0   0 hurraysms02:vold(pid238)
```

我们注意到，存放数据库的主要卷 c1t1d0 的繁忙程度始终处于 99~100,而写速度却只有

500K/s 左右，这个速度是极为缓慢的。

(%b percent of time the disk is busy (transactions in progress)

Kw/s kilobytes written per second)

根据我们的常识 T3 盘阵通常按 Char 写速度可以达到 10M/s 左右，而正常情况下的数据库随机写通常都在 1~2M 左右，显然此时的磁盘已经处于不正常状态，经过确认的确是硬盘发生了损坏，Raid5 的 Group 中损坏了一块硬盘，经过更换以后系统逐渐恢复正常。

附录　　数值在 Oracle 的内部存储

Oracle 在数据库内部通过相应的算法转换来进行数据存储,本文简单介绍 Oracle 的 Number 型数值存储及转换。可以通过 DUMP 函数来转换数字的存储形式，一个简单的输出类似如下格式:

```
SQL> select dump(1),dump(1,16) from dual;


DUMP(1)           DUMP(1,16)
----------------- -----------------
Typ=2 Len=2: 193,2 Typ=2 Len=2: c1,2
```

DUMP 函数的输出格式类似:

类型 <[长度]>，符号/指数位 [数字 1，数字 2，数字 3，.....，数字 20]

各位的含义如下:

1. 类型指字段数据类型，Number 型,Type=2 (类型代码可以从 Oracle 的文档上查到)
2. 长度指存储的字节数
3. 符号/指数位用于代表数字的正负及指数值
4. 数据存储

在存储上，Oracle 对正数和负数分别进行存储转换。而 0 既不属于正数，也不属于负数，是单独进行处理的。

```
SQL> select dump(0),dump(0,16) from dual;


DUMP(0)          DUMP(0,16)
---------------- ---------------
Typ=2 Len=1: 128 Typ=2 Len=1: 80
```

在以上输出中，0 以 16 机制 0x80 存储，仅包含了符号位，未包含数据位。为什么用 0x80 存储呢？

注意 0x80 其 2 进制表示为 1000 0000，2 Byte 如果同时用来表示正数和负数，那么 0x80 正好将编码进行了等分，所以 0x80 用来表示 0 数值。

有了 0 值划分之后，符号位大于 80 的就是正数，符号位小于 80 的就是负数。

接下来看看数值的存储，以下是数值 1 的存储输出：

```
SQL> select dump(1),dump(1,16) from dual;


DUMP(1)            DUMP(1,16)
------------------ ------------------
Typ=2 Len=2: 193,2 Typ=2 Len=2: c1,2
```

数值 1 的符号位为 193，表示这是一个正数，而 1 使用的是数值 2 来存储。这是因为在 C 语言中 0 作为字符串终结符被保留，所以数值 1 使用 2 存储，进一步的，所有的正数都是通过“加 1”进行存储表示的。

简单来说，正数以及负数按照如下规则存储：

- 正数：加 1 存储
- 负数：被 101 减

接下来再来看看指数位的作用，根据科学计数法可以知道，任何一个实数 S 都可以描述为 A.B X 10^n 的形式（A 表示整数部分，B 表示小数部分，而 n 表示 10 的指数部分）。当 S 大于 1 时，N 大于等于 0，S 小于 1 时，N 小于 0。

存储中的指数位可以通过如下方式换算出来

- 正数：指数=符号/指数位 - 193 （0xC1）
- 负数：指数=62（0x3E） - 第一字节

去除指数位，实际上从<数字 1>开始才是有效的数据位，根据指数及每一位的数值就可以将存储的数值还原出来，所以 Oracle 存储的数值计算方法为：

∑ <数字位 n> x 100^(指数-N)

注意:这里 N 是有效位数的顺序位，第一个有效位的 N=0

我们看一下以下的示例说明：

```
SQL> select dump(123456.789) from dual;
DUMP(123456.789)
--------------------------------
Typ=2 Len=6: 195,13,35,57,79,91
```

<指数位> =　195 - 193 = 2

<数字 1> =　13 - 1　　= 12 *100^(2-0) =120000

<数字 2> =　35 - 1　　= 34 *100^(2-1) =3400

<数字 3> =　57 - 1　　= 56 *100^(2-2) =56

<数字 4> =　79 - 1　　= 78 *100^(2-3) =.78

<数字 5> =　91 - 1　　= 90 *100^(2-4) =.009

∑　　=　　123456.789

```
SQL> select dump(-123456.789) from dual;
DUMP(-123456.789)
---------------------------------
Typ=2 Len=7: 60,89,67,45,23,11,102
```

<指数位> = 62 - 60 = 2
<数字 1> =101 - 89 = 12 *100^(2-0) =120000
<数字 2> =101 - 67 = 34 *100^(2-1) =3400
<数字 3> =101 - 45 = 56 *100^(2-2) =56
<数字 4> =101 - 23 = 78 *100^(2-3) =.78
<数字 5> =101 - 11 = 90 *100^(2-4) =.009
∑ = 123456.789(-)

注意在负数末尾会加入 102，也就是 16 进制的 0x66 作为标记，这可以被看作符号位，表示负数，同时也是为了排序的需要，-123456.789 在数据库中实际存储为 60,89,67,45,23,11 ，而-123456.78901 在数据库中实际存储为 60,89,67,45,23,11,91 ，可见，如果不在最后加上 102，在排序时会出现-123456.789<-123456.78901 的情况。由于正数的表示范围是 0x01 到 0x64，负数的表示范围是 0x65 到 0x02。因此，不会在表示数字时出现的 0x66。

参考文献：

Oracle 基本数据类型存储格式浅析（二）——数字类型 By Yangtingkun
Memory Management and Latching Improvements in Oracle9i and 10g By Tanel Poder
Alert Log Messages: Private Strand Flush Not Complete Metalink Note: 372557.1
About _log_parallelism_dynamic and _log_parallelism_max Metalink Note: 457967.1