[Oracle](http://www.linuxidc.com/topicnews.aspx?tid=12" \o "Oracle" \t "_blank) 的Redo 机制DB的一个重要机制，理解这个机制对DBA来说也是非常重要，之前的Blog里也林林散散的写了一些，前些日子看老白日记里也有说明，所以结合老白日记里的内容，对oracle 的整个Redo log 机制重新整理一下。

**一．Redo log 说明**

Oracle 的Online redo log 是为确保已经提交的事务不会丢失而建立的一个机制。 因为这种健全的机制，才能让我们在数据库crash时，恢复数据，保证数据不丢失。

**1.1 恢复分类**

恢复分两种：

（1）    Crash recovery

（2）    Media recovery

这两种的具体说明，参考：

Oracle 实例恢复时 前滚（roll forward） 后滚（rollback） 问题  <http://www.linuxidc.com/Linux/2011-03/33907.htm>

这两种的区别是：

（1）    Crash Recovery 是在启动时DB 自动完成，而MediaRecovery 需要DBA 手工的完成。

（2）    Crash Recovery 使用online redo log，Media Recovery 使用archived log 和 online redo log。

（3）    Media Recovery 可能还需要从备份中Restore datafile。

**1.2 Crash Recovery 过程**

       当数据库突然崩溃，而还没有来得及将buffer cache里的脏数据块刷新到数据文件里，同时在实例崩溃时正在运行着的事务被突然中断，则事务为中间状态，也就是既没有提交也没有回滚。这时数据文件里的内容不能体现实例崩溃时的状态。这样关闭的数据库是不一致的。

       下次启动实例时，Oracle会由SMON进程自动进行实例恢复。实例启动时，SMON进程会去检查控制文件中所记录的、每个在线的、可读写的数据文件的END SCN号。

       数据库正常运行过程中，该END SCN号始终为NULL，而当数据库正常关闭时，会进行完全检查点，并将检查点SCN号更新该字段。

       而崩溃时，Oracle还来不及更新该字段，则该字段仍然为NULL。当SMON进程发现该字段为空时，就知道实例在上次没有正常关闭，于是由SMON进程就开始进行实例恢复了。

**1.2.1 前滚**

       SMON进程进行实例恢复时，会从控制文件中获得检查点位置。于是，SMON进程到联机日志文件中，找到该检查点位置，然后从该检查点位置开始往下，应用所有的重做条目，从而在buffer cache里又恢复了实例崩溃那个时间点的状态。这个过程叫做前滚，前滚完毕以后，buffer cache里既有崩溃时已经提交还没有写入数据文件��脏数据块，也还有事务被突然终止，而导致的既没有提交又没有回滚的事务所弄脏的数据块。

**1.2.2 回滚**

       前滚一旦完毕，SMON进程立即打开数据库。但是，这时的数据库中还含有那些中间状态的、既没有提交又没有回滚的脏块，这种脏块是不能存在于数据库中的，因为它们并没有被提交，必须被回滚。打开数据库以后，SMON进程会在后台进行回滚。

       有时，数据库打开以后，SMON进程还没来得及回滚这些中间状态的数据块时，就有用户进程发出读取这些数据块的请求。这时，服务器进程在将这些块返回给用户之前，由服务器进程负责进行回滚，回滚完毕后，将数据块的内容返回给用户。

总之，Crash Recovery时，数据库打开会占用比正常关闭更长的时间。

**1.2.3 必须先前滚，在回滚**

       回滚段实际上也是以回滚表空间的形式存在的，既然是表空间，那么肯定就有对应的数据文件，同时在buffer cache 中就会存在映像块，这一点和其他表空间的数据文件相同。

       当发生DML操作时，既要生成REDO（针对DML操作本身的REDO Entry）也要生成UNDO（用于回滚该DML操作，记录在UNDO表空间中），但是既然UNDO信息也是使用回滚表空间来存放的，那么该DML操作对应的UNDO信息（在BUFFER CACHE生成对应中的UNDO BLOCK）就会首先生成其对应的REDO信息（UNDO BLOCK's REDO Entry）并写入Log Buffer中。

       这样做的原因是因为Buffer Cache中的有关UNDO表空间的块也可能因为数据库故障而丢失，为了保障在下一次启动时能够顺利进行回滚，首先就必须使用REDO日志来恢复UNDO段（实际上是先回复Buffer Cache中的脏数据块，然后由Checkpoint写入UNDO段中），在数据库OPEN以后再使用UNDO信息来进行回滚，达到一致性的目的。

       生成完UNDO BLOCK's REDO Entry后才轮到该DML语句对应的REDO Entry，最后再修改Buffer Cache中的Block，该Block同时变为脏数据块。

       实际上，简单点说REDO的作用就是记录所有的数据库更改，包括UNDO表空间在内。

**1.2.4  Crash Recovery 再细分**

Crash Recovery 可以在细分成两种：

（1）    实例恢复(InstanceRecovery)

（2）    崩溃恢复(CrashRecovery)

InstanceRecovery与CrashRecovery是存在区别的：针对单实例(singleinstance)或者RAC中所有节点全部崩溃后的恢复，我们称之为Crash Recovery。    而对于RAC中的某一个节点失败，存活节点(surviving instance)试图对失败节点线程上redo做应用的情况，我们称之为InstanceRecovery。

       不管是Instance Recovery还是Crash Recovery，都由2个部分组成：cache recovery和transaction recovery。

       根据[官方文档](http://download.oracle.com/docs/cd/B10500_01/server.920/a96533/instreco.htm)的介绍，Cache Recovery也叫Rolling Forward（前滚）；而Transaction Recovery也叫Rolling Back（回滚）。

**1.3 Redo log 说明**

REDO LOG 的数据是按照THREAD 来组织的，对于单实例系统来说，只有一个THREAD，对于RAC 系统来说，可能存在多个THREAD，每个数据库实例拥有一组独立的REDO LOG 文件，拥有独立的LOG BUFFER，某个实例的变化会被独立的记录到一个THREAD 的REDO LOG 文件中。

对于单实例的系统，实例恢复（Instance Recovery）一般是在数据库实例异常故障后数据库重启时进行，当数据库执行了SHUTDOWN ABORT 或者由于操作系统、主机等原因宕机重启后，在ALTER DATABASE OPEN 的时候，就会自动做实例恢复。

在RAC 环境中，如果某个实例宕了，或者实例将会接管，替宕掉的实例做实例恢复。除非是所有的实例都宕了，这样的话，第一个执行ALTER DATABASE OPEN 的实例将会做实例恢复。这也是REDO LOG 是实例私有的组件，但是REDO LOG 文件必须存放在共享存储上的原因。

Oracle 数据库的CACHE 机制是以性能为导向的，CACHE 机制应该最大限度的提高数据库的性能，因此CACHE 被写入数据文件总是尽可能的推迟。这种机制大大提高了数据库的性能，但是当实例出现故障时，可能出现一些问���。

首先是在实例故障时，可能某些事物对数据文件的修改并没有完全写入磁盘，可能磁盘文件中丢失了某些已经提交事务对数据文件的修改信息。

其次是可能某些还没有提交的事务对数据文件的修改已经被写入磁盘文件了。也有可能某个原子变更的部分数据已经被写入文件，而部分数据还没有被写入磁盘文件。

实例恢复就是要通过ONLINE REDO LOG 文件中记录的信息，自动的完成上述数据的修复工作。这个过程是完全自动的，不需要人工干预。

**1.3.1 如何确保已经提交的事务不会丢失？**

解决这个问题比较简单，Oracle 有一个机制，叫做Log-Force-at-Commit，就是说，在事务提交的时候，和这个事务相关的REDO LOG 数据，包括COMMIT 记录，都必须从LOG BUFFER 中写入REDO LOG 文件，此时事务提交成功的信号才能发送给用户进程。通过这个机制，可以确保哪怕这个已经提交的事务中的部分BUFFER CACHE 还没有被写入数据文件，就发生了实例故障，在做实例恢复的时候，也可以通过REDO LOG 的信息，将不一致的数据前滚。

**1.3.2 如何在数据库性能和实例恢复所需要的时间上做出平衡？**

      既确保数据库性能不会下降，又保证实例恢复的快速，解决这个问题，oracle是通过checkpoint 机制来实现的。

Oracle 数据库中，对BUFFER CAHCE 的修改操作是前台进程完成的，但是前台进程只负责将数据块从数据文件中读到BUFFERCACHE 中，不负责BUFFERCACHE 写入数据文件。BUFFERCACHE 写入数据文件的操作是由后台进程DBWR 来完成的。DBWR 可以根据系统的负载情况以及数据块是否被其他进程使用来将一部分数据块回写到数据文件中。这种机制下，某个数据块被写回文件的时间可能具有一定的随机性的，有些先修改的数据块可能比较晚才被写入数据文件。

而CHECKPOINT 机制就是对这个机制的一个有效的补充，CHECKPOINT 发生的时候，CKPT 进程会要求DBWR 进程将某个SCN 以前的所有被修改的块都被写回数据文件。这样一旦这次CHECKPOINT 完成后，这个SCN 前的所有数据变更都已经存盘，如果之后发生了实例故障，那么做实例恢复的时候，只需要冲这次CHECKPOINT 已经完成后的变化量开始就行了，CHECKPOINT 之前的变化就不需要再去考虑了。

**1.3.3 有没有可能数据文件中的变化已经写盘，但是REDO LOG 信息还在LOG BUFFER 中，没有写入REDO LOG 呢?**

      这里引入一个名词：Write-Ahead-Log，就是日志写入优先。日志写入优先包含两方面的算法:

第一个方面是，当某个BUFFER CACHE 的修改的变化矢量还没有写入REDO LOG 文件之前，这个修改后的BUFFER CACHE 的数据不允许被写入数据文件，这样就确保了再数据文件中不可能包含未在REDO LOG 文件中记录的变化；

第二个方面是，当对某个数据的UNDO 信息的变化矢量没有被写入REDOLOG 之前，这个BUFFERCACHE的修改不能被写入数据文件。

**二．LOG BUFFER 和LGWR**

**2.1 Redo Log 说明**

REDO LOG是顺序写的文件，每次写入的数据量很小，TEMP文件虽然也就有部分顺序读写的特点，但是TEMP每次读写的数据量较大，和REDO 的特性不同。UNDO是典型的随机读写的文件，索引更是以单块读为主的操作。

REDO LOG 的产生十分频繁，几乎每秒钟都有几百K 到几M 的RED LOG 产生，甚至某些大型数据库每秒钟产生的REDO LOG 量达到了10M 以上。不过前台进程每次产生的REDO量却不大，一般在几百字节到几K，而一般一个事务产生的REDO 量也不过几K到几十K。

基于REDO 产生的这个特点，如果每次REDO产生后就必须写入REDOLOG 文件，那么就会存在两个问题，一个是REDO LOG 文件写入的频率过高，会导致REDO LOG文件的IO 存在问题，第二个是如果由前台进程来完成REDO LOG 的写入，那么会导致大量并发的前台进程产生REDO LOG 文件的争用。

为了解决这两个问题，Oracle 在REDO LOG 机制中引入了LGWR 后台进程和LOGBUFFER。

LOG BUFFER 是Oracle 用来缓存前台进程产生的REDO LOG 信息的，有了LOG BUFFER，前台进程就可以将产生的REDO LOG 信息写入LOG BUFFER,而不需要直接写入REDO LOG 文件，这样就大大提高了REDO LOG 产生和保存的时间，从而提高数据库在高并发情况下的性能。既然前台进程不将REDOLOG 信息写入REDO LOG 文件了，那么就必须要有一个后台进程来完成这个工作。这个后台进程就是LGWR，LGWR 进程的主要工作就是将LOG BUFFER 中的数据批量写入到REDO LOG 文件中。对于Oracle 数据库中，只要对数据库的改变写入到REDO LOG 文件中了，那么就可以确保相关的事务不会丢失了。

引入LOG BUFFER 后，提高了整个数据库RDMBS 写日志的性能，但是如何确保一个已经提交的事务确确实实���被保存在数据库中，不会因为之后数据库发生故障而丢失呢？实际上在前面两节中我们介绍的REDO LOG 的一些基本的算法确保了这一点。

首先WRITE AHEAD LOG 协议确保了只要保存到REDO LOG 文件中的数据库变化一定能够被重演，不会丢失，也不会产生二义性。其次是在事务提交的时候，会产生一个COMMIT 的CV，这个CV 被写入LOG BUFFER 后，前台进程会发出一个信号，要求LGWR 将和这个事务相关的REDO LOG 信息写入到REDO LOG 文件中，只有这个事务相关的REDO LOG 信息已经确确实实被写入REDO LOG 文件的时候，前台进程才会向客户端发出事务提交成功的消息，这样一个事务才算是被提交完成了。在这个协议下，只要客户端收到了提交完成的消息，那么可以确保，该事务已经存盘，不会丢失了。

LGWR 会绕过操作系统的缓冲，直接写入数据文件中，以确保REDO LOG 的信息不会因为操作系统出现故障（比如宕机）而丢失要求确保写入REDO LOG 文件的数据。

实际上，虽然Oracle 数据库使用了绕过缓冲直接写REDO LOG 文件的方法，以避免操作系统故障导致的数据丢失，不过我们还是无法确保这些数据已经确确实实被写到了物理磁盘上。因为我们RDBMS 使用的绝大多数存储系统都是带有写缓冲的，写缓冲可以有效的提高存储系统写性能，不过也带来了另外的一个问题，就是说一旦存储出现故障，可能会导致REDO LOG 的信息丢失，甚至导致REDO LOG 出现严重损坏。存储故障的概率较小，不过这种小概率事件一旦发生还是会导致一些数据库事务的丢失，因此虽然Oracle 的内部算法可以确保一旦事务提交成功，事务就确认被保存完毕了，不过还是可能出现提交成功的事务丢失的现象。

实际上，Oracle 在设计REDO LOG 文件的时候，已经最大限度的考虑了REDO LOG 文件的安全性，REDO LOG 文件的BLOCK SIZE 和数据库的BLOCK SIZE 是完全不同的，REDO LOG 文件的BLOCK SIZE 是和操作系统的IO BLOCK SZIE 完全相同的，这种设计确保了一个REDO LOG BLOCK 是在一次物理IO 中同时写入的，因此REDOLOG BLOCK 不会出现块断裂的现象。