

深入浅出 Oracle——DBA 入门、 进阶与诊断案例

盖国强 编著

人 民 邮 电 出 版 社



前言

关于本书

根据 Gartner 公司的统计数据,在 2005 年,Oracle 数据库以 48.6% 的市场占有率继续稳居关系数据库市场的首位。在过去这一年中,国内的 Oracle 从业市场和学习环境都有了很大的发展和进步,市场进一步规范和成熟,从事数据库管理工作的朋友们也越来越多。

为了让更多进入 Oracle 领域的朋友能够快速了解和掌握 Oracle 技术,让具有一定经验和积累的 Oracle 从业人员继续深入学习,作者倾力撰写了本书。

本书作者活跃于国内著名 Oracle 技术论坛 ITPUB (www.itpub.net),并全力打造国内极具影响力的个人 Oracle 技术站点 Eygle.com (www.eygle.com)。本书从基础出发,逐层深入,并结合实际工作中的诊断案例进行全面讲解,力图从点到面,让读者对每个主题都有深入的了解和认识。

本书是 ITPUB 技术丛书的第三本,在《Oracle 数据库 DBA 专题技术精粹》和《Oracle 数据库性能优化》两书出版的两年多以来,ITPUB 和 Oracle 市场都有了长足的发展,希望本书的出现能为读者带来更深入的技术知识和更多的实践经验。

本书特点

本书每章的布局基本上分为 3 个部分,基础知识、进阶知识、结合实际的案例分析。基础知识部分可以作为初学者的入门参考,进阶部分则可以给广大 Oracle 技术爱好者作为深入学习的材料,案例分析作为实践部分希望对大家都能有所借鉴。

在数据库版本方面,本书内容更涉及 Oracle 8i/Oracle 9i/Oracle 10g,将 Oracle 的版本变化、功能改进,一以贯之地展现出来,让大家看到这些变革的真正原因以及 Oracle 的不断技术创新。关于 Oracle 技术的很多问题是因为跨越版本而存在的,所以我们必须了解一项技术的来龙去脉,才能知道一个革新、一个新特性的真正意义所在。

本书是作者多年实践工作的积累和总结,各章节更从 DBA 的成长历程入手,引导大家快速进入并深入 Oracle 知识的世界。

本书继续贯彻了作者“由点到线再及面”的学习方法，既可以让初学者参考学习，又可以帮助具备一定基础的中级 DBA 进行进阶学习，不同层次的学习者都能从本书的不同内容中受益。

本书结构

本书分为 9 章，具体结构划分如下。

● 第 1 章：数据库的启动和关闭，从基础入手，讲解 Oracle 数据库的启动和关闭，并深入探讨数据库启动关闭的核心本质及内部处理。

● 第 2 章：参数及参数文件，这一部分从 Oracle 启动必需的参数文件入手，讲解重要参数、参数文件对于 Oracle 的作用，并结合 RAC 环境，Oracle10g 环境介绍参数文件等的不断改进和变迁。

● 第 3 章：数据字典，深入到数据库的核心，全面了解数据字典的机制和重要性。

● 第 4 章：内存管理，Oracle 的内存管理非常重要，本章就 SGA、PGA 的管理进行探讨，并深入介绍 Oracle 内存管理技术在 Oracle 8i/9i/10g 不同版本中的变迁。

● 第 5 章：Buffer Cache 与 Shared Pool 原理，本章深入介绍了 Buffer Cache 和 Shared Pool 的原理，并涉及锁和热点块等深入话题。

● 第 6 章：重做，重做机制是 Oracle 恢复的保障，本章针对 Oracle 的重做机制进行探讨，并涉及重做的内部原理及工作机制。

● 第 7 章：回滚与撤销，回滚和事务密切相关，本章从基础出发，介绍 Oracle 的回滚机制，进而深入研究和探讨回滚机制的内部操作及 ORA-01555 错误等相关知识。

● 第 8 章：等待事件，等待事件在数据库性能诊断中起着极为重要的作用，在不同版本中，Oracle 一直在不断加强等待事件的功能，本章从等待事件入手，进一步讲解数据库性能诊断和优化知识。

● 第 9 章：性能诊断与 SQL 优化，这一章是实践的总结，通过一些实践的案例，介绍一种思路和方法给大家，解决问题是学习的最终目的。

本书的读者对象

本书适用于打算进入 Oracle 领域的初学者，也适用于具备一定数据库基础、打算深入学习 Oracle 技术的数据库从业人员，尤其适用于入门、进阶以及希望深入研究 Oracle 技术的数据库管理人员。

本书也可以作为各大中专院校相关专业的教学辅导和参考用书，或作为相关培训机构的培训教材。

本书约定

(1) 为了给读者提供更多的学习资源，同时弥补本书篇幅有限的遗憾，本书提供了部分的参考链接，许多本书无法详细介绍的问题都可以通过这些链接找到答案。相关文档可以从作者的网站 (www.eygle.com) 上找到。

(2) 本书所列出的插图、运行结果可能会与读者实际环境中的操作界面有所差别，这可能是由于操作系统平台、Oracle 版本的不同而引起的，在此特别说明，一切以实际情况为准。

(3) 广大读者如有好的建议，或在学习本书中遇到疑难问题，欢迎到作者网站 (<http://www.eygle.com>) 进行探讨，也可发电子邮件联系作者 (eygle@eygle.com) 或本书责任编辑 (dujie@ptpress.com)。



写在前面

开始写这本书是在 2006 年 3 月 3 日，这个写作计划已经制订了很久，手头也有了写好的章节，但是关于全书的主题、框架一直没有一个很好的构思，所以拖延了很久。

开始动笔时，打算写一本入门级的书给初学者看，希望通过这本书把自己学习 Oracle 的方法和经验介绍给刚刚接触 Oracle 的朋友们。

可是动起笔来，发现很难把一本书控制在初学者的范畴，所以写起来难度就逐渐深入了下去，不过也好，符合我原来的想法。去年曾经计划写一本《Oracle 诊断案例》的书，一直没有动笔，现在正好把两本书合为一本了。

在本书的写作过程中，我一直希望和努力摆脱传统技术书籍要么基础知识、要么代码实例、要么特定版本的单一模式，将基础知识、深入研究、性能调整、诊断案例等相关内容，按照每章一根主线展开，从而使读者通过每一章的阅读，能够对相关知识有一个纵向的深入认知。

书中贯彻的也是我一直主张的“由点到线再及面”的学习方法。特别是对于初学者，如果没有经过专门的培训和系统学习，那么就更应当自己通过实践和思考深入学习。在知识上，从某个角度来说，是“不患寡，而患不精深”。在我们遇到问题时，就应该不断深入研究，直至问题的核心本质。这样通过一个案例或实际问题的诊断和研究，我们就可以带动很多连带知识的学习，这样从一个点深入下去就形成一条线，再横向扩展就可以形成一个知识网，解决和研究的问题多了，就可以逐渐覆盖一个面，形成一个知识体系。这样慢慢地你就会觉得学习不再困难，而是一件得心应手的事情。

而且，认真思考和深入研究本身就是对 DBA 的一项基本素质要求。在网上经常被问及我的招聘要求，收录一点在这里给大家参考：

对于一个候选人来说，我希望他勤奋、严谨、具有钻研精神及独立思考能力。技术其实往往并不是我最关心的内容，因为具备了前面的素质之后，经过一两年的锻炼，一个人绝对不会知道得太少。

我自己正是通过这样一条学习之路走过来的，也希望可以通过这本书将这个思想传递出来。我愿意将我在《Oracle 数据库性能优化》一书的序言中提到的一段话再次引用一下：

兴趣 + 勤奋 + 坚持 + 方法 \approx 成功

很遗憾我不能给以上公式画上“=”，但是无关紧要，只要具备了以上因素，我想我们每个人都会离成功不远了。

2005 年由我担任主编之一，ITPUB 组织编写、人民邮电出版社出版的《Oracle 数据库性能优化》一书，在 Dearbook 举办的“2005 年最权威的图书评选”活动中排名第 9。这是对我的极大鼓励，也增强了我继续写作的动力。

然而独自写作一本书的压力是可想而知的，所以迟迟不肯动笔写作，一方面是因为构思的原因，一方面是觉得自己的所知和积累仍然有限，直到动笔以后，某一天，忽然觉得豁然开朗，整本书的体系结构和章节组织跃然纸上，于是就有了今天这本书。

这也是这本书先有了部分章节，后有名称的原因之一。

初学者在学习之初，往往因 Oracle 的博大、复杂而困惑，甚至就此止步于 Oracle 入门之前，可是如果能够渡过这个阶段，你可能会发现，一切并没有想像中那么复杂和困难。我希望本书能够陪伴大家走过这段日子，并且能够帮助大家开始深入的学习。

在开始本书的阅读之前，我想在这里介绍一下我所总结的“DBA 生存之四大守则”，这四大守则是我在长期的工作和学习过程中总结出来的，希望大家能够有所借鉴。

1. 备份重于一切

我们必须知道，系统总是要崩溃的，没有有效的备份只是在等哪一天死！我经常开玩笑地说，惟一会使 DBA 在梦中惊醒的就是：没有有效的备份。

在进行重要的操作（如恢复尝试、升级操作等）之前，一定要做好备份，保留现场，以便必要时可以从头再来。

2. 三思而后行

Think thrice before you act:

任何时候都要清楚你所做的一切，否则宁可不做！对于 DBA 来说，有时候一个回车，一条命令就会造成不可恢复的灾难，所以，必需清楚确认你所做的一切，以及这些操作可能带来的后果，并且在必要时保护现场。

DBA 切忌想当然。

3. rm 是危险的

要知道在 UNIX/Linux 下，这个操作意味着你可能将永远失去后面的东西，所以，确认你的操作！

太多的人在“rm -rf”上悲痛欲绝，当年写下这条守则时，是一个凌晨被一个朋友吵醒，他说误操作 rm -rf 删除掉了 200GB 的数据库，而且没有备份。我当时能告诉他的只有一句话：要保持冷静。

4. 你来制定规范

良好的规范是减少故障的基础。所以，作为一个 DBA，你需要来制定规范，以规范开发人员甚至系统人员，这样可以规避有意或是无意的误操作，减少数据库的风险。

见过太多管理混乱的开发环境，经常出现程序员测试时却错连生产环境误操作的案例，所以规范实在是非常之重要。

正所谓：不以规矩，不成方圆。

这四大守则之间是相互关联，密不可分的，希望每个 DBA 都能谨慎认真，少犯错误。

本书越临近出版，作者越是紧张，并且希望能够通过出版前的不断修正使得本书不断趋于完善。正如很多作者都说过的那样，一本书出版之后，就如同一个生命的诞生。这本书即将脱离我的控制，去经历它的历程。面对读者的评判，我能做的就是毫无保留地将自己的所知通过本书传达给读者，至于因作者水平所限而存在错漏之处，那也正是作者应该接受评判和指正之处。写作一本书，也正是一个学习的历程。

最后我要感谢好友冯春培 (biti_rainy)，他帮助我审阅了本书的第 4 章和第 5 章；感谢好友楼方鑫 (dcba)，他帮我审阅了本书的第 6 章和第 7 章；他们为我提出了很多宝贵的建议，使得本书更加趋于完善；当然还要感谢我的女友 Julia，正是因为有了她的鼓励和支持，才有了我持续不断写作的动力，并且最终完成了本书。

本书最后定稿是在 6 月 8 日，那一天正好是我的生日。

盖国强

2006 年 6 月 8 日 于北京

图书在版编目 (CIP) 数据

深入浅出 Oracle——DBA 入门、进阶与诊断案例 / 盖国强著.

—北京: 人民邮电出版社, 2006.7

ISBN 7-115-14989-5

I. 深... II. 盖... III. 关系数据库—数据库管理系统, Oracle IV. TP311.138

中国版本图书馆 CIP 数据核字 (2006) 第 076820 号

内 容 提 要

本书针对数据库的启动和关闭、参数及参数文件、数据字典、内存管理、Buffer Cache 与 Shared Pool 原理、重做、回滚与撤销、等待事件、性能诊断与 SQL 优化等几大 Oracle 热点主题, 从基础知识入手, 深入研究相关技术, 并结合性能调整及丰富的诊断案例, 力图将 Oracle 知识全面、系统、深入地展现给读者。

本书给出了大量取自实际工作现场的实例。在分析实例的过程中, 兼顾深度与广度, 不仅对实际问题的现象、产生原因和相关的原理进行了深入浅出的讲解, 更重要的是, 结合实际应用环境, 提供了一系列解决问题的思路和方法, 包括详细的操作步骤, 具有很强的实战性和可操作性, 满足面向实际应用的读者需求。

深入浅出 Oracle——DBA 入门、进阶与诊断案例

- ◆ 著 盖国强
责任编辑 杜 洁
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京隆昌伟业印刷有限公司印刷
新华书店总店北京发行所经销
- ◆ 开本: 787×1092 1/16
印张: 31.25
字数: 782 千字
印数: 1 - 5 000 册
- 2006 年 7 月第 1 版
2006 年 7 月北京第 1 次印刷

ISBN 7-115-14989-5/TP · 5549

定价: 65.00 元

读者服务热线: (010)67132692 印装质量热线: (010)67129223



目 录

第 1 章 数据库的启动和关闭	1
1.1 数据库的启动	1
1.1.1 启动数据库到 nomount 状态	1
1.1.2 启动数据库到 mount 状态	8
1.1.3 启动数据库 open 阶段	12
1.2 进阶内容	18
1.2.1 SCN	18
1.2.2 检查点	24
1.2.3 正常关闭数据库的状况	34
1.2.4 数据库异常关闭的情况	37
1.3 深入分析	41
1.3.1 获得数据库 Open 的跟踪文件	41
1.3.2 bootstrap\$及数据库初始化过程	42
1.3.3 BOOTSTRAP\$的重要性	44
1.3.4 BBED 工具的简要介绍	45
第 2 章 参数及参数文件	49
2.1 初始化参数的分类	49
2.1.1 推导参数 (Derived Parameters)	49
2.1.2 操作系统依赖参数	49
2.1.3 可变参数	49
2.1.4 初始化参数的获取	50
2.2 参数文件	52
2.2.1 PFILE 和 SPFILE	53
2.2.2 SPFILE 的创建	55

2.2.3	SPFILE 的搜索顺序	56
2.2.4	使用 PFILE/SPFILE 启动数据库	57
2.2.5	修改参数	59
2.2.6	重置 SPFILE 中设置的参数	66
2.2.7	是否使用了 SPFILE	67
2.2.8	SPFILE 的备份与恢复	68
2.2.9	如何设置 Events 事件	75
2.2.10	导出 SPFILE 文件	77
2.3	诊断案例	81
2.3.1	登录系统检查 alert.log 文件	82
2.3.2	尝试重新启动数据库	84
2.3.3	检查数据文件	84
2.3.4	mount 数据库，检查系统参数	85
2.3.5	检查参数文件	86
2.3.6	再次检查 alert 文件	86
2.3.7	修正 PFILE	88
2.3.8	启动数据库	89
第 3 章	数据字典	91
3.1	数据字典概述	91
3.2	内部 RDBMS (X\$) 表	91
3.3	数据字典表	94
3.4	动态性能视图	95
3.4.1	GV\$和 V\$视图	95
3.4.2	GV_、V_视图和 V\$、GV\$同义词	97
3.4.3	数据字典视图	99
3.4.4	进一步的说明	101
3.5	最后的验证	102
3.5.1	V\$PARAMETER 的结构	102
3.5.2	视图还是同义词	103
3.5.3	Oracle 如何通过同义词定位对象	104
第 4 章	内存管理	109
4.1	SGA 管理	109
4.1.1	什么是 SGA	109
4.1.2	SGA 与共享内存	118
4.1.3	SGA 管理的变迁	124
4.2	PGA 管理	138
4.2.1	什么是 PGA	138

4.2.2	PGA 的调整建议	147
4.3	Oracle 的内存分配和使用	151
4.3.1	诊断案例一: SGA 与 Swap	152
4.3.2	诊断案例二: SGA 设置过高导致的系统故障	156
4.3.3	诊断案例三: 如何诊断和解决 CPU 高度消耗 (100%) 问题	161
第 5 章	Buffer Cache 与 Shared Pool 原理	165
5.1	Buffer Cache 原理	165
5.1.1	LRU 与 Dirty List	165
5.1.2	Cache Buffers Lru Chain 闕锁竞争与解决	169
5.1.3	Cache Buffer Chain 闕锁竞争与解决	171
5.2	Shared Pool 的基本原理	186
5.2.1	Shared Pool 的设置说明	187
5.2.2	了解 X\$KSMSP 视图	195
5.2.3	诊断和解决 ORA-04031 错误	199
5.2.4	Library Cache Pin 及 Library Cache Lock 分析	209
5.2.5	诊断案例一: version_count 过高造成的 Latch 竞争解决	216
5.2.6	诊断案例二: 临时表引发的竞争	224
5.2.7	小结	228
第 6 章	重做 (Redo)	229
6.1	Redo 的作用	229
6.2	Redo 的内容	230
6.3	产生多少 Redo	235
6.4	Redo 写的触发条件	239
6.4.1	每 3 秒钟超时 (Timeout)	239
6.4.2	阈值达到	240
6.4.3	用户提交	241
6.4.4	在 DBWn 写之前	242
6.5	Redo Log Buffer 的大小设置	242
6.6	Commit 做了什么	243
6.7	日志的状态	243
6.8	日志的块大小	247
6.9	日志文件的大小	249
6.10	为什么热备份期间产生的 Redo 要比正常的多	251
6.11	能否不生成 Redo	256

6.11.1 NOLOGGING 对于数据库的影响	256
6.11.2 disable_logging 对于数据库的影响	263
6.11.3 FORCE LOGGING (强制日志) 模式	272
6.12 Redo 故障的恢复	272
6.12.1 丢失非活动日志组的故障恢复	273
6.12.2 丢失活动或当前日志文件的恢复	275
6.13 诊断案例一：通过 Clear 日志恢复数据库	281
6.14 诊断案例二：日志组过度激活的诊断	285
第 7 章 回滚与撤销	290
7.1 什么是回滚和撤销	290
7.2 回滚段存储的内容	291
7.3 并发控制和一致性读	292
7.4 回滚段的前世今生	293
7.5 回滚机制的深入研究	298
7.6 Oracle 9i 闪回查询的新特性	317
7.7 使用 ERRORSTACK 进行错误跟踪	320
7.8 Oracle 10g 闪回查询特性的增强	322
7.9 ORA-01555 错误	327
7.10 AUM 下如何重建 Undo 表空间	339
7.11 诊断案例一：使用 Flashback Query 恢复误 删除数据	340
7.12 诊断案例二：释放过度扩展的 Undo 空间	343
7.13 特殊情况的恢复	347
7.14 数值在 Oracle 的内部存储	351
第 8 章 等待事件	354
8.1 等待事件的源起	354
8.2 从等待发现瓶颈	358
8.2.1 V\$SESSION 和 V\$SESSION_WAIT	359
8.2.2 从 V\$SQLTEXT 中追踪	360
8.2.3 捕获相关 SQL	361
8.3 Oracle 10g 的增强	365
8.3.1 新增 V\$SESSION_WAIT_HISTORY 视图	365
8.3.2 ASH 新特性	366
8.3.3 自动负载信息库 AWR 的引入	373
8.3.4 自动数据库诊断监控 ADDM 的引入	375
8.4 顶级等待事件	376
8.5 重要等待事件	380
8.5.1 db file sequential read (数据文件顺序读取)	380

8.5.2	db file scattered read (数据文件离散读取)	382
8.5.3	direct path read/write (直接路径读/写)	386
8.5.4	日志文件相关等待	395
8.5.5	Enqueue (队列等待)	401
8.5.6	Latch Free (锁释放)	404
第 9 章	性能诊断与 SQL 优化	415
9.1	使用 AUTOTRACE 功能辅助 SQL 优化	415
9.1.1	AUTOTRACE 功能的启用	415
9.1.2	Oracle 10g AUTOTRACE 功能的增强	418
9.1.3	AUTOTRACE 功能的内部操作	421
9.1.4	使用 AUTOTRACE 功能辅助 SQL 优化	424
9.2	捕获问题 SQL 解决过度 CPU 消耗问题	427
9.2.1	使用 vmstat 检查系统当前情况	427
9.2.2	使用 Top 工具辅助诊断	428
9.2.3	检查进程数量	429
9.2.4	登录数据库	430
9.2.5	捕获相关 SQL	430
9.2.6	创建新的索引以消除全表扫描	435
9.2.7	观察系统状况	436
9.2.8	性能何以提高	437
9.2.9	小结	439
9.3	使用 SQL_TRACE/10046 事件进行数据库诊断	439
9.3.1	SQL_TRACE 及 10046 事件的基础介绍	439
9.3.2	诊断案例一: 隐式转换与索引失效	446
9.3.3	诊断案例二: 跟踪后台错误	451
9.3.4	10046 与等待事件	456
9.4	使用物化视图进行翻页性能调整	465
9.4.1	系统环境	466
9.4.2	问题描述	466
9.4.3	捕获排序 SQL 语句	466
9.4.4	确定典型问题 SQL	467
9.4.5	选择解决办法	471
9.4.6	进一步的调整优化	473
9.4.7	小结	475
9.5	一次横跨两岸的问题诊断	475
9.5.1	第一封求助邮件	475
9.5.2	第一次回复	479
9.5.3	进一步信息提供	479
9.5.4	进一步的诊断	481

9.5.5 最后的问题定位	482
9.5.6 小结	482
9.6 总结	482
后记	483

第1章 数据库的启动和关闭

Oracle Server 主要由两部分组成：Instance 和 Database。Instance 是指一组后台进程/线程和一块共享内存区域，而 Database 是指存储在磁盘上的一组物理文件。本章由数据库如何启动入手，开始和大家一起进入 Oracle 数据库的国度。

1.1 数据库的启动

首先来分析一下数据库的启动过程，Oracle 数据库的启动主要包含 3 个步骤：

- (1) 启动数据库到 nomount 状态；
- (2) 启动数据库到 mount 状态；
- (3) 启动数据库到 open 状态。

下面逐个来看看各个步骤的具体过程以其含义。

1.1.1 启动数据库到 nomount 状态

在启动的第一步骤，Oracle 首先寻找参数文件（pfile/spfile），然后根据参数文件中的设置，创建实例，分配内存，启动后台进程。

在这里可以看到，只要拥有了一个参数文件，就可以凭之启动实例（Instance），这一步骤并不需要任何控制文件或数据文件的参与。

在创建数据库时，如果在这一步骤就出现问题，那么通常可能是系统配置（内核参数等）存在问题，用户需要检查是否分配了足够的系统资源等。

来看一下启动到 nomount 状态的过程：

```
[oracle@jumper oracle]$ cd $ORACLE_HOME/dbs
[oracle@jumper dbs]$ ls
initconner.ora  init.ora  lkCONNER  orapwconner  spfileconner.ora  spfile.ora
[oracle@jumper dbs]$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 9.2.0.4.0 - Production on Wed Nov 3 14:57:22 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
Connected to an idle instance.
```

```
SQL> startup nomount
ORACLE instance started.
```

```
Total System Global Area  80811208 bytes
Fixed Size                  451784 bytes
Variable Size               37748736 bytes
Database Buffers            41943040 bytes
Redo Buffers                 667648 bytes
```

注意这里，Oracle 根据参数文件的内容，创建了 instance，分配了相应的内存区域，启动了相应的后台进程。

此时观察警报日志文件（alert_<sid>.log），可以看到这一阶段的启动过程，读取参数文件，应用参数启动实例，所有在参数文件中定义的非缺省参数都会记录在警报日志文件中：

```
Sat Apr 29 10:14:01 2006
Starting ORACLE instance (normal)
LICENSE_MAX_SESSION = 0
LICENSE_SESSIONS_WARNING = 0
SCN scheme 2
Using log_archive_dest parameter default value
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 9.2.0.4.0.
System parameters with non-default values:
  processes                = 150
  timed_statistics          = TRUE
  shared_pool_size          = 117440512
  large_pool_size           = 0
  java_pool_size            = 0
  .....
  aq_tm_processes           = 1
```

然后后台进程依次启动：

```
PMON started with pid=2
DBW0 started with pid=3
LGWR started with pid=4
CKPT started with pid=5
```

```

SMON started with pid=6
RECO started with pid=7
CJQ0 started with pid=8
QMN0 started with pid=9

```

这里注意一下 Oracle 选择参数文件的顺序。

在 Oracle 9i 里, Oracle 首选 `spfile<sid>.ora` 文件作为启动参数文件; 如果该文件不存在, Oracle 选择 `spfile.ora` 文件; 如果前两者都不存在, Oracle 将会选择 `init<sid>.ora` 文件; 如果以上 3 个文件都不存在, Oracle 将无法创建和启动 instance。

用户可以在 SQL*PLUS 中通过 `show parameter spfile` 命令来检查数据库是否使用了 `spfile` 文件, 如果 value 不为 Null, 则数据库使用了 `spfile` 文件:

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE
-----	-----	-----
spfile	string	?/dbs/spfile@.ora

注 意

这里的 “?” 代表 ORACLE_HOME, @代表数据库的 sid。

这时候也可以从操作系统查看启动了的后台进程:

```

[oracle@jumper oracle]$ ps -ef|grep ora_
oracle  19130    1  0 15:17 ?        00:00:00 ora_pmon_conner
oracle  19132    1  0 15:17 ?        00:00:00 ora_dbw0_conner
oracle  19134    1  0 15:17 ?        00:00:00 ora_lgwr_conner
oracle  19136    1  0 15:17 ?        00:00:00 ora_ckpt_conner
oracle  19138    1  0 15:17 ?        00:00:00 ora_smon_conner
oracle  19140    1  0 15:17 ?        00:00:00 ora_reco_conner
oracle  19142    1  0 15:17 ?        00:00:00 ora_cjq0_conner
oracle  19144    1  0 15:17 ?        00:00:00 ora_qmn0_conner
oracle  19180 19146  0 15:18 pts/1    00:00:00 grep ora_

```

```
SQL> shutdown immediate;
```

```
ORA-01507: database not mounted
```

```
ORACLE instance shut down.
```

```
SQL> !
```

现在更名 `spfile<sid>.ora` 文件, 此后 Oracle 将选择 `spfile.ora` 文件启动数据库:

```
[oracle@jumper dbs]$ mv spfileconner.ora spfileconner.ora.bak
[oracle@jumper dbs]$ exit
exit
```

```
SQL> startup nomount
ORACLE instance started.
.....
```

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE

spfile	string	?/dbs/spfile.ora

```
SQL> shutdown immediate;
ORA-01507: database not mounted
```

```
ORACLE instance shut down.
SQL> !
```

再更名 spfile.ora 文件，此时 Oracle 将选择 init<sid>.ora 文件启动数据库：

```
[oracle@jumper dbs]$ mv spfile.ora spfile.ora.bak
[oracle@jumper dbs]$ exit
exit
```

```
SQL> startup nomount
ORACLE instance started.
.....
```

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE

spfile	string	

```
SQL> shutdown immediate;
ORA-01507: database not mounted
```

```
ORACLE instance shut down.
SQL> !
```

如果这 3 个文件都不存在，Oracle 将无法启动：

```
[oracle@jumper dbs]$ mv initconner.ora initconner.ora.bak
[oracle@jumper dbs]$ exit
exit

SQL> startup
ORA-01078: failure in processing system parameters
LRM-00109: could not open parameter file '/opt/oracle/product/9.2.0/dbs/initconner.ora'
SQL>
```

可以看到这里出现了错误，报告无法找到参数文件，init<sid>.ora 是 Oracle 最后一个查找的参数文件。

在 Oracle 整个启动过程中，参数文件是写在应用程序中的硬代码，按照如上顺序进行查找，不能改变 Oracle 的搜索路径及行为，但是如果参数文件不在相应的位置，在 Linux/UNIX 系统上，可以通过符号链接来进行重定位。

在参数文件中，通常需要最少的参数是 db_name，设置了这个参数之后，数据库实例就可以启动，来看一个简单的测试。

随便命名一个实例（测试来自于 Linux 下，适用于 Linux/UNIX，对于 Windows 平台，需要通过 oradim 工具创建实例），然后尝试启动到 nomount 状态：

```
[oracle@jumper dbs]$ export ORACLE_SID=julia
[oracle@jumper dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Mon May 8 11:08:36 2006
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
Connected to an idle instance.

SQL> startup nomount;
ORA-01078: failure in processing system parameters
LRM-00109: could not open parameter file '/opt/oracle/product/9.2.0/dbs/initjulia.ora'
```

参数文件查找失败会给出提示信息，创建一个最简单的参数文件，然后就可以启动实例：

```
SQL> ! echo "db_name=julia" > /opt/oracle/product/9.2.0/dbs/initjulia.ora

SQL> startup nomount;
ORACLE instance started.

Total System Global Area  97588504 bytes
Fixed Size                  451864 bytes
Variable Size              46137344 bytes
```

Database Buffers	50331648 bytes
Redo Buffers	667648 bytes

缺省情况下，如果不设置，background_dump_dest 目录（警报日志文件 alert_<sid>.log 的存放地点）位于\$ORACLE_HOME/rdbms/log 目录下：

```
SQL> show parameter background_dump
```

NAME	TYPE	VALUE
background_dump_dest	string	?/rdbms/log

也可以顺便看下其他几个缺省路径的地点：

```
SQL> show parameter dump_dest
```

NAME	TYPE	VALUE
background_dump_dest	string	?/rdbms/log
core_dump_dest	string	?/dbs
user_dump_dest	string	?/rdbms/log

```
SQL> show parameter control_files
```

NAME	TYPE	VALUE
control_files	string	?/dbs/cntrl@.dbf

收录简单启动实例日志供大家参考：

```
[oracle@jumper dbs]$ cat $ORACLE_HOME/rdbms/log/alert_julia.log
Mon May 8 11:09:04 2006
Starting ORACLE instance (normal)
Mon May 8 11:09:04 2006
LICENSE_MAX_SESSION = 0
LICENSE_SESSIONS_WARNING = 0
SCN scheme 2
Using log_archive_dest parameter default value
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 9.2.0.4.0.
System parameters with non-default values:
```

```

db_name                = julia
PMON started with pid=2
DBW0 started with pid=3
LGWR started with pid=4
CKPT started with pid=5
SMON started with pid=6
RECO started with pid=7
[oracle@jumper dbs]$

```

这样，就通过了最少的参数需求启动了 Oracle 实例。

在使用 RMAN (Recovery Manager) 时存在更为特殊的情况，Oracle 允许在不存在参数文件的情况下启动一个实例，数据库的 db_name 会被缺省的命名为 DUMMY:

```

[oracle@jumper dbs]$ rman target /

Recovery Manager: Release 9.2.0.4.0 - Production
Copyright (c) 1995, 2002, Oracle Corporation.  All rights reserved.
connected to target database (not started)

RMAN> startup  nomount;

startup failed: ORA-01078: failure in processing system parameters
LRM-00109: could not open parameter file '/opt/oracle/product/9.2.0/dbs/initconner.ora'

trying to start the Oracle instance without parameter files ...
Oracle instance started

Total System Global Area      97588504 bytes

Fixed Size                    451864 bytes
Variable Size                 46137344 bytes
Database Buffers              50331648 bytes
Redo Buffers                   667648 bytes

RMAN> host ;

[oracle@jumper dbs]$ sqlplus "/" as sysdba

SQL*Plus: Release 9.2.0.4.0 - Production on Tue Mar 12 14:17:07 2006

```

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:

Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

With the Partitioning option

JServer Release 9.2.0.4.0 - Production

SQL> show parameter db_name

NAME	TYPE	VALUE
db_name	string	DUMMY

此时警告日志文件中会记录如下信息：

Starting up ORACLE RDBMS Version: 9.2.0.4.0.

System parameters with non-default values:

remote_login_passwordfile= EXCLUSIVE

db_name = DUMMY

PMON started with pid=2

DBW0 started with pid=3

.....

在实例创建以后，Oracle 就可以逐步导航，完成数据库的加载、打开等工作。

1.1.2 启动数据库到 mount 状态

启动到 nomount 状态以后，Oracle 就可以从参数文件中获得控制文件的位置信息，这一部分信息在参数文件中的记录类似如下所示（Oracle 缺省会创建 3 个控制文件，这 3 个控制文件的内容完全一致，是 Oracle 为了安全而采用的镜像手段，在生产环境中，通常应该将 3 个控制文件存放在不同的物理硬盘上，避免因为介质故障而同时损坏 3 个控制文件）：

```
*.control_files='/opt/oracle/oradata/conner/control01.ctl',
                '/opt/oracle/oradata/conner/control02.ctl',
                '/opt/oracle/oradata/conner/control03.ctl'
```

在 nomount 状态，可以查询 v\$parameter 视图，获得控制文件信息，这部分信息来自启动的参数文件；当数据库 mount 之后，可以查询 v\$controlfile 视图获得关于控制文件的信息，此时，这部分信息来自控制文件：

```
SQL> startup nomount;

ORACLE instance started.
```

```

Total System Global Area  80811208 bytes
Fixed Size                  451784 bytes
Variable Size              37748736 bytes
Database Buffers           41943040 bytes
Redo Buffers                667648 bytes
SQL> select * from v$controlfile;

no rows selected

SQL> show parameter control_files

NAME                           TYPE                           VALUE
-----
control_files                   string                          /opt/oracle/oradata/conner/control01.ctl,
                                   /opt/oracle/oradata/conner/control02.ctl,
                                   /opt/oracle/oradata/conner/control03.ctl

SQL> alter database mount;
Database altered.

SQL> select * from v$controlfile;
STATUS  NAME
-----
        /opt/oracle/oradata/conner/control01.ctl
        /opt/oracle/oradata/conner/control02.ctl
        /opt/oracle/oradata/conner/control03.ctl

```

在 mount 数据库的过程中，Oracle 需要找到控制文件并锁定控制文件。如果控制文件全部丢失此时就会报出如下错误：

```
ORA-00205: error in identifying controlfile, check alert log for more info
```

这时候 alert<sid>.log 文件中通常会记录更为详细的信息：

```
ORA-00202: controlfile: '/opt/oracle/oradata/conner/control01.ctl'
```

```
ORA-27037: unable to obtain file status
```

```
Linux Error: 2: No such file or directory
```

```
Additional information: 3
```

因为 Oracle 的 3 个（缺省的）控制文件内容完全相同，如果只是损失了其中 1~2 个，可以复制完好的控制文件，更改为相应的名称，就可以启动数据库；如果丢失了所有的控制

文件，那么就需要恢复或重建控制文件来打开数据库。

在正常 Mount 数据库的过程中，数据库的警报日志文件仅记录如下信息：

```
alter database mount
Sat Apr 29 10:20:42 2006
Successful mount of redo thread 1, with mount id 1408096182.
Sat Apr 29 10:20:42 2006
Database mounted in Exclusive Mode.
Completed: alter database mount
```

在这一步骤中，数据库需要计算 Mount id 并将其记录在控制文件中，然后开始启动 Heartbeat（心跳），每 3 秒更新一次控制文件。可以用以下命令间隔 3 秒转储 2 次控制文件信息：

```
alter session set events 'immediate trace name CONTROLF level 10';
```

在 Linux 上用 diff 命令比较两个文件，可以发现，控制文件在 Mount 状态下发生改变的只有这个 Heartbeat：

```
[oracle@jumper udump]$ diff conner_ora_25542.trc conner_ora_25706.trc
...
64c63
< heartbeat: 588983634 mount id: 1408096182
---
> heartbeat: 588983636 mount id: 1408096182
```

Heartbeat 表明实例已经被特定例程所 Mount，这个属性主要用于 OPS/RAC 环境。但是 Heartbeat 在单实例环境中同样存在。

可以从一个内部表（需要以 SYS 用户登录）中查询到当前的 Heartbeat 值：

```
SELECT CPHBT from X$KCCCP;
```

从 Oracle 9i 开始，Oracle 在数据库内部通过等待事件 control file heartbeat 来记录这个事件的相关等待：

```
SQL> select event#,name
      2  from v$event_name where name like '%heart%';

EVENT# NAME
-----
145 control file heartbeat
```

了解了启动的各个步骤，也就可以在发生问题的时候，快速定位，准确判断，从而快速解决问题。

启动到 Mount 状态，数据库必须具备的另外一个重要文件是口令文件，该文件位于 \$ORACLE_HOME/dbs 目录下，缺省的名称为 orapw<sid>。

口令文件中存放 sysdba/sysoper 用户的用户名及口令：

```
[oracle@jumper dbs]$ strings orapwconner
]|Z
ORACLE Remote Password file
CONNER
INTERNAL
AB27B53EDC5FEF41
8A8F025737A9097A
EYGLE
B726E09FE21F8E83
```

在数据库没有启动之前，数据库内建用户是无法通过数据库本身来验证身份的，通过口令文件，Oracle 可以实现对用户的身份认证，在数据库未启动之前登录，进而启动数据库。如果丢失了口令文件，在 mount 阶段就会出现错误：

```
[oracle@jumper dbs]$ mv orapwconner orapwconner.b
[oracle@jumper dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Sun Apr 30 15:01:02 2006
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
Connected to an idle instance.

SQL> startup nomount;
ORACLE instance started.

Total System Global Area 126948772 bytes
Fixed Size 452004 bytes
Variable Size 92274688 bytes
Database Buffers 33554432 bytes
Redo Buffers 667648 bytes
SQL> alter database mount;
alter database mount
*
ERROR at line 1:
ORA-01990: error opening password file '/opt/oracle/product/9.2.0/dbs/orapw'
ORA-27037: unable to obtain file status
Linux Error: 2: No such file or directory
Additional information: 3
SQL>
```

对于口令文件，Oracle 缺省查找 orapw<sid>文件，如果该文件不存在，则继续查找 orapw 文件，如果两者都不存在，则数据库将会出现错误。

如果口令文件丢失，通过 orapw 工具即可重建，所以在通常的备份策略中可以不必包含口令文件：

```
[oracle@jumper dbs]$ orapwd
Usage: orapwd file=<fname> password=<password> entries=<users>

where
    file - name of password file (mand),
    password - password for SYS (mand),
    entries - maximum number of distinct DBA and OPERs (opt),

There are no spaces around the equal-to (=) character.
```

初始化参数 remote_login_passwordfile 和口令文件有关，限于篇幅，本文不再过多介绍。

通常在 Linux/UNIX 平台下，在 \$ORACLE_HOME/dbs 目录下，还会存在另外一个文件，该文件命名规则为 lk<SID>，lk 指 lock，该文件在数据库启动时创建，用于操作系统对数据库的锁定。当数据库启动时获得锁定，数据库关闭时释放。

有时在系统出现异常时，可能数据库已经关闭，但是锁定并未释放，或者因为后台进程未正常停止等原因，会导致下次数据库无法启动，相关的错误信息类似如下：

```
Sun Apr 30 06:08:58 2006
ALTER DATABASE MOUNT
Sun Apr 30 06:08:58 2006
scumnt: failed to lock /export/product/oracle/app/dbs/lkBILL exclusive
Sun Apr 30 06:08:58 2006
ORA-09968: scumnt: unable to lock file
SVR4 Error: 11: Resource temporarily unavailable
Additional information: 20169
```

该文件内容通常只有一行，提示不要删除，该文件仅仅用于锁定：

```
bash-2.03$ more lkBILL
DO NOT DELETE THIS FILE!
```

这样的问题很少出现，通常在正常关闭数据库后即可重新启动。

1.1.3 启动数据库 open 阶段

由于控制文件中记录了数据库中数据文件、日志文件的位置信息、检查点信息等重要信息，所以在数据库的 open 阶段，Oracle 可以根据控制文件中记录的这些信息找到这些文件，然后进行检查点及完整性检查。

如果不存在问题就可以启动数据库，如果存在不一致或文件丢失则需要恢复。

进一步地说，实际上在数据库 open 的过程中，Oracle 进行的检查中包括以下两项：

第一次检查数据文件头中的检查点计数（Checkpoint cnt）是否和控制文件中的检查点计数（Checkpoint cnt）一致。此步骤检查用以确认数据文件是来自同一版本，而不是从备份中恢复而来（因为 Checkpoint Cnt 不会被冻结，会一直被修改）。

下面通过一个简单的测试来说明一下 Checkpoint Cnt（为了节省篇幅，省略了部分转储信息）的作用。

首先通过如下命令在不同条件下转储控制文件，第一步转储正常状态下的控制文件：

```
SQL> alter session set events 'immediate trace name CONTROLF level 10';
```

Session altered.

将系统表空间置于热备份状态（热备份状态会冻结表空间数据文件的检查点）：

```
SQL> alter tablespace system begin backup;
```

Tablespace altered.

再来转储控制文件：

```
SQL> alter session set events 'immediate trace name CONTROLF level 10';
```

Session altered.

手工执行检查点并转储控制文件：

```
SQL> alter system checkpoint;
```

System altered.

```
SQL> alter session set events 'immediate trace name CONTROLF level 10';
```

Session altered.

结束表空间的热备份状态，再次转储控制文件：

```
SQL> alter tablespace system end backup;
```

Tablespace altered.

```
SQL> alter session set events 'immediate trace name CONTROLF level 10';
```

Session altered.

简要地来看一下跟踪文件信息（仅研究 system 表空间记录）。

（1）正常情况下转储控制文件。

```

*****
DATA FILE RECORDS
*****

(blkno = 0x6, size = 180, max = 100, in-use = 24, last-recid= 574)

DATA FILE #1:

(name #4) /opt/oracle/oradata/hsjf/system01.dbf

creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1

tablespace 0, index=1 krfl=1 prev_file=0

unrecoverable scn: 0x0000.00000000 04/23/2004 01:20:52

Checkpoint cnt:1567 scn: 0x0000.0148181c 06/22/2004 18:58:46

Stop scn: 0xffff.ffffffff 06/22/2004 18:58:05

Creation Checkpointed at scn: 0x0000.000000ae 07/16/2003 03:40:10

.....

```

注意这里记录的检查点计数器及 SCN。

(2) 执行 Begin Backup 以后的。

注意到 Checkpoint cnt 增加了 1，对表空间执行 Begin Backup 会触发一次表空间检查点：

```

*****
DATA FILE RECORDS
*****

(blkno = 0x6, size = 180, max = 100, in-use = 24, last-recid= 574)

DATA FILE #1:

(name #4) /opt/oracle/oradata/hsjf/system01.dbf

creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1

tablespace 0, index=1 krfl=1 prev_file=0

unrecoverable scn: 0x0000.00000000 04/23/2004 01:20:52

Checkpoint cnt:1568 scn: 0x0000.01481939 06/22/2004 19:02:22

Stop scn: 0xffff.ffffffff 06/22/2004 18:58:05

Creation Checkpointed at scn: 0x0000.000000ae 07/16/2003 03:40:10

.....

```

可以注意到检查点计数器随之增加。

(3) 执行手工检查点。

在表空间热备份模式下，手工执行检查点后，可以看到，此时 Checkpoint cnt 增加，但是 SCN 不再改变。这是由于表空间处于热备份模式，数据文件检查点被冻结（热备模式下，数据库会生成额外的 Redo 日志，在本书后面的章节中会详细介绍）：

```

*****
DATA FILE RECORDS

```

```

*****
(blkno = 0x6, size = 180, max = 100, in-use = 24, last-recid= 574)
DATA FILE #1:
(name #4) /opt/oracle/oradata/hsjf/system01.dbf
creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1
tablespace 0, index=1 krfil=1 prev_file=0
unrecoverable scn: 0x0000.00000000 04/23/2004 01:20:52
Checkpoint cnt:1569 scn: 0x0000.01481939 06/22/2004 19:02:22
Stop scn: 0xffff.ffffffff 06/22/2004 18:58:05
Creation Checkpointed at scn: 0x0000.000000ae 07/16/2003 03:40:10
.....

```

(4) End Backup 后的情况。

此时数据文件头的冻结被取消，SCN 开始变化。

```

*****
DATA FILE RECORDS
*****
(blkno = 0x6, size = 180, max = 100, in-use = 24, last-recid= 574)
DATA FILE #1:
(name #4) /opt/oracle/oradata/hsjf/system01.dbf
creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1
tablespace 0, index=1 krfil=1 prev_file=0
unrecoverable scn: 0x0000.00000000 04/23/2004 01:20:52
Checkpoint cnt:1570 scn: 0x0000.01481941 06/22/2004 19:02:39
Stop scn: 0xffff.ffffffff 06/22/2004 18:58:05
Creation Checkpointed at scn: 0x0000.000000ae 07/16/2003 03:40:10
.....

```

这就是检查点计数器及其在不同模式下的变化。

如果检查点计数检查通过，则数据库进行第二次检查。第二次检查数据文件头的开始 SCN 和控制文件中记录的该文件的结束 SCN 是否一致，如果控制文件中记录的结束 SCN 等于数据文件头的开始 SCN，则不需要对那个文件进行恢复。

对每个数据文件都完成检查后，打开数据库，锁定数据文件，同时将每个数据文件的结束 SCN 设置为无穷大（稍后将详细解释这个过程）。

看一下以下测试，如果数据库中的某个文件丢失：

```

[oracle@jumper oracle]$ mv /opt/oracle/oradata/conner/eygle01.dbf /opt/oracle/oradata/ conner/eygle
01.dbf.bak

```

那么在启动数据库的时候，在 Open 阶段，Oracle 才会来检查这个文件的存在性，如果文件不存在，数据库就会给出错误信息，停止启动：

```

SQL> startup nomount;

ORACLE instance started.


Total System Global Area   80811208 bytes
Fixed Size                   451784 bytes
Variable Size               37748736 bytes
Database Buffers            41943040 bytes
Redo Buffers                 667648 bytes
SQL> alter database mount;

Database altered.


SQL> alter database open;

alter database open
*

ERROR at line 1:

ORA-01157: cannot identify/lock data file 4 - see DBWR trace file
ORA-01110: data file 4: '/opt/oracle/oradata/conner/eygle01.dbf'

```

注 意

仅在 open 阶段，Oracle 才尝试打开并锁定数据文件，如果丢失或出现问题，则会给出错误提示。这时候就需要 dba 的介入进行处理，根据不同情况进行相应的恢复。

现在来看看 alert<sid>.log 文件中记录的 open 过程中提示的错误信息：

```

Wed Nov  3 16:51:41 2004

alter database open

Wed Nov  3 16:51:41 2004

Errors in file /opt/oracle/admin/conner/bdump/conner_dbw0_19646.trc:
ORA-01157: cannot identify/lock data file 4 - see DBWR trace file
ORA-01110: data file 4: '/opt/oracle/oradata/conner/eygle01.dbf'
ORA-27037: unable to obtain file status
Linux Error: 2: No such file or directory
Additional information: 3
ORA-1157 signalled during: alter database open...

```

在数据库出现问题的时候，提示中给出的可能是不完整的信息，而警报日志中则记录了完整的错误过程和错误号。所以当数据库出现故障时，应该优先检查 alert_<sid>.log，从中发现关于故障的详细信息。

曾经在论坛上看到无数类似这样的提问：

- 我的数据库启动不了了，怎么办？
- 我的数据库慢，怎么办？
- 我的数据库 DOWN 了，怎么办？

面对这样的问题，我们往往无能为力，这里面没有任何实质性的信息，无从判断。所以说，在提问之前，应该想想你想传达怎样的信息给别人。如果只想说，我的数据库出问题了，那么别人只能了解这个事实，没有办法帮你。学会提问，也需要智慧。

在这里需要提醒大家的是，在数据库出现问题的时候，首先检查 `alert_<sid>.log` 文件，研究其中的警告信息或者提供他人寻求帮助，这是通常是解决问题的第一个步骤。

从警报日志文件来看一下完整的数据库启动过程：

```
Sat Apr 29 11:44:45 2006
alter database open
Sat Apr 29 11:44:45 2006
Thread 1 opened at log sequence 124
  Current log# 1 seq# 124 mem# 0: /opt/oracle/oradata/eygle/redo01.log
Successful open of redo thread 1.
Sat Apr 29 11:44:45 2006
SMON: enabling cache recovery
Sat Apr 29 11:44:46 2006
Undo Segment 1 Onlined
Undo Segment 2 Onlined
Undo Segment 3 Onlined
Undo Segment 4 Onlined
Undo Segment 5 Onlined
Undo Segment 6 Onlined
Undo Segment 7 Onlined
Undo Segment 8 Onlined
Undo Segment 9 Onlined
Undo Segment 10 Onlined
Successfully onlined Undo Tablespace 1.
Sat Apr 29 11:44:46 2006
SMON: enabling tx recovery
Sat Apr 29 11:44:46 2006
Database Characterset is ZHS16GBK
replication_dependency_tracking turned off (no async multimaster replication found)
Completed: alter database open
```

在完成数据库的验证和恢复过程后，数据库处于一致的状态，数据库还需要进行一系列的处理过程：将 Undo 段在线等操作，然后数据库可以提供访问，同时 SMON 可以开始进行

事务回滚等。

在启动日志里，读者可能注意到了这样一行：

```
Database Characterset is ZHS16GBK
```

在每次数据库的启动过程中，Oracle 都需要判断控制文件中记录的字符集和数据库中的字符集是否相符，如果相符，则记录如上一行日志；如果不相符合，则以数据库中的字符集为准更新控制文件中的字符集记录，类似的日志如下：

```
Updating character set in controlfile to ZHS16CGB231280
```

—— 提 示 ——

在 Oracle 8i 之前，可以通过 Update props\$表的方式修改字符集，从 Oracle 8i 开始，切记绝对不要使用同样的方式修改字符集。

如果细致一些，启动日志中的每条信息都是值得研究的。

1.2 进阶内容

控制文件在数据库中扮演着重要的角色，我们完全有必要来关注一下控制文件的结构及原理。开始之前，先来了解两个概念：SCN 和检查点。

1.2.1 SCN

1. SCN 的定义

SCN (System Change Number)，也就是通常所说的系统改变号，是数据库中非常重要的一个数据结构。

SCN 用以标识数据库在某个确切时刻提交的版本。在事务提交时，它被赋予一个惟一的标识事务的 SCN。SCN 同时被作为 Oracle 数据库的内部时钟机制，可被看作逻辑时钟，每个数据库都有一个全局的 SCN 生成器。

作为数据库内部的逻辑时钟，数据库事务依 SCN 而排序，Oracle 也依据 SCN 来实现一致性读 (Read Consistency) 等重要数据库功能。另外对于分布式事务 (Distributed Transactions)，SCN 也极为重要，这里不作更多介绍。

SCN 在数据库中是惟一的，并随时间而增加，但是可能并不连贯。除非重建数据库，SCN 的值永远不会被重置为 0。

一直以来，对于 SCN 有很多争议，很多人认为 SCN 是指 System Commit Number，而通常 SCN 在提交时才变化，所以很多时候，这两个名词经常在文档中反复出现。即使在 Oracle 的官方文档中，SCN 也常以 System Change/Commit Number 两种形式出现。

到底是哪个词其实不是最重要的，重要的是要知道 SCN 是 Oracle 内部的时钟机制，Oracle 通过 SCN 来维护数据库的一致性，并通过 SCN 实施 Oracle 至关重要的恢复机制。

SCN 在数据库中是无处不在的，常见的事务表、控制文件、数据文件头、日志文件、数据块头等都记录有 SCN 值。

冠以不同前缀，SCN 也有了不同的名称，如检查点 SCN (Checkpoint SCN)、Resetlogs SCN 等。

2. SCN 的获取方式

可以通过如下几种方式获得数据库的当前或近似 SCN。

(1) 从 Oracle9i 开始。

可以使用 `dbms_flashback.get_system_change_number` 来获得：

```
SQL> select dbms_flashback.get_system_change_number from dual;
```

```
GET_SYSTEM_CHANGE_NUMBER
```

```
-----
```

```
2982184
```

(2) Oracle 9i 前。

可以通过查询 `x$ktuxe` 获得系统最接近当前值的 SCN：

`X$KTUXE` 的含义是[K]ernel [T]ransaction [U]ndo Transa[x]tion [E]ntry (table)

```
SQL> select max(ktuxescnw*power(2,32)+ktuxescnb) from x$ktuxe;
```

```
MAX(KTUXESCNW*POWER(2,32)+KTUXESCNB)
```

```
-----
```

```
2980613
```

3. SCN 的进一步说明

系统当前 SCN 并不是在任何的数据库操作发生时都会改变，SCN 通常在事务提交或回滚时改变。在控制文件、数据文件头、数据块、日志文件头、日志文件 `change vector` 中都有 SCN，但其作用各不相同。

(1) 数据文件头中包含了该数据文件的 Checkpoint SCN，表示该数据文件最近一次执行检查点操作时的 SCN。

从控制文件的 `dump` 文件中，可以得到以下内容：

```
DATA FILE #1:
```

```
(name #4) /opt/oracle/oradata/conner/system01.dbf
```

```
creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1
```

```
tablespace 0, index=1 krfil=1 prev_file=0
```

```
unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
```

```
Checkpoint cnt:273 scn: 0x0000.0023aff1 11/22/2004 17:10:11
```

```
Stop scn: 0xffff.ffffffff 11/22/2004 16:58:49
```

```
Creation Checkpointed at scn: 0x0000.00000008 10/20/2004 20:59:35
```

```
thread:1 rba:(0x1.3.10)
```

```
.....
```

对于每一个数据文件都包含一个这样的条目，记录该文件的检查点 SCN 的值以及检查点发生的时间，这里的 Checkpoint SCN、Stop SCN 以及 Checkpoint Cnt 都是非常重要的数据结构，我们将会在下面检查点部分详细介绍。

同样可以通过命令转储数据文件头，观察其具体信息及检查点记录等：

```
SQL> alter session set events 'immediate trace name file_hdrs level 10';
```

```
Session altered.
```

```
SQL> @gettrcname
```

```
TRACE_FILE_NAME
```

```
-----  
/opt/oracle/admin/conner/udump/conner_ora_5862.trc
```

```
SQL> !
```

从跟踪文件中摘取 SYSTEM 表空间的记录作为参考：

```
DATA FILE #1:
```

```
(name #4) /opt/oracle/oradata/conner/system01.dbf
```

```
creation size=32000 block size=8192 status=0xe head=4 tail=4 dup=1
```

```
tablespace 0, index=1 krfil=1 prev_file=0
```

```
unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
```

```
Checkpoint cnt:319 scn: 0x0000.002e3016 12/03/2004 06:42:18
```

```
Stop scn: 0xffff.fffffff 12/01/2004 23:37:33
```

```
Creation Checkpointed at scn: 0x0000.00000008 10/20/2004 20:59:35
```

```
thread:1 rba:(0x1.3.10)
```

```
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
```

```
00000000 00000000
```

```
Offline scn: 0x0000.001cff67 prev_range: 0
```

```
Online Checkpointed at scn: 0x0000.001cff68 11/16/2004 14:10:35
```

```
thread:1 rba:(0x1.2.0)
```

```
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
```

```
00000000 00000000
```

```
Hot Backup end marker scn: 0x0000.00000000
```

```
aux_file is NOT DEFINED
```

```
FILE HEADER:
```

```
Software vsn=153092096=0x9200000, Compatibility Vsn=134217728=0x8000000
```

```
Db ID=3152029224=0xbbe02628, Db Name='CONNER'
```

```

Activation ID=0=0x0
Control Seq=1093=0x445, File size=32000=0x7d00
File Number=1, Blksiz=8192, File Type=3 DATA
Tablespace #0 - SYSTEM   rel_fn:1
Creation   at   scn: 0x0000.00000008 10/20/2004 20:59:35
Backup taken at scn: 0x0000.001aca21 11/14/2004 09:08:34 thread:1
  reset logs count:0x20541edb scn: 0x0000.001cff68 recovered at 12/01/2004 23:07:30
  status:0x4 root dba:0x004001a1 chkpt cnt: 319 ctl cnt:318
begin-hot-backup file size: 32000
Checkpointed at scn:   0x0000.002e3016 12/03/2004 06:42:18
  thread:1 rba:(0x35.2.10)
  enabled  threads:  01000000 00000000 00000000 00000000 00000000 00000000
                    00000000 00000000
Backup Checkpointed at scn:   0x0000.001aca21 11/14/2004 09:08:34
  thread:1 rba:(0xc6.4fff.10)
  enabled  threads:  01000000 00000000 00000000 00000000 00000000 00000000
                    00000000 00000000
External cache id: 0x0 0x0 0x0 0x0
Absolute fuzzy scn: 0x0000.00000000
Recovery fuzzy scn: 0x0000.00000000 01/01/1988 00:00:00
Terminal Recovery Stamp scn: 0x0000.00000000 01/01/1988 00:00:00

```

(2) 日志文件头中包含了 Low SCN 和 Next SCN。

Low SCN 和 Next SCN 这两个 SCN 表示该日志文件包含有介于 Low SCN 到 Next SCN 的重做信息，对于 Current 的日志文件（当前正在被使用的 Redo Logfile），其最终 SCN 不可知，所以 Next SCN 被置为无穷大，也就是 ffffffff。

来看一下日志文件的情况：

```

SQL> select * from v$log;

```

GROUP#	THREAD#	SEQUENCE#	BYTES	MEMBERS	ARC	STATUS	FIRST_CHANGE#	FIRST_TIM
1	1	50	10485760	1	YES	ACTIVE	2973017	02-DEC-04
2	1	51	10485760	1	NO	CURRENT	2984378	02-DEC-04
3	1	49	10485760	1	YES	INACTIVE	2966611	01-DEC-04

```

SQL> select dbms_flashback.get_system_change_number from dual;

```

```
GET_SYSTEM_CHANGE_NUMBER
```

```
-----
```

2984476

```
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	50	10485760	1	YES	INACTIVE	2973017	02-DEC-04
2	1	51	10485760	1	YES	INACTIVE	2984378	02-DEC-04
3	1	52	10485760	1	NO	CURRENT	2984481	02-DEC-04

可以看到, SCN **2984476** 显然位于 Log Group# 为 2 的日志文件中, 该日志文件包含了 SCN 自 **2984378** 至 **2984481** 的 Redo 信息。Oracle 在进行恢复时, 就需要根据低 SCN 和高 SCN 来确定需要的恢复信息位于哪一个日志或归档文件中。

如果通过控制文件转储, 可以在控制文件中找到关于日志文件的信息:

```
LOG FILE #1:
```

```
(name #1) /opt/oracle/oradata/conner/redo01.log
```

```
Thread 1 redo log links: forward: 2 backward: 0
```

```
siz: 0x5000 seq: 0x00000011 hws: 0x2 bsz: 512 nab: 0x2 flg: 0x1 dup: 1
```

```
Archive links: fwr: 0 back: 0 Prev scn: 0x0000.0023ac36
```

```
Low scn: 0x0000.0023afee 11/22/2004 17:10:06
```

```
Next scn: 0x0000.0023aff1 11/22/2004 17:10:11
```

```
LOG FILE #2:
```

```
(name #2) /opt/oracle/oradata/conner/redo02.log
```

```
Thread 1 redo log links: forward: 3 backward: 1
```

```
siz: 0x5000 seq: 0x00000012 hws: 0x2 bsz: 512 nab: 0x19 flg: 0x1 dup: 1
```

```
Archive links: fwr: 0 back: 0 Prev scn: 0x0000.0023afee
```

```
Low scn: 0x0000.0023aff1 11/22/2004 17:10:11
```

```
Next scn: 0x0000.0023b01e 11/22/2004 17:10:54
```

```
LOG FILE #3:
```

```
(name #3) /opt/oracle/oradata/conner/redo03.log
```

```
Thread 1 redo log links: forward: 0 backward: 2
```

```
siz: 0x5000 seq: 0x00000013 hws: 0x1 bsz: 512 nab: 0xffffffff flg: 0x8 dup: 1
Archive links: fwr: 0 back: 0 Prev scn: 0x0000.0023aff1
Low scn: 0x0000.0023b01e 11/22/2004 17:10:54
Next scn: 0xffff.ffffff 01/01/1988 00:00:00
```

可以注意到, Log File3 是当前的日志文件, 该文件拥有的 Next SCN 为无穷大。
同样, 可以通过直接 dump 日志文件的方式来进行转储:

```
SQL> select * from v$logfile;
```

GROUP#	STATUS	TYPE	MEMBER
1	ONLINE		/opt/oracle/oradata/conner/redo01.log
2	ONLINE		/opt/oracle/oradata/conner/redo02.log
3	ONLINE		/opt/oracle/oradata/conner/redo03.log

```
SQL> alter system dump logfile '/opt/oracle/oradata/conner/redo01.log';
```

```
System altered.
```

在 trace 文件中, 可以看到关于 SCN 的详细内容:

```
DUMP OF REDO FROM FILE '/opt/oracle/oradata/conner/redo01.log'
Opco des *. *
DBA's: (file # 0, block # 0) thru (file # 65534, block # 4194303)
RBA's: 0x000000.00000000.0000 thru 0xffffffff.ffffff.ffff
SCN's scn: 0x0000.00000000 thru scn: 0xffff.ffffff
Times: creation thru eternity
FILE HEADER:
    Software vsn=153092096=0x9200000, Compatibility Vsn=153092096=0x9200000
    Db ID=3152029224=0xbbe02628, Db Name='CONNER'
    Activation ID=3154332244=0xbc034a54
    Control Seq=1084=0x43c, File size=20480=0x5000
    File Number=1, Blksiz=512, File Type=2 LOG
descrip: "Thread 0001, Seq# 0000000050, SCN 0x0000002d5d59-0x0000002d89ba"
thread: 1 nab: 0x15be seq: 0x00000032 hws: 0x2 eot: 0 dis: 0
reset logs count: 0x20541edb scn: 0x0000.001cff68
Low scn: 0x0000.002d5d59 12/02/2004 11:25:40
Next scn: 0x0000.002d89ba 12/02/2004 15:29:42
Enabled scn: 0x0000.001cff68 11/16/2004 14:10:35
```

```

Thread closed scn: 0x0000.002d5d59 12/02/2004 11:25:40
Log format vsn: 0x80000000 Disk cksum: 0xd79c Calc cksum: 0xd79c
Terminal Recovery Stamp scn: 0x0000.00000000 01/01/1988 00:00:00
Most recent redo scn: 0x0000.00000000
Largest LWN: 0 blocks
End-of-redo stream : No
Unprotected mode
Miscellaneous flags: 0x0

```

这里不打算详细介绍具体命令的用法及更进一步的内容，因为对于一本书来说，这些内容还是太广阔了。在这里只希望给大家直观的认识，有兴趣的朋友可以由此开始进一步的探索。

1.2.2 检查点

1. 检查点 (Checkpoint) 的本质

许多文档把 Checkpoint 描述得非常复杂，为我们正确理解检查点带来了障碍，结果现在检查点变成了一个非常复杂的问题。实际上，检查点只是一个数据库事件，它存在的根本意义在于减少崩溃恢复 (Crash Recovery) 时间。

当修改数据时，需要首先将数据读入内存中 (Buffer Cache)，修改数据的同时，Oracle 会记录重做信息 (Redo) 用于恢复。因为有了重做信息的存在，Oracle 不需要在提交时立即将变化的数据写回磁盘 (立即写的效率会很低)，重做 (Redo) 的存在也正是为了在数据库崩溃之后，数据可以恢复。

最常见的情况，数据库可能因为断电而 Crash，那么内存中修改过的、尚未写入文件的数据将会丢失。在下次数据库启动之后，Oracle 可以通过重做日志 (Redo) 进行事务重演 (也就是进行前滚)，将数据库恢复到崩溃之前的状态，然后数据库可以打开提供使用，之后 Oracle 可以将未提交的事务进行回滚。

在这个过程中，通常大家最关心的是数据库要经历多久才能打开。也就是需要读取多少重做日志才能完成前滚。当然用户希望这个时间越短越好，Oracle 也正是通过各种手段在不断优化这个过程，缩短恢复时间。

检查点的存在就是为了缩短这个恢复时间。

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 Data) 从 Buffer Cache 写入磁盘，当写入完成之后，CKPT 进程更新控制文件和数据文件头，记录检查点信息，标识变更。

Checkpoint SCN 可以从数据库中查询得到：

```

SQL> select file#,CHECKPOINT_CHANGE#,to_char(CHECKPOINT_TIME,'yyyy-mm-dd hh24:mi:ss') CPT
       2  from v$datafile;

```

```
FILE# CHECKPOINT_CHANGE# CPT
```

```
-----
1      8904572779065 2006-06-05 16:25:19
2      8904572779065 2006-06-05 16:25:19
3      8904572779065 2006-06-05 16:25:19
.....
```

```
13 rows selected
```

```
SQL> select dbid,CHECKPOINT_CHANGE#
2 from v$database;
```

```
DBID CHECKPOINT_CHANGE#
```

```
-----
3965153484      8904572779065
```

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。

图 1-1 中标记了 3 个日志组，假定在 T1 时间点，数据库完成并记录了最后一次检查点，在 T2 时刻数据库 Crash。那么在下次数据库启动时，T1 时间点之前的 Redo 不再需要进行恢复，Oracle 需要重新应用的就是时间点 T1 至 T2 之间数据库生成的重做日志 (Redo)。

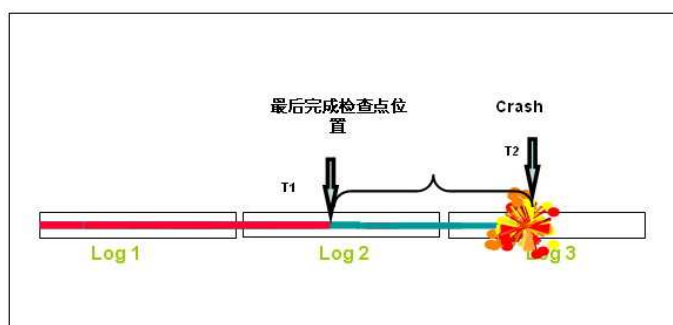


图 1-1 检查点之前的 Redo 不需进行恢复

从图 1-1 中也可以很容易地看出，检查点的频度对于数据库的恢复时间具有极大的影响，如果检查点的频率高，那么恢复时需要应用的重做日志就相对得少，恢复时间就可以缩短。然而，需要注意的是，数据库内部操作的相关性极强，过于频繁的检查点同样会带来性能问题，尤其是更新频繁的数据库。所以数据库的优化是一个系统工程，不能草率。

更进一步可以知道，如果 Oracle 可以在性能允许的情况下，使得检查点的 SCN 逐渐逼近 Redo 的最新变更，那么最终可以获得一个最佳平衡点，使得 Oracle 可以最大化的减少恢复时间。

为了实现这个目标，Oracle 在不同版本中一直在改进检查点的算法。

2. 常规检查点与增量检查点

为了区分，在 Oracle 8 之前，Oracle 实施的检查点通常被称为常规检查点（Conventional Checkpoint），这类检查点按一定的条件触发（log_checkpoint_interval、log_checkpoint_timeout 参数设置及 log switch 等条件触发）。

从 Oracle 8 开始，Oracle 引入了增量检查点（Incremental Checkpoint）的概念。

和以前的版本相比，在新版本中，Oracle 主要引入了检查点队列（Checkpoint Queue）机制，在数据库内部，每一个脏数据块都会被移动到检查点队列，按照 Low RBA 的顺序（第一次对此数据块修改对应的 Redo Byte Address）来排列，如果一个数据块进行过多次修改，该数据块在检查点队列上的顺序并不会发生变化。

当执行检查点时，DBWR 从检查点队列按照 Low RBA 的顺序写出，实例检查点因此可以不断增进、阶段性的，CKPT 进程使用非常轻量级的控制文件更新协议，将当前的最低 RBA 写入控制文件。

因为增量检查点可以连续地进行，因此检查点 RBA 可以比常规检查点更接近数据库的最后状态，从而在数据库的实例恢复中可以极大地减少恢复时间。

而且，通过增量检查点，DBWR 可以持续进行写出，从而避免了常规检查点出发的峰值写入对于 I/O 的过度征用，通过图 1-2 可以清楚地看到这一改进的意义。

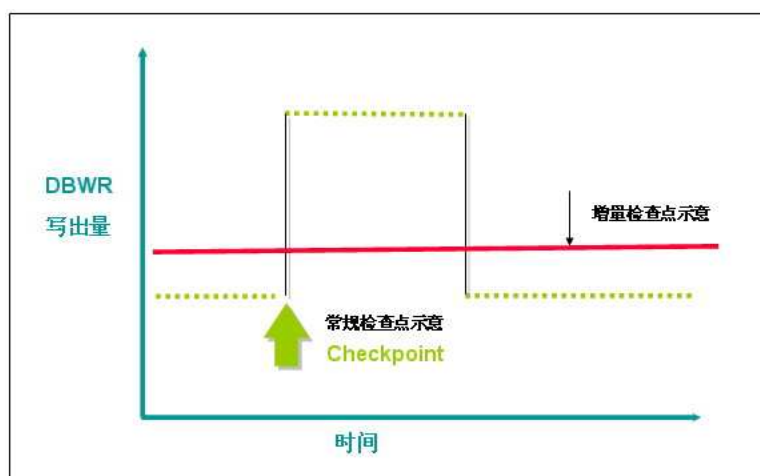


图 1-2 增量检查点对 DBWR 的影响

在数据库中，增量检查点是通过 Fast-Start Checkpointing 特性来实现的，从 Oracle 8i 开始，这一特性包含在 Oracle 企业版的 Fast-Start Fault Recovery 组件之中，通过查询 v\$option 视图，了解这一特性：

```
SQL> select * from v$version where rownum <2;
```

BANNER

Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production

```
SQL> col parameter for a30
SQL> col value for a20
SQL> select * from v$option
      2  where Parameter='Fast-Start Fault Recovery';
```

PARAMETER	VALUE
Fast-Start Fault Recovery	TRUE

该组件包含 3 个主要特性，可以加快系统在故障后的恢复，提高系统的可用性。

- Fast-Start Checkpointing;
- Fast-Start On-Demand Rollback;
- Fast-Start Parallel Rollback。

Fast-Start Checkpointing 特性在 Oracle 8i 中主要通过参数 FAST_START_IO_TARGET 来实现；在 Oracle 9i 中，Fast-Start Checkpointing 主要通过参数 FAST_START_MTTR_TARGET 来实现。

3. FAST_START_MTTR_TARGET

FAST_START_MTTR_TARGET 参数从 Oracle 9i 开始被引入，该参数定义数据库进行 Crash 恢复的时间，单位是秒，取值范围是在 0~3600 秒之间。

在 Oracle 9i 中，Oracle 推荐设置这个参数代替 FAST_START_IO_TARGET、LOG_CHECKPOINT_TIMEOUT 及 LOG_CHECKPOINT_INTERVAL 参数。

缺省情况下，在 Oracle 9i 中，FAST_START_IO_TARGET 和 LOG_CHECKPOINT_INTERVAL 参数已经被设置为 0。

```
SQL> show parameter fast_start_io

NAME                                TYPE        VALUE
-----
fast_start_io_target                integer      0
SQL> show parameter interval

NAME                                TYPE        VALUE
-----
log_checkpoint_interval             integer      0
```

从 Oracle 9i R2 开始，Oracle 引入了一个新的视图提供 MTTR 建议：

```
SQL> select MTTR_TARGET_FOR_ESTIMATE MtrEst,
      2      ADVICE_STATUS AD,
```

```
3      DIRTY_LIMIT DL,
4      ESTD_CACHE_WRITES ESTCW,
5      ESTD_CACHE_WRITE_FACTOR EstCWF,ESTD_TOTAL_WRITES ESTW,
6      ESTD_TOTAL_WRITE_FACTOR ETWF,ESTD_TOTAL_IOS ETIO
7  from v$mtrr_target_advice;
```

MTTREST AD	DL	ESTCW	ESTCWF	ESTW	ETWF	ETIO

34 ON	1000	22610363	1.3382	86049188	1.0711	1658237817
90 ON	7157	18841862	1.1151	82280687	1.0242	1654469316
180 ON	17081	16896371	1	80335196	1	1652523825
270 ON	27005	16136315	0.955	79575140	0.9905	1651763769
360 ON	36929	15662363	0.927	79101188	0.9846	1651289817

该视图评估在不同 FAST_START_MTTR_TARGET 设置下，系统需要执行的 I/O 次数等操作。用户可以根据数据库的建议，对 FAST_START_MTTR_TARGET 进行相应调整。

这个建议信息的收集受到 Oracle 9i 新引入的初始化参数 statistics_level 的控制（关于 statistics_level 参数，将会在其他章节会进行详细介绍），当该参数设置为 Typical 或 ALL 时，MTTR 建议信息被收集：

```
SQL> show parameter statistics_level
```

NAME	TYPE	VALUE

statistics_level	string	TYPICAL

也可以通过 v\$statistics_level 视图来查询 MTTR Advice 的当前设置：

```
SQL> select * from v$statistics_level
2  where STATISTICS_NAME='MTTR Advice'
3  /
```

STATISTICS_NAME	DESCRIPTION
SESSION_STATUS	SYSTEM_STATUS
ACTIVATION_LEVEL	STATISTICS_VIEW_NAME
SESSION_SETTABLE	

MTTR Advice	Predicts the impact of different MTTR settings on number of physical I/Os
ENABLED	ENABLED
TYPICAL	V\$MTTR_TARGET_ADVICE NO

数据库当前的实例恢复状态可以通过视图 v\$instance_recovery 查询得到：

```

SQL> select RECOVERY_ESTIMATED_IOS REIO,
2          ACTUAL_REDO_BKLS ARB,
3          TARGET_REDO_BKLS TRB,
4          LOG_FILE_SIZE_REDO_BKLS LFSRB,
5          LOG_CHKPT_TIMEOUT_REDO_BKLS LCTRB,
6          LOG_CHKPT_INTERVAL_REDO_BKLS LCIRB,
7          FAST_START_IO_TARGET_REDO_BKLS FSIOTRB,
8          TARGET_MTTR TMTTR,
9          ESTIMATED_MTTR EMTTR,
10         CKPT_BLOCK_WRITES CBW
11   from v$instance_recovery;

```

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTTR	EMTTR	CBW
10138	26582	26582	184320	26582			180	108	3725700

从 v\$instance_recovery 视图，可以看到当前数据库估计的平均恢复时间（MTTR）参数：ESTIMATED_MTTR。

ESTIMATED_MTTR 的估算值是基于 Dirty Buffer 的数量和日志块数量得出的，这个参数值告诉我们，如果此时数据库崩溃，那么进行实例恢复将会需要的时间。

在 v\$instance_recovery 视图中，TARGET_MTTR 代表的是期望的平均恢复时间，通常该参数应该等于 FAST_START_MTTR_TARGET 参数设置值（但是如果 FAST_START_MTTR_TARGET 参数定义的值极大或极小，TARGET_MTTR 可能不等于 FAST_START_MTTR_TARGET 的设置）。

当 ESTIMATED_MTTR 接近或超过 FAST_START_MTTR_TARGET 参数设置（v\$instance_recovery.TARGET_MTTR）时，系统就会触发检查点，执行写出之后，系统恢复信息将会重新计算：

```

SQL> /

```

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTTR	EMTTR	CBW
7063	23728	23729	184320	23729			180	80	3725701

在繁忙的系统中，可能会观察到 ESTIMATED_MTTR>TARGET_MTTR，这可能是因为 DBWR 正忙于写出，甚或出现 Checkpoint 不能及时完成的情况。

以下案例来自一个真实的生产系统。

当执行查询时（查询脚本同前，省略之）发现 ESTIMATED_MTTR>TARGET_MTTR：

SQL> /

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTR	EMTTR	CBW
30337	614392	184320	184320	614392			180	264	3727074

继续查询，ESTIMATED_MTTR 继续升高：

SQL> /

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTR	EMTTR	CBW
24059	614392	184320	184320	614392			180	303	3727076

此时查询 v\$session_wait，发现数据库处于 checkpoint incomplete 等待：

SQL> select sid,seq#,event from v\$session_wait;

SID	SEQ# EVENT
1	41486 pmon timer
42	213 rdbms ipc reply
4	4511 rdbms ipc message
7	1209 rdbms ipc message
5	6851 rdbms ipc message
8	34701 rdbms ipc message
20	6918 log file switch (checkpoint incomplete)
2	40139 db file parallel write
3	32433 db file parallel write
6	6296 smon timer

10 rows selected

查询 V\$LOG 视图，发现除了 Current 日志组外，所有日志组都处于 Active 状态：

SQL> select * from v\$log;

GROUP#	THREAD#	SEQUENCE#	BYTES	MEMBERS	ARCHIVED	STATUS
1	1	12104	104857600	1	NO	ACTIVE

2	1	12105	104857600	1 NO	ACTIVE
3903567337657					
3	1	12106	104857600	1 NO	CURRENT
3903567340158					
4	1	12102	104857600	1 NO	ACTIVE
3903567324018					
5	1	12103	104857600	1 NO	ACTIVE
3903567326497					

此时，通过 OS 查看 iostat 状态信息，发现系统 swap 已经很严重（si、sw 较高），CPU 等待 IO（wa）也很高：

```
[root@neirong root]# vmstat 2
```

procs		memory				swap		io		system			cpu		
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
0	3	217184	18160	19680	3383068	1	0	0	4	1	0	2	0	4	1
0	4	224764	17944	4884	3401520	4	1172	3858	9144	859	1114	12	2	30	55
1	4	226060	17948	4876	3402996	20	726	4302	9762	944	1181	13	6	17	64
0	4	226436	21556	4904	3401188	16	368	3646	8782	814	1118	12	2	18	68
2	2	226828	18124	4912	3405220	16	628	1978	9294	543	840	13	1	20	66
0	3	227068	19336	4928	3404364	48	162	1174	5682	434	784	7	1	38	54
0	3	227236	21392	4928	3402748	2	324	642	1856	450	873	0	0	34	66
0	3	227236	17888	4932	3406344	0	44	770	1660	453	898	0	0	35	64
0	3	227320	17900	4952	3406516	78	118	720	1680	449	873	0	0	27	73
0	3	227488	20232	4924	3404608	16	222	784	1710	443	849	0	1	36	63
0	4	227468	17968	4992	3406640	66	56	794	1584	455	867	0	1	32	67
0	5	228036	17880	4996	3406764	314	138	1976	8388	549	898	11	2	18	69
1	4	228340	18052	5000	3407628	40	386	1340	8878	468	798	12	1	18	69
1	3	228812	18088	5028	3407476	144	394	1398	9038	448	771	11	2	17	70

这意味着系统 IO 存在瓶颈或者系统有突发的大规模写操作。

通过 CKPT_BLOCK_WRITES 字段，可以看出检查点已经写出的数据块的数量，增量检查点的触发以及 DBWR 的持续写出，都会促使该值增加，我们继续查询，可以观察到随着 CKPT_BLOCK_WRITES 的增加，ESTIMATED_MTTR 开始减少：

```
SQL> /
```

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTR	EMTTR	CBW
23132	614392	184320	184320	614392			180	238	3727077

```
SQL> /
```

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTR	EMTTR	CBW
17059	614392	184320	184320	614392			180	183	3727088

当系统完成一个检查点之后：

```
SQL> select * from v$Log;
```

GROUP#	THREAD#	SEQUENCE#	BYTES	MEMBERS	ARCHIVED	STATUS
1	1	12104	104857600	1	NO	INACTIVE
3903567335116	2006-4-28 1					
2	1	12105	104857600	1	NO	ACTIVE
3903567337657	2006-4-28 1					
3	1	12106	104857600	1	NO	ACTIVE
3903567340158	2006-4-28 1					
4	1	12107	104857600	1	NO	ACTIVE
3903567342713	2006-4-28 1					
5	1	12108	104857600	1	NO	CURRENT
3903567345267	2006-4-28 1					

可以看到 ESTIMATED_MTTR 逐渐恢复到一个较为正常的状态：

```
SQL> /
```

REIO	ARB	TRB	LFSRB	LCTRB	LCIRB	FSIOTRB	TMTR	EMTTR	CBW
13076	183735	184320	184320	468895			180	138	3727103

4. Oracle 10g 自动检查点调整

从 Oracle 10g 开始，数据库可以实现自动调整的检查点，使用自动调整的检查点，Oracle 数据库可以利用系统的低 I/O 负载时段写出内存中的脏数据，从而提高数据库的效率。因此，即使数据库管理员设置了不合理的检查点相关参数，Oracle 仍然能够通过自动调整将数据库的 Crash Recovery 时间控制在合理的范围之内。

当 FAST_START_MTTR_TARGET 参数未设置时，自动检查点调整生效。

通常，如果必须严格控制实例或节点恢复时间，那么可以设置 FAST_START_MTTR_TARGET 为期望时间值；如果恢复时间不需要严格控制，那么可以不设置 FAST_START_MTTR_TARGET 参数，从而启用 Oracle 10g 的自动检查点调整特性。

当取消 FAST_START_MTTR_TARGET 参数设置之后：

```
SQL> show parameter fast_start_mttr
```

NAME	TYPE	VALUE
fast_start_mttr_target	integer	0

在启动数据库的时候，可以从 alert 文件中看到如下信息：

Wed Jan 11 16:28:12 2006

MTTR advisory is disabled because FAST_START_MTTR_TARGET is not set

检查 v\$instance_recovery 视图，可以发现 Oracle 10g 中的改变：

```
SQL> select RECOVERY_ESTIMATED_IOS REIOS,TARGET_MTTR TMTTR,
2 ESTIMATED_MTTR EMTTR,WRITES_MTTR WMTTR,WRITES_OTHER_SETTINGS WOSET,
3 CKPT_BLOCK_WRITES CKPTBW,
4 WRITES_AUTOTUNE WAUTO,WRITES_FULL_THREAD_CKPT WFTCKPT
5 from v$instance_recovery;
```

REIOS	TMTTR	EMTTR	WMTTR	WOSET	CKPTBW	WAUTO	WFTCKPT
19407	0	68	0	0	3649819	3506125	3130700

在以上视图中，WRITES_AUTOTUNE 字段值就是指由于自动调整检查点执行的写出次数，而 CKPT_BLOCK_WRITES 指的则是由于检查点写出的 Block 的数量。

关于检查点的机制问题，我们侧重介绍了原理，至于具体的算法实现，不需要去追究过多，只要明白了这些原理性的规则，理解 Oracle 就会变成轻松的事情。

Oracle 的算法改进是一种优化，对于数据库的调整优化也不外如此，借鉴 Oracle 的优化对于理解和优化 Oracle 数据库都具有极大的好处。

5. 从控制文件获取检查点信息

在控制文件的转储中，可以看到关于检查点进程进度的记录：

```
*****
CHECKPOINT PROGRESS RECORDS
*****
(blkno = 0x4, size = 104, max = 1, in-use = 1, last-recid= 0)
THREAD #1 - status:0x2 flags:0x0 dirty:20
low cache rba:(0x10.1d6.0) on disk rba:(0x10.1e1.0)
on disk scn: 0x0000.0023af04 11/22/2004 17:04:55
resetlogs scn: 0x0000.001cff68 11/16/2004 14:10:35
heartbeat: 542912976 mount id: 3154897498
```

```

MTTR statistics status: 3
Init time: Avg: 6693744, Times measured: 3
File open time: Avg: 4446, Times measured: 19
Log block read time: Avg: 32, Times measured: 18945
Data block handling time: Avg: 2397, Times measured: 111

```

```

*****

```

这里 **low cache rba**（recovery block address）指在 Cache 中，最低的 RBA 地址，在实例恢复或者崩溃恢复中，需要从这里开始恢复。

on disk rba 是磁盘上的最高的重做值，在进行恢复时应用重做至少要达到这个值。

1.2.3 正常关闭数据库的状况

接下来看一下数据库是怎样根据 SCN 和 Checkpoint 来进行一致性判断及恢复控制的。

我们知道，在控制文件和数据文件头上，对于每个数据文件都有一个“Checkpoint SCN”和“Stop SCN”。这些 Checkpoint 和 SCN 至关重要，Oracle 通过比较这些 SCN 值来确定数据库是否需要恢复。

下面是来自一个 Clean Shutdown 的数据库的控制文件和数据文件头的内容。

因为数据库在关闭之前执行了完全检查点，所以线程检查点 SCN 和所有数据文件检查点 SCN 和数据文件 Stop SCN 都一致。

首先通过 shutdown immediate 关闭数据库，然后在 Mount 状态转储获取控制文件内容：

```

[oracle@jumper udump]$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Nov 22 13:58:06 2004

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.


Connected to:

Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production


SQL> shutdown immediate

Database closed.

Database dismounted.

ORACLE instance shut down.


SQL> startup mount;

ORACLE instance started.

```

```

Total System Global Area   80811208 bytes
Fixed Size                  451784 bytes
Variable Size              37748736 bytes
Database Buffers           41943040 bytes
Redo Buffers                667648 bytes
Database mounted.

SQL> alter session set events 'immediate trace name CONTROLF level 12';

Session altered.

SQL> @gettrcname

TRACE_FILE_NAME
-----
/opt/oracle/admin/conner/udump/conner_ora_23234.trc

[oracle@jumper udump]$ ls
conner_ora_23234.trc

```

这个 trace 文件里就记录了控制文件的详细内容。

(1) 关于数据库的相关信息。

```

*****
DATABASE ENTRY
*****
(blkno = 0x1, size = 192, max = 1, in-use = 1, last-recid= 0)
DF Version: creation=0x9200000 compatible=0x8000000, Date   10/20/2004 20:59:20
DB Name "CONNER"
Database flags = 0x00404000
Controlfile Creation Timestamp   10/20/2004 20:59:20
Incmlpt recovery scn: 0x0000.00000000
Resetlogs scn: 0x0000.001cff68 Resetlogs Timestamp   11/16/2004 14:10:35
Prior resetlogs scn: 0x0000.00000001 Prior resetlogs Timestamp   10/20/2004 20:59:20
Redo Version: creation=0x9200000 compatable=0x9200000
#Data files = 5, #Online files = 5
Database checkpoint: Thread=1 scn: 0x0000.00235d1f  --这里是检查点 SCN
Threads: #Enabled=1, #Open=0, Head=0, Tail=0
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000

```

```
Max log members = 3, Max data members = 1
Arch list: Head=0, Tail=0, Force scn: 0x0000.002179cascn: 0x0000.001caf68
Controlfile Checkpointed at scn: 0x0000.0022cb87 11/22/2004 03:02:15
thread:0 rba:(0x0.0.0)
enabled threads: 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
```

(2) Redo 信息。

```
*****
REDO THREAD RECORDS
*****
(blkno = 0x4, size = 104, max = 1, in-use = 1, last-recid= 0)
THREAD #1 - status:0xe thread links forward:0 back:0
#logs:3 first:1 last:3 current:2 last used seq#:0xf
enabled at scn: 0x0000.001cff68 11/16/2004 14:10:35
disabled at scn: 0x0000.00000000 01/01/1988 00:00:00
opened at 11/16/2004 14:10:37 by instance conner
Checkpointed at scn: 0x0000.00235d1f 11/22/2004 16:12:54 --检查点信息
thread:1 rba:(0xf.490a.10)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
log history: 14
*****
```

(3) 数据文件的检查点信息。

抽取一个数据文件的信息作为示例：

```
*****
DATA FILE #4:
(name #8) /opt/oracle/oradata/conner/eygle01.dbf
creation size=25600 block size=8192 status=0xe head=8 tail=8 dup=1
tablespace 4, index=5 krfil=4 prev_file=0
unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
Checkpoint cnt:223 scn: 0x0000.00235d1f 11/22/2004 16:12:54 --检查点 SCN
Stop scn: 0x0000.00235d1f 11/22/2004 16:12:54 --Stop SCN
Creation Checkpointed at scn: 0x0000.0005fd18 10/25/2004 23:34:32
thread:1 rba:(0x21.1f1c.10)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
```

```

00000000 00000000
Offline scn: 0x0000.001cff67 prev_range: 0
Online Checkpointed at scn: 0x0000.001cff68 11/16/2004 14:10:35
thread:1 rba:(0x1.2.0)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
Hot Backup end marker scn: 0x0000.00000000
aux_file is NOT DEFINED

```

注意这里，数据库正常关闭后，由于执行了完全检查点，数据文件处于一致的状态，检查点 SCN 在此等于 Stop SCN。

在此情况下，由于数据库处于一致状态，如果数据文件没有损失，下次启动 Oracle 能够通过验证，顺利启动。

1.2.4 数据库异常关闭的情况

再来看看数据库异常关闭的情况。

通过 shutdown abort 可以模拟一次异常，当使用 shutdown abort 方式关闭数据库时，Oracle 会立即中断所有事务，关闭当前所有数据库连接，不执行检查点，立即关闭数据库。使用这种方式关闭数据库和断点以前的故障类似，数据库在下次启动时必须执行实例恢复才能够启动。除非在特别紧急的情况下，否则通常不建议使用这种方式关闭数据库。

来看以下测试：

```

SQL> shutdown abort;
ORACLE instance shut down.
SQL> startup mount;
ORACLE instance started.

Total System Global Area 80811208 bytes
Fixed Size 451784 bytes
Variable Size 37748736 bytes
Database Buffers 41943040 bytes
Redo Buffers 667648 bytes
Database mounted.
SQL> alter session set events 'immediate trace name CONTROLF level 12';

Session altered.

SQL> @gettrcname

```

```
TRACE_FILE_NAME
```

```
-----  
/opt/oracle/admin/conner/udump/conner_ora_23353.trc
```

看看此时控制文件的内容。

1. 数据库的相关信息

在 Database Entry 部分，可以看到数据库的 Thread Checkpoint 信息：

```
*****  
DATABASE ENTRY  
*****  
  
(blkno = 0x1, size = 192, max = 1, in-use = 1, last-recid= 0)  
DF Version: creation=0x9200000 compatible=0x8000000, Date 10/20/2004 20:59:20  
DB Name "CONNER"  
Database flags = 0x00404000  
Controlfile Creation Timestamp 10/20/2004 20:59:20  
Incmlpt recovery scn: 0x0000.00000000  
Resetlogs scn: 0x0000.001cff68 Resetlogs Timestamp 11/16/2004 14:10:35  
Prior resetlogs scn: 0x0000.00000001 Prior resetlogs Timestamp 10/20/2004 20:59:20  
Redo Version: creation=0x9200000 compatable=0x9200000  
#Data files = 5, #Online files = 5  
  
Database checkpoint: Thread=1 scn: 0x0000.00235d20 --检查点信息  
Threads: #Enabled=1, #Open=1, Head=1, Tail=1  
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000  
00000000 00000000  
Max log members = 3, Max data members = 1  
Arch list: Head=0, Tail=0, Force scn: 0x0000.002179cascn: 0x0000.001caf68  
Controlfile Checkpointed at scn: 0x0000.00235d20 11/22/2004 16:39:48  
thread:0 rba:(0x0.0.0)  
enabled threads: 00000000 00000000 00000000 00000000 00000000 00000000  
00000000 00000000  
*****
```

2. 控制文件记录的 redo 信息

在控制文件中，也可以找到 REDO THREAD 的检查点信息：

```
*****  
REDO THREAD RECORDS
```

```

*****

(blkno = 0x4, size = 104, max = 1, in-use = 1, last-recid= 0)
THREAD #1 - status:0xf thread links forward:0 back:0
#logs:3 first:1 last:3 current:2 last used seq#:0xf
enabled at scn: 0x0000.001cff68 11/16/2004 14:10:35
disabled at scn: 0x0000.00000000 01/01/1988 00:00:00
opened at 11/22/2004 16:39:48 by instance conner
Checkpointed at scn: 0x0000.00235d20 11/22/2004 16:39:48
thread:1 rba:(0xf.490a.10)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
log history: 14
*****

```

3. 数据文件检查点信息

同样，以下是控制文件中记录的数据文件检查点信息：

```

DATA FILE #4:
  (name #8) /opt/oracle/oradata/conner/eygle01.dbf
creation size=25600 block size=8192 status=0xe head=8 tail=8 dup=1
tablespace 4, index=5 krfil=4 prev_file=0
unrecoverable scn: 0x0000.00000000 01/01/1988 00:00:00
Checkpoint cnt:224 scn: 0x0000.00235d20 11/22/2004 16:39:48
Stop scn: 0xffff.ffffffff 11/22/2004 16:12:54
Creation Checkpointed at scn: 0x0000.0005fd18 10/25/2004 23:34:32
thread:1 rba:(0x21.1f1c.10)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
Offline scn: 0x0000.001cff67 prev_range: 0
Online Checkpointed at scn: 0x0000.001cff68 11/16/2004 14:10:35
thread:1 rba:(0x1.2.0)
enabled threads: 01000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000
Hot Backup end marker scn: 0x0000.00000000
aux_file is NOT DEFINED

```

注意此处，由于数据库是异常关闭，数据库没有完成最后的检查点，数据文件的 Stop SCN 仍然为无穷大（**ffffffff**）。

在以上的信息中，各部分的 Checkpoint SCN 都一致，但是数据文件的 Stop SCN 不等于 Checkpoint SCN，这意味着数据库上一次关闭没有执行完全检查点，是异常关闭。此时启动数据库需要进行恢复。

4. 数据库的实例恢复

在数据库异常关闭之后，下次启动时，Oracle 会自动执行实例恢复（Instance Recovery），实例恢复包括两个步骤：Cache Recovery 和 Transaction Recovery。

继续以上的测试，在启动数据库之后可以从 alert_<sid>.log 文件中获得数据库关于恢复的相关信息：

```
alter database open
Mon Nov 22 16:54:12 2004
Beginning crash recovery of 1 threads
Mon Nov 22 16:54:12 2004
Started first pass scan
Mon Nov 22 16:54:12 2004
Completed first pass scan
148 redo blocks read, 62 data blocks need recovery
Mon Nov 22 16:54:12 2004
Started recovery at
Thread 1: logseq 15, block 18699, scn 0.0
Recovery of Online Redo Log: Thread 1 Group 2 Seq 15 Reading mem 0
Mem# 0 errs 0: /opt/oracle/oradata/conner/redo02.log
Mon Nov 22 16:54:12 2004
Completed redo application
Mon Nov 22 16:54:12 2004
Ended recovery at
Thread 1: logseq 15, block 18847, scn 0.2337845
62 data blocks read, 62 data blocks written, 148 redo blocks read
Crash recovery completed successfully
```

注意到 Oracle 在恢复过程中，首先读取日志，从最后完成的检查点开始，应用所有重做记录，这个过程叫前滚（Rolling Forward），也就是 Cache Recovery 过程，完成前滚之后，数据库可以被打开提供访问和使用。

此后进入实例恢复的第二阶段，Oracle 回滚未提交事务。Oracle 使用两个特点来增加这个恢复阶段的效率，这两个特点是 Fast-Start On-Demand Rollback 和 Fast-Start Parallel Rollback（这些特点是 Fast-Start Fault Recovery 的组成部分，仅在 Oracle 8i 之后的企业版中可用）。

使用 Fast-Start On-Demand Rollback 特点，Oracle 自动允许在数据库打开之后开始新的事务，这通常只需要很短的 Cache Recovery 时间。如果一个用户试图访问被异常中止进程锁定

的记录，Oracle 回滚那些新事物请求的记录，也就是说，因需求而回滚。因而，新事物不需要等待漫长的事务回滚时间。在 Fast-Start On-Demand Rollback 中，后台进程 SMON 充当一个调度员，使用多个服务器进程并行回滚一个事物集。

Fast-Start Parallel Rollback 主要对于长时间运行的未提交事务有效，尤其是并行 INSERT、UPDATE 和 DELETE 等操作。SMON 自动决定何时开始并行回滚并且自动在多个进程之间分散工作。

Fast-Start Parallel Rollback 的一个特殊形式是内部事务恢复 (Intra-Transaction Recovery)。在内部事务恢复中，一个大的事务可以被拆分，分配给几个服务器进程并行回滚。可以通过初始化参数 FAST_START_PARALLEL_ROLLBACK 来控制并行回滚，该参数有 3 个参数值。

- FALSE: 禁用 Fast-Start Parallel Rollback。
- LOW: 限制恢复进程不能超过 2 倍的 CPU_COUNT。
- HIGH: 限制恢复进程不能超过 4 倍的 CPU_COUNT。

1.3 深入分析

现在再深入一下，研究在数据库 Open 的过程中，Oracle 实际需要执行的操作。

1.3.1 获得数据库 Open 的跟踪文件

数据库的信息都是存放在数据文件当中的，但是当数据库尚未打开之前，Oracle 是无法获得这部分数据的。那么 Oracle 是怎样完成这个从数据文件到内存的初始化过程的呢？

```
SQL> startup mount;
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.
SQL> alter session set sql_trace = true;
Session altered.
SQL> alter database open;
Database altered.
SQL>
```

这里通过 SQL_TRACE 获得一个跟踪文件，跟踪文件里将记录从 mount 到 open 的过程中，Oracle 所执行的后台操作。

1.3.2 bootstrap\$及数据库初始化过程

通过 tkprof 格式化跟踪文件之后，来看一下其中的内容。首先来参考跟踪文件的前面部分，这是第一个对象的创建：

```
create table bootstrap$ ( line# number not null, obj#
number not null, sql_text varchar2(4000) not null) storage (initial
50K objno 56 extents (file 1 block 377))
```

在这一步骤中，实际上 Oracle 是在内存中创建 bootstrap\$ 的结构，然后从数据文件的 file 1 block 377 读取数据到内存中，完成第一次初始化。

—— 提 示 ——

file 1 block 377 子句是内部语句，该语法对用户是不可用的。

可以从数据库中查询一下，file1 block 377 上存储的是什么对象：

```
SQL> select segment_name,file_id,block_id
2 from dba_extents where block_id=377;
```

SEGMENT_NAME	FILE_ID	BLOCK_ID
-----	-----	-----
BOOTSTRAP\$	1	377

File 1 Block 377 开始存放的正是 Bootstrap\$ 对象。

接下来再看 Trace 文件中的内容，继续向下，Oracle 执行的是：

```
select line#, sql_text
from
bootstrap$ where obj# != :1
```

在创建并从数据文件中装载了 bootstrap\$ 的内容之后，Oracle 开始递归的从该表中读取信息，加载数据。那么 bootstrap\$ 中记录的是什么信息呢？

在数据库中，bootstrap\$ 是一张实际存在的系统表：

```
SQL> desc bootstrap$
Name          Null?    Type
-----
LINE#         NOT NULL NUMBER
OBJ#         NOT NULL NUMBER
SQL_TEXT      NOT NULL VARCHAR2(4000)
```

来看一下这张表的具体内容：

```
SQL> select * from bootstrap$ where line# < 5;
```

LINE#	OBJ#	SQL_TEXT
-1	-1	8.0.0.0.0
0	0	CREATE ROLLBACK SEGMENT SYSTEM STORAGE (INITIAL 112K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 32765 OBJNO 0 EXTENTS (FILE 1 BLOCK 9))
2	2	CREATE CLUSTER C_OBJ#("OBJ#" NUMBER) PCTFREE 5 PCTUSED 40 INITRANS 2 MAXTRANS 255 STORAGE (INITIAL 136K NEXT 1024K MIN EXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 2 EXTENTS (FILE 1 BLOCK 25)) SIZE 800
3	3	CREATE INDEX I_OBJ# ON CLUSTER C_OBJ# PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE (INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 3 EXTENTS (FILE 1 BLOCK 49))
4	4	CREATE TABLE TAB\$("OBJ#" NUMBER NOT NULL,"DATAOBJ#" NUMBER,"TS#" NUMBER NOT NULL,"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"BOBJ#" NUMBER,"TAB#" NUMBER,"COLS" NUMBER NOT NULL,"CLUCOLS" NUMBER,"PCTFREE\$" NUMBER NOT NULL,"PCTUSED\$" NUMBER NOT NULL,"INITRANS" NUMBER NOT NULL,"MAXTRANS" NUMBER NOT NULL,"FLAGS" NUMBER NOT NULL,"AUDIT\$" VARCHAR2(38) NOT NULL,"ROWCNT" NUMBER,"BLKCNT" NUMBER,"EMPCNT" NUMBER,"AVGSPC" NUMBER,"CHNCNT" NUMBER,"AVGRLN" NUMBER,"AVGSPC_FLB" NUMBER,"FLBLCNT" NUMBER,"ANALYZETIME" DATE,"SAMPLESIZE" NUMBER,"DEGREE" NUMBER,"INSTANCES" NUMBER,"INTCOLS" NUMBER NOT NULL,"KERNELCOLS" NUMBER NOT NULL,"PROPERTY" NUMBER NOT NULL,"TRIGFLAG" NUMBER,"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) STORAGE (OBJNO 4

TABLE 1) CLUSTER C_OBJ#(OBJ#)

以上只查询了表中的 5 条记录，大家可以自行研究一下其他记录的内容。从这些语句中可以看出，bootstrap\$中实际上是记录了一些数据库系统基本对象的创建语句。Oracle 通过 bootstrap\$进行引导，进一步创建相关的重要对象，从而启动了数据库。

1.3.3 BOOTSTRAP\$的重要性

由上面的讨论可以知道 bootstrap\$表的重要，如果 bootstrap\$表发生损坏，则数据库将无法启动。读者可能曾经遭遇以下案例，bootstrap\$表被恶意修改，如果关闭数据库，之后将无法启动。

以下测试仅为说明问题需要，请勿模仿，在未做好备份之前，请勿试验此类操作：

```
SQL> col sql_text for a15
SQL> select * from bootstrap$ where rownum <2;

      LINE#      OBJ# SQL_TEXT
-----
      -1      -1 8.0.0.0.0

SQL> update bootstrap$ set sql_text = '9.0.0.0.0' where line#=-1;

1 row updated.

SQL> commit;

Commit complete.

SQL> select * from bootstrap$ where rownum <2;

      LINE#      OBJ# SQL_TEXT
-----
      -1      -1 9.0.0.0.0
```

如果不关闭数据库，该修改是无影响的，bootstrap\$也仅在数据库启动时才发挥其重要作用。继续往下看，如果 bootstrap\$损坏或者被恶意修改，在数据库启动时会收到如下错误：

```
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-01092: ORACLE instance terminated. Disconnection forced

```

进一步检查 alert 文件，可以发现更为详细的提示信息：

```

Sat Apr 29 17:14:22 2006
Errors in file /opt/oracle/admin/conner/udump/conner_ora_31111.trc:
ORA-00704: bootstrap process failure
ORA-00702: bootstrap verison '9.0.0.0.0' inconsistent with version '8.0.0.0.0'
Sat Apr 29 17:14:22 2006
Error 704 happened during db open, shutting down database
USER: terminating instance due to error 704
Instance terminated by USER, pid = 31111
ORA-1092 signalled during: ALTER DATABASE OPEN...

```

日志给出了详细的错误提示，在这种情况下，最好的方式是从备份中进行不完全恢复，如果没有备份，则恢复将会非常复杂和艰难。

1.3.4 BBED 工具的简要介绍

—— 声 明 ——

以下将简要介绍通过 Oracle 内部工具恢复以上故障的方法，但是此方法不受 Oracle 支持，也不受我推荐，介绍这个工具仅仅为了拓展大家对于 Oracle 的认识，提供另外一种恢复的可能性。

Oracle 随软件发布一个内部工具 BBED（某些版本未包含，某些平台需要自行编译），该工具通常位于 \$ORACLE_HOME/bin 目录下，其名称为 Block Browser/Editor 的缩写。也就是说，通过 BBED 工具，可以直接打开数据文件，修改其中的内容。

以上故障中由于“8.0.0.0.0”被修改为“9.0.0.0.0”而导致数据库无法启动，可以通过 BBED 对文件进行操作，修正该数据，从而使得数据库可以重新启动。

首先需要配置两个文件，一个包含文件列表，另一个包含启动参数等：

```

[oracle@jumper conner]$ more par.bbd
blocksize=8192
listfile=file.lst
mode=edit

```

```
[oracle@jumper conner]$ more file.lst
1 /opt/oracle/oradata/conner/system01.dbf      440401920
```

通过 BBED 调用 par.bbd 参数文件进行操作：

```
[oracle@jumper conner]$ bbed parfile=par.bbd
Password:

BBED: Release 2.0.0.0.0 - Limited Production on Sat Apr 29 17:17:53 2006

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

***** !!! For Oracle Internal Use only !!! *****
```

此处 BBED 需要提供一个口令验证，这个口令目前是 *blockedit*。

通过前文可以知道 bootstrap\$ 位于 file 1 block 377，这里就是我们需要搜索的开始。

通过 find 命令可以查找被修改的字串，本文不再详细介绍命令的使用，通过搜索定位可以知道“9.0.0.0.0”起始于 file 1 blok 378 offset 1276：

```
BBED> set file 1 block 378 offset 1276

FILE#            1
BLOCK#           378
OFFSET          1276

BBED> dump
File: /opt/oracle/oradata/conner/system01.dbf (1)
Block: 378          Offsets: 1276 to 1787          Db:0x0040017a
-----
392e302e 302e302e 302e302e 303e6466 033e6466 0a31362e 302e302e 302e302c
000302c1 3202c132 c3435245 41544520 554e4951 55452049 4e444558 20495f43
4f4e3220 4f4e2043 4f4e2428 434f4e23 29205043 54465245 45203130 20494e49
5452414e 53203220 4d415854 52414e53 20323535 2053544f 52414745 20282020
.....
```

将此处的 9 修改为 8 即可：

```
BBED> modify /c "8" offset 1276
Warning: contents of previous BIFILE will be lost. Proceed? (Y/N) Y
File: /opt/oracle/oradata/conner/system01.dbf (1)
Block: 378          Offsets: 1276 to 1787          Db:0x0040017a
-----
382e302e 302e302e 302e302e 303e6466 033e6466 0a31362e 302e302e 302e302c
```

```

000302c1 3202c132 c3435245 41544520 554e4951 55452049 4e444558 20495f43
4f4e3220 4f4e2043 4f4e2428 434f4e23 29205043 54465245 45203130 20494e49
5452414e 53203220 4d415854 52414e53 20323535 2053544f 52414745 20282020
.....

```

修改过后，Oracle 会认为该块损坏：

```

BBED> verify
DBVERIFY - Verification starting
FILE = /opt/oracle/oradata/conner/system01.dbf
BLOCK = 378

Block 378 is corrupt
***

Corrupt block relative dba: 0x0040017a (file 0, block 378)
Bad check value found during verification
Data in bad block -
  type: 6 format: 2 rdba: 0x0040017a
  last change scn: 0x0819.004c0295 seq: 0x1 flg: 0x06
  consistency value in tail: 0x02950601
  check value in block header: 0x9f21, computed block checksum: 0x1
  spare1: 0x0, spare2: 0x0, spare3: 0x0
***

DBVERIFY - Verification complete

Total Blocks Examined          : 1
Total Blocks Processed (Data) : 0
Total Blocks Failing   (Data) : 0
Total Blocks Processed (Index): 0
Total Blocks Failing   (Index): 0
Total Blocks Empty              : 0
Total Blocks Marked Corrupt     : 1
Total Blocks Influx             : 0

```

重新计算和应用校验位后，数据块可以恢复一致：

```

BBED> sum apply
Check value for File 1, Block 378:
current = 0x9f20, required = 0x9f20

```

```
BBED> verify
DBVERIFY - Verification starting
FILE = /opt/oracle/oradata/conner/system01.dbf
BLOCK = 378

DBVERIFY - Verification complete

Total Blocks Examined          : 1
Total Blocks Processed (Data) : 1
Total Blocks Failing   (Data) : 0
Total Blocks Processed (Index): 0
Total Blocks Failing   (Index): 0
Total Blocks Empty              : 0
Total Blocks Marked Corrupt    : 0
Total Blocks Influx            : 0

BBED>
```

此后数据库可以正常启动。

注 意

此处能够正常恢复，是因为从 8 至 9 的修改并为改变数据位数，否则 BBED 也很难奏效。此处的介绍仅为扩展大家的视野，使用该工具需要极为谨慎，并且自行负责。

参考信息与建议阅读

- | | |
|---|---|
| (1) Oracle(R) Database Administrator's Guide 10g Release 2 (10.2) | B14231-01 |
| (2) Oracle(R) Database Concepts 10g Release 2 (10.2) | B14220-02 |
| (3) Biti_rainy | http://www.itpub.net/199099.html |
-

第2章 参数及参数文件

在 Oracle 数据库中，有一系列的初始化参数用来进行数据库约束和资源限制，这些参数通常存储在一个参数文件中，在数据库实例启动时读取并加载。

初始化参数对数据库来说非常重要，很多参数通过合理的调整可以极大地提高数据库性能。本章对初始化参数和参数文件进行相关探讨。

2.1 初始化参数的分类

按照得出方式不同，初始化参数可以分为 3 类：推导参数、操作系统依赖参数和可变参数。

2.1.1 推导参数 (Derived Parameters)

推导参数通常来自于其他参数的运算，依赖其他参数得出。所以这类参数通常不需要修改。如果强制修改，那么修改值会覆盖推导值。

常见的此类参数有很多，例如，SESSIONS 参数，在 Oracle 文档中，该参数按以下公式运算得出：

$$SESSIONS=(1.1\times PROCESSES)+5$$

缺省情况下，当 PROCESSES 被修改时，此参数会自动计算并生效。

2.1.2 操作系统依赖参数

某些参数的有效值或者取值范围依赖或者受限于操作系统，如 db_cache_size 参数，设置 Oracle 使用的内存大小，该参数的最大值就要受限于物理内存。这一类参数通常被称为操作系统依赖参数。

2.1.3 可变参数

可变参数通常可以调整，有些设置的是限制条件，如 OPEN_CURSORS；有的参数是设

置容量，如 `DB_CACHE_SIZE` 等。这类参数通常可以为 DBA 或最终用户调整，从而产生限制或性能变化，对 Oracle 至关重要。

初始化参数通常还有一些其他分类方式。

- 按照修改方式划分，初始化参数又可以分为静态参数和动态参数。

静态参数只能在参数文件中修改，在重新启动后方能生效；动态参数可以动态调整，调整后通常可以立即生效。

- 按照获取方式不同，初始化参数又可以分为显示参数和隐含参数。

显示参数可以通过 `v$parameter` 查询得到；而隐含参数通常以 “_” 开头，必须通过查询系统表方能获得这些参数。

总之，虽然分类方式不同，但是参数都是这些，我们更多需要了解的是这些参数的用途。

2.1.4 初始化参数的获取

Oracle 的初始化参数可以通过 `V$PARAMETER` 视图查询得到，在 `SQL*PLUS` 之中，经常可以通过 `show parameter` 命令来显示某些参数的设置值，例如：

```
[oracle@danaly ~]$ sqlplus "/ as sysdba"
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Mar 7 15:27:52 2006
Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options

SQL> show parameter sga
```

NAME	TYPE	VALUE
lock_sga	boolean	FALSE
pre_page_sga	boolean	FALSE
sga_max_size	big integer	900M
sga_target	big integer	900M

通过 `sql_trace` 的跟踪，可以发现，其实这条命令的本质是通过如下一条 SQL 查询得到的：

```
SELECT NAME NAME_COL_PLUS_SHOW_PARAM,DECODE(TYPE,1,'boolean',2,'string',3,
'integer',4,'file',5,'number',6,'big integer', 'unknown') TYPE,
DISPLAY_VALUE VALUE_COL_PLUS_SHOW_PARAM
FROM
```

```
V$PARAMETER WHERE UPPER(NAME) LIKE UPPER('%sga%') ORDER BY
NAME_COL_PLUS_SHOW_PARAM,ROWNUM
```

通过 V\$PARAMETER 视图的创建语句，可以观察到，实际上 V\$PARAMETER 视图过滤掉了以 “_” 开头的一系列参数：

```
SELECT x.inst_id, x.indx + 1, ksppinm, ksppity, ksppstvl, ksppstdf,
       DECODE (BITAND (ksppiflg / 256, 1), 1, 'TRUE', 'FALSE'),
       DECODE (BITAND (ksppiflg / 65536, 3),
               1, 'IMMEDIATE',
               2, 'DEFERRED',
               3, 'IMMEDIATE',
               'FALSE'
       ),
       DECODE (BITAND (ksppstvf, 7), 1, 'MODIFIED', 4, 'SYSTEM_MOD', 'FALSE'),
       DECODE (BITAND (ksppstvf, 2), 2, 'TRUE', 'FALSE'), ksppdesc,
       ksppstcmnt
FROM x$ksppi x, x$ksppcv y
WHERE (x.indx = y.indx)
      AND ((TRANSLATE (ksppinm, '_', '#') NOT LIKE '#%') OR (ksppstdf = 'FALSE'))
      )
```

这些以 “_” 开头的初始化参数通常被称为隐含参数，Oracle 通常不建议修改这些参数，但是因为某些隐含参数有着特殊的功能，逐渐被越来越多的人所熟知。

通过以下查询，可以获得这些隐含参数：

```
set linesize 132
column name format a30
column value format a25
select
  x.ksppinm  name,
  y.ksppstvl value,
  y.ksppstdf isdefault,
  decode(bitand(y.ksppstvf,7),1,'MODIFIED',4,'SYSTEM_MOD','FALSE') ismod,
  decode(bitand(y.ksppstvf,2),2,'TRUE','FALSE') isadj
from
  sys.x$ksppi x,
  sys.x$ksppcv y
where
  x.inst_id = userenv('Instance') and
  y.inst_id = userenv('Instance') and
```

```

x.indx = y.indx and
x.kspinm like '%_&par%'
order by
translate(x.kspinm, '_', ' ')
/

```

常用的几个隐含参数有：

NAME	VALUE	ISDEFAULT	ISMOD	ISADJ
-----	-----	-----	-----	-----
_allow_resetlogs_corruption	FALSE	TRUE	FALSE	FALSE
_corrupted_rollback_segments		TRUE	FALSE	FALSE
_offline_rollback_segments		TRUE	FALSE	FALSE

后面的章节中会介绍这几个参数的重要用途。

2.2 参数文件

参数文件是一个包含一系列参数及参数对应值的操作系统文件，它有以下两种类型：

- 初始化参数文件（Initialization Parameters Files）：Oracle 9i 之前 Oracle 一直采用 pfile 方式存储初始化参数，该文件为文本文件；
- 服务器参数文件（Server Parameter Files）：从 Oracle 9i 开始，Oracle 引入的 spfile 文件，该文件为二进制格式。

从操作系统上，也可以看出这两者的区别：

```

[oracle@jumper oracle]$ cd $ORACLE_HOME/dbs
[oracle@jumper dbs]$ file initconner.ora
initconner.ora: ASCII text
[oracle@jumper dbs]$ file spfileconner.ora
spfileconner.ora: data

```

在 9i 以前，Oracle 使用 pfile 存储初始化参数设置，参数文件的修改需要手工进行，这些参数在实例启动时被读取，通过 pfile 的修改需要重起实例才能生效；从 Oracle 9i 开始，Oracle 引入 spfile 文件，使用 spfile 用户可以通过 ALTER SYSTEM 或者 ALTER SESSION 来修改参数，而不再需要通过手工修改。对于动态参数，所有更改可以立即生效，同时用户可以选择使更改只应用于当前实例还是同时应用到 spfile，对于静态参数，只能将变更应用到 spfile 文件，这些变更在数据库重启后生效。

这就使得所有对 spfile 的修改都可以通过命令行完成，我们可以彻底告别手工修改初始化参数文件的历史，这就大大减少了人为错误的发生。

另外 SPFILE 是一个二进制文件，可以使用 RMAN 进行备份，这样实际上 Oracle 把参数文件也纳入了 Oracle 的备份恢复体系。

2.2.1 PFILE 和 SPFILE

除了第一次启动数据库需要 pfile（然后可以根据 pfile 创建 spfile），用户可以不再需要 pfile，Oracle 强烈推荐使用 spfile，应用其新特性来存储和维护初始化参数设置。

当使用 DBCA 自定义（不使用模版）创建数据库时，在最后一个步骤，可以选择“生成数据库创建脚本”复选框，如图 2-1 所示，通过这个脚本，可以研究 Oracle 是怎样创建的数据库，也可以通过脚本执行，按步骤地手工创建数据库。



图 2-1 选择“生成数据库创建脚本”

以 Windows 为例，在 scripts 目录下，通常可以看到这样一些脚本（根据安装选项不同，脚本可能不同）：

```
C:\oracle\admin\eygle\scripts>dir

驱动器 C 中的卷是 SYSTEM
卷的序列号是 1CE0-895C

C:\oracle\admin\eygle\scripts 的目录

2005-01-06  13:33    <DIR>          .
2005-01-06  13:33    <DIR>          ..
2005-01-06  13:23                918 CreateDB.sql
```

2005-01-06	13:23	631 CreateDBCatalog.sql
2005-01-06	13:23	134 CreateDBFiles.sql
2005-01-06	13:23	781 eygle.bat
2005-01-06	13:23	2,847 init.ora
2005-01-06	13:24	409 postDBCcreation.sql
	6 个文件	5,720 字节
	2 个目录	470,794,240 可用字节

手工创建过程通常可以通过 eygle.bat 批处理文件执行开始，系统会根据脚本自动执行创建过程。

我不打算过多地介绍数据库创建过程，与本章内容有关的是，这里存在一个 **init.ora** 文件（或 init.ora.<时间戳>文件），这个文件是根据用户创建数据库之前定义的参数自动生成的，该参数被用来在创建过程中启动数据库，通过 CreateDB.sql 可以看到这个引用：

```
connect SYS/change_on_install as SYSDBA

set echo on

spool C:\oracle\ora92\assistants\dbca\logs\CreateDB.log

startup nomount pfile="C:\oracle\admin\eygle\scripts\init.ora";

CREATE DATABASE eygle
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 5
MAXLOGMEMBERS 3
MAXDATAFILES 100
DATAFILE 'd:\oradata\eygle\system01.dbf' SIZE 250M REUSE AUTOEXTEND ON NEXT 10240K
MAXSIZE UNLIMITED
EXTENT MANAGEMENT LOCAL
DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE 'd:\oradata\eygle\temp01.dbf' SIZE 40M
REUSE AUTOEXTEND ON NEXT 640K MAXSIZE UNLIMITED
UNDO TABLESPACE "UNDOTBS1" DATAFILE 'd:\oradata\eygle\undotbs01.dbf' SIZE 200M REUSE
AUTOEXTEND ON NEXT 5120K MAXSIZE UNLIMITED
CHARACTER SET ZHS16GBK
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1 ('d:\oradata\eygle\redo01.log') SIZE 10240K,
GROUP 2 ('d:\oradata\eygle\redo02.log') SIZE 10240K,
GROUP 3 ('d:\oradata\eygle\redo03.log') SIZE 10240K;

spool off

exit;
```

在数据库创建完成之后，Oracle 调用 postDBCreation.sql 脚本来进行一系列的后续处理，最后 Oracle 通过 init.ora 文件创建了 spfile 文件，该脚本的内容大致如下：

```
connect SYS/change_on_install as SYSDBA
set echo on
spool C:\oracle\ora92\assistants\dbca\logs\postDBCreation.log
@C:\oracle\ora92\rdbms\admin\utlrp.sql;
shutdown ;
connect SYS/change_on_install as SYSDBA
set echo on
spool C:\oracle\ora92\assistants\dbca\logs\postDBCreation.log
      create spfile='C:\oracle\ora92\database\spfileeygle.ora' FROM pfile= 'C:\oracle\
admin\eygle\scripts\init.ora';
startup ;
```

这就是从 Oracle 9i 开始的 pfile 和 spfile 的交接。

建议每个试图深入学习 Oracle 的用户都仔细研究一下自动建库的脚本，深入了解该过程非常有助于深入学习 Oracle。

2.2.2 SPFILE 的创建

从 Oracle 9i 开始，缺省情况下，Oracle 使用 spfile 启动数据库，从上一节的数据库创建过程也可以看到，spfile 必须由 pfile 创建，新创建的 spfile 在下次启动数据库时生效。

CREATE SPFILE 需要 SYSDBA 或者 SYSOPER 的权限，具体语法如下：

```
CREATE SPFILE[='SPFILE-NAME'] FROM PFILE[='PFILE-NAME']
```

例如：

```
SQL> create spfile from pfile;
```

缺省情况下，spfile 创建到以下系统缺省目录。

- 对于 UNIX，目录为 \$ORACLE_HOME/dbs；
- 对于 NT，目录为 \$ORACLE_HOME/database。

如果 SPFILE 已经存在，那么创建会返回以下错误：

```
SQL> create spfile from pfile;
```

```
create spfile from pfile
```

```
*
```

```
ERROR 位于第 1 行:
```

```
ORA-32002: 无法创建已由例程使用的 SPFILE
```

这也可以用来判断当前是否使用了 spfile 文件。

随着 spfile 的引入，Oracle 同时引入了一个视图用以记录 spfile 的参数设置信息，这个视图是 v\$spparameter。

然而意外的是，Oracle 并没有向其他文件一样，在运行期间保持锁定，让我们做以下试验：

```
SQL> host rename SPFILEEYGLEN.ORA SPFILEEYGLEN.ORA.BAK
```

```
SQL> alter system set db_cache_size=24M scope=both;
```

系统已更改。

```
SQL> host dir *.ora
```

驱动器 E 中的卷是 Doc

卷的序列号是 980C-8EFF

E:\Oracle\Ora9iR2\database 的目录

```
2003-02-10  14:35                2,048 PWDEyglen.ORA
```

```
                1 个文件                2,048 字节
```

```
                0 个目录    150,347,776 可用字节
```

```
SQL> alter system set db_cache_size=24M scope=spfile;
```

```
alter system set db_cache_size=24M scope=spfile
```

*

ERROR 位于第 1 行:

ORA-27041: 无法打开文件

OSD-04002: 无法打开文件

O/S-Error: (OS 2) 系统找不到指定的文件。

```
SQL> host rename SPFILEEYGLEN.ORA.BAK SPFILEEYGLEN.ORA
```

```
SQL> alter system set db_cache_size=24M scope=spfile;
```

系统已更改。

```
SQL>
```

由于运行期并不锁定 spfile，所以 spfile 可能会意外丢失，如果发生此类情况，Oracle 会不允许使用 create spfile from pfile 缺省命令来重建 spfile（因 ORA-32002 错误而失败），通常可以创建一个自定义名称的 spfile 文件，然后重命名为缺省名称即可。

2.2.3 SPFILE 的搜索顺序

重新启动数据库，使用 startup 命令，Oracle 将会按照以下顺序在缺省目录中搜索参数文件。

(1) spfile<ORACLE_SID>.ora, 其缺省目录如下。

- UNIX: \$ORACLE_HOME/dbs/
- NT: %ORACLE_HOME%\database

(2) spfile.ora, 其缺省目录如下。

- UNIX: \$ORACLE_HOME/dbs/
- NT: %ORACLE_HOME%\database

(3) init<ORACLE_SID>.ora, 其缺省目录如下。

- UNIX: \$ORACLE_HOME/dbs/
- NT: %ORACLE_HOME%\database

创建了 spfile, 重新启动数据库, Oracle 会按顺序搜索以上目录, spfile 就会自动生效。

2.2.4 使用 PFILE/SPFILE 启动数据库

如果想使用 pfile 启动数据库, 则可以在启动时指定 pfile 或者删除 spfile:

```
SQL> startup pfile='E:\Oracle\admin\eyglen\pfile\init.ora';
```

不能以同样的方式指定 spfile, 但是可以创建一个包含 spfile 参数的 pfile 文件, 指向 spfile。spfile 是一个自 Oracle 9i 引入的初始化参数, 类似于 IFILE 参数。spfile 参数用于定义非缺省路径的 spfile 文件。

可以在 pfile 链接到 spfile 文件, 同时在 pfile 中定义其他参数, 如果参数重复设置, 后读取的参数将取代先前的设置。

看一下以下例子, 当前使用 spfile 启动数据库, log_archive_start 参数设置为 True:

```
SQL> show parameter log_archive_start
```

NAME	TYPE	VALUE
log_archive_start	boolean	TRUE

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE
spfile	string	%ORACLE_HOME%\DATABASE\SPFILE%ORACLE_SID%.ORA

```
SQL>
```

修改 pfile 文件内容如下:

```
#Pfile link to SPFILE
SPFILE= 'E:\Oracle\Ora9iR2\database\SPFILEEYGLEN.ORA'
log_archive_start = false
```

可以预见这个 log_archive_start 参数设置将会代替 spfile 中的设置:

```
SQL> startup pfile='e:\initeyglen.ora'
```

ORACLE 例程已经启动。

```
Total System Global Area  135338868 bytes
Fixed Size                  453492 bytes
Variable Size               109051904 bytes
Database Buffers            25165824 bytes
Redo Buffers                 667648 bytes
```

数据库装载完毕。

数据库已经打开。

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE

spfile	string	E:\Oracle\Ora9iR2\database\SPFILEEYGLEN.ORA

```
SQL> show parameter log_archive_start
```

NAME	TYPE	VALUE

log_archive_start	boolean	FALSE

这就是 spfile 与 pfile 结合使用的一些技巧。

这样的用法其实在 RAC 的系统中非常常见，由于在 RAC 中，用户通常需要把 spfile 存储在共享磁盘上，所以常规的做法就是通过定义 pfile 文件，在 pfile 文件中对 spfile 文件进行重定向，下面是 RAC 环境中一个参数文件的设置范例：

```
[oracle@raclinux1 ~]$ cd $ORACLE_HOME/dbs
[oracle@raclinux1 dbs]$ more initRACDB1.ora
SPFILE='+MY_DG2/RACDB/spfileRACDB.ora'
```

在数据库启动之后，当然可以使用 ALTER SYSTEM 方式将参数修改直接固化到 spfile 文件中：

```
SQL> alter system set log_archive_start=false scope=spfile;
```

系统已更改。

提示

通过在 pfile 中调用 spfile，使用后设置的参数覆盖 spfile 中的参数设置，是解决 spfile 中参数设置错误的一种方法。

2.2.5 修改参数

可以通过 ALTER SYSTEM 或者导入导出来更改 spfile 的内容。

从 Oracle 9i 开始，ALTER SYSTEM 命令增加了一个新的选项 scope。scope 参数有 3 个可选值：memory、spfile 和 both。

- memory: 只改变当前实例运行，重新启动数据库后失效。
- spfile: 只改变 spfile 的设置，不改变当前实例运行，重新启动数据库后生效
- both: 同时改变实例及 spfile，当前更改立即生效，重新启动数据库后仍然有效。

针对 RAC 环境，ALTER SYSTEM 还可以指定 SID 参数，对不同实例进行不同设置。

所以通过 spfile 修改参数的完整命令如下：

```
alter system set <parameter_name> =<value> scope = memory|spfile|both [sid=<sid_name>]
```

下面通过简单的例子来看一下 scope 参数的几个用法。

1. scope=memory

修改当前实例的 db_cache_advice 参数为 OFF:

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	ON

```
SQL> alter system set db_cache_advice=off scope=memory;
```

System altered.

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	OFF

如果观察 alert_<sid>.log 文件，可以发现其中记录了如下一行：

```
Wed Apr 26 21:18:57 2006
```

```
ALTER SYSTEM SET db_cache_advice='OFF' SCOPE=MEMORY;
```

如果重新启动数据库，这个更改将会丢失：

```
SQL> startup force;
```

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.

Database opened.

SQL> show parameter db_cache_ad

```

NAME	TYPE	VALUE
db_cache_advice	string	ON

也就是说 `scope=memory` 的修改影响，不会跨越一次数据库的重新启动。

2. `scope=spfile`

当指定 `scope=spfile` 时，当前实例运行不受影响：

```
SQL> alter system set db_cache_advice=off scope=spfile;
```

```
System altered.
```

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	ON

同样可以从 `alert_<sid>.log` 文件中看到这个修改：

```
Wed Apr 26 21:24:02 2006
```

```
ALTER SYSTEM SET db_cache_advice='OFF' SCOPE=SPFILE;
```

这个修改将在下次数据库启动后生效：

```
SQL> startup force;
```

```
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.
```

```
Database opened.
```

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	OFF

但是需要知道的是，对于静态参数，只能指定 `scope=spfile` 进行修改。

通过 `scope=spfile` 修改的参数，虽然对当前实例无效，但是其参数值可以从 `v$spparameter` 视图中查询得到：

```
SQL> show parameter db_cache_advice
```

NAME	TYPE	VALUE
db_cache_advice	string	OFF

```
SQL> alter system set db_cache_advice=on scope=spfile;
```

```
System altered.
```

```
SQL> select name,value from v$spparameter
```

```
2  where name='db_cache_advice';
```

NAME	VALUE
db_cache_advice	ON

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	OFF

3. scope = both

使用 `both` 选项实际上等同于不带参数的 `ALTER SYSTEM` 语句。

```
SQL> alter system set db_cache_advice=off scope=both;
```

System altered.

```
SQL> alter system set db_cache_advice=off;
```

System altered.

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	OFF

在 alert_<sid>.log 文件中，可以看到如下信息：

Wed Apr 26 21:28:21 2006

```
ALTER SYSTEM SET db_cache_advice='OFF' SCOPE=BOTH;
```

Wed Apr 26 21:28:28 2006

```
ALTER SYSTEM SET db_cache_advice='OFF' SCOPE=BOTH;
```

注意不带 scope 参数和 scope=both 实际上是等价的。如果修改静态参数，那么需要指定 spfile 参数，不能指定 both 参数，否则数据库将会报错。

```
SQL> ALTER SYSTEM SET sql_trace=FALSE SCOPE=BOTH;
```

```
ALTER SYSTEM SET sql_trace=FALSE SCOPE=BOTH
```

*

ERROR 位于第 1 行:

ORA-02095: 无法修改指定的初始化参数

```
SQL> ALTER SYSTEM SET sql_trace=FALSE SCOPE=SPFILE;
```

系统已更改。

注 意

在 Oracle 10g 中，sql_trace 已经变为了一个动态参数。

4. RAC 环境中的修改

在 RAC 环境中，如果不指定 SID 名称，或者指定为 “*”，那么修改缺省的对所有实例生效，例如：

```
ALTER SYSTEM SET OPEN_CURSORS=500 SID='*' SCOPE=MEMORY;
```

如果需要修改指定的实例，则需要设置相应的 SID 参数，例如：

```
SQL> select sid,name,value from v$pparameter
```

```
2 where name='open_cursors';
```

SID	NAME	VALUE

*	open_cursors	300

```
SQL> alter system set open_cursors=150 scope=spfile sid='RACDB1';
```

System altered.

```
SQL> select sid,name,value from v$spparameter
2  where name='open_cursors';
```

SID	NAME	VALUE

*	open_cursors	300
RACDB1	open_cursors	150

需要注意的是，在 RAC 环境中，不同实例的 `undo_tablespace` 设置是不同的，当修改一个实例的 Undo 表空间时，一定要注意指定相应的实例，以避免修改错误：

```
SQL> select sid,name,value from v$spparameter
2  where name='undo_tablespace';
```

SID	NAME	VALUE

RACDB1	undo_tablespace	UNDOTBS1
RACDB2	undo_tablespace	UNDOTBS2

5. 在关闭数据库状态修改 spfile

可以在数据库 shutdown 时创建和修改 spfile，例如：

```
SQL> shutdown immediate
数据库已经关闭。
已经卸载数据库。
ORACLE 例程已经关闭。
SQL> create pfile from spfile;
文件已创建。
SQL> create spfile from pfile;
文件已创建。
SQL>
```

所以如果不慎错误地修改了参数导致数据库无法启动时，可以通过创建 `pfile` 文件，修改其中的参数，再由 `pfile` 创建 `spfile` 的方式解决，最后由 `spfile` 正常启动数据库。

例如，如下设置了 `db_block_buffers` 参数之后，数据库在下次启动时将会出错，因为该参数与 `db_cache_size` 不兼容：

```
SQL> alter system set db_block_buffers=1000 scope=spfile;

System altered.

SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup
ORA-00381: cannot use both new and old parameters for buffer cache size specification
```

此时可以由 `spfile` 创建 `pfile` 文件：

```
SQL> create pfile from spfile;

File created.
```

然后修改参数文件，删除其中的 `db_block_buffers` 参数：

```
*.db_block_buffers=1000
```

然后由 `pfile` 创建 `spfile` 启动数据库：

```
SQL> create spfile from pfile;

File created.

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.
Database opened.
```

`spfile` 的修改和使用方式，是一定要熟练的。

提示

这是修改 spfile 的第二种方式, 通过这种方式, 我们可以快速修正 spfile 中的错误参数定义。

在 spfile 引入之初, 很多人因为其使用和修改复杂而拒绝使用 spfile, 仍然沿用 pfile 文件, 其实不要小看 spfile 文件, spfile 可以在数据库中通过命令动态修改的特性, 是 Oracle 10g 中很多自动化特性的实现基础。

应当熟悉这样一个事实, Oracle 经常在现行版本中为下一版本做准备, 并优先推出部分功能, 这些功能因为其超前可能显得不够实用, 而当这些特性在新版本中再次出现时, 我们才忽然知道, 这些特性原来是如此的不可缺少。

下面以 Oracle 10g 的自动共享内存调整特性(具体内容将在后面章节详细介绍)来做一个简单说明。当在 Oracle 10g 中设置了 SGA_TARGET 参数启用了自动 SGA 调整之后, Oracle 会同时启用一系列的新的隐含参数来控制 SGA 各组件的大小。如果足够细心, 大家可能从 alert_<sid>.log 文件中注意到, 每次启动这些参数的设置通常都是不同的, 下面从生产环境中摘录两个片段给大家参考。

第一个启动信息:

```
Thu Jan 19 14:38:43 2006
Starting ORACLE instance (normal)
.....
LICENSE_MAX_USERS = 0
SYS auditing is disabled
Starting up ORACLE RDBMS Version: 10.2.0.1.0.
System parameters with non-default values:
  processes                = 150
  __shared_pool_size        = 75497472
  __large_pool_size         = 4194304
  __java_pool_size          = 4194304
  __streams_pool_size       = 0
  spfile                    = +ORADG/danaly/spfiledanaly.ora
  sga_target                 = 943718400
.....
  db_block_checksum         = FULL
  db_block_size              = 8192
  __db_cache_size           = 851443712
```

第二个启动信息:

```
Wed Apr 5 12:01:02 2006
Starting ORACLE instance (normal)
```

```

.....
LICENSE_MAX_USERS = 0
SYS auditing is disabled
ksdpec: called for event 13740 prior to event group initialization
Starting up ORACLE RDBMS Version: 10.2.0.1.0.
System parameters with non-default values:
  processes                = 150
  __shared_pool_size        = 113246208
  __large_pool_size         = 4194304
  __java_pool_size          = 12582912
  __streams_pool_size       = 0
  spfile                    = +ORADG/danely/spfiledanaly.ora
  sga_target                = 943718400
  .....
  db_block_checksum         = FULL
  db_block_size             = 8192
  __db_cache_size           = 805306368

```

这些参数的不同就是 Oracle 自动调整的结果，通过 spfile 的动态修改，这些参数值可以跨越数据库重新启动而继续生效。

—— 提 示 ——

由于 spfile 是一个二进制文件，所以不能通过手工方式修改，很多朋友通过手工修改而导致 spfile 损坏，使得该 spfile 不能用来启动数据库。我们要引以为戒。

2.2.6 重置 SPFILE 中设置的参数

虽然并不常用，但是 Oracle 仍然提供了重置参数的方法。当想恢复某个参数为缺省值时，可以使用如下命令：

```
alter system reset parameter <scop=memory|spfile|both> sid='sid|*'
```

该命令通常用于 RAC 环境中，在单实例环境中，需要指定 sid='*'，reset 一个参数，Oracle 将从 spfile 文件中去除该参数：

```

[oracle@jumper dbs]$ strings spfileconner.ora |grep open
*open_cursors=150
[oracle@jumper dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Tue May 9 11:19:45 2006
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

```

```

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production

SQL> alter system reset open_cursors scope=spfile sid='*';

System altered.

SQL> exit
Disconnected from Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production
[oracle@jumper dbs]$ strings spfileconner.ora |grep open

```

可以看到，reset 之后 open_cursors 参数在 spfile 文件中不再存在。

2.2.7 是否使用了 SPFILE

判断是否使用了 spfile，可以使用以下几种方法。

(1) 查询 v\$parameter 动态视图，如果以下查询返回空值，那么你在使用 pfile:

```

SQL> SELECT name,value FROM v$parameter WHERE name='spfile';

NAME                                VALUE
-----
spfile                             ?/dbs/spfile@.ora

```

(2) 或者可以使用 show 命令（实际上，show 命令的结果同样来自 v\$parameter 视图）来显示参数设置，如果以下结果 value 列返回空值，那么说明你在使用 pfile:

```

SQL> show parameter spfile

NAME                                TYPE      VALUE
-----
spfile                             string    ?/dbs/spfile@.ora
SQL>

```

(3) 查询 v\$spparameter 视图，如果以下查询返回 0 值，表示你在使用 pfile，否则表明你使用的是 spfile:

```
SQL> SELECT COUNT(*) FROM v$spparameter WHERE value IS NOT NULL;

COUNT(*)
-----
        32
```

或者使用以下查询，如果 **true** 值返回非 0 值，那么说明使用的是 **spfile**：

```
SQL> select isspecified, count(*) from v$spparameter group
2  by isspecified;

ISSPECIFIED    COUNT(*)
-----
FALSE          226
TRUE           33
```

更为直接的：

```
SQL> select decode(count(*), 1, 'spfile', 'pfile') USED
2  from v$spparameter
3  where rownum=1 and isspecified='TRUE'
4  /

USED
-----
spfile
```

2.2.8 SPFILE 的备份与恢复

在本章开头提到，Oracle 把 **spfile** 也纳入到 **RMAN** 的备份恢复策略当中，如果配置了控制文件自动备份（**autoback**），那么 Oracle 会在数据库发生重大变化（如增减表空间）时自动进行控制文件及 **spfile** 文件的备份。

下面来看一下这个过程。

（1）设置控制文件自动备份：

```
[oracle@jumper oracle]$ rman target /

Recovery Manager: Release 9.2.0.3.0 - Production

Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.

connected to target database: HSJF (DBID=1052178311)
```

```

RMAN> CONFIGURE CONTROLFILE AUTOBACKUP ON;

```

using target database controlfile instead of recovery catalog

old RMAN configuration parameters:

```

CONFIGURE CONTROLFILE AUTOBACKUP OFF;

```

new RMAN configuration parameters:

```

CONFIGURE CONTROLFILE AUTOBACKUP ON;

```

new RMAN configuration parameters are successfully stored

```

RMAN> exit

```

这个设置可以在数据库中通过如下方式查询得到:

```

[oracle@jumper bdump]$ sqlplus "/ as sysdba"

```

```

SQL*Plus: Release 9.2.0.3.0 - Production on Sat Jan 17 01:08:05 2004

```

```

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

```

```

Connected to:

```

```

Oracle9i Enterprise Edition Release 9.2.0.3.0 - Production

```

```

With the Partitioning, OLAP and Oracle Data Mining options

```

```

JServer Release 9.2.0.3.0 - Production

```

```

SQL> select * from v$rman_configuration;

```

CONF#	NAME	VALUE
1	CONTROLFILE AUTOBACKUP	ON

(2) 记录数据库变化:

```

SQL> create tablespace eygle

```

```

2  datafile '/data1/oracle/oradata/eygle01.dbf'

```

```

3  size 5M;

```

```

Tablespace created.

```

如果新创建一个表空间, 这时候检查 alert<sid>.log 文件, 可以在其中发现这样的备份信息:

```

Sat Jan 17 00:55:57 2004

```

```
Starting control autobackup
Control autobackup written to DISK device
      handle '/opt/oracle/product/9.2.0/dbs/c-1052178311-20040117-00'
Completed: create tablespace eygle
datafile '/data1/oracle/oradata/eygle01.dbf'
```

如果使用 RMAN 进行备份，在提示中可以看到如下信息：

```
RMAN> configure controlfile autobackup on;

old RMAN configuration parameters:
CONFIGURE CONTROLFILE AUTOBACKUP OFF;
new RMAN configuration parameters:
CONFIGURE CONTROLFILE AUTOBACKUP ON;
new RMAN configuration parameters are successfully stored


RMAN> run
2> {
3> allocate channel ch1 type disk format='e:\oracle\orabak\penny%t.arc';
4> backup archivelog all delete all input;
5> release channel ch1;
6> }

allocated channel: ch1
channel ch1: sid=13 devtype=DISK


Starting backup at 02-DEC-03
current log archived
channel ch1: starting archive log backupset
channel ch1: specifying archive log(s) in backup set
input archive log thread=1 sequence=63 recid=168 stamp=511712617
input archive log thread=1 sequence=64 recid=169 stamp=511712620
input archive log thread=1 sequence=65 recid=170 stamp=511712626
input archive log thread=1 sequence=66 recid=171 stamp=511712690
channel ch1: starting piece 1 at 02-DEC-03
channel ch1: finished piece 1 at 02-DEC-03
piece handle=E:\ORACLE\ORABAK\PENNY511712693.ARC comment=NONE
channel ch1: backup set complete, elapsed time: 00:00:03
```

```

channel ch1: deleting archive log(s)

archive log filename=E:\ORACLE\ORADATA\PENNY\ARCHIVE\1_63.DBF recid=168 stamp=511712617
archive log filename=E:\ORACLE\ORADATA\PENNY\ARCHIVE\1_64.DBF recid=169 stamp=511712620
archive log filename=E:\ORACLE\ORADATA\PENNY\ARCHIVE\1_65.DBF recid=170 stamp=511712626
archive log filename=E:\ORACLE\ORADATA\PENNY\ARCHIVE\1_66.DBF recid=171 stamp=511712690
Finished backup at 02-DEC-03

Starting Control File and SPFILE Autobackup at 02-DEC-03
piece handle=E:\ORACLE\ORA92\DATABASE\C-362775766-20031202-01 comment=NONE
Finished Control File and SPFILE Autobackup at 02-DEC-03

released channel: ch1

```

简单看一下自动备份的控制文件及 spfile 文件缺省的格式及命名规则：

c-IIIIIIII-YYYYMMDD-QQ

c -----控制文件

IIIIIIII-----DBID

YYYYMMDD-----时间戳

QQ-----序号 00~FF，十六进制表示

(3) 使用自动备份恢复 spfile 文件：

```

[oracle@jumper bdump]$ rman target /

Recovery Manager: Release 9.2.0.3.0 - Production
Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.
connected to target database: HSJF (DBID=1052178311)

RMAN> restore spfile to '/tmp/spfileeygle.ora' from autobackup;
Starting restore at 17-JAN-04
using target database controlfile instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=18 devtype=DISK
channel ORA_DISK_1: looking for autobackup on day: 20040117
channel ORA_DISK_1: autobackup found: c-1052178311-20040117-01
channel ORA_DISK_1: SPFILE restore from autobackup complete
Finished restore at 17-JAN-04

RMAN> exit
Recovery Manager complete.

```

```
[oracle@jumper bdump]$ ls -l /tmp/spfileeyle.ora
-rw-r----- 1 oracle dba 3584 1月 17 09:34 /tmp/spfileeyle.ora
```

同样可以通过这种方法恢复自动备份的控制文件，示例如下：

```
[oracle@jumper bdump]$ rman target /
Recovery Manager: Release 9.2.0.3.0 - Production
Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.
connected to target database: HSJF (DBID=1052178311)

RMAN> restore controlfile to '/tmp/control01.ctl' from autobackup;

Starting restore at 17-JAN-04

using target database controlfile instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=10 devtype=DISK
channel ORA_DISK_1: looking for autobackup on day: 20040117
channel ORA_DISK_1: autobackup found: c-1052178311-20040117-02
channel ORA_DISK_1: controlfile restore from autobackup complete
Finished restore at 17-JAN-04

RMAN> exit
Recovery Manager complete.

[oracle@jumper bdump]$ ls -l /tmp/control*
-rw-r----- 1 oracle dba 1892352 1月 17 09:44 /tmp/control01.ctl
```

Oracle 9i 自动备份控制文件的功能给我们带来了极大的收益，通过自动备份，在数据库出现紧急状况的时候，用户可能可以从这个自动备份中获得更为有效及时的控制文件。

有一点还要说明的是，如果数据库无法 **Mount**，就不能使用如上方式恢复自动备份的控制文件或者参数文件。

```
[oracle@jumper dbs]$ rman target /
Recovery Manager: Release 9.2.0.4.0 - Production
Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.
connected to target database: conner (not mounted)

RMAN> restore controlfile to '/tmp/control01.ctl' from autobackup;

Starting restore at 08-MAR-06

using target database controlfile instead of recovery catalog
```

```

allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=11 devtype=DISK
RMAN-00571: =====
RMAN-00569: ===== ERROR MESSAGE STACK FOLLOWS =====
RMAN-00571: =====
RMAN-03002: failure of restore command at 03/08/2006 11:38:29
RMAN-06495: must explicitly specify DBID with SET DBID command

```

此时, Oracle 需要用户提供数据库的 DBID, 才能找到相应的自动备份用以恢复。如果无法得知 DBID, 那么可以直接指定自动备份集来进行恢复:

```
RMAN> restore controlfile to '/tmp/control01.ctl' from 'c-3152029224-20051221-00';
```

```

Starting restore at 08-MAR-06

using target database controlfile instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=9 devtype=DISK
channel ORA_DISK_1: restoring controlfile
channel ORA_DISK_1: restore complete
Finished restore at 08-MAR-06

```

进一步分析, 如果数据库无法 nomount, 那么恢复 spfile 文件时会遇到如下错误:

```

[oracle@jumper dbs]$ rman target /
Recovery Manager: Release 9.2.0.4.0 - Production
Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.
connected to target database (not started)

RMAN> restore spfile to '/tmp/spfile.ora' from 'c-3152029224-20060509-00';

Starting restore at 09-MAY-06

RMAN-00571: =====
RMAN-00569: ===== ERROR MESSAGE STACK FOLLOWS =====
RMAN-00571: =====
RMAN-03002: failure of restore command at 05/09/2006 14:09:43
RMAN-12010: automatic channel allocation initialization failed
RMAN-06403: could not obtain a fully authorized session
ORA-01034: ORACLE not available

```

ORA-27101: shared memory realm does not exist

Linux Error: 2: No such file or directory

此时，可以手工临时编辑一个 pfile 文件启动实例，即可进行 spfile 恢复，也可以采用第一章介绍的方法，适用 RMAN 启动默认实例，进行 spfile 文件恢复。

启动默认实例：

```
[oracle@jumper dbs]$ rman target /
```

Recovery Manager: Release 9.2.0.4.0 - Production

Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.

connected to target database (not started)

```
RMAN> startup nomount;
```

startup failed: ORA-01078: failure in processing system parameters

LRM-00109: could not open parameter file '/opt/oracle/product/9.2.0/dbs/initconner.ora'

trying to start the Oracle instance without parameter files ...

Oracle instance started

Total System Global Area	97588504 bytes
--------------------------	----------------

Fixed Size	451864 bytes
------------	--------------

Variable Size	46137344 bytes
---------------	----------------

Database Buffers	50331648 bytes
------------------	----------------

Redo Buffers	667648 bytes
--------------	--------------

恢复 spfile 文件：

```
[oracle@jumper log]$ rman target /
```

Recovery Manager: Release 9.2.0.4.0 - Production

Copyright (c) 1995, 2002, Oracle Corporation. All rights reserved.

connected to target database: DUMMY (not mounted)

```
RMAN> restore spfile to '/tmp/spfile.ora' from 'c-3152029224-20060509-00';
```

Starting restore at 09-MAY-06

using target database controlfile instead of recovery catalog

allocated channel: ORA_DISK_1

channel ORA_DISK_1: sid=9 devtype=DISK

```
channel ORA_DISK_1: autobackup found: c-3152029224-20060509-00
channel ORA_DISK_1: SPFILE restore from autobackup complete
Finished restore at 09-MAY-06
```

最后还要强调的是，缺省情况下，这个自动备份功能是关闭的，强烈推荐大家用上面提到的方法打开该功能。

2.2.9 如何设置 Events 事件

Events 事件是 Oracle 的重要诊断工具及问题解决办法，很多时候需要通过 Events 设置来屏蔽或者更改 Oracle 的行为，下面来看一下怎样修改 spfile，增加 Events 事件设置：

```
SQL> alter system set event='10841 trace name context forever' scope=spfile;

System altered.

SQL> startup force;
ORACLE instance started.

Total System Global Area  101782380 bytes
Fixed Size                  451436 bytes
Variable Size              75497472 bytes
Database Buffers           25165824 bytes
Redo Buffers                667648 bytes
Database mounted.
Database opened.
SQL> show parameter event
```

NAME	TYPE	VALUE
event	string	10841 trace name context forever

顺便提一句，10841 事件是用于解决 Oracle 9i 中 JDBC Thin Driver 问题的一个方法，如果 alert.log 文件中出现以下错误提示：

```
Wed Jan  7 17:17:08 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1775.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []
Wed Jan  7 17:17:18 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1777.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []
```

```

Wed Jan  7 17:17:24 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1783.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:17:31 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1785.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:17:39 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1777.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:17:45 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1783.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:17:52 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1787.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:18:11 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1791.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

Wed Jan  7 17:18:19 2004
Errors in file /opt/oracle/admin/phsdb/udump/phsdb_ora_1785.trc:
ORA-00600: internal error code, arguments: [ttcgshnd-1], [0], [], [], [], [], []

```

那么，很不幸，你很可能是遇到了 Bug 1725012。

通过设置以上事件，可以屏蔽和解决这个 ORA-00600 错误，具体可以参考 Metalink 相关文档。

如果想取消 event 参数设置，同样可以参照 reset 参数的方法：

```

SQL> show parameter event

NAME                                TYPE                                VALUE
-----
event                               string                             10046 trace name context forev
                                         er,level 12

SQL> alter system reset event scope=spfile sid='*';

System altered.

SQL> startup force;

ORACLE instance started.

```

```
Total System Global Area  219223120 bytes
Fixed Size                  451664 bytes
Variable Size              134217728 bytes
Database Buffers           83886080 bytes
Redo Buffers                667648 bytes
```

Database mounted.

Database opened.

SQL> show parameter event

NAME	TYPE	VALUE
event	string	

2.2.10 导出 SPFILE 文件

spfile 文件可以导出为文本文件，使用导出、创建过程可以向 spfile 中添加参数。

```
SQL> create pfile='e:\initeyglen.ora' from spfile;
```

文件已创建。

```
SQL> shutdown immediate
```

数据库已经关闭。

已经卸载数据库。

ORACLE 例程已经关闭。

initeyglen.ora 文件的具体内容如下：

```
*.aq_tm_processes=1
*.background_dump_dest='e:\oracle\admin\eyglen\bdump'
*.compatible='9.2.0.0.0'
*.control_files='e:\oracle\oradata\eyglen\control01.ctl',
'e:\oracle\oradata\eyglen\control02.ctl',
'e:\oracle\oradata\eyglen\control03.ctl'
*.core_dump_dest='e:\oracle\admin\eyglen\cdump'
*.db_block_size=8192
*.db_cache_size=25165824
*.db_domain=""
```

```

*.db_file_multiblock_read_count=16
*.db_name='eyglen'
*.dispatchers='(PROTOCOL=TCP) (SERVICE=eyglenXDB)'
*.fast_start_mttr_target=300
*.hash_join_enabled=TRUE
*.instance_name='eyglen'
*.java_pool_size=33554432
*.job_queue_processes=10
*.large_pool_size=8388608
*.open_cursors=300
*.pga_aggregate_target=25165824
*.processes=150
*.query_rewrite_enabled='FALSE'
*.remote_login_passwordfile='EXCLUSIVE'
*.shared_pool_size=50331648
*.sort_area_size=524288
*.sql_trace=FALSE
*.star_transformation_enabled='FALSE'
*.timed_statistics=TRUE
*.undo_management='AUTO'
*.undo_retention=10800
*.undo_tablespace='UNDOTBS1'
*.user_dump_dest='e:\oracle\admin\eyglen\udump'

```

对于单机 Oracle 数据库，每个参数以一个 “*” 开头，这意味着该参数可以影响所有的实例；而对于并行环境（RAC），可以看到，不同实例的参数设置可以不同，这时候可以用实例名称来替换 “*”，例如：

```

RACDB1.instance_number=1
RACDB2.instance_number=2
RACDB1.local_listener='LISTENER_RACDB1'
RACDB2.local_listener='LISTENER_RACDB2'
RACDB1.undo_tablespace='UNDOTBS1'
RACDB2.undo_tablespace='UNDOTBS2'

```

生成了 pfile 之后，可以使用这个 pfile，或者手动修改其中的参数以启动数据库。比如在更改数据库的归档模式时，修改这个 pfile，增加一行：

```

*.log_archive_start=true

```

使用这个 pfile 启动数据库：

```
SQL> startup pfile='e:\initgylen.ora'
```

ORACLE 例程已经启动。

```
Total System Global Area  135338868 bytes
Fixed Size                  453492 bytes
Variable Size              109051904 bytes
Database Buffers           25165824 bytes
Redo Buffers                667648 bytes
```

数据库装载完毕。

数据库已经打开。

```
SQL> show parameter log_archive_start
```

NAME	TYPE	VALUE

log_archive_start	boolean	TRUE

SQL>

然后可以使用新的 pfile 创建 spfile:

```
SQL> create spfile from pfile='e:\initgylen.ora';
```

文件已创建。

重新启动数据库，新的 spfile 生效。

```
SQL> startup
```

ORACLE 例程已经启动。

```
Total System Global Area  135338868 bytes
Fixed Size                  453492 bytes
Variable Size              109051904 bytes
Database Buffers           25165824 bytes
Redo Buffers                667648 bytes
```

数据库装载完毕。

数据库已经打开。

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE

spfile	string	%ORACLE_HOME%\DATABASE\SPFILE%ORACLE_SID%.ORA

```
SQL> show parameter log_archive_start
```

NAME	TYPE	VALUE
log_archive_start	boolean	TRUE

SQL>

需要提醒的是，在 Oracle 10g 之前，LOG_ARCHIVE_START 控制数据库可否自动归档，很多用户在修改数据库的归档模式时常常忘记修改 LOG_ARCHIVE_START 参数，结果导致数据库重新启动后无法自动归档，最后挂起，影响服务。

最终 Oracle 认识到了这个问题，从 Oracle 10g 开始，修改数据库的归档模式不需要再设置 LOG_ARCHIVE_START 参数。

```
$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 10.1.0.3.0 - Production on Wed Apr 13 09:53:25 2005
```

```
Copyright (c) 1982, 2004, Oracle. All rights reserved.
```

```
Connected to:
```

```
Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - 64bit Production
```

```
With the Partitioning and Data Mining options
```

```
SQL> archive log list;
```

Database log mode	No Archive Mode
Automatic archival	Disabled
Archive destination	USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence	25
Current log sequence	27

```
SQL> show parameter log_archive_start
```

NAME	TYPE	VALUE
log_archive_start	boolean	FALSE

```
SQL> shutdown immediate;
```

```
Database closed.
```

```
Database dismounted.
```

```
ORACLE instance shut down.
```

```
SQL> startup mount;
```

ORACLE instance started.

Total System Global Area 3204448256 bytes

Fixed Size 1304912 bytes

Variable Size 651957936 bytes

Database Buffers 2550136832 bytes

Redo Buffers 1048576 bytes

Database mounted.

SQL> alter database archivelog;

Database altered.

SQL> alter database open;

Database altered.

SQL> archive log list;

Database log mode	Archive Mode
Automatic archival	Enabled
Archive destination	USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence	25
Next log sequence to archive	27
Current log sequence	27

SQL>

通过以上示例，可以清晰地看到 Oracle 10g 中的这一改变。

提 示

这是我们介绍的第三种修改 spfile 的方式。

2.3 诊断案例

这是实际生产系统中关于 spfile 的一个案例问题，具体的问题诊断及主要解决步骤说明如下。

- 操作系统：SUN Solaris 8。
- 数据库版本：Oracle 9.2.0.3。
- 问题描述：工程人员报告，数据库在重新启动时无法正常启动，检查发现 UNDO 表空间丢失。

2.3.1 登录系统检查 alert.log 文件

在此有必要简单地介绍一下警报日志文件。

警报日志文件由按时间顺序排列的消息和错误的记录组成。下列信息会记录在警报日志文件中：

- 内部错误（ORA-600、ORA-07445 等错误信息）和块损坏错误（ORA-1578）；
- 影响数据库结构和参数的操作和诸如 CREATE DATABASE、STARTUP、SHUTDOWN、ARCHIVE LOG 和 RECOVER 之类的语句；
- 例程启动时所有非缺省的初始化参数值。

数据库管理员定期检查警报日志文件是很重要的，这样就可以在问题变得严重之前发现它们，并及时处理，在我们的生产环境中，警报日志按照以 5 分钟为间隔进行检测，如果发现任何错误提示或警报信息，就发送邮件给数据库管理员，请求人为介入管理。

由于警报日志文件是不停累计的，所以在检查以后可以按照规定删除或整理警报日志文件。控制警报日志文件位置的初始化参数是 BACKGROUND_DUMP_DEST：

```
SQL> show parameter background_dump_dest
```

NAME	TYPE	VALUE
background_dump_dest	string	/opt/oracle/admin/danaly/bdump

其缺省文件名为 alert_<sid>.log。

注 意

由于警报日志文件的重要作用，当数据库出现故障时，通常我们最先的处理步骤是检查该文件，以发现相关错误信息或线索，快速找到问题所在。这是 DBA 必须明确的一个知识点。

检查警报日志文件：

```
SunOS 5.8
```

```
login: root
```

```
Password:
```

```
Last login: Thu Apr 1 11:39:16 from 10.123.7.162
```

```
Sun Microsystems Inc. SunOS 5.8 Generic Patch October 2001
```

```
You have new mail.
```

```
# su - oracle
```

```
bash-2.03$ cd $ORACLE_BASE/admin/*/bdump
```

```
bash-2.03$ vi *.log
```

```

"alert_gzhs.log" 7438 lines, 283262 characters
.....
Thu Apr  1 11:11:28 2004
Ended recovery at
  Thread 1: logseq 177, block 3, scn 0.33124794
    0 data blocks read, 0 data blocks written, 1 redo blocks read
Crash recovery completed successfully
Thu Apr  1 11:11:28 2004
LGWR: Primary database is in CLUSTER CONSISTENT mode
Thread 1 advanced to log sequence 178
Thread 1 opened at log sequence 178
  Current log# 1 seq# 178 mem# 0: /u01/oradata/gzhs/redo01.log
Successful open of redo thread 1.
Thu Apr  1 11:11:28 2004
ARC0: Evaluating archive    log 3 thread 1 sequence 177
Thu Apr  1 11:11:28 2004
ARC0: Beginning to archive log 3 thread 1 sequence 177
Creating archive destination LOG_ARCHIVE_DEST_1: '/u06/oradata/gzhs/arch/1_177.dbf'
Thu Apr  1 11:11:28 2004
SMON: enabling cache recovery
ARC0: Completed archiving    log 3 thread 1 sequence 177
Thu Apr  1 11:11:28 2004
Errors in file /oracle/admin/gzhs/udump/gzhs_ora_27781.trc:
ORA-30012:  \263\267\317\373\261\355\277\325\274\344  'UNDOTBS1'  \262\273\264\346\324\332\273\
362\300\340\320\315\262\273\325\375\310\267
Thu Apr  1 11:11:28 2004
Error 30012 happened during db open, shutting down database
USER: terminating instance due to error 30012
Instance terminated by USER, pid = 27781
ORA-1092 signalled during: alter database open...
:q

```

在警报日志末尾显示了数据库在 **Open** 状态因为错误而异常终止，最后出错的错误号是 **ORA-30012**，该错误的含义是：

```

[oracle@jumper oracle]$ oerr ora 30012
30012, 00000, "undo tablespace '%s' does not exist or of wrong type"
// *Cause:    the specified undo tablespace does not exist or of the

```

```
//          wrong type.
// *Action:  Correct the tablespace name and reissue the statement.
```

这说明是 UNDO 表空间不存在导致出现问题。

2.3.2 尝试重新启动数据库

尝试重新启动数据库，检查问题是否仍然存在：

```
bash-2.03$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.3.0 - Production on 星期四 4 月 1 11:43:52 2004
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

已连接到空闲例程。

SQL> startup
ORACLE 例程已经启动。

Total System Global Area 4364148184 bytes
Fixed Size 736728 bytes
Variable Size 1845493760 bytes
Database Buffers 2516582400 bytes
Redo Buffers 1335296 bytes
数据库装载完毕。
ORA-01092: ORACLE 例程终止。强行断开连接
```

在 Open 步骤，例程终止，问题重现。

2.3.3 检查数据文件

检查数据文件，看 UNDO 表空间是否存在。

```
bash-2.03$ cd /u01/oradata/gzhs
bash-2.03$ ls -l
total 55702458
-rw-r----- 1 oracle dba 1073750016 Apr 1 11:44 UNDOTBS2.dbf
-rw-r----- 1 oracle dba 1073750016 Apr 1 11:44 WAP12_BILLINGDETAIL.dbf
-rw-r----- 1 oracle dba 1073750016 Apr 1 11:44 WAP12_MAIN.dbf
.....
-rw-r----- 1 oracle dba 1073750016 Apr 1 11:44 WAP12_MVIEW.dbf
.....
```

可以发现存在文件 UNDOTBS2.dbf，其大小约为 1GB。

2.3.4 mount 数据库，检查系统参数

在 mount 状态，检查一下当前参数设置：

```
bash-2.03$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 9.2.0.3.0 - Production on 星期四 4 月 1 11:46:20 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
已连接到空闲例程。
```

```
SQL> startup mount;
```

```
ORACLE 例程已经启动。
```

```
Total System Global Area 4364148184 bytes
```

```
Fixed Size 736728 bytes
```

```
Variable Size 1845493760 bytes
```

```
Database Buffers 2516582400 bytes
```

```
Redo Buffers 1335296 bytes
```

```
数据库装载完毕。
```

```
SQL> select name from v$datafile;
```

```
NAME
```

```
-----  
/u01/oradata/gzhs/system01.dbf
```

```
.....
```

```
/u01/oradata/gzhs/UNDOTBS2.dbf
```

```
已选择 23 行。
```

```
SQL> show parameter undo
```

NAME	TYPE	VALUE
undo_management	string	AUTO
undo_retention	integer	10800
undo_suppress_errors	boolean	FALSE
undo_tablespace	string	UNDOTBS1

```
SQL> show parameter spfile
```

NAME	TYPE	VALUE

spfile	string	

发现系统没有使用 spfile，而初始化参数设置的 UNDO 表空间为 UNDOTBS1，数据库中登记的 UNDO 文件为 UNDOTBS2.dbf。

2.3.5 检查参数文件

检查一下 pfile 文件中的设置，发现 UNDO 表空间参数设置的是 UNDOTBS1。

```
bash-2.03$ cd $ORACLE_HOME/dbs
bash-2.03$ ls
init.ora          initgzhs.ora      initgzhs.ora.old  orapwgzhs
initdw.ora        initgzhs.ora.hurray lkGZHS            snapcf_gzhs.f
bash-2.03$ vi initgzhs.ora
"initgzhs.ora" [Incomplete last line] 105 lines, 3087 characters
.....
#####
# System Managed Undo and Rollback Segments
#####
undo_management=AUTO
undo_retention=10800
undo_tablespace=UNDOTBS1

:q!
```

这个设置是极其可疑的，也就是说，参数设置可能和数据库的实际情况不符。

2.3.6 再次检查 alert 文件

警告日志文件中记录了对于数据库重要操作的信息，可以从中查找对于 UNDO 表空间的操作。

(1) 创建数据库时的信息：

```
Sat Feb  7 20:30:12 2004
CREATE DATABASE gzhs
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 5
```

```

MAXLOGMEMBERS 3
MAXDATAFILES 100
DATAFILE '/u01/oradata/gzhs/system01.dbf' SIZE 500M REUSE AUTOEXTEND ON NEXT 10240K
MAXSIZE UNLIMITED
EXTENT MANAGEMENT LOCAL
DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE '/u01/oradata/gzhs/temp01.dbf' SIZE 1000M
REUSE AUTOEXTEND ON NEXT 250M MAXSIZE UNLIMITED
UNDO TABLESPACE "UNDOTBS1" DATAFILE '/u01/oradata/gzhs/undotbs01.dbf' SIZE 1000M
REUSE AUTOEXTEND ON NEXT 100M MAXSIZE UNLIMITED
CHARACTER SET ZHS16GBK
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1 ('/u01/oradata/gzhs/redo01.log') SIZE 256M,
GROUP 2 ('/u01/oradata/gzhs/redo02.log') SIZE 256M,
GROUP 3 ('/u01/oradata/gzhs/redo03.log') SIZE 256M

```

(2) 发现创建 UNDOTBS2 的记录信息:

```

Wed Mar 24 20:20:58 2004
/* OracleOEM */ CREATE UNDO TABLESPACE "UNDOTBS2" DATAFILE '/u01/oradata/
gzhs/UNDOTBS2.dbf' SIZE 1024M AUTOEXTEND ON NEXT 100M MAXSIZE UNLIMITED
Wed Mar 24 20:22:37 2004
Created Undo Segment _SYSSMU11$
Created Undo Segment _SYSSMU12$
Created Undo Segment _SYSSMU13$
Created Undo Segment _SYSSMU14$
Created Undo Segment _SYSSMU15$
Created Undo Segment _SYSSMU16$
Created Undo Segment _SYSSMU17$
Created Undo Segment _SYSSMU18$
Created Undo Segment _SYSSMU19$
Created Undo Segment _SYSSMU20$
Completed: /* OracleOEM */ CREATE UNDO TABLESPACE "UNDOTBS2"
Wed Mar 24 20:24:25 2004
Undo Segment 11 Online
Undo Segment 12 Online
Undo Segment 13 Online
Undo Segment 14 Online
Undo Segment 15 Online

```

```

Undo Segment 16 Onlined
Undo Segment 17 Onlined
Undo Segment 18 Onlined
Undo Segment 19 Onlined
Undo Segment 20 Onlined
Successfully onlined Undo Tablespace 15.
Undo Segment 1 Offlined
Undo Segment 2 Offlined
Undo Segment 3 Offlined
Undo Segment 4 Offlined
Undo Segment 5 Offlined
Undo Segment 6 Offlined
Undo Segment 7 Offlined
Undo Segment 8 Offlined
Undo Segment 9 Offlined
Undo Segment 10 Offlined
Undo Tablespace 1 successfully switched out.

```

(3) 新的 UNDO 表空间被应用:

```

Wed Mar 24 20:24:25 2004
ALTER SYSTEM SET undo_tablespace='UNDOTBS2' SCOPE=MEMORY;

```

可以发现问题就在这里，创建了新的 UNDO 表空间以后，因为使用的是 pfile 文件，切换表空间的修改只对当前实例生效，操作人员忘记了修改 pfile 文件。

如果使用 spfile，缺省的修改范围是 both，会同时修改 spfile 文件，就可以避免以上问题的出现。

(4) 删除了 UNDOTBS1 的信息:

```

Wed Mar 24 20:25:01 2004
/* OracleOEM */ DROP TABLESPACE "UNDOTBS1" INCLUDING CONTENTS AND DATAFILES CASCADE CONSTRAINTS
Wed Mar 24 20:25:03 2004
Deleted file /u01/oradata/gzhs/undotbs01.dbf
Completed: /* OracleOEM */ DROP TABLESPACE "UNDOTBS1" INCLUDING CONTENTS AND DATAFILES CASCADE CONSTRAINTS

```

这样再次重新启动数据库的时候，问题出现了，pfile 中定义的 UNDOTBS1 找不到了，而且操作是在很久以前，没人能回忆起来，甚至无法得知是什么人操作的。

2.3.7 修正 PFILE

找到了问题的根源，解决起来就简单了，修改 pfile 参数文件，就可以启动数据库：

```

bash-2.03$ vi initgzhs.ora
"initgzhs.ora" [Incomplete last line] 105 lines, 3087 characters
.....
#####
# System Managed Undo and Rollback Segments
#####
undo_management=AUTO
undo_retention=10800
undo_tablespace=UNDOTBS2

~
"initgzhs.ora" 105 lines, 3088 characters

```

2.3.8 启动数据库

重新启动数据库：

```

bash-2.03$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.3.0 - Production on 星期四 4 月 1 11:55:11 2004
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

连接到:

Oracle9i Enterprise Edition Release 9.2.0.3.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production

SQL> select * from v$version;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.3.0 - 64bit Production
PL/SQL Release 9.2.0.3.0 - Production
CORE 9.2.0.3.0 Production
TNS for Solaris: Version 9.2.0.3.0 - Production
NLSRTL Version 9.2.0.3.0 - Production

```

至此，问题得以完满解决。在这里可以看到，使用 `spfile` 可以免去手工修改 `pfile` 文件的麻烦，减少了犯错的可能。

参考信息与建议阅读

(1) Oracle Database Administrator's Guide 10g Release 2 (10.2) B14231-01

(2) Oracle 创建实例的最少参数需求

http://www.eygle.com/archives/2006/05/instance_nomount_parameter.html

(3) Oracle 中口令文件的作用及说明

<http://www.eygle.com/faq/passwordfile.htm>

(4) Oracle 9i 新特性 spfile

<http://www.eygle.com/faq/Oracle9i.New.Feature.Spfile.01.htm>

我们知道 Oracle 通过数据字典来管理和展现数据库信息，这些信息至关重要。正确理解这部分内容将有助于提高用户对 Oracle 数据库的认知，加强自学能力。本章将对 Oracle 的数据字典进行探讨。

3.1 数据字典概述

数据字典 (Data Dictionary) 是 Oracle 数据库的一个重要组成部分，是元数据 (Metadata) 的存储地点。Oracle RDBMS 使用数据字典记录和管理对象信息和安全信息等，用户和数据库系统管理员可以通过数据字典来获取数据库相关信息。

数据字典包括以下内容：

- 所有数据库 Schema 对象的定义 (表、视图、索引、聚簇、同义词、序列、过程、函数、包、触发器等)；
- 数据库的空间分配和使用情况；
- 字段的缺省值；
- 完整性约束信息；
- Oracle 用户名称、角色、权限等信息；
- 审计信息；
- 其他数据库信息。

总之，数据字典是数据库的核心，通过数据字典，Oracle 数据库基本上可以实现自解释。

一般来说，数据字典是只读的，通常不建议对任何数据字典表中的任何信息进行手工更新或改动，对于数据字典的修改很容易就会导致数据库紊乱，造成无法恢复的后果，而且 Oracle 公司不对此类操作带来的后果负责。

通常所说的数据字典由 4 部分组成：内部 RDBMS (X\$) 表、数据字典表、动态性能 (V\$) 视图和数据字典视图。

3.2 内部 RDBMS (X\$) 表

X\$表是 Oracle 数据库的核心部分，这些表用于跟踪内部数据库信息，维持数据库的正常

运行。X\$表是加密命名的，而且 Oracle 不作文档说明，这部分知识是 Oracle 公司的技术机密，Oracle 通过这些 X\$建立起其他大量视图，提供用户查询管理数据库之用。但是由于 X\$表记录了大量的有用信息，所以也不停地被全球的 DBA 不懈地探索着，最为人所熟知的有 X\$BH、X\$KSMSP 等。

X\$表是 Oracle 数据库的运行基础，在数据库启动时由 Oracle 应用程序动态创建。这部分表对数据库来说至关重要，所以 Oracle 不允许 SYSDBA 之外的用户直接访问，显示授权不被允许。

如果显示授权，用户会收到如下错误：

```
SQL> grant select on x$ksppi to eygle;
grant select on x$ksppi to eygle
      *
ERROR at line 1:
ORA-02030: can only select from fixed tables/views
```

Oracle 的解释是：

```
ORA-02030 can only select from fixed tables/views
Cause: An attempt is being made to perform an operation other than a retrieval from a fixed table/view.
Action: You may only select rows from fixed tables/views.
```

一句话，这些对象你最好只是查询。

发现、观察、研究 X\$表的一个好办法是借用 Oracle 的 AUTOTRACE 功能，当查询一些常用视图的时候，可以通过 AUTOTRACE 功能发现这些 View 的底层表。

以下是 Oracle 10gR2 中的一个示例：

```
SQL> set autotrace trace explain
SQL> select * from v$parameter;

Execution Plan
-----
Plan hash value: 1128103955

-----
| Id | Operation          | Name          | Rows | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |               |      |       |            |          |
|*  1 |  HASH JOIN         |               |      |       |            |          |
|*  2 |    FIXED TABLE FULL| X$KSPPI       |      |       |            |          |
|  3 |      FIXED TABLE FULL| X$KSPPCV      | 100 | 67700 |            |          |
-----
```

Predicate Information (identified by operation id):

```
1 - access("X"."INDX"="Y"."INDX")
      filter(TRANSLATE("KSPPINM",'_','#') NOT LIKE '##' OR
            "KSPSTDF"='FALSE' OR BITAND("KSPSTVF",5)>0)

2 - filter("X"."INST_ID"=USERENV('INSTANCE') AND
      TRANSLATE("KSPPINM",'_','#') NOT LIKE '##')
```

这些研究和探索是极有趣味的，如果能就此深入下去，一定能够时常发现意外的收获。

顺便介绍一个有意思的 X\$表，也许你曾经关注过 X\$KVIT，其名称含义为：[K]ernel Layer
Performance Layer [V] [I]nformation tables [T]ransitory Instance parameters。

这个视图记录的是和实例相关的一些内部参数设置，可以看到一些很有意思的内容：

```
SQL> select * from v$version;
```

BANNER

Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod

PL/SQL Release 10.2.0.1.0 - Production

CORE 10.2.0.1.0 Production

TNS for Linux: Version 10.2.0.1.0 - Production

NLSRTL Version 10.2.0.1.0 - Production

```
SQL> select kvittag,kvitval,kvitdsc
```

```
2 from x$kvit;
```

KVITTAG

KVITVAL KVITDSC

ksbcpu	4 number of logical CPUs in the system used by Oracle
ksbcpucore	0 number of physical CPU cores in the system used by Oracle
ksbcpusocket	2 number of physical CPU sockets in the system used by Oracle
ksbcpu_hwm	4 high water mark of number of CPUs used by Oracle
ksbcpucore_hwm	0 high water mark of number of CPU cores on system
ksbcpusocket_hwm	2 high water mark of number of CPU sockets on system
ksbcpu_actual	4 number of available CPUs in the system
ksbcpu_dr	1 CPU dynamic reconfiguration supported
kcbnbh	238518 number of buffers

kcbldq	25 large dirty queue if kcbclw reaches this
kcbfsp	40 Max percentage of LRU list foreground can scan for free
kcbcln	2 Initial percentage of LRU list to keep clean
kcbnbf	750 number buffer objects
kcbwst	0 Flag that indicates recovery or db suspension
kcteln	0 Error Log Number for thread open
kcvgcw	0 SGA: opcode for checkpoint cross-instance call
kcvgcw	0 SGA:opcode for pq checkpoint cross-instance call
17 rows selected	

不知道大家是否还记得，触发后台进程 DBWR 写动作的条件包含以下两个。

(1) 脏缓冲 (Dirty Buffers) 阈值 (threshold) 达到。

那么这个 threshold 是多少呢？从以上视图中，可以知道，这个值是 25%，即：

kcbldq	25 large dirty queue if kcbclw reaches this
--------	---

(2) No Free Buffer，也就是当进程扫描 LRU 一定数量的 Block 之后，如果还找不到足够的 Free 空间，则触发 DBWR 执行写出。

那么这个扫描数量是多少呢？从以上视图中，可以知道，这个比例为 40%，即：

kcbfsp	40 Max percentage of LRU list foreground can scan for free
--------	--

这些限制不能由参数调整，是数据库的内部限制。

通过这些内容，可以把数据库抽象的概念具体化，有兴趣的读者可以继续探索。本书将在第 5 章中对这些内容作进一步的说明。

3.3 数据字典表

数据字典表 (Data Dictionary Table) 用以存储表、索引、约束以及其他数据库结构的信息。这些对象通常以 “\$” 结尾 (如 tab\$、obj\$、ts\$等)，在创建数据库的时候通过运行 sql.bsq 脚本来创建。

sql.bsq 是非常重要的一个文件，其中包含了数据字典表的定义及注释说明，每个试图深入学习 Oracle 数据库的用户都应该仔细阅读一下该文件。该文件位于 \$ORACLE_HOME/dbms/admin 目录下，下面摘录一点内容来对该文件加以简单说明。

以下是曾经提到过的 bootstrap\$表的定义：

```
create table bootstrap$
( line#          number not null,                               /* statement order id */
  obj#           number not null,                               /* object number */
  sql_text       varchar2("M_VCSZ") not null                   /* statement */
  storage (initial 50K) /* to avoid space management during IOR I */
//                                                         /* "/" required for bootstrap */
```

undo\$字典表的部分结构如下：

```
create table undo$                                     /* undo segment table */
( us#          number not null,                        /* undo segment number */
  name         varchar2("M_IDEN") not null,           /* name of this undo segment */
  user#        number not null,                        /* owner: 0 = SYS(PRIVATE), 1 = PUBLIC */
  file#        number not null,                        /* segment header file number */
  block#       number not null,                        /* segment header block number */
  scnbas       number,                                /* highest commit time in rollback segment */
  scnwrp       number,                                /* scnbas - scn base, scnwrp - scn wrap */
  xactsqn      number,                                /* highest transaction sequence number */
  undosqn      number,                                /* highest undo block sequence number */
  inst#        number,                                /* parallel server instance that owns the segment */
  status$      number not null,                        /* segment status (see KTS.H): */
/* 1 = INVALID, 2 = AVAILABLE, 3 = IN USE, 4 = OFFLINE, 5 = NEED RECOVERY,
   * 6 = PARTLY AVAILABLE (contains in-doubt txs)
   */
  ts#          number,                                /* tablespace number */
  ugrp#        number,                                /* The undo group it belongs to */
  .....
)
```

这里不再过多引用，只要打开这个文件，可能读者会发现，很多困扰许久的问题，在这里可以轻易地找到注解及答案。

3.4 动态性能视图

动态性能（V\$）视图（Dynamic Performance View）记录了数据库运行时信息和统计数据，大部分动态性能视图被实时更新以反映数据库当前状态。

Oracle 通过动态性能视图将 Oracle 数据库的状态展示出来，提供给用户和数据库管理员，Oracle 对 V\$视图给出了详细的文档说明供开发管理人员参考，是研究和管理数据库的主要依据。

3.4.1 GV\$和 V\$视图

在数据库启动时，Oracle 动态创建 X\$表，在此基础上，Oracle 创建了 GV\$和 V\$视图。从 Oracle 8 开始，GV\$视图开始被引入，其含义为 Global V\$。除了一些特例以外，每个 V\$视图都有一个对应的 GV\$视图存在。

GV\$视图的产生是为了满足 OPS 环境的需要，在 OPS 环境中，查询 GV\$视图返回所有

实例信息，而每个 V\$视图是基于 GV\$视图，增加了 INST_ID 列的 WHERE 条件限制建立，只包含当前连接实例信息。

注意，每个 V\$视图都包含类似语句：

```
where inst_id = USERENV('Instance')
```

用于限制返回当前实例信息。

这里以 Oracle 10gR2 RAC 环境为例，看一下 GV\$和 V\$的输出异同。下面是 GV\$的输出，包含了 2 个实例的信息：

```
SQL> select inst_id,instance_name,status,version
2  from gv$instance;
```

INST_ID	INSTANCE_NAME	STATUS	VERSION
1	RACDB1	OPEN	10.2.0.1.0
2	RACDB2	OPEN	10.2.0.1.0

再看一下 V\$的输出：

```
SQL> select instance_number,instance_name,status
2  from v$instance;
```

INSTANCE_NUMBER	INSTANCE_NAME	STATUS
1	RACDB1	OPEN

Oracle 提供了一些特殊视图用以记录其他视图的创建方式。v\$fixed_view_definition 就是其中之一。从 GV\$FIXED_TABLE 和 V\$FIXED_TABLE 开始，看一下 GV\$视图和 V\$视图的结构及创建方式：

```
SQL> select view_definition from v$fixed_view_definition where view_name='V$FIXED_TABLE';
```

VIEW_DEFINITION

```
select NAME , OBJECT_ID , TYPE , TABLE_NUM from GV$FIXED_TABLE where inst_id =
USERENV('Instance')
```

这里看到 V\$FIXED_TABLE 基于 GV\$FIXED_TABLE 创建。

```
SQL> select view_definition from v$fixed_view_definition where view_name='GV$FIXED_TABLE';
```

VIEW_DEFINITION

```
select inst_id,kqftanam, kqftaobj, 'TABLE', indx from x$kqfta
```

```

union all
select inst_id,kqfvinam, kqfviobj, 'VIEW', 65537 from x$skqfvi
union all
select inst_id,kqfdtnam, kqfdtobj, 'TABLE', 65537 from x$skqfdt

```

这样就找到了 GV\$FIXED_TABLE 视图的创建语句，该视图基于 X\$表创建，然后 V\$视图基于 GV\$视图创建。

我们知道，GV\$视图和 V\$视图是在数据库创建过程中建立起来的，内置于数据库中，Oracle 通过 v\$fixed_view_definition 视图为我们展现这些定义。

3.4.2 GV_\$、V_\$视图和 V\$、GV\$同义词

在 GV\$和 V\$之后，Oracle 建立了 GV_\$和 V_\$视图，随后为这些视图建立了公用同义词。这些工作都是通过 catalog.sql 脚本（该脚本位于\$ORACLE_HOME/rdbms/admin/目录下）实现的。

从 catalog.sql 脚本中摘录一段：

```

create or replace view v_$fixed_table as select * from v$fixed_table;
create or replace public synonym v$fixed_table for v_$fixed_table;

create or replace view gv_$fixed_table as select * from gv$fixed_table;
create or replace public synonym gv$fixed_table for gv_$fixed_table;

```

从以上脚本中注意到，第一个视图 V_\$和 GV_\$视图基于 V\$和 GV\$视图首先被创建，然后基于 V_\$和 GV_\$视图的同义词被创建。

通过 V_\$视图，Oracle 把 V\$视图和普通用户隔离，V_\$视图的权限可以授予其他用户，而 Oracle 不允许任何对于 V\$视图的直接授权，看以下例子：

```

[oracle@jumper udump]$ sqlplus '/' as sysdba'

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jun 13 16:41:41 2005

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production

SQL> grant select on v$sga to eygle;

```

```
grant select on v$sga to eygle
      *

ERROR at line 1:
ORA-02030: can only select from fixed tables/views

SQL> grant select on v_$sga to eygle;

Grant succeeded.
```

对于内部 X\$表及 V\$视图的限制，Oracle 是通过软件机制实现的，而并非通过数据库权限控制。

所以，实际上通常大部分用户访问的 V\$对象，并不是视图，而是指向 V_\$视图的同义词；而 V_\$视图是基于真正的 V\$视图（这个视图是基于 X\$表建立的）创建的。

在进行数据访问时，Oracle 访问 VIEW 优先，然后是同义词。下面通过以下实验来验证一下这个结论。

首先参考 Oracle 处理机制，创建 X\$EYGLE、V\$EYGLE、V_\$EYGLE 和公用同义词 V\$EYGLE：

```
[oracle@jumper udump]$ sqlplus eygle/eygle

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jun 13 17:37:25 2005
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production

SQL> create table x$eygle as select username from dba_users;

Table created.

SQL> create view v$eygle as select * from x$eygle;

View created.

SQL> create view v_$eygle as select * from v$eygle;

View created.
```

```
SQL> create public synonym v$eygle for v_$eygle;
```

```
Synonym created.
```

然后在 sys 用户下创建 V\$EYGLE 视图:

```
SQL> connect / as sysdba
```

```
Connected.
```

```
SQL> create view v$eygle as select username,user_id from dba_users;
```

```
View created.
```

此时查询, 得到的 SYS 的 V\$EYGLE 信息:

```
SQL> desc v$eygle;
```

Name	Null?	Type

USERNAME	NOT NULL	VARCHAR2(30)
USER_ID	NOT NULL	NUMBER

当删除这个视图以后, 再次访问时, Oracle 选择访问了 V\$EYGLE 同义词:

```
SQL> drop view v$eygle ;
```

```
View dropped.
```

```
SQL> desc v$eygle
```

Name	Null?	Type

USERNAME	NOT NULL	VARCHAR2(30)

```
SQL>
```

v\$fixed_view_definition 视图是研究 Oracle 对象关系的一个入口, 仔细理解 Oracle 的数据字典机制, 将有助于深入了解和学习 Oracle 数据库知识。

3.4.3 数据字典视图

数据字典视图是在 X\$表和数据库字典表之上创建的视图, 在创建数据库时由 catalog.sql 脚本 (该脚本位于 \$ORACLE_HOME/rdbms/admin/目录下) 创建。

按照前缀的不同, 数据字典视图通常被分为以下 3 类。

- USER_类视图: 包含了用户所拥有的相关对象信息。

- **ALL_类视图**：包含了用户有权限访问的所有对象的信息。
- **DBA_类视图**：包含了数据库所有相关对象的信息。

通常 **USER_类视图** 不包含 **Owner** 字段，查询潜在的返回当前用户的对象信息，下面以 **USER_TABLES** 视图为例（篇幅原因，省略了部分内容）来看一下其创建及结构：

```
create or replace view USER_TABLES
(TABLE_NAME, TABLESPACE_NAME, CLUSTER_NAME, IOT_NAME,
 PCT_FREE, PCT_USED,
 .....
 DEGREE, INSTANCES, CACHE, TABLE_LOCK,
 SAMPLE_SIZE, LAST_ANALYZED, PARTITIONED,
 IOT_TYPE, TEMPORARY, SECONDARY, NESTED,
 BUFFER_POOL, ROW_MOVEMENT,
 GLOBAL_STATS, USER_STATS, DURATION, SKIP_CORRUPT, MONITORING,
 CLUSTER_OWNER, DEPENDENCIES, COMPRESSION)
as
select o.name, decode(bitand(t.property, 4194400), 0, ts.name, null),
       decode(bitand(t.property, 1024), 0, null, co.name),
       .....
from sys.ts$ ts, sys.seg$ s, sys.obj$ co, sys.tab$ t, sys.obj$ o,
     sys.obj$ cx, sys.user$ cu
where o.owner# = userenv('SCHEMAID')
    and o.obj# = t.obj#
    .....
    and t.dataobj# = cx.obj# (+)
    and cx.owner# = cu.user# (+)
/
```

注意到 **Where** 条件中有这样一个限制：

```
where o.owner# = userenv('SCHEMAID')
```

这就限制了当前查询只返回当前用户的 **SCHEMA** 对象信息。

而对于 **ALL_TABLES** 视图，在 **WHERE** 子句中，关于用户部分，增加了这样一个条件：

```
and (o.owner# = userenv('SCHEMAID')
    or o.obj# in
      (select oa.obj#
       from sys.objauth$ oa
       where grantee# in ( select kzsrorol
                           from x$kzsro
                           )
      )
```

```

    )
    or /* user has system privileges */
    exists (select null from v$enabledprivs
            where priv_number in (-45 /* LOCK ANY TABLE */,
                                   -47 /* SELECT ANY TABLE */,
                                   -48 /* INSERT ANY TABLE */,
                                   -49 /* UPDATE ANY TABLE */,
                                   -50 /* DELETE ANY TABLE */)
    )
)

```

这个条件扩展了关于用户有权限访问的对象信息，所以实际上 USER_TABLES 的结果是 ALL_TABLES 结果的一个子集。

DBA_TABLES 视图的 Where 条件中，没有关于 Owner 的限制，所以查询返回了数据库中所有表的信息：

```

where o.owner# = u.user#
   and o.obj# = t.obj#
   and bitand(t.property, 1) = 0
   and t.bobj# = co.obj# (+)
   and t.ts# = ts.ts#
   and t.file# = s.file# (+)
   and t.block# = s.block# (+)
   and t.ts# = s.ts# (+)
   and t.dataobj# = cx.obj# (+)
   and cx.owner# = cu.user# (+)
/

```

这就是这几类数据字典视图的区别所在。

3.4.4 进一步的说明

Oracle 的 X\$表信息可以从 v\$fixed_table 中查到：

```

SQL> select count(*) from v$fixed_table where name like 'X$%';

COUNT(*)
-----
      394

```

对于 Oracle 9iR2，共有 394 个 X\$对象被记录。

X\$表建立以后，基于 X\$表的 GV\$和 V\$视图得以创建。这部分视图也可以通过查询 V\$FIXED_TABLE 得到。

```
SQL> select count(*) from v$fixed_table where name like 'GV$%';

COUNT(*)
-----
259
```

这一部分共 259 个对象。

```
SQL> select count(*) from v$fixed_table where name like 'V$%';

COUNT(*)
-----
259
```

同样是 259 个对象。

v\$fixed_table 共记录了 $394 + 259 + 259$ 共 912 个对象。

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

COUNT(*)
-----
912
```

以上是 Oracle 9iR2 单机环境中的数据：

```
SQL> select * from v$version;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
PL/SQL Release 9.2.0.4.0 - Production
CORE      9.2.0.3.0      Production
TNS for Linux: Version 9.2.0.4.0 - Production
NLSRTL Version 9.2.0.4.0 - Production
```

而在 Oracle 10gR2 中, v\$fixed_table 中的对象数量增加到 1383 个(不同安装可能有所不同), 大家可以通过这个视图对 Oracle 数据库不同版本的变化进行对比和研究, 这里不再赘述。

3.5 最后的验证

最后通过 V\$PARAMETER 视图来追踪一下数据库的架构。

3.5.1 V\$PARAMETER 的结构

```
SQL> select view_definition from v$fixed_view_definition a where a.VIEW_NAME='V$PARAMETER';
```

VIEW_DEFINITION

```

-----
select  NUM , NAME , TYPE , VALUE , ISDEFAULT , ISSYS_MODIFIABLE , ISSYS_MODIFIA
        BLE , ISMODIFIED , ISADJUSTED , DESCRIPTION, UPDATE_COMMENT from GV$PARAMETER
where inst_id = USERENV('Instance')

```

可以看到 V\$PARAMETER 是由 GV\$PARAMETER 创建的，GV\$PARAMETER 则是由 X\$创建的。

```
SQL> select view_definition from v$fixed_view_definition a where a.VIEW_NAME='GV$PARAMETER';
```

VIEW_DEFINITION

```

-----
select x.inst_id,x.indx+1,ksppinm,ksppity,ksppstvl,ksppstdf, decode(bitand(kspp
iflg/256,1),1,'TRUE','FALSE'), decode(bitand(ksppiflg/65536,3),1,'IMMEDIATE',2,
'DEFERRED',                                3,'IMMEDIATE','FALSE'), decode(bit
and(ksppstvf,7),1,'MODIFIED',4,'SYSTEM_MOD','FALSE'), decode(bitand(ksppstvf,2)
,2,'TRUE','FALSE'), ksppdesc, ksppstcmnt from x$ksppi x, x$ksppcv y where (x.i
ndx = y.indx) and ((translate(ksppinm,'_','#') not like '#%') or (ksppstdf = 'F
ALSE'))

```

在这里，可以看到 GV\$PARAMETER 来源于 x\$ksppi 和 x\$ksppcv 两个 X\$表。x\$ksppi 和 x\$ksppcv 基本上包含所有数据库参数，v\$parameter 展现的是不包含 “_” 开头的参数。以 ‘_’ 开头的参数通常称为隐含参数，一般不为大家所知，不建议修改，但很多隐含参数因为功能强大而被经常使用，并被不停地探索和研究。

3.5.2 视图还是同义词

在非 SYS 用户下查询，很多朋友曾经提出过疑问，那就是，当访问 V\$PARAMETER 对象时，访问的是视图还是同义词？

如果读者还记得前面讲过的内容，那么你会知道，毫无疑问，这里访问的是同义词，因为除了 SYS 用户以外，其他用户不能查询 V\$视图，V\$视图也不能被授权给其他用户。那么这个问题实际上是不成立的。

```
SQL> connect / as sysdba
```

```
Connected.
```

```
SQL> grant select on v$parameter to eygle;
```

```
grant select on v$parameter to eygle
```

```
*
```

```
ERROR at line 1:
```

```
ORA-02030: can only select from fixed tables/views
```

```
SQL> connect eygle/eygle
```

```
Connected.
```

```
SQL> desc sys.v$parameter
```

```
ERROR:
```

```
ORA-04043: object sys.v$parameter does not exist
```

```
SQL> desc v$parameter
```

Name	Null?	Type
NUM		NUMBER
NAME		VARCHAR2(64)
TYPE		NUMBER
VALUE		VARCHAR2(512)
ISDEFAULT		VARCHAR2(9)
ISSES_MODIFIABLE		VARCHAR2(5)
ISSYS_MODIFIABLE		VARCHAR2(9)
ISMODIFIED		VARCHAR2(10)
ISADJUSTED		VARCHAR2(5)
DESCRIPTION		VARCHAR2(64)
UPDATE_COMMENT		VARCHAR2(255)

3.5.3 Oracle 如何通过同义词定位对象

如果愿意的话，可以进一步进行追溯，使用 10046 事件可以看到更多的东西。
通过 10046 事件跟踪查询：

```
[oracle@jumper udump]$ sqlplus eygle/eygle
```

```
SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jun 13 18:29:22 2005
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Connected to:
```

```
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

```
With the Partitioning option
```

```
JServer Release 9.2.0.4.0 - Production
```

```
SQL> alter session set events '10046 trace name context forever,level 12';
```

Session altered.

SQL> select count(*) from v\$parameter;

COUNT(*)

262

SQL> exit

Disconnected from Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

With the Partitioning option

JServer Release 9.2.0.4.0 - Production

注 意

关于 10046 事件的使用，读者可以参考以下链接 [http://www.eygle.com/case/ Use.sql_trace.to.Diagnose.database.htm](http://www.eygle.com/case/Use.sql_trace.to.Diagnose.database.htm)。

在这里不要使用 tkprof 格式化，因为 tkprof 可能会隐去重要信息（本文仅摘取几段重要跟踪信息，读者完全可以通过实验获得相同的输出）。

第一段重要代码是：

```
PARSING IN CURSOR #2 len=198 dep=1 uid=0 oct=3 lid=0 tim=1092440257023120 hv=2703824309
id='567681f0'
select obj#,type#,ctime,mtime,stime,status,dataobj#,flags,oid$, spare1, spare2 from obj$ where owner#=:1
and name=:2 and namespace=:3 and remoteowner is null and linkname is null and subname is null
END OF STMT
PARSE #2:c=0,e=1601,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257023088
BINDS #2:
bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
bfp=b701cf24 bln=22 avl=02 flg=05
value=25
bind 1: dty=1 mxl=32(11) mal=00 scl=00 pre=00 oacflg=18 oacfl2=1 size=32 offset=0
bfp=b701c7b4 bln=32 avl=11 flg=05
value="V$PARAMETER"
bind 2: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
bfp=b701c790 bln=24 avl=02 flg=05
value=1
```

Oracle 根据 3 个传入参数 owner#=25、name=V\$PARAMETER、namespace=1，来判断对象类型，按照表、视图优先规则来定位判断，对于本例这个查询是不会有结果的。

接下来 Oracle 继续判断，此时需要验证同义词了：

```
PARSING IN CURSOR #4 len=46 dep=1 uid=0 oct=3 lid=0 tim=1092440257028409 hv=3378994511
id='576eb040'

select node,owner,name from syn$ where obj#=:1

END OF STMT

PARSE #4:c=0,e=1278,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257028379

BINDS #4:

  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=08 oacfl2=1 size=24 offset=0
         bfp=b701b3cc bln=22 avl=03 flg=05
         value=841
```

传入绑定变量值是 841，看看 841 是什么：

```
SQL> select object_name,object_id,object_type from dba_objects where object_id=841;
```

OBJECT_NAME	OBJECT_ID	OBJECT_TYPE
V\$PARAMETER	841	SYNONYM

841 正是这个同义词，再继续看这个递归 SQL 的作用：

```
SQL> select node,owner,name from syn$ where obj#=841;
```

NODE	OWNER	NAME
	SYS	V_\$PARAMETER

原来这个 SQL 获得的是同义词的底层对象，这里得到了 V_\$PARAMETER，继续向下看：

```
PARSING IN CURSOR #8 len=37 dep=1 uid=0 oct=3 lid=0 tim=1092440257074273 hv=3468666020
id='576db210'

select text from view$ where rowid=:1

END OF STMT

PARSE #8:c=0,e=1214,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=0,tim=1092440257074242

BINDS #8:

  bind 0: dty=11 mxl=16(16) mal=00 scl=00 pre=00 oacflg=18 oacfl2=1 size=16 offset=0
         bfp=b7018770 bln=16 avl=16 flg=05
         value=000001CD.0013.0001

EXEC #8:c=0,e=972,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=4,tim=1092440257075602
```

这里 Oracle 执行查询访问 view\$视图，获得视图定义文本，看一下这里访问的是什么对

象，绑定变量传入的 rowid 值为 000001CD.0013.0001，注意这是个受限 rowid，查询时需要转换一下处理：

```
SQL> select obj# from view$ where dbms_rowid.rowid_to_restricted(rowid,0) = '000001CD.0013.0001';
```

```
OBJ#
```

```
-----
```

```
840
```

```
SQL> select object_name,object_type from dba_objects where object_id=840;
```

```
OBJECT_NAME                                OBJECT_TYPE
```

```
-----
```

```
V_$PARAMETER                                VIEW
```

这里 Oracle 访问的正是 V_\$PARAMETER 视图的定义方式，执行查询可以得到：

```
select text from view$ where obj#=840;
```

```
TEXT
```

```
-----
```

```
select  "NUM","NAME","TYPE","VALUE","ISDEFAULT","ISSES_MODIFIABLE","ISSYS_MODIFIABLE",
ISMODIFIED","ISADJUSTED","DESCRIPTION","UPDATE_COMMENT" from v$parameter
```

至此就完成了查询中的回溯及定位，当然，实际过程中 Oracle 后台的递归操作比这还要复杂得多，感兴趣的朋友可以按照文中的方法测试研究一下，这里不再赘述。

最后总结一下 SQL 语句中 Oracle 对于对象名的解析顺序，具体如下。

- (1) Oracle 首先查看在发出命令的用户模式中是否存在表或视图。
 - (2) 如果表或视图不存在，Oracle 检查私有同义词是否存在。
 - (3) 如果私有同义词存在，将使用这个同义词所引用的对象。
 - (4) 如果私有同义词不存在，检查同名的公共同义词是否存在。
 - (5) 如果公共同义词存在，将使用这个同义词所引用的对象。
 - (6) 如果公共同义词不存在，Oracle 返回消息 “ORA-00942 table or view does not exist”。
- 用伪代码大致描述一下这个过程就是：

```
Parse for Object T
if (TABLE t or VIEW t)
    return
elseif (SYNONYM t)
    return
elseif (public SYNONYM t)
    return
else
```

```
signal ORA-00942  
end
```

—— 参考信息与建议阅读 ——

(1) 使用 SQL_TRACE 进行数据库诊断

http://www.eygle.com/case/Use.sql_trace.to.Diagnose.database.htm

(2) Oracle 数据库创建脚本 sql.bsq 文件

(3) 关于数据库 open 的深入探究

<http://www.itpub.net/199099.html>

Oracle 实例启动时，就需要分配共享内存，启动后台进程，如何分配和设置共享内存参数，对于 Oracle 来说是至关重要的。不当的内存分配轻则影响性能，重则导致数据库故障，在生产实际中不容忽视。本章就 Oracle 的内存管理问题进行探讨。

4.1 SGA 管理

4.1.1 什么是 SGA

SGA 指系统全局区 (System Global Area)，是一块用于加载数据、对象并保存运行状态和数据库控制信息的一块内存区域，在数据库实例启动时分配，当实例关闭时释放，每个实例都拥有自己的 SGA 区。

在第 1 章曾经提到，当数据库启动到 nomount 状态时，SGA 已经分配，同时启动后台进程：

```
SQL> show sga
```

Total System Global Area	338390716 bytes
Fixed Size	102076 bytes
Variable Size	133308416 bytes
Database Buffers	204800000 bytes
Redo Buffers	180224 bytes

连接到 Oracle 数据库的用户都可以共享 SGA 中的数据，通常为了更优化的性能，我们总是期望在物理内存允许的情况下，设置更高的 SGA 区，以减少物理 I/O（SGA 中数据缓冲区的增大可以有效地减少物理读）。

SGA 主要由以下几个部分组成。

(1) Buffer Cache-缓冲区高速缓存，用于存储最近使用的数据块，这些数据块可能是被修改过的，也可能是未经修改的。我们知道，在 Oracle 对数据的处理过程中，代价最昂贵的就是物理 I/O (Physical I/O) 操作了，同样的数据从内存中得到要比从磁盘上读取快得多，

所以将尽可能多的数据保存在内存中，可以减少磁盘 I/O 操作，从而提高数据库的性能。

在 Oracle 9i 之前，Buffer Cache 的设置主要由两个参数决定：db_block_buffers 和 db_block_size。

db_block_buffers 设置分配给 Buffer Cache 的缓冲区数量，这个数值乘以 db_block_size 得出的才是 Buffer Cache 的大小。

```
$ sqlplus "/ as sysdba"
SQL*Plus: Release 8.1.7.0.0 - Production on Tue Apr 11 09:58:15 2006

(c) Copyright 2000 Oracle Corporation. All rights reserved.

Connected to:

Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
With the Partitioning option
JServer Release 8.1.7.4.0 - 64bit Production

SQL> select name,value
      2  from v$parameter where name in ('db_block_buffers','db_block_size');

NAME                VALUE
-----
db_block_buffers     25000
db_block_size        8192

SQL> select
      2  (select value from v$parameter where name='db_block_buffers')
      3  *
      4  (select value from v$parameter where name='db_block_size')
      5  / 1024 /1024 Buffer_Cache_MB
      6  from dual;

BUFFER_CACHE_MB
-----
195.3125
```

从 Oracle 9i 开始，Oracle 引入了一个新的初始化参数 db_cache_size。该参数用来定义主 Block Size（db_block_size 定义的块大小）的 Default 缓冲池的大小。

db_cache_size 最小值为一个粒度（Granule）。粒度也是 Oracle 9i 引入的一个新的概念，是连续虚拟内存分配的单位，其大小取决于估计的 SGA 的总大小（SGA 总大小由 SGA_MAX_SIZE 参数得到）：

- 如果估计的 SGA 大小<128MB，则值为 4MB；
- 否则值为 16MB。

这个 Granule 大小受到一个内部隐含参数 `_ksmg_granule_size` 的控制，以下是 Oracle 9iR2 中的测试输出（不同版本可能不同）：

```
SQL> select * from v$version;
```

BANNER

```
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

```
PL/SQL Release 9.2.0.4.0 - Production
```

```
CORE      9.2.0.3.0      Production
```

```
TNS for Linux: Version 9.2.0.4.0 - Production
```

```
NLSRTL Version 9.2.0.4.0 - Production
```

```
SQL> show parameter sga_max_size
```

NAME	TYPE	VALUE

sga_max_size	big integer	219223120

```
SQL> @GetParDescrb.sql
```

```
Enter value for par: _ksmg_granule_size
```

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

```
new   6:   AND x.ksppinm LIKE '%_ksmg_granule_size%'
```

NAME	VALUE	DESCRIB

_ksmg_granule_size	16777216	granule size in bytes

```
SQL> alter system set sga_max_size=120M scope=spfile;
```

```
System altered.
```

```
SQL> startup force;
```

```
ORACLE instance started.
```

```
Total System Global Area  126948772 bytes
```

```
.....
```

```
Database mounted.
```

Database opened.

SQL> show parameter sga_max_size

NAME	TYPE	VALUE
sga_max_size	big integer	126948772

SQL> @GetParDescrb.sql

Enter value for par: _ksmg_granule_size

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

new 6: AND x.ksppinm LIKE '%_ksmg_granule_size%'

NAME	VALUE	DESCRIB
_ksmg_granule_size	4194304	granule size in bytes

本例用到的 SQL 脚本 GetParDescrb.sql 代码如下：

```
set linesize 120
col name for a30
col value for a20
col describ for a60
SELECT x.ksppinm NAME, y.ksppstvl VALUE, x.ksppdesc describ
      FROM SYS.x$ksppi x, SYS.x$ksppcv y
     WHERE x.inst_id = USERENV ('Instance')
           AND y.inst_id = USERENV ('Instance')
           AND x.indx = y.indx
           AND x.ksppinm LIKE '%&par%'
/
```

内存空间总是有限的，Oracle 管理 Buffer Cache 使用的是 LRU 算法，但是这又带来另外一个问题，很多批处理的操作（如全表扫描等）可能会导致 Buffer Cache 的刷新，将经常使用的数据“挤出”Buffer Cache，在不同版本中，Oracle 不停地改进 LRU 算法，以避免这类操作的过度影响。

但是在此之外，Oracle 提供了 Buffer Cache 的多缓冲池技术从另外一个方面来解决这个问题。所谓的多缓冲池技术是指，根据不同数据的不同访问方式，将 Buffer Cache 分为 Default、Keep 和 Recycle 池 3 个部分。对于经常使用的数据，可以在建表时就指定将其存放在 Keep 池中；对于经常一次性读取使用的数据，可以将其存放在 Recycle 池中；Keep 池中的数据倾向于一直保存，Recycle 池中的数据倾向于即时老化，而 Default 池则存放未指定存储池的数据，按照 LRU 算法管理。

默认情况下，所有表都使用 DEFAULT 池，它的大小就是数据缓冲区 Buffer Cache 的大

小，由初始化参数 `db_cache_size`（8i 中是 `db_block_size*db_block_buffers`）决定。

如果在创建数据表或修改数据表时，指定 `STORAGE (BUFFER_POOL KEEP)` 或者 `STORAGE (BUFFER_POOL RECYCLE)` 语句，就设置了这张表使用 `KEEP` 或者 `RECYCLE` 缓冲区。这两个缓冲区的大小分别由初始化参数 `db_keep_cache_size` 和 `db_recycle_cache_size` 来决定。

在 Oracle 8i 中，只能修改参数文件，然后重新起动数据库，才能使对这两个参数的修改生效。在 Oracle 9i 中，可以动态修改，来看一下 Oracle 9i 中这几个参数的设置：

```
SQL> show parameter cache_size
```

NAME	TYPE	VALUE
db_16k_cache_size	big integer	0
db_2k_cache_size	big integer	20971520
db_32k_cache_size	big integer	0
db_4k_cache_size	big integer	0
db_8k_cache_size	big integer	0
db_cache_size	big integer	4194304
db_keep_cache_size	big integer	0
db_recycle_cache_size	big integer	0

```
SQL> alter system set db_keep_cache_size=4M;
```

System altered.

```
SQL> alter system set db_recycle_cache_size=4M;
```

System altered.

```
SQL> show parameter cache_size
```

NAME	TYPE	VALUE
db_16k_cache_size	big integer	0
db_2k_cache_size	big integer	20971520
db_32k_cache_size	big integer	0
db_4k_cache_size	big integer	0
db_8k_cache_size	big integer	0
db_cache_size	big integer	4194304

```

db_keep_cache_size          big integer 4194304
db_recycle_cache_size       big integer 4194304
SQL>

```

同时还可以看到，在 Oracle 9i 中存在一系列的 `db_nk_cache_size` 参数，这是 Oracle 9i 中引入的多块大小支持。

Oracle 9i 允许在同一个数据库中存在多种 `Block_size` 的表空间，分别支持：2k、4k、8k、16k 和 32k 的 `block_size`，其中，由 `db_block_size` 定义的块大小被称为主 `block_size`。

如果在数据库创建不同 `block_size` 的表空间，则需要分别设定 `db_nk_cache_size` 参数。各缓冲池的设置，可以通过查询 `v$buffer_pool` 得到：

```

SQL> select id,name,block_size,current_size,target_size
2   from v$buffer_pool;

```

ID	NAME	BLOCK_SIZE	CURRENT_SIZE	TARGET_SIZE
1	KEEP	8192	4	4
2	RECYCLE	8192	4	4
3	DEFAULT	8192	4	4
4	DEFAULT	2048	20	20

(2) Shared Pool（共享池）包含共享内存结构，如 SQL 区等。SQL 区包含 SQL 解析树、执行计划等信息，通过共享池，反复执行的 SQL 可以在不同 session 间得到共享。共享池的大小由参数 `shared_pool_size` 定义，在 Oracle 9i 中，最小值为一个 Granule 大小。

关于共享池的设置和优化是非常重要和复杂的，将在下一章进行专题探讨。

(3) Redo Log Buffer（日志缓冲区）存储重做日志条目（redo entries），日志记录数据库变更，最终将被写出到重做日志文件中，在数据库崩溃或故障时用于恢复；如果数据库运行在归档模式下，最终日志文件还会被写出到归档日志中，这些归档可以在介质恢复时用于进行数据恢复。日志缓冲区的大小由初始化参数 `log_buffer` 决定。

(4) Large Pool（大池）是 SGA 的一个可选组件，通常用于共享服务器模式（MTS）、并行计算或 RMAN 的备份恢复等操作。

(5) Java Pool（Java 池）主要用于 JVM 等 Java 选件。

(6) Streams pool 是 Oracle 10g 引入的概念，为 Oracle 的 Streams 功能所使用，如果未定义该参数，这部分内存将从 Shared Pool 中分配。

从图 4-1 中可以看一下 Oracle SGA 的组成。

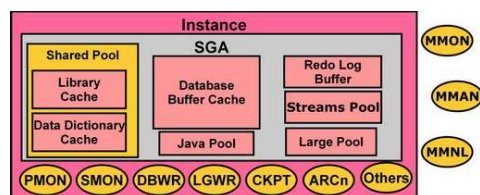


图 4-1 Oracle Instance 简图

对于 SGA 各部分设置，可以从数据库的视图中查询得到（以下数据来自 Oracle 9iR2）：

```
SQL> select * from v$sga;
```

NAME	VALUE
-----	-----
Fixed Size	731632
Variable Size	268435456
Database Buffers	117440512
Redo Buffers	811008

在 SQL*Plus 中通过一条常用的命令 show sga 看到的结果和以上查询相同：

```
SQL> show sga
```

```
Total System Global Area  387418608 bytes
Fixed Size                  731632 bytes
Variable Size               268435456 bytes
Database Buffers           117440512 bytes
Redo Buffers                811008 bytes
```

这里的 SGA 总和（387418608 bytes）受参数 SGA_MAX_SIZE 设置的影响：

```
SQL> show parameter sga_max_size
```

NAME	TYPE	VALUE
-----	-----	-----
sga_max_size	big integer	387418608

Fixed Size 部分是 SGA 中的固定部分，包含数据库和实例的状态等通用信息，后台进程需要访问这部分信息，不存储用户数据，通常只需要很小部分内存。

在 Oracle 9i 中，Variable Size 包括 shared_pool_size、java_pool_size 和 large_pool_size 部分，SGA_MAX_SIZE 去除 db_cache_size 部分也被归入可变部分，所以很多时候看到的可变部分内存要远高于可变内存组件大小：

```
SQL> select
```

```
2  (select value from v$parameter where name='large_pool_size') +
3  (select value from v$parameter where name='shared_pool_size') +
4  (select value from v$parameter where name='java_pool_size') Vsize
5  from dual;
```

```
VSIZE
```

```
-----
134217728
```

Database Buffers 指 Buffer Cache 的设置：

```
SQL> show parameter db_cache_size
```

NAME	TYPE	VALUE
db_cache_size	big integer	117440512

Redo Buffers 指日志缓冲区分配的内存大小，这个参数值通常比 log_buffers 参数设置略大：

```
SQL> show parameter log_buffer
```

NAME	TYPE	VALUE
log_buffer	integer	524288

这是因为 Log Buffer 并非按照数据块大小分配，在内存中通常需要设置保护页对 Log Buffer 进行保护。当前 SGA 的分配和使用具体信息还可以通过 V\$SGASTAT 视图查询得到：

```
SQL> select * from v$sgastat;
```

POOL	NAME	BYTES
	fixed_sga	731632
	buffer_cache	117440512
	log_buffer	787456
shared pool	krvxrr	253056
shared pool	enqueue	309528
shared pool	KGK heap	7000
shared pool	KQR L PO	445472
shared pool	KQR M PO	102944
shared pool	KQR S SO	3088
shared pool	sessions	459680
shared pool	sql area	3026184
shared pool	IM buffer	2098176
shared pool	KGLS heap	570392
shared pool	processes	193200
shared pool	db_handles	174000
shared pool	parameters	3192
shared pool	free memory	119517264
shared pool	transaction	327536
shared pool	PL/SQL DIANA	686416

shared pool FileOpenBlock	1191104
shared pool PL/SQL MPCODE	725416
shared pool library cache	4384456
shared pool miscellaneous	6799984
shared pool MTTR advisory	68344
shared pool PLS non-lib hp	3208
shared pool joxs heap init	4240
shared pool sim memory hea	180624
shared pool table definiti	448
shared pool trigger defini	2464
shared pool trigger inform	1752
shared pool trigger source	264
shared pool Checkpoint queue	513280
shared pool VIRTUAL CIRCUITS	349120
shared pool dictionary cache	3229952
shared pool KSXR receive buffers	1034000
shared pool character set object	274528
shared pool FileIdentificatonBlock	349824
shared pool message pool freequeue	940944
shared pool KSXR pending messages que	853952
shared pool event statistics per sess	1909440
shared pool fixed allocation callback	472
large pool free memory	16777216

42 rows selected.

大家可能也会注意到，在 V\$SGASTAT 中显示的 Shared Pool 大小和 shared_pool_size 设置的仍然不同，这是因为在共享池内存的分配和使用过程中会存在一定量的额外消耗，这部分内存存在 Oracle 10g 中被单独列出：

```
[oracle@danaly ~]$ sqlplus "/" as sysdba
```

```
SQL*Plus: Release 10.2.0.1.0 - Production on Wed Apr 12 18:37:41 2006
```

```
Copyright (c) 1982, 2005, Oracle. All rights reserved.
```

```
Connected to:
```

```
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
```

```
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options
```

```
SQL> select * from v$sgainfo;
```

NAME	BYTES RESIZEABLE
Fixed SGA Size	1222744 No
Redo Buffers	7163904 No
Buffer Cache Size	830472192 Yes
Shared Pool Size	96468992 Yes
Large Pool Size	4194304 Yes
Java Pool Size	4194304 Yes
Streams Pool Size	0 Yes
Granule Size	4194304 No
Maximum SGA Size	943718400 No
Startup overhead in Shared Pool	46137344 No
Free SGA Memory Available	0

```
11 rows selected.
```

关于 SGA 各参数的常规推荐设置，曾经是一个广为争议的话题，本书在不同章节将进行分别介绍；Biti 曾经在 ITPUB 的技术丛书《Oracle 数据库 DBA 专题技术精粹》一书中专门论及，大家也可以参考相关内容。

4.1.2 SGA 与共享内存

SGA 的设置 Linux/UNIX 上和一个操作系统内核参数有关，这个参数是 `shmmax`。不同操作系统，该参数设置的位置不同，在 Solaris 上，该参数由 `/etc/system` 文件中 `shmsys:shminfo_shmmax` 定义；在 Linux 上，该参数由 `/proc/sys/kernel/shmmax` 参数定义。

很多人将该参数理解为共享内存的大小，这是不对的。实际上 `shmmax` 内核参数定义的是系统允许的单个共享内存段的最大值，如果该参数设置小于 Oracle SGA 设置，那么 SGA 仍然可以创建成功，但是会被分配多个共享内存段。通常推荐通过调整 `shmmax` 设置，将 SGA 限制在一个共享内存段中。

在 Windows 系统中，由于系统采用多线程服务器（所有 Oracle Server Process 实际上都是一个进程中的线程），所以不存在共享内存的问题，无需进行特殊设置。

下面以 32 位 Linux 平台为例，来看一下 `shmmax` 参数对于数据库的影响。Linux 上该参数的缺省值通常为 32MB。

```
[root@neirong root]# more /proc/sys/kernel/shmmax
33554432
```

本例的操作系统版本为：

```
[root@neirong root]# cat /etc/redhat-release
Red Hat Enterprise Linux AS release 3 (Taroon Update 2)
[root@neirong root]# uname -r
2.4.21-15.ELsmp
```

可以通过 `ipcs` 命令查看此设置下共享内存的分配, 可以看到 Oracle 分配了多个共享内存段以满足 SGA 设置的需要:

```
[root@neirong root]# ipcs -sa

----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
0x00000000	884736	oracle	640	4194304	14	
0x00000000	917505	oracle	640	33554432	14	
0x00000000	950274	oracle	640	33554432	14	
0x00000000	983043	oracle	640	33554432	14	
0x00000000	1015812	oracle	640	33554432	14	
0x00000000	1048581	oracle	640	33554432	14	
0x00000000	1081350	oracle	640	33554432	14	
0x00000000	1114119	oracle	640	33554432	14	
0x00000000	1146888	oracle	640	33554432	14	
0x00000000	1179657	oracle	640	33554432	14	
0x00000000	1212426	oracle	640	33554432	14	
0x00000000	1245195	oracle	640	33554432	14	
0x00000000	1277964	oracle	640	33554432	14	
0x00000000	1310733	oracle	640	33554432	14	
0x00000000	1343502	oracle	640	33554432	14	
0x00000000	1376271	oracle	640	33554432	14	
0x00000000	1409040	oracle	640	33554432	14	
0x00000000	1441809	oracle	640	33554432	14	
0x00000000	1474578	oracle	640	33554432	14	
0x00000000	1507347	oracle	640	33554432	14	
0x00000000	1540116	oracle	640	33554432	14	
0x00000000	1572885	oracle	640	33554432	14	
0x00000000	1605654	oracle	640	33554432	14	
0x00000000	1638423	oracle	640	33554432	14	
0x00000000	1671192	oracle	640	33554432	14	
0x00000000	1703961	oracle	640	33554432	14	
0x7a9c9900	1736730	oracle	640	4194304	56	

----- Semaphore Arrays -----

key	semid	owner	perms	nsems
0xfc02e10	229376	oracle	640	154

----- Message Queues -----

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

可以看到为了创建 Oracle 的 SGA，系统共分配了 27 个共享内存段。

针对一个后台进程，使用 pmap 工具可以看到每个共享内存段的地址空间：

```
[root@neirong root]# ps -ef|grep dbw
```

```
oracle    3102      1  0 09:27 ?        00:00:26 ora_dbw0_hsmkt
root      7018   6923  0 15:48 pts/1    00:00:00 grep dbw
```

```
[root@neirong root]# pmap 3102
```

```
ora_dbw0_hsmkt[3102]
```

```
08048000 (37308 KB)    r-xp (68:06 1525072)  /opt/oracle/product/9.2.0/bin/oracle
0a4b7000 (8804 KB)    rw-p (68:06 1525072)  /opt/oracle/product/9.2.0/bin/oracle
0ad50000 (380 KB)      rw-p (00:00 0)
50000000 (4096 KB)    rw-s (00:04 884736)   /SYSV00000000
51000000 (32768 KB)   rw-s (00:04 917505)   /SYSV00000000
53000000 (32768 KB)   rw-s (00:04 950274)   /SYSV00000000
55000000 (32768 KB)   rw-s (00:04 983043)   /SYSV00000000
57000000 (32768 KB)   rw-s (00:04 1015812)  /SYSV00000000
59000000 (32768 KB)   rw-s (00:04 1048581)  /SYSV00000000
5b000000 (32768 KB)   rw-s (00:04 1081350)  /SYSV00000000
5d000000 (32768 KB)   rw-s (00:04 1114119)  /SYSV00000000
5f000000 (32768 KB)   rw-s (00:04 1146888)  /SYSV00000000
61000000 (32768 KB)   rw-s (00:04 1179657)  /SYSV00000000
63000000 (32768 KB)   rw-s (00:04 1212426)  /SYSV00000000
65000000 (32768 KB)   rw-s (00:04 1245195)  /SYSV00000000
67000000 (32768 KB)   rw-s (00:04 1277964)  /SYSV00000000
69000000 (32768 KB)   rw-s (00:04 1310733)  /SYSV00000000
6b000000 (32768 KB)   rw-s (00:04 1343502)  /SYSV00000000
6d000000 (32768 KB)   rw-s (00:04 1376271)  /SYSV00000000
6f000000 (32768 KB)   rw-s (00:04 1409040)  /SYSV00000000
71000000 (32768 KB)   rw-s (00:04 1441809)  /SYSV00000000
73000000 (32768 KB)   rw-s (00:04 1474578)  /SYSV00000000
```

```

75000000 (32768 KB)    rw-s (00:04 1507347)  /SYSV00000000
77000000 (32768 KB)    rw-s (00:04 1540116)  /SYSV00000000
79000000 (32768 KB)    rw-s (00:04 1572885)  /SYSV00000000
7b000000 (32768 KB)    rw-s (00:04 1605654)  /SYSV00000000
7d000000 (32768 KB)    rw-s (00:04 1638423)  /SYSV00000000
7f000000 (32768 KB)    rw-s (00:04 1671192)  /SYSV00000000
81000000 (32768 KB)    rw-s (00:04 1703961)  /SYSV00000000
83000000 (4 KB)        r--s (00:04 1736730)  /SYSV7a9c9900
83001000 (644 KB)      rw-s (00:04 1736730)  /SYSV7a9c9900
830a2000 (4 KB)        r--s (00:04 1736730)  /SYSV7a9c9900
830a3000 (3444 KB)     rw-s (00:04 1736730)  /SYSV7a9c9900
b6ec2000 (44 KB)       r-xp (68:03 32811)   /lib/libnss_files-2.3.2.so
b6ecd000 (4 KB)        rw-p (68:03 32811)   /lib/libnss_files-2.3.2.so
b6ece000 (512 KB)      rw-p (68:03 40360)   /dev/zero
b6f4e000 (1140 KB)     rw-p (00:00 0)
b706b000 (1224 KB)     r-xp (68:03 114692)  /lib/tls/libc-2.3.2.so
b719d000 (12 KB)       rw-p (68:03 114692)  /lib/tls/libc-2.3.2.so
b71a0000 (12 KB)       rw-p (00:00 0)
b71a3000 (72 KB)       r-xp (68:03 32795)   /lib/libnsl-2.3.2.so
b71b5000 (4 KB)        rw-p (68:03 32795)   /lib/libnsl-2.3.2.so
b71b6000 (8 KB)        rw-p (00:00 0)
.....
b75e7000 (4 KB)        r-xp (68:03 101245)  /etc/libcwait.so
b75e8000 (4 KB)        rw-p (68:03 101245)  /etc/libcwait.so
b75ea000 (4 KB)        rw-p (00:00 0)
b75eb000 (84 KB)       r-xp (68:03 32778)   /lib/ld-2.3.2.so
b7600000 (4 KB)        rw-p (68:03 32778)   /lib/ld-2.3.2.so
bfff8000 (32 KB)       rwxp (00:00 0)
mapped: 881332 KB      writable/private: 12056 KB      shared: 827392 KB

```

为了避免多个共享内存段，可以修改 `shmmax` 内核参数，使 SGA 存在于一个共享内存段中。通过修改 `/proc/sys/kernel/shmmax` 参数可以达到此目的。

```

[root@neirong root]# echo 1073741824 > /proc/sys/kernel/shmmax
[root@neirong root]# more /proc/sys/kernel/shmmax
1073741824

```

这里修改为 1GB。对于 `shmmax` 文件的修改，系统重新启动后会复位。可以通过修改 `/etc/sysctl.conf` 文件使更改永久化。

在该文件内添加以下一行，这个更改在系统重新启动后生效：

```
kernel.shmmax = 1073741824
```

修改 shmmax 之后，重启数据库使更改生效。

关闭数据库：

```
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> !
```

注意观察，关闭数据库之后，此时共享内存段已经释放：

```
[oracle@neirong oracle]$ ipcs -sa
```

```
----- Shared Memory Segments -----
```

key	shmid	owner	perms	bytes	nattch	status
-----	-------	-------	-------	-------	--------	--------

```
----- Semaphore Arrays -----
```

key	semid	owner	perms	nsems
-----	-------	-------	-------	-------

```
----- Message Queues -----
```

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

```
[oracle@neirong oracle]$ exit
```

```
exit
```

启动数据库：

```
SQL> startup
ORACLE instance started.
```

```
Total System Global Area  839980852 bytes
Fixed Size                  452404 bytes
Variable Size               201326592 bytes
Database Buffers            637534208 bytes
Redo Buffers                 667648 bytes
Database mounted.
Database opened.
```

此时观察，可以看到共享内存只需要一个内存段分配：

```
SQL> ! ipcs -sa
```

```

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x7a9c9900 1769472   oracle     640        859832320  35

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0xfc02e10 360448   oracle     640        154

----- Message Queues -----
key          msqid      owner      perms      used-bytes  messages

```

此时查看后台进程的 pmap 地址映射显示为：

```

[oracle@neirong bdump]$ pmap 4178
ora_lgwr_hsmkt[4178]
08048000 (37308 KB)   r-xp (68:06 1525072) /opt/oracle/product/9.2.0/bin/oracle
0a4b7000 (8804 KB)   rw-p (68:06 1525072) /opt/oracle/product/9.2.0/bin/oracle
0ad50000 (3320 KB)   rw-p (00:00 0)
50000000 (835584 KB)   rw-s (00:04 1835008) /SYSV7a9c9900
83000000 (4 KB)      r--s (00:04 1835008) /SYSV7a9c9900
83001000 (644 KB)   rw-s (00:04 1835008) /SYSV7a9c9900
830a2000 (4 KB)      r--s (00:04 1835008) /SYSV7a9c9900
830a3000 (3444 KB)   rw-s (00:04 1835008) /SYSV7a9c9900
b6bb7000 (4112 KB)   rw-p (00:00 0)
b6fbb000 (44 KB)     r-xp (68:03 32811)  /lib/libnss_files-2.3.2.so
b6fc6000 (4 KB)     rw-p (68:03 32811)  /lib/libnss_files-2.3.2.so
b6fc7000 (512 KB)   rw-p (68:03 40360)  /dev/zero
b7047000 (144 KB)   rw-p (00:00 0)
.....
b75ea000 (4 KB)     rw-p (00:00 0)
b75eb000 (84 KB)    r-xp (68:03 32778)  /lib/ld-2.3.2.so
b7600000 (4 KB)     rw-p (68:03 32778)  /lib/ld-2.3.2.so
bffc000 (16 KB)     rwxp (00:00 0)
mapped: 899660 KB      writable/private: 18096 KB      shared: 839680 KB

```

实际上，如果没有修改 shmmax 参数，Oracle 在启动过程中就会在 alert_<sid>.log 文件中记录如下警告：

```

Starting ORACLE instance (normal)
Thu Nov 17 09:27:29 2005
WARNING: EINVAL creating segment of size 0x0000000033400000

```

```
fix shm parameters in /etc/system or equivalent
```

这是一个 **WARNING** 的提示，说明是建议修正，但并非强制的内容。

在 Solaris 平台上，有时候也会看到类似的警报：

```
Sun Apr 30 05:35:20 2006
```

```
Starting ORACLE instance (normal)
```

```
Sun Apr 30 05:35:20 2006
```

```
WARNING: Not enough physical memory for SHM_SHARE_MMU segment of size 0x000000006d400000
flag=0x4000]
```

这通常是因为 **SGA** 设置过大，超过了物理内存而导致的，这种情况通过修正参数即可解决。

有时候这类警告也可能是因为数据库异常关闭，后台进程未正常退出，共享内存未及时释放引起的，对于这种情况，可以通过 **ipcs** 命令找到共享内存段 **id** (shared memory id)，然后通过 **ipcrm** 命令可以强制释放该共享内存段，完成这些特殊处理后，数据库通常就可以正常启动了。

4.1.3 SGA 管理的变迁

在 Oracle 9i 之前，**SGA** 一直是静态分配内存的。**SGA** 分配的内存空间可以被所有的 Oracle 进程/线程所共享，内存的大小是根据 **init.ora** 参数文件中的值计算得来的，一旦分配完毕，可用共享内存的大小就不能增大或缩小，如果 **DBA** 要增加数据库块缓冲区的数量，必须首先关闭例程修改初始化参数文件，然后重新启动该例程。

让我们从 Oracle 8i 开始，研究一下 Oracle **SGA** 管理的变迁。

1. Oracle 8i 中静态 SGA 的管理

在 Oracle 8i 之中，当需要修改 **SGA** 参数时（如 **shared_pool_size**），必需修改参数文件，重新启动数据库之后，修改才能生效。

可以从参数文件中找到这些重要参数，下面是一个 Oracle 8i 数据库的设置示例：

```
db_block_buffers = 25000
shared_pool_size = 104857600
large_pool_size = 8388608
java_pool_size = 10485760
log_buffer = 163840
```

当然这些参数的设置要受物理内存的限制，需要全面考虑。

2. Oracle 9i 动态 SGA 管理

从 Oracle 9i 开始，Oracle 推出了动态 **SGA** 调整，也就是说，允许不重新启动数据库而使得 **SGA** 的修改生效。

在 Oracle 9i 中，可以设置参数 **SGA_MAX_SIZE**，该参数用以控制各缓冲池使用的内存总和，本质上是在进程中预先分配一段虚拟地址备用而不分配物理内存，目的是防止和进程

私有地址段的冲突：

```
SQL> show parameter sga_max_size
```

NAME	TYPE	VALUE
sga_max_size	big integer	387418608

设置了该参数之后，可以通过在线方式修改 Oracle SGA 各内存组件的内存分配，经常可能用到类似如下命令（关于 Scope 参数等说明可以参考上一章的内容）：

```
alter system set db_cache_size = 2g scope=memory;
```

```
alter system set large_pool_size = 200m scope=memory;
```

```
alter system set java_pool_size = 200m scope=memory;
```

只要总的 SGA 内存设置不超过 SGA_MAX_SIZE 的设置，更改都可以立即生效（但是需要注意的是，在 Oracle 9iR1 中，动态减小内存设置可能会触发一些 Bug，在繁忙的生产系统中，缩减各组件的内存使用应该是相当慎重的）：

```
SQL> alter system set db_cache_size=100M;
```

```
System altered.
```

如果内存不足，则 Oracle 会给出错误，提示内存缺乏：

```
SQL> alter system set db_cache_size=500M;
```

```
alter system set db_cache_size=500M
```

```
*
```

```
ERROR at line 1:
```

```
ORA-02097: parameter cannot be modified because specified value is invalid
```

```
ORA-00384: Insufficient memory to grow cache
```

当然在动态修改这些参数时，存在一些常见的限制：

- （1）修改的内存大小必须是粒度（Granule）大小的整数倍，否则会自动向上取整；
- （2）SGA 总大小不能超过 SGA_MAX_SIZE；
- （3）SGA 最低配置为 3 个粒度（Granule），一个粒度用于固定的 SGA（包括重做缓冲区），一个粒度用于缓冲区高速缓存，一个粒度用于共享池。

通过 OEM，可以直观地了解一下 Oracle 9i SGA 各部分参数设置，如图 4-2 所示。

伴随着动态 SGA 管理的新特性，Oracle 推出了一系列内存设置建议功能，同时引入了一系列动态性能视图：

```
SQL> select tname from tab
```

```
2 where tname like '%ADVICE%';
```

TNAME

GV_\$DB_CACHE_ADVICE

GV_\$MTTR_TARGET_ADVICE

GV_\$PGATARGET_ADVICE_HISTOGRAM

GV_\$PGA_TARGET_ADVICE

GV_\$SHARED_POOL_ADVICE

V_\$DB_CACHE_ADVICE

V_\$MTTR_TARGET_ADVICE

V_\$PGA_TARGET_ADVICE

V_\$PGA_TARGET_ADVICE_HISTOGRAM

V_\$SHARED_POOL_ADVICE

10 rows selected.



图 4-2 Oracle 9i SGA 各部分参数设置

其中和 SGA 相关的是 V\$DB_CACHE_ADVICE 和 V\$SHARED_POOL_ADVICE，这些新功能通过在数据库运行时持续不断地收集信息，从而对内存的设置提供建议。

缓冲区高速缓存建议 (Buffer Cache Advisory) 受初始化参数 DB_CACHE_ADVICE 控制。该参数为动态参数，可用的值有 3 个 OFF、ON 和 READY。

DB_CACHE_ADVICE 不同参数值的含义分别如下。

- OFF：关闭建议并且不为建议分配内存。
- ON：开启建议并且 CPU 和内存开销都会发生。
- READY：关闭建议但是仍保留为建议分配的内存。

在某些版本中，如果在参数为 OFF 状态时，尝试将其设置为 ON，可能会出现 ORA-4031 错误，无法从共享池中分配内存；如果参数处于 READY 状态则可以将其设置为 ON 而不会

发生错误，这是因为需要的内存已经分配。

来看一下一个生产数据库中，Oracle 收集的 Buffer Cache 建议信息：

```
SQL> show parameter db_cache_ad
```

NAME	TYPE	VALUE
db_cache_advice	string	ON

```
SQL> select id,name,block_size,size_for_estimate sfe,size_factor sf,
2   estd_physical_read_factor eprf,estd_physical_reads epr
3   from v$db_cache_advice;
```

ID	NAME	BLOCK_SIZE	SFE	SF	EPRF	EPR
3	DEFAULT	8192	96	0.0938	63.0018	2579371363
3	DEFAULT	8192	192	0.1875	28.9921	1186971632
3	DEFAULT	8192	288	0.2813	17.2038	7043452876
3	DEFAULT	8192	384	0.375	10.577	4330360330
3	DEFAULT	8192	480	0.4688	6.3443	2597429002
3	DEFAULT	8192	576	0.5625	4.1087	1682151293
3	DEFAULT	8192	672	0.6563	2.897	1186065595
3	DEFAULT	8192	768	0.75	2.1604	8844976015
3	DEFAULT	8192	864	0.8438	1.5728	6439364907
3	DEFAULT	8192	960	0.9375	1.1605	4751342947
3	DEFAULT	8192	1024	1	1	4094120994
3	DEFAULT	8192	1056	1.0313	0.937	3836222877
3	DEFAULT	8192	1152	1.125	0.7783	3186337904
3	DEFAULT	8192	1248	1.2188	0.6519	2669051919
3	DEFAULT	8192	1344	1.3125	0.534	2186279350
3	DEFAULT	8192	1440	1.4063	0.4296	1758909643
3	DEFAULT	8192	1536	1.5	0.3529	1444916209
3	DEFAULT	8192	1632	1.5938	0.2946	1206059096
3	DEFAULT	8192	1728	1.6875	0.2416	989138472
3	DEFAULT	8192	1824	1.7813	0.2031	831457344
3	DEFAULT	8192	1920	1.875	0.1712	700889664

21 rows selected

可以看到，伴随 `db_cache_size` 的增大，估计的物理读（`estd_physical_reads`）在逐渐减少，我们的选择就在于在 `db_cache_size` 的设置和 `physical_reads` 之间寻找一个边际效益最高点，使用可以接受的内存设置，获得尽量低的物理读。

在 Oracle 9i 中，OEM 中提供了对以上数据的图形化展现，如图 4-3 所示，可以清楚地看到趋势曲线。

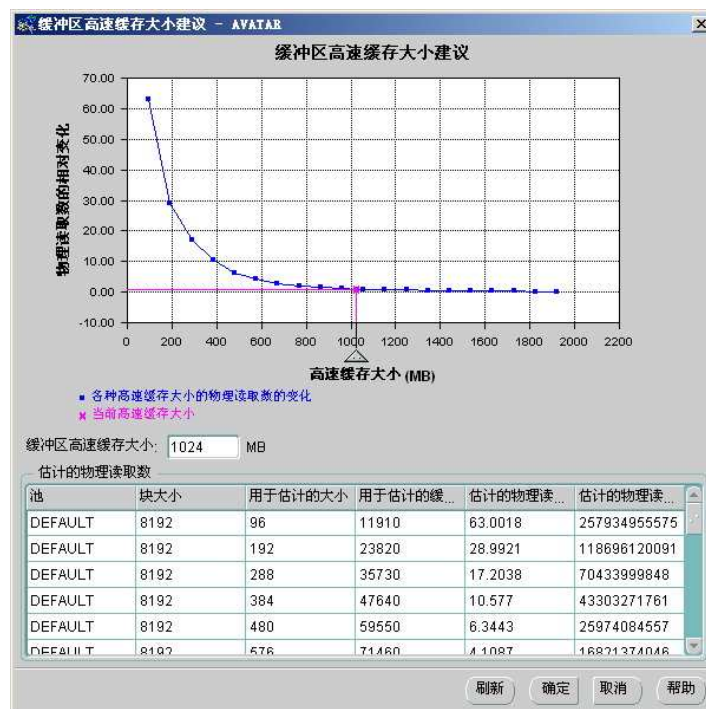


图 4-3 缓冲区高速缓存大小建议

而对于 Shred Pool 的建议，则受到另外一个初始化参数的影响，这个参数是：`STATISTICS_LEVEL`。

`STATISTICS_LEVEL` 控制数据库收集的统计信息的级别，该参数有 3 个选项。

- **BASIC**: 收集基本的统计信息。
- **TYPICAL**: 收集大部分的统计信息，这是系统的缺省设置，为了从 Oracle 9i/10g 的新特性中受益，用户始终应该将该参数设置为典型。
- **ALL**: 收集全部的统计信息。

可以通过 `v$statistics_level` 视图来查看该参数的影响范围：

```
SQL>select STATISTICS_NAME,SESSION_STATUS,
SYSTEM_STATUS,ACTIVATION_LEVEL,SESSION_SETTABLE from v$statistics_level;
```

STATISTICS_NAME	SESSION	SYSTEM	S	ACTIVAT	SES
Buffer Cache Advice	ENABLED	ENABLED	TYPICAL	NO	
MTTR Advice	ENABLED	ENABLED	TYPICAL	NO	

Timed Statistics	ENABLED	ENABLED	TYPICAL	YES
Timed OS Statistics	DISABLED	DISABLED	ALL	YES
Segment Level Statistics	ENABLED	ENABLED	TYPICAL	NO
PGA Advice	ENABLED	ENABLED	TYPICAL	NO
Plan Execution Statistics	DISABLED	DISABLED	ALL	YES
Shared Pool Advice	ENABLED	ENABLED	TYPICAL	NO

8 rows selected.

可以看到在 TYPICAL 设置下，除 Timed OS Statistics 和 Plan Execution Statistics 信息不收集外，其他信息都被收集。

其中，Buffer Cache Advice 受 db_cache_advice 参数独立控制，Timed Statistics 受 timed_statistics 参数独立控制。其他统计信息的收集都受到 STATISTICS_LEVEL 参数的控制。

当修改 STATISTICS_LEVEL 为 Basic 时，可以看到，除 Buffer Cache Advice 和 Timed Statistics 外，其他信息收集都被禁止。

```
SQL> alter system set statistics_level=basic;
```

System altered.

```
SQL> select STATISTICS_NAME, SESSION_STATUS,
SYSTEM_STATUS, ACTIVATION_LEVEL, SESSION_SETTABLE from v$statistics_level;
```

STATISTICS_NAME	SESSION_	SYSTEM_	S	ACTIVAT	SES

Buffer Cache Advice	ENABLED	ENABLED	TYPICAL	NO	
MTTR Advice	DISABLED	DISABLED	TYPICAL	NO	
Timed Statistics	ENABLED	ENABLED	TYPICAL	YES	
Timed OS Statistics	DISABLED	DISABLED	ALL	YES	
Segment Level Statistics	DISABLED	DISABLED	TYPICAL	NO	
PGA Advice	DISABLED	DISABLED	TYPICAL	NO	
Plan Execution Statistics	DISABLED	DISABLED	ALL	YES	
Shared Pool Advice	DISABLED	DISABLED	TYPICAL	NO	

8 rows selected.

可以通过查询 V\$SHARED_POOL_ADVICE 视图获得关于 Shared Pool 的建议信息：

```
SQL> select SHARED_POOL_SIZE_FOR_ESTIMATE SPSFE, SHARED_POOL_SIZE_FACTOR SPSF,
2 ESTD_LC_SIZE, ESTD_LC_MEMORY_OBJECTS ELMO, ESTD_LC_TIME_SAVED ELTS,
3 ESTD_LC_TIME_SAVED_FACTOR ELTSF, ESTD_LC_MEMORY_OBJECT_HITS ELMOH
```

```
4 from v$shared_pool_advice;
```

SPSFE	SPSF	ESTD_LC_SIZE	ELMO	ELTS	ELTSF	ELMOH
208	0.52	193	17972	8295638	0.9999	359670096
256	0.64	240	22083	8296182	0.9999	359708386
304	0.76	289	26310	8296446	1	359725903
352	0.88	336	30330	8296589	1	359735662
400	1	383	34632	8296693	1	359745443
448	1.12	430	42078	8296767	1	359752957
496	1.24	479	50650	8296816	1	359757737
544	1.36	526	59499	8296855	1	359761517
592	1.48	573	68609	8296891	1	359764717
640	1.6	620	77522	8296930	1	359767727
688	1.72	667	85969	8296953	1	359769641
736	1.84	716	96051	8296968	1	359770920
784	1.96	764	105940	8296987	1	359772425
832	2.08	795	112434	8296992	1	359772998

14 rows selected

通过以上数据可以得知,当前的 shared_pool_size 大小为 400MB (SHARED_POOL_SIZE_FACTOR 为 1 的当前设置), 通过当前参数也可查看到这个设置:

```
SQL> show parameter shared_pool_size
```

NAME	TYPE	VALUE
shared_pool_size	big integer	419430400

通过 OEM, 也可以直观地看到图形显示, 如图 4-4 所示。

通过以上数据可以看到, 当 shared_pool_size 设置为 304MB 时, 即可达到和现在相同的效果, 目前的 shared_pool_size 设置浪费了部分内存, 那么就可以动态调整 shared_pool_size 参数, 释放这部分内存, 留给其他内存组件使用。

```
SQL> alter system set shared_pool_size=304M;
```

当进行动态参数修改时, 修改 session 会处于等待状态, 等待事件为 background parameter adjustment:

```
SQL> select sid,seq#,event,SECONDS_IN_WAIT,state
```

```
2 from v$session_wait where sid=80;
```

SID	SEQ#	EVENT	SECONDS_IN_WAIT	STATE
80	46479	background parameter adjustment	928	WAITING

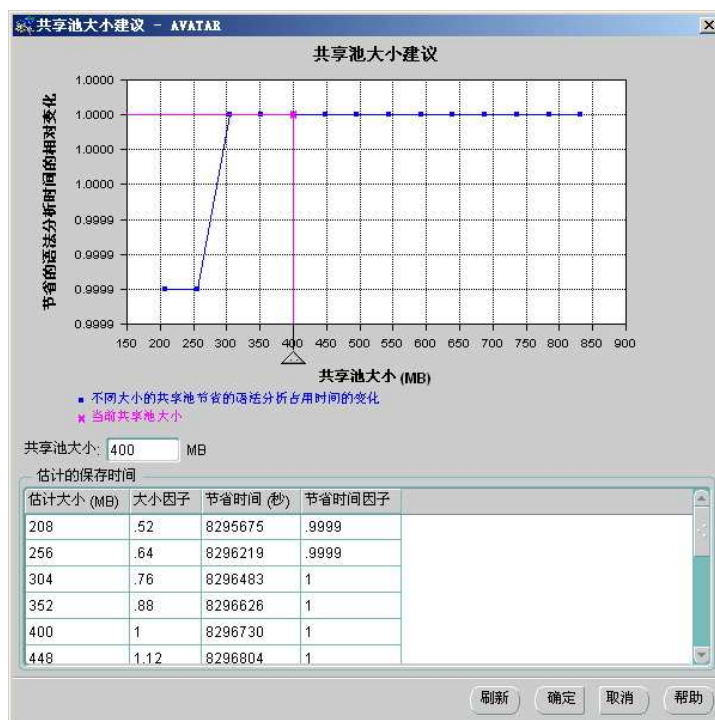


图 4-4 共享池大小建议

这个调整的时间可能极其漫长，从 v\$sqllock 视图中，还可以获得相关锁定信息：

```
SQL> select * from v$sqllock where sid=80;
```

ADDR	KADDR	SID	TYPE	ID1	ID2	LMODE
REQUEST	CTIME	BLOCK				
00000003CF3D6048	00000003CF3D6068	80	PE	44	0	4
1437	0					

锁定类型为 PE，即 Kernel Service system Parameters Enqueue，在修改系统参数时需要获取该锁定。

需要提醒的是，虽然 Oracle 9i 中，Oracle 提供了动态内存修改的功能，但是仍然建议在系统规划时做好设置，尽量避免运行时的动态调整。动态调整某些系统参数（如 undo_retention 等），在繁忙的系统中可能触发 Bug 而造成系统挂起。

3. Oracle 10g 自动共享内存管理

很多人可能注意到，Oracle 9i 的动态 SGA 调整的新特性虽然方便，但是仍然需要 DBA

去观察并修改这些设置，如果 Oracle 能够自动完成这个调整，那将是一个划时代的进步了。很快在 Oracle 10g 中，这个梦想得以实现。

在 Oracle 10g 之前，用户可能面对过这样的情况，数据库在白天需要处理大量的 OLTP 任务，这些任务需要大量的 Buffer Cache 内存，而在夜间系统可能需要运行大量的并行批处理任务，这些任务又需要大量的 Large Pool 内存，为了让这样一个系统在有限的资源下高效率运行，可能你需要在各类峰值业务来临之前对数据库进行不断的调整。

那么在 Oracle 10g 的自动共享内存管理 (Automatic Shared Memory Management, ASMM) 下，这些动作不再需要人工介入，当运行 OLTP 任务时，Buffer Cache 会获取大部分内存以达到良好的 I/O 性能。当系统需要运行 DSS 批处理任务时，内存会自动转移给 Large Pool，以便并行查询等可以获得更多的内存，更快的执行。

Oracle 10g 使用了一个新的初始化参数 SGA_TARGET，通过指定这个参数，让 Oracle 自动管理 SGA 中以下大多数的内存分配。SGA_TARGET 是个动态参数，但是该参数不能超过 SGA_MAX_SIZE 参数的设置，如果试图修改 SGA_TARGET 超越 SGA_MAX_SIZE 的限制，那么系统会给出错误信息：

```
SQL> show parameter sga_max
```

NAME	TYPE	VALUE
sga_max_size	big integer	300M

```
SQL> alter system set sga_target=400M;
```

```
alter system set sga_target=400M
```

```
*
```

```
ERROR at line 1:
```

```
ORA-02097: parameter cannot be modified because specified value is invalid
```

```
ORA-00823: Specified value of sga_target greater than sga_max_size
```

并非所有 SGA 组件都可以自动调整，可以自动分配的内存包括：

- Buffer Cache;
- Shared Pool;
- Java Pool;
- Large Pool。

启用自动共享内存管理，可以估算一个 SGA 的总大小，然后设置 SGA_TARGET 参数为非零值，Oracle 将启用自动共享内存管理。自动共享内存管理需要 STATISTICS_LEVEL 参数设置为 TYPICAL 或者 ALL。

可以通过 Oracle 10g 的 OEM 来启用或禁用自动共享内存管理，如图 4-5 所示。

为了更加直观，Oracle 还在 OEM 中提供了图形展现，提供建议数据，如图 4-6 所示。

Oracle 服务器根据系统运行的情况自动调整这些内存的大小，并记录在 spfile 中，进程重新启动时，不会丢失之前的调整结果。

但是以下几个初始化参数还是需要手工配置的：

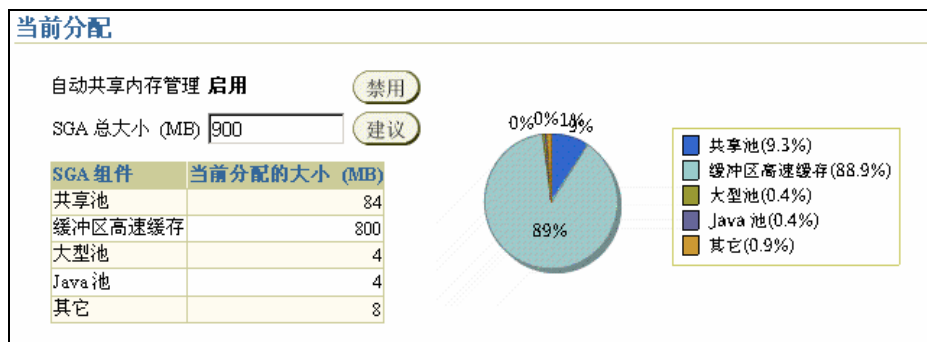


图 4-5 启用或禁用自动共享内存管理

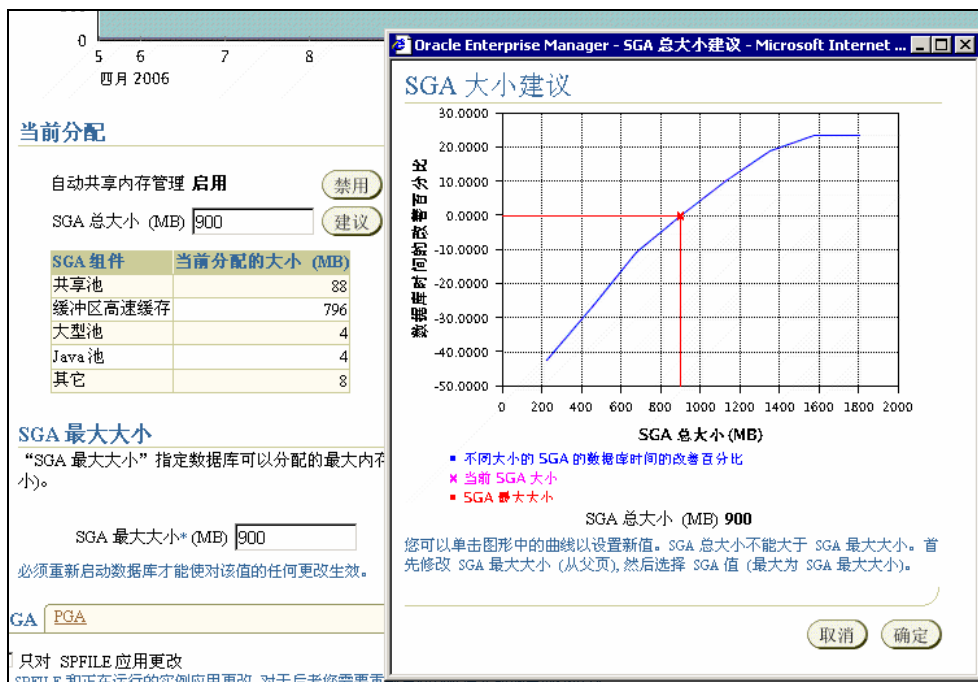


图 4-6 SGA 大小建议

- 非标准 BLOCK_SIZE 的 Cache;
- Keep/Recycle Buffer Cache;
- Redo Log Buffer;
- Stream Pool。

自动的共享内存管理引入了一个新的后台进程 MMAN (Memory Manager)。该进程用以动态调整内存组件。动态调整的依据来自系统不间断收集的内存建议。

从警告日志文件中, 可以看到该进程启动顺序为 3, 进程号为 4:

```
PMON started with pid=2, OS id=4464
PSP0 started with pid=3, OS id=4466
MMAN started with pid=4, OS id=4468
DBW0 started with pid=5, OS id=4470
```

```

LGWR started with pid=6, OS id=4472
CKPT started with pid=7, OS id=4474
SMON started with pid=8, OS id=4476
RECO started with pid=9, OS id=4478
CJQ0 started with pid=10, OS id=4480
MMON started with pid=11, OS id=4482

```

从数据库内部视图 `v$process` 中，也可以看到这些重要信息：

```
SQL> select pid,spid,program from v$process;
```

PID	SPID	PROGRAM
1		PSEUDO
2	4464	oracle@danaly.hurrray.com.cn (PMON)
3	4466	oracle@danaly.hurrray.com.cn (PSP0)
4	4468	oracle@danaly.hurrray.com.cn (MMAN)
5	4470	oracle@danaly.hurrray.com.cn (DBW0)
6	4472	oracle@danaly.hurrray.com.cn (LGWR)
7	4474	oracle@danaly.hurrray.com.cn (CKPT)
8	4476	oracle@danaly.hurrray.com.cn (SMON)
9	4478	oracle@danaly.hurrray.com.cn (RECO)
10	4480	oracle@danaly.hurrray.com.cn (CJQ0)
11	4482	oracle@danaly.hurrray.com.cn (MMON)
12	4484	oracle@danaly.hurrray.com.cn (MMNL)
13	4486	oracle@danaly.hurrray.com.cn (D000)
14	4488	oracle@danaly.hurrray.com.cn (S000)
15	4516	oracle@danaly.hurrray.com.cn (q000)
16	4491	oracle@danaly.hurrray.com.cn (ASMB)
17	4495	oracle@danaly.hurrray.com.cn (RBAL)
18	4811	oracledanaly@danaly.hurrray.com.cn
19	4783	oracledanaly@danaly.hurrray.com.cn
20	4787	oracledanaly@danaly.hurrray.com.cn
22	4508	oracle@danaly.hurrray.com.cn (O001)
23	4510	oracle@danaly.hurrray.com.cn (QMNC)
24	4518	oracle@danaly.hurrray.com.cn (q001)
25	4815	oracledanaly@danaly.hurrray.com.cn

24 rows selected

再从操作系统作一下观察：

```
[oracle@danaly bdump]$ ps -ef|grep ora_
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
oracle	4464	1	0	09:52	?	00:00:00	ora_pmon_danaly
oracle	4466	1	0	09:52	?	00:00:00	ora_psp0_danaly
oracle	4468	1	0	09:52	?	00:00:00	ora_mman_danaly
oracle	4470	1	0	09:52	?	00:01:05	ora_dbw0_danaly
oracle	4472	1	0	09:52	?	00:00:05	ora_lgwr_danaly
oracle	4474	1	0	09:52	?	00:00:01	ora_ckpt_danaly
oracle	4476	1	0	09:52	?	00:00:01	ora_smon_danaly
oracle	4478	1	0	09:52	?	00:00:00	ora_reco_danaly
oracle	4480	1	0	09:52	?	00:00:00	ora_cjq0_danaly
oracle	4482	1	0	09:52	?	00:00:02	ora_mmon_danaly
oracle	4484	1	0	09:52	?	00:00:00	ora_mmmn1_danaly
oracle	4486	1	0	09:52	?	00:00:00	ora_d000_danaly
oracle	4488	1	0	09:52	?	00:00:00	ora_s000_danaly
oracle	4491	1	0	09:52	?	00:00:00	ora_asmb_danaly
oracle	4495	1	0	09:52	?	00:00:00	ora_rbal_danaly
oracle	4508	1	0	09:53	?	00:00:00	ora_o001_danaly
oracle	4510	1	0	09:53	?	00:00:00	ora_qmnc_danaly
oracle	4516	1	0	09:53	?	00:00:00	ora_q000_danaly
oracle	4518	1	0	09:53	?	00:00:00	ora_q001_danaly
oracle	4833	4741	0	15:02	pts/0	00:00:00	grep ora_

可以注意到，在操作系统上一个进程的 PID 号，对应地，可以从数据库的 V\$PROCESS、SPID 得到，也就是说，v\$process 视图实际上是从操作系统到数据库的一个接口。可以通过 v\$process 视图把操作系统和数据库联系起来（为了不影响本文结构，将一个相关诊断案例放在本章末尾的诊断案例部分，大家可以作一个跳转读取该案例，也可以继续阅读）。

如果不想使用自动共享内存管理的新特性，Oracle 也允许使用手工管理，只需要简单地将 SGA_TARGET 参数设置为 0，Oracle 就会回到手工管理的模式，当前的各内存组件值会被计入 spfile，作为手工管理的初始值使用。

随着自动共享内存管理新特性的引入，许多相关参数的使用也发生了改变。当从开始就设置了 SGA_MAX_SIZE 参数，启用了自动共享内存管理之后，相关内存参数值会处于未设置状态：

```
SQL> col name for a40
SQL> col value for a30
SQL> select name,value
       2  from v$parameter
       3  where name in
```

```

4 ('large_pool_size','java_pool_size','shared_pool_size','streams_pool_size','db_cache_size')
5 /

```

NAME	VALUE
shared_pool_size	0
large_pool_size	0
java_pool_size	0
streams_pool_size	0
db_cache_size	0

而真正决定各组件大小的，是由一组新引入的参数决定：

```

SQL> SELECT x.ksppinm NAME, y.ksppstvl VALUE, x.ksppdesc describ
2   FROM SYS.x$ksppi x, SYS.x$ksppcv y
3   WHERE x.inst_id = USERENV ('Instance')
4   AND y.inst_id = USERENV ('Instance')
5   AND x.indx = y.indx
6   AND x.ksppinm like '%pool_size%'
7   /

```

NAME	VALUE	DESCRIB
_NUMA_pool_size	Not specified	aggregate size in bytes of NUMA pool
__shared_pool_size	96468992	Actual size in bytes of shared pool
shared_pool_size	0	size in bytes of shared pool
__large_pool_size	4194304	Actual size in bytes of large pool
large_pool_size	0	size in bytes of large pool
__java_pool_size	4194304	Actual size in bytes of java pool
java_pool_size	0	size in bytes of java pool
__streams_pool_size	0	Actual size in bytes of streams pool
streams_pool_size	0	size in bytes of the streams pool
_io_shared_pool_size	4194304	Size of I/O buffer pool from SGA
_backup_io_pool_size	1048576	memory to reserve from the large pool
global_context_pool_size		Global Application Context Pool Size in Bytes
olap_page_pool_size	0	size of the olap page pool in bytes

13 rows selected

这些由两个下划线开头的参数决定了当前 SGA 的分配，也是动态内存管理调整的参数。这些参数的更改会被记录到 spfile 文件当中，在下一次数数据库启动时仍然有效。

转储一下 spfile 文件：

```
[oracle@danaly dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 10.2.0.1.0 - Production on Mon Apr 17 13:46:14 2006

Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options

SQL> show parameter spfile

NAME                                TYPE                                VALUE
-----
spfile                              string                             +ORADG/danaly/spfiledanaly.ora
SQL> create pfile from spfile;

File created.

SQL> exit

Disconnected from Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options
```

检查其参数设置，可以清晰地看到这些参数及其参数值：

```
[oracle@danaly dbs]$ more initdanaly.ora
danaly.__db_cache_size=830472192
danaly.__java_pool_size=4194304
danaly.__large_pool_size=4194304
danaly.__shared_pool_size=96468992
danaly.__streams_pool_size=0
*.audit_file_dest='/opt/oracle/admin/danaly/adump'
*.background_dump_dest='/opt/oracle/admin/danaly/bdump'
*.compatible='10.2.0.1.0'
*.control_files='+ORADG/danaly/controlfile/current.256.600173845','+ORADG/danaly/controlfile/current.257.600173845'
```

.....

```
*.user_dump_dest='/opt/oracle/admin/danally/udump'
```

这是 Oracle 10g 在参数上的一些变化，大家可以自己通过实验研究一下，本文不再过多阐述。

通过 Oracle 10g 新增加的动态视图 v\$sga_dynamic_components，可以看到各动态组件调整的时间和调整类型等信息：

```
SQL> select COMPONENT,CURRENT_SIZE,MIN_SIZE,LAST_OPER_TYPE,
LAST_OPER_MODE,to_char(LAST_OPER_TIME,'yyyy-mm-dd hh24:mi:ss') LOT
2 from v$sga_dynamic_components;
```

COMPONENT	CURRENT_SIZE	MIN_SIZE	LAST_O	LAST_OPER_MODE	LOT
shared pool	96468992	92274688	GROW	IMMEDIATE	2006-04-17 10:04:08
large pool	4194304	4194304	STATIC		
java pool	4194304	4194304	STATIC		
streams pool	0	0	STATIC		
DEFAULT buffer cache	830472192	830472192	SHRINK	IMMEDIATE	2006-04-17 10:04:08
KEEP buffer cache	0	0	STATIC		
RECYCLE buffer cache	0	0	STATIC		
DEFAULT 2K buffer cache	0	0	STATIC		
DEFAULT 4K buffer cache	0	0	STATIC		
DEFAULT 8K buffer cache	0	0	STATIC		
DEFAULT 16K buffer cache	0	0	STATIC		
DEFAULT 32K buffer cache	0	0	STATIC		
ASM Buffer Cache	0	0	STATIC		

13 rows selected

4.2 PGA 管理

4.2.1 什么是 PGA

PGA-指的是程序全局区（Program Global Area），是服务器进程（Server Process）使用的一块包含数据和控制信息的内存区域，PGA 是非共享的内存，在服务器进程启动或创建时分配（在系统运行时，排序、连接等操作也可能需要进一步的 PGA 分配），并为 Server Process 排他访问。PGA 的内容依专用模式和共享服务器模式而不同，但是通常来说，PGA 中包含私有 SQL 区（存放绑定信息、运行时内存结构等）和 session 信息等内容。

所有服务器进程分配的 PGA 总和通常被称为 PGA 合计 (Aggregated PGA)。在 Oracle 9i 以前的版本中, PGA 由一系列的内存区域组成, 这些区域包括主要由*_area_size 参数控制。

在 Oracle 8i 的环境中, 这些参数主要有:

- Sort_Area_Size;
- Hash_Area_Size;
- Bitmap_Merge_Size;
- Create_Bitmap_Area_Size。

可以从数据库中得到这些参数设置的当前值:

```
$ sqlplus '/ as sysdba'
SQL*Plus: Release 8.1.7.0.0 - Production on Thu Apr 6 14:21:18 2006
(c) Copyright 2000 Oracle Corporation. All rights reserved.
```

Connected to:

Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production

With the Partitioning option

JServer Release 8.1.7.4.0 - 64bit Production

```
SQL> show parameter area_size
```

NAME	TYPE	VALUE
bitmap_merge_area_size	integer	1048576
create_bitmap_area_size	integer	8388608
hash_area_size	integer	131072
sort_area_size	integer	65536

可以通过手工修改 sort_area_size、hash_area_size 等参数值来控制 PGA 的使用。

在 Oracle 9i 之前, PGA 的计算和控制都是比较复杂的事情, 从 Oracle 9i 开始, Oracle 提供了一种 PGA 内存管理的新方法: 自动化 SQL 执行内存管理 (Automated SQL Execution Memory Management)。使用这个新特性, Oracle 可以自动调整 SQL 内存区, 而不用关闭数据库, 这一改进大大简化了 DBA 的工作, 同时也提高了 Oracle 数据库的性能。

为实现自动的 PGA 管理, Oracle 引入了几个新的初始化参数。

- PGA_AGGREGATE_TARGET: 此参数用来指定所有 session 总计可以使用最大 PGA 内存。这个参数可以被动态的更改, 取值范围从 10M~(4096G-1) bytes。

- WORKAREA_SIZE_POLICY: 此参数用于开关 PGA 内存自动管理功能, 该参数有两个选项 AUTO 和 MANUAL, 当设置为 AUTO 时, 数据库使用 Oracle 9i 提供的自动 PGA 管理功能, 当设置为 MANUAL 时, 则仍然使用 Oracle 9i 前手工管理的方式。缺省情况下, Oracle 9i 中 WORKAREA_SIZE_POLICY 被设置为 AUTO。

需要注意的是, 在 Oracle 9i 中, PGA_AGGREGATE_TARGET 参数仅对专用服务器模式

下 (Dedicated Server) 的专属连接有效, 但是对共享服务器 (Shared Server) 连接无效; 从 Oracle 10g 开始, PGA_AGGREGATE_TARGET 对专用服务器连接和共享服务器连接同时生效。

PGA_AGGREGATE_TARGET 参数同时限制全局 PGA 分配和私有工作区内存分配:

(1) 对于串行操作, 单个 SQL 操作能够使用的 PGA 内存按照以下原则分配:

MIN (5% PGA_AGGREGATE_TARGET, 100MB)

(2) 对于并行操作:

30% PGA_AGGREGATE_TARGET / DOP (DOP=Degree Of Parallelism 并行度)

要理解 PGA 的自动调整, 还需要区分可调整内存 (TUNABLE MEMORY SIZE) 与不可调整内存 (UNTUNABLE MEMORY SIZE)。可调整内存是由 SQL 工作区使用的, 其余部分是不可调整内存。

启用了自动 PGA 调整之后, Oracle 仍然需要遵循以下原则:

UNTUNABLE MEMORY SIZE + TUNABLE MEMORY SIZE <= PGA_AGGREGATE_TARGET

数据库系统只能控制可调整部分的内存分配, 如果可调整的部分过小, 则 Oracle 永远也不会强制启用这个等式。

另外, PGA_AGGREGATE_TARGET 参数在 CBO 优化器模式下, 对于 SQL 的执行计划会产生影响。Oracle 在评估执行计划时会根据 PGA_AGGREGATE_TARGET 参数评估在 Sort, HASH-JOIN 或 Bitmap 操作时能够使用的最大或最小内存, 从而选择最优的执行计划。

对于 PGA_AGGREGATE_TARGET 参数的设置, Oracle 提供这样一个建议方案。

- 对于 OLTP 系统

PGA_AGGREGATE_TARGET = (<Total Physical Memory > * 80%) * 20%

- 对于 DSS 系统

PGA_AGGREGATE_TARGET = (<Total Physical Memory > * 80%) * 50%

也就是说, 对于一个单纯的数据库服务器, 通常需要保留 20% 的物理内存给操作系统使用, 剩余 80% 可以分配给 Oracle 使用。Oracle 使用的内存分为两部分 SGA 和 PGA, 那么 PGA 可以占用 Oracle 消耗总内存的 20% (OLTP 系统) ~ 50% (DSS 系统)

—— 注 意 ——

在某些 OS 上单个进程使用的真实内存可能远大于在 Oracle 中看到的 PGA 大小, 如 AIX。

这只是一个建议设置, 更进一步的应该根据数据库的具体性能指标来调整和优化 PGA 的使用。

伴随这个新特性的引入, V\$PROCESS 视图增加了相应字段用来记录进程的 PGA 耗用:

```
$ ps -ef|grep ora|head -1
  oracle 26619      1   0 16:58:36 ?          0:00 oraclehsboss (LOCAL=NO)

$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Thu Apr 6 17:38:04 2006

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
```

Connected to:

Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production

With the Partitioning option

JServer Release 9.2.0.4.0 - Production

```
SQL>select pid,spid,username,pga_used_mem,pga_alloc_mem,pga_freeable_mem,pga_max_mem
2  from v$process where spid=26619;
```

PID	SPID	USERNAME	PGA_USED_MEM	PGA_ALLOC_MEM	PGA_FREEABLE_MEM	PGA_MAX_MEM
15	26619	oracle	168317	491717	0	491717

SQL 在工作区中以 3 种方式执行。

- **Optimal**（优化方式）：指所有处理可以在内存中完成。
- **Onepass**：大部分操作可以在内存中完成，但是需要使用到磁盘操作。
- **Multipass**：大量操作需要产生磁盘交互，性能极差。

通常对于 PGA 的优化目标，就是使得 **Optimal** 的执行尽量高，也就是尽量在内存中完成所有排序等操作；同时使 **Multipass** 操作尽量低，也就是要使磁盘交互尽量低。

也就是期望实现如下目标：

- workarea execution - optimal >= 90%；
- workarea execution - multipass = 0%。

以下是一个生产系统的查询结果：

```
SQL> SELECT NAME, VALUE,
2      100
3      * ( VALUE
4      / DECODE ((SELECT SUM (VALUE) FROM v$sysstat
5                  WHERE NAME LIKE 'workarea executions%'),
6                  0, NULL,
7                  (SELECT SUM (VALUE) FROM v$sysstat
8                  WHERE NAME LIKE 'workarea executions%')
9      )
10     ) pct
11 FROM v$sysstat
12 WHERE NAME LIKE 'workarea executions%'
13 /
```

NAME	VALUE	PCT
------	-------	-----

workarea executions - optimal	22478	97.9433551
workarea executions - onepass	397	1.72984749
workarea executions - multipass	75	.326797386

伴随自动 PGA 调整新特性的引入，Oracle 随之引入了一系列新的视图，V\$PGASTAT 就是其中的一个。

在 V\$PGASTAT 中有这样一个条目 global memory bound，该条目记录数据库允许的最高 PGA 内存使用量，可以从不同的 PGA 参数设置来观察一下 Oracle 运行的 PGA 上限。

```
SQL> alter system set pga_aggregate_target=&Nm;
Enter value for nm: 10m
old 1: alter system set pga_aggregate_target=&Nm
new 1: alter system set pga_aggregate_target=10m

System altered.

Elapsed: 00:00:00.05
SQL> SET autotrace traceonly
SQL> SELECT DISTINCT * FROM t WHERE ROWNUM < 500000;

20000 rows selected.

Elapsed: 00:03:04.12

.....
SQL> SET autotrace off
SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
2      last_execution, last_tempseg_size
3      FROM v$sql l, v$sql_workarea a
4      WHERE l.hash_value = a.hash_value
5      AND sql_text = 'SELECT DISTINCT * FROM t WHERE ROWNUM < 500000';

SQL_TEXT                                OPERATION_TYPE      POLIC
-----
LAST_MEMORY_USED/1024/1024 LAST_EXE LAST_TEMPSEG_SIZE
-----
SELECT DISTINCT * FROM t WHERE ROWNUM < 500000      GROUP BY (SORT)      AUTO
.548828125 206 PASSES                                62914560
```

Elapsed: 00:00:00.02

SQL>

SQL> SELECT NAME, VALUE / 1024 / 1024 MB

2 FROM v\$pgastat

3 WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');

NAME	MB

aggregate PGA target parameter	10
global memory bound	.5

SQL> alter system set pga_aggregate_target=&Nm;

Enter value for nm: 30M

old 1: alter system set pga_aggregate_target=&Nm

new 1: alter system set pga_aggregate_target=30M

System altered.

Elapsed: 00:00:00.05

SQL> SET autotrace traceonly

SQL> SELECT DISTINCT * FROM t WHERE ROWNUM < 500000;

20000 rows selected.

Elapsed: 00:00:53.30

.....

SQL> SET autotrace off

SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,

2 last_execution, last_tempseg_size

3 FROM v\$sql l, v\$sql_workarea a

4 WHERE l.hash_value = a.hash_value

5 AND sql_text = 'SELECT DISTINCT * FROM t WHERE ROWNUM < 500000';

SQL_TEXT	OPERATION_TYPE	POLIC
LAST_MEMORY_USED/1024/1024		

LAST_EXECUTION	LAST_TEMPSEG_SIZE	

```
-----
SELECT DISTINCT * FROM t WHERE ROWNUM < 500000      GROUP BY (SORT)      AUTO
L.48046875
```

```
6 PASSES                                57671680
```

```
Elapsed: 00:00:00.02
```

```
SQL>
```

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2      FROM v$pgastat
```

```
3      WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	30
global memory bound	1.5

```
Elapsed: 00:00:00.00
```

可以注意到，PGA 的 global memory bound 会一直处在 5% 的 PGA_AGGREGATE_TARGET 参数设置，直到 5% PGA_AGGREGATE_TARGET 超过 100MB，然后 global memory bound 被限制为 100MB，也就是满足前文提到的，对于串行操作，单个 SQL 操作能够使用的 PGA 内存按照以下原则分配：

```
MIN (5% PGA_AGGREGATE_TARGET, 100MB)
```

注意修改 PGA_AGGREGATE_TARGET 参数可以使用如下命令：

```
alter system set pga_aggregate_target=4096M ;
```

修改参数后，通常需要执行操作才能看到视图信息的变化：

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2      FROM v$pgastat
```

```
3      WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	10
global memory bound	.5

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2      FROM v$pgastat
```

```
3 WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	20
global memory bound	1

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2 FROM v$pgastat
```

```
3 WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	40
global memory bound	2

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2 FROM v$pgastat
```

```
3 WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	1024
global memory bound	51.1992188

```
SQL> SELECT NAME, VALUE / 1024 / 1024 MB
```

```
2 FROM v$pgastat
```

```
3 WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');
```

NAME	MB
aggregate PGA target parameter	4096
global memory bound	100

实际上这个 100MB 的上限是受到了另外一个隐含参数的控制, 该参数为 `_pga_max_size`, 该参数的缺省值为 200MB, 单进程串行操作 PGA 的上限不能超过该参数的 1/2。

```
SQL> SELECT x.kspinm NAME, y.kspstvl VALUE, x.kspdesc describ
```

```
2 FROM SYS.x$kspci x, SYS.x$kspcv y
```

```

3  WHERE x.inst_id = USERENV ('Instance')
4  AND y.inst_id = USERENV ('Instance')
5  AND x.indx = y.indx
6  AND x.ksppinm LIKE '%&par%'
7  /
Enter value for par: pga_max
old 6:  AND x.ksppinm LIKE '%&par%'
new 6:  AND x.ksppinm LIKE '%pga_max%'

NAME                                VALUE                                DESCRIB
-----
_pga_max_size                       209715200                          Maximum size of the PGA memory for one process

```

如果修改该参数，global memory bound 将可以突破 100MB 的上限：

```

SQL> alter system set "_pga_max_size"=400M;

System altered.

.....

SQL> SELECT NAME, VALUE / 1024 / 1024 MB
2  FROM v$pgastat
3  WHERE NAME IN ('aggregate PGA target parameter', 'global memory bound');

NAME                                MB
-----
aggregate PGA target parameter      4096
global memory bound                 200

```

对于 PGA 的控制，还有一系列的内部参数，列举如下，仅供参考：

```

SQL> 1
1  SELECT x.ksppinm NAME, y.ksppstvl VALUE, x.ksppdesc describ
2  FROM SYS.x$ksppi x, SYS.x$kspev y
3  WHERE x.inst_id = USERENV ('Instance')
4  AND y.inst_id = USERENV ('Instance')
5  AND x.indx = y.indx
6*  AND x.ksppinm LIKE '%&par%'

SQL> /

Enter value for par: smm
old 6:  AND x.ksppinm LIKE '%&par%'
new 6:  AND x.ksppinm LIKE '%smm%'

```

NAME	VALUE	DESCRIB

_smm_auto_min_io_size	56	Minimum IO size (in KB) used by sort/hash-join in auto mode
_smm_auto_max_io_size	248	Maximum IO size (in KB) used by sort/hash-join in auto mode
_smm_auto_cost_enabled	TRUE	if TRUE, use the AUTO size policy cost functions
_smm_control	0	provides controls on the memory manager
_smm_trace	0	Turn on/off tracing for SQL memory manager
_smm_min_size	128	minimum work area size in auto mode
_smm_max_size	2560	maximum work area size in auto mode (serial)
_smm_px_max_size	15360	maximum work area size in auto mode (global)
_smm_bound	0	overwrites memory manager automatically computed bound
_smm_advice_log_size	0	overwrites default size of the PGA advice workarea history log
_smm_advice_enabled	TRUE	if TRUE, enable v\$pga_advice

11 rows selected.

4.2.2 PGA 的调整建议

伴随自动 PGA 调整功能的引入，Oracle 同时引入相应的动态性能视图用于优化建议，PGA 的优化建议通过 v\$pga_target_advice 和 v\$pga_target_advice_histogram 提供。

v\$pga_target_advice 视图通过对不同 PGA 设置进行评估，给出在不同设置下的 PGA 命中率 and OverAlloc 等信息。

```
SQL> select PGA_TARGET_FOR_ESTIMATE/1024/1024 PGAMB,PGA_TARGET_FACTOR,ESTD_PGA_
CACHE_HIT_PERCENTAGE,ESTD_OVERALLOC_COUNT
```

```
2 from v$pga_target_advice;
```

```
PGAMB PGA_TARGET_FACTOR ESTD_PGA_CACHE_HIT_PERCENTAGE ESTD_OVERALLOC_
COUNT
```

37.5	0.125	35	391
75	0.25	43	58
150	0.5	61	0
225	0.75	70	0
300	1	70	0
360	1.2	71	0
420	1.4	72	0

480	1.6	72	0
540	1.8	72	0
600	2	72	0
900	3	72	0
1200	4	72	0
1800	6	72	0
2400	8	72	0

14 rows selected

可以看到，在以上输出中，当 PGA 设置为 150MB 时，即可消除 PGA 过载；在 PGA 设置为 420MB 时，可以达到最高命中率 72%；当前 PGA 设置为 300MB。根据这些信息，可以调整 PGA 为 420MB，从而提高性能。

从 Oracle 9i 或者 Oracle 10g 提供的 OEM 界面上，都可以直观地看到这些建议信息，图 4-7 是当前的 PGA 设置页。



图 4-7 当前的 PGA 设置页

单击“建议”按钮，可以看到 Oracle 对于 PGA 设置的建议，如图 4-8 所示。
本例修改后数据库的 PGA 分配为 420MB：

```
SQL> show parameter pga
```

NAME	TYPE	VALUE
pga_aggregate_target	big integer	420M

该曲线是根据 `v$pga_target_advice` 收集的信息绘制出来的，可以直接在图中拖动调整 PGA 的设置，确定后在 PGA 页应用该修改即可达到优化数据库的目的。

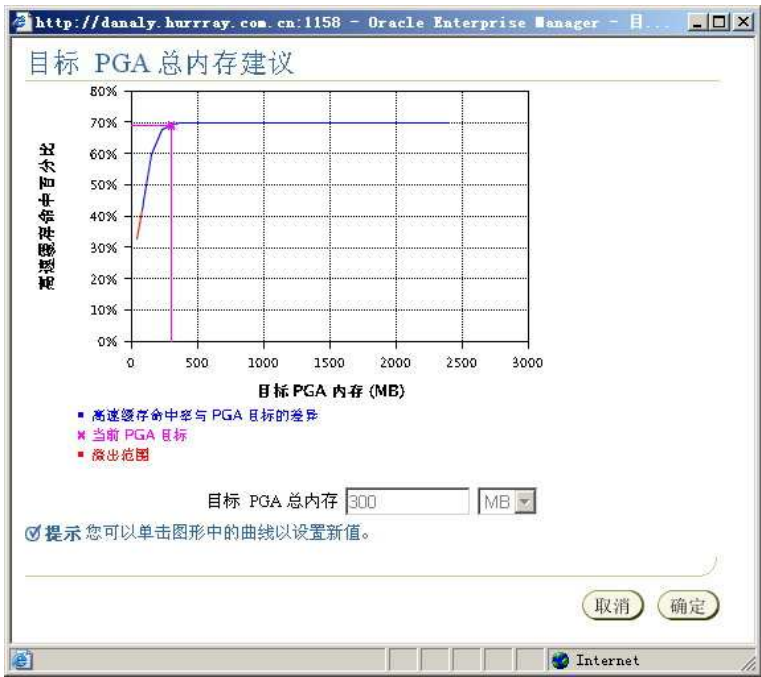


图 4-8 目标 PGA 总内存建议

调整 PGA 后，建议信息会刷新，统计信息需要重新评估：

```
SQL> select PGA_TARGET_FOR_ESTIMATE/1024/1024 PGAMB,PGA_TARGET_FACTOR,ESTD_PGA_
CACHE_HIT_PERCENTAGE,ESTD_OVERALLOC_COUNT
2   from v$pga_target_advice
3   /
```

PGAMB	PGA_TARGET_FACTOR	ESTD_PGA_CACHE_HIT_PERCENTAGE	ESTD_OVERALLOC_COUNT
52.5	0.125	100	1
105	0.25	100	0
210	0.5	100	0
315	0.75	100	0
420	1	100	0
504	1.2	100	0
588	1.4	100	0
672	1.6	100	0
756	1.8	100	0
840	2	100	0
1260	3	100	0

1680	4	100	0
2520	6	100	0
3360	8	100	0
14 rows selected			

v\$pga_target_advice_histogram 视图可以通过对不同工作区大小的采样评估提供统计信息供分析使用，其中几个重要字段有：

- LOW_OPTIMAL_SIZE-Histogram 评估区间内 Optimal 下限（bytes）；
- HIGH_OPTIMAL_SIZE-Histogram 评估区间内 Optimal 上限（bytes）；
- ESTD_OPTIMAL_EXECUTIONS-Histogram 评估区间内估计 Optimal 执行次数；
- ESTD_ONEPASS_EXECUTIONS-Histogram 评估区间内估计 Onepass 执行次数；
- ESTD_MULTIPASSES_EXECUTIONS-Histogram 评估区间内估计 Multipass 执行次数；
- ESTD_TOTAL_EXECUTIONS-Histogram 评估区间内估计执行总次数。

可以从 OEM 的直观柱状图中得到不同 PGA 设置下的评估数据，如图 4-9 所示。

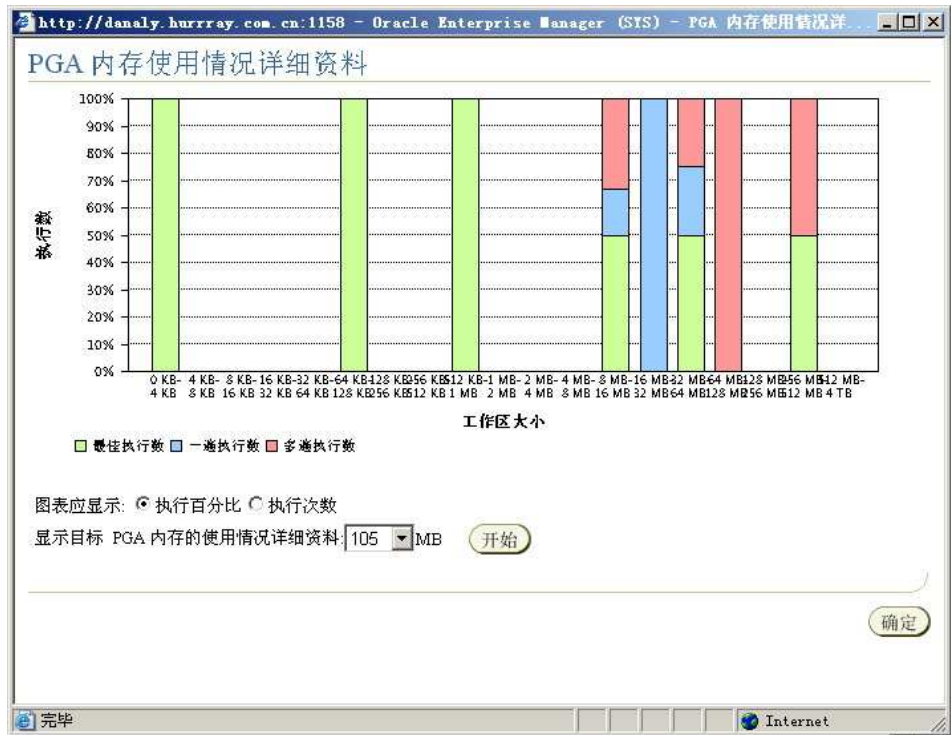


图 4-9 PGA 内存使用情况

同样地，这个柱状图的信息来自动态性能视图 v\$pga_target_advice_histogram，可以从数据库中查询得到这些数据，以上柱状图正是依据这些数据绘制得出的：

```
SQL> SELECT pga_target_factor factor, low_optimal_size / 1024 low,
2         ROUND (high_optimal_size / 1024) high,
3         estd_optimal_executions estd_opt, estd_onepass_executions estd_op,
```

```

4      estd_multipasses_executions estd_mp, estd_total_executions estd_exec
5  FROM v$pga_target_advice_histogram
6  WHERE pga_target_factor = 0.25 AND estd_total_executions > 0
7  /

```

FACTOR	LOW	HIGH	ESTD_OPT	ESTD_OP	ESTD_MP	ESTD_EXEC
0.25	262144	524288	1	0	1	2
0.25	65536	131072	0	0	2	2
0.25	32768	65536	4	2	2	8
0.25	16384	32768	0	2	0	2
0.25	8192	16384	3	1	2	6
0.25	512	1024	2038	0	0	2038
0.25	64	128	3	0	0	3
0.25	2	4	4540	0	0	4540

8 rows selected

PGA 的新特性使得 PGA 的管理得以极大简化。

4.3 Oracle 的内存分配和使用

Oracle 数据库在系统占用的内存分内两个部分：SGA 和 PGA。如何设置和规划 Oracle 的内存分配一直以来是一个广为争论的话题。在 Oracle 9i 之前，PGA 的内存使用估算一直比较复杂；从 Oracle 9i 开始，Oracle 的自动 PGA 管理新特性使得 Oracle 的内存规划得以简化。

根据 Oracle 的建议，Oracle 最多可以使用 80% 的物理内存，其余 20% 保留给操作系统使用，在这 80% 的内存中，对于 OLTP 系统，Oracle 建议分配 20% 给 PGA 使用；对于 DSS 系统，可以分配 50% 给 PGA 使用，再引述一下前文的等式：

- 对于 OLTP 系统

$$\text{PGA_AGGREGATE_TARGET} = (\text{<Total Physical Memory>} * 80\%) * 20\%$$

- 对于 DSS 系统

$$\text{PGA_AGGREGATE_TARGET} = (\text{<Total Physical Memory>} * 80\%) * 50\%$$

进一步归纳一下就是 $\text{SGA} + \text{PGA} \leq \text{<Total Physical Memory>} * 80\%$ ，也就是：

$$\text{SGA_MAX_SIZE} + \text{PGA_AGGREGATE_TARGET} \leq \text{<Total Physical Memory>} * 80\%$$

这是一个可以参考的数值，在为 Oracle 规划内存使用时，我们必须清楚，如果 Oracle 耗用的内存过高，甚至超过了系统的物理内存，那么系统的性能就会受到严重的影响，当系统执行任务时，如果没有足够的内存，那么系统就会进行分页或交换，以完成当前活动事务。

当系统执行分页时，会将当前没有使用的信息从内存转移到硬盘上，这样就可以为当前

需要内存的程序分配内存。如果频繁发生分页，系统性能就会严重降低，从而导致很多程序的执行时间变长。

当系统执行交换时，会将某些进程所分配的不活跃内存页（根据 LRU 算法）从内存转移到硬盘上，这样另一个活动进程就可以得到所需要的内存。交换基于系统循环时间。如果交换太过频繁，系统甚至会出现当机。

接下来通过实际生产中的案例，来看一下内存分配不当会带来什么问题。

4.3.1 诊断案例一：SGA 与 Swap

案例描述：用户报告，服务器启动一段时间以后，无法建立数据库连接。重新启动几分钟后，再次无法连接。

操作系统：SunOS 5.8，系统无法正常使用。

1. 登录数据库，检查系统进程

登录系统，检查系统进程，发现后台进程正常，有一定量的用户连接：

```
wapplatform:/>su - oracle
Sun Microsystems Inc. SunOS 5.8 Generic Patch October 2001
You have new mail.
/export/home1/oracle>ls
.....
/export/home1/oracle/admin>ps -ef|grep ora
oracle 25269 25258 0 13:58:36 pts/3 0:00 grep ora
oracle 25267 1 1 13:58:34 ? 0:00 oracleHSWAPDB (LOCAL=NO)
oracle 25193 1 0 13:57:03 ? 0:01 oracleHSWAPDB (LOCAL=NO)
oracle 25209 1 0 13:57:09 ? 0:00 oracleHSWAPDB (LOCAL=NO)
oracle 25244 1 1 13:58:23 ? 0:00 oracleHSWAPDB (LOCAL=NO)
oracle 25218 1 0 13:57:23 ? 0:00 oracleHSWAPDB (LOCAL=NO)
.....
oracle 25149 1 0 13:56:41 ? 0:01 ora_lgwr_HSWAPDB
oracle 25153 1 0 13:56:42 ? 0:01 ora_smon_HSWAPDB
oracle 25155 1 0 13:56:42 ? 0:00 ora_reco_HSWAPDB
oracle 25151 1 0 13:56:41 ? 0:00 ora_ckpt_HSWAPDB
oracle 25145 1 0 13:56:41 ? 0:00 ora_dbw0_HSWAPDB
oracle 25143 1 0 13:56:41 ? 0:00 ora_pmon_HSWAPDB
.....
```

2. 检查警报日志文件

发现如下大量提示信息：

```
Tue Mar 23 13:40:45 2004
```

```
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 11, op = fork, loc = skgpspawn5
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
```

```
Tue Mar 23 13:42:02 2004
```

```
skgpspawn failed:category = 27142, depinfo = 12, op = fork, loc = skgpspawn3
```

该提示说明系统无法 fork 新的数据库进程，数据库无法 spawn a new session。而且这里 ‘skgpspawn failed:category = 27142’ 实际上应该是 Oracle 的错误号，可以通过 Oracle 的手册查询到这个错误的具体内容，在 UNIX/Linux 上，可以通过 oerr 工具获得相关信息：

```
$ oerr ora 27142
27142, 0000, "could not create new process"
// *Cause: OS system call
// *Action: check errno and if possible increase the number of processes
```

3. 尝试连接数据库

当再次尝试连接数据库时，收到如下错误信息，无法连接数据库：

```
$ sqlplus "/ as sysdba"
SQL*Plus: Release 9.2.0.3.0 - Production on 星期二 3 月 23 14:14:06 2004
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
ERROR:
ORA-12540: TNS: 超出内部限制
请输入用户名:
ERROR:
ORA-12540: TNS: 超出内部限制
请输入用户名:
ERROR:
ORA-12540: TNS: 超出内部限制
SP2-0157: 在 3 次尝试之后无法 CONNECT 到 ORACLE, 退出 SQL*Plus
```

内部限制超过，通常说明某些系统资源不足。

4. 检查系统日志

检查系统日志信息，发现大量失败的 su 操作，有 Swap 区不足的报告：

```
wapplatform:/>dmesg
2004 年 03 月 23 日 星期二 14 时 00 分 32 秒 CST
Mar 23 11:23:40 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 3804 (su)
Mar 23 11:23:40 wapplatform last message repeated 8 times
Mar 23 11:23:56 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 3806 (ps)
Mar 23 11:23:56 wapplatform last message repeated 12 times
Mar 23 11:24:01 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 3808 (w)
Mar 23 11:24:01 wapplatform last message repeated 8 times
Mar 23 13:40:56 wapplatform su: [ID 810491 auth.crit] 'su root' failed for root on /dev/pts/2
Mar 23 13:46:26 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 24888
(sqlplus)
Mar 23 13:49:18 wapplatform su: [ID 810491 auth.crit] 'su oracle' failed for root on /dev/pts/6
Mar 23 13:54:03 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 25035 (su)
Mar 23 13:54:08 wapplatform genunix: [ID 470503 kern.warning] WARNING: Sorry, no swap space to grow
stack for pid 25036 (su)
```

现在基本可以判断是交换区的问题，当然和 Oracle SGA 设置有关。

5. 检查系统内存及交换区使用

通过 TOP 工具检查系统内存及 Swap 使用情况：

```
$ top

last pid: 25456; load averages: 0.67, 0.70, 0.69          14:10:03
93 processes: 91 sleeping, 2 on cpu
CPU states: 72.7% idle, 14.9% user, 2.7% kernel, 9.7% iowait, 0.0% swap
Memory: 1024M real, 34M free, 752M swap in use, 10M swap free

PID USERNAME THR PRI NICE SIZE RES STATE TIME CPU COMMAND
25199  oracle 1 40 0 674M 631M cpu/2 8:03 16.32% oracle
25209  oracle 1 30 0 675M 630M sleep 0:03 0.13% oracle
25159  oracle 1 48 0 674M 628M sleep 0:03 0.06% oracle
25384  oracle 1 58 0 2632K 1736K cpu/0 0:01 0.05% top
25145  oracle 143 58 0 682M 630M sleep 0:01 0.03% oracle
```

发现物理内存仅为 1GB，Free 部分为 34MB，交换区使用了 752MB，仅 Swap Free 部分仅余 10MB。由此可知，系统内存严重不足，Swap 区不足。

6. 检查数据库的 SGA 设置

发现 SGA 设置为 622299344 bytes，接近 600MB，这个 SGA 设置过高：

```
/export/home1/oracle>sqlplus "/" as sysdba"
SQL*Plus: Release 9.2.0.3.0 - Production on 星期二 3 月 23 14:02:30 2004
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
连接到:
Oracle9i Enterprise Edition Release 9.2.0.3.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production
SQL> show sga
Total System Global Area 622299344 bytes
Fixed Size                731344 bytes
Variable Size             268435456 bytes
Database Buffers          352321536 bytes
Redo Buffers              811008 bytes
SQL>
```

对于 RAM 小于 1GB 的系统，Dedicated 模式下，通常建议 Oracle 的 SGA 一般不应超过 1/2 物理内存，SGA 之外还应考虑到 PGA 及操作系统的内存分配。

7. 调整内容

第一步调整，减小 SGA，为系统保留足够的内存。

第二步调整，为系统增加 Swap 区。

```
wapplatform:/var/swap>cd /export/home1
wapplatform:/export/home1>mkdir swap
wapplatform:/export/home1>cd swap
wapplatform:/export/home1/swap>mkfile -v 1g swapfile1
swapfile1 1073741824 bytes
wapplatform:/export/home1/swap>swap -a /export/home1/swap/swapfile1
wapplatform:/export/home1/swap>swap -s
总数: 分配了 623160k 字节 + 保留 162704k = 已使用 785864k, 1010936k 可用
```

至此系统恢复正常，问题解决。

8. 问题总结

Oracle 数据库问题的解决从来就离不开操作系统，很多时候必须通过操作系统一级的手

袭来诊断并解决问题。关于操作系统，一般 Swap 区的推荐值为 $2 \times \text{RAM}$ 。

如果物理内存（RAM）很大，不一定非要把 Swap 设置为 $2 \times \text{Swap}$ ，通常可以设置 $\text{Swap} = \text{Ram}$ 或者小于物理内存（如内存超过 32GB 则完全可以设置 Swap 为 16GB）。

如果物理内存（RAM）过小，在系统繁忙期间，产生大量交换无法换到磁盘，就会出现问題，如本案例就是这样。

另外，如果系统物理内存较小，通常设置 $\text{SGA} < 1/2 \text{ RAM}$ ，要考虑为 Server Process 的 PGA 消耗及 OS 保留足够的内存空间。

4.3.2 诊断案例二：SGA 设置过高导致的系统故障

案例描述：这是一个大型生产系统，问题出现时系统累计大量用户进程，用户请求得不到及时响应，新的进程不断尝试建立连接，连接数很快被用完。最后系统处于挂起状态，无法进行服务响应。

数据库版本：9.2.0.3

操作系统：Solaris 8

1. 登录数据库，检查警告日志文件

接到问题报告之后，马上登录数据库服务器，检查 alert 文件。日志中记录如下错误信息，说明系统异步 IO 出现问题：

```
WARNING: aiowait timed out 2 times
Tue Aug 26 15:33:32 2003
WARNING: aiowait timed out 2 times
Tue Aug 26 15:33:34 2003
WARNING: aiowait timed out 2 times
Tue Aug 26 15:33:36 2003
WARNING: aiowait timed out 2 times
Tue Aug 26 15:33:38 2003
WARNING: aiowait timed out 2 times
Tue Aug 26 15:33:43 2003
WARNING: aiowait timed out 1 times
Tue Aug 26 15:33:46 2003
WARNING: aiowait timed out 1 times
Tue Aug 26 15:33:49 2003
WARNING: aiowait timed out 1 times
Tue Aug 26 15:33:51 2003
WARNING: aiowait timed out 1 times
Tue Aug 26 15:33:52 2003
WARNING: aiowait timed out 1 times
```

```
Tue Aug 26 15:33:53 2003
WARNING: aiowait timed out 1 times
.....
```

后来在其他平台上也发现类似的错误提示：

```
Tue Nov 11 14:08:24 2003
WARNING: aiowait timed out 1 times
Tue Nov 11 14:18:24 2003
WARNING: aiowait timed out 2 times
Tue Nov 11 14:28:24 2003
WARNING: aiowait timed out 3 times
Tue Nov 11 14:38:24 2003
WARNING: aiowait timed out 4 times
Tue Nov 11 14:38:24 2003
WARNING: aiowait timed out 5 times
Tue Nov 11 14:48:24 2003
WARNING: aiowait timed out 5 times
Tue Nov 11 14:58:24 2003
WARNING: aiowait timed out 6 times
Tue Nov 11 15:08:24 2003
WARNING: aiowait timed out 7 times
Tue Nov 11 15:08:24 2003
WARNING: aiowait timed out 7 times
```

注意到每次 **WARNING** 的间隔时间为 10 分钟。

注 意

由于警报日志文件（`alert_<sid>.log`）中会记录数据库出现故障时的错误信息等，所以我们处理数据库问题时，通常应该首先检查该文件，看是否可以从中发现线索。

我们知道在 **SUN** 的某些版本上异步 **IO** 存在问题，而异步 **IO** 缺省是打开的：

```
SQL> show parameter disk_a
NAME                TYPE                VALUE
-----
disk_asynch_io      Boolean              TRUE
```

针对此问题，暂时停用了数据库的异步 **IO** 写入。

2. 检查共享内存设置

`alert` 文件中还记录了以下警告信息：

```
Tue Aug 26 21:37:40 2003
```

```
WARNING: EINVAL creating segment of size 0x0000000190400000
fix shm parameters in /etc/system or equivalent
```

该信息说明系统内核参数设置不合理或者和 SGA 不匹配。检查 system 配置文件：

```
$ cat /etc/system
.....
set shmsys:shminfo_shmmax=4096000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=200
set shmsys:shminfo_shmseg=200
set semsys:seminfo_semmap=1024
set semsys:seminfo_semmni=2048
set semsys:seminfo_semmns=2048
set semsys:seminfo_semmnu=2048
set semsys:seminfo_semume=200
set semsys:seminfo_semmul=2048
```

发现最大共享内存段设置为 4GB。

3. 检查 SGA 设置

查看内核参数之后，检查 SGA 设置：

```
SQL*Plus: Release 9.2.0.3.0 - Production on 星期二 8 月 26 21:46:35 2003
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:

Oracle9i Enterprise Edition Release 9.2.0.3.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production

SQL> show sga

Total System Global Area   6695660272 bytes
Fixed Size                  740080 bytes
Variable Size              2399141888 bytes
Database Buffers           4294967296 bytes
Redo Buffers                811008 bytes
```

发现 SGA 设置接近 7GB（超过了 4GB，Oracle 将分配多个共享内存段），这也就是步骤 2 中警告提示出现的原因。

4. 交换区问题

用 Top 工具检查系统运行状况：

```
# /usr/local/bin/top

last pid: 16899; load averages: 0.82, 0.81, 0.83 21:49:05
1230 processes:1228 sleeping, 1 running, 1 on cpu
CPU states: 50.1% idle, 7.4% user, 8.6% kernel, 33.9% iowait, 0.0% swap
Memory: 8192M real, 118M free, 12G swap in use, 11G swap free

PID USERNAME THR PRI NICE SIZE RES STATE TIME CPU COMMAND
15751 oracle 11 44 0 6456M 6408M sleep 0:02 0.49% oracle
15725 oracle 11 58 0 6458M 6410M sleep 0:02 0.46% oracle
251 root 12 48 0 7096K 1944K sleep 126:00 0.45% picld
16540 oracle 11 58 0 6458M 6411M sleep 0:01 0.45% oracle
16766 root 1 43 0 3744K 2248K cpu/1 0:01 0.41% top
16408 oracle 11 58 0 6457M 6410M sleep 0:01 0.34% oracle
15989 oracle 11 58 0 6458M 6409M sleep 0:01 0.34% oracle
15919 oracle 11 58 0 6457M 6409M sleep 0:02 0.30% oracle
16404 oracle 11 58 0 6457M 6409M sleep 0:00 0.28% oracle
16327 oracle 11 55 0 6457M 6410M sleep 0:00 0.27% oracle
14870 oracle 11 58 0 6457M 6412M sleep 0:05 0.24% oracle
16851 oracle 11 35 0 6457M 6411M sleep 0:00 0.22% oracle
16467 oracle 11 58 0 6457M 6409M sleep 0:00 0.21% oracle
16163 oracle 11 58 0 6457M 6408M sleep 0:03 0.21% oracle
15159 oracle 11 58 0 6457M 6408M sleep 0:05 0.21% oracle
```

发现在 Top 输出中，使用了 12GB 的 Swap，而物理内存几乎耗尽：

Memory: 8192M real, 118M free, 12G swap in use, 11G swap free

至此可以初步做出以下判断：由于 SGA 设置过大（将近 7GB）导致运行时产生大量交换，大量 Swap 交换进而引发磁盘 I/O 问题。这也就应该是步骤 1 看到异步 I/O 错误的原因：

WARNING: aiowait timed out 1 times

大量交换导致数据库性能急剧下降，进而导致用户请求得不到快速响应，堵塞、累积，直至数据库失去响应。

5. 解决方案

此问题主要是由于 SGA 设置不当引起，马上缩小了 SGA 设置：

```
SQL> show sga

Total System Global Area 3591870848 bytes
Fixed Size                  735616 bytes
```

Variable Size	1442840576 bytes
Database Buffers	2147483648 bytes
Redo Buffers	811008 bytes

此时，数据库减少了交换，达到了稳定运行，用户请求可以得到快速响应，至此问题解决完成。

6. 系统调整后状态

调整后系统运行状况如下：

```
$ top

last pid: 12745; load averages: 0.46, 0.79, 0.65          22:22:49
228 processes: 227 sleeping, 1 on cpu
CPU states: 92.3% idle, 5.0% user, 1.6% kernel, 1.1% iowait, 0.0% swap
Memory: 8192M real, 3817M free, 4015M swap in use, 15G swap free

PID USERNAME THR PRI NICE SIZE RES STATE TIME CPU COMMAND
12610 oracle 1 51 0 3511M 22M sleep 0:04 1.96% oracle
12595 oracle 1 48 0 3511M 22M sleep 0:03 0.92% oracle
12630 oracle 1 38 0 3511M 21M sleep 0:01 0.84% oracle
12614 oracle 1 46 0 3511M 22M sleep 0:01 0.64% oracle
12620 oracle 1 58 0 3511M 22M sleep 0:01 0.53% oracle
12709 oracle 1 48 0 3511M 21M sleep 0:00 0.45% oracle
265 root 11 38 0 7032K 1920K sleep 3:16 0.42% picld
12729 oracle 1 0 0 3511M 20M sleep 0:00 0.26% oracle
12741 oracle 1 58 0 2768K 1760K cpu/3 0:00 0.19% top
12745 oracle 1 44 0 3506M 16M sleep 0:00 0.17% oracle
12711 oracle 1 48 0 3506M 16M sleep 0:00 0.11% oracle
12738 oracle 1 43 0 3506M 16M sleep 0:00 0.06% oracle
7606 oracle 1 45 0 17M 6928K sleep 0:07 0.05% tnslnr
12721 oracle 1 34 0 3506M 16M sleep 0:00 0.05% oracle
12723 oracle 1 53 0 3506M 16M sleep 0:00 0.05% oracle
```

该系统调整完以后，一直稳定运行至今。

7. 一点总结

这个案例和前面提到的另外一个极其相似，同样都是 SGA 设置不当引起的数据库问题。这些问题本身并不复杂，应该在数据库规划和建设阶段就避免掉。良好的规划和设计是系统稳定运行的基础。如果在生产环境中遇到这类问题，更重要的是快速判断，准确定位，及时

解决问题，减少故障对于业务系统的影响。所以，在这些案例处理的过程中，最重要的其实只是一个思路和想法。

8. 后续研究

在故障处理之后，进一步研究发现，这一问题在 Oracle 9.2.0.3 的 Solaris 平台上广泛存在，Oracle 为此记录了 Bug（Bug No: 2086687）并做出了改进。

在 Oracle 9.2.0.3 版本中，缺省情况下，Oracle 在异步 I/O 出现问题时，会连续 WARNING 100 次，每次间隔 10 分钟，也就是在 1000 分钟之后会给出 ORA-27083 错误：

```
[oracle@jumper oracle]$ oerr ORA 27083
27083, 00000, "skgfrliopo: waiting for async I/Os failed"
// *Cause: The aio_waitn() library call returned an error.
// *Action: Check errno.
```

这一设置让很多用户不满，因为连续 100 次的 I/O 超时可能已经给系统带来严重的影响，所以在 Oracle 9.2.0.4 版本中，Oracle 引入了一个新的隐含参数用以控制在报告 ORA-27083 错误前 WARNING 的次数，这个参数是 `_aiowait_timeouts`，该参数的缺省值为 100：

```
SQL> select * from v$version where rownum <2;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

SQL> @GetParDescrb.sql
Enter value for par: aiowait
old 6: AND x.ksppinm LIKE '%&par%'
new 6: AND x.ksppinm LIKE '%aiowait%'

NAME                                VALUE                                DESCRIB
-----
_aiowait_timeouts                    100                                Number of aiowait timeouts before error is reported
```

到这里，这个问题可以告一段落了。

4.3.3 诊断案例三：如何诊断和解决 CPU 高度消耗（100%）问题

很多时候服务器可能会经历 CPU 消耗 100% 的性能问题。排除系统的异常，这类问题通常都是因为系统中存在性能低下甚至存在错误的 SQL 语句，消耗了大量的 CPU 所致。本节将通过一个案例，就如何捕获这样的 SQL 给出一个通用的方法。

- 问题描述：系统 CPU 高度消耗，系统运行缓慢。
- 操作系统：Sun Solaris 8
- 数据库版本：Oracle 9.2.0.3

1. 首先通过 Top 命令查看

用 Top 工具检查系统运行状况：

```
$ top
```

load averages: 1.61, 1.28, 1.25 HSWAPJSDB 10:50:44

172 processes: 160 sleeping, 1 running, 3 zombie, 6 stopped, 2 on cpu

CPU states: 0.2% idle, 98.5% user, 1.3% kernel, 0.0% iowait, 0.0% swap

Memory: 4.0G real, 1.4G free, 1.9G swap in use, 8.9G swap free

PID	USERNAME	THR	PR	NCE	SIZE	RES	STATE	TIME	FLTS	CPU	COMMAND
20521	oracle	1	40	0	1.8G	1.7G	run	6:37	0	47.77%	oracle
20845	oracle	1	40	0	1.8G	1.7G	cpu02	0:41	0	40.98%	oracle
20847	oracle	1	58	0	1.8G	1.7G	sleep	0:00	0	0.84%	oracle
20780	oracle	1	48	0	1.8G	1.7G	sleep	0:02	0	0.83%	oracle
15828	oracle	1	58	0	1.8G	1.7G	sleep	0:58	0	0.53%	oracle
20867	root	1	58	0	4384K	2560K	sleep	0:00	0	0.29%	sshd2
20493	oracle	1	58	0	1.8G	1.7G	sleep	0:03	0	0.29%	oracle
20887	oracle	1	48	0	1.8G	1.7G	sleep	0:00	0	0.13%	oracle
20851	oracle	1	58	0	1.8G	1.7G	sleep	0:00	0	0.10%	oracle
20483	oracle	1	48	0	1.8G	1.7G	sleep	0:00	0	0.09%	oracle
20875	oracle	1	45	0	1064K	896K	sleep	0:00	0	0.07%	sh
20794	oracle	1	58	0	1.8G	1.7G	sleep	0:00	0	0.06%	oracle
20842	jiankong	1	52	2	1224K	896K	sleep	0:00	0	0.05%	sadc
20888	oracle	1	55	0	1712K	1272K	cpu00	0:00	0	0.05%	top
19954	oracle	1	58	0	1.8G	1.7G	sleep	84:25	0	0.04%	oracle

发现在进程列表里，存在两个高 CPU 耗用的 Oracle 进程，分别消耗了 47.77% 和 40.98% 的 CPU 资源。

2. 找到存在问题的进程信息

确认这是两个远程连接的用户进程：

```
$ ps -ef|grep 20521
oracle 20909 20875 0 10:50:53 pts/10 0:00 grep 20521
oracle 20521 1 47 10:43:59 ? 6:45 oraclejshs (LOCAL=NO)

$ ps -ef|grep 20845
oracle 20845 1 44 10:50:00 ? 0:55 oraclejshs (LOCAL=NO)
oracle 20918 20875 0 10:50:59 pts/10 0:00 grep 20845
```

3. 捕获存在问题的 SQL 语句

通过如下 `getsql.sql` 脚本，可以获取相关 SQL 语句：

```
SELECT /*+ ORDERED */
       sql_text
FROM v$sqltext a
WHERE (a.hash_value, a.address) IN (
        SELECT DECODE (sql_hash_value,
                        0, prev_hash_value,
                        sql_hash_value
                        ),
        DECODE (sql_hash_value, 0, prev_sql_addr, sql_address)
FROM v$session b
WHERE b.paddr = (SELECT addr
                  FROM v$process c
                  WHERE c.spid = '&pid'))
ORDER BY piece ASC
/
```

注意这里涉及了 3 个视图，并应用其关联进行数据获取。

首先需要输入一个 PID，这个 PID 即 Process ID，也就是在 Top 或 ps 中看到的 PID。通过 PID 和 `v$process.spid` 相关联，可以获得 Process 的相关信息，进而通过 `v$process.addr` 和 `v$session.paddr` 相关联，就可以获得和 session 相关的所有信息。再结合 `v$sqltext`，就可获得当前 session 正在执行的 SQL 语句。通过 `v$process` 视图，我们得以把操作系统和数据库关联了起来。

4. 连接数据库，找到问题 sql 及进程

通过 Top 工具，观察到 PID，进而应用我的 `getsql` 脚本，得到以下结果输出：

```
$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.3.0 - Production on Mon Dec 29 10:52:14 2003

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.3.0 - 64bit Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.3.0 - Production
```

```
SQL> @getsql
Enter value for spid: 20521
old 10: where c.spid = '&pid'
new 10: where c.spid = '20521'

SQL_TEXT
-----

select * from (select VC2URL,VC2PVDID,VC2MOBILE,VC2ENCRYPTFLAG,S
ERVICEID,VC2SUB_TYPE,CISORDER,NUMGUID,VC2KEY1, VC2NEEDDISORDER,V
C2PACKFLAG,datapertime from hsv_2cpsync where datapertime<=sysda
te and numguid>70000000000308 order by NUMGUId) where rownum<=20
```

那么这段代码就是当前正在疯狂消耗 CPU 的罪魁祸首, 接下来需要进行的工作就是找出这段代码的问题, 看是否可以通过优化提高其效率, 减少资源消耗。

5. 进一步的跟踪

如果需要进一步的跟踪详细信息, 可以通过 `dbms_system` 包来进行:

```
SQL> @getsid
Enter value for spid: 20521
old 3: select addr from v$process where spid = &spid)
new 3: select addr from v$process where spid = 20521)

SID SERIAL# USERNAME MACHINE
-----
45 38991 HSUSER_V51 hswapjsptl1.hurray.com.cn

SQL> exec dbms_system.set_sql_trace_in_session(45,38991,true);

PL/SQL procedure successfully completed.

SQL> !
```

关于 `dbms_system` 包的使用, 在后面的章节将会有详细的介绍。

6. 一点说明

很多时候, 高 CPU 消耗都是由于问题 SQL 导致的, 所以找到这些 SQL 通常也就找到了问题所在, 通过优化调整通常就可以解决问题。

但是有时候可能会发现, 这些最消耗 CPU 的进程是后台进程, 这一般是由于异常、Bug 或者恢复后的异常导致的, 那么就需要具体问题具体分析了。

第5章 Buffer Cache 与 Shared Pool 原理

Buffer Cache 与 Shared Pool 是 SGA 中的最重要和最复杂的两个部分,在此本书用一章的篇幅来对 Buffer Cache 和 Shared Pool 进行深入探讨。

5.1 Buffer Cache 原理

Buffer Cache 是 Oracle SGA 中的一个重要部分,通常的数据访问和修改都需要通过 Buffer Cache 来完成。当一个进程需要访问数据时,首先需要确定数据在内存中是否存在,如果数据在 Buffer 中存在,则需要根据数据的状态来判断是否可以直接访问还是需要构造一致性读取;如果数据在 Buffer 中不存在,则需要在 Buffer Cache 中寻找足够的空间以装载需要的数据,如果 Buffer Cache 中找不到足够的内存空间,则需要触发 DBWR 去写出脏数据,释放 Buffer 空间。

这样一个过程,描述起来并不复杂,但是在数据库的处理过程中实际上是相当复杂的。在以上的描述中,有几个问题需要考虑,首先,Oracle 如何才能快速定位到 Buffer 中是否存在需要的数据呢?如果请求的数据在 Cache 中不存在,那么 Oracle 又是如何去 Buffer Cache 中快速寻找内存空间呢?

5.1.1 LRU 与 Dirty List

在 Buffer Cache 中,Oracle 通过几个链表进行内存管理,其中最为熟知的是 LRU List 和 Dirty List (也经常被称为 Write List,从 Oracle 8i 开始,因为算法的改变,也被称为 Checkpoint Queue),各种 List 上存放的是指向具体 Buffer 的指针,图 5-1 简要地说明了这一关系。

LRU List 用于维护内存中的 Buffer,按照 LRU 算法进行管理(在不同版本中,管理方式有所不同,本章不打算详细介绍算法部分),数据库初始化时,所有的 Buffer 都被 Hash 到 LRU List 上管理。当需要从数据文件上读取数据时,首先要在 LRU List 上寻找 Free 的 Buffer,然后读取数据到 Buffer Cache 中;当数据被修改之后,状态变为 Dirty,就可以被移动至 Dirty List (Checkpoint Queue),Dirty List 上的都是候选的可以被 DBWR 写出到数据文件的 Buffer,一个 Buffer 要么在 LRU List 上,要么在 Dirty List 上存在,不能同时存在于多个 List。

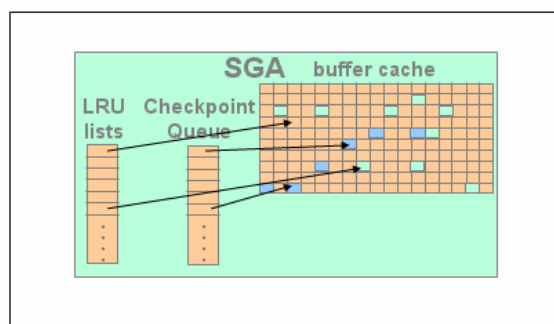


图 5-1 Buffer Cache 与 List

下面通过图 5-2 详细介绍一下 Buffer Cache 的原理及使用。

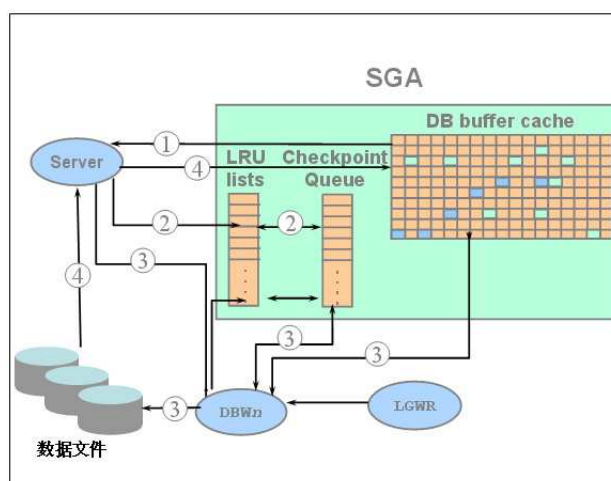


图 5-2 Buffer Cache 的原理及使用

(1) 当一个 Server 进程需要读数据到 Buffer Cache 中时，首先必须判断该数据在 Buffer 中是否存在(图中①所示过程)，如果存在且可用，则获取该数据，根据 LRU 算法在 LRU List 上移动该 Block；如果 Buffer 中不存在该数据，则需要从数据文件上读取。

(2) 在读取数据之前，Server 进程需要扫描 LRU List 寻找 Free 的 Buffer，扫描过程中 Server 进程会把发现的所有已经被修改过的 Buffer 移动到 Checkpoint Queue 上(图中②所示过程)，这些 Dirty Buffer 随后可以被写出到数据文件。

(3) 如果 Checkpoint Queue 超过了阈值，Server 进程就会通知 DBWn 去写出脏数据(图中③所示过程)；这也是触发 DBWn 写的一个条件，这个阈值曾经提到是 25%，也就是当检查点队列超过 25% 满就会触发 DBWn 的写操作：

```
SQL> select kvittag,kvitval,kvtdsc from x$kvit
```

```
2 where kvittag='kcbl dq';
```

KVITTAG

KVITVAL KVITDSC

kcbl dq

25 large dirty queue if kcbclw reaches this

如果 Server 进程扫描 LRU 超过一个阈值仍然不能找到足够的 Free Buffer，将停止寻找，转而通知 DBWn 去写出脏数据，释放内存空间。

同样这个阈值可以从以上字典表中查询得到，这个数字是 40%，也就是说当 Server 进程扫描 LRU 超过 40% 还没能找到足够的 Free Buffer 就会停止搜索，通知 DBWn 执行写出，这时进程会处于 free busy wait 等待：

```
SQL> select kvittag,kvitval,kvitdsc from x$kvit
```

```
2 where kvittag='kcbfsp';
```

```
KVITTAG
```

```
KVITVAL KVITDSC
```

```
-----
```

```
kcbfsp
```

```
40 Max percentage of LRU list foreground can scan for free
```

同时由于增量检查点的引入，DBWn 也会主动扫描 LRU List，将发现的 Dirty Buffer 移至 Checkpoint Queue，这个扫描也受一个内部约束，在 Oracle 9iR2 中，这个比例是 25%。

```
SQL> select kvittag,kvitval,kvitdsc from x$kvit
```

```
2 where kvittag='kcbdsp';
```

```
KVITTAG
```

```
KVITVAL KVITDSC
```

```
-----
```

```
kcbdsp
```

```
25 Max percentage of LRU list dbwriter can scan for dirty
```

(4) 找到足够的 Buffer 之后，Server 进程就可以将 Buffer 从数据文件读入 Buffer Cache (图中④所示过程)。

(5) 如果读取的 Block 不满足读一致性需求，则 Server 进程需要通过当前 Block 版本和回滚段构造前镜像返回给用户。

从 Oracle 8i 开始，LRU List 和 Dirty List 又分别增加了辅助 List (AUXILIARY List)，用于提高管理效率。引入了辅助 List 之后，当数据库初始化时，Buffer 首先存放在 LRU 的辅助 List 上 (AUXILIARY RPL_LST)，当被使用后移动到 LRU 主 List 上 (MAIN RPL_LST)，这样当用户进程搜索 Free Buffer 时，就可以从 LRU-AUX List 开始，而 DBWR 搜索 Dirty Buffer 时，则可以从 LRU-Main List 开始，从而提高了搜索效率和数据库性能。

可以通过如下命令转储 Buffer Cache 的内容，从而清晰地看到以上描述的数据结构：

```
alter session set events 'immediate trace name buffers level 4';
```

不同 level 转储的内容详细程度不同，此命令的可用级别主要有 1~10 级，其中各级别的含义如下。

- Level 1: 仅包含 Buffer Headers 信息。
- Level 2: 包含 Buffer Headers 和 Buffer 概要信息转储。
- Level 3: 包含 Buffer Headers 和完整 Buffer 内容转储。
- Level 4: Level 1 + Latch 转储 + LRU 队列。
- Level 5: Level 4 + Buffer 概要信息转储。
- Level 6 和 Level 7: Level 4 + 完整的 Buffer 内容转储。

- Level 8: Level 4 + 显示 users/waiters 信息。
- Level 9: Level 5 + 显示 users/waiters 信息。
- Level 10: Level 6 + 显示 users/waiters 信息。

转储仅限于在测试环境中使用，转储的跟踪文件可能非常巨大，为获取完整的跟踪文件，建议设置初始化参数 `max_dump_file_size` 为 `UNLIMITED`。

本节选取的环境为：

```
SQL> select * from v$version;
```

BANNER

Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

PL/SQL Release 9.2.0.4.0 - Production

CORE 9.2.0.3.0 Production

TNS for Linux: Version 9.2.0.4.0 - Production

NLSRTL Version 9.2.0.4.0 - Production

主要相关参数设置为：

```
SQL> show parameter max_dump
```

NAME	TYPE	VALUE
max_dump_file_size	string	UNLIMITED

```
SQL> show parameter db_cache_size
```

NAME	TYPE	VALUE
db_cache_size	big integer	16777216

从 Level 4 级跟踪文件的开头部分，可以获得如下信息：

Dump of buffer cache at level 4

.....

MAIN RPL_LST Queue header (NEXT_DIRECTION)[NULL]

MAIN RPL_LST Queue header (PREV_DIRECTION)[NULL]

AUXILIARY RPL_LST Queue header (NEXT_DIRECTION)[NULL]

AUXILIARY RPL_LST Queue header (PREV_DIRECTION)[NULL]

MAIN WRT_LST Queue header (NEXT_DIRECTION)[NULL]

MAIN WRT_LST Queue header (PREV_DIRECTION)[NULL]

AUXILIARY WRT_LST Queue header (NEXT_DIRECTION)[NULL]

AUXILIARY WRT_LST Queue header (PREV_DIRECTION)[NULL]

```

MAIN XOBJ_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN XOBJ_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY XOBJ_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY XOBJ_LST Queue header (PREV_DIRECTION)[NULL]
MAIN XRNG_LST Queue header (NEXT_DIRECTION)[NULL]
MAIN XRNG_LST Queue header (PREV_DIRECTION)[NULL]
AUXILIARY XRNG_LST Queue header (NEXT_DIRECTION)[NULL]
AUXILIARY XRNG_LST Queue header (PREV_DIRECTION)[NULL]
.....

```

还可以看到，Buffer Cache 中除了 RPL_LST 和 WRT_LST 外还存在其他分类的 List，作用各不相同，为了不使本章过于复杂，不再过多介绍。

同时在 Level 4 级的转储中，还可以看到主要 LST 的队列信息，这也是链表的一个最直观的表现：

```

MAIN RPL_LST Queue header (NEXT_DIRECTION)[51bfcccc,51bfe8dc]
0x51bfbcb8=>0x51bfbd40=>0x51bfbd9c=>0x51bfbeb8=>0x51bfbf74=>0x51bfc030=>0x51bfc0ec=>0x51bfc1a8
0x51bfc264=>0x51bfc320=>0x51bfc3dc=>0x51bfc498=>0x51bfc554=>0x51bfc610=>0x51bfc6cc=>0x51bfc788
0x51bfc844=>0x51bfc900=>0x51bfc9bc=>0x51bfca78=>0x51bfcbb4=>0x51bfcbf0=>0x51bfccac=>0x51bfcd68
0x51bfce24=>0x51bfcee0=>0x51bfcf9c=>0x51bfd058=>0x51bfd114=>0x51bfd1d0=>0x51bfd28c=>0x51bfd348

```

5.1.2 Cache Buffers Lru Chain 门锁竞争与解决

当用户进程需要读数据到 Buffer Cache 时，或 Cache Buffer 根据 LRU 算法进行管理时，就不可避免地要扫描 LRU List 获取可用 Buffer 或更改 Buffer 状态，我们知道，Oracle 的 Buffer Cache 是共享内存，可以为众多并发进程并发访问，所以在搜索的过程中必须获取 Latch（Latch 是 Oracle 的一种串行锁机制，用于保护共享内存结构），锁定内存结构，防止并发访问损坏内存中的数据。

这个用于锁定 LRU 的 Latch 就是经常见到的 Cache Buffers Lru Chain。

```

SQL> col name for a25
SQL> select addr,latch#,name,gets,misses,immediate_gets,immediate_misses
       2  from v$latch where name = 'cache buffers lru chain';

```

ADDR	LATCH#	NAME	GETS	MISSSES	IMMEDIATE_GETS	IMMEDIATE_MISSES
1200BFA8	93	cache buffers lru chain	40851181	11972	4608605853	2965271

Cache Buffers Lru Chain Latch 存在多个子 Latch，其数量受隐含参数 `_db_block_lru_latches`

控制：

```
SQL> @GetParDescrb.sql
```

```
Enter value for par: lru
```

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

```
new 6: AND x.ksppinm LIKE '%lru%'
```

NAME	VALUE	DESCRIB

_db_block_lru_latches	64	number of lru latches

可以从 v\$latch_children 视图查看当前各子 Latch 使用情况：

```
SQL> select addr,child#,name,gets,misses,immediate_gets igets,immediate_misses imisses
2 from v$latch_children where name = 'cache buffers lru chain';
```

ADDR	CHILD# NAME	GETS	MISSES	IGETS	IMISSES

14C7E7F4	64 cache buffers lru chain	174	0	0	0
14C7E328	63 cache buffers lru chain	174	0	0	0
.....					
14C73678	27 cache buffers lru chain	174	0	0	0
14C731AC	26 cache buffers lru chain	174	0	0	0
14C72CE0	25 cache buffers lru chain	174	0	0	0
14C72814	24 cache buffers lru chain	5168833	2009	570411112	391045
14C72348	23 cache buffers lru chain	5179461	1685	577348879	369675
14C71E7C	22 cache buffers lru chain	5079655	1526	570766961	368414
14C719B0	21 cache buffers lru chain	5082539	1466	579774239	373785
14C714E4	20 cache buffers lru chain	5077378	1288	572272107	360877
14C71018	19 cache buffers lru chain	5084936	1396	586932913	368418
14C70B4C	18 cache buffers lru chain	5120549	1309	572987405	363633
14C70680	17 cache buffers lru chain	5069659	1296	578336549	369442
14C701B4	16 cache buffers lru chain	174	0	0	0
.....					
14C6C358	3 cache buffers lru chain	174	0	0	0
14C6BE8C	2 cache buffers lru chain	174	0	0	0
14C6B9C0	1 cache buffers lru chain	174	0	0	0

64 rows selected.

如果该 Latch 竞争激烈，通常有如下方法可以采用。

(1) 适当增大 Buffer Cache，这样可以减少读数据到 Buffer Cache 的机会，减少扫描 LRU List 的竞争。

(2) 可以适当增加 LRU Latch 的数量，修改 `_db_block_lru_latches` 参数可以实现，但是该参数通常来说是足够的，除非在 Oracle Support 的建议下或确知该参数将带来的影响，否则不推荐修改。

(3) 通过多缓冲池技术，可以减少不希望的数据老化和全表扫描等操作对于 Default 池的冲击，从而可以减少竞争。

5.1.3 Cache Buffer Chain 门锁竞争与解决

在 LRU 和 Dirty List 这两个内存结构之外，Buffer Cache 的管理还存在另外两个重要的数据结构：Hash Bucket 和 Cache Buffer Chain。

1. Hash Bucket 和 Cache Buffer Chain

可以想象，如果所有的 Buffer Cache 中的所有 Buffer 都通过同一个结构管理，当需要确定某个 Block 在 Buffer 中是否存在时，将需要遍历整个结构，性能会相当低下。

为了提高效率，Oracle 引入了 Bucket 的数据结构，Oracle 把管理的所有 Buffer 通过一个内部的 Hash 算法运算后，存放到不同 Hash Bucket 中，这样通过 Hash Bucket 进行分割之后，众多的 Buffer 被分布到一定数量的 Bucket 之中，当用户需要在 Buffer 中定位数据是否存在时，只需要通过同样的算法获得 Hash 值，然后到相应的 Bucket 中查找少量的 Buffer 即可确定。每个 Buffer 存放的 Bucket 由 Buffer 的数据块地址 (DBA, Data Block Address) 运算决定。

Bucket 内部，通过 Cache Buffer Chain (Cache Buffer Chain 是一个双向链表) 将所有的 Buffer 通过 Buffer Header 信息联系起来。

Buffer Header 存放的是对应数据块的概要信息，包括数据块的文件号、块地址、状态等。要判断数据块在 Buffer 中是否存在，通过检查 Buffer header 即可确定。

如果大家去过老一点的图书馆，查找过手工索引，你可能记得这样的场景：树立在你面前的是一排柜子（那是相当的壮观），柜子又被分为很多小的抽屉，抽屉上按照不同的分类方法标注了相关信息，比如按开头字母顺序，如果要查询 Oracle 相关书籍，就需要找到标记有“O”的抽屉。打开抽屉，会看到一系列的卡片，这些卡片通常被一根铁门串起来（通常就是一个铁丝），每张卡片上会记录相关书籍的信息，可能包括书籍名称、作者、ISBN 号、出版日期等，当然这些卡片上还存储了一个重要的信息，就是书籍存放的书架位置信息，有了这个信息，通过翻阅这些卡片，就可以快速地找到我们想要的书籍，并且在需要时能够快速从图书馆浩如烟海的图书中找到我们需要的那一本。

在这里，图书馆就是我们的 Buffer Cache，这个 Cache 可能因为“图书数量”的增加而不断扩大；每个抽屉都是一个 Bucket，这个 Bucket 中存放了根据一定的分类方式（也就是通过 Hash 运算）归入的图书信息，也就是 Buffer Header；抽屉中的每张卡片就是一个 Buffer Header，这些 Buffer Header 上记录了关于数据块的重要信息，如 DBA 等；这些卡片在 Bucket 中，通过一个铁门串接起来，这就是 Cache Buffer Chain。

由于每个抽屉只有一根铁门，如果很多读者都想翻阅这个链上的卡片，那么就产生了

Cache Buffer Chain 的竞争，先来的那个读者持有了 Latch 就能不停地翻阅，其他读者只好不停地来检查，当然如果检查次数多了（超过了 `_spin_count`），也可以去休息室小憩一会，再来和其他读者争夺。

从 Oracle 9i 开始，对于 Cache Buffer Chain 的只读访问，其 Latch 可以被共享；也就是说，如果大家都只是翻一翻卡片，那么大家可以一起来看，但是如果有人要借走这本书，那么就只能独享这个 Latch 了。

这就是 Buffer Cache 与 Latch 竞争。

由于 Buffer 根据 Buffer Header 进行散列，从而最终决定存入哪一个 Hash Bucket，那么 Hash Bucket 的数量在一定程度上就决定了每个 Bucket 中 Buffer 数量的多少，也就间接影响了搜索的性能。

所以在不同版本中，Oracle 一直在修改算法，优化 Hash Bucket 的数量。可以想象，Bucket 的数量多一些，那么在同一时间就可以有更多的读者可以拿到不同的抽屉，进行数据访问；但是更多的抽屉，显然需要更多的存放空间，更多的管理成本，所以优化在什么时候都不是简单的一元方程。

Hash Bucket 的设置受一个隐含参数 `_db_block_hash_buckets` 的影响。在 Oracle 7 和 Oracle 3 中，该参数缺省值为 `db_block_buffers/4` 的下一个素数。在 Oracle 8i 中，该参数缺省为 `lb_block_buffers*2`。

```
SQL> select * from v$version where rownum <2;
BANNER
-----
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
SQL> @GetParDescrb.sql
Enter value for par: hash_buckets
old   6:   AND x.kspinm LIKE '%&par%'
new   6:   AND x.kspinm LIKE '%hash_buckets%'

NAME                                VALUE                                DESCRIB
-----
_db_block_hash_buckets              50000                                Number of database block hash buckets

SQL> show parameter db_block_buffers

NAME                                TYPE                                VALUE
-----
db_block_buffers                    integer                             25000
```

通过以上讨论可以知道，对应每个 Bucket，只存在一个 Chain，当用户试图搜索 Cache Buffer Chain 时，必须首先获得 Cache Buffer Chain Latch。那么 Cache Buffer Chain Latch 的设置就同样值得研究了。

在 Oracle 8i 之前，对于每一个 Hash Bucket，Oracle 使用一个独立的 Hash Latch 来维护，其缺省 Bucket 数量为 $\text{next_prime}(\text{db_block_buffers}/4)$ 。

由于过于严重的热点块竞争，从 Oracle 8i 开始，Oracle 改变了这个算法，首先 Bucket 数量开始增加，`_db_block_hash_buckets` 增加到 $2 * \text{db_block_buffers}$ ，而 `_db_block_hash_latches` 的数量也发生了变化。

- 当 Cache Buffers 少于 2052 Buffers:

`_db_block_hash_latches = power(2, trunc(log(2, db_block_buffers - 4) - 1))`

- 当 Cache Buffers 多于 131075 Buffers:

`_db_block_hash_latches = power(2, trunc(log(2, db_block_buffers - 4) - 6))`

- 当 Cache Buffers 位于 2052 与 131075 Buffers 之间:

`_db_block_hash_latches = 1024`

从 Oracle 8i 开始，`_db_block_hash_buckets` 的数量较以前增加了 8 倍，而 `_db_block_hash_latches` 的数量增加有限，这意味着，每个 Latch 需要管理多个 Bucket，但是由于 Bucket 数量的多倍增加，每个 Bucket 上的 Block 数量得以减少，从而使少量 Latch 管理更多 Bucket 成为可能。

图 5-3 简要地描述这个变化（图中省略了一些内容）：

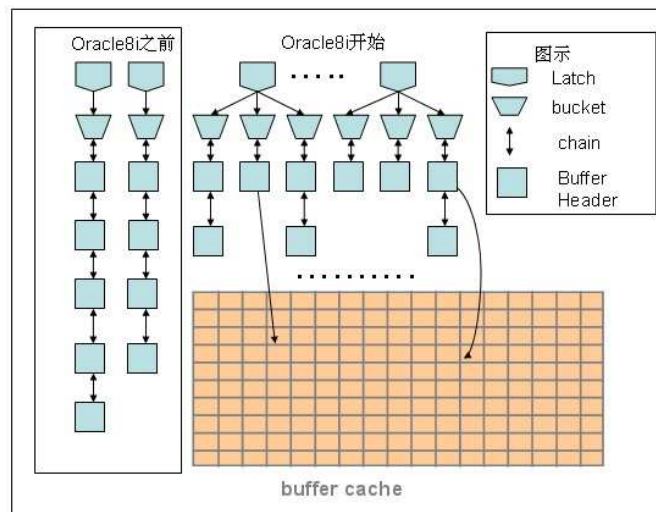


图 5-3 Oracle 8i 的变化

总结一下图 5-3 中描述的内容：

- (1) 从 Oracle 8i 开始，Bucket 的数量比以前大大增加；通过增加的 Bucket 的“稀释”使得每个 Bucket 上的 Buffer 数量大大减少。
- (2) 在 Oracle 8i 之前，`_db_block_hash_latches` 的数量和 Bucket 的数量是一致的，每个 Latch 管理一个 Bucket；从 Oracle 8i 开始每个 Latch 需要管理多个 Bucket，由于每个 Bucket 上的 Buffer 数量大大降低，所以 Latch 的性能反而得到了提高。
- (3) 每个 Bucket 存在一条 Cache Buffer Chain。
- (4) Buffer Header 上存在指向具体 Buffer 的指针。

以下是一个 Oracle 8i 数据库的查询输出：

```
SQL> @GetParDescrb.sql
```

```
Enter value for par: hash_latches
```

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

```
new   6:   AND x.ksppinm LIKE '%hash_latches%'
```

NAME	VALUE	DESCRIB

_db_block_hash_latches	1024	Number of database block hash latches

```
SQL> show parameter db_block_buffers
```

NAME	TYPE	VALUE

db_block_buffers	integer	25000

```
SQL>
```

下面通过试验来验证以上的一些说明。

首先我的测试库 `db_cache_size` 设置为 16MB，此时的 `_db_block_hash_buckets` 为 4001：

```
SQL> show parameter db_cache_size
```

NAME	TYPE	VALUE

db_cache_size	big integer	16777216

```
SQL> @GetParDescrb.sql
```

```
Enter value for par: bucket
```

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

```
new   6:   AND x.ksppinm LIKE '%bucket%'
```

NAME	VALUE	DESCRIB

_db_block_hash_buckets	4001	Number of database block hash buckets

转储一下 Buffer，获得跟踪文件：

```
SQL> alter session set events 'immediate trace name buffers level 10';
```

```
Session altered.
```

```
SQL> !
```

```
[oracle@jumper udump]$ ls
eygle_ora_1197.trc
```

跟踪文件中的 Cache Buffer Chain 的数量正好是 4001:

```
[oracle@jumper udump]$ grep CHAIN eygle_ora_1197.trc |wc -l
4001
```

某些 Chain 上可能没有 Buffer Header 信息 (标记为 NULL), 这些 Chain 的数据类似如下显示:

```
[oracle@jumper udump]$ grep CHAIN eygle_ora_1197.trc |head -20
CHAIN: 0 LOC: 0x0x532996b0 HEAD: [51ff3fac,51ff3fac]
CHAIN: 1 LOC: 0x0x532997d0 HEAD: [NULL]
CHAIN: 2 LOC: 0x0x532998f0 HEAD: [NULL]
CHAIN: 3 LOC: 0x0x53299a10 HEAD: [NULL]
CHAIN: 4 LOC: 0x0x53299b30 HEAD: [NULL]
CHAIN: 5 LOC: 0x0x53299c50 HEAD: [NULL]
CHAIN: 6 LOC: 0x0x53299d70 HEAD: [NULL]
CHAIN: 7 LOC: 0x0x53299e90 HEAD: [NULL]
CHAIN: 8 LOC: 0x0x53299fb0 HEAD: [51ff3ef0,51ff3ef0]
CHAIN: 9 LOC: 0x0x5329a0d0 HEAD: [NULL]
CHAIN: 10 LOC: 0x0x5329a1f0 HEAD: [NULL]
CHAIN: 11 LOC: 0x0x5329a310 HEAD: [NULL]
CHAIN: 12 LOC: 0x0x5329a430 HEAD: [NULL]
CHAIN: 13 LOC: 0x0x5329a550 HEAD: [NULL]
CHAIN: 14 LOC: 0x0x5329a670 HEAD: [NULL]
CHAIN: 15 LOC: 0x0x5329a790 HEAD: [NULL]
CHAIN: 16 LOC: 0x0x5329a8b0 HEAD: [517f6174,517e99b4]
CHAIN: 17 LOC: 0x0x5329a9d0 HEAD: [NULL]
CHAIN: 18 LOC: 0x0x5329aaf0 HEAD: [NULL]
CHAIN: 19 LOC: 0x0x5329ac10 HEAD: [NULL]
```

摘录一个 Chain 8 的数据给大家参考:

```
CHAIN: 8 LOC: 0x0x53299fb0 HEAD: [51ff3ef0,51ff3ef0]
    BH (0x0x51ff3ef0) file#: 1 rdba: 0x00400ac1 (1/2753) class 4 ba: 0x0x51e08000
    set: 3 dbwrid: 0 obj: 391 objn: -1
    hash: [53299fb0,53299fb0] lru: [51ff3ff4,51ff3e7c]
    LRU flags:
    ckptq: [NULL] fileq: [NULL]
    st: CR md: NULL rsop: 0x(nil) tch: 1
```

```

cr:[[scn: 0x0000.000bed07],[xid: 0x0000.000.00000000],[uba: 0x00000000.0000.00],[cls: 0x0000.
000bed07],[sfl: 0x0]]

buffer ts: 0 rdba: 0x00400ac1 (1/2753)

scn: 0x0000.00000fd4 seq: 0x01 flg: 0x04 tail: 0x0fd41001

frmt: 0x02 chkval: 0x4d32 type: 0x10=DATA SEGMENT HEADER - UNLIMITED

Extent Control Header

-----

Extent Header:: spare1: 0      spare2: 0      #extents: 1      #blocks: 7

                last map  0x00000000  #maps: 0      offset: 4128

Highwater:: 0x00400ac3  ext#: 0      blk#: 1      ext size: 7

#blocks in seg. hdr's freelists: 0

#blocks below: 1

mapblk 0x00000000  offset: 0

                Unlocked

Map Header:: next 0x00000000  #extents: 1  obj#: 391  flag: 0x40000000

Extent Map

-----

0x00400ac2  length: 7

nfl = 1, nfb = 1 typ = 2 nxf = 0 ccnt = 0

SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000

```

这个 Chain 中存在一个 BH 信息。注意其中包含“**hash:[53299fb0,53299fb0] lru: [51ff3ff4,51ff3e7c]**”。

“hash:[53299fb0,53299fb0]”中的两个数据分别代表 X\$BH 中的 NXT_HASH 和 PRV_HASH，也就是指同一个 Hash Chain 上的下一个 BH 地址和上一个 Buffer 地址。如果某个 Chain 只包含一个 BH，那么这两个值将同时指向该 Chain 地址。

“lru:[51ff3ff4,51ff3e7c]”中的两个数据分别代表 X\$BH 中的 NXT_REPL 和 PRV_REPL，也就是 LRU 上的下一个 Buffer 和上一个 Buffer。

说 Buffer Header 是一个双向链就是从这里实现的，从 Oracle 8i 开始，由于 Bucket 数量的增加，通常不容易见到包含多个 BH 的 Bucket，以下是从一个生产环境中转储的 Buffer 信息，选取的 Chain 包含两个 Buffer Header 供大家参考：

```

CHAIN: 1598 LOC: 0x407200880 HEAD: [3c4f8bd00,3e9fc6000]

BH (0x3c4f8bd00) file#: 52 rdba: 0x0d01d126 (52/119078) class 1 ba: 0x3c413a000

set: 6 dbwrid: 0 obj: 148015 objn: 148015

hash: [3e9fc6000,407200880] lru: [38bfbd468,3bffca068]

LRU flags: hot_buffer

ckptq: [NULL] fileq: [NULL]

```

```

st: XCURRENT md: NULL rsop: 0x0 tch: 5
LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]
buffer tsn: 41 rdba: 0x0d01d126 (52/119078)
scn: 0x0819.184596ea seq: 0x01 flg: 0x06 tail: 0x96ea0601
frmt: 0x02 chkval: 0x5332 type: 0x06=trans data
.....
BH (0x3e9fc6000) file#: 51 rdba: 0x0cc1c8ce (51/116942) class 1 ba: 0x3e9880000
set: 6 dbwrid: 0 obj: 151861 objn: 151861
hash: [407200880,3c4f8bd00] lru: [3a3fdf068,3fcfd168]
LRU flags: hot_buffer
ckptq: [NULL] fileq: [NULL]
st: XCURRENT md: NULL rsop: 0x0 tch: 0
LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]
buffer tsn: 42 rdba: 0x0cc1c8ce (51/116942)
scn: 0x0819.193f1d34 seq: 0x01 flg: 0x06 tail: 0x1d340601
frmt: 0x02 chkval: 0xd8d1 type: 0x06=trans data

```

了解了以上管理算法，很容易想象，如果大量进程对相同的 Block 进程进行操作，那么必然引发 Cache Buffer Chain 的竞争，也就是通常所说的热点块的竞争。

下面从 Buffer Header 继续我们的讨论。

2. X\$BH 与 Buffer Header

Buffer Header 数据，可以从数据库的字典表中查询得到，这张字典表是 X\$BH。X\$BH 中的 BH 就是指 Buffer Headers，每个 Buffer 在 X\$BH 中都存在一条记录：

```

Connected to:
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
With the Partitioning option
JServer Release 8.1.7.4.0 - 64bit Production

SQL> select count(*) from x$bh;

COUNT(*)
-----
25000

SQL> show parameter db_block_buffers;

NAME                                TYPE    VALUE

```

```
-----
db_block_buffers                integer 25000
```

Buffer Header 中存储每个 Buffer 容纳的数据块的文件号、块地址、状态等重要信息，根据这些信息，结合 dba_extents 视图，可以很容易地找到每个 Buffer 对应的对象信息：

```
SQL> desc x$bh;

Name                                Null?    Type
-----
ADDR                                RAW(8)
.....
HLADDR                              RAW(8)
.....
TS#                                  NUMBER
FILE#                               NUMBER
DBARFIL                             NUMBER
DBABLK                              NUMBER
CLASS                               NUMBER
STATE                               NUMBER
MODE_HELD                           NUMBER
CHANGES                            NUMBER
.....
TCH                                  NUMBER
```

X\$BH 中还有一个重要字段 TCH，TCH 为 Touch 的缩写，表示一个 Buffer 的访问次数，Buffer 被访问的次数越多，说明该 Buffer 越“抢手”，也就可能存在热点块竞争的问题。

下面做几个简单的查询供大家参考。

可以通过以下查询获得当前数据库最繁忙的 Buffer（以下查询来自一个生产数据库）：

```
SQL> SELECT *
2    FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
3               FROM x$bh
4               ORDER BY tch DESC)
5    WHERE ROWNUM < 11;
```

ADDR	TS#	FILE#	DBARFIL	DBABLK	TCH
FFFFFFFF7AFD1110	32	33	33	4732	1079
FFFFFFFF7AFD1110	32	33	33	4956	1079
FFFFFFFF7AFD1110	32	33	33	66	1078
FFFFFFFF7AFD1110	32	33	33	4492	1078

FFFFFFFF7AFD0E40	28	20	20	348	1078
FFFFFFFF7AFD0E40	28	20	20	355	1078
FFFFFFFF7AFD0E40	28	20	20	351	1078
FFFFFFFF7AFD0E40	28	20	20	349	1078
FFFFFFFF7AFD0E40	28	20	20	364	1078
FFFFFFFF7AFD0E40	28	20	20	360	1078

10 rows selected.

再结合 dba_extents 中的信息，可以查询得到这些热点 Buffer 都来自哪些对象：

```
SQL> SELECT e.owner, e.segment_name, e.segment_type
2          FROM dba_extents e,
3          (SELECT *
4            FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
5                      FROM x$bh
6                      ORDER BY tch DESC)
7            WHERE ROWNUM < 11) b
8          WHERE e.relative_fno = b.dbarfil
9          AND e.block_id <= b.dbablk
10         AND e.block_id + e.blocks > b.dbablk;
```

OWNER	SEGMENT_NAME	SEGMENT_TYPE
BOSSV2	HY_AREA	TABLE
BOSSV2	HYUIDX_PLATFORMID	INDEX
BOSSV2	HYUIDX_MOBILESEG	INDEX
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	HSOLAP_RPTJOB	TABLE
HSQUERY	PK_HSOLAP_RPTJOB	INDEX

10 rows selected.

除了查询 X\$BH 之外，也可以从 Buffer Cache 的转储信息中，看到 Buffer Header 的具体内容，以下信息来自 Level 1 级的 Buffer Dump：

```

Dump of buffer cache at level 1
BH (0x0x51fe8000) file#: 0 rdba: 0x00000000 (0/0) class 0 ba: 0x0x51c00000
  set: 3 dbwrid: 0 obj: 0 objn: 0
  hash: [532996b0,532996b0] lru: [51fe8104,532bec4c]
  LRU flags: on_auxiliary_list
  ckptq: [NULL] fileq: [NULL]
  st: FREE md: NULL rsop: 0x(nil) tch: 0
BH (0x0x51be8814) file#: 0 rdba: 0x00000200 (0/512) class 0 ba: 0x0x51816000
  set: 3 dbwrid: 0 obj: 0 objn: 0
  hash: [532996b8,532996b8] lru: [51be8918,51be87a0]
  LRU flags: on_auxiliary_list
  ckptq: [NULL] fileq: [NULL]
  st: FREE md: NULL rsop: 0x(nil) tch: 0
BH (0x0x513f87a4) file#: 1 rdba: 0x00405cde (1/23774) class 1 ba: 0x0x512ce000
  set: 3 dbwrid: 0 obj: 8 objn: 14
  hash: [532996c0,532996c0] lru: [513f8730,513f88a8]
  LRU flags: moved_to_tail
  ckptq: [NULL] fileq: [NULL]
  st: XCURRENT md: NULL rsop: 0x(nil) tch: 0
  flags: only_sequential_access
  LRBA: [0x0.0.0] HSCN: [0xffff.ffffffff] HSUB: [255] RRBA: [0x0.0.0]

```

在 Oracle 10g 之前,数据库的等待事件中,所有 Latch 等待被归入 Latch Free 等待事件中,在 Statspack 的 report 中,如果在 Top 5 等待事件中看到 Latch Free 这一等待处于较高的位置,就需要我们介入进行研究和解决。

3. 热点块竞争与解决

这里通过一个生产环境的实际情况对热点块的问题进行分析和探讨。这是一个典型的性能低下的数据库,Top 5 等待事件都值得关注。可以注意到这个数据库中 Latch Free 是最严重的竞争。

Top 5 Wait Events			
~~~~~			
Event	Waits	Wait Time (cs)	% Total Wt Time
-----			
latch free	149,015	14,299,942	53.72
db file scattered read	1,781,670	2,793,591	10.49
buffer busy waits	45,001	2,386,174	8.96
log file switch (checkpoint incomplete)	19,527	1,991,844	7.48

enqueue	6,523	1,809,849	6.80
---------	-------	-----------	------

由于 Latch Free 是一个汇总等待事件，我们需要从 v\$latch 视图获得具体的 Latch 竞争主要是由哪些 Latch 引起的。在 Statspack report 中同样存在这样一部分数据：

Latch Sleep breakdown for DB: HHCIMS Instance: hhcims Snaps: 154 -174

-> ordered by misses desc

Latch Name	Get Requests	Misses	Spin & Sleeps	Sleeps 1->4
cache buffers chains	5,186,232,773	617,065	80,524	610224/6250/158/433/0
library cache	108,899,100	144,642	61,327	99020/31173/13558/891/0
row cache objects	83,115,714	46,477	1,610	44892/1565/17/3/0
shared pool	7,091,076	7,403	3,669	5426/420/1464/93/0
cache buffers lru chain	18,885,002	6,405	400	6024/369/9/3/0
enqueuees	41,504,627	1,559	110	1486/59/6/8/0
redo writing	370,920	1,524	178	1347/176/1/0/0
redo allocation	4,170,274	1,184	95	1098/78/7/1/0

注意到 Cache Buffers Chains 正是主要的 Latch 竞争。实际上，这个数据库在繁忙的时段，基本上处于停顿状态，大量进程等待 Latch Free 竞争，这些获得 session 的等待事件可以很容易地从 v\$session_wait 视图中查询得到：

SID	SEQ#	EVENT
4	14378	latch free
43	1854	latch free
176	977	latch free
187	4393	latch free
111	8715	latch free
209	48534	latch free

```

379      1008 latch free
455      1974 latch free
478     24713 latch free
388      444 latch free
369      855 latch free
264      567 enqueue
438      563 enqueue
355      563 enqueue
531      567 enqueue
513      819 enqueue
612      55 refresh controlfile command

```

如果需要具体确定热点对象，可以从 `v$latch_children` 中查询具体的子 Latch 信息，以下是一个生产环境中的部分信息摘录：

```

SQL> SELECT *
      2 FROM (SELECT   addr, child#, gets, misses, sleeps, immediate_gets igets,
      3               immediate_misses imiss, spin_gets sgets
      4               FROM v$latch_children
      5               WHERE NAME = 'cache buffers chains'
      6               ORDER BY sleeps DESC)
      7 WHERE ROWNUM < 11;

```

ADDR	CHILD#	GETS	MISSES	SLEEPS	IGETS	IMISS	SGETS
0000000406F24860	1093	1759617932	518883201	2587204	9765719	32224	0
0000000406F180E0	1081	509119236	13183658	623513	9720398	16398	0
0000000406F19180	1082	3264036800	7000899	336205	9669611	10077	0
0000000406E7F500	934	3252332119	40742230	218898	10945581	43083	0
00000004072A4A30	1757	1356574143	8959581	134591	7165568	17482	0
0000000406B6A4D0	175	1252888934	873697	103414	14699959	5887	0
0000000406F1A220	1083	2190392429	2054269	98862	9692515	5444	0
0000000406E462A0	879	840225741	4716843	79799	10789540	5743	0
0000000406BAAF70	237	2697876674	7349818	78180	8168375	3124	0
0000000406B9A470	221	3051410653	6924588	69161	9942446	4118	0

10 rows selected.

`X$BH` 中还存在另外一个关键字段 `HLADDR`，即 Hash Chain Latch Address，这个字段可以和 `v$latch_child.addr` 进行关联，这样就可以把具体的 Latch 竞争和数据块关联起来，再结

合 dba_extents 视图，就可以找到具体的热点竞争对象，找到具体热点竞争对象之后，可以结合 v\$sqlarea 或者 v\$sqltext，找到频繁操作这些对象的 SQL，然后对其进行优化，即可缓解或解决热点块竞争的问题。

通过以下查询可以实现以上思想，获取当前持有最热点数据块的 Latch 及 Buffer 信息：

```
SQL> SELECT b.addr, a.ts#, a.dbarfil, a.dbablk, a.tch, b.gets, b.misses, b.sleeps
2   FROM (SELECT *
3           FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch, hladdr
4                     FROM x$bh
5                     ORDER BY tch DESC)
6           WHERE ROWNUM < 11) a,
7   (SELECT addr, gets, misses, sleeps
8           FROM v$latch_children
9           WHERE NAME = 'cache buffers chains') b
10  WHERE a.hladdr = b.addr
11  /
```

ADDR	TS#	DBARFIL	DBABLK	TCH	GETS	MISSES	SLEEPS
0000000406AF3670	42	51	209612	216 2068740950	1396285	18734	
0000000406B14C70	42	51	209644	216 1840436663	1671667	20622	
0000000406B36270	42	51	209676	216 3447942770	2985525	29424	
0000000406B57870	42	51	209708	216 1902183007	1913809	18806	
0000000406B78E70	42	51	209740	216 2035257361	2314202	21304	
0000000406B9A470	42	51	209772	216 3061630199	6932017	69275	
0000000406E962C0	41	50	558579	217 723102299	32807	1112	
00000004071BDF70	41	50	190251	217 3409957883	657572	9196	
00000004071F9310	41	50	587887	217 1492232461	88013	4418	
000000040721A710	32	33	4703	224 756763990	69890	1470	

10 rows selected.

利用前面提到的 SQL，可以找到这些热点 Buffer 的对象信息：

```
SQL> SELECT distinct e.owner, e.segment_name, e.segment_type
2   FROM dba_extents e,
3   (SELECT *
4       FROM (SELECT   addr, ts#, file#, dbarfil, dbablk, tch
5                 FROM x$bh
6                 ORDER BY tch DESC)
```

```

7          WHERE ROWNUM < 11) b
8      WHERE e.relative_fno = b.dbarfil
9          AND e.block_id <= b.dbablk
10         AND e.block_id + e.blocks > b.dbablk;

OWNER          SEGMENT_NAME          SEGMENT_TYPE
-----
BOSSV2          HYUIDX_MOBILESEG          INDEX
BOSSV2          HY_PLATFORM              TABLE
BOSSV2          MAIDX_BATTASK_USERLIST_DATSEND INDEX PARTITION
BOSSV2          MAIDX_BATTASK_USERLIST_STATUS INDEX PARTITION
HSQUERY         HSOLAP_RPTJOB            TABLE

```

结合 v\$sqltext 或 v\$sqlarea，可以找到操作这些对象的相关 SQL，继续查询：

```

SQL> break on hash_value skip 1
SQL> SELECT /*+ rule */ hash_value,sql_text
2      FROM v$sqltext
3      WHERE (hash_value, address) IN (
4          SELECT a.hash_value, a.address
5          FROM v$sqltext a,
6              (SELECT DISTINCT a.owner, a.segment_name, a.segment_type
7                  FROM dba_extents a,
8                      (SELECT dbarfil, dbablk
9                          FROM (SELECT dbarfil, dbablk
10                              FROM x$bh
11                              ORDER BY tch DESC)
12                              WHERE ROWNUM < 11) b
13                      WHERE a.relative_fno = b.dbarfil
14                          AND a.block_id <= b.dbablk
15                          AND a.block_id + a.blocks > b.dbablk) b
16          WHERE a.sql_text LIKE '%' || b.segment_name || '%'
17          AND b.segment_type = 'TABLE')
18  ORDER BY hash_value, address, piece
19  /

HASH_VALUE SQL_TEXT
-----
175397557 select sum(snum) from HS_UNISMS_ORDERLOG_XUDING f where f.snum>:

```

```

1 and f.c_id=:2

180602532 select * from HS_UNISMS_PAYLOG_99DVD t where status=:1 and t.re
pterrorcode=9999 order by t.paylog_id

388500828 select * from HS_UNISMS_PAYLOG_99DVD t where status=:1 and t.re
pterrorcode=9999 and t.paytime >=to_date('2006-05-14 00:00:00',
'yyyy-mm-dd hh24:mi:ss') and t.paytime <=to_date('2006-05-16 00
:00:00','yyyy-mm-dd hh24:mi:ss') order by t.paylog_id

724740081 select HS_SCSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

773054905 select count(*) from HS_UNISMS_PAYLOG_99DVD t where t.paytime>=t
o_date(:1,'yyyy-mm-dd hh24:mi:ss') and t.paytime<=to_date(:2,'yy
yy-mm-dd hh24:mi:ss') and t.status=0

1071368535 select count(snum) from HS_UNISMS_ORDERLOG_XUDING f where f.snum
>:1 and f.c_id=:2

1109655340 insert into UM_PUT_M_ORDER values (UM_M_ORDER_SEQ.NEXTVAL,:1,:2,
:3,:4,sysdate,:5,:6,:7,:8,:9,:10,null,:11,:12,:13)

1451196467 select HS_ZJSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

1542951187 SELECT * FROM UM_PUT_M_ORDER

1669311552 insert into hm_user_info(USER_ID,PASSWORD,user_add2,user_seq_id)
values(:1,:2,:3,HM_USER_INFO_SEQ.NEXTVAL)

1669935485 insert into HS_UNISMS_PAYLOG_99DVD values (HS_UNISMS_PAYLOG_99DV
D_seq.NEXTVAL,:1,sysdate,:2,:3,:4,null,9999,9999,9999,9999,00000
0,null,null)

1697705163 insert into hm_user_info(USER_ID,M_PHONE,password,user_seq_id) v
alues(:1,:2,:3,HM_USER_INFO_SEQ.NEXTVAL)

1866895271 select * from UM_PUT_M_ORDER

```

```

1979706000 select mobile,c_id,status,snum from HS_UNISMS_ORDERLOG_XUDING t
              where t.status=:1

2152157475 select HS_UNISMS_ORDERLOG_SEQ.NEXTVAL from DUAL

2248706418 select * from UM_PUT_M_ORDER where MOBILEID=:1 and ORDERID=:2 or
              der by datetime desc

2336420675 select HS_JSSMS_ORDERLOG_SEQ.NEXTVAL from DUAL

2963798180 select * from HS_UNISMS_PAYLOG_99DVD t  where status=:1 and t.re
              pterrorcode=9999  and t.paytime <=to_date('2006-05-15 12:00:00',
              'yyyy-mm-dd hh24:mi:ss')  order by t.paylog_id

.....

83 rows selected.

```

找到这些 SQL 之后，剩下的问题就简单了，可以通过优化 SQL 减少数据的访问，避免或优化某些容易引起争用的操作（如 Connect By 等操作）来减少热点块竞争。

## 5.2 Shared Pool 的基本原理

Shared pool 是 Oracle SGA 设置中最复杂也是最重要的一部分内容，Oracle 通过 Shared Pool 来实现 SQL 共享、减少代码硬解析等，从而提高数据库的性能。在某些版本中，如果设置不当，Shared Pool 可能会极大地影响数据库的正常运行。

在 Oracle 7 之前，Shared Pool 并不存在，每个 Oracle 连接都有一个独立的 Server 进程与之相关联，Server 进程负责解析、优化所有 SQL 和 PL/SQL 代码。典型地，在 OLTP 环境中，很多代码具有相同或类似的结构，反复的独立解析浪费了大量的时间以及资源，Oracle 最终认识到这个问题，并且从 PL/SQL 开始尝试把这部分可共享的内容进行独立存储和管理，于是 Shared Pool 作为一个独立的 SGA 组件开始被引入，并且其功能和作用被逐渐完善和发展起来。

在这里注意到，Shared Pool 最初被引入的目的，也就是它的本质功能是**实现共享**。如果你的系统代码是完全异构的（假设你的代码从不绑定变量，从不反复执行），那么你会发现，这时候 Shared Pool 完全就成为了一个负担，它在徒劳无功地进行无谓的努力：**保存代码、执行计划等期待重用**，并且客户端要不停地获取 Latch，试图寻找共享代码，却始终一无所获。如果真是如此，那这是我们最不愿看到的情况，Shared Pool 变得有害无益。当然这是极端，可是在性能优化中我们发现，大多数性能低下的系统都存在这样的通病：**代码极少共享，缺乏或从不实行变量绑定**。优化这些系统的根本方法就是优化代码，使代码（在保证性能的前

提下)可以充分共享,减少无谓的反复硬/软解析。

实际上,Oracle 引入 Shared Pool 就是为了帮助用户实现代码的共享和重用。了解了这一点之后,开发人员在应用开发的过程中,也应该有意识的提高自己的代码水平,以期减少数据库的压力。这应该是对开发人员的最初和最基本的要求。

### 5.2.1 Shared Pool 的设置说明

Shared Pool 的大小通过初始化参数 `shared_pool_size` 设置。对于 Shared Pool 的设置,一直以来是最具有争议的一部分内容。一方面很多人建议可以把 Shared Pool 设置得稍大,以充分 Cache 代码和避免 ORA-04031 错误的出现;另一方面又有很多人建议不能把 Shared Pool 设置得过大,因为过大可能会带来管理上的额外负担,从而会影响数据库的性能。

至于哪一种说法更为准确,这个管理上的额外负担究竟指什么,这并不是几句话就能予以定论的,下面我通过一点内部分析,试图从内部原理上回答这个问题。

下面将用到 Shared Pool 的转储,所以首先需要介绍一下相关的命令。

可以通过如下命令转储 Shared Pool 共享内存的内容:

```
SQL> alter session set events 'immediate trace name heapdump level 2';
Session altered.
```

转储的内容被记录在一个 trace 文件中,这个 trace 文件可以在 `udmp` 目录下找到。为了比较 Oracle 9i 以及 Oracle 8i 的不同,本测试中了两个版本的 trace 文件。

#### (1) Oracle 9iR2:

```
SQL> @gettrcname

TRACE_FILE_NAME
-----
/opt/oracle/admin/hsjf/udump/hsjf_ora_24983.trc
```

#### (2) Oracle 8.1.5:

```
SQL> @gettrcname

TRACE_FILE_NAME
-----
/usr/oracle8/admin/guess/udump/guess_ora_22038.trc
```

### — 注 意 —

`alter session set events 'immediate trace name heapdump level 2'` 是一条内部命令,指定 Oracle 把 Shared Pool 的内存结构在 Level 2 级转储出来。

注意 `gettrcname.sql` 是我用来获取 trace 文件名称的一个脚本,代码如下:

```
SELECT      d.VALUE
           || '

```

```

    || LOWER (RTRIM (i.INSTANCE, CHR (0)))
    || '_ora_'
    || p.spid
    || '.trc' trace_file_name
FROM (SELECT p.spid
      FROM v$mystat m, v$session s, v$process p
      WHERE m.statistic# = 1 AND s.SID = m.SID AND p.addr = s.paddr) p,
(SELECT t.INSTANCE
      FROM v$thread t, v$parameter v
      WHERE v.NAME = 'thread'
      AND (v.VALUE = 0 OR t.thread# = TO_NUMBER (v.VALUE))) i,
(SELECT VALUE
      FROM v$parameter
      WHERE NAME = 'user_dump_dest') d
/
```

读者也可以在我的网站找到相关脚本及更多详细说明。

Shared Pool 通过 Free Lists 管理 Free 内存块 (Chunk)，Free 的内存块 (Chunk) 按不同 size 被划分到不同的部分 (Bucket) 进行管理。结合 Dump 文件，可以通过图 5-4 对 Shared Pool 的 Free List 管理进行说明。

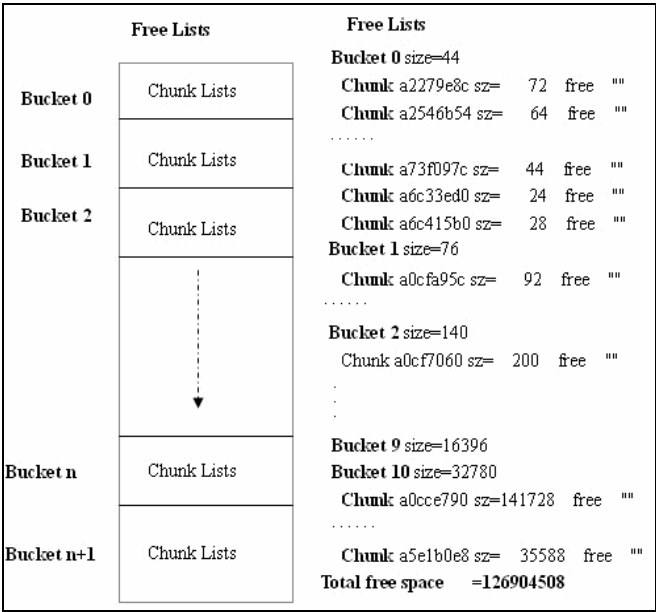


图 5-4 Shared Pool 自由列表

在 Oracle 8.1.5 中，不同 Bucket 管理的内存块的 size 范围如下所示 (size 显示的是下边界)：

```
SQL> select * from v$version ;
```

## BANNER

```
-----
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production
```

```
PL/SQL Release 8.1.5.0.0 - Production
```

```
CORE Version 8.1.3.0.0 - Production
```

```
TNS for Solaris: Version 8.1.5.0.0 - Production
```

```
NLSRTL Version 3.4.0.0.0 - Production
```

```
oracle:/usr/oracle8/admin/guess/udump>cat guess_ora_22038.trc|grep Bucket
```

```
Bucket 0 size=44
```

```
Bucket 1 size=76
```

```
Bucket 2 size=140
```

```
Bucket 3 size=268
```

```
Bucket 4 size=524
```

```
Bucket 5 size=1036
```

```
Bucket 6 size=2060
```

```
Bucket 7 size=4108
```

```
Bucket 8 size=8204
```

```
Bucket 9 size=16396
```

```
Bucket 10 size=32780
```

## — 注 意 —

本文所引用的所有 trace 文件都可以在我的网站 (<http://www.eygle.com>) 上找到。

注意观察这个输出结果，在这里，小于 76 bytes 的块都位于 Bucket 0 上；大于 32780 的块，都在 Bucket 10 上。中间的 Bucket 边界值遵循：

```
Bucket Size(N) = 2 * Bucket Size(N-1)
```

```
76    - 44   = 32
```

```
140   - 76   = 64
```

```
268   - 140  = 128
```

```
524   - 268  = 256
```

```
1036  - 524  = 512
```

```
2060  - 1036 = 1024
```

```
4108  - 2060 = 2048
```

```
8204  - 4108 = 4196
```

```
16396 - 8204 = 8192
```

```
32780 - 16396 = 16384
```

初始时，数据库启动以后，Shared Pool 多数是连续内存块。但是当空间分配使用以后，内存块开始被分割，碎片开始出现，Bucket 列表开始变长。

Oracle 请求 Shared Pool 空间时，首先进入相应的 Bucket 进行查找。如果找不到，则转向下一个非空的 Bucket，获取第一个 Chunk。分割这个 Chunk，剩余部分会进入相应的 Bucket，进一步增加碎片。

最终的结果是，由于不停分割，每个 Bucket 上的内存块会越来越多，越来越碎小。通常 Bucket 0 的问题会最为显著，在这个测试的小型数据库上，Bucket 0 上的碎片已经达到 9030 个，而 shared_pool_size 设置仅为 150MB。

通常如果每个 Bucket 上的 Chunk 多于 2000 个，就被认为是 Shared Pool 碎片过多。而 Shared Pool 的碎片过多，是 Shared Pool 产生性能问题的主要原因。

碎片过多会导致搜索 Free List 的时间过长，而 Free lists 的管理和搜索都需要获得和持有一个非常重要的 Latch，就是 Shared Pool Latch。Latch 是 Oracle 数据库内部提供的一种低级锁，通过串行机制保护共享内存不被并发更新/修改所损坏。Latch 的持有通常都非常短暂（通常微秒级），但是对于一个繁忙的数据库，这个串行机制往往会成为极大的性能瓶颈。关于 Latch 的机制这里不作过多介绍，那需要太多的篇幅。

继续前面的话题，如果 Free Lists 链表过长，搜索这个 Free Lists 的时间就会变长，从而可能导致 Shared Pool Latch 被长时间持有，在一个繁忙的系统中，这会引起严重的 Shared Pool Latch 竞争。在 Oracle 9i 之前，这个重要的 Shared Pool Latch 只有一个，所以长时间持有将会导致严重的性能问题。

而在大多数情况下，用户请求的都是相对小的内存块（Chunk），这样搜索 Bucket 0 往往消耗了大量的时间以及资源，Latch 的争用此时就会成为一个非常严重的问题。

从 Oracle 8.1.7 开始，Oracle 对 Shared Pool 的管理进行了改进，在 Oracle 8.1.7.4 中：

```
SQL> select * from v$version;
```

```
BANNER
```

```
-----  
Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
```

```
PL/SQL Release 8.1.7.4.0 - Production
```

```
CORE      8.1.7.0.0      Production
```

```
TNS for Solaris: Version 8.1.7.4.0 - Production
```

```
NLSRTL Version 3.4.1.0.0 - Production
```

新的 Bucket 划分为：

```
bash-2.03$ grep Bucket testora8_ora_13684.trc
```

```
Bucket 0 size=32
```

```
Bucket 1 size=40
```

```
Bucket 2 size=48
```

```
Bucket 3 size=56
```

```
Bucket 4 size=64
```

```

Bucket 5 size=72
.....

Bucket 198 size=1616
Bucket 199 size=1624
Bucket 200 size=1672
Bucket 201 size=1720
.....

Bucket 248 size=3976
Bucket 249 size=4024
Bucket 250 size=4120
Bucket 251 size=8216
Bucket 252 size=16408
Bucket 253 size=32792
Bucket 254 size=65560

```

可以看到，初始 Oracle 分配了 255 个 Bucket，Bucket 分配如下：

- 0~199 以 8 bytes 递增；
- 200~248 以 48 bytes 递增；
- 249~250 递增 96；
- 250~251 递增 4096；
- 251~252 递增 8192；
- 252~253 递增 16384；
- 253~254 递增 32768。

通过进一步细分 Bucket，Oracle 可以强化对于共享池的管理。

在 Oracle 9i 中，Oracle 进一步改写了 Shared Pool 管理的算法，来看一下 Oracle 9i 中的处理方式，首先记录一下测试环境：

```

SQL> select * from v$version;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.3.0 - Production
PL/SQL Release 9.2.0.3.0 - Production
CORE 9.2.0.3.0 Production
TNS for Linux: Version 9.2.0.3.0 - Production
NLSRTL Version 9.2.0.3.0 - Production

```

转储 Shared Pool：

```

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

```

```
Session altered.
```

```
SQL> @gettrcname
```

```
TRACE_FILE_NAME
```

```
-----  
/opt/oracle/admin/hsjf/udump/hsjf_ora_24983.trc
```

```
SQL>
```

```
SQL> !
```

```
[oracle@jumper oracle]$ cd $admin
```

```
[oracle@jumper udump]$ cat hsjf_ora_24983.trc|grep Bucket
```

```
Bucket 0 size=16
```

```
Bucket 1 size=20
```

```
Bucket 2 size=24
```

```
Bucket 3 size=28
```

```
Bucket 4 size=32
```

```
Bucket 5 size=36
```

```
Bucket 6 size=40
```

```
Bucket 7 size=44
```

```
Bucket 8 size=48
```

```
Bucket 9 size=52
```

```
Bucket 10 size=56
```

```
Bucket 11 size=60
```

```
.....<这里省略了部分内容>....
```

```
Bucket 235 size=3116
```

```
Bucket 236 size=3180
```

```
Bucket 237 size=3244
```

```
Bucket 238 size=3308
```

```
Bucket 239 size=3372
```

```
Bucket 240 size=3436
```

```
Bucket 241 size=3500
```

```
Bucket 242 size=3564
```

```
Bucket 243 size=3628
```

```
Bucket 244 size=3692
```

```
Bucket 245 size=3756
```

```
Bucket 246 size=3820
```

```

Bucket 247 size=3884
Bucket 248 size=3948
Bucket 249 size=4012
Bucket 250 size=4108
Bucket 251 size=8204
Bucket 252 size=16396
Bucket 253 size=32780
Bucket 254 size=65548

```

观察以上输出，在 Oracle 9i 中，Free Lists 被划分为 0~254，共 255 个 Bucket。

每个 Bucket 容纳的 size 范围可以进一步细分：

- Bucket 0~199 容纳 size 以 4 字节递增。
- Bucket 200~249 容纳 size 以 64 字节递增。

从 Bucket 249 开始，Oracle 各 Bucket 步长进一步增加。

- Bucket 249:  $4012 \sim 4107 = 96$
- Bucket 250:  $4108 \sim 8203 = 4096$
- Bucket 251:  $8204 \sim 16395 = 8192$
- Bucket 252:  $16396 \sim 32779 = 16384$
- Bucket 253:  $32780 \sim 65547 = 32768$
- Bucket 254:  $\geq 65548$

对比 Oracle 8i，在 Oracle 9i 中 Shared Pool 管理最为显著的变化就是，对于数量众多的 Chunk，Oracle 增加了更多的 Bucket 来管理（这与之前讲过的 Buffer Cache 的增强非常相似）。

0~199 共 200 个 Bucket，size 以 4 为步长递增；200~249 共 50 个 Bucket，size 以 64 递增。这样每个 Bucket 中容纳的 Chunk 数量大大减少，查找的效率得以提高。

所以，在 Oracle 9i 之前，如果盲目地增大 shared_pool_size 或设置过大的 shared_pool_size，往往会适得其反。这就是也许你曾经听过的“过大的 Shared_Pool 会带来管理上的负担”。

而且在 Oracle 9i 中，为了增加对于大共享池的支持，Shared Pool Latch 从原来的一个增加到现在的 7 个。如果用户的系统有 4 个或 4 个以上的 CPU，并且 shared_pool_size 大于 250MB，那么 Oracle 可以把 Shared Pool 分割为多个子缓冲池进行管理，每个 subpool 都拥有独立的结构、LRU 和 Shared Pool Latch。以下查询显示的就是这些 Latch：

```

SQL> select addr, name, gets, misses, spin_gets
      2  from    v$latch_children where name = 'shared pool';

```

ADDR	NAME	GETS	MISSES	SPIN_GETS
0000000380068F38	shared pool	0	0	0
0000000380068E40	shared pool	0	0	0
0000000380068D48	shared pool	0	0	0
0000000380068C50	shared pool	0	0	0

0000000380068B58 shared pool	0	0	0
0000000380068A60 shared pool	0	0	0
0000000380068968 shared pool	13808572	3089	3087

7 rows selected.

子缓冲的数量由一个新引入的隐含参数设置 **_KGHDSIDX_COUNT**。可以手工调整该参数（仅限于试验环境研究用），以观察共享池管理的变化：

```
SQL> alter system set "_kgghdsidx_count"=2 scope=sfile;
```

System altered.

```
SQL> startup force;
```

ORACLE instance started.

.....

```
SQL> col KSPINM for a20
```

```
SQL> col KSPSTVL for a20
```

```
SQL> select a.kspinm, b.kspstvl
```

```
2 from x$ksppi a, x$ksppsv b
```

```
3 where a.indx = b.indx and a.kspinm = '_kgghdsidx_count';
```

KSPINM

KSPSTVL

-----

_kgghdsidx_count

2

```
SQL> col name for a20
```

```
SQL> select addr, name, gets, misses, spin_gets
```

```
2 from v$latch_children where name = 'shared pool';
```

ADDR	NAME	GETS	MISSES	SPIN_GETS
50043078	shared pool	0	0	0
50042FB0	shared pool	0	0	0
50042EE8	shared pool	0	0	0
50042E20	shared pool	0	0	0
50042D58	shared pool	0	0	0
50042C90	shared pool	8166	0	0

```

50042BC8 shared pool                298          0          0

7 rows selected.

```

但是需要注意的是, 在 Oracle 9i 中, 这些新特性同时也带来了一些相应的 Bug, 跟 Shared Pool 多缓冲池相关的 Bug 有 3 316 003, 该 Bug 在 9205 中得到了修正, 读者可以参考 Metalink 上的相关链接。

通过这一系列的算法改进, Oracle 9i 中的 Shared Pool 管理得以增强, 较好地解决了大 Shared Pool 的性能问题; 在 Oracle 8i 中, 过大的 Shared Pool 设置可能带来的栓锁争用等性能问题在某种程度上得到了解决。

所以说, 如果是在 Oracle 9i 中, 设置较大的 Shared Pool 并不一定会给用户带来和 Oracle 8i 同样的麻烦。

在论坛上经常看到很多人对于 Shared_Pool 的建议一直就是 200~300MB, 而且一直认为这就是 Shared Pool 性能问题的关键, 实际上, 这是不确切的。

### 5.2.2 了解 X\$KSMSP 视图

Shared Pool 的空间分配和使用情况, 可以通过一个内部视图来观察, 这个视图就是 X\$KSMSP。X\$KSMSP 的名称含义为[K]ernal [S]torage [M]emory Management [S]GA Hea[P], 其中每一行都代表着 Shared Pool 中的一个 Chunk。

来看一下 X\$KSMSP 的结构:

```

SQL> desc x$ksmsp

```

Name	Null?	Type
ADDR		RAW(4)
INDX		NUMBER
INST_ID		NUMBER
KSMCHIDX		NUMBER
KSMCHDUR		NUMBER
KSMCHCOM		VARCHAR2(16)
KSMCHPTR		RAW(4)
KSMCHSIZ		NUMBER
KSMCHCLS		VARCHAR2(8)
KSMCHTYP		NUMBER
KSMCHPAR		RAW(4)

这里要关注以下几个字段。

- (1) x\$ksmsp.ksmchcom, 是注释字段, 每个内存块被分配以后, 注释会添加在该字段中。
- (2) x\$ksmsp.ksmchsiz, 代表块大小。
- (3) x\$ksmsp.ksmchcls, 列代表类型, 主要有 4 类, 说明如下:

### ■ free

**Free Chunks:** 不包含任何对象的 Chunk，可以不受限制的被自由分配。

### ■ recreate

**Recreatable Chunks:** 包含可以被临时移出内存的对象，在需要的时候，这个对象可以被重新创建。例如，许多存储共享 SQL 代码的内存都是可以重建的。

### ■ freeable

**Freeable Chunks:** 包含 session 周期或调用的对象，随后可以被释放。这部分内存有时候可以全部或部分提前释放。但是注意，由于某些对象是中间过程产生的，这些对象不能临时被移出内存（因为不可重建）。

### ■ perm

**Permanent Memory Chunks:** 包含永久对象，通常不能独立释放。

从以上引用的 trace 文件中，摘出开头一段，可以清楚地看到 Oracle 对这部分 Chunk 的记录情况：

```
HEAP DUMP heap name="sga heap" desc=0x80000030
extent sz=0xfc4 alt=48 het=32767 rec=9 flg=2 opc=0
parent=0 owner=0 nex=0 xsz=0x1
EXTENT 0
  Chunk a7412000 sz= 23801020 perm "perm" alo=23801020
  Chunk a8ac4cbc sz= 68 free " "
  Chunk a8ac4d00 sz= 560 freeable "library cache" ds=a3e4fd24
  Chunk a8ac4f30 sz= 588 freeable "sql area" ds=a6bcc328
  Chunk a8ac517c sz= 448 freeable "library cache" ds=a57b6a38
  Chunk a8ac533c sz= 1072 freeable "partitioning d" ds=a4002688
  Chunk a8ac576c sz= 2036 freeable "library cache" ds=a6c29e3c
  Chunk a8ac5f60 sz= 560 freeable "library cache" ds=a2decda8
  Chunk a8ac6190 sz= 96 freeable "library cache"
  Chunk a8ac61f0 sz= 20 free " "
  Chunk a8ac6204 sz= 176 recreate "KGL handles" latch=0
  Chunk a8ac62b4 sz= 560 recreate "library cache" latch=8000cc38
  ds a6c33f54 sz= 1680 ct= 3
  a400a0dc sz= 560
  a24aee88 sz= 560
```

可以通过查询 X\$KSMSP 视图来考察 Shared Pool 中存在的内存片的数量。不过要注意，Oracle 的某些版本（如 10.1.0.2）在某些平台上（如 HP-UX PA-RISC 64-bit）查询该视图时可能会导致过度的 CPU 耗用，这是由 Bug 引起的。

下面来进行测试，在这个测试数据库中，初始启动数据库时，X\$KSMSP 中存在 2259 个 Chunk：

```
SQL> select count(*) from x$ksmsp;
```

```

COUNT(*)
-----
      2259

```

执行查询：

```
SQL> select count(*) from dba_objects;
```

```

COUNT(*)
-----
     10491

```

此时 Shared Pool 中的 Chunk 数量增加：

```
SQL> select count(*) from x$ksmsp;
```

```

COUNT(*)
-----
      2358

```

这就是由于 Shared Pool 中进行 SQL 解析，请求空间，进而导致请求 Free 空间分配、分割，从而产生了更多、更细碎的内存 Chunk。

由此可以看出，如果数据库系统中存在大量的硬解析，不停请求分配 Free 的 Shared Pool 内存，除了必须的 Shared Pool Latch 等竞争外，还不可避免地会导致 Shared Pool 中产生更多的内存碎片（当然在内存回收时，可能看到 Chunk 数量减少的情况）。

进行以下测试，首先重新启动数据库：

```
SQL> startup force;
```

```
ORACLE instance started.
```

```

Total System Global Area  47256168 bytes
Fixed Size                  451176 bytes
Variable Size              29360128 bytes
Database Buffers           16777216 bytes
Redo Buffers                667648 bytes
Database mounted.
Database opened.

```

创建一张临时表用以保存之前 X\$KSMSMP 的状态：

```

SQL> CREATE GLOBAL TEMPORARY TABLE e$ksmsp ON COMMIT PRESERVE ROWS AS
  2  SELECT      a.ksmchcom,
  3              SUM (a.CHUNK) CHUNK,
  4              SUM (a.recr) recr,

```

```

5          SUM (a.freeabl) freeabl,
6          SUM (a.SUM) SUM
7      FROM (SELECT    ksmchcom, COUNT (ksmchcom) CHUNK,
8                     DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
9                     DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
10                    SUM (ksmchsiz) SUM
11          FROM x$ksmsp GROUP BY ksmchcom, ksmchcls) a
12  where 1 = 0
13  GROUP BY a.ksmchcom;

```

Table created.

保存当前 Shared Pool 状态:

```

SQL> INSERT INTO E$KSMSP
2  SELECT    a.ksmchcom,
3          SUM (a.CHUNK) CHUNK,
4          SUM (a.recr) recr,
5          SUM (a.freeabl) freeabl,
6          SUM (a.SUM) SUM
7      FROM (SELECT    ksmchcom, COUNT (ksmchcom) CHUNK,
8                     DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
9                     DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
10                    SUM (ksmchsiz) SUM
11          FROM x$ksmsp
12          GROUP BY ksmchcom, ksmchcls) a
13  GROUP BY a.ksmchcom
14  /

```

41 rows created.

执行查询:

```
SQL> select count(*) from dba_objects;
```

```

COUNT(*)
-----
10492

```

比较前后 Shared Pool 内存分配的变化:

```
SQL> select a.ksmchcom,a.chunk,a.sum,b.chunk,b.sum,(a.chunk - b.chunk) c_diff,(a.sum -b.sum) s_diff
```

```

2  from
3  (SELECT    a.ksmchcom,
4             SUM (a.CHUNK) CHUNK,
5             SUM (a.recr) recr,
6             SUM (a.freeabl) freeabl,
7             SUM (a.SUM) SUM
8  FROM (SELECT    ksmchcom, COUNT (ksmchcom) CHUNK,
9             DECODE (ksmchcls, 'recr', SUM (ksmchsiz), NULL) recr,
10            DECODE (ksmchcls, 'freeabl', SUM (ksmchsiz), NULL) freeabl,
11            SUM (ksmchsiz) SUM
12      FROM x$ksmsp
13      GROUP BY ksmchcom, ksmchcls) a
14 GROUP BY a.ksmchcom) a,e$ksmsp b
15 where a.ksmchcom = b.ksmchcom and (a.chunk - b.chunk) <>0
16 /

```

KSMCHCOM	CHUNK	SUM	CHUNK	SUM	C_DIFF	S_DIFF
KGL handles	313	102080	302	98416	11	3664
KGLS heap	274	365752	270	360424	4	5328
KQR PO	389	198548	377	192580	12	5968
free memory	93	2292076	90	2381304	3	-89228
library cache	1005	398284	965	381416	40	16868
sql area	287	547452	269	490052	18	57400

6 rows selected.

简单分析一下以上结果, 首先 free memory 的大小减少了 89228 (增加到另外 5 个组件中), 这说明 SQL 解析存储占用了一定的内存空间, 而 Chunk 数从 90 增加到 93, 这说明内存碎片增加了, 而碎片增加是共享池性能下降的开始。

### 5.2.3 诊断和解决 ORA-04031 错误

关于 ORA-04031 错误, 网络上相关的参考文档已经很多, 我在这里只做简要说明。Shared Pool 的根本问题只有一个, 就是碎片过多带来的性能影响。前面已经讲到了 Oracle 8i 和 Oracle 9i 在 Shared Pool 管理上的不同, 下面将对如何诊断和解决 ORA-04031 错误进行一些分析和讨论。

#### 1. 什么是 ORA-04031 错误

当尝试在共享池分配大块的连续内存失败 (很多时候是由于碎片过多, 而并非真是内存

不足) 时, Oracle 首先清除共享池中当前没使用的所有对象, 使空闲内存块合并。如果仍然没有足够大的单块内存可以满足需要, 就会产生 ORA-04031 错误。

Shared Pool 的内存分配算法相当复杂, ORA-04031 错误出现的原因也众多, 经过简化, 可以通过以下一段伪代码来描述 ORA-04031 错误的产生:

```

Scan free lists                --扫描 Free Lists
if (request size of RESERVED Pool size) --如果请求 RESERVED POOL 空间
    scan reserved list         --扫描保留列表
if (chunk found)               --如果发现满足条件的内存块
    check chunk size and perhaps truncate --检查大小, 可能需要分割
    return                     --返回
do LRU operation for n objects --如果并非请求 RESERVED POOL 或不能发现足够内存
    scan free lists            --则转而执行 LRU 操作, 释放内存, 重新扫描
    if (request sizes exceeds reserved pool min alloc) --如果请求大于 _shared_pool_reserved_min_alloc
        scan reserved list     --扫描保留列表
    if (chunk found)           --如果发现满足条件的内存块
        check chunk size and perhaps truncate --检查大小, 可能需要分割
        return                 --在 Freelist 或 reserved list 找到则成功返回

signal ORA-4031 error          --否则报告 ORA-04031 错误。

```

Oracle 关于 ORA-04031 错误的解释及建议如下:

```

04031, 00000, "unable to allocate %s bytes of shared memory (%s,%s,%s,%s)"
// *Cause: More shared memory is needed than was allocated in the shared
// pool.
// *Action: If the shared pool is out of memory, either use the
// dbms_shared_pool package to pin large packages,
// reduce your use of shared memory, or increase the amount of
// available shared memory by increasing the value of the
// INIT.ORA parameters "shared_pool_reserved_size" and
// "shared_pool_size".
// If the large pool is out of memory, increase the INIT.ORA
// parameter "large_pool_size".

```

## 2. 内存泄漏

在 Oracle 9iR2 之前的很多 ORA-04031 的错误都和内存泄漏的 Bug 有关, 所以是否及时应用相关 Patch 是非常重要的。

在 Oracle 8.1.7 中, 几乎每个人都曾经遇到 ORA-04031 的问题, 这同样是因为 Bug 导致的 (也就是在下面列表中可以看到的 Bug: 1397603)。

表 5-1 是 Oracle 发布的不同版本会导致 ORA-04031 错误的 Bug 列表，特摘录在此供大家参考。如果数据库经常出现这个错误（首先需要确认，shared_pool_size 不是设置得非常小），用户就应该确认是否符合下列 Bug 的特征：

**表 5-1** 导致 ORA-04031 错误的 Bug 列表

BUG 号	描 述	解 决 方 法	修正版本
<Bug:1397603>	ORA-4031/SGA memory leak of PERMANENT memory occurs for buffer handles	_db_handles_cached = 0	901/8172
<Bug:1640583>	ORA-4031 due to leak / cache buffer chain contention from AND-EQUAL access	Not available	8171/901
<Bug:1318267>	INSERT AS SELECT statements may not be shared when they should be if TIMED_STATISTICS. It can lead to ORA-4031	_SQLEXEC_PROGRESSION _COST=0	8171/8200
<Bug:1193003>	Cursors may not be shared in 8.1 when they should be	Not available	8162/8170/ 901
<Bug:2104071>	ORA-4031/excessive "miscellaneous" shared pool usage possible (many pins)	None-> This is known to affect the XML parser	8174/9013/9201
<Note:263791.1>	Several number of BUGs related to ORA-4031 erros were fixed in the 9.2.0.5 patchset	Not available	9205

### 3. 绑定变量和 cursor_sharing

如果 shared_pool_size 设置得足够大，又可以排除 Bug 的因素，那么大多数的 ORA-04031 错误都是由共享池中的大量 SQL 代码等导致了过多的内存碎片而引起的，可能的主要原因有：

- （1）SQL 没有足够的共享。
- （2）大量不必要的解析调用。
- （3）没有使用绑定变量。

实际上，应用的编写和调整始终是最重要的内容，Shared Pool 的调整根本上要从应用入手。使用绑定变量可以充分降低 Shared Pool 和 Library Cache 的 Latch 竞争，从而提高性能。

如果用户的应用没有很好地使用绑定变量，那么 Oracle 从 8.1.6 开始提供了一个新的初始化参数用以在 Server 端进行强制变量绑定，这个参数是：cursor_sharing。

最初这个参数有两个可选设置：exact 和 force。缺省的是 exact，表示精确匹配；force 表示在 Server 端执行强制绑定。在 8i 的版本里使用这个参数对某些应用可以带来极大的性能提高，但是同时也存在一些副作用，比如优化器无法生成精确的执行计划，SQL 执行计划发生改变等（所以如果启用 cursor_sharing 参数时，一定要确认应用在此模式下经过充分的测试）。

从 Oracle 9i 开始, Oracle 引入了绑定变量 Peeking 的机制, SQL 在第一次执行时, 首先在 session 的 PGA 中使用具体值生成精确的执行计划, 以期提高执行计划的准确性。然而 Peeking 的方式只在第一次硬解析时生效, 所以仍然可能存在问题, 导致后续的 SQL 错误的执行。关于 Oracle 9i 绑定变量的 Peeking 在此不作过多论述, 有兴趣的朋友可以参考以下文章: <http://www.eygle.com/sql/Peeking.of.User-Defined.Bind.Variables.htm>

同时, 在 Oracle 9i 中, cursor_sharing 参数有了第三个选项: similar。该参数指定 Oracle 在存在柱状图信息时, 对于不同的变量值重新解析, 从而可以利用柱状图更为精确地制定 SQL 执行计划。即当存在柱状图信息时, similar 的表现和 exact 相同; 当柱状图信息不存在时, similar 的表现和 force 相同。关于这个内容 bit_rainy 网友曾经有过精彩的论述, 具体可以参考他的个人 Blog (<http://blog.itpub.net/post/330/1648>)。

但是需要注意的是, 在某些版本(如 Oracle9.2.0.5)中, 设置 cursor_sharing 为 similar, 可能导致 SQL 的 version_count 过高的 Bug。

#### 4. 使用 Flush Shared Pool 缓解共享池问题

前面提到, 本质上 ORA-04031 错误多数是由于 SQL 编写不当引起, 所以如果能够修改应用绑定变量是最好的解决之道。当然如果不能修改应用, 或者不能强制变量绑定, 那么 Oracle 还可以提供一种应急处理方法, 强制刷新共享池。

```
alter system flush shared_pool;
```

刷新共享池可以帮助合并碎片 (Small Chunks), 强制老化 SQL, 释放共享池, 但是这通常是不推荐的做法, 因为:

(1) Flush Shared Pool 会导致当前未使用的 cursor 被清除出共享池, 如果这些 SQL 随后需要执行, 那么数据库将经历大量的硬解析, 系统将会经历严重的 CPU 争用, 数据库将会产生激烈的 latch 竞争。

(2) 如果应用没有使用绑定变量, 大量类似的 SQL 不停执行, 那么 Flush Shared Pool 可能只能带来短暂的改善, 数据库很快就会回到原来的状态。

(3) 如果 Shared Pool 很大, 并且系统非常繁忙, 刷新 Shared Pool 可能会导致系统挂起, 对于类似系统尽量在系统空闲时进行。

从 Oracle 9i 开始, Oracle 的共享池算法发生了改变, Flush Shared Pool 的方法已经不再推荐使用。

#### 5. SHARED_POOL_RESERVED_SIZE 参数的设置及作用

还有一个参数是需要提及的: shared_pool_reserved_size。该参数指定了保留的共享池空间, 用于满足大的连续的共享池空间请求。

当共享池出现过多碎片, 请求大块空间会导致 Oracle 大范围地查找并释放共享池内存来满足请求, 由此可能会带来较为严重的性能下降, 设置合适的 shared_pool_reserved_size 参数, 结合 shared_pool_reserved_min_alloc 参数可以避免由此导致的性能下降。

这个参数理想值应该大到足以满足任何对 RESERVED LIST 的内存请求, 而无需数据库从共享池中刷新对象。这个参数的缺省值是 shared_pool_size 的 5%, 通常这个参数的建议值为 shared_pool_size 参数的 10%~20% 大小, 最大不得超过 shared_pool_size 的 50%。

同样的，在 trace 文件中，可以找到关于保留列表（RESERVED LIST）的内存信息：

RESERVED FREE LIST:

Chunk a6c6d778 sz= 7864320 R-free " "

Total reserved free space = 7864320

shared_pool_reserved_min_alloc 这个参数的值控制保留内存的使用和分配。如果在共享池空闲列表中请求一个足够尺寸的大块内存，但没有找到合适的空间，内存就从保留列表（RESERVED LIST）中分配一块比这个参数值大的空间。

在 Oracle 9i 中，该参数的缺省值是 4400，这是一个隐含参数，可以使用如下脚本查询其初始值：

```
SQL> select
  2   x.ksppinm name,
  3   y.ksppstvl value,
  4   y.ksppstdf isdefault,
  5   decode(bitand(y.ksppstvf,7),1,'MODIFIED',4,'SYSTEM_MOD','FALSE') ismod,
  6   decode(bitand(y.ksppstvf,2),2,'TRUE','FALSE') isadj
  7 from
  8   sys.x$ksppi x,
  9   sys.x$ksppcv y
 10 where
 11   x.inst_id = userenv('Instance') and
 12   y.inst_id = userenv('Instance') and
 13   x.indx = y.indx and
 14   x.ksppinm like '%_&par%'
 15 order by
 16   translate(x.ksppinm, ' _', '')
 17 /
```

Enter value for par: shared_pool_reserved_min_alloc

old 14: x.ksppinm like '%_&par%'

new 14: x.ksppinm like '%_shared_pool_reserved_min_alloc%'

NAME	VALUE	ISDEFAULT	ISMOD	ISADJ
-----				
_shared_pool_reserved_min_alloc	4400	TRUE	FALSE	FALSE

### 注 意

这段代码可以用于查询所有的隐含参数。

这个参数默认的值对于大多数系统来说都足够了。如果系统经常出现的 ORA-04031

错误都是请求大于 4400byte 的内存块，那么就可能需要增加 `shared_pool_reserved_size` 参数设置。

如果主要的引发 LRU 合并、老化并出现 ORA-04031 错误的内存请求在 4100~4400 byte 之间，那么降低 `shared_pool_reserved_min_alloc` 同时适当增大 `shared_pool_reserved_size` 参数值通常会有所帮助。设置 `shared_pool_reserved_min_alloc=4100` 可以增加 Shared Pool 成功满足请求的概率。

需要注意的是，这个参数的修改应当结合 Shared Pool Size 和 Shared Pool Reserved Size 的大小。

设置 `shared_pool_reserved_min_alloc=4100` 是经过证明的可靠方式，不建议设置更低。

查询 `v$shared_pool_reserved` 视图可以判断共享池问题的引发原因：

```
SQL> SELECT free_space, avg_free_size, used_space,
2  avg_used_size, request_failures, last_failure_size
3  FROM v$shared_pool_reserved;
```

FREE_SPACE	AVG_FREE_SIZE	USED_SPACE	AVG_USED_SIZE	REQUEST_FAILURES	LAST_FAILURE_SIZE
1966564	24278.5679	2062076	25457.7284	0	0

如果 `request_failures > 0` 且 `last_failure_size > shared_pool_reserved_min_alloc`，那么 ORA-04031 错误就可能是因为共享池保留空间缺少连续空间所致。要解决这个问题，可以考虑加大 `shared_pool_reserved_min_alloc` 来降低缓冲进共享池保留空间的对象数目，并增大 `shared_pool_reserved_size` 和 `shared_pool_size` 来加大共享池保留空间的可用内存。

如果 `request_failures > 0` 且 `last_failure_size < shared_pool_reserved_min_alloc`，或者 `request_failures = 0` 且 `last_failure_size < shared_pool_reserved_min_alloc`，那么是由于在库高速缓冲缺少连续空间而导致了 ORA-04031 错误。

对于这一类情况应该考虑降低 `shared_pool_reserved_min_alloc`，以放入更多的对象到共享池保留空间中并加大 `shared_pool_size`。

## 6. 其他

此外，某些特定的 SQL，较大的指针或者大的 Package 都可能导致 ORA-04031 错误。在很多 ERP 软件中，这样的情况非常常见。在这种情况下，可以考虑把这个大的对象 pin 到共享池中，减少其动态请求、分配所带来的负担。

使用 `DBMS_SHARED_POOL.KEEP` 系统包可以把这些对象 pin 到内存中，`SYS.STANDARD`、`SYS.DBMS_STANDARD` 等都是常见的候选对象。

## —— 注 意 ——

要使用 `DBMS_SHARED_POOL` 系统包，首先需要运行 `dbmspool.sql` 脚本，该脚本会自动调用 `prvtpool.plb` 脚本创建所需对象。

引发 ORA-04031 错误的因素还有很多，通过设置相关参数如 `session_cached_cursors`、

cursor_space_for_time 等也可以解决一些性能问题并带来针对性的性能改善,本章不再对此详细讨论。

## 7. 模拟 ORA-04031 错误

Oracle 9iR2 开始引入了段级统计信息 (Segment Statistics) 收集的新特性,其中一个新引入的视图是 v\$segstat, 查询该视图会引发 Shared Pool 的内存泄露 (在 9201~9206 版本中都存在此问题, 本测试案例来自 Windows 平台 9206), 可以利用这一问题来模拟 ORA-00431 错误。

以下是一段测试代码:

```
set heading off
column what format a40
column value format a30

select 'db instance' what, user || '@' || global_name value from global_name
UNION
select '# rows in v$segstat', to_char(count(*)) from v$segstat;

set linesize 200
set time on
set serveroutput on size 300000

declare
    l_temp          char(1);
    l_before        number;
    l_after         number := 0;
    l_loop_times    pls_integer := 1000;    -- try 1000
    l_sleep         number    := 0.00;    -- makes no difference

    cursor c_seg is select * from v$segstat;
    r_seg  c_seg%ROWTYPE;

    function get_mem return number is
        cursor c_mem is select bytes from v$sstat
            where name = 'free memory' and pool = 'shared pool';
        r_mem  c_mem%ROWTYPE;
    begin
        open c_mem; fetch c_mem into r_mem; close c_mem;
```

```

        return r_mem.bytes;
    end get_mem;

begin
    l_after := get_mem();

    for x in 1..l_loop_times loop
        l_before := l_after;

        OPEN c_seg; FETCH c_seg INTO r_seg; CLOSE c_seg;

        l_after := get_mem();
        dbms_output.put_line ('Loop ' || x || ': (' ||
            to_char(sysdate,'hh24:mi:ss') || ') from ' ||
            to_char(l_before,'999,999,999') || ' to ' ||
            to_char(l_after,'999,999,999') || ' (loss of ' ||
            to_char((l_after-l_before),'9,999,999') || ')');
        dbms_lock.sleep(l_sleep);
    end loop;
end;
/

```

首先来看看之前的状态（测试环境，已经把 Shared Pool 调整降低）：

```

SQL> select * from v$sgastat
      2  where name in('miscellaneous','free memory') and pool='shared pool';

```

POOL	NAME	BYTES
shared pool free memory		<b>3927884</b>
shared pool miscellaneous		5752896

执行以上代码：

```

20:49:57 SQL> @d:\mem.leak.sql

```

# rows in v\$segstat	1034
db instance	SYS@EYGLE
Loop 1: (20:50:00) from	782,072 to 769,336 (loss of 12,736)
Loop 2: (20:50:00) from	769,336 to 949,276 (loss of -179,940)

```

Loop 3: (20:50:00) from      949,276 to      978,872 (loss of      -29,596)
Loop 4: (20:50:00) from      978,872 to      970,436 (loss of       8,436)
Loop 5: (20:50:00) from      970,436 to      962,012 (loss of       8,424)
Loop 6: (20:50:00) from      962,012 to      949,364 (loss of      12,648)
Loop 7: (20:50:00) from      949,364 to      946,280 (loss of       3,084)
Loop 8: (20:50:00) from      946,280 to      993,064 (loss of     -46,784)
Loop 9: (20:50:00) from      993,064 to      984,640 (loss of       8,424)
Loop 10: (20:50:00) from      984,640 to     1,092,628 (loss of    -107,988)

declare
*

ERROR at line 1:

ORA-04031: unable to allocate 4212 bytes of shared memory ("shared pool","unknown object","sga
heap(1,0)","obj stat memor")

ORA-06512: at line 26

20:50:10 SQL> alter session set events 'immediate trace name heapdump level 2';

Session altered.

20:50:20 SQL> select * from v$sghostat

20:50:20      2   where name in('miscellaneous','free memory') and pool='shared pool';

shared pool free memory              888268
shared pool miscellaneous            10144656

```

转储共享内存，可以看到：

```

.....

Bucket 248 size=3948
  Chunk 7b69b5a0 sz=    3956    free    "          "
  Chunk 7b6d16b4 sz=    3956    free    "          "
Bucket 249 size=4012
  Chunk 7b5da178 sz=    4064    free    "          "
  Chunk 7b66360c sz=    4060    free    "          "
  Chunk 7b554204 sz=    4060    free    "          "
  Chunk 7b6e2af4 sz=    4060    free    "          "
  Chunk 7b76b8b4 sz=    4052    free    "          "
Bucket 250 size=4108
Bucket 251 size=8204
Bucket 252 size=16396

```

```

Bucket 253 size=32780
Bucket 254 size=65548
Total free space    =    180396

```

当前最大的 Chunk Size 是 4052，所以请求 4212 时出现了 ORA-04031 错误。如果系统的 ORA-04031 错误通常都是在 4200 左右出现，如前文提到的，可以通过修改 `_shared_pool_reserved_min_alloc` 参数设置来减少 ORA-04031 错误的出现，下面来比较一下。

(1) 缺省情况下，`_shared_pool_reserved_min_alloc = 4400`，在 ORA-04031 错误情况下，看一下保留池的使用：

#### RESERVED FREE LIST:

```

Chunk 7a000038 sz=    85940  R-free    "          "
Chunk 7a400038 sz=    85940  R-free    "          "
Chunk 7a800038 sz=    85940  R-free    "          "
Chunk 7ac00038 sz=    85940  R-free    "          "
Chunk 7b000038 sz=    85940  R-free    "          "
Chunk 7b400038 sz=    85940  R-free    "          "
Total reserved free space    =    515640

```

保留池保留了 85940 的 Chunk Size 未被使用。

(2) 修改 `_shared_pool_reserved_min_alloc = 4100`，在 ORA-04031 错误情况下，来看一下保留池的使用，修改方式如下（修改参数后需要重新启动数据库）：

```
21:01:43 SQL> alter system set "_shared_pool_reserved_min_alloc"=4100 scope=spfile;
```

System altered.

修改后保留池的使用情况：

#### RESERVED FREE LIST:

```

Chunk 7b414994 sz=     1624  R-free    "          "
Chunk 7b014994 sz=     1624  R-free    "          "
Chunk 7ac14988 sz=     1636  R-free    "          "
Chunk 7a814994 sz=     1624  R-free    "          "
Chunk 7a414988 sz=     1636  R-free    "          "
Chunk 7a014994 sz=     1624  R-free    "          "
Total reserved free space    =       9768

```

可以看到，在修改了 `_shared_pool_reserved_min_alloc` 参数以后，保留池的使用更为充分，从而使得 ORA-04031 错误的出现得以延迟。

### — 提 示 —

本例提供一种方法模拟 ORA-04031 错误，读者可以在测试环境中模拟和研究 ORA-04031 问题，但是严禁在生产环境中使用。

### 5.2.4 Library Cache Pin 及 Library Cache Lock 分析

Oracle 使用两种数据结构来进行 Shared Pool 的并发访问控制：lock 和 pin。lock 比 pin 具有更高的级别。

lock 在 handle 上获得，在 pin 一个对象之前，必须首先获得该 handle 的锁定。锁定主要有 3 种模式：Null、Share 和 Exclusive。在读取访问对象时，通常需要获取 Null（空）模式以及 Share（共享）模式的锁定。在修改对象时，需要获得 Exclusive（排他）锁定。

在锁定了 Library Cache 对象以后，一个进程在访问之前必须 pin 该对象。同样 pin 有 3 种模式：Null、Shared 和 Exclusive。只读模式时获得共享 pin，修改模式获得排他 pin。

通常访问、执行过程和 Package 时，获得的都是共享 pin，如果排他 pin 被持有，那么数据库此时就要产生等待。

在很多 Statspack 的 report 中，可能看到以下等待事件：

Top 5 Wait Events			
~~~~~			
Event	Waits	Wait Time (cs)	% Total Wt Time
library cache lock	75,884	1,409,500	48.44
latch free	34,297,906	1,205,636	41.43
library cache pin	563	142,491	4.90
db file scattered read	146,283	75,871	2.61
enqueue	2,211	13,003	.45

这里的 Library Cache Lock 和 Library Cache Pin 都是用户关心的，接下来就研究一下这几个等待事件。

1. LIBRARY CACHE PIN 等待事件

Oracle 文档上这样介绍这个等待事件：Library Cache Pin 是用来管理 Library Cache 的并发访问的，pin 一个 Object 会引起相应的 heap 被载入内存中（如果此前没有被加载），pins 可以在三个模式下获得：Null、Share 和 Exclusive，可以认为 pin 是一种特定形式的锁。

当 Library Cache Pin 等待事件出现时，通常说明该 pin 被其他用户以非兼容模式持有。

Library Cache Pin 的等待时间为 3 秒钟，其中有 1 秒钟用于 PMON 后台进程，即在取得 pin 之前最多等待 3 秒钟，否则就超时。

Library Cache Pin 的参数如下，有用的主要是 P1 和 P2。

- P1——KGL Handle address
- P2——Pin address
- P3——Encoded Mode & Namespace

Library Cache Pin 通常是发生在编译或重新编译 PL/SQL、VIEW、TYPES 等 Object 时，编译通常都是显性的，如安装应用程序、升级、安装补丁程序等，另外，alter、grant 和 revoke

等操作也会使 Object 变得无效，可以通过 Object 的 LAST_DDL_TIME 观察这些变化。

当 Object 变得无效时，Oracle 会在第一次访问此 Object 时试图去重新编译它，如果此时其他 session 已经把此 Object pin 到 Library Cache 中，就会出现问题，特别是当有大量的活动 session 并且存在较复杂的 dependence 时。在某种情况下，重新编译 Object 可能会花几个小时时间，从而阻塞其他试图访问此 Object 的进程。

下面通过一个例子来模拟及解释这个等待。

(1) 创建测试用存储过程。

```
SQL> create or replace PROCEDURE pining
  2  IS
  3  BEGIN
  4      NULL;
  5  END;
  6  /
```

Procedure created.

SQL>

```
SQL> create or replace procedure calling
  2  is
  3  begin
  4      pining;
  5      dbms_lock.sleep(3000);
  6  end;
  7  /
```

Procedure created.

(2) 模拟竞争。

首先执行 calling 过程，在 calling 过程中调用 pining 过程。此时 pining 过程上获得共享 pin，如果此时尝试对 pining 进行授权或重新编译，将产生 Library Cache Pin 等待，直到 calling 执行完毕。

session 1:

```
SQL> exec calling
```

此时 calling 开始执行。

session 2:

```
SQL> grant execute on pining to eygle;
```

此时 session 2 挂起。下面开始对此进行分析和研究。

从 v\$session_wait 入手，可以得到哪些 session 正在经历 Library Cache Pin 的等待。

```
SQL> select sid,seq#,event,p1,p1raw,p2,p2raw,p3,p3raw,state
2  from v$session_wait where event like 'library%';

SID          SEQ#  EVENT                                P1 P1RAW      P2 P2RAW      P3
WAIT_TIME SECONDS_IN_WAIT STATE
-----
8            268 library cache pin      1389785868 52D6730C 1387439312 52B2A4D0 301
2 WAITING
```

等待 3 秒就超时，SEQ#会发生变化。

```
SQL>

SID          SEQ#  EVENT                                P1 P1RAW      P2 P2RAW      P3
WAIT_TIME SECONDS_IN_WAIT STATE
-----
8            269 library cache pin      1389785868 52D6730C 1387439312 52B2A4D0 301
2 WAITING

SQL>

SID          SEQ#  EVENT                                P1 P1RAW      P2 P2RAW      P3
WAIT_TIME SECONDS_IN_WAIT STATE
-----
8            270 library cache pin      1389785868 52D6730C 1387439312 52B2A4D0 301
0 WAITING
```

在这个输出中，P1 列是 Library Cache Handle Address，Pn 字段是十进制表示，PnRAW 字段是十六进制表示。

Library Cache Pin 等待的对象的 handle 地址为 52D6730C。通过这个地址，查询 X\$KGLOBAL 视图就可以得到对象的具体信息。

—— 注 意 ——
X\$KGLOBAL 的名称含义为[K]ernel [G]eneric [L]ibrary Cache Manager [OB]ject。

```
col KGLNAOWN for a10
col KGLNAOBJ for a20
select ADDR,KGLHDADR,KGLHDPAR,KGLNAOWN,KGLNAOBJ,KGLNAHSH,KGLHDOBJ
from X$KGLOBAL
where KGLHDADR ='52D6730C'
```

```
/
```

ADDR	KGLHADDR	KGLHDPAR	KGLNAOWN	KGLNAOBJ	KGLNAHSH	KGLHDOBJ

404F9FF0 52D6730C	52D6730C	SYS		PINING	2300250318	52D65BA4

这里 **KGLNAHSH** 代表该对象的 Hash Value，由此可知，在 pinning 对象上正经历 Library Cache Pin 的等待，然后引入另外一个内部视图 **X\$KGLPN**。

—— 注 意 ——
X\$KGLPN 的名称含义为 [K]ernel [G]eneric [L]ibrary Cache Manager object [P]i[N]s。

```
select a.sid,a.username,a.program,b.addr,b.KGLPNADR,b.KGLPNUSE,b.KGLPNSEES,b.KGLPNHDL,
b.kGLPNLCK, b.KGLPNMOD, b.KGLPNREQ
from v$session a,x$kglnp b
where a.saddr=b.kglnpnuse and b.kglnpnhdl = '52D6730C' and b.KGLPNMOD<>0
/
```

SID	USERNAME	PROGRAM	ADDR	KGLPNADR	KGLPNUSE	KGLPNSEES	KGLPNHDL	KGLPNLCK	KGLPNMOD	KGLPNREQ

13	SYS	sqlplus@eygle.com (TNS V1-V3)	404FA034	52B2A518	51E2013C	51E2013C	52D6730C	52B294C8	2	0

通过联合 **v\$session**，可以获得当前持有该 handle 的用户信息。在本测试中，SID =13 的用户正持有该 handle。那么这个用户正在等什么呢？

```
SQL> select * from v$session_wait where sid=13;
```

SID	SEQ#	EVENT	P1TEXT	P1	P1RAW	P2TEXT
P2P2RAW	P3TEXT	P3 P3RAW	WAIT_TIME	SECONDS_IN_WAIT	STATE	

13	25	PL/SQL lock timer	duration	120000	0001D4C0	0
00	0 00		0	1200	WAITING	

由上可知，这个用户正在等待一次 **PL/SQL Lock Timer** 计时。
得到了 **SID**，就可以通过 **v\$session.SQL_HASH_VALUE**、**v\$session.SQL_ADDRESS** 等字段关联 **v\$sqltext**、**v\$sqlarea** 等视图获得当前 session 正在执行的操作。

```
SQL> select sql_text from v$sqlarea where v$sqlarea.hash_value='3045375777';
```

```
SQL_TEXT
```

```
-----
BEGIN calling; END;
```

这里得到这个用户正在执行 `calling` 这个存储过程，接下来的工作就应该去检查 `calling` 在做什么了。这个 `calling` 做的工作是 `dbms_lock.sleep (3000)`，也就是 PL/SQL Lock Timer 正在等待的原因，至此就找到了 Library Cache Pin 的原因。

简化一下以上查询。

(1) 获得 Library Cache Pin 等待的对象：

```
SELECT addr, kglhdadr, kglhdpdr, kglnaown, kglnaobj, kglnahsh, kglhdobj
      FROM x$kglob
     WHERE kglhdadr IN (SELECT p1raw
                        FROM v$session_wait
                        WHERE event LIKE 'library%')

/

ADDR          KGLHDADR KGLHDPAR KGLNAOWN   KGLNAOBJ          KGLNAHSH
KGLHDOBJ
-----
404F2178 52D6730C 52D6730C SYS          PINING            2300250318 52D65BA4
```

(2) 获得持有等待对象的 session 信息：

```
SELECT a.SID, a.username, a.program, b.addr, b.kglpnadr, b.kglpnuse,
       b.kglpnse, b.kglpnhdl, b.kglpnlck, b.kglpnmod, b.kglpnreq
      FROM v$session a, x$kgln b
     WHERE a.saddr = b.kglpnuse
           AND b.kglpnmod <> 0
           AND b.kglpnhdl IN (SELECT p1raw
                              FROM v$session_wait
                              WHERE event LIKE 'library%')

/

SQL>
      SID USERNAME   PROGRAM                                ADDR
KGLPNADR KGLPNUSE KGLPNSES KGLPNHDL KGLPNLCK   KGLPNMOD   KGLPNREQ
-----
13 SYS          sqlplus@eygle.com (TNS V1-V3)         404F6CA4 52B2A518 51E2013C
51E2013C 52D6730C 52B294C8          2          0
```

(3) 获得持有对象用户执行的代码：

```
SELECT sql_text
```

```

FROM v$sqlarea
WHERE (v$sqlarea.address, v$sqlarea.hash_value) IN (
    SELECT sql_address, sql_hash_value
    FROM v$session
    WHERE SID IN (
        SELECT SID
        FROM v$session a, x$kglnp b
        WHERE a.saddr = b.kglnpnuse
        AND b.kglnpnmod <> 0
        AND b.kglnphdl IN (SELECT p1raw
                            FROM v$session_wait
                            WHERE event LIKE 'library%'))
/

SQL_TEXT
-----

BEGIN calling; END;

```

在 grant 之前和之后，可以转储一下 Shared Pool 的内容，以进行观察和比较。

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 32';
Session altered.
```

在 grant 之前，从前面的查询获得 pining 的 handle 是 52D6730C:

```

*****
BUCKET 67790:

  LIBRARY OBJECT HANDLE: handle=52d6730c
  name=SYS.PINING
  hash=891b08ce timestamp=09-06-2004 16:43:51
  namespace=TABL/PRCD/TYPE flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-lill=0000-0011-0011 lock=N pin=S latch#=1
--在 Object 上存在共享 pin
--在 handle 上存在 Null 模式锁定,此模式允许其他用户继续以 Null/shared 模式锁定该对象
  lwt=0x52d67324[0x52d67324,0x52d67324] ltm=0x52d6732c[0x52d6732c,0x52d6732c]
  pwt=0x52d6733c[0x52b2a4e8,0x52b2a4e8] ptm=0x52d67394[0x52d67394,0x52d67394]
  ref=0x52d67314[0x52d67314, 0x52d67314] lnd=0x52d673a0[0x52d67040,0x52d6afcc]
  LIBRARY OBJECT: object=52d65ba4
  type=PRCD flags=EXS/LOC[0005] pflags=NST [01] status=VALD load=0
  DATA BLOCKS:
    data#      heap  pointer status pins change    alloc(K)  size(K)

```

0 52d65dac 52d65c90 I/P/A	0 NONE	0.30	0.55
4 52d65c40 52d67c08 I/P/A	1 NONE	0.44	0.48

在发出 grant 命令后:

BUCKET 67790:

LIBRARY OBJECT HANDLE: handle=52d6730c

name=SYS.PINING

hash=891b08ce timestamp=09-06-2004 16:43:51

namespace=TABL/PRCD/TYPE flags=KGHP/TIM/SML/[02000000]

kkkk-dddd-lill=0000-0011-0011 lock=X pin=S latch#=1

--由于 calling 执行未完成, 在 Object 上仍让保持共享 pin

--由于 grant 会导致重新编译该对象, 所以在 handle 上的排他锁已经被持有

--进一步的需要获得 Object 上的 Exclusive Pin, 由于 Shared Pin 被 calling 持有, 所以 Library Cache Pin 等待出现

lwt=0x52d67324[0x52d67324,0x52d67324] ltm=0x52d6732c[0x52d6732c,0x52d6732c]

pwt=0x52d6733c[0x52b2a4e8,0x52b2a4e8] ptm=0x52d67394[0x52d67394,0x52d67394]

ref=0x52d67314[0x52d67314, 0x52d67314] lnd=0x52d673a0[0x52d67040,0x52d6afcc]

LIBRARY OBJECT: object=52d65ba4

type=PRCD flags=EXS/LOC[0005] pflags=NST [01] status=VALD load=0

DATA BLOCKS:

data#	heap	pointer	status	pins	change	alloc(K)	size(K)
-------	------	---------	--------	------	--------	----------	---------

0 52d65dac 52d65c90 I/P/A	0 NONE	0.30	0.55
4 52d65c40 52d67c08 I/P/A	1 NONE	0.44	0.48

实际上 recompile 过程包含以下步骤, 同时来看一下 lock 和 pin 是如何交替发挥作用的。

- 存储过程的 Library Cache Object 以排他模式被锁定, 这个锁定是在 handle 上获得的。Exclusive 锁定可以防止其他用户执行同样的操作, 同时防止其他用户创建新的引用此过程的对象。

- 以 Shared 模式 pin 该对象, 以执行安全和错误检查。
- 共享 pin 被释放, 重新以排他模式 pin 该对象, 执行重编译。
- 使所有依赖该过程的对象失效。
- 释放 Exclusive Lock 和 Exclusive Pin。

2. LIBRARY CACHE LOCK 等待事件

如果此时再发出一条 grant 或 compile 的命令, 那么 Library Cache Lock 等待事件将会出现。
session 3:

```
SQL> alter procedure pining compile;
```

此进程挂起，查询 v\$session_wait 视图可以获得以下信息：

```
SQL> select * from v$session_wait;
```

SID	SEQ#	EVENT	P1TEXT	P1	P1RAW	P2TEXT
P2 P2RAW	P3TEXT	P3	P3RAW	WAIT_TIME	SECONDS	STATE
-----	-----	-----	-----	-----	-----	-----
11	143	library cache pin	handle address	1390239716	52DD5FE4	pin address 1387617456
52B55CB0	100	*mode+namespace 301 0000012D		0	6	WAITING
13	18	library cache lock	handle address	1390239716	52DD5FE4	lock address 1387433984
52B29000	100	*mode+namespace 301 0000012D		0	3	WAITING
8	415	PL/SQL lock timer	duration	120000	0001D4C0	
00		0 00		0	63	WAITING
....						
13 rows selected						

由于 handle 上的 lock 已经被 session 2 以 Exclusive 模式持有，所以 session 3 产生了等待。

可以看到，在生产数据库中权限的授予、对象的重新编译都可能会导致 Library Cache Pin 等待的出现，所以应该尽量避免在高峰期进行以上操作。

另外，测试的案例本身就说明，如果 Package 或过程中存在复杂的、交互的依赖关系就极易导致 Library Cache Pin 的出现，所以在应用开发的过程中，也应该注意这方面的问题。

5.2.5 诊断案例一：version_count 过高造成的 Latch 竞争解决

本节是关于 Shared Pool 的一个诊断案例，案例本身可能并不重要，重要的是给大家一个解决问题的思路，并且通过这个案例可以进一步了解 Oracle 的工作原理。

问题起因是公司要进行短信群发，群发的时候每隔一段时间就会发生一次消息队列拥堵的情况。在数据库内部实际上是向一个数据表中记录发送日志。数据库版本是 Oracle 8.1.5。

在一个拥堵时段我开始诊断：

```
SQL> select sid,event,p1,p1raw from v$session_wait;
```

SID	EVENT	P1	P1RAW
-----	-----	-----	-----
76	latch free	2147535824	8000CBD0
83	latch free	2147535824	8000CBD0
148	latch free	3415346832	CB920E90
288	latch free	2147535824	8000CBD0

```

285 latch free                2147535824 8000CBD0
196 latch free                2147535824 8000CBD0
317 latch free                2147535824 8000CBD0
   3 log file parallel write    1 00000001
  13 log file sync              2705 00000A91
  60 SQL*Net message to client 1413697536 54435000
 239 SQL*Net message to client 1413697536 54435000
...ignore some idle waiting here...
  11 SQL*Net message from client 675562835 28444553
  12 SQL*Net message from client 1413697536 54435000

170 rows selected.

```

在这次查询中，发现了大量的 Latch Free 等待，再次查询时这些等待消失，应用也恢复了正常。

```
SQL> select sid,event,p1,p1raw from v$session_wait where event not like 'SQL*Net%';
```

SID EVENT	P1 P1RAW
2 pmon timer	300 0000012C
1 rdbms ipc message	300 0000012C
4 rdbms ipc message	300 0000012C
6 rdbms ipc message	180000 0002BF20
18 rdbms ipc message	6000 00001770
102 rdbms ipc message	6000 00001770
178 rdbms ipc message	6000 00001770
194 rdbms ipc message	6000 00001770
311 rdbms ipc message	6000 00001770
3 log file parallel write	1 00000001
148 log file sync	2547 000009F3
273 log file sync	2544 000009F0
190 log file sync	2545 000009F1
5 smon timer	300 0000012C

```
14 rows selected.
```

接下来，来看这些 Latch Free 等待的是哪些 Latch：

```
SQL> select addr,latch#,name,gets,spin_gets from v$latch order by spin_gets;
```

ADDR	LATCH# NAME	GETS	SPIN_GETS
80001398	3 session switching	111937	0
80002010	6 longop free list	37214	0
.....			
80001330	2 session allocation	261826230	428312
800063E0	64 multiblock read objects	1380614923	1366278
800026B8	11 messages	207935758	1372606
80001218	0 latch wait list	203479569	1445342
80006310	62 cache buffers chains	3.8472E+10	2521699
8000A17C	92 row cache objects	1257586714	2555872
80007F80	74 redo writing	264722932	4458044
80006700	67 cache buffers lru chain	5664313769	30046921
8000CBD0	98 shared pool	122433688	59070585
8000CC38	99 library cache	4414533796	1037032730
142 rows selected.			

可以注意到，在当前数据库中竞争最严重的两个 Latch 是 Shared Pool 和 Library Cache。这两个 Latch 是 Shared Pool 管理中最重要也是最常见的 Latch 竞争。

Shared Pool Latch 用于共享池中内存空间的分配和回收，如果 SQL 没有充分共享，反复解析的过程将是十分昂贵的，这个问题已经在上文中论述过。

而 Library Cache Latches 用于保护 Cache 在内存中的 SQL 以及对象定义等，当需要向 Library Cache 中增加新的 SQL 时，Library Cache Latch 必须被获得。在解析 SQL 过程中，Oracle 搜索 Library Cache 查找匹配的 SQL，如果没有可共享的 SQL 代码，Oracle 将分析 SQL，获得 Library Cache Latch 向 Library Cache 中插入新的 SQL 代码。

Library Cache Latch 的数量受一个隐含参数 `_kg1_latch_count` 的控制，其缺省值大于或等于 CPU_COUNT 的素数，最大值不能超过 66。

下面简化一下 SQL 的执行过程，以说明这两个 Latch 在 SQL 解析过程中所起的作用。

(1) 首先需要获得 Library Cache Latch，根据 SQL 的 HASH_VALUE 值在 Library Cache 中寻找是否存在可共享代码。如果找到则为软解析，Server 进程获得该 SQL 执行计划，转向第 (4) 步；如果找不到共享代码则执行硬解析。

(2) 释放 Library Cache Latch，获取 Shared Pool Latch，查找并锁定自由空间。

(3) 释放 Shared Pool Latch，重新获得 Library Cache Latch，将 SQL 及执行计划插入到 Library Cache 中。

(4) 释放 Library Cache Latch，保持 Null 模式的 Library Cache Pin/Lock。

(5) 开始执行。

可以看到，如果系统中存在过度的硬解析，系统的性能必然受到反复解析、Latch 争用的折磨。

可以通过查询 v\$sysstat 视图获得关于数据库解析的详细信息：

```
SQL> select name,value from v$sysstat where name like 'parse%';
```

NAME	VALUE

parse time cpu	66
parse time elapsed	505
parse count (total)	801
parse count (hard)	193
parse count (failures)	0

通过 $[\text{parse count (total)} - \text{parse count (hard)}] / \text{parse count (total)}$ 得出的软解析率经常被用作衡量数据库性能的一个重要指标。现在，回到原来的问题上来，过多的 Shared Pool 和 Library Cache 竞争，显然极有可能是 SQL 的过度解析造成的。

进一步检查 v\$sqlarea，可以发现：

```
SQL> select sql_text,VERSION_COUNT,INVALIDATIONS,PARSE_CALLS,
OPTIMIZER_MODE,PARSING_USER_ID,PARSING_SCHEMA_ID,ADDRESS,HASH_VALUE
2 from v$sqlarea where version_count >1000;
```

```
SQL_TEXT
-----
VERSION_COUNT      INVALIDATIONS      PARSE_CALLS      OPTIMIZER_MODE
PARSING_USER_ID PARSING_SCHEMA_ID ADDRESS  HASH_VALUE
-----
insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,
MSGSTATUS,AREAAID,IFIDDEST,IFIDSRC,ADDRSRC
,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,PLANID,FEETYPE,FEEVALUE,DATACODING,
FLAGS,SMLen,SMCONT) values (:b0,:b1,:b2,:b3,:b
4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b21,:b22)
7023              0              1596 MULTIPLE CHILDREN PRESENT              36
36 C82AF1C8 3974744754
```

这就是写日志记录的代码，这段代码使用了绑定变量，但是 version_count 却有 7023 个，也就是这个 SQL 有 7023 个子指针，这是不可想象的。

通过前面几节的内容可以知道，如果这个 SQL 有 7023 个子指针，就意味着这些子指针都将存在于同一个 Bucket 的链表上。那么这也就意味着，如果同样 SQL 再次执行，Oracle 将不得不搜索这个链表以寻找可以共享的 SQL。这将导致大量的 Library Cache Latch 的竞争。

应该注意数据库中 version_count 过多的 SQL 语句，version_count 过高通常会导致 Library Cache Latch 的长时间持有，从而影响性能，所以很多时候应该尽量避免这种情况的出现。最

简单的，比如 scott 和 eygle 两个用户同时执行：

```
Select * from emp;
```

如果 scott 和 eygle 各拥有一张 emp 表，那么这条 SQL 将存在两个子指针，而显然两者代码不能共享。所以，虽然 Oracle 支持不同用户拥有同名对象，但还是应该尽量避免。

继续问题的研究，这时我开始猜测问题的原因：

(1) 可能代码存在问题，在每次执行之前程序修改某些 session 参数，导致 SQL 不能共性。

(2) 可能是 8.1.5 的 v\$sqlarea 记录存在问题，刚才看到的结果是假象。

(3) Oracle 的 Bug。

继续诊断，最直接地，dump 内存来看：

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 4';
```

查看 trace 文件得到如下结果（摘录包含该段代码的片断）：

```
BUCKET 21049:

  LIBRARY OBJECT HANDLE: handle=c82af1c8

  name=

  insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,
MSGSTATUS,AREAD,IFIDDEST,IFIDSRC,
  ADDR SRC,ADDR DEST,ADDR FEE,ADDR USER,SERVICE CODE,PLAN ID,FEETYPE,FEEVALUE,DAT
ACODING,FLAGS,SMLN,SMCONT) values
(:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b21,:b22)

  hash=ece9cab2 timestamp=09-09-2004 12:51:29

  namespace=CRSR flags=RON/TIM/PNO/LRG/[10010001]

  kkkk-dddd-llll=0000-0001-0001 lock=N pin=S latch=5

  lwt=c82af1e0[c82af1e0,c82af1e0] ltm=c82af1e8[c82af1e8,c82af1e8]

  pwt=c82af1f8[c82af1f8,c82af1f8] ptm=c82af250[c82af250,c82af250]

  ref=c82af1d0[c82af1d0,c82af1d0]

  LIBRARY OBJECT: object=c1588e84

  type=CRSR flags=EXS[0001] pflags= [00] status=VALD load=0

  CHILDREN: size=7024

  child#      table reference      handle

  -----

      0 c1589040  c1589008 c668c2bc

      1 c1589040  bfd179c4 c6ec9ee8

      2 c1589040  bfd179e0 c2dd9b3c

      3 c1589040  bfd179fc c5a46614

  .....

  .....ignore losts of child cursor here.....
```

```

.....
7016 c641fcb8 c0ae4f2c c60196d0
7017 c641fcb8 c0ae4f48 c4675d2c
7018 c641fcb8 c0ae4f64 bd5e2750
7019 c641fcb8 c0ae4f80 c09b1bb0
7020 c641fcb8 c0ae4f9c bf2d6044
7021 c641fcb8 c0ae4fb8 c332c1c4
7022 c641fcb8 c0ae4fd4 cbdde0f8
DATA BLOCKS:
data#      heap  pointer status pins change
-----
0 c3ef2c50 c1588f08 I/P/A      0 NONE

```

这里确实存在 7023 个子指针，查询 v\$sql 得到相同的结果：

```

SQL> select CHILD_NUMBER,EXECUTIONS,OPTIMIZER_MODE,OPTIMIZER_COST,PARSING_USER_
D,PARSING_SCHEMA_ID,ADDRESS,HASH_VALUE
2 from v$sql where HASH_VALUE='3974744754';

CHILD_NUMBER    EXECUTIONS    OPTIMIZER_    OPTIMIZER_COST    PARSING_USER_ID
PARSING_SCHEMA_ID ADDRESS    HASH_VALUE
-----
0              12966 CHOOSE              238150              36              36
C82AF1C8 3974744754
1              7111 CHOOSE              238150              36              36
C82AF1C8 3974744754
.....
7020           625 CHOOSE              237913              36              36
C82AF1C8 3974744754
7021          10101 CHOOSE              237913              36              36
C82AF1C8 3974744754
7022           7859 CHOOSE              237913              36              36
C82AF1C8 3974744754

7023 rows selected.

```

这里确实存在 7023 个子指针，第（2）种猜测被否定了，同时查看源代码发现也不存在第（1）种情况。那么只能是第（3）种情况了，Oracle 的 Bug，那就需要找到对应的解决办法。

搜索 MetaLink，发现 Bug：1210242，该 Bug 描述为：

On certain SQL statements cursors are not shared when TIMED_STATISTICS is enabled.

碰巧这个数据库的 TIMED_STATISTICS 设置为 true，修改 TIMED_STATISTICS 为 false 以后，观察 v\$sql，发现有效子指针很快下降到 2 个。

```
SQL> select CHILD_NUMBER,OPTIMIZER_COST,OPTIMIZER_MODE,EXECUTIONS,ADDRESS from
v$sql where hash_value=3974744754 and OPTIMIZER_MODE='CHOOSE';
```

CHILD_NUMBER	OPTIMIZER_COST	OPTIMIZER_MODE	EXECUTIONS	ADDRESS
0	238167	CHOOSE	63943	C82AF1C8
1	238300	CHOOSE	28915	C82AF1C8

第二天下降到只有一个：

```
SQL> select CHILD_NUMBER,OPTIMIZER_COST,OPTIMIZER_MODE,EXECUTIONS,ADDRESS from
v$sql where hash_value=3974744754 and OPTIMIZER_MODE='CHOOSE';
```

CHILD_NUMBER	OPTIMIZER_COST	OPTIMIZER_MODE	EXECUTIONS	ADDRESS
0	238702	CHOOSE	578124	C82AF1C8

短信群发从此正常。

对于这个问题，另外一个可选的方法是设置一个隐含参数：

```
_sqlxexec_progression_cost = 0
```

这个参数的具体含义为：SQL execution progression monitoring cost threshold，即 SQL 执行进度监控成本阈值。

这个参数根据 COST 来决定需要监控的 SQL。执行进度监控会引入额外的函数调用和 Row Sources 这可能导致 SQL 的执行计划或成本发生改变，从而产生不同的子指针。

_sqlxexec_progression_cost 的缺省值为 1000，成本大于 1000 的所有 SQL 都会被监控，如果该参数设置为 0，那么 SQL 的执行进度将不会被跟踪。

执行进度监控信息会被记录到 v\$session_longops 视图中，如果 TIME_STATISTICS 参数设置为 False，那么这个信息就不会被记录。所以，TIME_STATISTICS 参数和 _sqlxexec_progression_cost 是解决问题的两个途径。

通过查询也可以看到，在这个数据库中，OPTIMIZER_COST>1000 的 SQL 主要有以下 5 个：

```
SQL> select distinct(sql_text) from v$sql where OPTIMIZER_COST >1000;
```

SQL_TEXT

```
insert into sms_detail_error (msgdate,addruser,msgid,areaid,reason,spnumber,msgt
ime,ifiddest,msqkey,servicecode,planid,feetype,feevalue,smcont,submittimes,submi
```

```

tdate,submittime,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20)

insert into sms_detail_success (msgdate,addruser,msgid,areaid,spnumber,msgtime,ifiddest,servicecode,planid,feetype,feevvalue,smcont,submittimes,submitdate,submittime,respdate,respstime,repdate,repptime,msqkey) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19)

insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,MSGSTATUS,ARE Aid,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,P
LANID,FEETYPE,FEEVALUE,DATA CODING,FLAGS,SML EN,SMCONT) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b21,:b22)

insert into sms_resprept_error (msgdate,areaid,addruser,msgid,submittimes,submitdate,submittime,msgid_gw,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept,servicecode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12)

insert into sms_statusrept (repdate,addruser,msgid_gw,repptime,status type,msgid_gw,msgstate,errorcode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7)

```

而这 5 个 SQL 中，在 v\$sqlarea 中，有 4 个 version_count 都在 10 以上：

```
SQL> select sql_text,version_count from v$sqlarea where version_count>10;
```

```
SQL_TEXT
```

```
-----
```

```
VERSION_COUNT
```

```
-----
```

```

insert into sms_detail_error (msgdate,addruser,msgid,areaid,reason,spnumber,msgtime,ifiddest,msqkey,servicecode,planid,feetype,feevvalue,smcont,submittimes,submitdate,submittime,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20)

```

42

```
insert into sms_log (MSGDATE,MSGTIME,MSGID,MSGKIND,MSGTYPE,MSGTYPE_MOMT,MSGLEN,MSGSTATUS,ARE Aid,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,P
```

```

SGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICEC
ODE,P
LANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,SMLen,SMCONT) values (:b0,:b1,:b2,:b3,:b
4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b20,:b2
1,:b22)

7026

insert into sms_resprept_error (msgdate,areaid,addruser,msgid,submittimes,submit
date,submittime,msgid_gw,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_re
pt,servicecode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12)

301

insert into sms_statusrept (reptdate,addruser,msgid_gw,reptime,statustype,msgid
_stus,msgstate,errorcode) values (:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7)

41

```

具体可以参考 Metalink: Note 62143。至此，这个关于 Shared Pool 的问题找到了原因，并得以及时解决。

最近在 Oracle 9i 中遇到过另外一个类似案例，同样是 Library Cache Latch 和 Shared Pool Latch 竞争，类似的在 v\$sqlarea 中发现大量高 version_count 的 SQL。

对于 version_count 过高的问题，可以查询 v\$sql_shared_cursor 视图，这个视图会给出 SQL 不能共享的具体原因，如果是正常因素导致的，相应的字段会被标记为“Y”；对于异常的情况（如本案例），查询结果可能显示的都是“N”，这就表明 Oracle 认为这种行为是正常的，在当前系统设置下，这些 SQL 不应该被共享，那么可以判断是某个参数设置引起的。

和 Cursor Sharing 关系最大的一个初始化参数就是：cursor_sharing，在这个案例中 cursor_sharing 参数被设置为 similar，正是这个设置导致了大量子指针不能共享。

搜索 Metalink，可以获得相关说明，当 cursor_sharing 参数设置为 similar，并且数据库存在相关柱状图（Histograms）信息时，对于每一条新执行的 SQL，Oracle 都通过硬解析以获得更为精确的执行计划，这最终导致了 version_count 过高，这是 cursor_sharing = similar 的正常行为，而并非 Bug。

了解了这个行为之后，解决这个问题也就并不复杂了，可以将 cursor_sharing 设置为 Exact 或者 Force 以避免此问题，或者通过删除柱状图信息（Histograms）来防止不必要的硬解析，实际上，如果数据不存在失衡分布，也不必要收集柱状图信息。

这两个案例给我们的另外一个启示就是，当需要设置某些特殊的参数来影响数据库的行为时，必须明白这些设置会给数据库带来的影响，这样一方面可以避免问题的出现，另一方面在问题出现时，也可以快速地发现问题的根源，快速地解决问题。

5.2.6 诊断案例二：临时表引发的竞争

这是帮助一个网友解决的一个问题（通过 MSN 交流），以下是问题的解决过程及思路，

供大家参考。

问：如果一个 DB 里面的几个存储过程总是跑不完，同样的存储过程在其他的 6 个省都很正常，数据库里没有锁，数据库和 Server 上面的空间足够。正常的情况几分钟就能运行完，现在都 n 多小时了还没有运行完，会是什么原因呢？

答：检查 v\$session_wait，看系统在等什么？

—— 提 示 ——

如果系统慢，通常是存在等待，v\$session_wait 是应该优先检查的视图。

下面是这位网友发过来的查询结果，这里截取了主要的部分：

SID	SEQ#	EVENT	P1TEXT	P1	P1RAW	P2TEXT	P2
13	7210	library cache pin	handle address	3172526924	BD18EB4C	pin	address
		3205742908					
33	16179	library cache pin	handle address	3172526924	BD18EB4C	pin	address
		3206485860					
32	14721	library cache pin	handle address	3172526924	BD18EB4C	pin	address
		3206555324					
27	54913	library cache pin	handle address	3172526924	BD18EB4C	pin	address
		3205741540					
30	16169	library cache lock	handle address	3174604528	BD389EF0	lock	address
		3206478252					

可以看到，数据库目前正在经历 Library Cache Pin 和 Library Cache Lock 的等待和竞争。我要求网友执行本章上文中讲到的 SQL，并提供结果：

```
SQL> select ADDR,KGLHDADR,KGLHDPAR,KGLNAOWN,KGLNAOBJ,KGLNAHSH,KGLHDOBJ
2   from X$KGLGB
3  where KGLHDADR ='BD18EB4C'
4  ;

ADDR      KGLHDADR KGLHDPAR  KGLNAOWN  KGLNAOBJ  KGLNAHSH KGLHDOBJ
-----
01920880  BD18EB4C  BD18EB4C          truncate table iptt_pm_all  653109544 BD18E8D4

SQL> SELECT a.SID, a.username, a.program, b.addr, b.kglpnadr, b.kglpnuse,
2          b.kglpnuses, b.kglpnhdl, b.kglpnlck, b.kglpnmod, b.kglpnreq
3 FROM v$session a, x$kglpn b
4 WHERE a.saddr = b.kglpnuse
5 AND b.kglpnmod <> 0
6 AND b.kglpnhdl IN (SELECT p1raw
```

```

7          FROM v$session_wait
8          WHERE event LIKE 'library%')
9  ;

      SID USERNAME
-----
PROGRAM                                ADDR      KGLPNADR KGLPNUSE
-----
KGLPNSES KGLPNHDL KGLPNLCK   KGLPNMOD   KGLPNREQ
-----

      30 IPNMS
sqlplus@gs-db (TNS V1-V3)                0191BEC0 BF2024C4 BE0AE940
BE0AE940 BD18EB4C BF1FA208                3          0

      54 IPNMS
sqlplus@gs-db (TNS V1-V3)                0191BEC0 BF13814C BE0BB360
BE0BB360 BD389EF0 00                      3          0

SQL> SELECT sql_text
2  FROM v$sqlarea
3  WHERE (v$sqlarea.address, v$sqlarea.hash_value) IN (
4  SELECT sql_address, sql_hash_value
5  FROM v$session
6  WHERE SID IN (
7  SELECT SID
8  FROM v$session a, x$kglpn b
9  WHERE a.saddr = b.kglpnuse
10 AND b.kglpnmod <> 0
11 AND b.kglpnhdl IN (SELECT p1raw
12 FROM v$session_wait
13 WHERE event LIKE 'library%'))
14 ;

SQL_TEXT
-----
truncate table iptt_pm_all

```

至此，发现了导致问题的关键所在，持有 pin 的用户在执行 truncate table iptt_pm_all 的操作。

问：这个 truncate 是嵌在过程里面的？

答：是的，在一个 loop 中间的。每半个小时调用一次，类似的怎么也有 10 个程序吧。公用 iptt_pm_all 临时表。

我请求查看网友的代码，在一个 Procedure 中发现了大量以下语句（做了适当简化）：

```
update iptt_pm_all p
    set n27 = (SELECT count(*)
                FROM iptca_interface b
               WHERE p.int_id = b.related_node
                  AND b.ifttype = 18
                  AND b.IFOPERSTATUS IN (1,5));

insert into iptt_pm_all (col_time, int_id, ipaddr,
                        n1, c1, n2, c2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
                        n17, n18, n19, n20, c3, n21, c4, n22, c5, n23, n24, n25, n26, n27)
    select compress_day, int_id, object_ip_addr, .....
    from iptaws_gwgj_hour
    where compress_day = v_time.col_time;

SELECT col_time,n21,c4,n22,
       c5,n1,c1,n2,c2,
       sum(n3),sum(n4),sum(n5),sum(n7),sum(n8),sum(n9),
       sum(n10),sum(n11),sum(n12),sum(n13),sum(n14),
       sum(n16),sum(n17),sum(n18),sum(n19),sum(n20)
    FROM iptt_pm_all
   GROUP BY col_time,n21,c4,n22,c5,n1,c1,n2,c2;

v_dsqli := 'truncate table iptt_pm_all';
EXECUTE IMMEDIATE v_dsqli;
```

类似的存储过程还有很多。我请求获取 Shared Pool 的转储文件用于分析，Level 32 级。

```
ALTER SESSION SET EVENTS 'immediate trace name LIBRARY_CACHE level 32';
```

限于篇幅，这里不再列举 dump 文件内容，需要注意的是，在生产环境上使用以上命令应该十分慎重。如果 Shared Pool 很大，转储文件可能非常巨大，而且可能引发性能问题和 Bug。

根据 trace 文件及 MetaLink 说明，最终发现问题是由于 truncate 临时表时不适当地请求了排他锁所致，理论上 truncate 临时表无需排他锁定，但是 Oracle 使用了与处理常规表同样的方式处理临时表的锁定，从而导致了 Library Cache Pin 和 Library Cache Lock 的竞争，而且该问题并未作为 Bug 修正。

由于该问题主要当多户交叉访问时引起，所以建议对于不同用户改用独立的临时表，此问题就可得以避免。

5.2.7 小结

Shared Pool 的管理是 Oracle 内存管理中相对复杂的一部分内容，在性能调整时也是非常重要的内容，深刻理解 Shared Pool 的实现，有助于进一步了解 Oracle 的实现及内部机制。本章就这一方面进行了一点探索，由于个人能力及认知有限，错漏之错在所难免，期待大家指正。

限于篇幅，本章作了适当简化，更完整的内容你可以在我的网站（www.eygle.com）上找到。

第6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撤消性。本章将对 Oracle 的重做机制进行讨论。

6.1 Redo 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 3 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 最终会写出为归档日志文件)。

在 Oracle 的 SGA 中, 存在一块共享内存, 称为 Redo Log Buffer, 如图 6-1 所示。

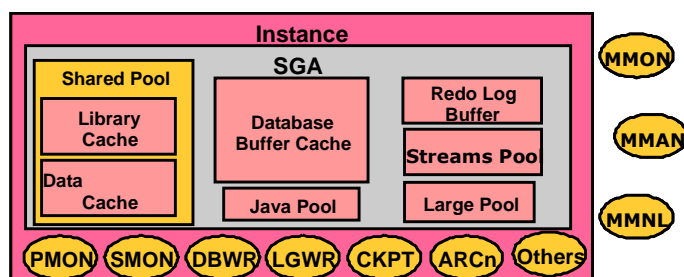


图 6-1 Oracle Instance

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 不断地把 Redo Log Buffer 的内容写出到 Redo Log File 中。

当用户在 Buffer Cache 中修改数据时，Oracle 并不会立即将修改数据写出到数据文件上，因为那样做效率会很低，到目前为止，计算机系统中最繁忙的部分是磁盘的 I/O 操作，Oracle 这样做的目的是为了减少 IO 的次数，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的性能提升。

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。缺省情况下，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要尽量得短。Oracle 通过检查点（Checkpoint）来缩减恢复时间。回顾一下第 1 章中所提到的内容：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 进程会相应地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。

6.2 Redo 的内容

Oracle 通过 Redo 来实现快速提交，一方面是因为 Redo Log File 可以连续、顺序地快速

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，首先需要了解两个概念：改变向量和重做记录。

■ 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。

■ 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 6-2 所示。

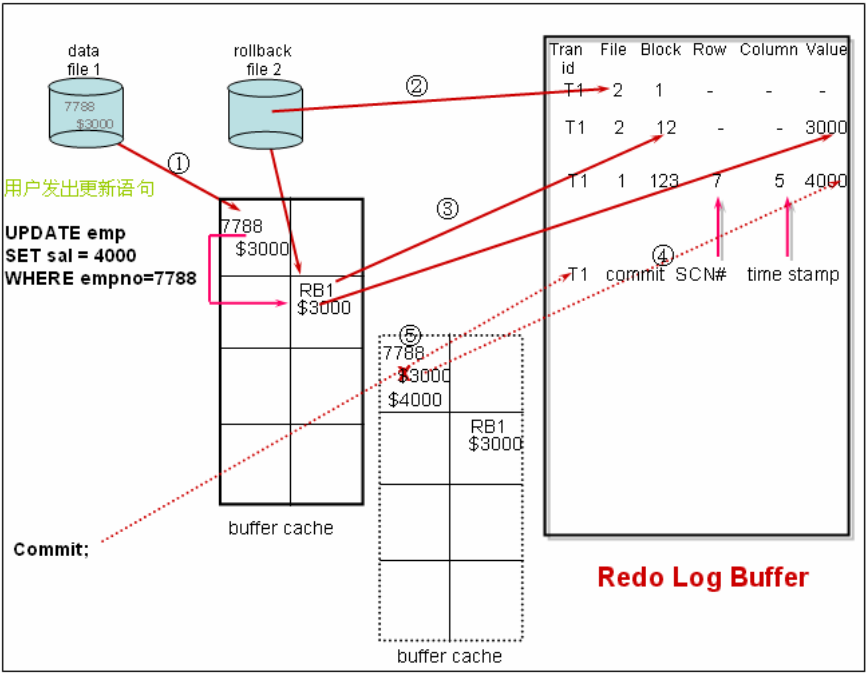


图 6-2 更新 (Update) 操作引起的重做

假定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, 这里尽量简化了情况):

- (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)。

下面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	3.9035E+12	26-MAR-06
2	1	309	10485760	1	NO	INACTIVE	3.9035E+12	19-MAR-06
3	1	311	10485760	1	NO	CURRENT	3.9035E+12	26-MAR-06
4	1	308	1048576	1	NO	INACTIVE	3.9035E+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
2 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
```

此脚本在本书中多次使用，现收录于下：

```
SELECT      d.VALUE
           || '/'
           || LOWER (RTRIM (i.INSTANCE, CHR (0)))
           || '_ora_'
           || p.spid
           || '.trc' trace_file_name
FROM (SELECT p.spid
      FROM SYS.v$mystat m, SYS.v$session s, SYS.v$process p
      WHERE m.statistic# = 1 AND s.SID = m.SID AND p.addr = s.paddr) p,
      (SELECT t.INSTANCE
      FROM SYS.v$thread t, SYS.v$parameter v
      WHERE v.NAME = 'thread'
            AND (v.VALUE = 0 OR t.thread# = TO_NUMBER (v.VALUE))) i,
      (SELECT VALUE
      FROM SYS.v$parameter
      WHERE NAME = 'user_dump_dest') d
/
```

(5) 获取 Trace 文件。

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

■ 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）可以知道这是 UNDO 表空间，通过 UBA 机 DBA 的换算能够找到相应的 Block，限于篇幅，本文不再做过多的介绍。

```
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
```

■ 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）。

```
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
```

■ 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

```

■ 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
```

■ 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 8KB 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（如事务号、文件号、块号、行号、字段等）即可，这个数据量大大减少。

—— 注 意 ——

以上的部分转储信息只是为了说明 Oracle 的运行机制，大家从中了解原理即可，有兴趣的朋友可以继续深入研究，更为详细的内容可以从 www.eygle.com 上得到。

6.3 产生多少 Redo

但凡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 功能时

当在 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：

```

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
.....
      28  rows processed

```

2. 通过 v\$mystat 查询

Oracle 通过 v\$mystat 视图记录当前 session 的统计信息，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
2  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，可以通过 v\$sysstat 视图来查询得到：

```
SQL> col value for 999999999999999
```

```
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，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 生成量并不带表归档日志的大小，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\$sarchived_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$sarchived_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_171916.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$sarchived_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	16.8974366

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$sarchived_log)
4 GROUP BY TRUNC (completion_time)
5 /
```

TRUNC(COM	DAY_GB
-----------	--------

```

-----
28-APR-06 8.63226318
29-APR-06 11.6235332
30-APR-06 16.7366991
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 16.8974366
10-MAY-06 19.4107008
11-MAY-06 11.2606988

```

```
14 rows selected.
```

根据每日归档的生成量，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。

6.4 Redo 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，非常熟悉的 LGWR 写触发条件就有以下几种。

6.4.1 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地观察到这些事件：

```

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

```

6.4.2 阈值达到

在各种文档上经常会看到的两个触发日志写的条件是：

- Redo Log Buffer 1/3 满；
- Redo Log Buffer 具有 1MB 脏数据。

这两者都是限制条件，在触发时是协同生效的。

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 1MB，此参数在 `X$KSPPSV` 中显示的 0 值，意为缺省值。

注 意

`X$KSPPSV` 名称的含义为 [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。

```
20:33:15 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

获得隐含参数，请参考如下脚本：

```

set linesize 132
column name format a30
column value format a25
select
    x.ksppinm name,
    y.ksppstvl value,

```

```

y.kspstdf isdefault,
decode(bitand(y.kspstvf,7),1,'MODIFIED',4,'SYSTEM_MOD','FALSE') ismod,
decode(bitand(y.kspstvf,2),2,'TRUE','FALSE') isadj
from
sys.x$kspci x,
sys.x$ksppcv y
where
x.inst_id = userenv('Instance') and
y.inst_id = userenv('Instance') and
x.indx = y.indx and
x.kspinm like '%_&par%'
order by
translate(x.kspinm, ' _', '')
/

```

经常见到有人推荐 Log Buffer 设置为 3MB 大小,就是因为当 Redo Log Buffer 为 3MB 时,以上两个条件可能同时达到。

6.4.3 用户提交

当一个事务提交时,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,这个事务是不可恢复的。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 来避免 Redo File Sync 的等待,但是将无法保证事务的恢复性。

```
20:46:02 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 (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)。

6.4.4 在 DBWn 写之前

如果 DBWR 将要写出的数据的高 RBA 超过 LGWR 的 on-Disk RBA，则 DBWR 将通知 LGWR 去执行写出。

在 Oracle 8i 之前，此时 DBWR 将等待 Log File Sync 事件。从 Oracle 8i 开始，DBWR 把这些 Block 放入一个 Defer 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。

在一个生产库中，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	316.3	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	16.3	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。

6.5 Redo Log Buffer 的大小设置

Redo Log Buffer 的大小由初始化参数 LOG_BUFFER 定义，该参数的缺省值如下：

Max(512 KB, 128 KB * CPU_COUNT)

通常这一缺省值是足够的，从上一节可以知道，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，根据上文的介绍，一般来说 3MB 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，可以考虑增大 Log Buffer 以缩减竞争。

6.6 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。

那么应该记住的一个原则是：保证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 允许用户对重做日志文件进行镜像。

6.7 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

最常见的状态有以下几种：CURRENT、ACTIVE、INACTIVE 和 UNUSED。

1. CURRENT

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，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，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 文件中清晰地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，可以使读者更好地了解 Oracle 的运行机制。

可以对这个问题做一个简单分析，引起 Checkpoint incomplete 可能有以下多种原因：

- 日志文件过小，切换过于频繁；
- 日志组太少，不能满足正常事务量的需要；
- 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；
- 由于数据文件磁盘 I/O 瓶颈，DBWR 写出过于缓慢；
- 由于事务量具大，DBWR 负荷过高，不堪重负。

针对不同的原因，可以从不同角度着手解决问题：

- 适当增加日志文件大小；
- 适当增加日志组数；
- 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）；
- 改善磁盘 I/O 性能；
- 使用多个 DBWR 进程或使用异步 I/O 等。

总之，只要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。必须强调的是，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

INACTIVE 的日志是非活动日志，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
3.9035E+12 26-MAR-06
          2          1          309    10485760             1 NO         INACTIVE
3.9035E+12 19-MAR-06
          3          1          311    10485760             1 NO         CURRENT
3.9035E+12 26-MAR-06
          4          1          308     1048576             1 NO         INACTIVE
3.9035E+12 19-MAR-06
```

注意到此时所有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，通常可以通过检查警报日志文件（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

UNUSED 是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。

6.8 日志的块大小

初始化参数 LOG_BUFFER 决定了 Redo Log Buffer 的大小，这个参数的缺省值为

Max(512K,128K*CPU_count)。

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 512bytes 为单位,如 Solaris、AIX、Windows NT/2000、Linux 等。

这个 Log Size 可以从 Oracle 的内部视图获得:

```
SQL> select max(lebsz) from x$kccl;
```

```
MAX(LEBSZ)
```

```
-----
```

```
512
```

也可以从 v\$sysstat 中的统计信息中通过计算粗略得到,主要有以下几个统计信息。

- 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 16bytes。

由此可以粗略地计算 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]$ dbfsiz redo01.log
```

```
Database file: redo01.log
```

```
Database file type: file system
```

```
Database file size: 20480 512 byte blocks
```

```
[oracle@jumper conner]$ dbfsiz system01.dbf
```

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

从以上输出中，可以看到，日志文件的 **Block Size** 是 512bytes，而数据文件的 **Block Size** 为 8192bytes。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 redo_hdr level 10';
```

查看跟踪文件可以得到类似以下信息：

```
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: fwr: 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
```

可以得到，bsz 就是 512bytes。

提 示

本节提供的多种方法，旨在开阔大家的视野以及思路，对 Oracle 了解得越多，学习研究起来就能够更加得心应手。

6.9 日志文件的大小

前面提到，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。所以如何合理地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，那么就会丢失数据，所以更大的日志文件可能意味着更多的数据损失风险。所以数据库的任何一个调整都需要慎重。

让我们从 Oracle 在不同版本的变化中，揣摩一下 Oracle 的心思。在 Oracle 8i 之中，缺省的 Redo Log File 大小是 1MB：

```
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"
```

```
2 from v$log;
```

GROUP#	THREAD#	SEQUENCE#	M bytes
1	1	364	1
2	1	365	1
3	1	363	1

而在 Oracle 9iR2 中，这个缺省值更改为 100MB:

```
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"
```

```
2 from v$log;
```

GROUP#	THREAD#	SEQUENCE#	M bytes
1	1	130	100
2	1	128	100
3	1	129	100

进一步的，在 Oracle 10gR1 中，这个 Redo Log File 的缺省值被改变为 10MB 大小:

```
SYS AS SYSDBA on 18-MAR-06 >select * from v$version where rownum <2;
```

```
BANNER
```

```
-----
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - 64bi
```

```
SYS AS SYSDBA on 18-MAR-06 > select group#,thread#,sequence#,bytes/1024/1024 "M bytes"
```

```
2 from v$log;
```

GROUP#	THREAD#	SEQUENCE#	M bytes
--------	---------	-----------	---------

1	1	173	10
2	1	174	10
3	1	172	10

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, 用户在优化调整过程中, 实际上需要考虑得更多。一般来说, 在实际生产环境中, 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 2GB, 在非常繁忙的业务系统中, 由于受限于日志大小, 能将日志控制在 10 分钟左右就已经算不错了。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是在缺省设置下运行, 这就导致了设置不合理和资源浪费。

6.10 为什么热备份期间产生的 Redo 要比正常的多

还要知道的是, 在数据库处于热备份状态时, 会产生比平常更多的日志。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, 这里还需要简单介绍一下 SPLIT Block 的概念。

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 8KB 的 db_block_size。当热备份数据文件时, 要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 来自于数据块向数据文件 (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。

来看一下测试, 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 512bytes 的日志。然后用 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

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，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，如果数据块大小等于操作系统块大小，则可以设置该参数为 **False**，以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 **Block**，从而可以避免 **Split Block** 的生成，所以不会产生额外的 **Redo**。

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

6.11 能否不生成 Redo

6.11.1 NOLOGGING 对于数据库的影响

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：

- 创建索引或重建索引时；
- 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；
- **CTAS** 方式创建数据表时；
- 大对象（**LOB**）的操作；
- 一些 **ALTER TABLE** 操作，如 **MOVE**、**SPLIT** 等。

关于 **NOLOGGING** 的作用，在 **ITPUB** 论坛上曾经有过深入的讨论，总结起来就是，

NOLOGGING 与表模式 (LOGGING/NOLOGGING)、插入模式 (APPEND/NO APPEND) 及数据库运行模式 (归档/非归档) 都有关系, 具体可归纳如表 6-1 所示。

表 6-1 数据库模式、表模式、插入模式及 Redo 生成的关系

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, 所以这里以归档模式为例, 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'
```

将数据库启动在归档模式:

```
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup mount
ORACLE instance started.

Total System Global Area  235999908 bytes
Fixed Size                  451236 bytes
Variable Size              201326592 bytes
Database Buffers           33554432 bytes
Redo Buffers                667648 bytes
Database mounted.

SQL> alter database archivelog;
```

Database altered.

SQL> alter database open;

Database altered.

测试对于常规表，Redo 的生成，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.
```

更多内容，可以参考 ITPUB 网上的讨论 (<http://www.itpub.net/242761.html>)。

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。

下面通过一个测试来看一下 **NOLOGGING** 对于数据恢复的影响，本测试的环境为：

```
SQL> select * from v$version;
```

```
BANNER
```

```
-----
```

```
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

```
PL/SQL Release 9.2.0.4.0 - Production
```

```
CORE      9.2.0.3.0      Production
TNS for Linux: Version 9.2.0.4.0 - Production
NLSRTL Version 9.2.0.4.0 - Production
```

首先对表空间 eygle 进行热备份:

```
SQL> select name from v$datafile;
```

```
NAME
```

```
-----
/opt/oracle/oradata/conner/system01.dbf
```

```
/opt/oracle/oradata/conner/undotbs1.dbf
```

```
/opt/oracle/oradata/conner/eygle01.dbf
```

```
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
```

由此可见 NOLOGGING 对于数据库的影响。

而如果在 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

```

6.11.2 disable_logging 对于数据库的影响

除了以上常规操作之外，Oracle 还存在一个内部参数，可以使数据库关闭日志记录，从而实现某些特殊需要或测试目的，这个参数是：

```
SQL> @GetHiddenParameter
```

```
Enter value for par: logging
```

```
old 14: x.ksppinm like '%&par%'
```

```
new 14: x.ksppinm like '%logging%'
```

NAME	VALUE	ISDEFAULT	ISMOD	ISADJ

_disable_logging	FALSE	FALSE	FALSE	FALSE

可以动态设置这个参数：

```
SQL> alter system set "_disable_logging"=true;
```

```
System altered.
```

下面通过测试来看一下这个参数的作用及意义，本测试采用的数据库环境为（数据库处于非归档模式）：

```
SQL> select * from v$version;
```

```

BANNER
-----

```

Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production

PL/SQL Release 9.2.0.4.0 - Production

CORE 9.2.0.3.0 Production

TNS for Solaris: Version 9.2.0.4.0 - Production

NLSRTL Version 9.2.0.4.0 - Production

SQL> archive log list;

Database log mode	No Archive Mode
Automatic archival	Enabled
Archive destination	/opt/oracle9/oradata/testora9/archive
Oldest online log sequence	133
Current log sequence	135

首先进行日志切换（此步骤是为了使日志文件中的内容单纯化，并非必须）：

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	ACTIVE	260777420	19-MAR-06	
2	1	134	104857600	1 NO	CURRENT	260779297	19-MAR-06	
3	1	132	104857600	1 YES	INACTIVE	260777060	19-MAR-06	

SQL> select * from v\$logfile;

GROUP#	STATUS	TYPE	MEMBER
1	ONLINE		/opt/oracle9/oradata/testora9/redo01.log
2	ONLINE		/opt/oracle9/oradata/testora9/redo02.log
3	ONLINE		/opt/oracle9/oradata/testora9/redo03.log

在其他 session 修改数据：

SQL> update emp set sal=10 where empno=7788;

1 row updated.

```
SQL> commit;
```

```
Commit complete.
```

转储日志:

```
SQL> ALTER SYSTEM DUMP LOGFILE '/opt/oracle9/oradata/testora9/redo02.log';
```

```
System altered.
```

```
SQL> @gettrcname
```

```
TRACE_FILE_NAME
```

```
-----  
/opt/oracle9/admin/testora9/udump/testora9_ora_5843.trc
```

可以看到, 日志文件中不再记录该事务的 Redo 信息:

```
SQL> ! cat /opt/oracle9/admin/testora9/udump/testora9_ora_5843.trc
```

```
.....
```

```
*** 2006-03-19 21:00:59.139
```

```
*** SESSION ID:(11.5) 2006-03-19 21:00:59.122
```

```
DUMP OF REDO FROM FILE '/opt/oracle9/oradata/testora9/redo02.log'
```

```
Opcodes *.*
```

```
DBA's: (file # 0, block # 0) thru (file # 65534, block # 4194303)
```

```
RBA's: 0x000000.00000000.0000 thru 0xffffffff.ffffffff.ffff
```

```
SCN's scn: 0x0000.00000000 thru scn: 0xffff.ffffffff
```

```
Times: creation thru eternity
```

```
FILE HEADER:
```

```
Software vsn=153092096=0x9200000, Compatibility Vsn=153092096=0x9200000
```

```
Db ID=1628068176=0x610a5950, Db Name='TESTORA9'
```

```
Activation ID=1628067152=0x610a5550
```

```
Control Seq=1026=0x402, File size=204800=0x32000
```

```
File Number=2, Blksiz=512, File Type=2 LOG
```

```
descrip:"Thread 0001, Seq# 0000000134, SCN 0x00000f8b2d21-0xffffffffffff"
```

```
thread: 1 nab: 0xffffffff seq: 0x00000086 hws: 0x1 eot: 1 dis: 0
```

```
reset logs count: 0x1f435110 scn: 0x0000.00000001
```

```
Low scn: 0x0000.0f8b2d21 03/19/2006 20:59:22
```

```

Next scn: 0xffff.ffffffff 01/01/1988 00:00:00
Enabled scn: 0x0000.00000001 04/26/2004 15:56:10
Thread closed scn: 0x0000.0f8b2d21 03/19/2006 20:59:22
Log format vsn: 0x80000000 Disk cksum: 0xc1ed Calc cksum: 0xc1ed
Terminal Recovery Stamp scn: 0x0000.00000000 01/01/1988 00:00:00
Most recent redo scn: 0x0000.00000000
Largest LWN: 0 blocks
End-of-redo stream : No
Unprotected mode
Miscellaneous flags: 0x0
END OF REDO DUMP

```

需要注意的是，由于不记录日志，在进行数据恢复时，这些数据是无法恢复的，请看下面的简单测试。

(1) 创建测试数据：

```
SQL> create table t as select * from dba_users;
```

Table created.

```
SQL> select count(*) from t;
```

```

COUNT(*)
-----
          12

```

(2) Abort 关闭数据库，模拟实例失效。

```
SQL> shutdown abort;
```

ORACLE instance shut down.

(3) 重新启动。经过自动的实例恢复打开之后，发现先前创建的 t 表已不存在。

```
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.
SQL> select count(*) from t;
select count(*) from t
                *
ERROR at line 1:
ORA-00942: table or view does not exist
```

由于未产生相应日志，数据库 Crash 或 shutdown abort 之后，上一次成功完成的检查点之后变化的数据将无法恢复。

(4) 观察 alert 文件。从日志中可以看到在 instance recovery 中，没有数据被恢复，只有成功完成的上次检查点之前数据可以被获取，之后数据都将丢失。

```
Wed Oct 19 20:38:38 2005
Beginning crash recovery of 1 threads
Wed Oct 19 20:38:38 2005
Started first pass scan
Wed Oct 19 20:38:39 2005
Completed first pass scan
    0 redo blocks read, 0 data blocks need recovery
Wed Oct 19 20:38:39 2005
Started recovery at
    Thread 1: logseq 2, block 201, scn 0.897632464
Recovery of Online Redo Log: Thread 1 Group 1 Seq 2 Reading mem 0
    Mem# 0 errs 0: /opt/oracle/oradata/conner/redo01.log
Wed Oct 19 20:38:39 2005
Completed redo application
Wed Oct 19 20:38:39 2005
Ended recovery at
    Thread 1: logseq 2, block 201, scn 0.897652465
0 data blocks read, 0 data blocks written, 0 redo blocks read
Crash recovery completed successfully
```

但是需要注意的是，在 Oracle 9.2.0.6 版本中，设置该参数会触发 Bug 3868748，该 Bug 会导致数据库无法启动。启动时警告日志文件中会记录类似如下错误：

```
ORA-07445: exception encountered:
core dump [kcrfwcint()+1625] [SIGFPE] [Integer divide by zero] [0x828739D] [] []
```

但是好在这个参数是动态的，在存在 Bug 版本中，可以修改数据库当前值来进行测试：

```
SQL> alter system set "_disable_logging"=true scope=memory;

System altered.
```

这个 Bug 的影响范围并非全部，Oracle 9.2.0.6、Oracle 9.2.0.5、Oracle 9.2.0.4 等，都不受这个 Bug 影响。Oracle 声称修正该 Bug 的版本是 Oracle 10gR2:

```
Bug# 3868748 Instance crashes when trying to use "_disable_logging"=true
Fixed: 10.2.0.1
```

另外，如果数据库运行在归档模式下，设置该参数会导致日志文件损坏。因为在设置该参数之后，归档进程无法识别该日志文件格式，会将该日志文件标记为损坏，看以下测试：

设置该参数后，切换日志：

```
SQL> archive log list;

Database log mode          Archive Mode
Automatic archival         Enabled
Archive destination        /opt/oracle/oradata/conner/archive
Oldest online log sequence 20
Next log sequence to archive 23
Current log sequence        23
SQL> alter system switch logfile;

System altered.

SQL> select group#,status from v$Log;

GROUP# STATUS
-----
1 INACTIVE
2 ACTIVE
3 INACTIVE
4 CURRENT
SQL> show parameter disable

NAME                                TYPE        VALUE
-----
_disable_logging                    boolean     TRUE
```

此时可以在警告日志中看到日志损坏的错误信息：

```
Thu Apr 13 23:33:25 2006
ARC0: Evaluating archive   log 2 thread 1 sequence 23
ARC0: Beginning to archive log 2 thread 1 sequence 23
Creating archive destination LOG_ARCHIVE_DEST_1: '/opt/oracle/oradata/conner/archive/1_23.dbf'
ARC0: Log corruption near block 3849 change 0 time ?
```

ARC0: All Archive destinations made inactive due to error 354

Thu Apr 13 23:33:25 2006

Errors in file /opt/oracle/admin/conner/bdump/conner_arc0_21506.trc:

ORA-00354: corrupt redo log block header

ORA-00353: log corruption near block 3849 change 0 time 04/13/2006 21:13:03

ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/conner/redo02.log'

ARC0: Archiving not possible: error count exceeded

ARC0: Failed to archive log 2 thread 1 sequence 23

ARCH: Archival stopped, error occurred. Will continue retrying

Thu Apr 13 23:33:26 2006

ORACLE Instance conner - Archival Error

ARCH: Connecting to console port...

Thu Apr 13 23:33:26 2006

ORA-16038: log 2 sequence# 23 cannot be archived

ORA-00354: corrupt redo log block header

ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/conner/redo02.log'

ARCH: Connecting to console port...

ARCH:

Thu Apr 13 23:33:26 2006

ORA-16038: log 2 sequence# 23 cannot be archived

ORA-00354: corrupt redo log block header

ORA-00312: online log 2 thread 1: '/opt/oracle/oradata/conner/redo02.log'

设置了 `_disable_logging` 参数，可以禁用日志的生成，从而提高某些测试的性能。让我们简单测试一下看看禁用日志可以使得数据库如何加速。

禁用日志情况下：

SQL> connect / as sysdba

Connected.

SQL> shutdown immediate;

Database closed.

Database dismounted.

ORACLE instance shut down.

SQL> startup

ORACLE instance started.

Total System Global Area 286755168 bytes

Fixed Size 731488 bytes

Variable Size 167772160 bytes

```

Database Buffers          117440512 bytes
Redo Buffers              811008 bytes
Database mounted.
Database opened.
SQL> connect eygle/eygle
Connected.
SQL> show parameter disable

```

NAME	TYPE	VALUE

_disable_logging	boolean	TRUE

测试创建 100 万数据表：

```

SQL> create table test as select * from dba_objects where 1=0;

Table created.

SQL> set timing on
SQL> begin
  2  for i in 1 .. 10000 loop
  3  insert into test select * from dba_objects where rownum < 101;
  4  commit;
  5  end loop;
  6  end;
  7  /

```

PL/SQL procedure successfully completed.

Elapsed: 00:00:40.46

```
SQL> truncate table test;
```

Table truncated.

Elapsed: 00:00:52.72

大约时间用了 40 秒，再看正常日志生成下：

```

SQL> connect / as sysdba
Connected.
SQL> alter system set "_disable_logging"=false;

```

System altered.

Elapsed: 00:00:00.05

SQL> shutdown immediate;

Database closed.

Database dismounted.

ORACLE instance shut down.

SQL> startup

ORACLE instance started.

Total System Global Area 286755168 bytes

Fixed Size 731488 bytes

Variable Size 167772160 bytes

Database Buffers 117440512 bytes

Redo Buffers 811008 bytes

Database mounted.

Database opened.

SQL> show parameter disable

NAME	TYPE	VALUE

_disable_logging	boolean	FALSE

SQL> connect eygle/eygle

Connected.

SQL> set timing on

SQL>

SQL> begin

2 for i in 1 .. 10000 loop

3 insert into test select * from dba_objects where rownum < 101;

4 commit;

5 end loop;

6 end;

7 /

PL/SQL procedure successfully completed.

```
Elapsed: 00:01:54.04
SQL>
SQL> truncate table test;

Table truncated.

Elapsed: 00:01:01.56
```

此时大约用时 1 分 54 秒，两者差距大约为 $114 / 40 = 2.85$ 倍。可以看出两者的差距是显著的，大家有兴趣的话可以自己测试一下。

—— 注 意 ——

本节说明仅供参考及测试用，请勿在生产环境中测试，测试前适当做好备份。

6.11.3 FORCE LOGGING（强制日志）模式

当使用 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 Oracle 9iR2 中，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
```

6.12 Redo 故障的恢复

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。

6.12.1 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
```

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:

```
[oracle@jumper bdump]$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 9.2.0.4.0 - Production on Wed May 10 11:07:01 2006
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Connected to an idle instance.
```

```
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，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

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，在下次成功操作后状态会恢复正常）。

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

```

6.12.2 丢失活动或当前日志文件的恢复

Oracle 通过日志文件保证提交成功的数据不丢失。可是在故障中，用户可能损失了当前的 (CURRENT) 日志文件。这又分为两种情况：此时数据库是正常关闭的和此时数据库是异常关闭。

1. 在损失当前日志时，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 Oracle 8i 中可以通过 4.9.1 节中介绍的类似方法进行解决，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 Oracle 9i 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，数据库是异常关闭的

如果在损失当前日志时，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，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，这里不再赘述。

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 提供了一种内部手段可以用于强制性数据库打开、忽略一致性问题。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，这里再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，在第 3 章中曾经提到过 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`，来看一下这个参数的说明：

```
SQL> select ksppinm,ksppdesc from x$ksppi
```

```
2 where ksppinm like '%resetlogs_%';
```

```
KSPPINM
```

```
KSPDESC
```

```
-----
_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，来看一下下面的例子：

```
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'

```

启动失败，日志文件损坏。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，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，直接打开：

```

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;

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.

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;

Database closed.

Database dismounted.

ORACLE instance shut down.

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.
```

幸运的话，数据库就可以成功 Open，如果不幸，则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，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 consistency 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，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。

6.13 诊断案例一：通过 Clear 日志恢复数据库

我收到此次故障时，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。登录数据库进行检查，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/jsbs/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/jsbs/bdump/jsbs_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/jsbs/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 jsbs - 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'

```

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 shs_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
kcrfail: 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

可以看到在其他人进行的多次重启切换过程中，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

并且可以看到归档进程从错误中被释放出来，数据库恢复了正常。

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。

6.14 诊断案例二：日志组过度激活的诊断

这是一个和 Redo 相关的诊断案例。

系统平台：SunOS 5.8 Generic_108528-23 sun4u sparc SUNW, Ultra-Enterprise

数据库版本：8.1.5.0.0

问题描述：响应缓慢，应用请求已经无法返回。

这时登录数据库检查，发现 Redo 日志组除 CURRENT 外都处于 ACTIVE 状态：

```
oracle:/oracle/oracle8>sqlplus "/" as sysdba"
```

```
SQL*Plus: Release 8.1.5.0.0 - Production on Thu Jun 23 18:56:06 2005
```

```
(c) Copyright 1999 Oracle Corporation. All rights reserved.
```

Connected to:

Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production

With the Partitioning and Java options

PL/SQL Release 8.1.5.0.0 - Production

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。

接下来检查一下 DBWR 的繁忙程度：

SQL> !

oracle:/oracle/oracle8>ps -ef|grep ora_

```

oracle 2273    1  0   Mar 31 ?        57:40 ora_smon_hysms02
oracle 2266    1  0   Mar 31 ?        811:42 ora_dbw0_hysms02
oracle 2264    1 16   Mar 31 ?       16999:57 ora_pmon_hysms02
oracle 2268    1  0   Mar 31 ?       1649:07 ora_lgwr_hysms02
oracle 2279    1  0   Mar 31 ?         8:09 ora_snp1_hysms02
oracle 2281    1  0   Mar 31 ?         4:22 ora_snp2_hysms02
oracle 2285    1  0   Mar 31 ?         9:40 ora_snp4_hysms02
oracle 2271    1  0   Mar 31 ?       15:57 ora_ckpt_hysms02
oracle 2283    1  0   Mar 31 ?         5:37 ora_snp3_hysms02
oracle 2277    1  0   Mar 31 ?         5:58 ora_snp0_hysms02
oracle 2289    1  0   Mar 31 ?         0:00 ora_d000_hysms02
oracle 2287    1  0   Mar 31 ?         0:00 ora_s000_hysms02

```

```
oracle 2275      1  0   Mar 31 ?           0:04 ora_reco_hysms02
oracle 21023 21012  0 18:52:59 pts/65    0:00 grep ora_
```

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	16.52%	oracle
2264	oracle	1	0	0	1358M	1316M	run	283.3H	16.36%	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

可以看到 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    16.4    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	huraysms02: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	huraysms02: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	huraysms02: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	436.5	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	huraysms02:vold(pid238)

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 500KB/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)

根据常识，T3 盘阵通常按 Char 写速度可以达到 10MB/s 左右，以前测试过一些 TPCC

指标,具体可以参考我网站上的相关文章:Use bonnie to Test system IO speed(<http://www.eygle.com/unix/Use.Bonnie++.To.Test.IO.speed.htm>)。

而正常情况下的数据库随机写通常都在 1~2MB/s 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。

为了保证数据库中多个用户间的读一致性和能够回退事务，Oracle 必须拥有一种机制，能够为变更的数据构造一种前镜像（Before Image）数据（保存修改之前的旧值），以保证能够回滚或撤销对数据库所作的修改。这就是回滚（或撤销）。

在上一章中提到 Redo，如果说 Redo 是用来保证在故障时事务可以被恢复；那么 Undo 则是用来保证事务可以被回退或者撤销。本章将着重介绍回滚（Rollback）与撤销（Undo）方面的知识。

7.1 什么是回滚和撤销

首先介绍一下什么是回滚和撤销。从 Oracle 6 版本到 Oracle 9i 版本，Oracle 用数据库中的回滚段（Rollback）来提供撤销数据（Undo Data）；而从 Oracle 9i 开始，Oracle 还提供了一种新的撤销数据（Undo Data）管理方式，就是使用 Oracle 自动管理的撤销（Undo）表空间（Automatic Undo Management，通常可以被缩写为 AUM）。

事务使用回滚段来记录变化前的数据或者撤销信息，让我们回忆一下上一章中讲过的一个例子。假定发出了一个更新语句：

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

来看一下这个语句是怎样执行的（为了叙述方便，这里尽量简化了情况）：

- （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）。

注意到在一个事务的进行过程中，Redo 和 Undo 是交替出现的，而且对于数据库来说都非常重要。

在以上步骤中，对于回滚段的操作存在多处，在事务开始时，首先需要在回滚表空间获得一个事务槽，分配空间，然后创建前镜像，此后事务的修改才能进行，Oracle 必须以此来保证事务是可以回退的。

如果用户提交（Commit）了事务，Oracle 会在日志文件记录提交，并且写出日志，同时会在回滚段中把该事务标记为已提交；如果用户回滚（Rollback）事务，则 Oracle 需要从回滚段中把前镜像数据读取出来，修改数据缓冲区，完成回滚，这个过程本身也要产生 Redo，所以回退这个操作是很昂贵的。

在 Oracle 的性能优化中，有一个性能指标称为平均事务回滚率（Rollback per transaction），用来衡量数据库的提交与回滚效率。在 Statspack 的报告中，可以从开头部分找到这个指标：

```
Load Profile
~~~~~
.....
% Blocks changed per Read:    0.37    Recursive Call %:    1.14
Rollback per transaction %:   38.22    Rows per Sort:    11.83
```

该参数计算公式如下：

$$\text{Round}(\text{User rollbacks} / (\text{user commits} + \text{user rollbacks}), 4) * 100\%$$

其中 user commits 和 user rollbacks 数据来自系统的统计信息，可以从 V\$SYSSTAT 视图中得到，在 Statspack 中也包含着部分数据的输出，本例选择的报告相关部分摘录如下：

```
.....
user commits                31,910            12.9            0.6
user rollbacks              19,740            8.0            0.4
.....
```

按照公式计算就可以得出平均事务回滚率：

$$\text{Round}(19740 / (31910 + 19740), 4) = .3822$$

这个指标应该接近于 0，如果该指标过高，则说明数据库的回滚过多。回滚过多不仅说明数据库经历了太多的无效操作，而且这些操作会极大影响数据库性能。

7.2 回滚段存储的内容

在上一章中讲过，Redo 中只会记录少量信息，这些信息足以重演事务；同样 Undo 中也只记录精简信息，这些信息足以撤销事务。

- 对于 INSERT 操作，回滚段只需要记录插入记录的 rowid，如果回退，只需将该记录根据 rowid 删除即可。
- 对于 UPDATE 操作，回滚段只需要记录被更新字段的旧值即可（前镜像），回退时通过旧值覆盖新值即可完成回退。
- 对于 DELETE 操作，Oracle 则必须记录整行的数据，在回退时，Oracle 通过一个反句操作恢复删除的数据。

通过以上介绍可以简单总结一下，对于相同数据量的数据操作，通常 INSERT 产生最少的 Undo，UPDATE 产生的 Undo 居中，而 DELETE 操作产生的 Undo 最多。这也就是我们经常看到的：当一个大的 DELETE 操作失败或者回滚，总是需要很长的时间，并且会有大量的 Redo 生成。所以通常在进行大规模数据删除操作时，推荐通过分批删除分次提交，以减少对于回滚段的占用和冲击。

7.3 并发控制和一致性读

允许多用户并发访问是数据库必需满足的功能，那么怎样实现并发访问、控制和数据修改就成为了一个重要问题。

一方面 Oracle 通过锁定机制实现数据库的并发控制；一方面通过多版本(Multi-versioning Model) 模型来进行并发数据访问。通过多版本架构，Oracle 实现了读取和写入的分离，使得写入不阻塞读取；读取不阻塞修改。这是 Oracle 数据库区别于其他数据库的一个重要特征。

多版本模型在 Oracle 数据库中是通过一致性读来实现的，一致性读也正是回滚表空间的主要作用之一。

Oracle 一方面不允许其他用户读取未提交数据，一方面要保证用户读取的数据要来自同一时间点。通过图 7-1 来看一下什么是 Oracle 的一致性读取 (Consistent Reads)。

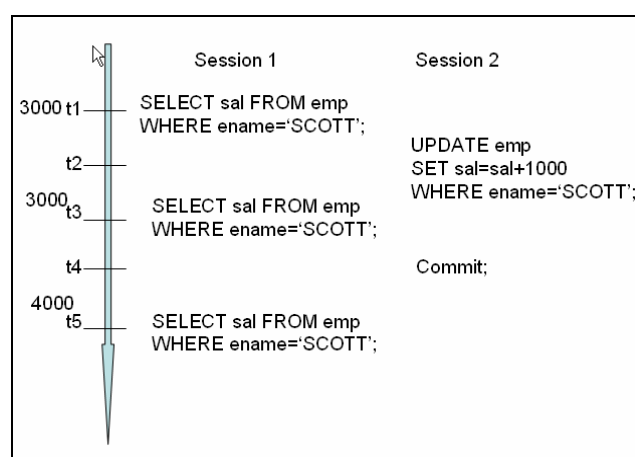


图 7-1 一致性读取示意图

假定员工 Scott 的薪水为 3000，那么：

- (1) 在 T1 时间，在 Session1 查询可以得到这个结果；
- (2) 在 T2 时间 Session2 进行更新，将 SCOTT 的薪水增加 3000，并未提交；
- (3) 在 T3 时间 Session1 再次查询，注意此时，Oracle 不会允许其他用户看到未提交数据，所以此时，Oracle 需要通过回滚段记录的前镜像进行一致性读，将 3000 恢复出来提供给用户，这是一致性读的作用；
- (4) 在 T4 时间，Session2 提交该更改，此时数据修改被永久化；
- (5) 在 T5 时间，其他用户再次查询时，将会看到变化后的数据，也就是“4000”。

在上文曾经提到，Oracle 内部使用 SCN 作为数据库时钟，这里查询结果集就是根据 SCN

来进行判断的，每个数据块头部都会记录一个提交 SCN，当数据更改提交后，提交 SCN 同时被修改，这个 SCN 在查询时可以用来进行一致性读判断。

在图 7-1 中，假定查询开始的时间为 T1，则在查询获取的数据块中，如果数据块的提交 SCN 小于 T1，则 Oracle 接受该数据，如果提交 SCN 大于 T1 或者数据被锁定修改尚未记录 COMMIT SCN，则 Oracle 需要通过回滚段构造前镜像来返回结果，这就是一致性读的本质含义。

7.4 回滚段的前世今生

在 Oracle 9i 之前，回滚表空间创建之后，Oracle 随后创建回滚段供数据库使用，也可以手工创建或者删除回滚段进行维护，在开始事务之前，也可以通过如下命令指定用户想要使用的回滚段：

```
set transaction use rollback segment <rollback_segment_name>;
```

以下是从 Oracle 8i 数据库创建日志摘录的部分信息，这就是 Oracle 8i 的基本管理方式：

Sat Apr 24 16:27:23 2004

```
CREATE TABLESPACE RBS DATAFILE '/data1/oracle/oradata/8.1.7/rbs01.dbf' SIZE 256M REUSE  
AUTOEXTEND ON NEXT 5120K
```

```
MINIMUM EXTENT 512K
```

```
DEFAULT STORAGE ( INITIAL 512K NEXT 512K MINEXTENTS 8 MAXEXTENTS UNLIMITED )
```

Completed: CREATE TABLESPACE RBS DATAFILE '/data1/oracle/oradata/8.1.7/rbs01.dbf'

....

Sat Apr 24 16:27:36 2004

```
CREATE PUBLIC ROLLBACK SEGMENT RBS0 TABLESPACE RBS
```

```
STORAGE ( OPTIMAL 4096K )
```

Completed: CREATE PUBLIC ROLLBACK SEGMENT RBS0 TABLESPACE RBS

Sat Apr 24 16:27:36 2004

```
CREATE PUBLIC ROLLBACK SEGMENT RBS1 TABLESPACE RBS
```

```
STORAGE ( OPTIMAL 4096K )
```

Completed: CREATE PUBLIC ROLLBACK SEGMENT RBS1 TABLESPACE RBS

.....

Sat Apr 24 16:27:37 2004

```
CREATE PUBLIC ROLLBACK SEGMENT RBS11 TABLESPACE RBS
```

```
STORAGE ( OPTIMAL 4096K )
```

Completed: CREATE PUBLIC ROLLBACK SEGMENT RBS11 TABLESPACE RB

Sat Apr 24 16:27:37 2004

```
ALTER ROLLBACK SEGMENT "RBS0" ONLINE
```

Completed: ALTER ROLLBACK SEGMENT "RBS0" ONLINE

```

Sat Apr 24 16:27:37 2004
ALTER ROLLBACK SEGMENT "RBS1" ONLINE
Completed: ALTER ROLLBACK SEGMENT "RBS1" ONLINE
.....
Sat Apr 24 16:27:37 2004
ALTER ROLLBACK SEGMENT "RBS10" ONLINE
Completed: ALTER ROLLBACK SEGMENT "RBS10" ONLINE
Sat Apr 24 16:27:37 2004
ALTER ROLLBACK SEGMENT "RBS11" ONLINE
Completed: ALTER ROLLBACK SEGMENT "RBS11" ONLINE

```

可以从数据库中查询这些回滚段的状态：

```

SQL> col segment_name for a10
SQL> select segment_name,tablespace_name,status from dba_rollback_segs;

SEGMENT_NAME TABLESPACE_NAME          STATUS
-----
SYSTEM        SYSTEM                        ONLINE
RBS0          RBS                            ONLINE
RBS1          RBS                            ONLINE
.....
RBS10         RBS                            ONLINE
RBS11         RBS                            ONLINE

13 rows selected.

```

从 Oracle 9i 开始，Oracle 引入了自动管理的 Undo 表空间，如果选择使用自动的 Undo 表空间的管理，那么用户不再能够创建或删除回滚段，也不再需要为事务指定回滚段，这一切将由 Oracle 自动进行。

```

SQL> select * from v$version where rownum <2;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

SQL> show parameter undo

NAME                                TYPE          VALUE
-----

```

undo_management	string	AUTO
undo_retention	integer	10800
undo_suppress_errors	boolean	FALSE
undo_tablespace	string	UNDOTBS1

SQL>

伴随自动的 Undo 管理功能的引入，Oracle 随之引入了几个新的初始化参数。

- **undo_management**: 用来定义数据库使用的回滚段是否使用自动管理模式。该参数有两个可选项，AUTO 表示自动管理，MANUAL 表示手工管理。
- **undo_tablespace**: 用来定义在自动管理模式下，当前实例使用哪个 Undo 表空间。
- **undo_suppress_errors**: 表示当使用自动管理模式时，如果使用不再支持的操作时（如为事务指定回滚段）是否返回出错信息。设置为 True 时不返回出错信息，操作无效但是可以继续，设置为 False 时，则操作不能继续，这实际上是一个向后兼容的参数。

```
SQL> set transaction use rollback segment rbs1;
```

```
set transaction use rollback segment rbs1
```

```
*
```

```
ERROR 位于第 1 行:
```

```
ORA-30019: 自动撤销模式中的回退段操作非法
```

```
SQL> show parameter undo_suppress_errors
```

NAME	TYPE	VALUE
undo_suppress_errors	boolean	FALSE

```
SQL> set transaction use rollback segment rbs1;
```

```
set transaction use rollback segment rbs1
```

```
*
```

```
ERROR 位于第 1 行:
```

```
ORA-30019: 自动撤销模式中的回退段操作非法
```

```
SQL> alter session set undo_suppress_errors=true;
```

会话已更改。

```
SQL> set transaction use rollback segment rbs1;
```

事务处理集。

该参数在 Oracle 10g 中已经被舍弃：

```
SQL> select * from v$version where rownum <2;
```

```
BANNER
```

```
-----  
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - 64bi
```

```
SQL> show parameter undo
```

NAME	TYPE	VALUE
undo_management	string	AUTO
undo_retention	integer	100000
undo_tablespace	string	UNDOTBS1

■ **undo_retention**: 表示在自动管理模式下，当回滚段变得非激活之后，回滚段中的数据在被覆盖前保留的时间，该参数单位是秒。在 Oracle 9iR2 中，这个参数的缺省值为 10800 秒，也就是 3 个小时。通过该参数的调节作用，在繁忙的查询系统中，可以有效地避免 ORA-01555 错误，这是 Oracle 9i AUM 的一大增强。

在自动管理的 Undo 表空间下，回滚段的个数是 Oracle 根据数据库的繁忙程度自动分配或者回收的，缺省情况下，数据库创建时初始化 10 个回滚段：

```
Mon Apr 26 15:56:27 2004  
CREATE UNDO TABLESPACE UNDOTBS1 DATAFILE '/opt/oracle9/oradata/testora9/undotbs01.dbf'  
SIZE 200M REUSE AUTOEXTEND ON NEXT 5  
120K MAXSIZE UNLIMITED  
Mon Apr 26 15:56:33 2004  
Mon Apr 26 15:56:33 2004  
Created Undo Segment _SYSSMU1$  
Created Undo Segment _SYSSMU2$  
Created Undo Segment _SYSSMU3$  
Created Undo Segment _SYSSMU4$  
Created Undo Segment _SYSSMU5$  
Created Undo Segment _SYSSMU6$  
Created Undo Segment _SYSSMU7$  
Created Undo Segment _SYSSMU8$  
Created Undo Segment _SYSSMU9$  
Created Undo Segment _SYSSMU10$  
Undo Segment 1 Online  
Undo Segment 2 Online  
Undo Segment 3 Online
```

```

Undo Segment 4 Onlined
Undo Segment 5 Onlined
Undo Segment 6 Onlined
Undo Segment 7 Onlined
Undo Segment 8 Onlined
Undo Segment 9 Onlined
Undo Segment 10 Onlined
Successfully onlined Undo Tablespace 1.

```

也可以从数据库中查询得到：

```
SQL> select * from v$rollname;
```

```

      USN NAME
-----
0 SYSTEM
1 _SYSSMU1$
2 _SYSSMU2$
3 _SYSSMU3$
4 _SYSSMU4$
5 _SYSSMU5$
6 _SYSSMU6$
7 _SYSSMU7$
8 _SYSSMU8$
9 _SYSSMU9$
10 _SYSSMU10$

```

```
11 rows selected.
```

在系统繁忙时，可以从数据库的 `alert_<SID>.log` 文件中看到回滚段的动态创建和释放过程：

```

Tue Mar 21 00:06:51 2006
Created Undo Segment _SYSSMU11$
Tue Mar 21 00:06:51 2006
Undo Segment 11 Onlined
Tue Mar 21 00:06:52 2006
Created Undo Segment _SYSSMU12$
Undo Segment 12 Onlined
Tue Mar 21 00:06:53 2006
Created Undo Segment _SYSSMU13$

```

```

Tue Mar 21 00:06:54 2006
Created Undo Segment _SYSSMU14$
Tue Mar 21 00:06:54 2006
Undo Segment 13 Online
Tue Mar 21 00:06:54 2006
Undo Segment 14 Online
.....
Created Undo Segment _SYSSMU34$
Undo Segment 34 Online
.....
Tue Mar 21 03:47:39 2006
SMON offlining US=11
SMON offlining US=12
SMON offlining US=13
SMON offlining US=14
.....
SMON offlining US=33
SMON offlining US=34

```

动态创建和释放，这也正是自动管理的 Undo 表空间的优势之一。

7.5 回滚机制的深入研究

如果大家有兴趣深入了解一下回滚段的机制，那么请跟随我将前面的例子进一步深化。

1. 从 DML 更新事务开始

重新来看这个更新语句：

```

SQL> connect scott/tiger
Connected.
SQL> select * from emp;

```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK		7902 17-DEC-80	800		20
7499	ALLEN	SALESMAN		7698 20-FEB-81	1600	300	30
7521	WARD	SALESMAN		7698 22-FEB-81	1250	500	30
7566	JONES	MANAGER		7839 02-APR-81	2975		20

7654 MARTIN	SALESMAN	7698 28-SEP-81	1250	1400	30
7698 BLAKE	MANAGER	7839 01-MAY-81	2850		30
7782 CLARK	MANAGER	7839 09-JUN-81	2450		10
7788 SCOTT	ANALYST	7566 19-APR-87	3000		20
7839 KING	PRESIDENT	17-NOV-81	5000		10
7844 TURNER	SALESMAN	7698 08-SEP-81	1500	0	30
7876 ADAMS	CLERK	7788 23-MAY-87	1100		20
7900 JAMES	CLERK	7698 03-DEC-81	950		30
7902 FORD	ANALYST	7566 03-DEC-81	3000		20
7934 MILLER	CLERK	7782 23-JAN-82	1300		10

14 rows selected.

SQL> UPDATE emp SET sal = 4000 Where empno= 7788;

1 row updated.

SQL> select * from emp where empno=7788;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	SCOTT	ANALYST	7566	19-APR-87	4000		20

先不提交这个事务，在另外窗口新开 Session，使用 SYS 用户查询相关信息，进行进一步的分析研究。

2. 获得事务信息

从事务表中可以获得关于这个事务的信息，该事务位于 6 号回滚段（XIDUSN），在 6 号回滚段上，该事务位于第 23 号事务槽（XIDSLOT）：

SQL> SELECT xidusn, xidslot, xidsqn, ubablk, ubafil, ubarec FROM v\$transaction;					
XIDUSN	XIDSLOT	XIDSQN	UBABLK	UBAFIL	UBAREC
6	23	14030	85	2	63

从 V\$ROLLSTAT 视图中也可以获得事务信息，XACTS 字段代表的是活动事务的数量，同样看到该事务位于 6 号回滚段：

SQL> select usn,writes,rssize,xacts,hwmsize,shrinks,wraps from v\$rollstat;						
USN	WRITES	RSSIZE	XACTS	HWMSIZE	SHRINKS	WRAPS

0	4680	385024	0	385024	0	0
1	20674	1171456	0	1171456	0	0
2	29240	1171456	0	1171456	0	0
3	31652	1171456	0	1171456	0	0
4	22968	1171456	0	1171456	0	0
5	30406	1171456	0	1171456	0	0
6	31282	1171456	1	1171456	0	0
7	21510	1171456	0	1171456	0	0
8	28472	1171456	0	1171456	0	1
9	69830	1171456	0	1171456	0	0
10	23328	1171456	0	1171456	0	0

11 rows selected.

3. 获得回滚段名称并转储段头信息

查询 V\$ROLLNAME 视图获得回滚段名称，并转储回滚段头信息：

SQL> select * from v\$rollname a where a.usn=6;
USN NAME

6 _SYSSMU6\$
SQL> alter system dump undo header '_SYSSMU6\$';
System altered

生成的跟踪文件如下：

SQL> @gettrcname
TRACE_FILE_NAME

/opt/oracle/admin/conner/udump/conner_ora_15309.trc

4. 获得跟踪文件信息

注意这就是前边多次提到过的回滚段头的信息，其中包括事务表信息，从以下的跟踪文件中，可以清晰地看到这些内容：

Undo Segment: _SYSSMU6\$ (6)

Extent Control Header

Extent Header:: spare1: 0 spare2: 0 #extents: 2 #blocks: 15
 last map 0x00000000 #maps: 0 offset: 4080
Highwater:: 0x00800055 ext#: 1 blk#: 4 ext size: 8
#blocks in seg. hdr's freelists: 0
#blocks below: 0
mapblk 0x00000000 offset: 1
Unlocked
Map Header:: next 0x00000000 #extents: 2 obj#: 0 flag: 0x40000000

Extent Map

0x0080004a length: 7
0x00800051 length: 8

Retention Table

Extent Number:0 Commit Time: 1144779956
Extent Number:1 Commit Time: 1143540568

TRN CTL:: seq: 0x02de chd: 0x0025 ctl: 0x0023 inc: 0x00000000 nfb: 0x0000
mgc: 0x8201 xts: 0x0068 flg: 0x0001 opt: 2147483646 (0x7fffffe)
uba: 0x00800055.02de.3d scn: 0x0819.003f5d3e

Version: 0x01

FREE BLOCK POOL::

uba: 0x00000000.02de.3c ext: 0x1 spc: 0x30a
uba: 0x00000000.02de.22 ext: 0x1 spc: 0x144e
uba: 0x00000000.02d8.18 ext: 0x6 spc: 0x1206
uba: 0x00000000.0000.00 ext: 0x0 spc: 0x0
uba: 0x00000000.0000.00 ext: 0x0 spc: 0x0

TRN TBL::

index state cflags wrap# uel scn dba parent-xid

<为了排版关系，省略了一些条目，并且截去了最后一个列的信息，完整内容可以从作者的网站得到>

0x14	9	0x00	0x36ce	0x0016	0x0819.00402706	0x00800055	0x0000.000.00000000
0x15	9	0x00	0x36ce	0x001b	0x0819.004036d1	0x00800055	0x0000.000.00000000
0x16	9	0x00	0x36ce	0x0019	0x0819.00402af9	0x00800055	0x0000.000.00000000
0x17	10	0x80	0x36ce	0x0001	0x0819.004066aa	0x00800055	0x0000.000.00000000
0x18	9	0x00	0x36ce	0x0015	0x0819.004032df	0x00800055	0x0000.000.00000000
0x19	9	0x00	0x36ce	0x0018	0x0819.00402eec	0x00800055	0x0000.000.00000000
0x1a	9	0x00	0x36ce	0x001c	0x0819.00403eb9	0x00800055	0x0000.000.00000000
.....							

回顾前面的事务信息，该事务正好占用的是第 23 号事务槽（0x17），状态（State）为 10 代表的是活动事务。

5. 转储前镜像信息

再来看 DBA（Data Block Address），这个 DBA 指向的就是包含这个事务的前镜像的数据块地址 **0x00800055**。

来看一下这个地址如何换算。DBA 代表数据块的存储地址，由 10 位文件号和 22 位数据块（Block）组成。将 **0x00800055** 转换为二进制就是 0000 0000 1000 0000 0000 0000 0101 0101。

前 10 位代表文件号为 2，后 22 位代表 Block 号为 85。经过转换后，该前镜像信息位于 file 2 block 85。这和从事务表中查询得到的数据完全一致：

```
SQL> SELECT xidusn, xidslot, xidsqn, ubablk, ubafil, ubarec FROM v$transaction;
```

XIDUSN	XIDSLOT	XIDSQN	UBABLK	UBAFIL	UBAREC
-----	-----	-----	-----	-----	-----
6	23	14030	85	2	63

—— 提 示 ——

很多深入研究的内容在数据库内部都有完整的体现，不过通常我们很少注意，只有将两者结合起来学习、研究和理解，我们才能深刻地理解到 Oracle 的本质。希望大家在阅读这部分内容的时候能够耐心、细致，有所收获。

为了同时说明一些其他内容，继续先前 Scott 用户的事务，再更新 2 条记录：

```
SQL> update emp set sal=4000 where empno=7788;
```

1 row updated.

```
SQL> update emp set sal=4000 where empno=7782;
```

1 row updated.

```
SQL> update emp set sal=4000 where empno=7698;
```

```
1 row updated.
```

将回滚段中的这个 Block 转储出来：

```
SQL> alter system dump datafile 2 block 85;
```

```
System altered
```

这是跟踪文件开始部分的信息：

```
Start dump data blocks tsn: 1 file#: 2 minblk 85 maxblk 85
```

```
buffer tsn: 1 rdba: 0x00800055 (2/85)
```

```
scn: 0x0819.004066c8 seq: 0x01 flg: 0x00 tail: 0x66c80201
```

```
frmt: 0x02 chkval: 0x0000 type: 0x02=KTU UNDO BLOCK
```

```
*****
```

```
UNDO BLK:
```

```
xid: 0x0006.017.000036ce seq: 0x2de cnt: 0x3f irb: 0x3f icl: 0x0 flg: 0x0000
```

注意，这部分信息中有一个参数 **irb:0x3f**，irb 指的是回滚段中记录的最近的未提交变更开始之处，如果开始回滚，这是起始的搜索点。

接下来是回滚信息的偏移量，最后一个偏移地址正是 0x3f 的信息：

Rec Offset	Rec Offset	Rec Offset	Rec Offset	Rec Offset

0x01 0x1f98	0x02 0x1f54	0x03 0x1efc	0x04 0x1ea4	0x05 0x1e54
0x06 0x1e10	0x07 0x1dbc	0x08 0x1d64	0x09 0x1d14	0x0a 0x1cd0
0x0b 0x1c74	0x0c 0x1c1c	0x0d 0x1bcc	0x0e 0x1b88	0x0f 0x1b30
0x10 0x1ad8	0x11 0x1a88	0x12 0x1a44	0x13 0x19f0	0x14 0x1998
0x15 0x1948	0x16 0x1904	0x17 0x18ac	0x18 0x1854	0x19 0x1804
0x1a 0x17c0	0x1b 0x1768	0x1c 0x1710	0x1d 0x16c0	0x1e 0x167c
0x1f 0x1624	0x20 0x15cc	0x21 0x157c	0x22 0x14a4	0x23 0x13fc
0x24 0x1354	0x25 0x12ac	0x26 0x1204	0x27 0x115c	0x28 0x10b4
0x29 0x100c	0x2a 0x0f64	0x2b 0x0ebc	0x2c 0x0e14	0x2d 0x0d6c
0x2e 0x0cc4	0x2f 0x0c1c	0x30 0x0b74	0x31 0x0acc	0x32 0x0a24
0x33 0x097c	0x34 0x08d4	0x35 0x082c	0x36 0x0784	0x37 0x06dc
0x38 0x0634	0x39 0x058c	0x3a 0x04e4	0x3b 0x043c	0x3c 0x0394
0x3d 0x0310	0x3e 0x02b4	0x3f 0x0258		

```
*-----
```

从接下来的信息中找到 0x3f 信息：

```
*-----
* Rec #0x3f  slt: 0x17  objn: 7961(0x00001f19)  objd: 7961  tblspc: 0(0x00000000)
*      Layer:  11 (Row)   opc: 1   rci 0x3e
Undo type:  Regular undo   Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
rdba: 0x00000000
*-----
KDO undo record:
KTB Redo
op: 0x02  ver: 0x01
op: C  uba: 0x00800055.02de.3e
KDO Op code: URP row dependencies Disabled
      xtype: XA  bdba: 0x00405c5a  hdba: 0x00405c59
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 5(0x5) flag: 0x2c lock: 0 ckix: 0
ncol: 8 nnew: 1 size: 1
col  5: [ 3]  c2 1d 33
```

c2 1d 33 转换为十进制就是 2850（关于数字值的内部存储及转换方式请参考本章结尾的相关部分）。这是最后更新记录的前镜像，Oracle 就是这样通过回滚段保留前镜像信息的：

```
update emp set sal=4000 where empno=7698;
```

在这条 Undo 记录上，还记录一个数据 rci，该参数代表的就是 Undo Chain（同一事务中的多次修改，根据 Chain 链接关联）的下一个偏移量，此处为 0x3e，找到 0x3e 这条 Undo 记录：

```
*-----
* Rec #0x3e  slt: 0x17  objn: 7961(0x00001f19)  objd: 7961  tblspc: 0(0x00000000)
*      Layer:  11 (Row)   opc: 1   rci 0x3d
Undo type:  Regular undo   Last buffer split:  No
Temp Object:  No
Tablespace Undo:  No
rdba: 0x00000000
*-----
KDO undo record:
KTB Redo
op: 0x02  ver: 0x01
op: C  uba: 0x00800055.02de.3d
```

```
KDO Op code: URP row dependencies Disabled
  xtype: XA  bdba: 0x00405c5a  hdba: 0x00405c59
itli: 2  ispac: 0  maxfr: 4863
tabn: 0 slot: 6(0x6) flag: 0x2c lock: 0 ckix: 0
ncol: 8 nnew: 1 size: 1
col 5: [ 3]  c2 19 33
```

这里记录的 c2 19 33 转换为十进制就是 2450，是第二条更新的数据：

```
update emp set sal=4000 where empno=7782;
```

这里的 rci 指向下一条记录 0x3d，找到 0x3d：

```
*-----
* Rec #0x3d  slt: 0x17  objn: 7961(0x00001f19)  objd: 7961  tblspc: 0(0x00000000)
*          Layer: 11 (Row)  opc: 1  rci 0x00
Undo type: Regular undo  Begin trans  Last buffer split: No
Temp Object: No
Tablespace Undo: No
rdba: 0x00000000
*-----
uba: 0x00800055.02de.3c ctl max scn: 0x0819.003f594b prv tx scn: 0x0819.003f5d3e
KDO undo record:
KTB Redo
op: 0x04  ver: 0x01
op: L  itl: xid: 0x0002.01d.000038ea uba: 0x008000c3.04b1.0c
                        flg: C---  lkc: 0  scn: 0x0819.0036f14f
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
```

这里 c2 1f 转换为十进制是 3000，正是第一条更新的记录：

```
update emp set sal=4000 where empno=7788;
```

这是这个事务中最后一条更新的数据，所以其 Undo Chain 的指针为 0x00，表示这是最后一条记录。也可以从 x\$bh 中找到这些数据块：

```
SQL> select b.segment_name,a.file#,a.dbarfil,a.dbablk,a.class,a.state
2  from x$bh a,dba_extents b
3  where b.RELATIVE_FNO = a.dbarfil
```

```

4  and b.BLOCK_ID <= a.dbablk and b.block_id + b.blocks > a.dbablk
5  and b.owner='SCOTT' and b.segment_name='EMP'
6  /

```

SEGMENT_NAME	FILE#	DBARFIL	DBABLK	CLASS	STATE
EMP	1	1	23641	4	1
EMP	1	1	23642	1	1

注意 class 为 4 的是段头，class 为 1、块号为 23642 的为数据块。如果此时在其他进程查询 scott.emp 表，Oracle 需要构造一致性读，通过前镜像把变化前的数据展现给用户：

```
SQL> select * from scott.emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800.00		20
7499	ALLEN	SALESMAN	7698	1981-2-20	1600.00	300.00	30
7521	WARD	SALESMAN	7698	1981-2-22	1250.00	500.00	30
7566	JONES	MANAGER	7839	1981-4-2	2975.00		20
7654	MARTIN	SALESMAN	7698	1981-9-28	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	1981-5-1	2850.00		30
7782	CLARK	MANAGER	7839	1981-6-9	2450.00		10
7788	SCOTT	ANALYST	7566	1987-4-19	3000.00		20
7839	KING	PRESIDENT		1981-11-17	5000.00		10
7844	TURNER	SALESMAN	7698	1981-9-8	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1987-5-23	1100.00		20
7900	JAMES	CLERK	7698	1981-12-3	950.00		30
7902	FORD	ANALYST	7566	1981-12-3	3000.00		20
7934	MILLER	CLERK	7782	1982-1-23	1300.00		10

14 rows selected

再来查询：

```

SQL> select b.segment_name,a.file#,a.dbarfil,a.dbablk,a.class,a.state,decode(bitand(flag,1), 0, 'N', 'Y') DIRTY
2  from x$bh a,dba_extents b
3  where b.RELATIVE_FNO = a.dbarfil
4  and b.BLOCK_ID <= a.dbablk and b.block_id + b.blocks > a.dbablk
5  and b.owner='SCOTT' and b.segment_name='EMP'
6  /

```

SEGMENT_NAME	FILE#	DBARFIL	DBABLK	CLASS	STATE	DIRTY
-----	-----	-----	-----	-----	-----	-----
EMP	1	1	23641	4	1	N
EMP	1	1	23642	1	3	Y
EMP	1	1	23642	1	1	N

注意到此时，Buffer Cache 中多出一个数据块，也就是 23642 存在 2 份，其中 state 为 3 的就是一致性读构造的前镜像。

6. 转储数据块信息

在前镜像信息中，Oracle 还记录了前镜像对应的数据块的地址，可以从 bdba 记录中获得这部分信息，以先前的一个数据为例，bdba:0x00405c5a 记录了更改的数据块的地址，0x00405c5a 经过转换为二进制就是 0000 0000 0100 0000 0101 1100 0101 1010，也正是 file 1 block 23642。

再将数据表中的 Block 转储出来，看看其中记录了什么样的信息：

```
SQL> alter system dump datafile 1 block 23642;

System altered.

检查跟踪文件，获取数据块信息：

Start dump data blocks tsn: 0 file#: 1 minblk 23642 maxblk 23642
buffer tsn: 0 rdba: 0x00405c5a (1/23642)
scn: 0x0819.00406ddf seq: 0x01 flg: 0x04 tail: 0x6ddf0601
frmt: 0x02 chkval: 0x6b6a type: 0x06=trans data
Block header dump:  0x00405c5a
Object id on Block? Y
seg/obj: 0x1f19 csc: 0x819.406ddf itc: 2 flg: O typ: 1 - DATA
    fsl: 0   fnx: 0x0 ver: 0x01

    Itl          Xid          Uba          Flag  Lck          Scn/Fsc
    ---          -          -          ---  ---          -
    0x01    0x0009.02e.000036b0  0x00800081.0302.11  C---    0   scn 0x0819.0036f205
    0x02    0x0006.017.000036ce  0x00800055.02de.3f  ----    3   fsc 0x0002.00000000
```

这里存在 ITL 事务槽信息，ITL 事务槽指 Interested Transaction List (ITL)，事务必须获得一个 ITL 事务槽才能够进行数据修改。ITL 内容主要包括：

- Xid——Transaction ID;
- Uba——Undo Block Address;
- Lck——Lock Status。

—— 注 意 ——
Xid=Undo.Segment.Number+Transaction.Table.Slot.Number+Wrap

在以上输出中，看到 itl2 (0x02) 上存在活动事务。将 Xid=0x0006.017.000036ce 分解一下。该事务指向 6 号回滚段，Slot 号为 0x17（转换为十进制正好是 23），Wrap#为 36ce，正是 Dump 回滚段看到的那个事务。

index	state	cflags	wrap#	uel	scn	dba	parent-xid	nub
itmt_num								

.....								
0x17	10	0x80	0x36ce	0x0001	0x0819.004066aa	0x00800055	0x0000.000.00000000	
0x00000001	0x00000000							

可以看到，在数据块上同样存在指向回滚段的事务信息。

Uba 代表的是 Undo Block Address，指向具体的回滚段，可以看到该 ITL 上 uba=0x00800055.02de.3f，将这个 UBA 进行分解：

- 0x00800055 正是前镜像的地址；
- seq: 02de 是顺序号；
- 3f 是 UNDO 记录的开始地址（irb 信息）。

Uba 的内容和 Undo 中的信息完全相符：

UNDO BLK:

xid: 0x0006.017.000036ce seq: 0x2de cnt: 0x3f irb: 0x3f icl: 0x0 flg: 0x0000

继续向下可以找到这 3 条被修改的记录，锁定位信息 LB 指向 0x2 号 ITL 事务槽：

```
tab 0, row 5, @0x1d19
tl: 40 fb: --H-FL-- lb: 0x2 cc: 8
col 0: [ 3] c2 4d 63
col 1: [ 5] 42 4c 41 4b 45
col 2: [ 7] 4d 41 4e 41 47 45 52
col 3: [ 3] c2 4f 28
col 4: [ 7] 77 b5 05 01 01 01 01
col 5: [ 2] c2 29
col 6: *NULL*
col 7: [ 2] c1 1f
tab 0, row 6, @0x1d41
tl: 40 fb: --H-FL-- lb: 0x2 cc: 8
col 0: [ 3] c2 4e 53
col 1: [ 5] 43 4c 41 52 4b
col 2: [ 7] 4d 41 4e 41 47 45 52
col 3: [ 3] c2 4f 28
col 4: [ 7] 77 b5 06 09 01 01 01
col 5: [ 2] c2 29
```

```
col 6: *NULL*
col 7: [ 2]  c1 0b
tab 0, row 7, @0x1e54
tl: 40 fb: --H-FL-- lb: 0x2  cc: 8
col 0: [ 3]  c2 4e 59
col 1: [ 5]  53 43 4f 54 54
col 2: [ 7]  41 4e 41 4c 59 53 54
col 3: [ 3]  c2 4c 43
col 4: [ 7]  77 bb 04 13 01 01 01
col 5: [ 2]  c2 29
col 6: *NULL*
col 7: [ 2]  c1 15
```

至此，整个事务过程被完全解析。
下面来看一下这个事务的内部流程，如图 7-2 所示。

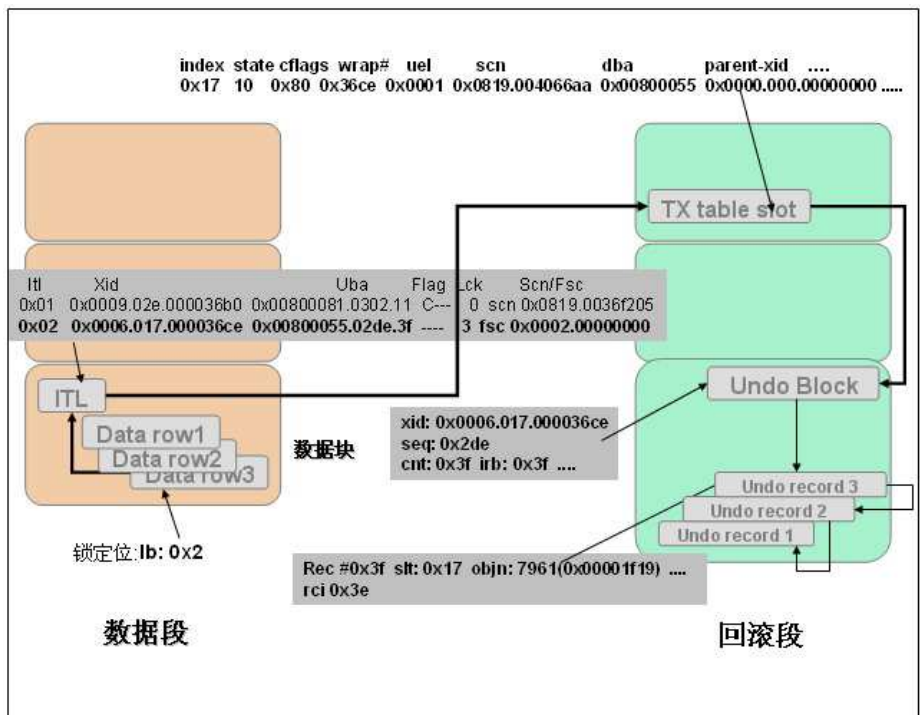


图 7-2 事务内部示意图

- (1) 首先当一个事务开始时，需要在回滚段事务表上分配一个事务槽。
- (2) 在数据块头部获取一个 ITL 事务槽，该事务槽指向回滚段头的事务槽。
- (3) 在修改数据之前，需要记录前镜像信息，这个信息以 UNDO RECORD 的形式存储在回滚段中，回滚段头事务槽指向该记录。
- (4) 锁定修改行，修改行锁定位 (lb-lock byte) 指向 ITL 事务槽。

(5) 数据修改可以进行。

这就是一个事务的基本流程。

7. 块清除 (Block Cleanouts)

当用户发出提交 (Commit) 之后, Oracle 怎样来处理。通过上一章的内容可以知道, Oracle 需要写出 Redo 来保证故障时数据可以被恢复; 我们也知道 Oracle 并不需要在提交时就写出变更的数据块。那么在提交时, Oracle 需要对数据块进行哪些操作呢?

在事务需要修改数据时, 必须分配 ITL 事务槽, 必须锁定行, 必须分配回滚段事务槽和回滚空间记录前镜像。当事务提交时, Oracle 需要将回滚段上的事务表信息标记为非活动, 以便空间可以重用; 那么还有 ITL 事务信息和锁定信息需要清除, 以记录提交。

由于 Oracle 在数据块上存储了 ITL 和锁定等事务信息, 所以 Oracle 必须在事务提交之后清除这些事务数据, 这就是块清除。块清除主要清除的数据有行级锁、ITL 信息 (包括提交标志、SCN 等)。

如果提交时修改过的数据块仍然在 Buffer Cache 之中, 那么 Oracle 可以清除 ITL 信息, 这叫做快速块清除 (Fast Block Cleanout), 快速块清除还有一个限制, 当修改的块数量超过 Buffer Cache 的约 10%, 则对超出部分不再进行快速块清除。

如果提交事务时, 修改过的数据块已经被写回到数据文件上 (或大量修改超出 10% 的部分), 再次读出该数据块进行修改, 显然成本过于高昂, 对于这种情况, Oracle 选择延迟块清除 (Delayed Block Cleanout), 等到下一次访问该 Block 时再来清除 ITL 锁定信息, 这就是延迟块清除。Oracle 通过延迟块清除来提高数据库的性能, 加快提交操作。快速提交是最普遍的情况, 来看一下延迟块清除的处理。

继续前面的测试:

```
SQL> update emp set sal=4000 where empno=7788;
```

```
1 row updated.
```

```
SQL> update emp set sal=4000 where empno=7782;
```

```
1 row updated.
```

```
SQL> update emp set sal=4000 where empno=7698;
```

```
1 row updated.
```

更新完成之后, 强制刷新 Buffer Cache, 将 Buffer Cache 中的数据都写出到数据文件:

```
SQL> alter session set events = 'immediate trace name flush_cache';
```

```
Session altered.
```

此时再提交事务:

```
SQL> commit;
```

```
Commit complete.
```

由于此时更新过的数据已经写出到数据文件，Oracle 将执行延迟块清除，将此时的数据块和回滚段转储出来：

```
[oracle@jumper udump]$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 9.2.0.4.0 - Production on Tue Apr 18 14:37:14 2006
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Connected to:
```

```
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

```
With the Partitioning option
```

```
JServer Release 9.2.0.4.0 - Production
```

```
SQL> alter system dump datafile 1 block 23642;
```

```
System altered.
```

```
SQL> alter system dump undo header '_SYSSMU6$';
```

```
System altered.
```

```
SQL> alter system dump datafile 2 block 85;
```

```
System altered.
```

看数据块上的信息，ITL 事务信息仍然存在：

```
Start dump data blocks tsn: 0 file#: 1 minblk 23642 maxblk 23642
```

```
buffer tsn: 0 rdba: 0x00405c5a (1/23642)
```

```
scn: 0x0819.00406ddf seq: 0x01 flg: 0x04 tail: 0x6ddf0601
```

```
frmt: 0x02 chkval: 0x6b6a type: 0x06=trans data
```

```
Block header dump: 0x00405c5a
```

```
Object id on Block? Y
```

```
seg/obj: 0x1f19 csc: 0x819.406ddf itc: 2 flg: O typ: 1 - DATA
```

```
fsl: 0 fnx: 0x0 ver: 0x01
```

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
-----	-----	-----	------	-----	---------

```
0x01    0x0009.02e.000036b0  0x00800081.0302.11  C---    0  scn 0x0819.0036f205
0x02    0x0006.017.000036ce  0x00800055.02de.3f  ----    3  fsc 0x0002.00000000
```

数据块的锁定信息仍然存在：

```
tab 0, row 5, @0x1d19
tl: 40 fb: --H-FL-- lb: 0x2  cc: 8
col  0: [ 3]  c2 4d 63
col  1: [ 5]  42 4c 41 4b 45
col  2: [ 7]  4d 41 4e 41 47 45 52
col  3: [ 3]  c2 4f 28
col  4: [ 7]  77 b5 05 01 01 01 01
col  5: [ 2]  c2 29
col  6: *NULL*
col  7: [ 2]  c1 1f
tab 0, row 6, @0x1d41
tl: 40 fb: --H-FL-- lb: 0x2  cc: 8
col  0: [ 3]  c2 4e 53
col  1: [ 5]  43 4c 41 52 4b
col  2: [ 7]  4d 41 4e 41 47 45 52
col  3: [ 3]  c2 4f 28
col  4: [ 7]  77 b5 06 09 01 01 01
col  5: [ 2]  c2 29
col  6: *NULL*
col  7: [ 2]  c1 0b
tab 0, row 7, @0x1e54
tl: 40 fb: --H-FL-- lb: 0x2  cc: 8
col  0: [ 3]  c2 4e 59
col  1: [ 5]  53 43 4f 54 54
col  2: [ 7]  41 4e 41 4c 59 53 54
col  3: [ 3]  c2 4c 43
col  4: [ 7]  77 bb 04 13 01 01 01
col  5: [ 2]  c2 29
col  6: *NULL*
col  7: [ 2]  c1 15
```

再来看回滚段的信息：

```
0x17    9    0x00  0x36ce  0xffff  0x0819.0040791b  0x00800055  0x0000.000.00000000
0x00000001  0x00000000
```

事务提交，事务表已经释放。如果此时查询 scott.emp 表，数据库将产生延迟块清除：

```
SQL> set autotrace on
```

```
SQL> select * from scott.emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	4000		30
7782	CLARK	MANAGER	7839	09-JUN-81	4000		10
7788	SCOTT	ANALYST	7566	19-APR-87	4000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10

.....

14 rows selected.

Execution Plan

```

0      SELECT STATEMENT Optimizer=CHOOSE
1    0   TABLE ACCESS (FULL) OF 'EMP'
```

Statistics

```

0      recursive calls
0      db block gets
5      consistent gets
2      physical reads
60     redo size
1305   bytes sent via SQL*Net to client
503    bytes received via SQL*Net from client
2      SQL*Net roundtrips to/from client
0      sorts (memory)
0      sorts (disk)
14     rows processed
```

注意到查询在此时产生了物理读和 Redo，这个 Redo 就是因为延迟块清除导致的。再次查询，则不会继续生成 Redo 了：

```
SQL> /

      EMPNO ENAME      JOB              MGR HIREDATE          SAL          COMM
DEPTNO
-----
      7369 SMITH        CLERK              7902 17-DEC-80        800              20
      7499 ALLEN        SALESMAN           7698 20-FEB-81       1600           300        30
      7521 WARD         SALESMAN           7698 22-FEB-81       1250           500        30
      7566 JONES        MANAGER           7839 02-APR-81       2975              20
      7654 MARTIN        SALESMAN           7698 28-SEP-81       1250          1400        30
      7698 BLAKE        MANAGER           7839 01-MAY-81       4000              30
      7782 CLARK        MANAGER           7839 09-JUN-81       4000              10
      7788 SCOTT        ANALYST           7566 19-APR-87       4000              20
.....

14 rows selected.

Execution Plan
-----
      0      SELECT STATEMENT Optimizer=CHOOSE
      1      0      TABLE ACCESS (FULL) OF 'EMP'

Statistics
-----
      0      recursive calls
      0      db block gets
      4      consistent gets
      0      physical reads
      0      redo size
.....
```

再次转储一下该 Block 来看看：

```
SQL> alter system dump datafile 1 block 23642;
```

```
System altered.
```

看到此时 ITL 事务信息已经清除，但是注意，这里的 Xid 和 Uba 信息仍然存在：

```
Start dump data blocks tsn: 0 file#: 1 minblk 23642 maxblk 23642
buffer tsn: 0 rdba: 0x00405c5a (1/23642)
scn: 0x0819.00407961 seq: 0x01 flg: 0x00 tail: 0x79610601
frmt: 0x02 chkval: 0x0000 type: 0x06=trans data
Block header dump:  0x00405c5a
Object id on Block? Y
seg/obj: 0x1f19 csc: 0x819.407961 itc: 2 flg: O typ: 1 - DATA
    fsl: 0   fnx: 0x0 ver: 0x01
```

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0009.02e.000036b0	0x00800081.0302.11	C---	0	scn 0x0819.0036f205
0x02	0x0006.017.000036ce	0x00800055.02de.3f	C---	0	scn 0x0819.0040791b

数据行的锁定位也已经清除：

```
tab 0, row 5, @0x1d19
tl: 40 fb: --H-FL-- lb: 0x0  cc: 8
col  0: [ 3]  c2 4d 63
col  1: [ 5]  42 4c 41 4b 45
col  2: [ 7]  4d 41 4e 41 47 45 52
col  3: [ 3]  c2 4f 28
col  4: [ 7]  77 b5 05 01 01 01 01
col  5: [ 2]  c2 29
col  6: *NULL*
col  7: [ 2]  c1 1f
tab 0, row 6, @0x1d41
tl: 40 fb: --H-FL-- lb: 0x0  cc: 8
col  0: [ 3]  c2 4e 53
col  1: [ 5]  43 4c 41 52 4b
col  2: [ 7]  4d 41 4e 41 47 45 52
col  3: [ 3]  c2 4f 28
col  4: [ 7]  77 b5 06 09 01 01 01
col  5: [ 2]  c2 29
col  6: *NULL*
col  7: [ 2]  c1 0b
tab 0, row 7, @0x1e54
tl: 40 fb: --H-FL-- lb: 0x0  cc: 8
col  0: [ 3]  c2 4e 59
```

```

col 1: [ 5] 53 43 4f 54 54
col 2: [ 7] 41 4e 41 4c 59 53 54
col 3: [ 3] c2 4c 43
col 4: [ 7] 77 bb 04 13 01 01 01
col 5: [ 2] c2 29
col 6: *NULL*
col 7: [ 2] c1 15

```

8. 提交之后的 Undo 信息

当提交事务之后，回滚段事务表标记该事务为非活动，继续再来看一下回滚段数据块的信息。看到这里 **irb** 指向了 **0x40**，此前的事务已经不可回滚。

```

Start dump data blocks tsn: 1 file#: 2 minblk 85 maxblk 85
buffer tsn: 1 rdba: 0x00800055 (2/85)
scn: 0x0819.00407baa seq: 0x01 flg: 0x04 tail: 0x7baa0201
frmt: 0x02 chkval: 0x053d type: 0x02=KTU UNDO BLOCK

```

```

*****

```

UNDO BLK:

```

xid: 0x0006.025.000036ce seq: 0x2de cnt: 0x40 irb: 0x40 icl: 0x0 flg: 0x0000

```

看一下偏移量列表也已经新增了一条信息 **0x40 0x01b0**:

Rec Offset	Rec Offset	Rec Offset	Rec Offset	Rec Offset

.....				
0x33 0x097c	0x34 0x08d4	0x35 0x082c	0x36 0x0784	0x37 0x06dc
0x38 0x0634	0x39 0x058c	0x3a 0x04e4	0x3b 0x043c	0x3c 0x0394
0x3d 0x0310	0x3e 0x02b4	0x3f 0x0258	0x40 0x01b0	

再看前镜像 **0x3d 0x3e 0x3f** 的信息，仍然存在:

```

*-----
* Rec #0x3d slt: 0x17 objn: 7961(0x00001f19) objd: 7961 tblspc: 0(0x00000000)
*      Layer: 11 (Row) opc: 1 rci 0x00
Undo type: Regular undo Begin trans Last buffer split: No
Temp Object: No
Tablespace Undo: No
rdba: 0x00000000
*-----
uba: 0x00800055.02de.3c ctl max scn: 0x0819.003f594b prv tx scn: 0x0819.003f5d3e

```

```

KDO undo record:
KTB Redo
op: 0x04 ver: 0x01
op: L itl: xid: 0x0002.01d.000038ea uba: 0x008000c3.04b1.0c
           flg: C--- lkc: 0 scn: 0x0819.0036f14f
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

*-----
* Rec #0x3e slt: 0x17 objn: 7961(0x00001f19) objd: 7961 tblspc: 0(0x00000000)
*      Layer: 11 (Row) opc: 1 rci 0x3d
Undo type: Regular undo Last buffer split: No
Temp Object: No
Tablespace Undo: No
rdba: 0x00000000
*-----

KDO undo record:
KTB Redo
op: 0x02 ver: 0x01
op: C uba: 0x00800055.02de.3d
KDO Op code: URP row dependencies Disabled
      xtype: XA bdba: 0x00405c5a hdba: 0x00405c59
itli: 2 ispac: 0 maxfr: 4863
tabn: 0 slot: 6(0x6) flag: 0x2c lock: 0 ckix: 0
ncol: 8 nnew: 1 size: 1
col 5: [ 3] c2 19 33

*-----

```

可以猜想，虽然这个事务已经提交，不可以回滚了，但是在覆盖之前，这个前镜像信息仍然存在，通过某种手段，应该仍然可以获得这个信息。这个猜想显然是成立的。

7.6 Oracle 9i 闪回查询的新特性

从 Oracle 9i 开始，Oracle 开始提供闪回查询特性（Flashback Query），允许将回滚段中的

数据进行闪回。通过这个例子来看一下这个从 Oracle 9i 开始提供的新特性。

首先注意到这里存在一个信息 `ctl max scn: 0x0819.003f594b`，这个转换为 SCN 值就是：

```
SQL> select (to_number('819','xxx')*power(2,32) + to_number('3f594b','xxxxxxx')) scn
2   from dual;

          SCN
-----
8903471356235
```

可以查询一下当前数据库的 SCN：

```
SQL> select dbms_flashback.get_system_change_number scn from dual;

          SCN
-----
8903471437610
```

通过特定的语法，可以将 SCN 为 8903728479868 的历史状态数据查询出来：

```
SQL> select * from emp as of scn 8903471356235 where empno in (7788,7782,7698);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER		7839 01-MAY-81	2850		30
7782	CLARK	MANAGER		7839 09-JUN-81	2450		10
7788	SCOTT	ANALYST		7566 19-APR-87	3000		20

在结果中，注意到 3 名员工的薪水恢复到了之前值。而在当前的查询中，这个数值是变化后的 4000：

```
SQL> select * from emp where empno in (7788,7782,7698);
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER		7839 01-MAY-81	4000		30
7782	CLARK	MANAGER		7839 09-JUN-81	4000		10
7788	SCOTT	ANALYST		7566 19-APR-87	4000		20

由于这个查询需要从 Undo 中获取前镜像信息，如果 Undo 中的信息被覆盖，则以上查询将会失败。测试一下，当新建 Undo 表空间，切换 Undo 表空间，再将原表空间 Offline 之后：

```
SQL> create undo tablespace undotbs datafile '/opt/oracle/oradata/conner/undotbs.dbf' size 2M;

Tablespace created.
```

```
SQL> alter system set undo_tablespace=undotbs;
```

System altered.

```
SQL> alter tablespace UNDOTBS1 offline;
```

Tablespace altered.

```
SQL> alter session set events = 'immediate trace name flush_cache';
```

Session altered.

再来查询，此时出现错误，记录该文件已经不可读取：

```
SQL> select * from emp as of scn 8903471356235 where empno in (7788,7782,7698);
```

```
select * from emp as of scn 8903471356235 where empno in (7788,7782,7698)
```

*

ERROR at line 1:

ORA-00376: file 2 cannot be read at this time

ORA-01110: data file 2: '/opt/oracle/oradata/conner/undotbs1.dbf'

将 UNDOTBS1 重新启用：

```
SQL> alter tablespace UNDOTBS1 online;
```

Tablespace altered.

```
SQL> alter system set undo_tablespace=UNDOTBS1;
```

System altered.

此时前镜像信息再次可以查询，在其他 Session 执行大量事务，使得前镜像信息被覆盖：

```
SQL> begin
```

```
2   for i in 1 .. 2000 loop
```

```
3     update emp set sal=4000;
```

```
4   rollback;
```

```
5   end loop;
```

```
6   end;
```

```
7   /
```

PL/SQL procedure successfully completed.

```
SQL> /
```

```
PL/SQL procedure successfully completed.
```

```
SQL> /
```

观察回滚段的使用：

```
SQL> select usn,xacts,RSSIZE,HWMSIZE from v$rollstat where usn=6;
```

USN	XACTS	RSSIZE	HWMSIZE
6	1	7331840	7331840

那么再次查询就可能收到如下错误：

```
SQL> select * from emp as of scn 8903471356235 where empno in (7788,7782,7698);
```

```
select * from emp as of scn 8903471356235 where empno in (7788,7782,7698)
```

```
*
```

```
ERROR at line 1:
```

```
ORA-01555: snapshot too old: rollback segment number 6 with name "_SYSSMU6$" too small
```

ORA-01555 错误出现，说明要查询的前镜像信息已经失去。

7.7 使用 ERRORSTACK 进行错误跟踪

ERRORSTACK 是 Oracle 提供的接口，用于诊断 Oracle 的错误信息。

诊断事件可以在 **Session** 级设置，也可以在系统级设置，通常如果要诊断全局错误，最好在系统级设置。设置了 **ERRORSTACK** 事件之后，Oracle 会将出错时的信息记入跟踪文件之中，用户就可以通过跟踪文件进行错误诊断和排查了。

继续上面的测试，可以通过 **ERRORSTACK** 事件来跟踪 ORA-01555 错误：

```
SQL> ALTER SYSTEM SET EVENTS '1555 TRACE NAME ERRORSTACK LEVEL 4';
```

```
System altered.
```

```
SQL> select * from emp as of scn 8903471356235 where empno in (7788,7782,7698);
```

```
select * from emp as of scn 8903471356235 where empno in (7788,7782,7698)
```

```
*
```

```
ERROR at line 1:
```

```
ORA-01555: snapshot too old: rollback segment number 6 with name "_SYSSMU6$" too small
```

检查警告日志文件，可以得到如下信息：

```
Wed Apr 19 16:46:51 2006
OS Pid: 1274 executed alter system set events '1555 TRACE NAME ERRORSTACK LEVEL 4'
Wed Apr 19 16:47:06 2006
ORA-01555 caused by SQL statement below (Query Duration=0 sec, SCN: 0x0819.003f594b):
Wed Apr 19 16:47:06 2006
select * from emp as of scn 8903471356235 where empno in (7788,7782,7698)
Wed Apr 19 16:47:06 2006
Errors in file /opt/oracle/admin/conner/udump/conner_ora_1274.trc:
ORA-01555: snapshot too old: rollback segment number 6 with name "???" too small
```

这里注意到，触发 ORA-01555 错误的语句被记录，出现错误的 SCN 也被纪录，这个 SCN: 0x0819.003f594b 正是我们的 ctl max scn: 0x0819.003f594b，进一步的，找到 conner_ora_1274.trc 跟踪文件，就可以获得关于这次错误的相信信息用于诊断。

摘录一点重要信息，简要说明一下。

(1) 错误信息：

```
*** 2006-04-19 16:47:06.606
ksedmp: internal or fatal error
ORA-01555: snapshot too old: rollback segment number 6 with name "???" too small
Current SQL statement for this session:
select * from emp as of scn 8903471356235 where empno in (7788,7782,7698)
```

(2) 数据块信息。这里的块头就包含了 ITL 信息，根据这个 ITL 信息中的 Uba，Oracle 可以去定位回滚段，查询前镜像信息，如果不存在，就可能出现 ORA-01555 错误。

```
Block header dump: 0x00405c5a
Object id on Block? Y
seg/obj: 0x1f19 csc: 0x819.407961 itc: 2 flg: O typ: 1 - DATA
fsl: 0 fnx: 0x0 ver: 0x01
```

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0009.02e.000036b0	0x00800081.0302.11	C---	0	scn 0x0819.0036f205
0x02	0x0006.017.000036ce	0x00800055.02de.3f	C---	0	scn 0x0819.0040791b

也可以根据 Xid，获得回滚段号、事务槽等信息，主动查询或转储回滚段信息进行研究观察。限于篇幅，本文不再过多介绍。

以上测试环境为：

```
SQL> select * from v$version;
```

```
BANNER
```

```

Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
PL/SQL Release 9.2.0.4.0 - Production
CORE      9.2.0.3.0      Production
TNS for Linux: Version 9.2.0.4.0 - Production
NLSRTL Version 9.2.0.4.0 - Production

```

7.8 Oracle 10g 闪回查询特性的增强

Oracle 9i 提供的闪回特性增强，为恢复带来了极大的方便，但是 Oracle 9i 的闪回查询只能提供某个时间点的数据视图，并不能告诉用户这样的数据经过了几个事务、怎样的修改（UPDATE、INSERT、DELETE 等），而这些信息在回滚段中是存在的，在 Oracle 10g 中，Oracle 进一步加强了闪回查询的特性，提供了以下两种闪回查询：

- 闪回版本查询（Flashback Versions Query）
- 闪回事务查询（Flashback Transaction Query）

闪回版本查询允许使用一个新的 **VERSIONS** 子句查询两个时间点或者 SCN 之间的数据版本。这些版本可以按照事务进行区分，闪回版本查询只返回提交数据，未提交数据不被显示。

通过以下示例，可以很容易地理解闪回版本查询的作用。首先创建一个测试表，执行一系列的 DML 操作：

```

EYGLE on 30-MAR-05 >create table t as select username,user_id from dba_users;
Table created.

```

```

EYGLE on 30-MAR-05 >select * from t;

```

USERNAME	USER_ID
SYSTEM	5
SYS	0
TEST	25
EYGLE	26
SCOTT	29
DIP	19
TRANS	27
TEST1	28
OPERATOR	31
WMSYS	23
DBSNMP	22
OUTLN	11

12 rows selected.

EYGLE on 30-MAR-05 >delete from t where username='OUTLN';

1 row deleted.

EYGLE on 30-MAR-05 >commit;

Commit complete.

EYGLE on 30-MAR-05 >delete from t where username='TEST1';

1 row deleted.

EYGLE on 30-MAR-05 >commit;

Commit complete.

此时的数据:

EYGLE on 30-MAR-05 >select * from t;

USERNAME	USER_ID
SYSTEM	5
SYS	0
TEST	25
EYGLE	26
SCOTT	29
DIP	19
TRANS	27
OPERATOR	31
WMSYS	23
DBSNMP	22

10 rows selected.

再执行一系列 DML 操作并提交:

EYGLE on 30-MAR-05 >update t set user_id=1 where username='EYGLE';

1 row updated.

EYGLE on 30-MAR-05 >commit;

Commit complete.

EYGLE on 30-MAR-05 >delete from t where user_id >10;

7 rows deleted.

EYGLE on 30-MAR-05 >commit;

```

Commit complete.
EYGLE on 30-MAR-05 >select * from t;

USERNAME                                USER_ID
-----
SYSTEM                                  5
SYS                                     0
EYGLE                                   1

EYGLE on 30-MAR-05 >insert into t values('PENNY',2);
1 row created.
EYGLE on 30-MAR-05 >commit;
Commit complete.

```

至此已经交替执行了多个事务，进行了众多的数据修改，现在的测试表已经与最初完全不同了。如果使用 Oracle 9i 的闪回查询，就很难区分这些不同事务的变更，找到合适的、正确的数据将变得极为困难。

再来看看 Oracle 10g 的闪回版本查询，通过使用 **VERSIONS** 子句，和对数据表引入了一系列的伪列（**version_starttime** 等），可以获得对数据表的所有事务操作，注意以下输出中 **versions_operation** 代表了不同类型的操作（**D-DELETE**、**I-INSERT**、**U-UPDATE**），**VERSIONS_XID** 是一个重要数据，代表了不同版本的事务 ID：

```

EYGLE on 30-MAR-05 >select versions_starttime, versions_endtime, versions_xid,
2      versions_operation, username,user_id
3      from t versions between timestamp minvalue and maxvalue
4      /

VERSIONS_STARTTIME VERSIONS_ENDTIME  VERSIONS_XID    V USERNAME  USER_ID
-----
30-MAR-05 09.34.49 AM                000A000B000000F1 D DBSNMP      22
30-MAR-05 09.34.49 AM                000A000B000000F1 D WMSYS        23
30-MAR-05 09.34.49 AM                000A000B000000F1 D OPERATOR     31
30-MAR-05 09.34.49 AM                000A000B000000F1 D TRANS        27
30-MAR-05 09.34.49 AM                000A000B000000F1 D DIP           19
30-MAR-05 09.34.49 AM                000A000B000000F1 D SCOTT         29
30-MAR-05 09.34.49 AM                000A000B000000F1 D TEST          25
30-MAR-05 09.34.15 AM                00010019000000F0F U EYGLE         1
30-MAR-05 09.33.51 AM                00080016000000EF D TEST1         28
30-MAR-05 09.33.23 AM                0004000A0000005EF D OUTLN         11

```

```

                                SYSTEM      5
                                SYS         0
30-MAR-05 09.34.49 AM          TEST        25
30-MAR-05 09.34.15 AM          EYGLE       26
30-MAR-05 09.34.49 AM          SCOTT       29
30-MAR-05 09.34.49 AM          DIP         19
30-MAR-05 09.34.49 AM          TRANS       27
30-MAR-05 09.33.51 AM          TEST1      28
30-MAR-05 09.34.49 AM          OPERATOR    31
30-MAR-05 09.34.49 AM          WMSYS      23
30-MAR-05 09.34.49 AM          DBSNMP     22
30-MAR-05 09.33.23 AM          OUTLN     11
30-MAR-05 09.49.24 AM          00080006000000EF I PENNY 2

23 rows selected.

```

通过以上输出，根据 `VERSIONS_XID` 可以清晰地区分不同事务在不同时间对数据所作的更改。

具备了 Flashback Version Query 查询的基础，就可以进行基于 Flashback Version Query 的事务级恢复，这就是 Flashback Transaction Query。Flashback Transaction Query 可以从 `FLASHBACK_TRANSACTION_QUERY` 视图获得指定事务的历史信息以及 `Undo_SQL`，通过这个 `UNDO_SQL`，就可以撤销特定的提交事务。Flashback Transaction Query 需要用到 `FLASHBACK_TRANSACTION_QUERY` 视图，先看一下视图：

```
SQL> desc FLASHBACK_TRANSACTION_QUERY;
```

Name	Type	Nullable	Default	Comments
XID	RAW(8)	Y		Transaction identifier
START_SCN	NUMBER	Y		Transaction start SCN
START_TIMESTAMP	DATE	Y		Transaction start timestamp
COMMIT_SCN	NUMBER	Y		Transaction commit SCN
COMMIT_TIMESTAMP	DATE	Y		Transaction commit timestamp
LOGON_USER	VARCHAR2(30)	Y		Logon user for transaction
UNDO_CHANGE#	NUMBER	Y		1-based undo change number
OPERATION	VARCHAR2(32)	Y		forward operation for this undo
TABLE_NAME	VARCHAR2(256)	Y		table name to which this undo applies
TABLE_OWNER	VARCHAR2(32)	Y		owner of table to which this undo applies
ROW_ID	VARCHAR2(19)	Y		rowid to which this undo applies
UNDO_SQL	VARCHAR2(4000)	Y		SQL corresponding to this undo

该视图的定义为：

```
select xid, start_scn, start_timestamp,
       decode(commit_scn, 0, commit_scn, 281474976710655, NULL, commit_scn)
       commit_scn, commit_timestamp,
       logon_user, undo_change#, operation, table_name, table_owner,
       row_id, undo_sql
from sys.x$ktuqqry
```

注 意

对于 x\$ktuqqry 的查询非常耗时，所以请注意评估你的恢复成本。

在以下测试中，x\$ktuqqry 中存在大约 19 万记录，查询一次需要近 6 分钟。

```
SYS AS SYSDBA on 30-MAR-05 >select count(addr) from x$ktuqqry;
```

```
COUNT(ADDR)
```

```
-----
```

```
196015
```

继续前面的测试，如果需要撤销 Xid=000A000B000000F1 的事务，可以通过如下步骤进行：

```
SYS AS SYSDBA on 30-MAR-05 >set autotrace on
```

```
SYS AS SYSDBA on 30-MAR-05 >SELECT UNDO_SQL FROM FLASHBACK_TRANSACTION_QUERY
2 WHERE XID = '000A000B000000F1';
```

```
UNDO_SQL
```

```
-----
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('DBSNMP','22');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('WMSYS','23');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('OPERATOR','31');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('TRANS','27');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('DIP','19');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('SCOTT','29');
```

```
insert into "EYGLE"."T"("USERNAME","USER_ID") values ('TEST','25');
```

```
8 rows selected.
```

```
Elapsed: 00:05:55.30
```

```
Execution Plan
```

```

-----
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=25 Card=1 Bytes=2008)
1      0      FIXED TABLE (FULL) OF 'X$KTUQQRY' (TABLE (FIXED)) (Cost=25 Card=1 Bytes=2008)

Statistics
-----

393454  recursive calls
          0  db block gets
1562425  consistent gets
          4644  physical reads
          0  redo size
          1069  bytes sent via SQL*Net to client
           664  bytes received via SQL*Net from client
              2  SQL*Net roundtrips to/from client
          23166  sorts (memory)
              0  sorts (disk)
              8  rows processed

```

通过执行相应的 Undo 语句可以撤销该事务，通过这些新特性，Oracle 提供了一种“回滚”提交事务的手段，极大地方便了用户应对不同情况的数据库恢复。

7.9 ORA-01555 错误

上文提到了 ORA-01555 错误，那么现在来讨论一下 ORA-01555 错误是怎样产生的。

由于回滚段是循环使用的，当事务提交以后，该事务占用的回滚段事务表会被标记为非活动，回滚段空间可以被覆盖重用。那么一个问题就出现了，如果一个查询需要使用被覆盖的回滚段构造前镜像实现一致性读，那么此时就会出现 Oracle 著名的 ORA-01555 错误。

ORA-01555 错误的另外一个原因是因为延迟块清除（Delayed Block Cleanout）。当一个查询触发延迟块清除时，Oracle 需要去查询回滚段获得该事务的提交 SCN，如果事务的前镜像信息已经被覆盖，并且查询 SCN 也小于回滚段中记录的最小提交 SCN，那么 Oracle 将无从判断查询 SCN 和事务提交 SCN 的大小，此时出现延迟块清除导致的 ORA-01555 错误。

另外一种导致 ORA-01555 错误的情况出现在使用 `sqlldr` 直接方式加载（`direct=true`）数据时。当通过 `sqlldr direct=true` 方式加载数据时，由于不产生重做和回滚信息，Oracle 直接指定 `Cached Commit SCN` 给加载数据，在访问这些数据时，有时会产生 ORA-01555 错误。

假定在时间 T 用户 A 发出一条更新语句，更新 SCOTT 用户的 SAL；用户 B 在 Ty 时间发出查询语句，查询 SCOTT 用户的 SAL；用户 A 的更新在 Tx 时间提交，提交可能为快速提交清除，也可能是延迟块清除；用户 B 的查询在 Tz 时间输出。

来看一下数据库在不同情况下的内部处理，如图 7-3 所示。

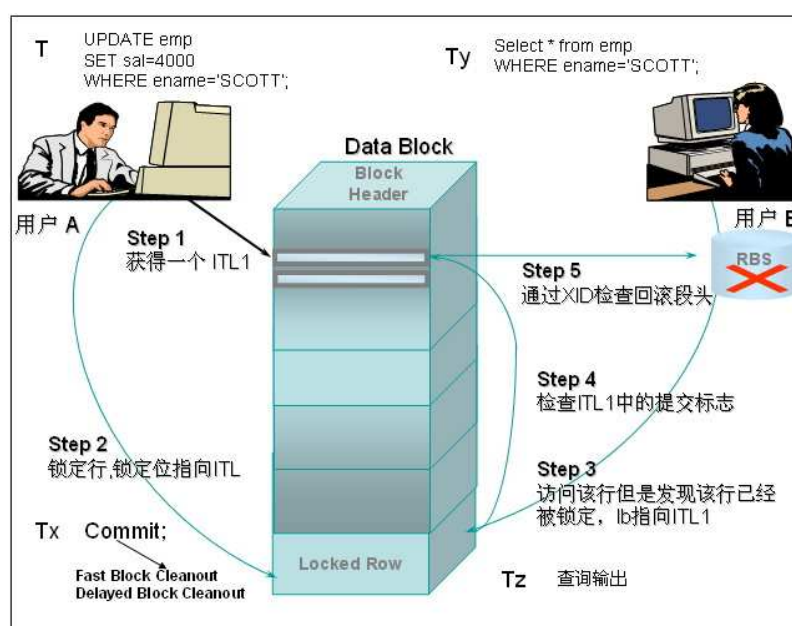


图 7-3 事务处理示意图

(1) 如果 $Ty < T < Tz < Tx$ ，那么查询需要构造一致性读，由于事务尚未提交，可以通过回滚段构造前镜像，完成一致性读取。

(2) 如果 $Ty < T < Tx < Tz$ ，由于 Ty 查询时间小于 T 事务更新时间，那么数据库需要构造一致性读取，而 Tz 查询完成时间小于 Tx 提交时间，那么前镜像就有可能被覆盖，不可获取。

如果 Tx 的提交方式为 Fast Block Cleanout，那么回滚段信息不可用时就会出现一致性读 ORA-01555 错误。如果 Tx 的提交方式为 Delayed Block Cleanout，那么回滚段信息不可用时 Oracle 将无法判断 Ty 和 Tx 的时间先后关系。如果 $Ty > Tx$ ，那么 Oracle 可以正常进行块清除，并将块清除后的数据返回给用户 B；如果 $Ty < T$ ，那么 Oracle 需要继续构造一致性读返回给用户 B；Oracle 无法判断这两种情况，就会出现延迟块清除 ORA-01555 错误。

ORA-01555 的直观解释是“Snapshot too old”，也就是快照太旧，其根本含义就是查询需要的前镜像过于“久远”，已经无法找到了。可以想象，如果一个历时数个小时或 10 数小时的查询，如果最后遭遇 ORA-01555 错误而失败，会是多么令人沮丧的一件事。一直以来，ORA-01555 都是 Oracle 最为头痛的问题之一。

当数据出现 ORA-01555 错误时，Oracle 会将错误信息记录在警告日志中，从生产数据库中摘录 ORA-01555 错误信息举例如下。这是一段 ORA-01555 错误信息，其中 Query Duration=14616 sec 指查询经历的时间，这个时间已经超过了 undo_retention 的缺省值 10800 的设置：

Mon Aug 1 15:03:39 2005

ORA-01555 caused by SQL statement below (Query Duration=14616 sec, SCN: 0x0000.1e5294a9):

Mon Aug 1 15:03:39 2005

select sp.vc2planguid

```

        from cy_sinfo      pvd,
            cy_serviceinfo svr,
            cy_serviceplan sp,
            cy_feetype      ft
    where pvd.vc2spguid = svr.vc2spguid and
          svr.vc2serviceguid = sp.vc2serviceguid and
          sp.vc2ftguid = ft.vc2ftguid and

          to_char(sysdate, 'yyyymmddhh24miss') between
            nvl(pvd.vc2startdate,
                to_char(sysdate, 'yyyymmddhh24miss')) and
            nvl(pvd.vc2enddate,
                to_char(sysdate, 'yyyymmddhh24miss')) and

          to_char(sysdate, 'yyyymmddhh24miss') between
            nvl(sp.vc2startdate,
                to_char(sysdate, 'yyyymmddhh24miss')) and
            nvl(svr.vc2enddate,
                to_char(sysdate, 'yyyymmddhh24miss')) and

```

这个错误是由于一个 job 任务的执行所致：

Mon Aug 1 15:03:39 2005

Errors in file /opt/oracle/admin/hsboss/bdump/hsboss_j000_1088.trc:

ORA-12012: error on auto execute of job 181

ORA-01555: snapshot too old: rollback segment number 8 with name "_SYSSMU8\$" too small

ORA-06512: at "CYUSER.CYPKG_BILLING", line 385

ORA-06512: at line 1

在 Oracle 9i 的文档中这样描述 ORA-01555 错误：

ORA-01555 snapshot too old: rollback segment number *string* with name "*string*" too small

Cause: Rollback records needed by a reader for consistent read are overwritten by other writers.

Action: If in Automatic Undo Management mode, increase the setting of UNDO_RETENTION. Otherwise, use larger rollback segments.

可以看到，在 Oracle 9i 自动管理的 Undo 表空间模式下，undo_retention 参数的引入正是为了减少 ORA-01555 错误的出现。这个参数设置当事务提交之后，回滚段变得非激活，回滚段中的数据在被覆盖前保留的时间，该参数以秒为单位，9iR1 初始值为 900 秒，在 Oracle 9iR2 增加为 10800 秒。

显然该参数设置的越高就越能够减少 ORA-01555 错误的出现，但是保留时间和存储空间

是紧密相关的，如果 Undo 表空间的存储空间有限，那么 Oracle 就会选择回收已提交事务占用的空间，而置 `undo_retention` 参数的设置于不顾。

在 Oracle 9i 的 AUM 模式下，`undo_retention` 实际上是一个非但保（NO Guaranteed）限制。也就是说，如果有其他事务需要回滚空间，而空间出现不足时，这些信息仍然会被覆盖。虽然很多时候这是我们不希望看到的。

从 Oracle 10g 开始，Oracle 引入了自动的 `undo_retention` 调整（Automatic Undo Retention Tuning），缺省情况下，这个功能被启用，Oracle 动态地收集系统的事务信息，自动调整以满足最长运行查询的需要。当然如果空间不足，那么 Oracle 满足最大允许的长时间查询，而不再需要用户手工调整。

这个新特性的引入伴随着几个新的隐含初始化参数，主要的参数有两个：

```
SQL> select ksppinm,ksppdesc
      2  from x$ksppi where ksppinm like '%&var%'
      3  /

Enter value for var: undo_autotune
old   2: from x$ksppi where ksppinm like '%&var%'
new   2: from x$ksppi where ksppinm like '%undo_autotune%'

KSPPINM                                KSPPDESC
-----
_undo_autotune                        enable auto tuning of undo_retention

SQL> /

Enter value for var: collect_undo_stats
old   2: from x$ksppi where ksppinm like '%&var%'
new   2: from x$ksppi where ksppinm like '%collect_undo_stats%'

KSPPINM                                KSPPDESC
-----
_collect_undo_stats                  Collect Statistics v$undostat
```

`_undo_autotune` 和 `_collect_undo_stats` 这两个参数缺省都是打开的。

同时 Oracle 10g 增加了 Guarantee 控制，也就是说，用户可以指定 Undo 表空间必须满足 `undo_retention` 的限制。

```
SQL> alter tablespace undotbs1 retention guarantee;

Tablespace altered

SQL> alter tablespace undotbs1 retention noguarantee;

Tablespace altered
```

在 DBA_TABLESPACES 视图中增加了 RETENTION 字段用以描述该选项：

```
SQL> select tablespace_name,contents,retention from dba_tablespaces;
```

TABLESPACE_NAME	CONTENTS	RETENTION
SYSTEM	PERMANENT	NOT APPLY
UNDOTBS1	UNDO	NOGUARANTEE
SYSAUX	PERMANENT	NOT APPLY
TEMP	TEMPORARY	NOT APPLY
EYGLE	PERMANENT	NOT APPLY
ITPUB	PERMANENT	NOT APPLY
BIGTBS	PERMANENT	NOT APPLY
.....		
14 rows selected		

当 Undo 表空间设置为 Guarantee，那么提交事务的回滚空间必须被保留足够的时间，如果 Undo 表空间的空间不足，那么新事务会因空间不足而失败，而不是选择之前的覆盖。

从各个不同版本回滚段的管理变迁，可以看出，Oracle 一直在进步。

Oracle 提供一个内部事件（10203 事件）可以用来跟踪数据库的块清除操作，10203 事件可以通过以下命令设置，设置后需要重新启动数据库该参数方能生效：

```
alter system set event="10203 trace name context forever" scope=spfile;
```

设置 10203 事件后，下面通过实验来简单介绍一下 ORA-01555 错误的成因。为了实验的方便，先创建一个手工管理不可扩展的小 Undo 表空间：

```
SQL> create undo tablespace undotbs
2 datafile '/opt/oracle/oradata/conner/undotbs.dbf' size 2m autoextend off;

Tablespace created.

SQL> alter system set undo_tablespace=undotbs;

System altered.

SQL> alter system set undo_management=manual scope=spfile;

System altered.

SQL> alter system set event="10203 trace name context forever" scope=spfile;

System altered.
```

```

SQL> shutdown immediate;

Database closed.

Database dismounted.

ORACLE instance shut down.

SQL> startup

ORACLE instance started.


Total System Global Area  126948772 bytes
Fixed Size                  452004 bytes
Variable Size              92274688 bytes
Database Buffers           33554432 bytes
Redo Buffers                667648 bytes
Database mounted.
Database opened.
SQL> select * from v$rollname;

      USN NAME
-----
0 SYSTEM

SQL> create rollback segment rbs01 tablespace undotbs;

Rollback segment created.

SQL> alter rollback segment rbs01 online;

Rollback segment altered.

SQL> select * from v$rollname;

      USN NAME
-----
0 SYSTEM
21 RBS01

```

再来执行一个任务，先打开一个游标，执行一个查询（在适当步骤加入了数据块转储及回滚段转储命令，并将跟踪信息同步进行讲解）：

```
SQL> var cemp refcursor
SQL> begin
  2   open :cemp for select * from emp where empno=7788;
  3   end;
  4   /
```

PL/SQL procedure successfully completed.

```
SQL> alter system dump datafile 1 block 23642;
```

System altered.

此时无 DML 事务进行，此时的数据块 ITL 状态：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.045.000003b7	0x010000b1.2b39.04	C---	0	scn 0x0819.0045cd56
0x02	0x0015.03f.000003b6	0x010000b6.2b2b.06	C---	0	scn 0x0819.0045cb5c

然后更新一条记录并且提交：

```
SQL> update emp set sal=4000 where empno=7788;
```

1 row updated.

```
SQL> alter system dump datafile 1 block 23642;
```

System altered.

此时数据块上 ITL 及具体记录上都记录了锁定信息，本例里 ITL2（0x02）被使用：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.045.000003b7	0x010000b1.2b39.04	C---	0	scn 0x0819.0045cd56
0x02	0x0015.049.000003b8	0x010000ac.2b46.02	----	1	fsc 0x0000.00000000

数据行 LB 指向了 ITL 2：

```
tab 0, row 7, @0x1e7f
tl: 40 fb: --H-FL-- lb: 0x2  cc: 8
col 0: [ 3] c2 4e 59
col 1: [ 5] 53 43 4f 54 54
col 2: [ 7] 41 4e 41 4c 59 53 54
col 3: [ 3] c2 4c 43
col 4: [ 7] 77 bb 04 13 01 01 01
col 5: [ 2] c2 29
```

```
col 6: *NULL*
col 7: [ 2] c1 15
```

继续 SQL*Plus 中的操作，转储一下回滚段头信息：

```
SQL> alter system dump undo header RBS01;

System altered.
```

回滚段第 0x49 号事务槽被使用：

index	state	cflags	wrap#	uel	scn	dba	parent-xid	nub
0x49	10	0x80	0x03b8	0x0000	0x0819.0045cea3	0x010000ac	0x0000.000.00000000	

继续提交该事务，并再次转储回滚段头信息：

```
SQL> commit;

Commit complete.

SQL> alter system dump undo header RBS01;

System altered.
```

可以看到该事务已经被提交，回滚段事务状态变为非激活（state=9），提交的 SCN 为 **0x0819.0045cea4**。

index	state	cflags	wrap#	uel	scn	dba	parent-xid	nub
0x49	9	0x80	0x03b8	0xffff	0x0819.0045cea4	0x010000ac	0x0000.000.00000000	

来看此时的数据块：

```
SQL> alter system dump datafile 1 block 23642;

System altered.
```

ITL 已经记录了事务提交，SCN/FSC 表示 Commit SCN 或快速提交（Fast Commit Fsc）的 SCN，这个 SCN 和 Undo 中的提交 SCN 一致，是准确的 SCN：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.045.000003b7	0x010000b1.2b39.04	C---	0	scn 0x0819.0045cd56
0x02	0x0015.049.000003b8	0x010000ac.2b46.02	--U-	1	fsc 0x0000.0045cea4

此时的数据行锁定位并不需要清除：

```
tab 0, row 7, @0x1e7f
tl: 40 fb: --H-FL-- lb: 0x2 cc: 8
```

```
col 0: [ 3]  c2 4e 59
col 1: [ 5]  53 43 4f 54 54
col 2: [ 7]  41 4e 41 4c 59 53 54
col 3: [ 3]  c2 4c 43
col 4: [ 7]  77 bb 04 13 01 01 01
col 5: [ 2]  c2 29
col 6: *NULL*
col 7: [ 2]  c1 15
```

此时 10203 事件的跟踪信息被记录，块清除记录了确切的 SCN 信息：

```
Begin cleaning out block ...
Found active transactions
Block cleanout record, scn: 0x0819.0045cea6 ver: 0x01 opt: 0x02, entries follow...
    itli: 2  flg: 2  scn: 0x0819.0045cea4
Block cleanout under the cache...
Block cleanout record, scn: 0x0819.0045cea6 ver: 0x01 opt: 0x02, entries follow...
    itli: 2  flg: 2  scn: 0x0819.0045cea4
... clean out dump complete.
Start dump data blocks tsn: 0 file#: 1 minblk 23642 maxblk 23642
```

继续在 SQL*Plus 中的操作，执行一个批处理更新，由于只有一个用户回滚段，先前的事务信息很快被覆盖：

```
SQL> begin
2   for i in 1 .. 100 loop
3     update emp set sal=4000;
4     rollback;
5   end loop;
6 end;
7 /

PL/SQL procedure successfully completed.

SQL> alter system dump datafile 1 block 23642;
```

再看此时的数据块信息，数据块被写出，锁定位被清除，ITL Flag 标志位置为提交状态 (C-Commit)：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.045.000003b7	0x010000b1.2b39.04	C---	0	scn 0x0819.0045cd56
0x02	0x0015.049.000003b8	0x010000ac.2b46.02	C---	0	scn 0x0819.0045cea4

数据行锁定定位锁定信息清除：

```
tab 0, row 7, @0x1e81
tl: 40 fb: --H-FL-- lb: 0x0  cc: 8
col 0: [ 3]  c2 4e 59
col 1: [ 5]  53 43 4f 54 54
col 2: [ 7]  41 4e 41 4c 59 53 54
col 3: [ 3]  c2 4c 43
col 4: [ 7]  77 bb 04 13 01 01 01
col 5: [ 2]  c2 29
col 6: *NULL*
col 7: [ 2]  c1 15
```

最后输出游标查询结果，由于游标打开时间在更新操作之前，Oracle 需要构造一致性读，获取前镜像信息，而前镜像信息已经被覆盖，所以出现 ORA-01555 错误，这是最常见的 1555 错误来源，时间过长的查询很容易因 ORA-01555 错误而失败：

```
SQL> print :comp
ERROR:
ORA-01555: snapshot too old: rollback segment number 21 with name "RBS01" too small

no rows selected
```

再来看看第二种情况，同样构造一个游标打开进行查询：

```
SQL> var comp refcursor
SQL> begin
  2   open :comp for select * from emp where empno=7788;
  3   end;
  4   /

PL/SQL procedure successfully completed.
```

然后更新数据：

```
SQL> update emp set sal=4000 where empno=7788;

1 row updated.
SQL> alter system dump undo header RBS01;

System altered.
```

此时的 Undo 事务表事务槽信息如下：

index	state	cflags	wrap#	uel	scn	dba	parent-xid	nub
:tmt_num								

0x43	10	0x80	0x03bd	0x0001	0x0819.0045f09c	0x010000b7	0x0000.000.00000000
0x00000001	0x00000000						

在提交之前，强制刷新 Buffer Cache，写出脏数据：

```
SQL> alter session set events = 'immediate trace name flush_cache';
```

Session altered.

然后提交，此时，Oracle 将执行延迟块清除：

```
SQL> commit;
```

Commit complete.

```
SQL> alter system dump datafile 1 block 23642;
```

System altered.

```
SQL> alter system dump undo header RBS01;
```

System altered.

由于事务已经提交，回滚段事务表已经被标记为非活动，此时的提交 SCN 为 **0x0819.0045f09e**：

index	state	cflags	wrap#	uel	scn	dba	parent-xid	nub
0x43	9	0x80	0x03bd	0xffff	0x0819.0045f09e	0x010000b7	0x0000.000.00000000	
0x00000001	0x00000000							

此时的数据块已经被写出到数据文件，Oracle 执行延迟块清除，ITL 事务槽及锁定信息仍然在 Block 上存在：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.043.000003bd	0x010000b7.2b7b.04	----	1	fsc 0x0000.00000000
0x02	0x0015.041.000003bd	0x010000b7.2b7b.02	C---	0	scn 0x0819.0045f093

执行批处理更新，覆盖前镜像信息：

```
SQL>
```

```
SQL> begin
```

```
2   for i in 1 .. 100 loop
```

```
3     update emp set sal=4000;
```

```
4     rollback;
```

```
5   end loop;
```

```
6 end;
```

7 /

PL/SQL procedure successfully completed.

此时的 ORA-01555 错误就是因为延迟块清除所导致的：

```
SQL> print :comp
```

ERROR:

ORA-01555: snapshot too old: rollback segment number 21 with name "RBS01" too small

no rows selected

此时的延迟块清除被跟踪记录：

Begin cleaning out block ...

Found all committed transactions

Block cleanout record, scn: 0xffff.ffffffff ver: 0x01 opt: 0x01, entries follow...

itli: 1 flg: 2 scn: 0x0819.0045f09e

Block cleanout under the cache...

Block cleanout record, scn: 0x0819.0045f09f ver: 0x01 opt: 0x01, entries follow...

itli: 1 flg: 2 scn: 0x0819.0045f09e

... clean out dump complete.

如果此时 Dump 日志文件，也可以看到块清除信息（看到此时，由于延迟块清除也产生了日志，也就是在查询时可能看到的日志生成）：

REDO RECORD - Thread:1 RBA: 0x000068.00000006.0010 LEN: 0x003c VLD: 0x01

SCN: 0x0819.0045f09f SUBSCN: 1 04/20/2006 21:50:33

CHANGE #1 TYP:0 CLS: 1 AFN:1 DBA:0x00405c5a SCN:0x0819.0045f09c SEQ: 1 OP:4.1

Block cleanout record, scn: 0x0819.0045f09f ver: 0x01 opt: 0x01, entries follow...

itli: 1 flg: 2 scn: 0x0819.0045f09e

这里的 OP（Operation Code）：4.1 就是指 Block Cleanout。来看一下此时的数据块信息：

```
SQL> alter system dump datafile 1 block 23642;
```

System altered.

数据块的 ITL 和锁定信息都被清除：

Itl	Xid	Uba	Flag	Lck	Scn/Fsc
0x01	0x0015.043.000003bd	0x010000b7.2b7b.04	C---	0	scn 0x0819.0045f09e
0x02	0x0015.041.000003bd	0x010000b7.2b7b.02	C---	0	scn 0x0819.0045f093

需要注意的是，可能存在另外一种情况，就是当执行延迟块清除时，回滚段或原回滚表空间已经被删除，此时 Oracle 仍然可以通过字典表 UNDO\$ 来获得 SCN 信息，执行块清除。

关于 Oracle 的提交处理及块清除机制是一个极其复杂的过程，这里对这部分内容进行了

适当简化说明，旨在让大家能够对 Oracle 的回滚机制，块清除机制有所了解。

7.10 AUM 下如何重建 Undo 表空间

曾经有朋友问到，在迁移（同平台）的时候由于 Undo 表空间过大，不打算要现在的 Undo 文件，想要重建一个，该如何做，是否需要通过一些隐含参数来做特殊处理？这个前提是拥有一个有效的冷备份（或者 Clean Shutdown 的数据库）。拥有冷备份这个操作是很简单的，也不需要隐使用隐含参数。

以下是一个简单的测试过程，重建 Undo 表空间的步骤和此类似。

（1）假定拥有一个 Clean shutdown 的数据库：

```
C:\Documents and Settings\gqgai>sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.6.0 - Production on Fri Mar 4 20:55:59 2005

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.6.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.6.0 - Production

20:55:59 SQL> shutdown immediate;

Database closed.

Database dismounted.

ORACLE instance shut down.
```

（2）同平台迁移时可以放弃 Undo 表空间，这时候启动会报错 ORA-01157：

```
20:56:14 SQL> startup

ORACLE instance started.

Total System Global Area  59842188 bytes
Fixed Size                  454284 bytes
Variable Size              33554432 bytes
Database Buffers           25165824 bytes
Redo Buffers                667648 bytes
Database mounted.

ORA-01157: cannot identify/lock data file 2 - see DBWR trace file
ORA-01110: data file 2: 'D:\ORADATA\EYGLE\UNDOTBS01.DBF'
```

(3) 删除 Undo 文件启动数据库:

```
20:59:28 SQL> alter database datafile 'D:\ORADATA\EYGLE\UNDOTBS01.DBF' offline drop;
```

Database altered.

Elapsed: 00:00:00.00

```
20:59:34 SQL> alter database open;
```

Database altered.

Elapsed: 00:00:02.04

```
20:59:40 SQL> select name from v$datafile;
```

NAME

D:\ORADATA\EYGLE\SYSTEM01.DBF

D:\ORADATA\EYGLE\UNDOTBS01.DBF

D:\ORADATA\EYGLE\PERFSTAT01.DBF

D:\ORADATA\EYGLE\EYGLE.DBF

(4) 重建 Undo 表空间, 并切换为当前 Undo 表空间:

```
21:00:55 SQL> create undo tablespace undotbs2
```

```
21:01:03 2 datafile 'd:\oradata\eygle\undotbs2.dbf' size 10M;
```

Tablespace created.

Elapsed: 00:00:02.02

```
21:06:29 SQL> ALTER SYSTEM SET undo_tablespace='UNDOTBS2';
```

System altered.

Elapsed: 00:00:00.01

然后数据库即可恢复正常使用。

7.11 诊断案例一: 使用 Flashback Query 恢复误删除数据

这是一个实际生产环境中的恢复案例, 当时接到研发工程师的电话, 说误删除了部分重

要数据，并且已经提交，需要恢复。

于是登录到数据库上查看，由于是 Oracle 9iR2，首先尝试使用 Flashback Query 闪回数据。首先确认数据库的 SCN 变化：

```
SQL> col fscn for 99999999999999999999
SQL> col nscn for 99999999999999999999
SQL> select name,FIRST_CHANGE# fscn,NEXT_CHANGE# nscn,FIRST_TIME from v$sarchived_log;
```

NAME	FSCN	NSCN	FIRST_TIME
.....			
/mwarch/oracle/1_52414.dbf	12929942881	12929943706	2005-06-22 14:38:32
/mwarch/oracle/1_52415.dbf	12929943706	12929944623	2005-06-22 14:38:35
/mwarch/oracle/1_52416.dbf	12929944623	12929945392	2005-06-22 14:38:38
/mwarch/oracle/1_52417.dbf	12929945392	12929945888	2005-06-22 14:38:41
/mwarch/oracle/1_52418.dbf	12929945888	12929945965	2005-06-22 14:38:44
/mwarch/oracle/1_52419.dbf	12929945965	12929948945	2005-06-22 14:38:45
/mwarch/oracle/1_52420.dbf	12929948945	12929949904	2005-06-22 14:46:05
/mwarch/oracle/1_52421.dbf	12929949904	12929950854	2005-06-22 14:46:08
/mwarch/oracle/1_52422.dbf	12929950854	12929951751	2005-06-22 14:46:11
/mwarch/oracle/1_52423.dbf	12929951751	12929952587	2005-06-22 14:46:14
.....			
/mwarch/oracle/1_52498.dbf	12930138975	12930139212	2005-06-22 15:55:57
/mwarch/oracle/1_52499.dbf	12930139212	12930139446	2005-06-22 15:55:59
/mwarch/oracle/1_52500.dbf	12930139446	12930139682	2005-06-22 15:56:00
/mwarch/oracle/1_52501.dbf	12930139682	12930139915	2005-06-22 15:56:02
/mwarch/oracle/1_52502.dbf	12930139915	12930140149	2005-06-22 15:56:03
/mwarch/oracle/1_52503.dbf	12930140149	12930140379	2005-06-22 15:56:05
/mwarch/oracle/1_52504.dbf	12930140379	12930140610	2005-06-22 15:56:05
/mwarch/oracle/1_52505.dbf	12930140610	12930140845	2005-06-22 15:56:07

14811 rows selected.

当前的 SCN 为：

```
SQL> select dbms_flashback.get_system_change_number fscn from dual;
```

FSCN
.....
12930142214

使用应用用户尝试闪回：

```
SQL> connect username/password
```

Connected.

现有数据：

```
SQL> select count(*) from hs_passport;
```

```

COUNT(*)
-----
      851998

```

创建恢复表：

```
SQL> create table hs_passport_recov as select * from hs_passport where 1=0;
```

Table created.

选择合适的 SCN 向前恢复：

```
SQL> select count(*) from hs_passport as of scn 12929970422;
```

```

COUNT(*)
-----
      861686

```

尝试多个 SCN，获取最佳值（如果能得知具体时间，那么可以获得准确的数据闪回）：

```
SQL> select count(*) from hs_passport as of scn &scn;
```

Enter value for scn: 12929941968

```
old   1: select count(*) from hs_passport as of scn &scn
```

```
new   1: select count(*) from hs_passport as of scn 12929941968
```

```

COUNT(*)
-----
      861684

```

```
SQL> /
```

Enter value for scn: 12929928784

```
old   1: select count(*) from hs_passport as of scn &scn
```

```
new   1: select count(*) from hs_passport as of scn 12929928784
```

```

COUNT(*)
-----

```

825110

SQL> /

Enter value for scn: 12928000000

old 1: select count(*) from hs_passport as of scn &scn

new 1: select count(*) from hs_passport as of scn 12928000000

select count(*) from hs_passport as of scn 12928000000

*

ERROR at line 1:

ORA-01466: unable to read data - table definition has changed

最后选择恢复到 SCN 为 12929941968 的时间点:

SQL> insert into hs_passport_recov select * from hs_passport as of scn 12929941968;

861684 rows created.

SQL> commit;

Commit complete.

研发人员确认, 已经可以满足需要, 找回误删除部分数据, 至此闪回恢复成功完成。

7.12 诊断案例二: 释放过度扩展的 Undo 空间

从 Oracle 9i 开始, 当使用 AUM 管理时, 通常会选择设置 Undo 表空间自动扩展, 这就带来了另外一个问题, 经常会出现 Undo 表空间过度扩展而不能回缩的问题。这类问题有的是因为 Bug 引起的, 以下的案例来自 Oracle 10g, 但是对于 Oracle 9i 同样适用。

系统环境如下:

OS: Red Hat Enterprise Linux AS release 4 (Nahant)

DB: Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production

一台 Oracle 10gR2 数据库报出如下错误:

ORA-1653: unable to extend table SYSMAN.MGMT_SYSTEM_ERROR_LOG by 8 in tablespace SYSAUX

ORA-1653: unable to extend table SYSMAN.MGMT_SYSTEM_ERROR_LOG by 8 in tablespace SYSAUX

ORA-1653: unable to extend table SYSMAN.MGMT_SYSTEM_ERROR_LOG by 8 in tablespace SYSAUX

ORA-1653: unable to extend table SYSMAN.MGMT_SYSTEM_ERROR_LOG by 8 in tablespace SYSAUX

ORA-1653: unable to extend table SYSMAN.MGMT_SYSTEM_ERROR_LOG by 8 in tablespace SYSAUX

登录检查, 发现 SYSAUX 表空间空间用尽, 不能扩展, 尝试手工扩展 SYSAUX 表空间:

alter database datafile '+ORADG/danally/datafile/sysaux.266.600173881' resize 800m

Tue Nov 29 23:31:38 2005

```
ORA-1237 signalled during: alter database datafile '+ORADG/danaly/datafile/sysaux.266.600173881' resize
300m...
```

出现 ORA-1237 错误，提示空间不足。这时候我才认识到是磁盘空间可能被用完了。是谁“偷偷地”用了那么多空间呢（本来有几十个 GB 的 Free 磁盘空间的）？

检查数据库表空间占用空间情况：

```
SQL> select tablespace_name,sum(bytes)/1024/1024/1024 GB
2   from dba_data_files group by tablespace_name
3   union all
4   select tablespace_name,sum(bytes)/1024/1024/1024 GB
5   from dba_temp_files group by tablespace_name order by GB;
```

TABLESPACE_NAME	GB
UNDOTBS2	.09765625
SYSTEM	.478515625
SYSAUX	.634765625
.....	
IVRCN_TS_DATA	2
MMS_TS_DATA1	2
CM_TS_DEFAULT	5
TEMP	20.5498047
UNDOTBS1	27.1582031

15 rows selected.

不幸地，发现 Undo 表空间已经扩展至 27GB，而 TEMP 表空间也扩展至 20GB，这两个表空间加起来占用了 47GB 的磁盘空间，因而导致了空间不足。

显然曾经有大事务占用了大量的 Undo 表空间和 Temp 表空间，Oracle 的 AUM (Auto Undo Management) 从出生以来就经常出现只扩展、不收缩 (shrink) 的情况（通常可以设置足够的 Undo 表空间大小，然后取消其自动扩展属性）。

现在可以采用以下步骤回收 Undo 空间。

(1) 确认文件：

```
SQL> select file_name,bytes/1024/1024 from dba_data_files
2   where tablespace_name like 'UNDOTBS1';
```

FILE_NAME

BYTES/1024/1024

```
-----
+ORADG/danaly/datafile/undotbs1.265.600173875
27810
```

(2) 检查 Undo Segment 状态，发现有的回滚段大小已经扩展到了约 3GB 的大小：

```
SQL> select usn,xacts,rssize/1024/1024/1024,hwmsize/1024/1024/1024,shrinks
2 from v$rollstat order by rssize;
```

USN	XACTS	RSSIZE/1024/1024/1024	HWMSIZE/1024/1024/1024	SHRINKS
0	0	.000358582	.000358582	0
2	0	.071517944	.071517944	0
3	0	.13722229	.13722229	0
9	0	.236984253	.236984253	0
10	0	.625144958	.625144958	0
5	1	1.22946167	1.22946167	0
8	0	1.27175903	1.27175903	0
4	1	1.27895355	1.27895355	0
7	0	1.56770325	1.56770325	0
1	0	2.02474976	2.02474976	0
6	0	2.9671936	2.9671936	0

11 rows selected.

(3) 创建新的 Undo 表空间：

```
SQL> create undo tablespace undotbs2;
```

Tablespace created.

(4) 切换 Undo 表空间为新的 Undo 表空间：

```
SQL> alter system set undo_tablespace=undotbs2 scope=both;
```

System altered.

—— 注 意 ——

此处使用 spfile 时，需要让修改同时变更到 spfile 文件。

(5) 等待原 Undo 表空间所有 UNDO SEGMENT OFFLINE：

```
SQL> select usn,xacts,status,rssize/1024/1024/1024,hwmsize/1024/1024/1024,shrinks
2 from v$rollstat order by rssize;
```

USN	XACTS STATUS	RSSIZE/1024/1024/1024	HWMSIZE/1024/1024/1024	SHRINKS
14	0 ONLINE	.000114441	.000114441	0
19	0 ONLINE	.000114441	.000114441	0
11	0 ONLINE	.000114441	.000114441	0
12	0 ONLINE	.000114441	.000114441	0
13	0 ONLINE	.000114441	.000114441	0
20	0 ONLINE	.000114441	.000114441	0
15	1 ONLINE	.000114441	.000114441	0
16	0 ONLINE	.000114441	.000114441	0
17	0 ONLINE	.000114441	.000114441	0
18	0 ONLINE	.000114441	.000114441	0
0	0 ONLINE	.000358582	.000358582	0
6	0 PENDING OFFLINE	2.9671936	2.9671936	0

12 rows selected.

确认原回滚表空间所有回滚段都正常 OFFLINE:

11:32:11 SQL> /

USN	XACTS STATUS	RSSIZE/1024/1024/1024	HWMSIZE/1024/1024/1024	SHRINKS
15	1 ONLINE	.000114441	.000114441	0
11	0 ONLINE	.000114441	.000114441	0
12	0 ONLINE	.000114441	.000114441	0
13	0 ONLINE	.000114441	.000114441	0
14	0 ONLINE	.000114441	.000114441	0
20	0 ONLINE	.000114441	.000114441	0
16	0 ONLINE	.000114441	.000114441	0
17	0 ONLINE	.000114441	.000114441	0
18	0 ONLINE	.000114441	.000114441	0
19	0 ONLINE	.000114441	.000114441	0
0	0 ONLINE	.000358582	.000358582	0

11 rows selected.

(6) 删除原 Undo 表空间:

```
11:34:00 SQL> drop tablespace undotbs1 including contents;
```

```
Tablespace dropped.
```

(7) 检查空间情况，由于使用了 ASM 管理，可以用 10gR2 提供的 asmcmd 工具来查看空间占用情况：

```
[oracle@danaly ~]$ export ORACLE_SID=+ASM
```

```
[oracle@danaly ~]$ asmcmd
```

```
ASMCMD> du
```

```
Used_MB      Mirror_used_MB
```

```
21625        21625
```

```
ASMCMD> exit
```

此时空间已经释放。本案例包含了常规 Undo 表空间重建、切换等过程，这也是 Undo 表空间常规维护操作之一，需要熟记。

7.13 特殊情况的恢复

在很多情况下，特别是在使用隐含参数强制打开数据库之后，可能会在出现 ORA-00600 4194 错误，在 alert 文件中，记录主要错误日志：

```
Sat Jan 21 13:55:21 2006
```

```
Errors in file /opt/oracle/admin/conner/bdump/conner_smon_17113.trc:
```

```
ORA-00600: internal error code, arguments: [4194], [43], [46], [], [], [], []
```

```
Sat Jan 21 13:55:21 2006
```

```
Errors in file /opt/oracle/admin/conner/udump/conner_ora_17121.trc:
```

```
ORA-00600: internal error code, arguments: [4194], [45], [44], [], [], [], []
```

4194 错误通常说明 Undo 段出现问题，最好的办法是通过备份进行恢复，如果没有备份，那么可以通过特殊的初始化参数进行强制启动，本文就 Oracle 的隐含参数进行恢复说明（由于实际情况可能各不相同，进行测试前请先行备份），仅供参考。

首先确定当前回滚段名称，这可以从 alert 文件中获得：

```
Sat Jan 21 13:55:21 2006
```

```
Undo Segment 11 Onlined
```

```
Undo Segment 12 Onlined
```

```
Undo Segment 13 Onlined
```

```
Successfully onlined Undo Tablespace 16.
```

对应的 AUM（Auto Undo Management）下的回滚段名称为：

```
'_SYSSMU11$','_SYSSMU12$','_SYSSMU13$'
```

修改 init<sid>.ora 参数文件，使用 Oracle 隐含参数_corrupted_rollback_segments 将回滚段

标记为损坏，此时启动数据库，Oracle 会跳过对于这些回滚段的相关操作，强制启动数据库。

```
._corrupted_rollback_segments='_SYSSMU11$','_SYSSMU12$','_SYSSMU13$'
```

使用 init<sid>.ora 参数文件启动数据库：

```
[oracle@jumper dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 9.2.0.4.0 - Production on Sat Jan 21 13:56:47 2006

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to an idle instance.

SQL> startup pfile=initconner.ora

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.
```

此时数据库正常 Open。观察 alert 文件可以获得如下信息：

```
Sat Jan 21 13:57:03 2006
SMON: enabling tx recovery
SMON: about to recover undo segment 11
SMON: mark undo segment 11 as needs recovery
SMON: about to recover undo segment 12
SMON: mark undo segment 12 as needs recovery
SMON: about to recover undo segment 13
SMON: mark undo segment 13 as needs recovery
Sat Jan 21 13:57:03 2006
Database Characterset is ZHS16GBK
Sat Jan 21 13:57:03 2006
SMON: about to recover undo segment 11
SMON: mark undo segment 11 as needs recovery
SMON: about to recover undo segment 12
SMON: mark undo segment 12 as needs recovery
SMON: about to recover undo segment 13
SMON: mark undo segment 13 as needs recovery
Sat Jan 21 13:57:04 2006
Created Undo Segment _SYSSMU1$
```

```

Undo Segment 1 Onlined
Completed: ALTER DATABASE OPEN
Sat Jan 21 14:02:11 2006
SMON: about to recover undo segment 11
SMON: mark undo segment 11 as needs recovery
SMON: about to recover undo segment 12
SMON: mark undo segment 12 as needs recovery
SMON: about to recover undo segment 13
SMON: mark undo segment 13 as needs recovery

```

此时可以重新创建新的 Undo 表空间，删除出现问题的表空间，修改参数文件，由参数文件生成新的 spfile，重新启动数据库：

```

SQL> create undo tablespace undotbs1
      2  datafile '/opt/oracle/oradata/conner/undotbs1.dbf' size 10M;
Tablespace created.
SQL> alter system set undo_tablespace=undotbs1;
System altered.

SQL> drop tablespace undotbs2;
Tablespace dropped.

```

此时的 alert 文件记录如下：

```

Sat Jan 21 14:03:29 2006
create undo tablespace undotbs1
datafile '/opt/oracle/oradata/conner/undotbs1.dbf' size 10M
Sat Jan 21 14:03:29 2006
Created Undo Segment _SYSSMU2$
Created Undo Segment _SYSSMU3$
Created Undo Segment _SYSSMU4$
Created Undo Segment _SYSSMU5$
Created Undo Segment _SYSSMU6$
Created Undo Segment _SYSSMU7$
Created Undo Segment _SYSSMU8$
Created Undo Segment _SYSSMU9$
Created Undo Segment _SYSSMU10$
Created Undo Segment _SYSSMU14$
Starting control autobackup
Control autobackup written to DISK device

```

```

        handle '/opt/oracle/product/9.2.0/dbs/c-3152029224-20060121-00'

Completed: create undo tablespace undotbs1
datafile '/opt/ora
Sat Jan 21 14:03:43 2006
Undo Segment 2 Onlined
Undo Segment 3 Onlined
Undo Segment 4 Onlined
Undo Segment 5 Onlined
Undo Segment 6 Onlined
Undo Segment 7 Onlined
Undo Segment 8 Onlined
Undo Segment 9 Onlined
Undo Segment 10 Onlined
Undo Segment 14 Onlined
Successfully onlined Undo Tablespace 1.
Undo Segment 1 Offlined
Undo Tablespace 16 successfully switched out.
Sat Jan 21 14:03:43 2006
ALTER SYSTEM SET undo_tablespace='UNDOTBS1' SCOPE=MEMORY;
Sat Jan 21 14:07:18 2006
SMON: about to recover undo segment 11
SMON: mark undo segment 11 as needs recovery
SMON: about to recover undo segment 12
SMON: mark undo segment 12 as needs recovery
SMON: about to recover undo segment 13
SMON: mark undo segment 13 as needs recovery
Sat Jan 21 14:08:06 2006
drop tablespace undotbs2
Sat Jan 21 14:08:07 2006
Starting control autobackup
Control autobackup written to DISK device
        handle '/opt/oracle/product/9.2.0/dbs/c-3152029224-20060121-01'

Completed: drop tablespace undotbs2

```

修改参数文件，变更 Undo 表空间，并取消 `_corrupted_rollback_segments` 设置：

```
*.undo_tablespace='UNDOTBS1'
```

由参数文件创建 spfile 文件。

```

SQL> create spfile from pfile;
File created.
SQL> shutdown immediate;
Database closed.
Database dismounted.
ORACLE instance shut down.
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.

```

重启数据库，观察 alert 文件：

```

Sat Jan 21 14:08:36 2006
Undo Segment 2 Onlined
Undo Segment 3 Onlined
Undo Segment 4 Onlined
Undo Segment 5 Onlined
Undo Segment 6 Onlined
Undo Segment 7 Onlined
Undo Segment 8 Onlined
Undo Segment 9 Onlined
Undo Segment 10 Onlined
Undo Segment 14 Onlined
Successfully onlined Undo Tablespace 1.

```

此时数据库可以正常启动，通过以上方法恢复数据库，通常会导致数据库内部存在不一致的状况，通常建议立即进行全库 exp，然后重新建库，再通过 imp 恢复数据库。

7.14 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，本节简单介绍 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) from dual;
```

```
DUMP(1)
```

```
-----
```

```
Typ=2 Len=2: 193,2
```

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]

主要有以下几个组成部分。

(1) 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。

■ 在存储上，Oracle 对正数和负数分别进行存储转换。

正数：加 1 存储（为了避免 Null）。

负数：被 101 减，如果总长度小于 21 个字节，最后加一个 102（是为了排序的需要）。

■ 指数位换算。

正数：指数=符号/指数位 - 193（最高位为 1 是代表正数）

负数：指数=62 - 第一字节

(4) 从<数字 1>开始是有效的数据位。

从<数字 1>开始是最高有效位，所存储的数值计算方法为： $\sum \text{每个<数字位>} \times 100^{\text{（指数 N）}}$ ，其中 N 是有效位数的顺序位，第一个有效位的 N=0。

下面举例进一步说明：

```
SQL> select dump(123456.789) from dual;
```

```
DUMP(123456.789)
```

```
-----
```

```
Typ=2 Len=6: 195,13,35,57,79,91
```

<数字 1> 13 - 1 = $12 \times 100^{(2-0)}$ 120000

<数字 2> 35 - 1 = $34 \times 100^{(2-1)}$ 3400

<指数>: 195 - 193 = 2

<数字 3> 57 - 1 = $56 \times 100^{(2-2)}$ 56

<数字 4> 79 - 1 = $78 \times 100^{(2-3)}$.78

<数字 5> 91 - 1 = $90 \times 100^{(2-4)}$.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> $62 - 60 = 2$ (最高位是 0, 代表为负数)

<数字 1> $101 - 89 = 12 \times 100^{(2-0)}$ 120000

<数字 2> $101 - 67 = 34 \times 100^{(2-1)}$ 3400

<数字 3> $101 - 45 = 56 \times 100^{(2-2)}$ 56

<数字 4> $101 - 23 = 78 \times 100^{(2-3)}$.78

<数字 5> $101 - 11 = 90 \times 100^{(2-4)}$.009

123456.789(-)

现在再考虑一下为什么在最后加 102 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。

很长时间以来，通过什么样的手段来衡量数据库的状况、发现数据库的问题、优化数据库的性能一直是人们广为争论的话题。从 Oracle 7.1.12 开始，Oracle 引入了等待事件，随即等待事件成为了数据库性能优化的一个重要指导。

当一个进程连接到数据库之后，进程所经历的种种等待就开始被记录，并且通过一系列的动态性能视图进行展现。通过等待事件用户可以很快地发现数据库的性能瓶颈，从而进行针对性能的优化和分析。本章将着重介绍等待事件在 Oracle 研究及优化过程中的作用。

8.1 等待事件的源起

等待事件的概念是在 Oracle 7.0.12 中引入的，大致有 100 个等待事件。在 Oracle 8.0 中这个数目增加到了大约 150 个，在 Oracle 8i 中大约有 220 个事件，在 Oracle 9iR2 中大约有 400 个等待事件，而在最近的 Oracle 10gR2 中，大约有 874 个等待事件。

虽然不同的版本和组件安装可能会有不同数目的等待事件，但是这些等待事件都可以通过查询 V\$EVENT_NAME 视图获得：

```
SQL> select * from v$version;
```

BANNER

```
-----  
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod  
PL/SQL Release 10.2.0.1.0 - Production  
CORE      10.2.0.1.0      Production  
TNS for Linux: Version 10.2.0.1.0 - Production  
NLSRTL Version 10.2.0.1.0 - Production
```

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

```
COUNT(*)
```

```
-----
      874
```

研究 Oracle 的等待事件，V\$EVENT_NAME 视图是一个很好的开始，这个视图记录着当前数据库支持的等待事件及其基本信息。

Oracle 的等待事件，主要可以分为两类，即空闲（idle）等待事件和非空闲（non-idle）等待事件。空闲事件指 Oracle 正等待某种工作，在诊断和优化数据库的时候，不用过多注意这部分事件。非空闲等待事件专门针对 Oracle 的活动，指数据库任务或应用运行过程中发生的等待，这些等待事件是，在调整数据库的时候应该关注与研究的。

在 Oracle 10g 之前，Oracle 的 Statspack 会创建一个视图 stats\$idle_event 记录空闲等待事件：

```
SQL> select * from stats$idle_event;
```

```
EVENT
```

```
-----
smon timer
```

```
pmon timer
```

```
rdbms ipc message
```

```
Null event
```

```
parallel query dequeue
```

```
pipe get
```

```
client message
```

```
SQL*Net message to client
```

```
SQL*Net message from client
```

```
SQL*Net more data from client
```

```
dispatcher timer
```

```
virtual circuit status
```

```
lock manager wait for remote message
```

```
PX Idle Wait
```

```
wakeup time manager
```

```
15 rows selected.
```

从 Oracle 10g 开始，Oracle 对等待事件进行了更为详细的分类，V\$EVENT_NAME 视图也增加了相关分类的字段：

```
SQL> desc v$event_name
```

```
  Name
```

```
Null?
```

```
  Type
```

```

-----
EVENT#                                NUMBER
EVENT_ID                             NUMBER
NAME                                  VARCHAR2(64)
PARAMETER1                           VARCHAR2(64)
PARAMETER2                           VARCHAR2(64)
PARAMETER3                           VARCHAR2(64)
WAIT_CLASS_ID                         NUMBER
WAIT_CLASS#                           NUMBER
WAIT_CLASS                            VARCHAR2(64)

```

V\$EVENT_NAME 视图中的 PARAMETER1、PARAMETER2、PARAMETER3 非常重要，对于不同的等待事件参数的意义各不相同：

```

SQL> select name,PARAMETER1,PARAMETER2,PARAMETER3 from v$event_name
      2  where name ='db file scattered read';

```

NAME	PARAMETER1	PARAMETER2	PARAMETER3
db file scattered read	file#	block#	blocks

看一下 Oracle 10gR2 中主要分类及各类等待事件的个数：

```

SQL> SELECT   wait_class#, wait_class_id, wait_class, COUNT (*) AS "count"
      2  FROM v$event_name
      3  GROUP BY wait_class#, wait_class_id, wait_class
      4  ORDER BY wait_class#
      5  /

```

WAIT_CLASS#	WAIT_CLASS_ID	WAIT_CLASS	count
0	1893977003	Other	590
1	4217450380	Application	12
2	3290255840	Configuration	23
3	4166625743	Administrative	46
4	3875070507	Concurrency	24
5	3386400367	Commit	1
6	2723168908	Idle	62
7	2000153315	Network	26
8	1740759767	User I/O	17

9	4108307767 System I/O	24
10	2396326234 Scheduler	2
11	3871361733 Cluster	47

12 rows selected.

也可以通过查询 `V$SYSTEM_WAIT_CLASS` 视图获得各类主要等待事件的等待时间和等待次数等信息：

```
SQL> select * from v$system_wait_class
2 order by time_waited;
```

WAIT_CLASS_ID	WAIT_CLASS#	WAIT_CLASS	TOTAL_WAITS	TIME_WAITED
3875070507	4	Concurrency	8433	751
4217450380	1	Application	366	2558
1893977003	0	Other	15690	14765
3386400367	5	Commit	30520	49246
3290255840	2	Configuration	5701	102057
2000153315	7	Network	6261634	103300
4108307767	9	System I/O	1258815	1613868
1740759767	8	User I/O	9027852	3358285
2723168908	6	Idle	4402568	794064698

9 rows selected

在 Oracle 10g 中，主要的空闲等待事件主要有：

```
SQL> select name,wait_class from v$event_name where wait_class='Idle';
```

NAME	WAIT_CLASS
pmon timer	Idle
rdbms ipc message	Idle
i/o slave wait	Idle
wait for unread message on broadcast channel	Idle
wait for unread message on multiple broadcast channels	Idle
class slave wait	Idle
KSV master wait	Idle
watchdog main loop	Idle

DIAG idle wait	Idle
ges remote message	Idle
gcs remote message	Idle
SGA: MMAN sleep for component shrink	Idle
LNS ASYNC archive log	Idle
LNS ASYNC dest activation	Idle
LNS ASYNC end of log	Idle
.....	
virtual circuit status	Idle
dispatcher timer	Idle
jobq slave wait	Idle
pipe get	Idle
.....	
single-task message	Idle
SQL*Net message from client	Idle
SQL*Net message from dblink	Idle
PL/SQL lock timer	Idle
EMON idle wait	Idle
.....	
HS message to agent	Idle
ASM background timer	Idle
JS external job	Idle
62 rows selected.	

8.2 从等待发现瓶颈

等待事件所以为众多 DBA 所关注与研究，是因为通过等待事件可以迅速发现数据库瓶颈，并及时解决问题。在网上，我曾经发起过一个讨论，让大家列举“列举你认为最重要的 9 个动态性能视图”，很多人的回复里都选择了和等待事件相关的几个视图，它们是 V\$SESSION、V\$SESSION_WAIT 和 V\$SYSTEM_EVENT。

来看一下这几个视图的作用及意义。V\$SESSION 视图记录的是数据库当前连接的 session 信息。V\$SESSION_WAIT 视图记录的是当前数据库连接的活动 session 正在等待的资源或事件信息。由于 V\$SESSION 记录的是动态信息，和 session 的生命周期相关，而并不记录历史信息，所以 Oracle 提供另外一个视图 V\$SYSTEM_EVENT 来记录数据库自启动以来所有等待事件的汇总信息。通过 V\$SYSTEM_EVENT 视图，用户可以迅速地获得数据库运行的总体概况。

8.2.1 V\$SESSION 和 V\$SESSION_WAIT

由于 V\$SESSION 记录当前连接数据库的 SESSION 信息，而 V\$SESSION_WAIT 视图记录这些 SESSION 的等待，很多时候要联合这两个视图进行查询以获取更多的诊断信息。从 Oracle 10g 开始，为了方便用户，Oracle 开始将这两个视图进行整合。

在 Oracle 10gR1 中，Oracle 在 V\$SESSION 中增加关于等待事件的字段，实际上也就是把原来 V\$SESSION_WAIT 视图中的所有字段全部整合到了 V\$SESSION 视图中（如果进一步研究你会发现，实际上 V\$SESSION 的底层查询语句及 X\$表已经有了变化）。

这一变化使得用户的查询得以简化，但是也使得 V\$SESSION_WAIT 开始变得多余。

此外 V\$SESSION 中还增加了 BLOCKING_SESSION 等字段，以前需要通过 dba_waiters 等视图才能获得的信息，现在也可以直接从 V\$SESSION 中得到了。

在 Oracle 10gR2 中，Oracle 又为 V\$SESSION 增加了额外几个字段：SERVICE_NAME、SQL_TRACE、SQL_TRACE_WAITS 和 SQL_TRACE_BINDS。这几个字段显示了当前 session 连接方式及是否启用了 sql_trace 跟踪等。

在新的数据库版本中，Oracle 在小处动的手脚也是非常多的，而这些小手脚无疑会给用户的管理维护带来极大的方便。

以下是 Oracle 9iR2 中 V\$SESSION_WAIT 视图的结构：

```
SQL> desc v$session_wait
```

Name	Null?	Type
SID		NUMBER
SEQ#		NUMBER
EVENT		VARCHAR2(64)
P1TEXT		VARCHAR2(64)
P1		NUMBER
P1RAW		RAW(4)
P2TEXT		VARCHAR2(64)
P2		NUMBER
P2RAW		RAW(4)
P3TEXT		VARCHAR2(64)
P3		NUMBER
P3RAW		RAW(4)
WAIT_TIME		NUMBER
SECONDS_IN_WAIT		NUMBER
STATE		VARCHAR2(19)

其中 Event 代表等待事件的名称；p<n>text 用以描述具体的参数；p<n>分别代表以十进

制定义的参数 (Parameter) 参数值; p<n>Raw 是以十六进制表示的参数值。

对于不同 Event, 具体参数表示的含义也不相同, 用户可以通过 V\$EVENT_NAME 视图来查看这些参数的定义。

8.2.2 从 V\$SQLTEXT 中追踪

在数据库出现瓶颈时, 通常可以从 V\$SESSION_WAIT 找到那些正在等待资源的 session, 通过 session 的 sid, 联合 V\$SESSION 和 V\$SQLTEXT 视图就可以捕获这些 session 正在执行的 SQL 语句。

以下是一个生产数据库的问题诊断和解决过程, 读者可以从中体会一下等待事件在解决问题中的指导作用。该生产环境的操作系统为 Solaris 8, 数据库版本为 8.1.7.4, 业务及开发人员报告系统运行缓慢, 已经影响业务系统正常使用, 请求协助诊断。

数据库运行缓慢, 可以判断数据库可能经历了等待, 那么可以通过 V\$SESSION_WAIT 视图来入手。查询 V\$SESSION_WAIT 获取各进程等待事件:

```
SQL> select sid,event,p1,p1text from v$session_wait;
```

SID EVENT	P1 P1TEXT
124 latch free	1.6144E+10 address
1 pmon timer	300 duration
2 rdbms ipc message	300 timeout
.....	
140 buffer busy waits	17 file#
66 buffer busy waits	17 file#
10 db file sequential read	17 file#
18 db file sequential read	17 file#
54 db file sequential read	17 file#
49 db file sequential read	17 file#
48 db file sequential read	17 file#
46 db file sequential read	17 file#
45 db file sequential read	17 file#
.....	
244 rows selected.	

对于本案例, 发现存在大量 db file scattered read 及 db file sequential read 等待, 并且全表扫描的等待都位于文件号为 17 的数据文件上。显然全表扫描等操作成为系统最严重的性能影响因素。

— 说 明 —

db file scattered read (DB 文件分散读取) 这种情况通常显示与全表扫描相关的等待。当数据库进行全表扫描时, 基于性能的考虑, 数据会分散 (scattered) 读入 Buffer Cache。如果这个等待事件比较显著, 可能说明对于某些全表扫描的表, 没有创建索引或者没有创建合适的索引, 可能需要检查这些数据表已确定是否进行了正确的设置。

然而这个等待事件不一定意味着性能低下, 在某些条件下 Oracle 会主动使用全表扫描来替换索引扫描以提高性能, 这和访问的数据量有关, 在 CBO 下 Oracle 会进行更为智能的选择, 在 RBO 下 Oracle 更倾向于使用索引。

8.2.3 捕获相关 SQL

确定这些进程因为数据访问产生了等待, 考虑捕获这些 SQL 以发现问题。这里用到了以下脚本 getsqlbysid.sql, 该脚本通过已知 session 的 sid, 联合 V\$SESSION、V\$SQLTEXT 视图, 获得相关 session 正在执行的完整 SQL 语句。

```
SELECT    sql_text
          FROM v$sqltext a
          WHERE a.hash_value = (SELECT sql_hash_value
                                FROM v$session b
                                WHERE b.SID = '&sid')

          ORDER BY piece ASC
/
```

使用该脚本, 通过从 V\$SESSION_WAIT 中获得的等待全表或索引扫描的进程 sid, 捕获问题 SQL:

```
SQL> @getsqlbysid
Enter value for sid: 18

old   5: where b.sid='&sid'
new   5: where b.sid='18'

SQL_TEXT
-----

select i.vc2title,i.numinfoguid  from  hs_info i where i.intenab
ledflag = 1  and i.intpublishstate = 1  and i.datpublishdate <=
sysdate  and i.numcatalogguid = 2047 order by i.datpublishdate d
esc, i.numorder desc

SQL> /
Enter value for sid: 54

old   5: where b.sid='&sid'
```

```
new    5: where b.sid='54'
```

SQL_TEXT

```
-----
select i.vc2title,i.numinfoguid  from  hs_info i where i.intenab
ledflag = 1  and i.intpublishstate = 1  and i.datpublishdate <=
sysdate  and i.numcatalogguid = 33 order by i.datpublishdate des
c, i.numorder desc
```

SQL> /

Enter value for sid: 49

```
old    5: where b.sid='&sid'
```

```
new    5: where b.sid='49'
```

SQL_TEXT

```
-----
select i.vc2title,i.numinfoguid  from  hs_info i where i.intenab
ledflag = 1  and i.intpublishstate = 1  and i.datpublishdate <=
sysdate  and i.numcatalogguid = 26 order by i.datpublishdate des
c, i.numorder desc
```

对几个进程进行跟踪，分别得到以上 SQL 语句，这些 SQL 可能就是问题产生的根源。以上语句如果良好编码应该使用绑定变量，但是目前这个不是我们关心的。

使用该应用用户连接，检查以上 SQL 的执行计划：

```
SQL> set autotrace trace explain
```

```
SQL> select i.vc2title,i.numinfoguid
2  from  hs_info i where i.intenabedflag = 1
3  and i.intpublishstate = 1  and i.datpublishdate <=sysdate
4  and i.numcatalogguid = 3475
5  order by i.datpublishdate desc, i.numorder desc  ;
```

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=228 Card=1 Bytes=106)
1      0      SORT (ORDER BY) (Cost=228 Card=1 Bytes=106)
2      1      TABLE ACCESS (FULL) OF 'HS_INFO' (Cost=218 Card=1 Bytes=106)
```

```
SQL> select count(*) from hs_info;
```

```
COUNT(*)
```

```
-----
```

```
227404
```

通过执行计划，看到以上查询使用了全表扫描，而该表这里有 22 万记录，全表扫描已经不再适合。检查该表，存在以下索引：

```
SQL> select index_name,index_type from user_indexes where table_name='HS_INFO';
```

INDEX_NAME	INDEX_TYPE
HSIDX_INFO1	FUNCTION-BASED NORMAL
HSIDX_INFO_SEARCHKEY	DOMAIN
PK_HS_INFO	NORMAL

检查索引键值：

```
SQL> select index_name,column_name
       2 from user_ind_columns where table_name='HS_INFO';
```

INDEX_NAME	COLUMN_NAME
HSIDX_INFO1	NUMORDER
HSIDX_INFO1	SYS_NC00024\$
HSIDX_INFO_SEARCHKEY	VC2INDEXWORDS
PK_HS_INFO	NUMINFOGUID

```
SQL> desc hs_info
```

Name	Null?	Type
NUMINFOGUID	NOT NULL	NUMBER(15)
NUMCATALOGGUID	NOT NULL	NUMBER(15)
INTTEXTTYPE	NOT NULL	NUMBER(38)
VC2TITLE	NOT NULL	VARCHAR2(60)
VC2AUTHOR		VARCHAR2(100)
.....		

检查发现在 numcatalogguid 字段上并没有索引，该字段具有很好的区分度，考虑在该字段创建索引以消除全表扫描。

```
SQL> create index hs_info_NUMCATALOGGUID on hs_info(NUMCATALOGGUID);
```

Index created.

SQL> set autotrace trace explain

SQL> select i.vc2title,i.numinfoguid

```

2   from   hs_info i where i.intenabedflag = 1
3   and i.intpublishstate = 1   and i.datpublishdate <=sysdate
4   and i.numcatalogguid = 3475
5   order by i.datpublishdate desc, i.numorder desc ;

```

Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=1 Bytes=106)
1    0      SORT (ORDER BY) (Cost=12 Card=1 Bytes=106)
2    1      TABLE ACCESS (BY INDEX ROWID) OF 'HS_INFO' (Cost=2 Card=1 Bytes=106)
3    2      INDEX (RANGE SCAN) OF 'HS_INFO_NUMCATALOGGUID' (NON-UNIQUE)
Cost=1 Card=1)

```

观察系统状况，原大量等待消失：

SQL> select sid,event,p1,p1text from v\$session_wait where event not like 'SQL%';

SID EVENT	P1 P1TEXT
1 pmon timer	300 duration
2 rdbms ipc message	300 timeout
3 rdbms ipc message	300 timeout
6 rdbms ipc message	180000 timeout
59 rdbms ipc message	6000 timeout
118 rdbms ipc message	6000 timeout
275 rdbms ipc message	30000 timeout
147 rdbms ipc message	6000 timeout
62 rdbms ipc message	6000 timeout
11 rdbms ipc message	30000 timeout
4 rdbms ipc message	300 timeout
305 db file sequential read	17 file#
356 db file sequential read	17 file#
19 db file scattered read	17 file#
5 smon timer	300 sleep time

```
15 rows selected.
```

至此，此问题得以解决。

通过以上案例，可以知道从等待事件进行追踪的诊断方法，这种方法在日常数据库诊断中很常用，在后面章节中还将进一步介绍。

8.3 Oracle 10g 的增强

虽然 V\$SESSION_WAIT 记录的信息如此重要，但是这些重要的信息是随 session 而消失的，如果希望获得数据库的历史状态及 session 的历史等待信息等数据，那是不可得的。

所以很多时候很难回答这样的问题：

- 这个系统昨天是什么样子的？
- 今天和昨天相比有什么不同？
- 1 个小时前的那次性能下滑是哪个用户引起的？
- 是哪些事件使我们今天用了更多的时间来等待？

你也可能一次又一次地听到 Oracle Support 这样问：

- 问题出现时系统是怎样的状况？
- 问题出现时系统有哪些等待？
- 你能否重现（Reproduce）问题以便我们判断？

很多这样的问题是极其使人恼火的，我们当然不希望问题重现（Reproduce）再次引起当机或业务损失，而那些问题看起来分明是不作为的责任推卸。可是事实是，失去了现场和当时的状态以及 session 的实时信息，也的确很难判断问题的所在。

从 Oracle 10g 开始，Oracle 开始改变这一切。所以赘述这么多，我只想更郑重地告诉大家，这一改变是多么的重要。

8.3.1 新增 V\$SESSION_WAIT_HISTORY 视图

为了更有效地保留 session 信息，Oracle 10g 新增加了一个 V\$SESSION_WAIT_HISTORY 视图，该视图用以记录活动 session 最近 10 次的等待事件。

```
SQL> select event,p1text,p1,p2text,p2,p3text,p3,wait_time
2 from v$session_wait_history where sid=120;
```

EVENT	P1TEXT	P1 P2TEXT	P2 P3TEXT	P3	WAIT_TIME
db file sequential read	file#	14 block#	97456 blocks	1	0
row cache lock	cache id	11 mode	0 request	3	49
row cache lock	cache id	11 mode	0 request	3	0
db file sequential read	file#	10 block#	260171 blocks	1	1

db file sequential read	file#	14 block#	570536 blocks	1	10
db file sequential read	file#	14 block#	6363 blocks	1	12
db file sequential read	file#	14 block#	35285 blocks	1	9
db file sequential read	file#	14 block#	40674 blocks	1	9
db file sequential read	file#	14 block#	69631 blocks	1	1
db file sequential read	file#	14 block#	82498 blocks	1	3
10 rows selected					

通过这个视图，用户可以将 V\$SESSION_WAIT 延伸，获取更多的相关信息辅助数据库问题诊断。这是 Oracle 迈出一小步。

8.3.2 ASH 新特性

如果说 V\$SESSION_WAIT_HISTORY 是一小步，那么 ASH 则是 Oracle 迈出根本变革的一大步。从 Oracle 10g 开始，Oracle 引入了 ASH 新特性，也就是活动 session 历史信息记录 (Active Session History, ASH)。

ASH 以 V\$SESSION 为基础，每秒钟采样一次，记录活动会话等待的事件。因为记录所有会话的活动是非常昂贵的，所以不活动的会话不会被采样，这一点从 ASH 的“A”上就可以看出。采样工作由 Oracle 10g 新引入的一个后台进程 MMNL 来完成。

是否启用 ASH 功能，受到一个内部隐含参数控制：

```
SQL> @GetHparDes.sql
Enter value for par: ash_en
old 6: AND x.ksppinm LIKE '%&par%'
new 6: AND x.ksppinm LIKE '%ash_en%'

NAME                                VALUE                                DESCRIB
-----
...
_ash_enable                         TRUE                                To enable or disable Active Session sampling and flushing
```

而采样时间同样由另一个内部隐含参数决定：

```
SQL> @GetHparDes.sql
Enter value for par: ash_sampling
old 6: AND x.ksppinm LIKE '%&par%'
new 6: AND x.ksppinm LIKE '%ash_sampling%'

NAME                                VALUE                                DESCRIB
-----
```

<code>_ash_sampling_interval</code>	1000	Time interval between two successive Active Session samples in millisecs
-------------------------------------	------	--

1000ms，正好是 1s 的时间。此处引用的脚本 `GetHparDes.sql` 内容如下：

```
SELECT x.ksppinm NAME, y.ksppstvl VALUE, x.ksppdesc describ
FROM SYS.x$ksppi x, SYS.x$ksppcv y
WHERE x.inst_id = USERENV ('Instance')
AND y.inst_id = USERENV ('Instance')
AND x.indx = y.indx
AND x.ksppinm LIKE '%&par%'
/
```

注 意

隐含参数通常具有特殊的作用，一般不建议用户查询或者修改，本章大量引用隐含参数的目的只有一个，那就是希望大家知道，所有在文档中见到的限制、约束、阈值、比率都是有来源的，只要足够细心，我们就能找出数据库的真相，不再靠记忆来学习。

很多人可能更关心性能，如此频繁的采样是否会极大地影响数据库的性能呢？采样的性能影响无疑是存在的，但是因为 Oracle 的采样工具可以直接访问 Oracle 10g 内部结构，所以是极其高效的，对于性能的影响也非常小，这也正是 Oracle 提供优化或诊断工具的优势所在。

ASH 信息被设计为在内存中滚动的，在需要的时候早期的信息是会被覆盖的。ASH 记录的信息可以通过 `V$ACTIVE_SESSION_HISTORY` 视图来访问，对于每个活动 session，每次采样会在这个视图中记录一行信息。这部分内存存在 SGA 中分配：

```
SQL> select * from v$sgastat where name like '%ASH%';
```

POOL	NAME	BYTES
shared pool	ASH buffers	6291456

注意，ASH buffers 的最小值为 1MB，最大值不超过 30MB，大小按照以下算法分配：

```
Max ( Min (cpu_count * 2MB, 5% * SHARED_POOL_SIZE, 30MB), 1MB)
```

在以上公式中，如果 `SHARED_POOL_SIZE` 未显示设置，则限制为 `2%*SGA_TARGET`。这一算法在 Oracle 10g 的不同版本中，可能有所不同。

根据这个算法，这里的采样系统分配的 ASH Buffers 为 6MB：

```
SQL> select name,value,display_value from v$parameter
2 where name in ('shared_pool_size','cpu_count');
```

NAME	VALUE	DISPLAY_VALUE
cpu_count	4	4

Parameter	Value	Limit
shared_pool_size	125829120	120M

另外一个生产系统中，这一内存分配为 8MB：

```
SQL> select * from v$version where rownum <2;
BANNER
-----
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod

SQL> show parameter cpu_count
NAME                                TYPE                                VALUE
-----
cpu_count                           integer                             4

SQL> show parameter sga_target
NAME                                TYPE                                VALUE
-----
sga_target                           big integer                         900M

SQL> show parameter shared_pool_size
NAME                                TYPE                                VALUE
-----
shared_pool_size                       big integer                         0

SQL> select * from v$sgastat where name like 'ASH%';
POOL          NAME                                BYTES
-----
shared pool    ASH buffers                                8388608
```

记录在 SGA 中的 ASH 信息，可以通过 V\$SESSION_WAIT_HISTORY 进行查询：

```
SQL> desc v$session_wait_history
Name          Type          Nullable Default Comments
-----
SID           NUMBER        Y
SEQ#          NUMBER        Y
EVENT#        NUMBER        Y
EVENT         VARCHAR2(64)  Y
P1TEXT        VARCHAR2(64)  Y
P1            NUMBER        Y
P2TEXT        VARCHAR2(64)  Y
P2            NUMBER        Y
P3TEXT        VARCHAR2(64)  Y
P3            NUMBER        Y
```

WAIT_TIME	NUMBER	Y
WAIT_COUNT	NUMBER	Y

可以通过 Oracle 提供的工具生成 ASH 的报告，报告可以以几分钟为跨度对数据库进行精确分析；也可以以数小时或数天为时间跨度，为数据库提供概要分析。

生成 ASH 报告主要可以通过以下两种方式。

1. 脚本方式

调用 \$ORACLE_HOME/rdbms/admin/ashrpt.sql 脚本，回答一系列问题之后，就可以生成一个 ASH 的报告，报告包括 Top 等待事件、Top SQL、Top SQL 命令类型、Top Sessions 等内容，摘录部分报告内容如下。

使用以下脚本：

```
SQL> @?/rdbms/admin/ashrpt.sql

Current Instance
~~~~~

      DB Id      DB Name      Inst Num Instance
-----
3965153484 DANALY              1 danaly
.....
ASH Samples in this Workload Repository schema
~~~~~
```

数据库可用的采样数据：

```
Oldest ASH sample available: 31-Mar-06 08:31:52 [ 4325 mins in the past]
Latest ASH sample available: 04-Sep-06 22:39:11 [ ##### mins in the past]
.....
```

用户定义概要：

```
Summary of All User Input
-----
Format          : TEXT
DB Id           : 3965153484
Inst num        : 1
Begin time      : 02-Apr-06 08:37:42
End time        : 03-Apr-06 08:37:59
Slot width      : Default
Report targets  : 0
```

Report name : ashprt_1_0403_0837.txt

生成的报告:

ASH Report For DANALY/danaly

DB Name	DB Id	Instance	Inst Num	Release	RAC	Host

DANALY	3965153484	danaly	1	10.2.0.1.0	NO	danaly.hurrr

CPU's	SGA Size	Buffer Cache	Shared Pool	ASH Buffer Size

4	900M (100%)	772M (85.8%)	210M (23.3%)	8.0M (0.9%)

Analysis Begin Time: 02-Apr-06 08:37:42

Analysis End Time: 03-Apr-06 08:37:59

Elapsed Time: 1,440.3 (mins)

Sample Count: 2,946

Average Active Sessions: 0.03

Avg. Active Session per CPU: 0.01

Report Target: None specified

Top User Events DB/Inst: DANALY/danaly (Apr 02 08:37 to 08:37)

Avg Active

Event	Event Class	% Activity	Sessions

CPU + Wait for CPU	CPU	22.84	0.01
log file sync	Commit	18.23	0.01

Top Background Events DB/Inst: DANALY/danaly (Apr 02 08:37 to 08:37)

Avg Active

Event	Event Class	% Activity	Sessions

log file parallel write	System I/O	21.83	0.01
control file parallel write	System I/O	15.44	0.01
db file parallel write	System I/O	15.41	0.01
CPU + Wait for CPU	CPU	5.26	0.00

.....			
Top SQL Command Types		DB/Inst: DANALY/danaly (Apr 02 08:37 to 08:37)	
.....			
	Distinct	Avg Active	
SQL Command Type	SQLIDs % Activity	Sessions	

INSERT	8	11.30	0.00
SELECT	54	6.79	0.00
PL/SQL EXECUTE	21	2.17	0.00
UPDATE	10	2.07	0.00

Top SQL Statements		DB/Inst: DANALY/danaly (Apr 02 08:37 to 08:37)	

SQL ID	Planhash % Activity	Event % Event	

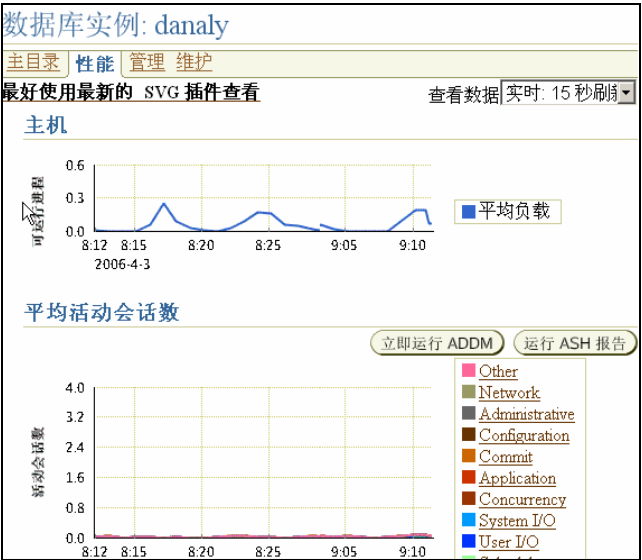
74y62ap82k1xk	2315018254	7.74 CPU + Wait for CPU	7.74
INSERT INTO	MGMT_CURRENT_METRICS	(TARGET_GUID,	KEY_VALUE,
COLLECTION_TIMESTAMP,			
METRIC_GUID, VALUE, STRING_VALUE) VALUES (:B1 , :B2 , :B3 , :B4 , :B5 , :B6)			
.....			

Top Sessions		DB/Inst: DANALY/danaly (Apr 02 08:37 to 08:37)	
.....			
End of Report			
Report written to ashprt_1_0403_0837.txt			

2. OEM 图形方式

使用 OEM，可以在性能页中单击“运行 ASH 报告”按钮生成 ASH 报告，如图 8-1 所示。

OEM 生成的 ASH 报告非常清晰和直观，ASH 的概况信息如图 8-2 所示，等待事件信息如图 8-3 所示，等待参数信息如图 8-4 所示，Top SQL 信息如图 8-5 所示。只要试用一下就可以感受到 ASH 的强大功能。



DB Name	DB Id	Instance	Inst num	Release	RAC	Host
DANALY	3965153484	danaly	1	10.2.0.1.0	NO	danaly.hurray.com.cn

CPUs	SGA Size	Buffer Cache	Shared Pool	ASH Buffer Size
4	900M (100%)	768M (85.3%)	214M (23.7%)	8.0M (0.9%)

	Sample Time	Data Source
Analysis Begin Time:	31-Mar-06 21:01:20	V\$ACTIVE_SESSION_HISTORY
Analysis End Time:	31-Mar-06 22:06:20	V\$ACTIVE_SESSION_HISTORY
Elapsed Time:	65.0 (mins)	
Sample Count:	2,421	
Average Active Sessions:	0.62	
Avg. Active Session per CPU:	0.16	
Report Target:	None specified	

图 8-2 ASH 的概况信息

Top User Events

Event	Event Class	% Activity	Avg Active Sessions
db file scattered read	User I/O	27.05	0.17
SQL*Net more data from dblink	Network	9.25	0.06
CPU + Wait for CPU	CPU	8.55	0.05
db file sequential read	User I/O	3.97	0.02
direct path read temp	User I/O	3.39	0.02

[Back to Top Events](#)

[Back to Top](#)

Top Background Events

Event	Event Class	% Activity	Avg Active Sessions
db file parallel write	System I/O	24.20	0.15
log file parallel write	System I/O	14.21	0.09
control file parallel write	System I/O	1.57	0.01
CPU + Wait for CPU	CPU	1.03	0.01

图 8-3 等待事件信息

Top Event P1/P2/P3 Values						
Event	% Event	P1 Value, P2 Value, P3 Value	% Activity	Parameter 1	Parameter 2	Parameter 3
db file scattered read	27.05	"10","926205","16"	0.08	file#	block#	blocks
db file parallel write	24.20	"1","0","2147483647"	13.96	requests	interrupt	timeout
		"2","0","2147483647"	3.39			
		"3","0","2147483647"	1.53			
log file parallel write	14.21	"2","26","2"	1.40	files	blocks	requests
SQL*Net more data from dblink	9.25	"675562835","1","0"	2.81	driver id	#bytes	NOT DEFINED
		"675562835","11","0"	2.73			
		"675562835","5","0"	2.11			
db file sequential read	4.05	"3","80812","1"	0.04	file#	block#	blocks

图 8-4 等待参数信息

Top SQL Statements					
SQL ID	Planhash	% Activity	Event	% Event	SQL Text
gz9x4u45c91mb	1976783940	22.02	SQL*Net more data from dblink	9.00	DELETE FROM CM_TB_MMS_DATA_NEW...
			db file scattered read	6.90	
			direct path read temp	3.22	
5p1nu5zz3d5vq	2780303235	11.90	db file scattered read	9.38	delete from CM_TB_MMS_DATA_OLD...
			db file sequential read	1.28	
			CPU + Wait for CPU	1.24	
3zmwscncd42zg	2489428218	4.63	db file scattered read	4.13	select count(*) from CM_TB_MMS...
gs6rw17g1zbsn	1954154723	3.30	db file scattered read	2.48	delete from CM_TB_MMS_DATA_NEW...
7aksy1t0qms5s	4060839695	2.44	db file scattered read	1.32	DELETE FROM CM_TB_MMS_DATA_NEW...
			CPU + Wait for CPU	1.07	

图 8-5 Top SQL 信息

8.3.3 自动负载信息库 AWR 的引入

内存中记录的 ASH 信息始终是有限的,为了保存历史数据,这些信息最终需要写入磁盘。这些历史信息的存储,引出了 Oracle 10g 的另外一个新特性:自动负载信息库 (Automatic Workload Repository, AWR)。

AWR 收集关于该特定数据库的操作统计信息和其他统计信息,Oracle 以固定的时间间隔(默认为每小时一次)为其所有重要统计信息和负载信息执行一次快照,并将这些快照存储在 AWR 中。这些信息在 AWR 中保留给定的时间(默认为一周),然后被清除。执行快照的频率及其保持时间都可以自定义,以满足不同环境的独特需要。

AWR 的采样工作由后台进程 MMON 每 60 分钟执行一次,ASH 信息同样会被采样写出到 AWR 负载库。虽然 ASH Buffers 被设计为保留 1 小时的信息,但是很多时候这个内存是不够的,当 ASH Buffers 写满之后,另外一个后台进程 MMNL 将会主动将 ASH 信息写出。由于数据量巨大,把所有的 ASH 数据写到磁盘上是不可接受的。一般是在写到磁盘的时候过滤这个数据,写出的数据占采样数据的 10%,写出时通过 direct-path insert 完成,尽量减少日志生成,从而最小化数据库性能的影响。

通过图 8-6 可以直观地理解 ASH 及 AWR 的关系。

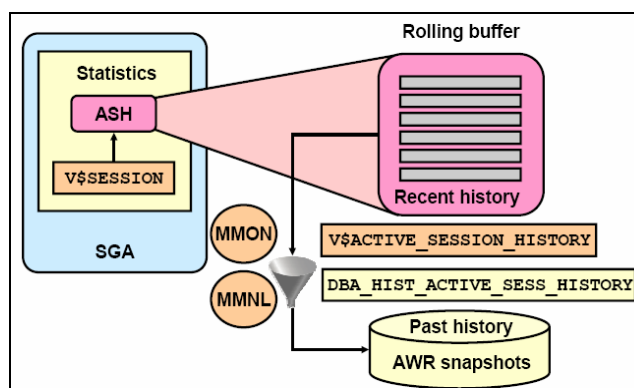


图 8-6 ASH 及 AWR 的关系

AWR 的行为受到数据库的另外一个重要初始化参数 `STATISTICS_LEVEL` 影响，该参数有 3 个可选值。

- **BASIC**: 设置为 **BASIC** 时，AWR 的统计信息收集和所有自我调整的特性都被关闭。
- **TYPICAL**: 设置为 **TYPICAL** 时，数据库收集部分统计信息，这些信息为典型的数据库监控需要，是数据库的缺省设置。

- **ALL**: 所有可能的统计信息都被收集。

ASH 信息的写出比例受一个隐含参数控制：

```
SQL> @GetHparDes.sql
Enter value for par: filter_ratio
old 6: AND x.ksppinm LIKE '%&par%'
new 6: AND x.ksppinm LIKE '%filter_ratio%'
```

NAME	VALUE	DESCRIB
_____	_____	_____
_ash_disk_filter_ratio	10	Ratio of the number of in-memory samples to the number of samples actually written to disk

写出到 AWR 负载库的 ASH 信息记录在 AWR 的基础表 `WRH$ACTIVE_SESSION_HIST` 中，`WRH$ACTIVE_SESSION_HIST` 是一个分区表，Oracle 会自动进行数据清理。`WRH$ACTIVE_SESSION_HIST` 记录的这些历史信息可以通过 `DBA_HIST_ACTIVE_SESS_HISTORY` 视图进行聚合查询，通过简化后的图 8-7 来看一下 Oracle 以 session 为起点的一系列用以追踪和诊断的数据库对象。

简单总结一下：

- `V$SESSION` 代表数据库活动的开始，视为源起；
- `V$SESSION_WAIT` 视图用以实时记录活动 session 的等待情况，是当前信息；
- `V$SESSION_WAIT_HISTORY` 是对 `V$SESSION_WAIT` 的简单增强，记录活动 session 的最近 10 次等待；
- `V$ACTIVE_SESSION_HISTORY` 是 ASH 的核心，用以记录活动 session 的历史等待信息，每秒采样一次，这部分内容记录在内存中，期望值是记录 1 个小时的内容；

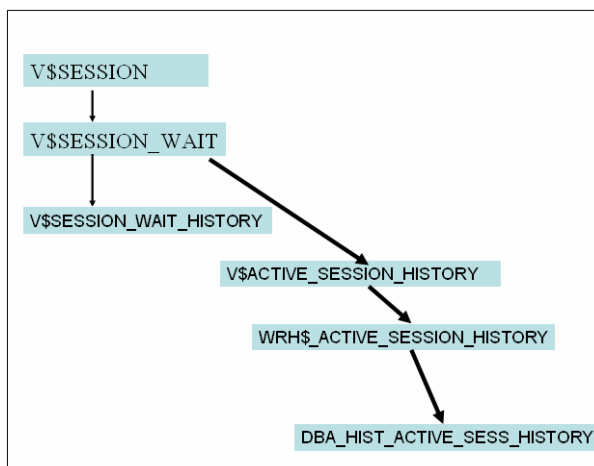


图 8-7 一系列用以追踪和诊断的数据库对象

■ **WRH\$_ACTIVE_SESSION_HISTORY** 是 **V\$ACTIVE_SESSION_HISTORY** 在 AWR 的存储地，**V\$ACTIVE_SESSION_HISTORY** 中记录的信息会被定期（每小时一次）的刷新到负载库中，并缺省保留一个星期用于分析；

■ **DBA_HIST_ACTIVE_SESS_HISTORY** 视图是 **WRH\$_ACTIVE_SESSION_HISTORY** 视图和其他几个视图的联合展现，通常通过这个视图进行历史数据的访问。

可以看到，关于 session 信息的记录，Oracle 从不同的粒度进行了增强，目的只有一个，那就是全面真实地记录、监控和反映数据库的运行状况。

AWR 记录的信息还远不止于此，通过系统的自动采样，AWR 可以收集数据库运行的各方面统计信息及等待等重要数据，并提供给数据库诊断分析使用。当然 AWR 的信息需要独立存储，在 Oracle 10g 中，新增的 SYSAUX 表空间是这类信息的存储地：

```
SQL> select OCCUPANT_NAME,OCCUPANT_DESC,SCHEMA_NAME,SPACE_USAGE_KBYTES/1024
'MB'
2  from V$SYSAUX_OCCUPANTS WHERE OCCUPANT_NAME LIKE '%AWR%'
3  /
```

OCCUPANT_NAME	OCCUPANT_DESC	SCHEMA_NAME	MB
SM/AWR	Server Manageability - Automatic Workload Repository	SYS	250.875

在 Oracle 10g 之前的版本中，类似的功能是由 Statspack 实现，但是 Statspack 需要由用户自行安装调度，并且其收集的信息十分有限。一直提到的 session 历史信息，Statspack 是无法提供的。AWR 大大强化了这部分信息，由于 AWR 收集的信息十分完备，所以经常被称为‘数据库的数据仓库’。

8.3.4 自动数据库诊断监控 ADDM 的引入

有了 AWR 这个“数据仓库”之后，Oracle 自然可以在此基础之上实现更高级别的智能

应用，更大程度地发挥 AWR 的作用，这就是 Oracle 10g 引入的另外一个功能自动数据库诊断监控程序（Automatic Database Diagnostic Monitor, ADDM），通过 ADDM，Oracle 试图使数据库的维护、管理和优化工作变得更加自动和简单。

ADDM 可以定期检查数据库的状态，根据内建的专家系统，自动确定潜在的数据库性能瓶颈，并提供调整措施和建议。由于这一切都是内建在 Oracle 数据库系统之内的，其执行效率很高，几乎不影响数据库的总体性能。

新版 Oracle Enterprise Manager 可以以一种方便直观的形式提供 ADDM 的结果和建议，并引导管理员逐步实施 ADDM 的建议，快速解决性能问题。

通过图 8-8 可以直观地看出 AWR 及 ADDM 的关系。

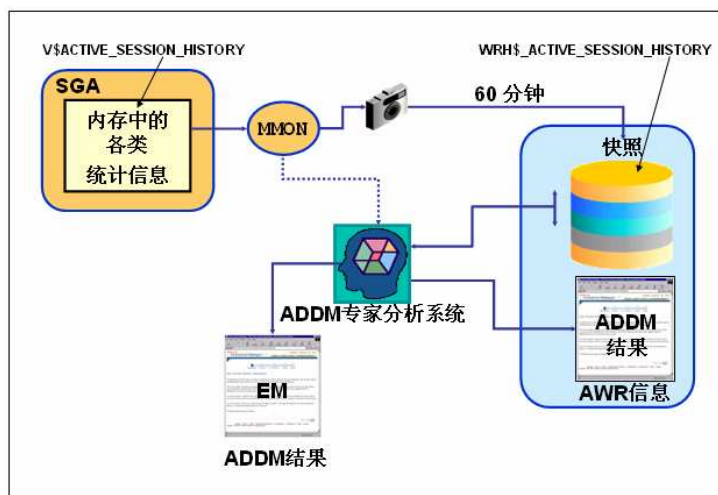


图 8-8 AWR 及 ADDM 的关系

对于 ADDM，这里不再做过多的介绍。

8.4 顶级等待事件

上文还提到另外一个重要视图 V\$SYSTEM_EVENT，该视图记录的是数据库自启动以来等待事件的汇总。通过查询该视图，可以快速获得数据库等待事件的总体概况，了解数据库运行的基本状态：

```
SQL> SELECT *
2    FROM (SELECT    event, time_waited
3              FROM v$system_event
4             ORDER BY time_waited DESC)
5    WHERE ROWNUM < 10;
```

EVENT

TIME_WAITED

SQL*Net message from client	9.2256E+10
rdbms ipc message	1.2383E+10
pmon timer	1834857492
smon timer	1771582338
jobq slave wait	414242315
db file scattered read	54344796
enqueue	29142826
latch free	18022667
db file sequential read	11925101
log file sync	9500670

10 rows selected.

以上是一个产品环境中的 Top10 等待事件，由于 Top5 等待都是空闲等待，所以不必过多关注，但是接下来的 5 个等待事件都是常见的重要等待事件，如果能够对其进行针对性优化，数据库性能将会得到大幅度提升。

在 Oracle 的 Statspack Report 中，有一部分信息为 Top 5 Wait Events（在 Oracle 9i 中更改为 Top 5 Time Events），这部分信息就是来自 V\$SYSTEM_EVENT 视图的采样。以下是一个 Statspack 的诊断报告：

DB Name	DB Id	Instance	Inst Num	Release	OPS	Host
K2	1999167370	k2	1	8.1.5.0.0	NO	k2

这是一个 8.1.5 的数据库系统，通过脚本增强，可以在 8.1.5 的数据库上使用 Statspack 来进行数据库诊断。

Start Id	End Id	Start Time	End Time	Snap Length (Minutes)
170	176	25-Feb-03 10:00:11	25-Feb-03 15:00:05	299.90

Cache Sizes

~~~~~

|                   |           |
|-------------------|-----------|
| db_block_buffers: | 64000     |
| db_block_size:    | 8192      |
| log_buffer:       | 8388608   |
| shared_pool_size: | 157286400 |

.....

| Top 5 Wait Events       |            |           |         |
|-------------------------|------------|-----------|---------|
| ~~~~~                   |            |           |         |
|                         |            | Wait      | % Total |
| Event                   | Waits      | Time (cs) | Wt Time |
| -----                   |            |           |         |
| db file scattered read  | 16,842,920 | 3,490,719 | 43.32   |
| latch free              | 844,272    | 3,270,073 | 40.58   |
| buffer busy waits       | 114,421    | 933,136   | 11.58   |
| db file sequential read | 2,067,910  | 117,750   | 1.46    |
| enqueue                 | 464        | 110,840   | 1.38    |
| -----                   |            |           |         |

这里的 Top 5 Wait Events 是诊断的重要依据。

这是一个典型的性能低下的系统，几个重要的等待事件都在 Top 5 中出现，其中，前 3 个等待极为显著，需要进行相应的调整。

在 5 小时的采样间隔内，其中 db file scattered read 累计等待时间约 10 小时，已经成为影响系统性能的主要原因。了解了这些以后就可以进一步查看 Statspack Report 中相关的 SQL 部分，看是否存在可疑的 SQL 语句。

| SQL ordered by Gets for DB: K2 Instance: k2 Snaps: 170 - 176         |          |           |       |            |
|----------------------------------------------------------------------|----------|-----------|-------|------------|
|                                                                      | Gets     | % of      |       |            |
| Buffer Gets                                                          | Executes | per Exec  | Total | Hash Value |
| -----                                                                |          |           |       |            |
| SQL statement                                                        |          |           |       |            |
| -----                                                                |          |           |       |            |
| 6,480,163                                                            | 12       | 540,013.6 | 2.4   | 3791855498 |
| SELECT "PROCESS_REQ"."WORK_ID", "PROCESS_REQ"."STOCK_NO", "PROCESS_R |          |           |       |            |
| 3,784,566                                                            | 16       | 236,535.4 | 1.4   | 2932917818 |
| SELECT * FROM FIND_LATER_WO ORDER BY NOTE,ORDER_NO                   |          |           |       |            |
| 1,200,976                                                            | 3        | 400,325.3 | .4    | 4122791109 |
| SELECT "ITEM_STOCK"."ITEM_NO", "ITEM"."NOTE", "ITEM"                 |          |           |       |            |
| 923,944                                                              | 9        | 102,660.4 | .3    | 2200071737 |
| SELECT "ITEM_STOCK"."ITEM_NO", "ITEM_STOCK"."STOCK_NO",              |          |           |       |            |
| 921,301                                                              | 3        | 307,100.3 | .3    | 2218843294 |
| SELECT "ITEM_STOCK"."ITEM_NO", "ITEM"."NOTE", "ITEM"                 |          |           |       |            |

```

          911,285          3    303,761.7    .3    1769130587
SELECT  "LISTS"."ITEM_NO" ,          "LISTS"."SUB_ITEM" ,          "LISTS"

          831,439          2    415,719.5    .3    1349577999
SELECT  "GROUP_OPER"."ITEM_NO" ,          "GROUP_OPER"."PROCESS_ID" ,

          802,918          1    802,918.0    .3    3613809507
SELECT  "LISTS"."ITEM_NO" ,          "LISTS"."SUB_ITEM" ,          "ITEM".

          800,548          2    400,274.0    .3    2643788247
SELECT "ITEM_STOCK"."ITEM_NO",          "ITEM"."NOTE",          "ITEM"

          666,085          2    333,042.5    .2    3391363608
SELECT "ITEM_STOCK"."ITEM_NO",          "ITEM_STOCK"."STOCK_NO",
.....

```

注意到以上很多查询导致的 **Buffer Gets** 都非常庞大,所以非常有理由怀疑索引存在问题,甚至缺少必要的索引。以上记录的是 SQL 的片段,通过 Hash Value 值再结合 V\$SQL\_TEXT,可以获得完整的 SQL 语句(可以参考上文的案例)。

在这次诊断中,紧接着去查询的是 v\$session\_longops 视图,一个分组查询的结果如下:

| TARGET                     | COUNT(*) |
|----------------------------|----------|
| SA.PPBT_GRAPHOBJTABLE      | 418      |
| SA.PPBT_PPBT OBJRELATTABLE | 53       |

发现这些问题 SQL 的全表扫描(结合 v\$session\_longops 视图中的 OPNAME)主要集中在 PPBT\_GRAPHOBJTABLE 和 PPBT\_PPBT OBJRELATTABLE 两张数据表上。进一步研究发现这两个数据表上没有任何索引,并且有相当的数据量:

```

SQL> select count(*) from SA.PPBT_PPBT OBJRELATTABLE;

COUNT(*)
-----
1209017

SQL> select count(*) from SA.PPBT_GRAPHOBJTABLE;

COUNT(*)
-----
2445

```

在创建了合适的索引后，系统性能得到了大幅提高！

## 8.5 重要等待事件

在了解了等待事件的作用和变迁之后，让我们来了解一下重要的等待事件。

### 8.5.1 db file sequential read（数据文件顺序读取）

**db file sequential read** 是个非常常见的 I/O 相关的等待事件，通常显示与单个数据块相关的读取操作，在大多数的情况下，读取一个索引块或者通过索引读取一个数据块时，都会记录这个等待。

这个等待事件有 3 个参数 P1、P2、P3，其中 P1 代表 Oracle 要读取的文件的绝对文件号，P2 代表 Oracle 从这个文件中开始读取的起始数据块块号，P3 代表读取的 BLOCK 数量，通常这个值为 1，表明是单个 BLOCK 被读取。

```
SQL> select name,parameter1,parameter2,parameter3
       2  from v$event_name where name='db file sequential read';
```

| NAME                    | PARAMETER1 | PARAMETER2 | PARAMETER3 |
|-------------------------|------------|------------|------------|
| db file sequential read | file#      | block#     | blocks     |

在 Oracle 10g 中，这个等待事件被归入 User I/O 一类：

```
SQL> select name,wait_class
       2  from v$event_name where name='db file sequential read';
```

| NAME                    | WAIT_CLASS |
|-------------------------|------------|
| db file sequential read | User I/O   |

如果这个等待事件比较显著，可能表示在多表连接中，表的连接顺序存在问题，可能没有正确的使用驱动表；或者可能索引的使用存在问题，并非索引总是最好的选择。

在大多数情况下，通过索引可以更为快速地获取记录，所以对于一个编码规范、调整良好的数据库，这个等待事件很大通常是正常的。

但是在很多情况下，使用索引并不是最佳的选择，比如读取较大表中大量的数据，全表扫描可能会明显快于索引扫描，所以在开发中就应该注意，对于这样的查询应该进行避免使用索引扫描。

从 Oracle 9iR2 开始，Oracle 引入了段级统计信息收集的新特性，可以通过查询 V\$SEGMENT\_STATISTICS 视图，找出物理读取显著的索引段或者是表段，研究其数据结构，看能否通过重建或者重新规划分区、存储参数等手段降低其 I/O 访问。

Oracle 9iR2 中，收集的统计信息共有 11 类：

```
SQL> select * from v$segstat_name;
```

| STATISTIC# NAME                      | SAMPLED |
|--------------------------------------|---------|
| -----                                |         |
| 0 logical reads                      | YES     |
| 1 buffer busy waits                  | NO      |
| 2 db block changes                   | YES     |
| 3 physical reads                     | NO      |
| 4 physical writes                    | NO      |
| 5 physical reads direct              | NO      |
| 6 physical writes direct             | NO      |
| 8 global cache cr blocks served      | NO      |
| 9 global cache current blocks served | NO      |
| 10 ITL waits                         | NO      |
| 11 row lock waits                    | NO      |

11 rows selected.

在 Oracle 10gR2 中，这类统计信息增加为 15 个：

```
SQL> select * from v$segstat_name;
```

| STATISTIC# NAME               | SAM |
|-------------------------------|-----|
| -----                         |     |
| 0 logical reads               | YES |
| 1 buffer busy waits           | NO  |
| 2 gc buffer busy              | NO  |
| 3 db block changes            | YES |
| 4 physical reads              | NO  |
| 5 physical writes             | NO  |
| 6 physical reads direct       | NO  |
| 7 physical writes direct      | NO  |
| 9 gc cr blocks received       | NO  |
| 10 gc current blocks received | NO  |
| 11 ITL waits                  | NO  |
| 12 row lock waits             | NO  |
| 14 space used                 | NO  |

|                    |    |
|--------------------|----|
| 15 space allocated | NO |
| 17 segment scans   | NO |

15 rows selected.

对于 CBO 模式下的数据库，应当及时收集统计信息，使 SQL 可以选择正确的执行计划，避免因统计信息陈旧而导致的执行错误等。

### 8.5.2 db file scattered read（数据文件离散读取）

我们已经多次见到 db file scattered read 等待事件，在生产环境之中，这个等待事件可能更为常见。我们就此把讨论进一步展开。

从 V\$EVENT\_NAME 视图可以看到，该事件有 3 个参数，分别代表文件号、起始数据块号、数据块的数量：

```
SQL> select * from v$version where rownum <2;
```

BANNER

-----

Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production

```
SQL> select * from v$event_name where name='db file scattered read';
```

| EVENT# | NAME                   | PARAMETER1 | PARAMETER2 | PARAMETER3 |
|--------|------------------------|------------|------------|------------|
| 206    | db file scattered read | file#      | block#     | blocks     |

起始数据块号加上数据块的数量，这意味着 Oracle Session 正在等待多块连续读操作的完成。这个操作可能与全表扫描（Full table Scan）或者快速全索引扫描（Index Fast Full Scan）的连续读取相关。根据经验，通常大量的 db file scattered read 等待可能意味着应用问题或者索引缺失。

在实际环境的诊断过程中，可以通过 V\$SESSION\_WAIT 视图发现 session 的等待，再结合其他视图找到存在问题的 SQL 等根本原因，从而从根本上解决问题。此类诊断案例，请参考 8.2 节中的案例。

当这个等待事件比较显著时，也可以结合 V\$SESSION\_LONGOPS 动态性能视图来进行诊断，该视图中记录了长时间（运行时间超过 6 秒的）运行的事务，可能很多是全表扫描操作（不管怎样，这部分信息都是值得注意的），上一个案例就是通过 V\$SESSION\_LONGOPS 快速发现了问题所在。

从 Oracle 9i 开始，Oracle 新增加了一个视图 V\$SQL\_PLAN 用于记录当前系统 Library Cache 中 SQL 语句的执行计划，可以通过这个视图找到存在问题的 SQL 语句，以下是我在一个生产系统中查询得到的结果：

```
$ sqlplus "/ as sysdba"
```

SQL\*Plus: Release 9.2.0.4.0 - Production on Fri Mar 31 15:59:40 2006

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Connected to:

Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production

With the Partitioning option

JServer Release 9.2.0.4.0 - Production

SQL> @getplan

Enter value for waitevent: free buffer waits

old 15: AND b.event = '&waitevent')

new 15: AND b.event = 'free buffer waits')

| HASH_VALUE | CHILD_NUMBER | OPERATION               | OBJECT          | COST  | KBYTES  |
|------------|--------------|-------------------------|-----------------|-------|---------|
| 2838180055 | 0            | INSERT STATEMENT CHOOSE | Cost=41733      |       | 41733   |
| 2838180055 | 0            | TABLE ACCESS FULL       | I_CM_POWER_TEMP | 41733 | 1356468 |

进而可以通过 V\$SQL\_TEXT 视图获得这个问题 session 正在执行的 SQL 语句:

SQL> select sid,event from v\$session\_wait;

SID EVENT

1 pmon timer

4 rdbms ipc message

7 rdbms ipc message

5 rdbms ipc message

8 rdbms ipc message

21 free buffer waits

49 free buffer waits

2 db file parallel write

3 db file parallel write

6 smon timer

.....

16 rows selected.

SQL>@ GetSqlBySid

Enter value for sid: 49

old 5: where b.sid='&sid'

new 5: where b.sid='49'

SQL\_TEXT

```
insert into i_cm_power_new(PNAME,YYS,SPHM,SJH,SENTTIME,NOTES,PLACE,RMK) select PNAME,YYS,SPHM,SJH,SENTTIME,NOTES,PLACE,RMK FROM i_cm_power_temp
```

通过 V\$SQL\_PLAN 视图，可以获得大量有用的信息，比如获得全表扫描的对象：

```
SQL> select distinct object_name,object_owner from v$sql_plan p
2  where p.operation='TABLE ACCESS' and p.options='FULL'
3  and object_owner = 'MKT'
4  /
```

| OBJECT_NAME       | OBJECT_OWNER |
|-------------------|--------------|
| HD_TEMP           | MKT          |
| I_CM_BILL         | MKT          |
| I_CM_IVR_BUTTON   | MKT          |
| .....             |              |
| TOOLS_HD          | MKT          |
| TOOLS_HD_NEW      | MKT          |
| TOOLS_HD_NEW_BAK  | MKT          |
| TOOLS_IVRBLIST    | MKT          |
| TOOLS_USER_CANCEL | MKT          |

29 rows selected

或者获得全索引扫描对象：

```
SQL> select distinct object_name,object_owner from v$sql_plan p
2  where p.operation='INDEX'
3  and p.options='FULL SCAN' ;
```

| OBJECT_NAME        | OBJECT_OWNER |
|--------------------|--------------|
| FK_ITEM_LEVEL_CODE | AVATAR       |

|                      |        |
|----------------------|--------|
| FK_ITEM_SELLCNT_CODE | AVATAR |
| FK_MYZZIM_CRTDATE    | AVATAR |
| I_SYSAUTH1           | SYS    |
| SYS_C008211          | WLLM   |

进而可以通过 V\$SQL\_PLAN 和 V\$SQLTEXT 联合，获得这些查询的 SQL 语句。查找全表扫描的 SQL 语句可以参考以下语句：

```
SELECT    sql_text
          FROM v$sqltext t, v$sql_plan p
          WHERE t.hash_value = p.hash_value
              AND p.operation = 'TABLE ACCESS'
              AND p.options = 'FULL'
          ORDER BY p.hash_value, t.piece;
```

查找 Fast Full Index 扫描的 SQL 语句可以参考以下语句：

```
SELECT    sql_text
          FROM v$sqltext t, v$sql_plan p
          WHERE t.hash_value = p.hash_value
              AND p.operation = 'INDEX'
              AND p.options = 'FULL SCAN'
          ORDER BY p.hash_value, t.piece;
```

这些信息对于发现数据库问题，优化数据库性能具有极强的指导意义。

本例中用到的 SQL 代码（脚本名称 getplan.sql）如下：

```
SET linesize 120
COL operation      format a55
COL cost           format 99999
COL kbytes         format 999999
COL object         format a25
SELECT    hash_value, child_number,
          LPAD (' ', 2 * DEPTH)
          || operation
          || ' '
          || options
          || DECODE (ID,
                    0, SUBSTR (optimizer, 1, 6) || ' Cost=' || TO_CHAR (COST)
                    ) operation,
          object_name OBJECT, COST, ROUND (BYTES / 1024) kbytes
          FROM v$sql_plan
```

```
WHERE hash_value IN (
    SELECT a.sql_hash_value
    FROM v$session a, v$session_wait b
    WHERE a.SID = b.SID
    AND b.event = '&waitevent')
ORDER BY hash_value, child_number, ID;
```

在 Oracle 10g 中, Oracle 对等待事件进行了分类, db file scattered read 事件被归入 User I/O 一类:

|                        |        |                                      |               |               |               |                                        |
|------------------------|--------|--------------------------------------|---------------|---------------|---------------|----------------------------------------|
| SQL>                   | select | name,PARAMETER1                      | p1,PARAMETER2 | p2,PARAMETER3 | p3,PARAMETER4 | WAIT_CLASS_ID, WAIT_CLASS#, WAIT_CLASS |
|                        | 2      | from v\$event_name                   |               |               |               |                                        |
|                        | 3      | where name='db file scattered read'; |               |               |               |                                        |
| NAME                   | P1     | P2                                   | P3            | WAIT_CLASS_ID | WAIT_CLASS#   | WAIT_CLASS                             |
| -----                  |        |                                      |               |               |               |                                        |
| db file scattered read | file#  | block#                               | blocks        | 1740759767    | 8             | User I/O                               |

完成对等待事件的分类之后, Oracle 10g 的 ADDM 可以很容易地通过故障树分析定位到问题所在, 帮助用户快速发现数据库的瓶颈及瓶颈的根源, 这就是 Oracle 的 ADDM 专家系统的设计思想。

通过图 8-9 可以直观清晰地看到这个等待模型和 ADDM 结合实现的 Oracle 专家诊断系统。

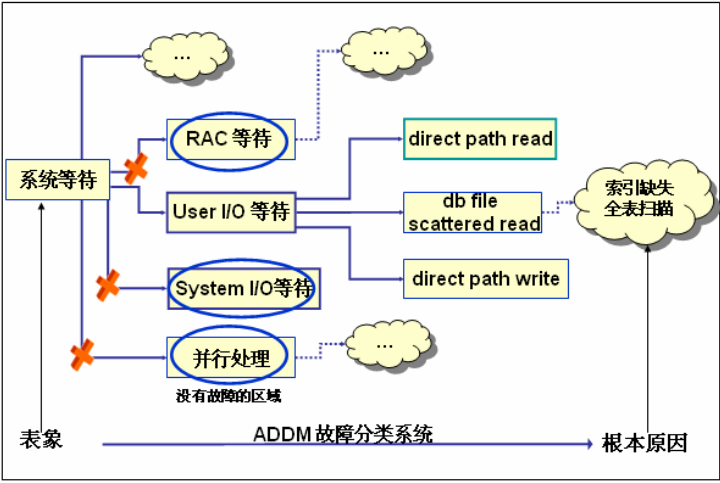


图 8-9 Oracle 专家诊断系统

8.5.3 direct path read/write (直接路径读/写)

直接路径读 (Direct Path Read) 通常发生在 Oracle 直接读数据到 PGA 时, 这个读取不需

要经过 SGA。直接路径读等待事件的 3 个参数分别是：file#（指绝对文件号）、first block#和 block 数量。在 Oracle 10g 中，这个等待事件被归于 User I/O 一类。

这类读取通常在以下情况被使用：

- 磁盘排序 IO 操作；
- 并行查询从属进程；
- 预读操作。

最为常见的是第一种情况。在 DSS 系统中，存在大量的 Direct path Read 是很正常的，但是在 OLTP 系统中，通常显著的直接路径读（Direct Path Read）都意味着系统应用存在问题，从而导致大量的磁盘排序读取操作。

直接路径写（Direct Path Write）通常发生在 Oracle 直接从 PGA 写数据到数据文件或临时文件，这个写操作可以绕过 SGA。直接路径写等待事件的 3 个参数分别是：file#（指绝对文件号）、first block#和 block 数量。在 Oracle 10g 中，这个等待事件同 Direct Path Read 一样被归于 User I/O 一类。

这类写入操作通常在以下情况被使用：

- 直接路径加载；
- 并行 DML 操作；
- 磁盘排序；
- 对未缓存的“LOB”段的写入，随后会记录为 direct path write (lob) 等待。

最为常见的直接路径写，多数因为磁盘排序导致。对于这一写入等待，应该找到 I/O 操作最为频繁的数据文件（如果有过多的排序操作，很有可能就是临时文件），分散负载，加快其写入操作。

如果系统存在过多的磁盘排序，会导致临时表空间操作频繁，对于这种情况，可以考虑为不同用户分配不同的临时表空间，使用多个临时文件，写入不同磁盘或者裸设备，从而降低竞争，提高性能；对于 Oracle 8i 的数据库，应该考虑使用本地管理（Local）的临时表空间，而不是字典（DICTIONARY）管理。

从 DBA\_TABLESPACES 视图可以获得这部分信息：

```
SQL> select tablespace_name,EXTENT_MANAGEMENT
2 from dba_tablespaces;
```

| TABSPACE_NAME | EXTENT_MAN |
|---------------|------------|
| SYSTEM        | DICTIONARY |
| RBS           | DICTIONARY |
| TEMP          | DICTIONARY |
| USERS         | LOCAL      |

4 rows selected.

可以看一个 Statspack 报告的典型例子：

| DB Name | DB Id | Instance | Inst Num | Release | OPS | Host |
|---------|-------|----------|----------|---------|-----|------|
|---------|-------|----------|----------|---------|-----|------|

```

-----
DB          294605295          db          1  8.1.5.0.0  NO          IBM

                                         Snap Length
Start Id    End Id      Start Time          End Time          (Minutes)
-----
          65          66  08-11 月-03 16:32:42  08-11 月-03 16:54:00          21.30

```

这是一个 20 分钟的采样报告，可以看到 **direct path read/write** 的等待都很显著：

```

Top 5 Wait Events
~~~~~
 Wait % Total
Event Waits Time (cs) Wt Time

direct path write 98,631 3,651 44.44
log file switch completion 62 2,983 36.31
direct path read 37,434 1,413 17.20
db file sequential read 86 109 1.33
control file sequential read 3,862 34 .41

```

这可能意味着系统存在大量的磁盘排序操作。基于此，继续向下追查相关排序部分统计数据：

```

Instance Activity Stats for DB: DB Instance: db Snaps: 65 -

Statistic Total per Second per Trans

.....
sorts (disk) 64 0.1 0.4
sorts (memory) 861 0.7 4.7
sorts (rows) 2,804,580 2,194.5 15,159.9

```

64 次的 **sort disk**，相当显著的磁盘排序。

在 Statspack 的报告中，存在一个性能指标，称为内存排序率（In-memory Sort Ratio），用于衡量系统的排序操作，这个指标就是由以上两个统计信息 **Sort (disk)** 和 **Sort (memory)** 得出：

$$\text{In-memory Sort Ratio} = \text{Sort (memory)} / [\text{sorts (disk)} + \text{Sort (memory)}]$$

对于本例，这个比率计算值为：

$$\text{In-memory Sort Ratio} = 861 / (861 + 64) \approx 93.08 \%$$

从 Statspack 的报告中，也可以获得这个信息：

## Instance Efficiency Percentages (Target 100%)

```

~~~~~
Buffer Nowait Ratio:      100.00
Buffer Hit Ratio:        20.27
Library Hit Ratio:       98.81
Redo NoWait Ratio:       99.74
In-memory Sort Ratio:    93.08
Soft Parse Ratio:        98.66
Latch Hit Ratio:         100.00

```

对于显著的磁盘排序，可以很容易地猜测到，临时表空间的读写操作肯定相当频繁，从 Statspack 报告中文件 I/O 部分的统计数据可以验证：

File IO Statistics for DB: GHCXSDB Instance: ghcxsdbs Snaps: 65 - 66

| Tablespace | Filename                               |             |             |         |           |               |
|------------|----------------------------------------|-------------|-------------|---------|-----------|---------------|
| -----      |                                        |             |             |         |           |               |
|            | Reads                                  | Avg Blks Rd | Avg Rd (ms) | Writes  | Tot Waits | Avg Wait (ms) |
| -----      |                                        |             |             |         |           |               |
| PERFSTAT   | D:\ORACLE\ORADATA\PERFSTAT.DBF         |             |             |         |           |               |
|            | 88                                     | 1.0         | 12.5        | 821     | 0         |               |
| RBS        | D:\ORACLE\ORADATA\GHCXSDB\RBS01.DBF    |             |             |         |           |               |
|            | 7                                      | 1.0         | 0.0         | 1,399   | 0         |               |
| SYSTEM     | D:\ORACLE\ORADATA\GHCXSDB\SYSTEM01.DBF |             |             |         |           |               |
|            | 17                                     | 1.0         | 11.8        | 50      | 0         |               |
| TEMP       | D:\ORACLE\ORADATA\GHCXSDB\TEMP01.DBF   |             |             |         |           |               |
|            | 223,152                                | 1.5         | 0.2         | 371,303 | 0         |               |

对于这种情况，在 Oracle 9i 之前，可以适当增加 `sort_area_size` 的大小；从 Oracle 9i 开始，可以适当增大 `pga_aggregate_target`，以缩减磁盘排序对于硬盘的写入，从而提高系统及应用相应。但是，通常应该及时检查应用，确认是否因为应用问题而导致了过度排序，从而从根本上解决问题。

在 Oracle 10g 中，为了区分特定的对于临时文件的直接读写操作，Oracle 对 Direct Path Read/Write 进行了分离，将这类操作分列出来：

```

SQL> select event#,name,WAIT_CLASS
      2  from v$event_name where name like 'direct%';

```

| EVENT# NAME                | WAIT_CLASS |
|----------------------------|------------|
| -----                      | -----      |
| 161 direct path read       | User I/O   |
| 162 direct path read temp  | User I/O   |
| 163 direct path write      | User I/O   |
| 164 direct path write temp | User I/O   |

现在的 direct path read/write temp 就是单指对于临时文件的直接读写操作。结合 Oracle 10g 的一些特性，进一步研究一下直接路径读/写与临时文件。

首先在一个 session 中执行一个能够引发磁盘排序的查询：

```
SQL> select sid from v$mystat where rownum <2;

      SID
-----
      148

SQL> select a.table_name,b.object_name,b.object_type
2  from t1 a ,t2 b
3  where a.table_name = b.object_name
4  order by b.object_name,b.object_type;
```

在另外 session 查询相应等待事件：

```
SQL> select event,p1text,p1,p2text,p2,p3text,p3
2  from v$session_wait_history where sid=148
3  /
```

| EVENT                               | P1TEXT            | P1 P2TEXT            | P2 P3TEXT        | P3    |
|-------------------------------------|-------------------|----------------------|------------------|-------|
| -----                               | -----             | -----                | -----            | ----- |
| <b>direct path read temp</b>        | file number       | <b>201</b> first dba | 621872 block cnt | 31    |
| direct path read temp               | file number       | 201 first dba        | 621872 block cnt | 31    |
| direct path read temp               | file number       | 201 first dba        | 70232 block cnt  | 31    |
| direct path read temp               | file number       | 201 first dba        | 70232 block cnt  | 31    |
| direct path read temp               | file number       | 201 first dba        | 387706 block cnt | 15    |
| SQL*Net message to client driver id | 1650815232 #bytes |                      | 1                | 0     |
| direct path read temp               | file number       | 201 first dba        | 409915 block cnt | 31    |
| direct path read temp               | file number       | 201 first dba        | 409915 block cnt | 31    |
| direct path read temp               | file number       | 201 first dba        | 198777 block cnt | 16    |

```

direct path read temp      file number      201 first dba      198777 block cnt      16

10 rows selected

```

可以看到最近的 10 次等待，direct path read temp 就是这个查询引起的磁盘排序。注意这里的 file number 为 201，而实际上，临时文件的文件号仅为 1：

```

SQL> select file#,name
      2  from v$tempfile;

FILE# NAME
-----
      1 +ORADG/danally/tempfile/temp.267.600173887

```

如果通过 10046 事件跟踪，也可以获得类似的结果（与以上数据并非来自同一数据库）：

```

WAIT #1: nam='direct path write' ela= 11 p1=202 p2=16584 p3=7
WAIT #1: nam='direct path write' ela= 2 p1=202 p2=16591 p3=7
WAIT #1: nam='direct path write' ela= 2 p1=202 p2=16598 p3=7
WAIT #1: nam='direct path write' ela= 8 p1=202 p2=16605 p3=1
WAIT #1: nam='direct path read' ela= 81 p1=202 p2=12937 p3=31

```

在 Oracle 文档中，file# 被定义为绝对文件号（the absolute file number），这里的原因何在呢？研究这个问题的起因在于有朋友问起 V\$TEMPSEG\_USAGE 这个视图，从这个视图出发动手研究一下这个对象究竟来自何方。

查询 dba\_objects 视图，发现 V\$TEMPSEG\_USAGE 原来是一个同义词。

```

SQL> select object_type from dba_objects
      2 where object_name='V$TEMPSEG_USAGE';

OBJECT_TYPE
-----
SYNONYM

```

再追本溯源，原来 V\$TEMPSEG\_USAGE 是 V\_\$SORT\_USAGE 的同义词，也就是和 V\_\$SORT\_USAGE 同源。

```

SQL> select * from dba_synonyms
      2 where synonym_name='V$TEMPSEG_USAGE';

OWNER      SYNONYM_NAME      TABLE_OWNE TABLE_NAME      DB_LINK
-----
PUBLIC     V$TEMPSEG_USAGE SYS      V_$SORT_USAGE

```

如果再进一步，可以看到这个视图的构建语句：

```

SQL> SELECT view_definition FROM v$fixed_view_definition
      2 WHERE view_name='GV$SORT_USAGE';

```

## VIEW\_DEFINITION

```
-----
select x$ktssso.inst_id, username, username, ktssoses, ktssosno, prev_sql_addr, p
rev_hash_value, ktssotsn, decode(ktssocnt, 0, 'PERMANENT', 1, 'TEMPORARY'), deco
de(ktssosegt, 1, 'SORT', 2, 'HASH', 3, 'DATA', 4, 'INDEX', 5, 'LOB_DATA', 6, 'LO
B_INDEX', 'UNDEFINED'), ktssofno, ktssobno, ktsssoexts, ktssoblks, ktssorfno fro
m x$ktssso, v$session where ktssoses = v$session.saddr and ktssosno = v$session.s
erial#
```

格式化一下，v\$sort\_usage 的创建语句如下：

```
SELECT x$ktssso.inst_id, username, username, ktssoses, ktssosno, prev_sql_addr,
      prev_hash_value, ktssotsn,
      DECODE (ktssocnt, 0, 'PERMANENT', 1, 'TEMPORARY'),
      DECODE (ktssosegt,
              1, 'SORT',
              2, 'HASH',
              3, 'DATA',
              4, 'INDEX',
              5, 'LOB_DATA',
              6, 'LOB_INDEX',
              'UNDEFINED'
      ),
      ktssofno, ktssobno, ktsssoexts, ktssoblks, ktssorfno
FROM x$ktssso, v$session
WHERE ktssoses = v$session.saddr AND ktssosno = v$session.serial#
/
```

注意到在 Oracle 文档中 SEGFILE# 的定义为：

| SEGFILE# | NUMBER | File number of initial extent |
|----------|--------|-------------------------------|
|----------|--------|-------------------------------|

在视图中，这个字段来自 x\$ktssso.ktssofno，也就是说这个字段实际上代表的是绝对文件号。那么这个绝对文件号怎样与临时文件关联呢？能否与 V\$TEMPFILE 中的 file# 字段关联呢？

再来看一下 V\$TEMPFILE 的来源，V\$TEMPFILE 由如下语句创建：

```
SELECT tf.inst_id, tf.tfnun, TO_NUMBER (tf.tfrcr_scn),
      TO_DATE (tf.tfrcr_tim, 'MM/DD/RR HH24:MI:SS', 'NLS_CALENDAR=Gregorian'),
      tf.tftsn, tf.tfrfn,
      DECODE (BITAND (tf.tfsta, 2), 0, 'OFFLINE', 2, 'ONLINE', 'UNKNOWN'),
```

```

        DECODE (BITAND (tf.tfsta, 12),
                0, 'DISABLED',
                4, 'READ ONLY',
                12, 'READ WRITE',
                'UNKNOWN'
        ),
        fh.fhtmpfsz * tf.tfbsz, fh.fhtmpfsz, tf.tfcsz * tf.tfbsz, tf.tfbsz,
        fn.fnnam
FROM x$kcctf tf, x$kcctfn fn, x$kcvtfhtmp fh
WHERE fn.fnfno = tf.tfnum
      AND fn.fnfno = fh.htmpxfil
      AND tf.tffnh = fn.fnum
      AND tf.tfdup != 0
      AND fn.fntyp = 7
      AND fn.fnnam IS NOT NULL

```

考察 x\$kcctf 底层表，注意到 TFAFN（Temp File Absolute File Number）在这里存在：

```
SQL> desc x$kcctf
```

| Name         | Null? | Type          |
|--------------|-------|---------------|
| ADDR         |       | RAW(4)        |
| INDX         |       | NUMBER        |
| INST_ID      |       | NUMBER        |
| TFNUM        |       | NUMBER        |
| <b>TFAFN</b> |       | <b>NUMBER</b> |
| TFCSZ        |       | NUMBER        |
| TFBSZ        |       | NUMBER        |
| TFSTA        |       | NUMBER        |
| TFCRC_SCN    |       | VARCHAR2(16)  |
| TFCRC_TIM    |       | VARCHAR2(20)  |
| TFFNH        |       | NUMBER        |
| TFFNT        |       | NUMBER        |
| TFDUP        |       | NUMBER        |
| TFTSN        |       | NUMBER        |
| TFTSI        |       | NUMBER        |
| TFRFN        |       | NUMBER        |
| TFPFT        |       | NUMBER        |

而这个字段在构建 v\$tempfile 时并未出现, 所以不能通过 v\$sort\_usage 和 v\$tempfile 直接关联绝对文件号。可以简单构建一个排序段使用, 然后来继续研究一下:

```
SQL> select username,segtype,segfile#,segbld#,extents,segrfno#
2  from v$sort_usage;
USERNAME SEGTYPE      SEGFILE#  SEGBLD#  EXTENTS  SEGRFNO#
-----
SYS      LOB_DATA          9      18953      1        1
```

看到这里的 SEGFILE#=9, 而在 v\$tempfile 是找不到这个信息的:

```
SQL> select file#,rfile#,ts#,status,blocks
2  from v$tempfile;
FILE#    RFILE#    TS# STATUS    BLOCKS
-----
1        1        2 ONLINE     38400
```

可以从 x\$kcctf 中获得这些信息, 并可以看到 v\$tempfile.file# 实际上来自 x\$kcctf.tfnum, 是临时文件的文件号; 而绝对文件号是 x\$kcctf.tfafn, 只有这个字段才可以与 v\$sort\_usage.segfile# 关联。

```
SQL> select indx, tfnum, tfafn, tfcsz
2  from x$kcctf;
INDX      TFNUM      TFAFN      TFCSZ
-----
0         1         9         38400
1         2        10        12800
```

再进一步, 实际上, 为了分离临时文件号和数据文件号, Oracle 对临时文件的编号以 db\_files 为起点, 所以临时文件的绝对文件号应该等于 db\_files+file#。

看前面引用到的 Oracle 10g 数据库的设置:

```
SQL> select indx,tfnum,tfafn,tfcsz
2  from x$kcctf;
INDX      TFNUM      TFAFN      TFCSZ
-----
0         1        201        2560
```

db\_files 参数的缺省值为 200:

```
SQL> show parameter db_files
NAME                                TYPE        VALUE
-----
db_files                            integer     200
```

```
db_files                                integer      200
```

```
SQL> select file#,name from v$tempfile;
```

```
FILE# NAME
```

```
-----
1 +ORADG/danaly/tempfile/temp.267.600173887
```

```
SQL>
```

所以在 Oracle 文档中，v\$tempfile.file#被定义为 The absolute file number 是不确切的。偶尔地，用户可能会在警报日志文件中看到类似以下的错误：

```
***

Corrupt block relative dba: 0x00c0008a (file 202, block 138)
Bad header found during buffer read
Data in bad block -
  type: 8 format: 2 rdba: 0x0140008a
  last change scn: 0x0000.431f8beb seq: 0x1 flg: 0x08
  consistency value in tail: 0x8beb0801
  check value in block header: 0x0, block checksum disabled
  spare1: 0x0, spare2: 0x0, spare3: 0x0

***
```

这里的 file 202 其实指的就是临时文件。

以上的整个过程更主要的是说明一个思路，供大家在解决或研究问题时参考。

#### 8.5.4 日志文件相关等待

第 6 章已详细介绍了重做的相关知识，Redo 对于数据库来说非常重要，与一系列等待事件和日志相关，通过 v\$event\_name 视图可以找到这些等待事件：

```
SQL> select name from v$event_name
```

```
2  where name like '%log%';
```

```
NAME
```

```
-----
log switch/archive
```

```
log file sequential read
```

```
log file single write
```

```
log file parallel write
```

```

log buffer space
log file switch (checkpoint incomplete)
log file switch (archiving needed)
log file switch (clearing log file)
switch logfile command
log file switch completion
log file sync
STREAMS capture process waiting for archive log
rfrxptarcurllog

13 rows selected.

```

这里摘录几个重要事件，简要介绍一下。

### 1. Log File Switch（日志文件切换）

**Log File Switch** 当日志文件发生切换时出现，在数据库进行日志切换时，**LGWR** 需要关闭当前日志组，切换并打开下一个日志组，在这个切换过程中，数据库的所有 **DML** 操作都处于停顿状态，直至这个切换完成。

**Log file Switch** 主要包含两个子事件。

(1) **log file switch (archiving needed)**，即日志切换（需要归档）。

这个等待事件出现时通常是因为日志组循环写满以后，在需要覆盖先前日志时，发现日志归档尚未完成，出现该等待。由于 **Redo** 不能写出，该等待出现时，数据库将陷于停顿状态。

出现该等待，可能表示 **I/O** 存在问题、归档进程写出缓慢，也有可能是日志组设置不合理等原因导致。针对不同原因，可以考虑采用的解决办法有：

- 可以考虑增大日志文件和增加日志组；
- 移动归档文件到快速磁盘；
- 调整 **log\_archive\_max\_processes** 参数等。

(2) **log file switch (checkpoint incomplete)**，即日志切换（检查点未完成）。

当所有的日志组都写满之后，**LGWR** 试图覆盖某个日志文件，如果这时数据库没有完成写出由这个日志文件所保护的脏数据时（检查点未完成），该等待事件出现。该等待出现时，数据库同样将陷于停顿状态。同时警告日志文件中会记录如下信息：

```

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

```

该等待事件通常表示 **DBWR** 写出速度太慢或者 **I/O** 存在问题。为解决该问题，可能需要考虑增加额外的 **DBWR** 或者增加日志组或日志文件大小。

**log file switch** 引起的等待都是非常重要的，如果出现就应该引起重视，并由 **DBA** 介入

进行及时处理。

## 2. Log File Sync (日志文件同步)

当一个用户提交或回滚数据时，LGWR 将会话期的重做由日志缓冲器写入到重做日志中，LGWR 完成任务以后会通知用户进程。日志文件同步过程 (Log File Sync) 必须等待这一过程成功完成。对于回滚操作，该事件记录从用户发出 Rollback 命令到回滚完成的时间。

如果该等待过多，可能说明 LGWR 的写出效率低下，或者系统提交过于频繁。针对该问题，可以通过 log file parallel write 等待事件或 User Commits、User Rollback 等统计信息来观察提交或回滚次数。

可能的解决方案主要有：

- 提高 LGWR 性能，尽量使用快速磁盘，不要把 redo log file 存放在 RAID5 的磁盘上；
- 使用批量提交；
- 适当使用 NOLOGGING/UNRECOVERABLE 等选项。

可以通过如下公式计算平均 Redo 写大小：

$$\text{avg.redo write size} = (\text{Redo block written/redos writes}) * 512 \text{ bytes}$$

如果系统产生 Redo 很多，而每次写的较少，一般说明 LGWR 被过于频繁的激活了。可能导致过多的 Redo 相关 Latch 的竞争，而且 Oracle 可能无法有效地使用 piggyback 的功能。

下面从一个 Statspack 中提取一些数据来研究一下这个问题，Report 的概要信息如下：

| DB Name      | DB Id             | Instance           | Inst Num   | Release           | OPS             | Host |
|--------------|-------------------|--------------------|------------|-------------------|-----------------|------|
| -----        |                   |                    |            |                   |                 |      |
| DB           | 1222010599        | oracle             | 1          | 8.1.7.4.5         | NO              | sun  |
|              | Snap Id           | Snap Time          | Sessions   |                   |                 |      |
| -----        |                   |                    |            |                   |                 |      |
| Begin Snap:  | 3473              | 13-Oct-04 13:43:00 | 540        |                   |                 |      |
| End Snap:    | 3475              | 13-Oct-04 14:07:28 | 540        |                   |                 |      |
| Elapsed:     | 24.47 (mins)      |                    |            |                   |                 |      |
|              |                   |                    |            |                   |                 |      |
| Cache Sizes  |                   |                    |            |                   |                 |      |
| ~~~~~        |                   |                    |            |                   |                 |      |
|              | db_block_buffers: | 102400             |            | log_buffer:       | 20971520        |      |
|              | db_block_size:    | 8192               |            | shared_pool_size: | 600M            |      |
|              |                   |                    |            |                   |                 |      |
| Load Profile |                   |                    |            |                   |                 |      |
| ~~~~~        |                   |                    |            |                   |                 |      |
|              |                   |                    | Per Second |                   | Per Transaction |      |
|              |                   |                    | -----      |                   | -----           |      |
|              | Redo size:        |                    | 28,458.11  |                   | 2,852.03        |      |
|              | .....             |                    |            |                   |                 |      |

等待事件如下：

| Event                             | Waits         | Timeouts | Time (cs)    | (ms)      | /txn       |
|-----------------------------------|---------------|----------|--------------|-----------|------------|
| -----                             |               |          |              |           |            |
| <b>log file sync</b>              | <b>14,466</b> | <b>2</b> | <b>4,150</b> | <b>3</b>  | <b>1.0</b> |
| db file sequential read           | 17,202        | 0        | 2,869        | 2         | 1.2        |
| latch free                        | 24,841        | 13,489   | 2,072        | 1         | 1.7        |
| direct path write                 | 121           | 0        | 1,455        | 120       | 0.0        |
| db file parallel write            | 1,314         | 0        | 1,383        | 11        | 0.1        |
| <b>log file sequential read</b>   | <b>1,540</b>  | <b>0</b> | <b>63</b>    | <b>0</b>  | <b>0.1</b> |
| ....                              |               |          |              |           |            |
| <b>log file switch completion</b> | <b>1</b>      | <b>0</b> | <b>3</b>     | <b>30</b> | <b>0.0</b> |
| refresh controlfile command       | 23            | 0        | 1            | 0         | 0.0        |
| <b>LGWR wait for redo copy</b>    | <b>46</b>     | <b>0</b> | <b>0</b>     | <b>0</b>  | <b>0.0</b> |
| ....                              |               |          |              |           |            |
| <b>log file single write</b>      | <b>4</b>      | <b>0</b> | <b>0</b>     | <b>0</b>  | <b>0.0</b> |

可以看到，这里 log file sync 和 db file parallel write 等等待事件同时出现，那么可能的一个原因是 I/O 竞争导致了性能问题，实际用户环境正是日志文件和数据文件同时存放在 RAID5 的磁盘上，存在性能问题，需要调整。

统计信息如下：

| Statistic                      | Total      | per Second | per Trans |
|--------------------------------|------------|------------|-----------|
| -----                          |            |            |           |
| ....                           |            |            |           |
| redo blocks written            | 93,853     | 63.9       | 6.4       |
| redo buffer allocation retries | 1          | 0.0        | 0.0       |
| redo entries                   | 135,837    | 92.5       | 9.3       |
| redo log space requests        | 1          | 0.0        | 0.0       |
| redo log space wait time       | 3          | 0.0        | 0.0       |
| redo ordering marks            | 0          | 0.0        | 0.0       |
| redo size                      | 41,776,508 | 28,458.1   | 2,852.0   |
| redo synch time                | 4,174      | 2.8        | 0.3       |
| redo synch writes              | 14,198     | 9.7        | 1.0       |
| redo wastage                   | 4,769,200  | 3,248.8    | 325.6     |
| redo write time                | 3,698      | 2.5        | 0.3       |
| redo writer latching time      | 0          | 0.0        | 0.0       |
| redo writes                    | 14,572     | 9.9        | 1.0       |
| ....                           |            |            |           |
| sorts (disk)                   | 4          | 0.0        | 0.0       |
| sorts (memory)                 | 179,856    | 122.5      | 12.3      |

|                                   |               |            |            |
|-----------------------------------|---------------|------------|------------|
| sorts (rows)                      | 2,750,980     | 1,874.0    | 187.8      |
| ....                              |               |            |            |
| transaction rollbacks             | 36            | 0.0        | 0.0        |
| transaction tables consistent rea | 0             | 0.0        | 0.0        |
| transaction tables consistent rea | 0             | 0.0        | 0.0        |
| user calls                        | 1,390,718     | 947.4      | 94.9       |
| <b>user commits</b>               | <b>14,136</b> | <b>9.6</b> | <b>1.0</b> |
| user rollbacks                    | 512           | 0.4        | 0.0        |
| write clones created in backgroun | 0             | 0.0        | 0.0        |
| write clones created in foregroun | 11            | 0.0        | 0.0        |
| -----                             |               |            |            |

根据统计信息可以计算平均日志写大小：

$$\begin{aligned}
 \text{avg.redo write size} &= (\text{Redo Block Written}/\text{Redo Writes}) \times 512 \text{ bytes} \\
 &= (93853 / 14572) \times 512 \text{ bytes} \\
 &= 3\text{KB}
 \end{aligned}$$

这个平均值过小了，说明系统的提交过于频繁。从以上的统计信息中，可以看到平均每秒数据库的提交数量是 9.6 次。如果可能，在设计应用时应选择合适的提交批量，从而提高数据库的效率。

| Latch Sleep breakdown for DB: DPSHDB Instance: dpshdb Snaps: 3473 -3475 |               |            |                             |
|-------------------------------------------------------------------------|---------------|------------|-----------------------------|
| -> ordered by misses desc                                               |               |            |                             |
| Latch Name                                                              | Get Requests  | Misses     | Spin & Sleeps 1->4          |
| -----                                                                   |               |            |                             |
| row cache objects                                                       | 12,257,850    | 113,299    | 64 113235/64/0/0/0          |
| shared pool                                                             | 3,690,715     | 60,279     | 15,857 52484/588/6546/661/0 |
| library cache                                                           | 4,912,465     | 29,454     | 8,876 23823/2682/2733/216/0 |
| cache buffers chains                                                    | 10,314,526    | 2,856      | 33 2823/33/0/0/0            |
| <b>redo writing</b>                                                     | <b>76,550</b> | <b>937</b> | <b>1 936/1/0/0/0</b>        |
| session idle bit                                                        | 2,871,949     | 225        | 1 224/1/0/0/0               |
| messages                                                                | 107,950       | 159        | 2 157/2/0/0/0               |
| session allocation                                                      | 184,386       | 44         | 6 38/6/0/0/0                |
| checkpoint queue latch                                                  | 96,583        | 1          | 1 0/1/0/0/0                 |

由于过度频繁的提交，LGWR 过度频繁地激活，这里就出现了 Redo Writing 的 Latch 竞争。

### 3. log file single write

该事件仅与写日志文件头块相关，通常发生在增加新的组成员和增进序列号（Log Switch）时。头块写单个进行，因为头块的部分信息是文件号，每个文件不同。更新日志文件头这个操作在后台完成，一般很少出现等待，无需太多关注。

在 Log Switch 过程中，LGWR 需要改写日志文件头，有时可以观察到该等待事件的增加：

```
SQL> select event,time_waited from v$system_event where event='log file single write';
```

| EVENT                 | TIME_WAITED |
|-----------------------|-------------|
| log file single write | 2848        |

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

System altered.

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

System altered.

```
SQL> select event,time_waited from v$system_event where event='log file single write';
```

| EVENT                 | TIME_WAITED |
|-----------------------|-------------|
| log file single write | 2853        |

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

System altered.

```
SQL> select event,time_waited from v$system_event where event='log file single write';
```

| EVENT                 | TIME_WAITED |
|-----------------------|-------------|
| -----                 |             |
| log file single write | 2855        |

#### 4. Log File Parallel Write

从 Log Buffer 写 Redo 记录到日志文件，主要指常规写操作（相对于 Log File Sync）。如果 Log Group 存在多个组成员，当 Flush Log Buffer 时，写操作是并行的，这时候此等待事件可能出现。

尽管这个写操作并行处理，直到所有 I/O 操作完成该写操作才会完成（如果磁盘支持异步 I/O 或者使用 IO SLAVE，那么即使只有一个 Redo Log File Member，也有可能出现此等待）。这个参数和 Log File Sync 时间相比较，可以用来衡量 Log File 的写入成本，通常称为同步成本率。

#### 5. Log Buffer Space（日志缓冲空间）

当数据库产生日志的速度比 LGWR 的写出速度快，或者是当日志切换（Log Switch）太慢时，就会发生这种等待。这个等待出现时，通常表明 Redo Log Buffer 过小，为解决这个问题，可以考虑增大日志文件的大小，或者增加日志缓冲器的大小。

另外一个可能的原因是磁盘 I/O 存在瓶颈，可以考虑使用写入速度更快的磁盘。在允许的条件下设置，可以考虑使用裸设备来存放日志文件，提高写入效率。在一般的系统中，最低的标准是，不要把日志文件和数据文件存放在一起，因为通常日志文件只写不读，分离存放可以获得性能提升，尽量使用 RAID10 而不是 RAID5 磁盘来存储日志文件。

以下是一个 Log Buffer 存在问题的 Statspack Top5 等待事件的系统：

| Top 5 Wait Events           |               |                |             |
|-----------------------------|---------------|----------------|-------------|
| ~~~~~                       |               |                |             |
|                             | Wait          |                | % Total     |
| Event                       | Waits         | Time (cs)      | Wt Time     |
| -----                       | -----         | -----          | -----       |
| log file parallel write     | 1,436,993     | 1,102,188      | 10.80       |
| <b>log buffer space</b>     | <b>16,698</b> | <b>873,203</b> | <b>8.56</b> |
| log file sync               | 1,413,374     | 654,587        | 6.42        |
| control file parallel write | 329,777       | 510,078        | 5.00        |
| db file scattered read      | 425,578       | 132,537        | 1.30        |
| -----                       | -----         | -----          | -----       |

Log Buffer Space 等待事件出现时，数据库将陷于停顿状态，所有的和日志生成相关的操作全部不能进行，所以这个等待事件应该引起充分的重视。

#### 8.5.5 Enqueue（队列等待）

Enqueue 是一种保护共享资源的锁定机制。该锁定机制保护共享资源，以避免因并发操作而损坏数据，比如通过锁定保护一行记录，避免多个用户同时更新。Enqueue 采用排队机

制，即 FIFO（先进先出）来控制资源的使用。

Enqueue 是一组锁定事件的集合，如果数据库中这个等待事件比较显著，还需要进一步追踪是哪一个类别的锁定引发了数据库等待。

从 Oracle 10g 开始，Oracle 对于队列等待进行了细分，V\$EVENT\_NAME 视图中可以查询这些细分后的等待事件，简要摘录几个示例如下：

```
SQL> select name,wait_class
```

```
2 from v$event_name where name like '%enq%';
```

| NAME                             | WAIT_CLASS    |
|----------------------------------|---------------|
| enq: PW - flush prewarm buffers  | Application   |
| enq: RO - contention             | Application   |
| enq: RO - fast object reuse      | Application   |
| enq: KO - fast object checkpoint | Application   |
| enq: TM - contention             | Application   |
| enq: ST - contention             | Configuration |
| enq: HW - contention             | Configuration |
| enq: SS - contention             | Configuration |
| enq: TX - row lock contention    | Application   |
| enq: TX - allocate ITL entry     | Configuration |
| enq: TX - index contention       | Concurrency   |
| .....                            |               |

Enqueue 等待常见的有 ST、HW、TX、TM 等。ST Enqueue 用于空间管理和字典管理的表空间（DMT）的区间分配。在 DMT 中，典型的是对于 uet\$ 和 fet\$ 数据字典表的争用。对于支持 LMT 的版本，应该尽量使用本地管理表空间，或者考虑手工预分配一定数量的区（Extent），以减少动态扩展时发生的严重队列竞争。

通过一个实例来看一下：

| DB Name | DB Id    | Instance | Inst Num | Release      | OPS Host |
|---------|----------|----------|----------|--------------|----------|
| DB      | 40757346 | aaa      | 1        | 8.1.7.4.0 NO | server   |

|             | Snap Id      | Snap Time           | Sessions |
|-------------|--------------|---------------------|----------|
| Begin Snap: | 2845         | 31-10 月-03 02:10:16 | 46       |
| End Snap:   | 2848         | 31-10 月-03 03:40:05 | 46       |
| Elapsed:    | 89.82 (mins) |                     |          |

对于一个 Statspack 的 Report，采样时间是非常重要的维度，离开时间做参考，任何等待

都不足以说明问题。

| Top 5 Wait Events           |               |                   |              |
|-----------------------------|---------------|-------------------|--------------|
| ~~~~~                       |               |                   |              |
| Event                       | Wait          |                   | % Total      |
|                             | Waits         | Time (cs)         | Wt Time      |
| -----                       |               |                   |              |
| <b>enqueue</b>              | <b>53,793</b> | <b>16,192,686</b> | <b>67.86</b> |
| rdbms ipc message           | 19,999        | 5,927,350         | 24.84        |
| pmon timer                  | 1,754         | 538,797           | 2.26         |
| smon timer                  | 17            | 522,281           | 2.19         |
| SQL*Net message from client | 94,525        | 520,104           | 2.18         |
| -----                       |               |                   |              |

在 Statspack 分析中, Top 5 等待事件是我们最为关注的部分。这个系统中, 除了 Enqueue 等待事件以外, 其他 4 个都属于空闲等待事件, 无须关注。

来关注一下 Enqueue 等待事件, 在 89.82 (mins) 的采样间隔内, 累计 Enqueue 等待长达 16192686cs, 即 45 小时左右。这个等待已经太过显著, 实际上这个系统也正因此遭遇了巨大的困难, 观察到队列等待以后, 就应该关注队列等待在等待什么资源。快速跳转 Statspack 的其他部分, 看到以下详细内容:

Enqueue activity for DB: DB Instance: aaa Snaps: 2716 -2718

-> ordered by waits desc, gets desc

| Enqueue | Gets  | Waits |
|---------|-------|-------|
| -----   |       |       |
| ST      | 1,554 | 1,554 |
| -----   |       |       |

看到主要队列等待在等待 ST 锁定, 对于 DMT, 可以说这个等待和 FET\$、UET\$ 的争用紧密相关。再回过头来研究捕获的 SQL 语句:

-> End Buffer Gets Threshold: 10000

-> Note that resources reported for PL/SQL includes the resources used by all SQL statements called within the PL/SQL code. As individual SQL statements are also reported, it is possible and valid for the summed total % to exceed 100

| Buffer Gets                                                      | Executions | Gets per Exec | % Total | Hash Value |
|------------------------------------------------------------------|------------|---------------|---------|------------|
| -----                                                            |            |               |         |            |
| 4,800,073                                                        | 10,268     | 467.5         | 51.0    | 2913840444 |
| select length from fet\$ where file#=:1 and block#=:2 and ts#=:3 |            |               |         |            |

```

      803,187      10,223      78.6      8.5      528349613
delete from uet$ where ts#=:1 and segfile#=:2 and segblock#=:3 a
nd ext#=:4

      454,444      10,300      44.1      4.8      1839874543
select file#,block#,length from uet$ where ts#=:1 and segfile#=:
2 and segblock#=:3 and ext#=:4

      23,110      10,230      2.3      0.2      3230982141
insert into fet$ (file#,block#,ts#,length) values (:1,,:2,,:3,,:4)

      21,201      347      61.1      0.2      1705880752
select file# from file$ where ts#=:1
....
      9,505      12      792.1      0.1      1714733582
select f.file#, f.block#, f.ts#, f.length from fet$ f, ts$ t whe
re t.ts#=f.ts# and t.dflextpct!=0 and t.bitmapped=0

      6,426      235      27.3      0.1      1877781575
delete from fet$ where file#=:1 and block#=:2 and ts#=:3

```

可以看到数据库频繁操作 UET\$、FET\$ 系统表已经成为了系统的主要瓶颈。

至此，已经可以准确地为该系统定位问题，相应的解决方案也很容易确定，在 8.1.7 一节中，使用 LMT 代替 DMT，这是解决问题的根本办法，当然实施起来还要进行综合考虑，实际情况还要复杂得多。其他类别的队列竞争，这里不再过多介绍。

### 8.5.6 Latch Free（闕锁释放）

Latch Free 通常被称为闕锁释放，这个名称常常引起误解，实际上应该在前面加上一个“等待”（wait），当数据库出现这个等待时，说明有进程正在等待某个 Latch 被释放，也就是 Waiting Latch Free。

Latch 是一种低级排队（串行）机制，用于保护 SGA 中共享内存结构。Latch 就像是一种快速的被获取和释放的内存锁，用于防止共享内存结构被多个用户同时访问。

其实不必把 Latch 想得过于复杂，Latch 通常是操作系统是利用内存中的某个位置，通过设置变量为 0 或非 0，来表示 Latch 是否已经被取得，大多数的操作系统，是使用 test and set 的方式来完成 Latch 持有检查或持有的。

#### 1. Latch 的分类

在数据库内部，Oracle 通过 v\$latch 视图记录不同类型 Latch 的统计数据，按获取和等待

方式不同进行分类，Latch 请求的类型可分为两类：willing-to-wait 和 immediate。

- willing-to-wait: 是指如果所请求的 Latch 不能立即得到，请求进程将等待一段很短的时间后再次发出请求。进程一直重复此过程直到得到 Latch。

- immediate: 是指如果所请求的 Latch 不能立即得到，请求进程就不再等待，而是继续执行下去。

在 v\$latch 中的以下字段记录了 willing-to-wait 请求。

- GETS: 记录成功地以 willing-to-wait 请求类型请求一个 Latch 的次数。

- MISSES: 记录初始以 willing-to-wait 请求类型请求一个 Latch 不成功，而进程进入等待的次数。

- SLEEPS: 记录初始以 willing-to-wait 请求类型请求一个 Latch 不成功，进程等待获取 Latch 时进入休眠的次数。

在 v\$latch 中的以下字段记录了 immediate 类请求。

- IMMEDIATE\_GETS: 记录以 immediate 请求类型成功地获得一个 Latch 的次数。

- IMMEDIATE\_MISSES: 记录以 immediate 请求类型请求一个 Latch 不成功的次数。

Oracle 的 Latch 机制是竞争，其处理类似于网络里的 CSMA/CD，所有用户进程争夺 Latch，对于愿意等待类型（willing-to-wait）的 Latch，如果一个进程在第一次尝试中没有获得 Latch，那么它会等待并且再次尝试，如果系统存在多个 CPU，那么此进程将围绕该 Latch 开始自旋（spin），如果经过 `_spin_count` 次争夺不能获得 Latch，然后该进程转入睡眠状态，持续一段指定长度的时间，然后再次醒来，按顺序重复以前的步骤。这一过程可以通过图 8-10 来说明。

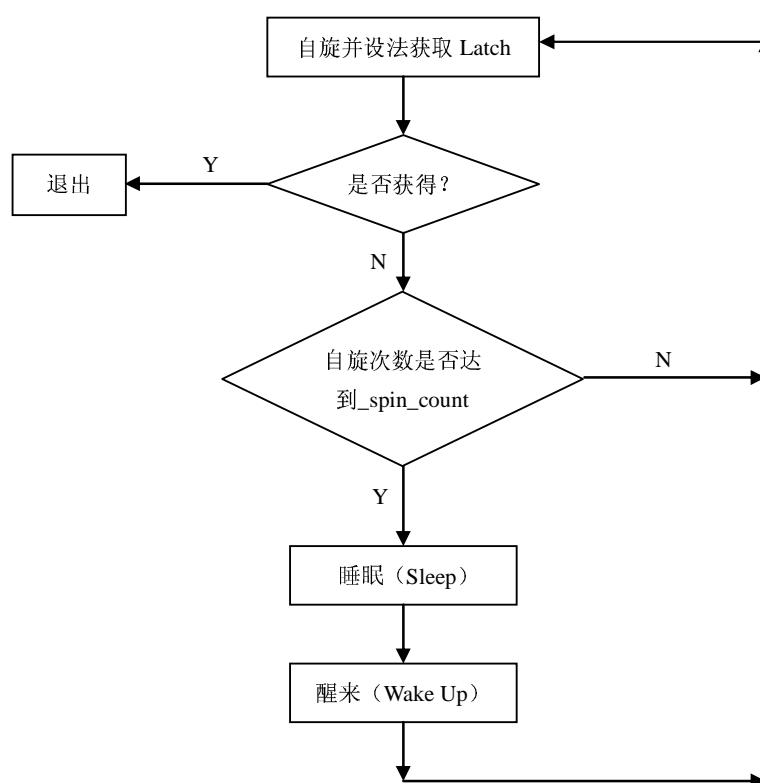


图 8-10 用户进程争夺 Latch 的过程

spin 的次数受隐含参数\_spin\_count 影响，该参数的缺省值为 2000。以下数据取自 Oracle 10gR2 Linux 环境：

```
SQL> select * from v$version;
```

BANNER

-----  
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod

PL/SQL Release 10.2.0.1.0 - Production

CORE 10.2.0.1.0 Production

TNS for Linux: Version 10.2.0.1.0 - Production

NLSRTL Version 10.2.0.1.0 - Production

该系统存在 4 个 CPU：

```
SQL> show parameter cpu_count
```

| NAME      | TYPE    | VALUE |
|-----------|---------|-------|
| -----     |         |       |
| cpu_count | integer | 4     |

spin\_count 的缺省值即为 2000：

```
SQL> @GetHparDes.sql
```

Enter value for par: spin

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

new 6: AND x.ksppinm LIKE '%spin%'

| NAME        | VALUE | DESCRIB                            |
|-------------|-------|------------------------------------|
| -----       |       |                                    |
| _spin_count | 2000  | Amount to spin waiting for a latch |

从以上过程可以看到，在 spin 的过程中，进程会一直持有 CPU，spin 的机制是假设 Latch 可以被快速释放（正常情况下 Latch 的持有时间是微秒级，相对 spin 机制如果直接采用 Sleep 方式引起的上下文切换会相当昂贵，所以 Oracle 针对 Latch 引入了 spin 算法），如果其他 CPU 上的其他进程释放了 Latch，spin 进程就可以立即获得这个 Latch。如果系统只有单 CPU，那就谈不上 spin 了。另一方面也可以看到，Latch 竞争是非常昂贵的，可能导致严重的 CPU 耗用，所以 Latch 竞争在任何时候都应该引起充分的重视。经过 spin 后成功获得 Latch 的次数被记录在 v\$latch.spin\_gets 字段。下面通过图 8-11 来看一下 Latch 的示意。

来看一下 willing-to-wait 和 immediate 两类 Latch 的大致数量，以下查询来自 Oracle 10gR2（同以上数据库）：

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

COUNT(\*)

```

-----
382
SQL> select count(*) from v$latch where IMMEDIATE_GETS + IMMEDIATE_MISSES >0;
COUNT(*)
-----
35
SQL> select count(*) from v$latch where IMMEDIATE_GETS + IMMEDIATE_MISSES =0;
COUNT(*)
-----
347

```

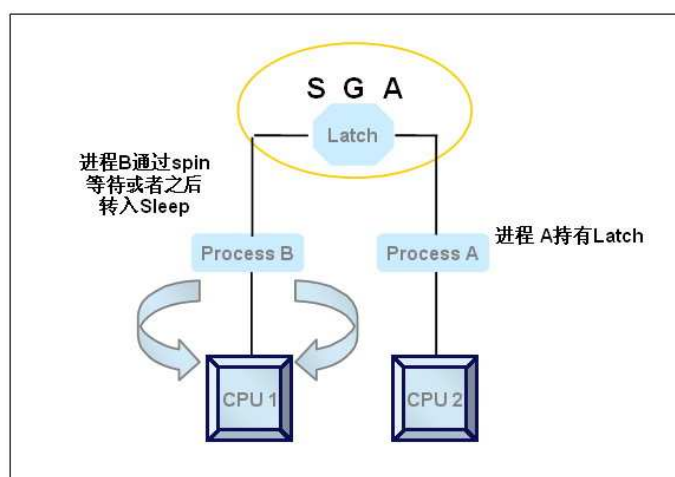


图 8-11 Latch 竞争示意图

看到 willing-to-wait 类型的等待事件占了绝大部分，immediate 类型的仅为少数：

```

SQL> SELECT  NAME, immediate_gets, immediate_misses, spin_gets
2      FROM v$latch
3  WHERE immediate_gets + immediate_misses > 0
4  ORDER BY immediate_gets DESC;

```

| NAME                    | IMMEDIATE_GETS   | IMMEDIATE_MISSES | SPIN_GETS  |
|-------------------------|------------------|------------------|------------|
| cache buffers lru chain | 259891274        | 209819           | 213249     |
| cache buffers chains    | 258525736        | 1470             | 18065      |
| <b>redo copy</b>        | <b>247810939</b> | <b>7184</b>      | <b>0</b>   |
| <b>redo allocation</b>  | <b>247808297</b> | <b>9909</b>      | <b>926</b> |
| checkpoint queue latch  | 56443129         | 4945             | 3825       |
| simulator lru latch     | 18277055         | 874              | 12027      |

|                                            |         |      |      |
|--------------------------------------------|---------|------|------|
| cache table scan latch                     | 7145539 | 2541 | 0    |
| SGA IO buffer pool latch                   | 2468707 | 0    | 0    |
| hash table column usage latch              | 694245  | 0    | 0    |
| In memory undo latch                       | 189592  | 0    | 1464 |
| active service list                        | 181530  | 0    | 1    |
| Memory Management Latch                    | 181372  | 0    | 1    |
| SQL memory manager latch                   | 178333  | 0    | 0    |
| KTF sga latch                              | 176088  | 0    | 0    |
| post/wait queue                            | 66086   | 0    | 0    |
| library cache                              | 40787   | 2    | 104  |
| enqueue hash chains                        | 26261   | 1    | 179  |
| object queue header heap                   | 11443   | 0    | 0    |
| MQL Tracking Latch                         | 10800   | 0    | 0    |
| row cache objects                          | 5323    | 0    | 80   |
| longop free list parent                    | 553     | 0    | 0    |
| process allocation                         | 479     | 0    | 0    |
| msg queue                                  | 92      | 0    | 0    |
| process queue reference                    | 45      | 0    | 0    |
| kmcpvec latch                              | 35      | 0    | 0    |
| object stats modification                  | 3       | 0    | 0    |
| query server process                       | 3       | 0    | 0    |
| active checkpoint queue latch              | 2       | 0    | 1    |
| alert log latch                            | 2       | 0    | 0    |
| slave class                                | 2       | 0    | 0    |
| JS mem alloc latch                         | 2       | 0    | 0    |
| multiblock read objects                    | 2       | 0    | 1160 |
| hash table modification latch              | 2       | 0    | 0    |
| rules engine evaluation context statistics | 2       | 0    | 0    |
| RSM SQL latch                              | 1       | 0    | 0    |
| 35 rows selected.                          |         |      |      |

需要注意的是, `immediate` 类型的 Latch 通常是因为存在多个可用 Latch, 最常见的如 Redo Copy Latch, 当 process 想要取得 Redo Copy Latch 时, 它首先要求其中一个 Latch, 如果可以取得就持有该 Latch, 如果不能获取, 它会立刻转向要求另一个 Redo Copy Latch, 只有所有的 Redo Copy Latch 都无法取得时, 才会 sleep 与 wait。

`immediate` 的另外一种原因是每个 Latch 都有 level 的概念 (level=1-14), 当一个 process 需要取得一组 Latches 时, 为避免死锁, 取得 Latches 有一定的顺序, 即 process 新请求的 Latch

的 level, 应该大于 process 目前所握有的 Latch 的 level。所以如果 process 要求的新 Latch 的 level 小于目前所持有的 Latch 的 level, 正常情况下, Oracle 要求 process 先释放目前所持有的所有 Latch, 再依序取得这些 Latch。为节省时间, Oracle 允许进程以 no-wait 方式要求较低 level 的 Latch, 如果成功取得, 既可以避免 deadlock 又可以节省时间。

## 2. Redo Copy Latch

下面以 Redo Copy Latch 为例简要说明一下 immediate 类型及 Latch 的处理。

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 Oracle 7 和 Oracle 8 中, `_LOG_SIMULTANEOUS_COPIES` 缺省情况下等于 CPU 的数量。从 Oracle 8.1.3 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> @GetHparDes.sql
Enter value for par: copies
old 6: AND x.ksppinm LIKE '%&par%'
new 6: AND x.ksppinm LIKE '%copies%'

NAME                                VALUE      DESCRIB
-----
_log_simultaneous_copies            8          number of simultaneous copies into redo buffer(# of copy latches)
```

从 `v$latch` 视图中可以得到关于 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，process 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 Post，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 Post。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，可能意味着用户的提交过于频繁。通过系统统计信息或 Statspack 可以获得这些信息，具体可参考关于“Log File Sync 等待事件”的内容。

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 来检查是否相应的 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，读者可以根据自己的侧重进行阅读或选择忽略此节。

### 3. Redo Allocation Latch

与 Redo 相关的另外一个常见 Latch 是 Redo Allocation Latch，在上一节中也多次提到这个 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      |
| <b>redo copy</b>        | <b>247810939</b> | <b>7184</b>      | <b>0</b>   |
| <b>redo allocation</b>  | <b>247808297</b> | <b>9909</b>      | <b>926</b> |
| checkpoint queue latch  | 56443129         | 4945             | 3825       |

在 Oracle 9iR2 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 Redo Log Buffer 区域，每个区域都按照初始化参数 log\_buffer 大小分配。

如果用户使用的是高端服务器，有超过 16 个处理器，正在经历非常高的 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 V\$LATCH 视图观察 Redo Allocation Latch 竞争的累计等待时间。使用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 的比率超过 1%，或者 IMMEDIATE\_MISSES 对（IMMEDIATE\_GETS+IMMEDIATE\_MISSES）的比率超过 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。用户可以从低值（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 Oracle 9iR2 中，该参数的缺省值为 1：

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production
```

```
SQL> show parameter log_p
```

| NAME            | TYPE    | VALUE |
|-----------------|---------|-------|
| -----           |         |       |
| log_parallelism | integer | 1     |

在 Oracle 10g 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。

缺省情况下, `_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 会动态地选择并行度, 当然不超过最大允许值, 这是 Oracle 10g 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                              |
|---------------------------------------|-------|--------------------------------------|
| -----                                 |       |                                      |
| <code>_log_parallelism</code>         | 1     | Number of log buffer strands         |
| <code>_log_parallelism_max</code>     | 2     | Maximum number of log buffer strands |
| <code>_log_parallelism_dynamic</code> | TRUE  | Enable dynamic strands               |

在 Oracle 10gR2 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.

#### 4. Oracle 10g 的 Latch 增强

在 Oracle 10g 之前, Latch Free 同 Enqueue 一样, 是一个汇总等待, 从 Oracle 10g 开始, 这个等待被分解:

```
SQL> select name,wait_class
```

```
2 from v$event_name where name like '%latch%';
```

| NAME | WAIT_CLASS |
|------|------------|
|------|------------|

|                             |               |
|-----------------------------|---------------|
| latch: cache buffers chains | Concurrency   |
| latch: redo writing         | Configuration |
| latch: redo copy            | Configuration |
| latch: Undo Hint Latch      | Concurrency   |
| latch: In memory undo latch | Concurrency   |
| latch: MQL Tracking Latch   | Concurrency   |
| latch: row cache objects    | Concurrency   |
| latch: shared pool          | Concurrency   |
| latch: library cache        | Concurrency   |
| latch: library cache lock   | Concurrency   |
| latch: library cache pin    | Concurrency   |

.....

39 rows selected.

最常见的 Latch 集中于 Buffer Cache 的竞争和 Shared Pool 的竞争。与 Buffer Cache 相关的主要 Latch 竞争有 Cache Buffers Chain 和 Cache Buffers LRU Chain, 与 Shared Pool 相关的主要 Latch 竞争有 Shared Pool Latch 和 Library Cache Latch 等。

Buffer Cache 的 Latch 竞争经常是由于热点块竞争引起; Shared Pool 的 Latch 竞争通常是由于 SQL 的大量硬解析引起。这些重要的 Latch 竞争已在第 5 章详细介绍, 此处不再讨论。

Oracle 的等待事件这里不可能一一列举, 本章提供了一些思路和方法, 希望有助于大家认识 Oracle 的等待事件, 并找到一些研究和深入的方法。

## 参考信息与建议阅读

(1) 一个命题：列举你认为最重要的 9 个动态性能视图

[http://www.eygle.com/archives/2005/12/9\\_most\\_important\\_views.html](http://www.eygle.com/archives/2005/12/9_most_important_views.html)

(2) 列举你认为最重要的 9 个动态性能视图

<http://www.itpub.net/471751.html>

(3) My answer for 9 个动态性能视图

[http://www.eygle.com/archives/2005/12/my\\_answer\\_for\\_9\\_views.html](http://www.eygle.com/archives/2005/12/my_answer_for_9_views.html)

---

## 第9章 性能诊断与SQL优化

对于一个数据库系统，从应用的角度来说，用户都期望它有良好的性能，稳定的运行。所以怎样使一个数据库能够高性能稳定运行就变得非常重要，对于大多数数据库维护人员来说，直接面临的挑战就是在出现问题时，需要快速发现并迅速解决数据库性能等问题，并维护系统持续高效运行。

本章将通过一些实际生产中遇到的案例进行剖析讲解，希望大家能够从中领会到诊断性能问题的思路和方法，并对具体问题，特别是SQL问题进行常规处理。

### 9.1 使用 AUTOTRACE 功能辅助 SQL 优化

Oracle SQL\*Plus 提供一个 AUTOTRACE 的功能，可以用于跟踪 SQL 的执行计划，收集统计信息，经常被作为 SQL 的优化工具之一而被广泛使用。

#### 9.1.1 AUTOTRACE 功能的启用

在 Oracle 10g 之前，缺省情况下 AUTOTRACE 功能并未打开，需要通过以下步骤手工启用该功能。

##### 1. 创建基础表

这可以通过运行 \$ORACLE\_HOME\rdbms\admin\utlxplan 脚本完成，该脚本用于创建 plan\_table 表：

```
D:\oracle\ora92>sqlplus /nolog
```

```
SQL*Plus: Release 9.2.0.1.0 - Production on 星期二 6月 3 15:16:03 2003
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
SQL> connect / as sysdba
```

已连接。

```
SQL> @?\rdbms\admin\utlxplan
```

表已创建。

为了使多个用户可以共享同一个 `plan_table`, 可以为它创建一个同义词, 并授权给 `public`:

```
SQL> create public synonym plan_table for plan_table;
```

同义词已创建。

```
SQL> grant all on plan_table to public ;
```

授权成功。

## 2. 创建 `plustrace` 角色

这需要运行 `$ORACLE_HOME\sqlplus\admin\plustrce.sql` 脚本:

```
SQL> @?\sqlplus\admin\plustrce
```

```
SQL>
```

```
SQL> drop role plustrace;
```

```
drop role plustrace
```

```
*
```

```
ERROR 位于第 1 行:
```

```
ORA-01919: 角色'PLUSTRACE'不存在
```

```
SQL> create role plustrace;
```

角色已创建

```
SQL>
```

```
SQL> grant select on v_$sesstat to plustrace;
```

授权成功。

```
SQL> grant select on v_$statname to plustrace;
```

授权成功。

```
SQL> grant select on v_$session to plustrace;
```

授权成功。

```
SQL> grant plustrace to dba with admin option;
```

授权成功。

```
SQL>
```

```
SQL> set echo off
```

### 3. 一点增强

DBA 用户首先被授予了 `plustrace` 角色，然后可以手工把 `plustrace` 授予 `public`，这样所有用户都将拥有 `plustrace` 角色的权限，所有数据库用户也就拥有了使用 `AUTOTRACE` 功能的权限。

```
SQL> grant plustrace to public ;
```

授权成功。

完成以上步骤就可以使用 `AUTOTRACE` 的功能了。

`AUTOTRACE` 有几个常用选项，简单说明如下。

- `SET AUTOTRACE OFF`: 不生成 `AUTOTRACE` 报告，这是缺省模式。
- `SET AUTOTRACE ON EXPLAIN`: `AUTOTRACE` 只显示优化器执行路径报告。
- `SET AUTOTRACE ON STATISTICS`: 只显示执行统计信息。
- `SET AUTOTRACE ON`: 包含执行计划和统计信息。
- `SET AUTOTRACE TRACEONLY`: 同 `SET AUTOTRACE ON`，但是不显示查询输出。

在 `SQL*Plus` 中，`AUTOTRACE` 的基本输出类似以下：

```
SQL> set autotrace on
```

```
SQL> select * from v$version;
```

BANNER

```
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
```

```
PL/SQL Release 9.2.0.4.0 - Production
```

```
CORE      9.2.0.3.0      Production
```

```
TNS for Linux: Version 9.2.0.4.0 - Production
```

```
NLSRTL Version 9.2.0.4.0 - Production
```

Execution Plan

```

0      SELECT STATEMENT Optimizer=CHOOSE
1      0      FIXED TABLE (FULL) OF 'X$VERSION'

```

#### Statistics

```

-----
18   recursive calls
0    db block gets
2    consistent gets
0    physical reads
0    redo size
628  bytes sent via SQL*Net to client
503  bytes received via SQL*Net from client
2    SQL*Net roundtrips to/from client
0    sorts (memory)
0    sorts (disk)
5    rows processed

```

### 9.1.2 Oracle 10g AUTOTRACE 功能的增强

在 Oracle 10g Release 2 中，AUTOTRACE 的功能已经被极大加强和改变。先来看一下什么地方发生了改变：

```

SQL> set autotrace on
SQL> select * from v$version;

```

#### BANNER

```

-----
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Prod
PL/SQL Release 10.2.0.1.0 - Production
CORE      10.2.0.1.0      Production
TNS for Linux: Version 10.2.0.1.0 - Production
NLSRTL Version 10.2.0.1.0 - Production

```

#### Execution Plan

```

-----
Plan hash value: 1078166315
-----

```

| Id  | Operation        | Name       | Rows | Bytes | Cost (%CPU) | Time     |  |
|-----|------------------|------------|------|-------|-------------|----------|--|
| 0   | SELECT STATEMENT |            | 1    | 47    | 0 (0)       | 00:00:01 |  |
| * 1 | FIXED TABLE FULL | X\$VERSION | 1    | 47    | 0 (0)       | 00:00:01 |  |

Predicate Information (identified by operation id):

1 - filter("INST\_ID"=USERENV('INSTANCE'))

Statistics

```

1 recursive calls
0 db block gets
0 consistent gets
0 physical reads
0 redo size
674 bytes sent via SQL*Net to client
385 bytes received via SQL*Net from client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
5 rows processed

```

### 注 意

此时 AUTOTRACE 的输出被良好格式化，并给出关于执行计划部分的简要注释。

其实这里并没有带来新的技术，从 Oracle 9i 开始，Oracle 提供了一个新的工具 `dbms_xplan` 用以格式化和查看 SQL 的执行计划，其原理是通过对 `plan_table` 的查询和格式化提供更友好的用户输出。`dbms_xplan` 的调用的语法类似：

```
select * from table(dbms_xplan.display(format=>'BASIC'))
```

具体用法可以参考 Oracle 官方文档。

实际上，从 Oracle 9i 开始就可以使用如下方式调用 `dbms_xplan`：

```

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning option
JServer Release 9.2.0.4.0 - Production

```

```
SQL> explain plan for
      2  select count(*) from dual;
```

Explained.

```
SQL> @?/rdbms/admin/utlxplp;
```

PLAN\_TABLE\_OUTPUT

```
-----
| Id | Operation                | Name          | Rows  | Bytes | Cost  |
-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT         |               |       |       |       |
|  1 |   SORT AGGREGATE         |               |       |       |       |
|  2 |    TABLE ACCESS FULL    | DUAL          |       |       |       |
-----
```

Note: rule based optimization

10 rows selected.

utlxplp.sql 脚本中正是调用了 dbms\_xplan:

```
SQL> get ?/rdbms/admin/utlxplp;
.....
40* select * from table(dbms_xplan.display());
41
```

而在 Oracle 10gR2 中，Oracle 简化了这个过程，一个 AUTOTRACE 就完成了所有的输出，这是易用性上的一个进步。在使用 Oracle 的过程中，我们经常能够感受到 Oracle 针对用户需求或易用性的改进，这也许是很多人喜爱 Oracle 的一个原因吧。

如果足够细心还会注意到，在 Oracle 10g 中 plan\_table 不再需要创建，Oracle 缺省增加了一个字典表 PLAN\_TABLE\$，然后基于 PLAN\_TABLE\$ 创建公用同义词供用户使用。

使用 AUTOTRACE 功能的另外一个好处就是，用户可以很容易地发现各种视图的底层基础表，这可以作为学习和研究的一个手段：

```
SQL> set autotrace trace explain
SQL> select * from plan_table;
```

## Execution Plan

Plan hash value: 103984305

| Id | Operation         | Name         | Rows | Bytes | Cost (%CPU) | Time     |
|----|-------------------|--------------|------|-------|-------------|----------|
| 0  | SELECT STATEMENT  |              | 1    | 11081 | 2 (0)       | 00:00:01 |
| 1  | TABLE ACCESS FULL | PLAN_TABLE\$ | 1    | 11081 | 2 (0)       | 00:00:01 |

## Note

- dynamic sampling used for this statement

## 9.1.3 AUTOTRACE 功能的内部操作

当使用 AUTOTRACE 功能时，在数据库内部，Oracle 实际上是启动了 2 个 Session 连接，一个 Session 用于执行查询等操作，另外一个 Session 用于记录执行计划和输出最终结果等操作。

下面来进一步深入了解 AUTOTRACE 功能。在启用 AUTOTRACE 之前，注意当前只有一个用户 Session 连接：

```
SQL> select sid,serial#,username from v$session
2 where username is not null;
```

| SID | SERIAL# | USERNAME |
|-----|---------|----------|
| 8   | 5       | SYS      |

在启用 AUTOTRACE 功能后，此时，另外一个 Session 被创建：

```
SQL> set autotrace on
```

```
SQL> /
```

| SID | SERIAL# | USERNAME |
|-----|---------|----------|
| 8   | 5       | SYS      |
| 9   | 14      | SYS      |

## Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE
1      0      FIXED TABLE (FULL) OF 'X$KSUSE'

```

#### Statistics

```

-----
0      recursive calls
0      db block gets
0      consistent gets
0      physical reads
0      redo size
530    bytes sent via SQL*Net to client
503    bytes received via SQL*Net from client
2      SQL*Net roundtrips to/from client
0      sorts (memory)
0      sorts (disk)
2      rows processed

```

而且注意，这两个 Session 都是由一个进程衍生创建的：

```

SQL> select a.sid,a.serial#,a.username,b.pid,b.spid
2   from v$session a ,v$process b
3   where a.PADDR = b.addr and a.username is not null;

```

| SID | SERIAL# | USERNAME | PID | SPID  |
|-----|---------|----------|-----|-------|
| 8   | 5       | SYS      | 9   | 28653 |
| 9   | 14      | SYS      | 9   | 28653 |

#### Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE
1      0      MERGE JOIN
2      1      FIXED TABLE (FULL) OF 'X$KSUPR'
3      1      SORT (JOIN)
4      3      FIXED TABLE (FULL) OF 'X$KSUSE'

```

而此处的 V\$PROCESS.SPID 正是操作系统的进程号：

```
SQL> ! ps -ef|grep 28653|grep -v grep
oracle  28653 28652  0 11:03 ?        00:00:00 oracleeygle (DESCRIPTION=(LOCAL=YES)(ADDRESS=
PROTOCOL=beq)))
```

这就是通常所说的，一个进程在数据库中可能对应多个 Session。

通过在全局启用 10046 事件（具体使用方法可以参考本章后面的内容），可以得到 AUTOTRACE 的内部操作。设置 10046 事件可以采用如下命令（在 spfile 中设置，需要重新启动数据库后方能生效）：

```
alter system set event='10046 trace name context forever,level 12' scope=spfile;
```

通过 tkprof 格式化跟踪文件：

```
[oracle@jumper udump]$ tkprof eygle_ora_28653.trc auto.log aggregate=no
```

```
TKPROF: Release 9.2.0.4.0 - Production on Fri May 12 11:17:54 2006
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
[oracle@jumper udump]$ ll
```

```
total 168
```

```
-rw-r--r-- 1 oracle dba 91729 May 12 11:17 auto.log
```

```
-rw-r----- 1 oracle dba 69254 May 12 11:04 eygle_ora_28653.trc
```

检查跟踪文件或格式化后的日志，可以发现以上两个 Session（在日志中显示为 8.5 和 9.14）的内部操作：

```
*** SESSION ID:(8.5) 2006-05-12 11:03:42.892
```

```
.....
```

```
*** SESSION ID:(9.14) 2006-05-12 11:04:33.935
```

主要的操作步骤如下。

### 1. 执行计划的输出

通过以下 SQL 完成信息记录：

```
insert into plan_table (statement_id, timestamp, operation, options,
object_node, object_owner, object_name, object_instance, object_type,
search_columns, id, parent_id, position, other,optimizer, cost, cardinality,
bytes, other_tag, partition_start, partition_stop, partition_id,
distribution, cpu_cost, io_cost, temp_space, access_predicates,
filter_predicates )
values
(:1,SYSDATE,:2,:3,:4,:5,:6,:7,:8,:9,:10,:11,:12,:13,:14,:15,:16,:17,:18,:19,
:20,:21,:22,:23,:24,:25,:26,:27)
```

通过以下 SQL 完成执行计划查询输出：

```
SELECT ID ID_PLUS_EXP,PARENT_ID PARENT_ID_PLUS_EXP,LPAD(' ',2*(LEVEL-1))
      ||OPERATION||DECODE(OTHER_TAG,NULL,',*')||DECODE(OPTIONS,NULL,', '
      ('||OPTIONS||'))||DECODE(OBJECT_NAME,NULL,', ' OF '||OBJECT_NAME||')
      ||DECODE(OBJECT_TYPE,NULL,', ' (||OBJECT_TYPE||'))||DECODE(ID,0,
      DECODE(OPTIMIZER,NULL,', ' Optimizer='||OPTIMIZER))||DECODE(COST,NULL,', '
      (Cost='||COST||DECODE(CARDINALITY,NULL,', ' Card='||CARDINALITY)
      ||DECODE(BYTES,NULL,', ' Bytes='||BYTES)||')) PLAN_PLUS_EXP,OBJECT_NODE
      OBJECT_NODE_PLUS_EXP
FROM
      PLAN_TABLE START WITH ID=0 AND STATEMENT_ID=:1 CONNECT BY PRIOR ID=PARENT_ID
      AND STATEMENT_ID=:1 ORDER BY ID,POSITION
```

## 2. 统计信息输出

主要通过以下 SQL 获取统计信息名称、编号等信息：

```
SELECT STATISTIC# S, NAME
FROM
      SYS.V_$STATNAME WHERE NAME IN ('recursive calls','db block gets','consistent
      gets','physical reads','redo size','bytes sent via SQL*Net to client',
      'bytes received via SQL*Net from client','SQL*Net roundtrips to/from
      client','sorts (memory)','sorts (disk)') ORDER BY S
```

通过以下查询获得输出值：

```
SELECT PT.VALUE
FROM
      SYS.V_$SESSTAT PT WHERE PT.SID=:1 AND PT.STATISTIC# IN (7,40,41,42,115,236,
      237,238,242,243) ORDER BY PT.STATISTIC#
```

了解了这些内部操作，更有利于学习和理解 Oracle 的运行机制。本书中介绍的很多方法都可以辅助大家进行进一步的深入研究，希望大家在阅读中更多地注意一下方法。

### 9.1.4 使用 AUTOTRACE 功能辅助 SQL 优化

曾经遇到这样一个案例，有朋友在论坛中提出帮助请求，问以下这样一条 SQL 是否可以优化：

```
SELECT *
      FROM sys_user
      WHERE user_code = 'zhangyong'
      OR user_code IN (SELECT grp_code
```

```

FROM sys_grp
WHERE sys_grp.user_code = 'zhangyong')

```

首先可以通过 SQL\*Plus 的 AUTOTRACE 功能查看该 SQL 的执行计划:

```

Execution Plan
-----
0  SELECT STATEMENT Optimizer=RULE
1  0    FILTER
2  1      TABLE ACCESS (FULL) OF 'SYS_USER'
3  1      INDEX (UNIQUE SCAN) OF 'PK_SYS_GRP' (UNIQUE)

Statistics
-----
14  recursive calls
4   db block gets
30590 consistent gets
0   physical reads
.....
0    sorts (memory)
0   sorts (disk)
3   rows processed

```

注意到该 SQL 的逻辑读高达 30590, 优化该 SQL 在根本上需要降低逻辑读。相关数据的记录情况如下:

```
SQL> select count(distinct user_code) from sys_grp;
```

```
COUNT(DISTINCTUSER_CODE)
```

```
-----
14580
```

```
SQL> select count(distinct grp_code) from sys_grp;
```

```
COUNT(DISTINCTGRP_CODE)
```

```
-----
300
```

```
SQL> select count(distinct user_code) from sys_user;
```

```
COUNT(DISTINCTUSER_CODE)
```

-----  
15190

通过执行计划，可以知道，该 SQL 通过全表扫描访问记录数为 15190 的 SYS\_USER 表，通过索引唯一键扫描访问 PK\_SYS\_GRP，两者过滤(FILTER)返回结果集，全表扫描及 FILTER 操作导致了大量的逻辑读。

可以尝试通过 OR 展开、索引访问避免全表扫描和 FILTER 操作，改写后的 SQL 如下：

```
SELECT *
  FROM sys_user
 WHERE user_code = 'zhangyong'
UNION ALL
SELECT *
  FROM sys_user
 WHERE user_code <> 'zhangyong'
    AND user_code IN (SELECT grp_code
                      FROM sys_grp
                      WHERE sys_grp.user_code = 'zhangyong')
```

通过 UNION ALL 将 SQL 展开，从而避免了 FILTER 操作，表联合部分通过 NESTED LOOPS 实现。改写后的 SQL 执行计划如下：

```
Statistics
-----
      0  recursive calls
      0  db block gets
    130 consistent gets
      0  physical reads
.....
      1  sorts (memory)
      0  sorts (disk)
      3  rows processed

Execution Plan
-----
   0      SELECT STATEMENT Optimizer=RULE
   1   0    UNION-ALL
   2   1      TABLE ACCESS (BY INDEX ROWID) OF 'SYS_USER'
   3   2        INDEX (UNIQUE SCAN) OF 'PK_SYS_USER' (UNIQUE)
   4   1      NESTED LOOPS
   5   4        VIEW OF 'VW_NSO_1'
```

|    |   |                                                       |
|----|---|-------------------------------------------------------|
| 6  | 5 | SORT (UNIQUE)                                         |
| 7  | 6 | TABLE ACCESS (BY INDEX ROWID) OF 'SYS_GRP'            |
| 8  | 7 | INDEX (RANGE SCAN) OF 'FK_SYS_USER_CODE' (NON-UNIQUE) |
| 9  | 4 | TABLE ACCESS (BY INDEX ROWID) OF 'SYS_USER'           |
| 10 | 9 | INDEX (UNIQUE SCAN) OF 'PK_SYS_USER' (UNIQUE)         |

注意到 SQL 的逻辑读从原来的 30590 降低到 130，性能得到了极大提高。同时注意到，改写后的 SQL 引入了一个排序，排序来自于这一步：

|   |   |                                                       |
|---|---|-------------------------------------------------------|
| 6 | 5 | SORT (UNIQUE)                                         |
| 7 | 6 | TABLE ACCESS (BY INDEX ROWID) OF 'SYS_GRP'            |
| 8 | 7 | INDEX (RANGE SCAN) OF 'FK_SYS_USER_CODE' (NON-UNIQUE) |

在 SYS\_GRP 表中，user\_code 是非唯一键值，在 in 值判断里，要做 SORT UNIQUE 排序，去除重复值，这里的 UNION ALL 是不需要排序的。

## 9.2 捕获问题 SQL 解决过度 CPU 消耗问题

在生产环境中，可能会经常遇到 CPU 过度使用而影响系统性能或正常运行的问题。大多数情况下，系统的性能问题都是由不良 SQL 代码引起的，那么作为 DBA，怎样发现和解决这些 SQL 问题就显得尤为重要。

本案例的系统环境如下。

- 操作系统：Solaris 8
- 数据库版本：8.1.7.4
- 问题描述：业务及开发人员报告系统运行缓慢，已经影响业务系统正常使用，请求协助诊断。

### 9.2.1 使用 vmstat 检查系统当前情况

首先登录数据库主机，检查当前系统状况。使用 vmstat 检查，发现 CPU 资源已经耗尽，大量任务位于运行队列：

```
bash-2.03$ vmstat 3

procs      memory          page          disk          faults        cpu
r b w    swap  free  re  mf pi po fr de sr s6 s9 s1 sd   in   sy   cs us sy id
0 0 0 5504232 1464112 0 0 0 0 0 0 0 0 0 1 1 0 4294967196 0 0 -84 -5 -145
131 0 0 5368072 1518360 56 691 0 2 2 0 0 0 0 1 0 0 3011 7918 2795 97 3 0
131 0 0 5377328 1522464 81 719 0 2 2 0 0 0 0 1 0 0 2766 8019 2577 96 4 0
130 0 0 5382400 1524776 67 682 0 0 0 0 0 0 0 0 0 0 3570 8534 3316 97 3 0
134 0 0 5373616 1520512 127 1078 0 2 2 0 0 0 1 0 0 0 3838 9584 3623 96 4 0
```

```

136 0 0 5369392 1518496 107 924 0 5 5 0 0 0 0 0 2920 8573 2639 97 3 0
132 0 0 5364912 1516224 63 578 0 0 0 0 0 0 0 0 3358 7944 3119 97 3 0
129 0 0 5358648 1511712 189 1236 0 0 0 0 0 0 0 0 3366 10365 3135 95 5 0
129 0 0 5354528 1511304 120 1194 0 0 0 0 0 0 0 4 0 3235 8864 2911 96 4 0

```

对于 `vmstat` 的用法及输出，这里简要说明一下，`vmstat` 是 UNIX 平台上一个常用的工具，可以帮助用户查看系统内存及 CPU 使用情况。

`vmstat` 最常用的两个参数是 `t[n]`，`t` 表示采样间隔，`n` 表示采样次数。例如，`vmstat 5 5` 表示在 T(5)秒时间内进行 N(5)次采样。

对于前 3 列 `procs` 输出，分别代表以下含义。

- `r`: 指运行队列中的进程数，如果这个参数经常超过 CPU 数量可能说明 CPU 存在瓶颈。

- `b`: IO 被 Block 的进程数。

- `w`: idle 的被 SWAP 的进程数。

最后一项 `cpu` 标识系统 CPU 资源的分配和使用情况，最后一列 `idle` 值通常被用来衡量系统 CPU 的空闲情况。

在本案例中，系统 CPU 资源已经耗尽，`idle` 为 0，并且运行队列大量进程排队等待。

### 9.2.2 使用 Top 工具辅助诊断

通过 `Top` 工具，可以查看进程 CPU 耗用情况，如果存在进程异常，可以通过 `Top` 定位，为进一步诊断提供依据。

对于本案例，观察进程 CPU 耗用，发现没有明显过高 CPU 使用的进程。

```
$ top
```

```

last pid: 28313;  load averages: 99.90, 117.54, 125.71          23:28:38
296 processes: 186 sleeping, 99 running, 2 zombie, 9 on cpu
CPU states:  0.0% idle, 96.5% user,  3.5% kernel,  0.0% iowait,  0.0% swap
Memory: 4096M real, 1404M free, 2185M swap in use, 5114M swap free

```

| PID   | USERNAME | THR | PRI | NICE | SIZE  | RES   | STATE | TIME | CPU   | COMMAND |
|-------|----------|-----|-----|------|-------|-------|-------|------|-------|---------|
| 27082 | oracle8i | 1   | 33  | 0    | 1328M | 1309M | run   | 0:17 | 1.29% | oracle  |
| 26719 | oracle8i | 1   | 55  | 0    | 1327M | 1306M | sleep | 0:29 | 1.11% | oracle  |
| 28103 | oracle8i | 1   | 35  | 0    | 1327M | 1304M | run   | 0:06 | 1.10% | oracle  |
| 28161 | oracle8i | 1   | 25  | 0    | 1327M | 1305M | run   | 0:04 | 1.10% | oracle  |
| 26199 | oracle8i | 1   | 45  | 0    | 1328M | 1309M | run   | 0:42 | 1.10% | oracle  |
| 26892 | oracle8i | 1   | 33  | 0    | 1328M | 1310M | run   | 0:24 | 1.09% | oracle  |
| 27805 | oracle8i | 1   | 45  | 0    | 1327M | 1306M | cpu/1 | 0:10 | 1.04% | oracle  |
| 23800 | oracle8i | 1   | 23  | 0    | 1327M | 1306M | run   | 1:28 | 1.03% | oracle  |

|                |   |    |                   |      |              |
|----------------|---|----|-------------------|------|--------------|
| 25197 oracle8i | 1 | 34 | 0 1328M 1309M run | 0:57 | 1.03% oracle |
| 21593 oracle8i | 1 | 33 | 0 1327M 1306M run | 2:12 | 1.01% oracle |
| 27616 oracle8i | 1 | 45 | 0 1329M 1311M run | 0:14 | 1.01% oracle |
| 27821 oracle8i | 1 | 43 | 0 1327M 1306M run | 0:10 | 1.00% oracle |
| 26517 oracle8i | 1 | 33 | 0 1328M 1309M run | 0:33 | 0.97% oracle |
| 25785 oracle8i | 1 | 44 | 0 1328M 1309M run | 0:46 | 0.96% oracle |
| 26241 oracle8i | 1 | 45 | 0 1327M 1306M run | 0:42 | 0.96% oracle |

从 Top 的输出中可以发现有大量进程处于 **running** 的运行状态，CPU 消耗很平均，单进程消耗大约在 1% 左右，基本可以排除个别进程异常导致 CPU 问题的可能。

关于单进程异常 CPU 消耗问题可以参考第 3 章中的解决方法。

### 9.2.3 检查进程数量

对于一个生产数据库系统，稳定运行的进程数量通常是可知的。

#### 提示

对于稳定运行的生产系统，数据库的运行状况通常是稳定的，如果绘制出性能曲线，会发现每个星期的曲线几乎是重合的，对数据库系统的运行状况及性能指标具有充分认识和了解是必须的。

看一下当前系统的进程数量，从而进行比较判断：

```
bash-2.03$ ps -ef|grep ora|wc -l
258
bash-2.03$ ps -ef|grep ora|wc -l
275
bash-2.03$ ps -ef|grep ora|wc -l
274
bash-2.03$ ps -ef|grep ora|wc -l
278
bash-2.03$ ps -ef|grep ora|wc -l
277
bash-2.03$ ps -ef|grep ora|wc -l
366
```

发现此时系统存在大量 **Oracle** 进程，大约在 300 左右，大量进程消耗了几乎所有 CPU 资源，而正常情况下 **Oracle** 连接数应该在 100 左右。

由此，可以基本判断，是数据库或应用出现了问题，导致进程任务无法完成，不断累积，从而出现大量队列等待。这些等待在数据库中应该有具体的体现，接下来需要登录数据库进行检查。

### 9.2.4 登录数据库

判断数据库可能经历了等待,那么 Oracle 数据库提供了相关视图供用户查询和发现问题, v\$session\_wait 是首先值得关注的。查询 v\$session\_wait 获取各进程等待事件:

```
SQL> select sid,event,p1,p1text from v$session_wait;
```

| SID EVENT                   | P1 P1TEXT          |
|-----------------------------|--------------------|
| -----                       | -----              |
| 124 latch free              | 1.6144E+10 address |
| 1 pmon timer                | 300 duration       |
| 2 rdbms ipc message         | 300 timeout        |
| .....                       |                    |
| 140 buffer busy waits       | 17 file#           |
| 66 buffer busy waits        | 17 file#           |
| 10 db file sequential read  | 17 file#           |
| 18 db file sequential read  | 17 file#           |
| 54 db file sequential read  | 17 file#           |
| 49 db file sequential read  | 17 file#           |
| 48 db file sequential read  | 17 file#           |
| 46 db file sequential read  | 17 file#           |
| 45 db file sequential read  | 17 file#           |
| .....                       |                    |
| 234 db file sequential read | 17 file#           |
| 233 db file sequential read | 17 file#           |
| 230 db file sequential read | 17 file#           |
| 333 db file sequential read | 17 file#           |
| 330 db file scattered read  | 17 file#           |
| 310 db file scattered read  | 17 file#           |
| .....                       |                    |
| 244 rows selected.          |                    |

对于本案例,发现存在大量 db file scattered read 及 db file sequential read 等待。显然全表扫描等操作成为系统最严重的性能影响因素。

### 9.2.5 捕获相关 SQL

确定这些进程因为数据访问产生了等待,考虑捕获这些 SQL 以发现问题。

这里用到了以下脚本 `getsqlbysid.sql`，该脚本通过已知 session 的 `SID`，联合 `v$session`、`v$sqltext` 视图，获得相关 session 正在执行的完整 SQL 语句。

```
SELECT    sql_text
      FROM v$sqltext a
      WHERE a.hash_value = (SELECT sql_hash_value
                             FROM v$session b
                             WHERE b.SID = '&sid')

      ORDER BY piece ASC
/
```

使用该脚本，通过从 `v$session_wait` 中获得的等待全表或索引扫描的进程 `SID`，可以捕获可能存在问题的 SQL 语句：

```
SQL> @getsqlbysid
Enter value for sid: 18

old   5: where b.sid='&sid'
new   5: where b.sid='18'

SQL_TEXT
-----
select i.vc2title,i.numinfoguid  from  hs_info i where i.intenab
ledflag = 1  and i.intpublishstate = 1  and i.datpublishdate <=
sysdate  and i.numcatalogguid = 2047 order by i.datpublishdate d
esc, i.numorder desc

SQL> /
Enter value for sid: 54

old   5: where b.sid='&sid'
new   5: where b.sid='54'

SQL_TEXT
-----
select i.vc2title,i.numinfoguid  from  hs_info i where i.intenab
ledflag = 1  and i.intpublishstate = 1  and i.datpublishdate <=
sysdate  and i.numcatalogguid = 33 order by i.datpublishdate des
c, i.numorder desc

SQL> /
Enter value for sid: 49
```

```
old    5: where b.sid='&sid'
new    5: where b.sid='49'
```

SQL\_TEXT

```
-----
select i.vc2title,i.numinfoguid  from   hs_info i where i.intenab
ledflag = 1   and i.intpublishstate = 1   and i.datpublishdate <=
sysdate   and i.numcatalogguid = 26 order by i.datpublishdate des
c, i.numorder desc
```

对几个进程进行跟踪，分别得到以上 SQL 语句，这些 SQL 可能就是问题产生的根源（以上语句如果良好编码应该使用绑定变量，但是目前这个不是我们关心的）。

使用该应用用户连接，通过 AUTOTRACE 功能检查以上 SQL 的执行计划：

```
SQL> set autotrace trace explain
```

```
SQL> select i.vc2title,i.numinfoguid
      2  from   hs_info i where i.intenabedflag = 1
      3  and i.intpublishstate = 1   and i.datpublishdate <=sysdate
      4  and i.numcatalogguid = 3475
      5  order by i.datpublishdate desc, i.numorder desc  ;
```

Execution Plan

```
-----
      0      SELECT STATEMENT Optimizer=CHOOSE (Cost=228 Card=1 Bytes=106)
      1      0      SORT (ORDER BY) (Cost=228 Card=1 Bytes=106)
      2      1      TABLE ACCESS (FULL) OF 'HS_INFO' (Cost=218 Card=1 Bytes=106)
```

```
SQL> select count(*) from hs_info;
```

```

COUNT(*)
-----
      227404
```

通过执行计划，看到以上查询使用了全表扫描，而该表这里有 22 万记录，全表扫描已经不再适合。

通常对于小表，Oracle 建议通过全表扫描进行数据访问，对于大表则应该通过索引以加快数据查询，当然如果查询要求返回表中大部分或者全部数据，那么全表扫描可能仍然是最好的选择。

从 V\$SYSSTAT 视图中，可以查询得到关于全表扫描的系统统计信息：

```
SQL> col name for a30
```

```
SQL> select name,value from v$sysstat
       2  where name in ('table scans (short tables)','table scans (long tables)');
```

| NAME                       | VALUE |
|----------------------------|-------|
| table scans (short tables) | 828   |
| table scans (long tables)  | 101   |

其中 table scans (short tables) 指对于小表的全表扫描的此时；table scans (long tables) 指对于大表的全表扫描的次数。

从 Statspack 的报告中，也可以找到这部分信息：

```
Instance Activity Stats for DB: CELLSTAR Instance: ora8i Snaps: 20 -
```

| Statistic                  | Total       | per Second | per Trans |
|----------------------------|-------------|------------|-----------|
| table scan blocks gotten   | 38,228,349  | 37.0       | 26.9      |
| table scan rows gotten     | 546,452,583 | 528.9      | 383.8     |
| table scans (direct read)  | 5,784       | 0.0        | 0.0       |
| table scans (long tables)  | 5,990       | 0.0        | 0.0       |
| table scans (rowid ranges) | 5,850       | 0.0        | 0.0       |
| table scans (short tables) | 1,185,275   | 1.2        | 0.8       |

通常，如果一个数据库的 table scans (long tables) 过多，那么 db file scattered read 等待事件可能同样非常显著，与以上数据来自同一个 Report 的 Top5 等待事件就是如此：

```
Top 5 Wait Events
```

| Event                         | Waits          | Time (cs)      | % Total     |
|-------------------------------|----------------|----------------|-------------|
| log file parallel write       | 1,436,993      | 1,102,188      | 10.80       |
| log buffer space              | 16,698         | 873,203        | 8.56        |
| log file sync                 | 1,413,374      | 654,587        | 6.42        |
| control file parallel write   | 329,777        | 510,078        | 5.00        |
| <b>db file scattered read</b> | <b>425,578</b> | <b>132,537</b> | <b>1.30</b> |

在数据库内部，很多信息和现象都是紧密相关的，只要加深对于数据库的了解，在优化和诊断数据库问题时就能够得心应手。

Oracle 通过一个内部参数 `_small_table_threshold` 来定义大表和小表的界限。缺省情况下该参数等于 2% 的 Buffer 数量，如果表的大小小于该参数定义，那么 Oracle 认为该表为小表，否则 Oracle 认为该表为大表。

看一下 Oracle 9iR2 中的情况：

```
SQL> @GetParDescrb.sql
```

```
Enter value for par: small
```

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

```
new 6: AND x.ksppinm LIKE '%small%'
```

| NAME                   | VALUE | DESCRIB                                        |
|------------------------|-------|------------------------------------------------|
| -----                  |       |                                                |
| _small_table_threshold | 200   | threshold level of table size for direct reads |

以上数据库中，200 正好约为 Buffer 数量的 2%：

```
SQL> show parameter db_cache_size
```

| NAME          | TYPE        | VALUE    |
|---------------|-------------|----------|
| -----         |             |          |
| db_cache_size | big integer | 83886080 |

```
SQL> select (83886080/8192)*2/100 from dual;
```

|                       |
|-----------------------|
| (83886080/8192)*2/100 |
| -----                 |
| 204.8                 |

所以要区分大小表 (Long/Short) 是因为全表扫描可能引起 Buffer Cache 的抖动，缺省情况下，大表的全表扫描会被置于 LRU 的末端，以期尽快老化，减少 Buffer 的占用。从 Oracle 3i 开始，Oracle 的多缓冲池管理技术 (Default/Keep/Recycle 池) 给了我们另外一个选择，对于不同大小、不同使用频率的数据表，从建表之初就可以指定其存储 Buffer，以使得内存使用更加有效。

继续以上的案例，在实际处理中，检查全表扫描的数据表，发现存在以下索引：

```
SQL> select index_name,index_type from user_indexes where table_name='HS_INFO';
```

| INDEX_NAME           | INDEX_TYPE            |
|----------------------|-----------------------|
| -----                |                       |
| HSIDX_INFO1          | FUNCTION-BASED NORMAL |
| HSIDX_INFO_SEARCHKEY | DOMAIN                |
| PK_HS_INFO           | NORMAL                |

检查索引键值：

```
SQL> select index_name,column_name
       2 from user_ind_columns where table_name='HS_INFO';
```

| INDEX_NAME           | COLUMN_NAME   |
|----------------------|---------------|
| HSIDX_INFO1          | NUMORDER      |
| HSIDX_INFO1          | SYS_NC00024\$ |
| HSIDX_INFO_SEARCHKEY | VC2INDEXWORDS |
| PK_HS_INFO           | NUMINFOGUID   |

```
SQL> desc hs_info
```

| Name                  | Null?           | Type              |
|-----------------------|-----------------|-------------------|
| NUMINFOGUID           | NOT NULL        | NUMBER(15)        |
| <b>NUMCATALOGGUID</b> | <b>NOT NULL</b> | <b>NUMBER(15)</b> |
| INTTEXTTYPE           | NOT NULL        | NUMBER(38)        |
| VC2TITLE              | NOT NULL        | VARCHAR2(60)      |
| VC2AUTHOR             |                 | VARCHAR2(100)     |
| NUMPREVINFOGUID       |                 | NUMBER(15)        |
| NUMNEXTINFOGUID       |                 | NUMBER(15)        |
| NUMORDER              | NOT NULL        | NUMBER(15)        |
| .....                 |                 |                   |

### 9.2.6 创建新的索引以消除全表扫描

检查发现在 NUMCATALOGGUID 字段上并没有索引，该字段具有很好的区分度，考虑在该字段创建索引以消除全表扫描。

```
SQL> create index hs_info_NUMCATALOGGUID on hs_info(NUMCATALOGGUID);
```

Index created.

```
SQL> set autotrace trace explain
```

```
SQL> select i.vc2title,i.numinfoguid
  2  from   hs_info i where i.intenabledflag = 1
  3  and i.intpublishstate = 1 and i.datpublishdate <=sysdate
  4  and i.numcatalogguid = 3475
  5  order by i.datpublishdate desc, i.numorder desc ;
```

Execution Plan

-----

```

0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=1 Bytes=106)
1    0      SORT (ORDER BY) (Cost=12 Card=1 Bytes=106)
2    1        TABLE ACCESS (BY INDEX ROWID) OF 'HS_INFO' (Cost=2 Card=1 Bytes=106)
3    2          INDEX (RANGE SCAN) OF 'HS_INFO_NUMCATALOGGUID' (NON-UNIQUE)
Cost=1 Card=1)

```

### 9.2.7 观察系统状况

原大量等待消失:

```
SQL> select sid,event,p1,p1text from v$session_wait where event not like 'SQL%';
```

| SID EVENT                   | P1 P1TEXT      |
|-----------------------------|----------------|
| 1 pmon timer                | 300 duration   |
| 2 rdbms ipc message         | 300 timeout    |
| 3 rdbms ipc message         | 300 timeout    |
| 6 rdbms ipc message         | 180000 timeout |
| 59 rdbms ipc message        | 6000 timeout   |
| 118 rdbms ipc message       | 6000 timeout   |
| 275 rdbms ipc message       | 30000 timeout  |
| 147 rdbms ipc message       | 6000 timeout   |
| 62 rdbms ipc message        | 6000 timeout   |
| 11 rdbms ipc message        | 30000 timeout  |
| 4 rdbms ipc message         | 300 timeout    |
| 305 db file sequential read | 17 file#       |
| 356 db file sequential read | 17 file#       |
| 19 db file scattered read   | 17 file#       |
| 5 smon timer                | 300 sleep time |

15 rows selected.

在另外的 session 里，持续观察的 CPU 使用情况:

```

bash-2.03$ vmstat 3
procs      memory          page          disk          faults          cpu
r b w    swap  free  re  mf pi po fr de sr s6 s9 s1 sd  in  sy  cs us sy id
34 0 0 5343016 1465416 44 386 77 0 0 0 0 0 0 0 0 3197 8486 2902 92 8 0
31 0 0 5331568 1459696 178 1491 122 0 0 0 0 0 0 3 0 3237 9461 3005 89 11 0
31 0 0 5317792 1453008 76 719 80 0 0 0 0 0 0 0 0 3292 8736 3025 93 7 0

```

```

31 2 0 5311144 1449552 235 1263 69 2 2 0 0 0 1 0 0 3473 9535 3357 88 12 0
25 0 0 5300240 1443920 108 757 18 2 2 0 0 0 1 1 0 2377 7876 2274 95 5 0
19 0 0 5295904 1441840 50 377 0 0 0 0 0 0 0 1 0 1915 6598 1599 98 1 0

```

----以上为创建索引之前部分

----以下为创建索引之后部分，CPU 使用率恢复正常

| procs |   | memory |         | page    |     |      |    | disk |    |    |    | faults |    |    |    | cpu  |       |      |      |    |    |    |
|-------|---|--------|---------|---------|-----|------|----|------|----|----|----|--------|----|----|----|------|-------|------|------|----|----|----|
| r     | b | w      | swap    | free    | re  | mf   | pi | po   | fr | de | sr | s6     | s9 | s1 | sd | in   | sy    | cs   | us   | sy | id |    |
| 0     | 0 | 0      | 4955872 | 1287136 | 737 | 6258 | 16 | 0    | 0  | 0  | 0  | 0      | 0  | 3  | 0  | 2890 | 11777 | 4432 | 44   | 12 | 44 |    |
| 1     | 0 | 0      | 4887888 | 1256464 | 809 | 6234 | 8  | 2    | 2  | 0  | 0  | 0      | 0  | 2  | 0  | 2809 | 12066 | 4247 | 45   | 12 | 43 |    |
| 0     | 0 | 0      | 4828912 | 1228200 | 312 | 2364 | 13 | 5    | 5  | 0  | 0  | 0      | 2  | 1  | 0  | 2410 | 6816  | 3492 | 38   | 6  | 57 |    |
| 0     | 0 | 0      | 4856816 | 1240168 | 8   | 138  | 0  | 0    | 0  | 0  | 0  | 0      | 0  | 1  | 0  | 0    | 2314  | 4026 | 3232 | 34 | 4  | 62 |
| 0     | 0 | 0      | 4874176 | 1247712 | 0   | 86   | 0  | 0    | 0  | 0  | 0  | 0      | 0  | 0  | 0  | 0    | 2298  | 3930 | 3324 | 35 | 2  | 63 |
| 2     | 0 | 0      | 4926088 | 1270824 | 34  | 560  | 0  | 0    | 0  | 0  | 0  | 0      | 0  | 0  | 0  | 0    | 2192  | 4694 | 2612 | 29 | 16 | 55 |
| 0     | 0 | 0      | 5427320 | 1512952 | 53  | 694  | 0  | 0    | 0  | 0  | 0  | 0      | 0  | 3  | 2  | 0    | 2443  | 5085 | 3340 | 33 | 12 | 55 |
| 0     | 0 | 0      | 5509120 | 1553136 | 0   | 37   | 0  | 0    | 0  | 0  | 0  | 0      | 0  | 0  | 0  | 0    | 2309  | 3908 | 3321 | 35 | 1  | 64 |

至此，此问题得以解决。

### 9.2.8 性能何以提高

回答这个问题似乎是多余的，我只想重申一点：有效地降低 SQL 的逻辑读是 SQL 优化的基本原则之一。

来比较一下前后两种执行方式的逻辑读取及性能差异。

(1) 全表扫描的性能：

```

SQL> select i.vc2title,i.numinfo guid
2   from   hs_info i where i.intenableflag = 1
3   and i.intpublishstate = 1   and i.datpublishdate <=sysdate
4   and i.numcataloguid = 3475
5   order by i.datpublishdate desc, i.numorder desc ;

```

352 rows selected.

Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=541 Card=1 Bytes=106)
1      0      SORT (ORDER BY) (Cost=541 Card=1 Bytes=106)
2      1      TABLE ACCESS (FULL) OF 'HS_INFO' (Cost=531 Card=1 Bytes=106)

```

## Statistics

```

-----
0 recursive calls
25 db block gets
3499 consistent gets
258 physical reads
0 redo size
.....
2 sorts (memory)
0 sorts (disk)
352 rows processed

```

## (2) 使用索引的性能:

```

SQL> select i.vc2title,i.numinfo guid
2 from   hs_info i where i.intenableflag = 1
3 and i.intpublishstate = 1 and i.datpublishdate <=sysdate
4 and i.numcatalogguid = 3475
5 order by i.datpublishdate desc, i.numorder desc;

```

352 rows selected.

## Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=1 Bytes=106)
1    0      SORT (ORDER BY) (Cost=12 Card=1 Bytes=106)
2    1      TABLE ACCESS (BY INDEX ROWID) OF 'HS_INFO' (Cost=2 Card=1 Bytes=106)
3    2      INDEX (RANGE SCAN) OF 'HS_INFO_NUMCATALOGGUID' (NON-UNIQUE)
Cost=1 Card=1)

```

## Statistics

```

-----
0 recursive calls
0 db block gets
89 consistent gets
0 physical reads
0 redo size
.....

```

```

1  sorts (memory)
0  sorts (disk)
352 rows processed

```

consistent gets 从 3499 降到 89，可以看到性能得到了巨大的提高。

### 9.2.9 小结

通常，开发人员很少注意 SQL 代码的效率，他们更着眼于功能的实现。至于性能问题通常被认为是次要的，而且在应用系统开发初期，由于数据库数据量较少，对于查询 SQL 语句等，不容易体现出各种 SQL 句法的性能差异。

但是一旦这些应用作为生产系统上线运行，随着数据库中数据量的增加，大量并发访问，系统的响应速度就可能会成为系统需要解决的最主要问题之一。在少量用户下性能可以接受的 SQL，在大量用户并发的条件下就可能成为性能瓶颈。

在这个案例中，开发人员很难相信仅一条 SQL 语句就导致了整个数据库的性能下降。然而事实就是如此，一条低效的 SQL 语句就可能毁掉整个数据库，所以在系统设计及开发过程中，必须考虑到诸多细节，严格的测试也是提早发现问题的有效方法。

如果不幸以上环节都被忽略，那么，DBA（也许就是你）是最后的一环，就必须能够快速诊断并解决各种复杂问题。

## 9.3 使用 SQL\_TRACE/10046 事件进行数据库诊断

SQL\_TRACE/10046 事件是 Oracle 提供的用于进行 SQL 跟踪的手段，是强有力的辅助诊断工具。在日常的数据库问题诊断和解决中，SQL\_TRACE 是非常常用的方法。当在数据库中启用 SQL\_TRACE 或者设置 10046 事件之后，Oracle 将会启动内核跟踪程序，持续记录会话的相关信息，并写入到相应 trace 文件中。跟踪记录的内容包括 SQL 的解析过程、SQL 的执行计划、绑定变量的使用及会话中发生的等待事件等。

在本章之前，我们多次提到和使用过 SQL\_TRACE/10046 功能，本节就 SQL\_TRACE/10046 事件的使用进行探讨，并通过具体案例对 SQL\_TRACE 的使用进行说明。

### 9.3.1 SQL\_TRACE 及 10046 事件的基础介绍

首先先对 SQL\_TRACE 及 10046 事件进行一些基本介绍，以使大家能对这个工具有所了解，并熟悉其使用方法。

#### 1. SQL\_TRACE 说明

先来关注一下 Oracle 官方文档 (Oracle 9iR2 文档) 对 SQL\_TRACE 的说明，如表 9-1 所示。

SQL\_TRACE 的取值可以启用或禁用 SQL Trace 工具。设置 SQL\_TRACE 为 true 可以收集信息用于性能优化或问题诊断；DBMS\_SYSTEM 包也可以用来实现同样的功能。

表 9-1 SQL\_TRACE 的说明

| 参 数 类 型 | 布 尔 型        |
|---------|--------------|
| 缺省值     | false        |
| 参数类别    | 静态           |
| 取值范围    | true   false |

### 警 告

设置初始化参数 SQL\_TRACE 为 true 会对整个实例产生严重的性能影响,所以在产品环境中如非必要,确保不要设置这个参数。如果只是对特定的 session 启用跟踪,可以使用 ALTER SESSION 或 DBMS\_SYSTEM.SET\_SQL\_TRACE\_IN\_SESSION 来设置。如果必须在数据库级启用 SQL\_TRACE,则需要保证以下条件以最小化性能影响:

- (1) 至少保证有 25% 的 CPU idle;
- (2) 为 USER\_DUMP\_DEST 分配足够的空间;
- (3) 条带化磁盘以减轻 IO 负担。

### 注 意

如果使用 ALTER SESSION SET SQL\_TRACE 来修改 session 级设置,这个设置并不会在 v\$parameter 动态性能视图中体现出来,所以,这个参数仍然被认为是静态参数。

在使用 SQL\_TRACE 之前,几个注意事项需要简单说明一下:

#### ■ 初始化参数 TIMED\_STATISTICS

参数 TIMED\_STATISTICS 最好设置为 True,否则一些重要信息不会被收集。

#### ■ 设置 MAX\_DUMP\_FILE\_SIZE

该参数设置跟踪文件的大小限制,可以以操作系统块为单位设置;也可以以 KB 或 MB 为单位设置;如果跟踪的信息较多,可以干脆设置为 UNLIMITED。从 9i 开始,该参数默认值为 UNLIMITED。

在 session 级可以设置如下:

```
SQL> alter session set MAX_DUMP_FILE_SIZE=unlimited;
```

```
Session altered.
```

记住前面的警告,你需要有足够的空间保存 trace 文件,跟踪过程产生的 trace 文件可能远远大于你的想象。

SQL\_TRACE 可以作为初始化参数或者通过 alert system (从 10g 开始) 在全局启用,也可以通过命令行方式在具体 session 启用。

(1) 在全局启用 SQL\_TRACE。

在参数文件 (pfile/spfile) 中指定:

```
sql_trace = true
```

在全局启用 SQL\_TRACE 会导致所有进程的活动被跟踪,包括后台进程及所有用户进程,这通常会导致比较严重的性能问题,所以在生产环境中要谨慎使用。

## 提示

通过在全局启用 SQL\_TRACE，可以跟踪到所有后台进程的活动，很多在文档中的抽象说明，通过跟踪文件的实时变化，就可以清晰地看到各个进程之间的紧密协调。

(2) 在当前 session 级设置。

大多数时候使用 SQL\_TRACE 来跟踪当前进程。通过跟踪当前进程可以发现当前操作的后台数据库的递归活动（这在研究数据库新特性时尤其有效）、用于研究 SQL 执行及发现后台错误等。

在 session 级启用和停止 SQL\_TRACE 方式如下。

启用当前 session 的跟踪：

```
SQL> alter session set sql_trace=true;
```

Session altered.

此时的 SQL 操作将被跟踪：

```
SQL> select count(*) from dba_users;
```

```

COUNT(*)
-----
        34

```

结束跟踪：

```
SQL> alter session set sql_trace=false;
```

Session altered.

(3) 跟踪其他用户进程。

在很多时候需要跟踪其他用户的进程，而不是当前用户，这可以通过 Oracle 提供的系统包 DBMS\_SYSTEM.SET\_SQL\_TRACE\_IN\_SESSION 来完成。

SET\_SQL\_TRACE\_IN\_SESSION 过程要提供 3 个参数：

```
SQL> desc dbms_system
```

...

```
PROCEDURE SET_SQL_TRACE_IN_SESSION
```

| Argument Name | Type    | In/Out Default? |
|---------------|---------|-----------------|
| SID           | NUMBER  | IN              |
| SERIAL#       | NUMBER  | IN              |
| SQL_TRACE     | BOOLEAN | IN              |
| .....         |         |                 |

通过查询 v\$session 可以获得 SID、SERIAL#等信息。

获得进程信息后，选择需要跟踪的进程，设置跟踪，具体如下：

```
SQL> select sid,serial#,username from v$session
2  where username is not null;
```

| SID | SERIAL# | USERNAME |
|-----|---------|----------|
| 8   | 2041    | SYS      |
| 9   | 437     | EYGLE    |

设置跟踪：

```
SQL> exec dbms_system.set_sql_trace_in_session(9,437,true)
```

PL/SQL procedure successfully completed.

...

可以等候片刻，跟踪 session 执行任务，捕获 SQL 操作...

如果确定某个功能或模块存在问题，可以在此期间有意识的调用以确保可以捕获问题代码

...

停止跟踪：

```
SQL> exec dbms_system.set_sql_trace_in_session(9,437,false)
```

PL/SQL procedure successfully completed.

如果要对其他用户的参数进行设置，可能需要用到 DBMS\_SYSTEM 包中的另外一个过程：

```
SQL> desc dbms_system
```

...

```
PROCEDURE SET_INT_PARAM_IN_SESSION
```

| Argument Name | Type           | In/Out Default? |
|---------------|----------------|-----------------|
| SID           | NUMBER         | IN              |
| SERIAL#       | NUMBER         | IN              |
| PARNAM        | VARCHAR2       | IN              |
| INTVAL        | BINARY_INTEGER | IN              |

...

比如设置 MAX\_DUMP\_FILE\_SIZE 等参数，可以参考如下：

```
SQL> select sid,serial#,username from v$session where username is not null;
```

| SID | SERIAL# USERNAME |
|-----|------------------|
| 18  | 1605 EYGLE       |

SQL> begin

```

2 sys.dbms_system.set_bool_param_in_session(18, 1605, 'timed_statistics', true);
3 sys.dbms_system.set_int_param_in_session(18, 1605, 'max_dump_file_size', 2147483647);
4 sys.dbms_system.set_sql_trace_in_session(18, 1605, true);
5 end;
6 /

```

PL/SQL procedure successfully completed.

DBMS\_SYSTEM 包功能强大，值得仔细研究。

## 2. 10046 事件说明

10046 事件是 Oracle 提供的内部事件，是对 SQL\_TRACE 的增强。10046 事件可以设置以下 4 个级别。

- Level 1: 启用标准的 SQL\_TRACE 功能，等价于 SQL\_TRACE。
- Level 4: 等价于 Level 1+绑定值 (bind values)。
- Level 8: 等价于 Level 1+等待事件跟踪。
- Level 12: 等价于 Level 1+Level 4+Level 8。

类似 SQL\_TRACE，10046 事件可以在全局设置，也可以在全局 session 级设置。

(1) 在全局设置。

在参数文件中增加：

```
event="10046 trace name context forever,level 12"
```

此设置对所有用户的所有进程生效、包括后台进程。

(2) 对当前 session 设置。

通过 alter session 的方式修改，需要 alter session 的系统权限：

```
SQL> alter session set events '10046 trace name context forever';
```

Session altered.

```
SQL> alter session set events '10046 trace name context forever, level 8';
```

Session altered.

```
SQL> alter session set events '10046 trace name context off';
```

Session altered.

(3) 对其他用户 session 设置。

通过 DBMS\_SYSTEM.SET\_EV 系统包来实现：

```
SQL> desc dbms_system
```

...

PROCEDURE SET\_EV

| Argument Name | Type           | In/Out Default? |
|---------------|----------------|-----------------|
| SI            | BINARY_INTEGER | IN              |
| SE            | BINARY_INTEGER | IN              |
| EV            | BINARY_INTEGER | IN              |
| LE            | BINARY_INTEGER | IN              |
| NM            | VARCHAR2       | IN              |
| .....         |                |                 |

其中的参数 SI、SE 来自 v\$session 视图。

查询获得需要跟踪的 session 信息：

```
SQL> select sid,serial#,username from v$session where username is not null;
```

| SID | SERIAL# | USERNAME |
|-----|---------|----------|
| 8   | 2041    | SYS      |
| 9   | 437     | EYGLE    |

执行跟踪：

```
SQL> exec dbms_system.set_ev(9,437,10046,8,'eygle');
```

PL/SQL procedure successfully completed.

结束跟踪：

```
SQL> exec dbms_system.set_ev(9,437,10046,0,'eygle');
```

PL/SQL procedure successfully completed.

### 3. 获取跟踪文件

以上生成的跟踪文件位于 user\_dump\_dest 目录中，位置及文件名可以通过以下 SQL 查询获得：

```
SQL> select
```

```

2      d.value||'/'||lower(rtrim(i.instance, chr(0)))||'_ora_'||p.spid||'.trc' trace_file_name
3  from
4      ( select p.spid
5          from sys.v$mystat m,sys.v$session s,sys.v$process p
6          where m.statistic# = 1 and s.sid = m.sid and p.addr = s.paddr) p,
7      ( select t.instance from sys.v$thread t,sys.v$parameter v
8          where v.name = 'thread' and (v.value = 0 or t.thread# = to_number(v.value))) i,
9      ( select value from sys.v$parameter where name = 'user_dump_dest') d
10 /

```

TRACE\_FILE\_NAME

-----  
/opt/oracle/admin/hsjf/udump/hsjf\_ora\_1026.trc

#### 4. 读取当前 session 设置的参数

当通过 alter session 的方式设置了 SQL\_TRACE, 这个设置是不能通过 show parameter 的方式得到的, 而需要通过 dbms\_system.read\_ev 来获取:

```

SQL> set feedback off
SQL> set serveroutput on

SQL> declare
2      event_level number;
3  begin
4      for event_number in 10000..10999 loop
5          sys.dbms_system.read_ev(event_number, event_level);
6          if (event_level > 0) then
7              sys.dbms_output.put_line(
8                  'Event ' ||
9                  to_char(event_number) ||
10                 ' is set at level ' ||
11                 to_char(event_level)
12             );
13          end if;
14      end loop;
15  end;
16 /

```

Event 10046 is set at level 1

### 9.3.2 诊断案例一：隐式转换与索引失效

下面通过几个案例来看一下 SQL\_TRACE 在数据库诊断及优化过程中的应用。

#### 1. 问题描述

这是帮助一个公司进行优化的诊断案例，应用是一个后台新闻发布系统。前端展现是一个大型网站。Java 开发应用，通过中间件连接池连接数据库。

- 操作系统：Sun OS 5.8。
- 数据库版本：Oracle 8.1.7。
- 问题描述：通过链接访问新闻页极其缓慢，后台发布管理具有同样问题。通常需要十几秒才能返回。这种性能是用户不能忍受的，需要进行优化，找到问题所在。

以下是当时的诊断及问题解决过程，添加了必要的说明，供大家参考。

#### 2. 检查并跟踪数据库进程

由于发布系统是非实时系统，诊断时是晚上，基本无用户访问。我选择在前台单击相关页面，同时进行后台进程跟踪。

查询 v\$session 视图，获取进程信息：

```
SQL> select sid,serial#,username from v$session;
```

| SID | SERIAL# | USERNAME |
|-----|---------|----------|
| 1   | 1       |          |
| 2   | 1       |          |
| 3   | 1       |          |
| 4   | 1       |          |
| 5   | 1       |          |
| 6   | 1       |          |
| 7   | 284     | IFLOW    |
| 11  | 214     | IFLOW    |
| 12  | 164     | SYS      |
| 16  | 1042    | IFLOW    |

10 rows selected.

除了 SYS 及后台进程外，其他 3 个进程是我的诊断目标，我对这几个进程启用相关进程 SQL\_TRACE:

```
SQL> exec dbms_system.set_sql_trace_in_session(7,284,true)
```

```
PL/SQL procedure successfully completed.
```

```
SQL> exec dbms_system.set_sql_trace_in_session(11,214,true)
```

```
PL/SQL procedure successfully completed.
```

```
SQL> exec dbms_system.set_sql_trace_in_session(16,1042,true)
```

```
PL/SQL procedure successfully completed.
```

此时在前台对相关页面进行刷新，等候一段时间，关闭 SQL\_TRACE。

```
SQL> exec dbms_system.set_sql_trace_in_session(7,284,false)
```

```
PL/SQL procedure successfully completed.
```

```
SQL> exec dbms_system.set_sql_trace_in_session(11,214,false)
```

```
PL/SQL procedure successfully completed.
```

```
SQL> exec dbms_system.set_sql_trace_in_session(16,1042,false)
```

```
PL/SQL procedure successfully completed.
```

### 3. 检查 trace 文件

在 user\_dump\_dest 目录下，可以找到生成的跟踪文件，然后通过 Oracle 提供的格式化工具 tkprof 对 trace 文件进行格式化处理，检查发现以下语句是可疑的：

```
*****
```

```
select auditstatus,categoryid,auditlevel
from
categoryarticleassign a,category b where b.id=a.categoryid and articleId=
20030700400141 and auditstatus>0
```

| call    | count | cpu  | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse   | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0    |
| Execute | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0    |

|       |   |      |      |   |      |   |   |
|-------|---|------|------|---|------|---|---|
| Fetch | 1 | 0.81 | 0.81 | 0 | 3892 | 0 | 1 |
| ----- |   |      |      |   |      |   |   |
| total | 3 | 0.81 | 0.81 | 0 | 3892 | 0 | 1 |

这里显然是根据 `articleId` 进行新闻内容读取的，`auditstatus>0` 应该是限制只能读取审查过的内容。注意这里很可疑的是 `query` 读取有 3892，按 8KB 的 `block_size` 来计算的话，这大约是 30MB 的逻辑读取。

这个内容引起了我的注意。如果遇到过类似的问题，大家在这里就应该可以大致猜到问题的原因了。如果没有遇到过的朋友，可以在这里思考一下再往下看。

这是 `trace` 文件里的另外一段：

\*\*\*\*\*

```
select auditstatus,categoryid
from
categoryarticleassign where articleId=20030700400138 and categoryId in ('63',
'138','139','140','141','142','143','144','168','213','292','341','346',
'347','348','349','350','351','352','353','354','355','356','357','358',
'359','360','361','362','363','364','365','366','367','368','369','370',
'371','372','383','460','461','462','463','621','622','626','629','631',
'634','636','643','802','837','838','849','850','851','852','853','854',
'858','859','860','861','862','863','-1')
```

| call    | count | cpu  | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| -----   |       |      |         |      |       |         |      |
| Parse   | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0    |
| Execute | 1     | 0.00 | 0.00    | 0    | 0     | 0       | 0    |
| Fetch   | 1     | 4.91 | 4.91    | 0    | 2835  | 7       | 1    |
| -----   |       |      |         |      |       |         |      |
| total   | 3     | 4.91 | 4.91    | 0    | 2835  | 7       | 1    |

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 41

Rows      Row Source Operation

```
-----
1  TABLE ACCESS FULL CATEGORYARTICLEASSIGN
```

\*\*\*\*\*

注意到，这里有一个全表扫描存在，对 categoryarticleassign 表的访问，最终是通过全表扫描完成的。

#### 4. 登录数据库检查相应索引及表结构

登录数据库，获得索引及表结构信息：

```
SQL> select index_name,table_name,column_name from user_ind_columns
      2  where table_name=upper('categoryarticleassign');
```

| INDEX_NAME               | TABLE_NAME                   | COLUMN_NAME      |
|--------------------------|------------------------------|------------------|
| <b>IDX_ARTICLEID</b>     | <b>CATEGORYARTICLEASSIGN</b> | <b>ARTICLEID</b> |
| IND_ARTICLEID_CATEG      | CATEGORYARTICLEASSIGN        | ARTICLEID        |
| IND_ARTICLEID_CATEG      | CATEGORYARTICLEASSIGN        | CATEGORYID       |
| IDX_SORTID               | CATEGORYARTICLEASSIGN        | SORTID           |
| PK_CATEGORYARTICLEASSIGN | CATEGORYARTICLEASSIGN        | ARTICLEID        |
| PK_CATEGORYARTICLEASSIGN | CATEGORYARTICLEASSIGN        | CATEGORYID       |
| PK_CATEGORYARTICLEASSIGN | CATEGORYARTICLEASSIGN        | ASSIGNTYPE       |
| IDX_CAT_ARTICLE          | CATEGORYARTICLEASSIGN        | AUDITSTATUS      |
| IDX_CAT_ARTICLE          | CATEGORYARTICLEASSIGN        | ARTICLEID        |
| IDX_CAT_ARTICLE          | CATEGORYARTICLEASSIGN        | CATEGORYID       |
| IDX_CAT_ARTICLE          | CATEGORYARTICLEASSIGN        | ASSIGNTYPE       |

11 rows selected.

注意到该表上 ARTICLEID 字段建有 IDX\_ARTICLEID 索引，而该索引在以上查询中都没有被用到，接下来检查表结构：

```
SQL> desc categoryarticleassign
```

| Name        | Null?    | Type          |
|-------------|----------|---------------|
| CATEGORYID  | NOT NULL | NUMBER        |
| ARTICLEID   | NOT NULL | VARCHAR2(14)  |
| ASSIGNTYPE  | NOT NULL | VARCHAR2(1)   |
| AUDITSTATUS | NOT NULL | NUMBER        |
| SORTID      | NOT NULL | NUMBER        |
| UNPASS      |          | VARCHAR2(255) |

这里 ARTICLEID 是个字符型 (VARCHAR2) 数据，而在查询中给入的条件是：

```
articleId= 20030700400141
```

在这个查询中，20030700400141 被认为是一个数字值。Oracle 在执行这个 SQL 时发生了潜在的数据类型转换（把 ARTICLEID 转换为 Number 和 20030700400141 进行比较），从而导致了索引失效。

在 SQL\*Plus 中执行类似查询：

```
SQL> select auditstatus,categoryid
2   from
3   categoryarticleassign where articleId=20030700400132;

AUDITSTATUS CATEGORYID
-----
          9           94
          0          383
          0          695

Elapsed: 00:00:02.62

Execution Plan
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=110 Card=2 Bytes=38)
1    0  TABLE ACCESS (FULL) OF 'CATEGORYARTICLEASSIGN' (Cost=110 Card=2 Bytes=38)
```

发现执行的是全表扫描，索引被忽略，这显然不是我们想看到的。

## 5. 解决方法

解决这个问题是简单的，在参数两侧各添加一个单引号 (')，即可解决这个问题。对于用单引号引起来的数字，Oracle 会认为是字符串，这样就消除了隐式类型转换，索引得以被正确使用。

对于类似的查询，可以发现索引被正确使用，Query 模式读取降低为 2，执行该 SQL 几乎不需要花费 CPU 时间了。

```
*****

select unpass
from
categoryarticleassign where articleid='20030320000682' and categoryid='113'

call      count      cpu      elapsed      disk      query      current      rows
-----
-----
```

|                                         |                                                   |      |      |   |   |   |   |
|-----------------------------------------|---------------------------------------------------|------|------|---|---|---|---|
| Parse                                   | 1                                                 | 0.00 | 0.00 | 0 | 0 | 0 | 0 |
| Execute                                 | 1                                                 | 0.00 | 0.00 | 0 | 0 | 0 | 0 |
| Fetch                                   | 1                                                 | 0.00 | 0.00 | 0 | 2 | 0 | 0 |
| -----                                   |                                                   |      |      |   |   |   |   |
| total                                   | 3                                                 | 0.00 | 0.00 | 0 | 2 | 0 | 0 |
| Misses in library cache during parse: 1 |                                                   |      |      |   |   |   |   |
| Optimizer goal: CHOOSE                  |                                                   |      |      |   |   |   |   |
| Parsing user id: 20                     |                                                   |      |      |   |   |   |   |
| Rows                                    | Row Source Operation                              |      |      |   |   |   |   |
| -----                                   |                                                   |      |      |   |   |   |   |
| 0                                       | TABLE ACCESS BY INDEX ROWID CATEGORYARTICLEASSIGN |      |      |   |   |   |   |
| 1                                       | INDEX RANGE SCAN (object id 3080)                 |      |      |   |   |   |   |
| *****                                   |                                                   |      |      |   |   |   |   |

至此，这个问题得到了完满的解决。

## 6. 小结

在 Oracle 开发中，应该尽量避免使用隐式的数据类型转换，因为隐式数据类型转换可能会带来索引失效的问题，给系统埋下隐患。

这些问题在开发阶段就应该被避免。使用显式的数据类型转换应该被作为规则确定下来。使用函数导致索引失效的问题与此类似。

在很多系统中可以看到，大量的性能问题都是由细小的疏忽所致，而且由于问题具有一定的隐蔽性等而不易被发现和排查，所以这些问题一旦爆发出来，就会给诊断和优化带来相当的难度，所以完善的规范和良好的编码对于一个系统来说是至关重要的。

### 9.3.3 诊断案例二：跟踪后台错误

#### 1. 问题描述

很多时候，在进行数据库操作时，如 drop user、drop table 等，经常会遇到这样的错误：

```
ORA-00604: error occurred at recursive SQL level 1 .
```

单从这样的提示来看，很多时候是没有丝毫用处的，也无法确定问题出在何处。本案例就这一类问题提供了一个思路及方法，供大家参考。

#### 2. drop user 出现问题

这是一个生产环境的数据库，在 drop user 时报出以下错误后退出：

```
ORA-00604: error occurred at recursive SQL level 1
```

```
ORA-00942: table or view does not exist .
```

关于 recursive SQL 错误，这里有必要做个简单的说明。

当发出一条简单的 SQL 命令以后，Oracle 数据库要在后台解析这条命令，并转换为 Oracle 数据库的一系列后台操作。这些后台操作统称为递归 SQL。

比如 create table 这样一条简单的 DDL 命令，Oracle 数据库在后台，实际上要把这个命令转换为对 obj\$、tab\$、col\$ 等底层表的插入操作；对于 drop table 操作，则是在这些系统表中进行反向删除操作，大家同样可以通过 SQL\_TRACE 进行后台跟踪，进一步了解 Oracle 数据库的后台操作。

Oracle 所做的工作有时可能比我们想象的要复杂得多。

### 3. 跟踪问题

Oracle 提供 SQL\_TRACE 的功能，可以用于跟踪 Oracle 数据库的后台递归操作。

通过跟踪文件，可以找到问题的所在，以下是跟踪过程：

```
SQL> alter session set sql_trace=true;
```

```
Session altered.
```

```
SQL> drop user wapcomm;
```

```
ORA-00604: error occurred at recursive SQL level 1
```

```
ORA-00942: table or view does not exist .
```

```
SQL> alter session set sql_trace=false;
```

格式化（使用 tkprof 工具）跟踪文件后，获得以下输出（摘录部分）：

```
*****
The following statement encountered a error during parse:
DELETE FROM SDO_GEOM_METADATA_TABLE WHERE SDO_OWNER = 'WAPCOMM'
Error encountered: ORA-00942
*****

alter session set sql_trace=true

.....

drop user wapcomm

.....
```

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: SYS

\*\*\*\*\*

.....(省略部分递归 SQL...)

\*\*\*\*\*

delete from user\_history\$

where

user# = :1                    -----后台的递归删除操作...

.....

Rows        Row Source Operation

-----

1    DELETE USER\_HISTORY\$

1    TABLE ACCESS FULL USER\_HISTORY\$

\*\*\*\*\*

declare

  stmt varchar2(200);

BEGIN

  if dictionary\_obj\_type = 'USER' THEN

    stmt := 'DELETE FROM SDO\_GEOM\_METADATA\_TABLE ' ||

      ' WHERE SDO\_OWNER = ''' || dictionary\_obj\_name || ''';

    EXECUTE IMMEDIATE stmt;

  end if;

end;

.....

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 31        (recursive depth: 1)

\*\*\*\*\*

```
alter session set sql_trace=false
.....
```

```
Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: SYS
```

使用 tkprof 格式化以后，Oracle 把错误信息首先呈现出来。可以看到 ORA-00942 错误是由 SDO\_GEOM\_METADATA\_TABLE 表/视图不存在所致，问题由此可以定位。

对于这一类的错误，在定位问题之后，解决的方法就要根据问题的具体原因而定了。

#### 4. 问题定位

对于本案例，通过 Metalink (<http://metalink.oracle.com>) 可容易获得以下解释：

##### Problem Description

-----

The Oracle Spatial Option has been installed and you are encountering the following errors while trying to drop a user, who has no spatial tables, connected as SYSTEM:

ERROR at line 1:

ORA-00604: error occurred at recursive SQL level 1

ORA-00942: table or view does not exist

ORA-06512: at line 7

A 942 error trace shows the failing SQL statement as:

```
DELETE FROM SDO_GEOM_METADATA_TABLE WHERE SDO_OWNER = '<user>'
```

##### Solution Description

-----

(1) Create a synonym for SDO\_GEOM\_METADATA\_TABLE under SYSTEM which points to MDSYS.SDO\_GEOM\_METADATA\_TABLE.

(2) Now the user can be dropped connected as SYSTEM.

对于本例，为 MDSYS.SDO\_GEOM\_METADATA\_TABLE 创建一个同义词即可解决，是相对简单的情况。MDSYS.SDO\_GEOM\_METADATA\_TABLE 是 Spatial 对象，如果未使用 Spatial 选项，可以删除。

```
SQL> connect / as sysdbaConnected.
```

```

SQL> select * from dba_sdo_geom_metadata order by owner;

select * from dba_sdo_geom_metadata order by owner
*

ERROR at line 1:
ORA-00942: table or view does not exist
ORA-04063: view "MDSYS.DBA_SDO_GEOM_METADATA" has errors


SQL> select object_name from dba_objects where object_name like '%SDO%';

OBJECT_NAME
-----
ALL_SDO_GEOM_METADATA
ALL_SDO_INDEX_INFO
ALL_SDO_INDEX_METADATA
DBA_SDO_GEOM_METADATA
DBA_SDO_INDEX_INFO
DBA_SDO_INDEX_METADATA
....
DBA_SDO_GEOM_METADATA
DBA_SDO_INDEX_INFO
...
SDO_WITHIN_DISTANCE
USER_SDO_GEOM_METADATA
USER_SDO_INDEX_INFO
USER_SDO_INDEX_METADATA

88 rows selected.


SQL> drop user MDSYS cascade;

User dropped.


SQL> select owner,type_name from dba_types where type_name like 'SDO%';

no rows selected


SQL>

```

```

SQL> alter session set sql_trace=true;

Session altered.

SQL> drop user wapcomm;

User dropped.

SQL> alter session set sql_trace=false;

Session altered.

SQL> exit
Disconnected from Oracle8i Enterprise Edition Release 8.1.7.4.0 - 64bit Production
With the Partitioning option
JServer Release 8.1.7.4.0 - 64bit Production

```

这时用户得以顺利 drop。

## 5. 小结

使用 SQL\_TRACE 可以跟踪数据库的很多后台操作，有利于发现问题的根本所在。

很多时候，想要研究 Oracle 的内部活动或后台操作，也可以通过 SQL\_TRACE 跟踪。这是深入学习学习 Oracle 的必经之途。

### 9.3.4 10046 与等待事件

如果需要获得更多的跟踪信息，就需要用到 10046 事件。如上所述，10046 事件是 SQL\_TRACE 功能的增强，可以通过 10046 跟踪获得更多的信息，包括非常有用的等待事件等。

在进行数据库问题诊断及性能优化时，经常需要查询的几个重要视图包括 v\$session\_wait、v\$system\_event 等，这些视图中主要记录的就是等待事件。

通过调整以降低等待，是提高性能的一个方法。这些等待事件来自所有数据库操作，对于不同进程的等待可以通过动态性能视图 v\$session\_wait 等来查询；对于数据库全局等待可以通过 v\$system\_event 等视图来获得。同样地，可以通过对具体 session 的跟踪获得每个 session 的执行情况及等待事件。

#### 1. 10046 事件的使用

下面通过一个简单的测试，来看一下 10046 事件的强大之处。

```
SQL> create table t as select * from dba_objects;
```

Table created.

--创建一张测试表

SQL> select file\_id,block\_id,blocks from dba\_extents where segment\_name='T';

| FILE_ID | BLOCK_ID | BLOCKS |
|---------|----------|--------|
| 1       | 21601    | 8      |
| 1       | 21609    | 8      |
| 1       | 21617    | 8      |
| 1       | 21625    | 8      |
| 1       | 21633    | 8      |
| 1       | 23433    | 8      |
| 1       | 23441    | 8      |
| 1       | 23449    | 8      |
| 1       | 23457    | 8      |
| 1       | 23465    | 8      |

10 rows selected.

--查看其空间使用情况

SQL> alter session set events '10046 trace name context forever,level 12';

Session altered.

--启用 10046 事件跟踪

SQL> select count(\*) from t;

| COUNT(*) |
|----------|
| 6207     |

--由于表上未建立索引,所以此处应该引发一次全表扫描

```
SQL> alter session set events '10046 trace name context off';
```

```
Session altered.
```

```
--停用跟踪
```

然后来检查一下 Oracle 生成的跟踪文件:

```
SQL> !
```

```
[oracle@eygle udump]$ ls
```

```
rac1_ora_20695.trc
```

```
[oracle@eygle udump]$ cat rac1_ora_20695.trc |grep scatt
```

```
WAIT #1: nam='db file scattered read' ela= 11657 p1=1 p2=21602 p3=7
```

```
WAIT #1: nam='db file scattered read' ela= 1363 p1=1 p2=21609 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1297 p1=1 p2=21617 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1346 p1=1 p2=21625 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1313 p1=1 p2=21633 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 6226 p1=1 p2=23433 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1316 p1=1 p2=23441 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1355 p1=1 p2=23449 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 1320 p1=1 p2=23457 p3=8
```

```
WAIT #1: nam='db file scattered read' ela= 884 p1=1 p2=23465 p3=5
```

注意这里的等待事件“db file scattered read”，这意味着这里使用了全表扫描来访问数据。其中 p1、p2、p3 分别代表了文件号、起始数据块块号、读取数据块的数量。各参数的含义也可以从 v\$event\_name 视图中获得:

```
SQL> select name,PARAMETER1 p1,PARAMETER2 p2,PARAMETER3 p3
```

```
2 from v$event_name where name='db file scattered read';
```

| NAME                   | P1    | P2     | P3     |
|------------------------|-------|--------|--------|
| db file scattered read | file# | block# | blocks |

在数据库内部，这些等待时间最后都会累计到 v\$system\_event 动态性能视图中，是数据库性能诊断的一个重要参考:

```
SQL> select event,time_waited from v$system_event
```

```
2 where event='db file scattered read';
```

| EVENT                  | TIME_WAITED |
|------------------------|-------------|
| db file scattered read | 51          |

## 2. 10046 与 db\_file\_multiblock\_read\_count

这里有必要提到另外一个相关的初始化参数 `db_file_multiblock_read_count`，这个参数代表 Oracle 在执行全表扫描时每次 IO 操作可以读取的数据块的数量。

在前面的测试中，我的 `db_file_multiblock_read_count` 参数设置为 16，由于 extent 大小为 3 个 block，Oracle 的一次 IO 操作不能跨越 extent，所以前面的全表扫描每次只能读取 8 个 block，进行了 10 次 IO 读取。

来看一下进一步的测试：

```
SQL> create tablespace eygle
```

```
2 datafile '/dev/raw/raw2' size 100M
```

```
3 extent management local uniform size 256K;
```

Tablespace created.

```
SQL> alter table t move tablespace eygle;
```

Table altered.

```
SQL> select file_id,block_id,blocks from dba_extents where segment_name='T';
```

| FILE_ID | BLOCK_ID | BLOCKS |
|---------|----------|--------|
| 4       | 9        | 32     |
| 4       | 41       | 32     |
| 4       | 73       | 32     |

```
SQL> show parameter read_count
```

| NAME                          | TYPE    | VALUE |
|-------------------------------|---------|-------|
| db_file_multiblock_read_count | integer | 16    |

```
SQL> alter session set events '10046 trace name context forever,level 12';
```

Session altered.

```
SQL> select count(*) from t;
```

```

COUNT(*)
-----
6207

SQL> alter session set events '10046 trace name context off';

Session altered.

SQL> select
  2      d.value||'/'||lower(rtrim(i.instance, chr(0)))||'_ora_'||p.spid||'.trc' trace_file_name
  3  from
  4      ( select p.spid
  5          from sys.v$mystat m,sys.v$session s,sys.v$process p
  6          where m.statistic# = 1 and s.sid = m.sid and p.addr = s.paddr) p,
  7      ( select t.instance from sys.v$thread t,sys.v$parameter v
  8          where v.name = 'thread' and (v.value = 0 or t.thread# = to_number(v.value))) i,
  9      ( select value from sys.v$parameter where name = 'user_dump_dest') d
 10  /

TRACE_FILE_NAME
-----
/opt/oracle/admin/rac/udump/rac1_oracle_20912.trc

SQL> !
[oracle@eygle rac]$ cat /opt/oracle/admin/rac/udump/rac1_oracle_20912.trc |grep scatt
WAIT #1: nam='db file scattered read' ela= 12170 p1=4 p2=10 p3=16
WAIT #1: nam='db file scattered read' ela= 2316 p1=4 p2=26 p3=15
WAIT #1: nam='db file scattered read' ela= 2454 p1=4 p2=41 p3=16
WAIT #1: nam='db file scattered read' ela= 2449 p1=4 p2=57 p3=16
WAIT #1: nam='db file scattered read' ela= 2027 p1=4 p2=73 p3=13

```

看到此时，Oracle 只需要 5 次 IO 操作就完成了全表扫描。

通常较大的 `db_file_multiblock_read_count` 设置可以加快全表扫描的执行，但是根据经验大于 32 的设置通常不会带来更大的性能提高。

### 3. 10046 与执行计划的选择

需要注意的是，增大 `db_file_multiblock_read_count` 参数的设置，会使全表扫描的成本降低，在 CBO 优化器下可能会使 Oracle 更倾向于使用全表扫描而不是索引访问。

看一下进一步的测试：

```
SQL> desc t
```

| Name           | Null? | Type          |
|----------------|-------|---------------|
| OWNER          |       | VARCHAR2(30)  |
| OBJECT_NAME    |       | VARCHAR2(128) |
| SUBOBJECT_NAME |       | VARCHAR2(30)  |
| OBJECT_ID      |       | NUMBER        |
| DATA_OBJECT_ID |       | NUMBER        |
| OBJECT_TYPE    |       | VARCHAR2(18)  |
| CREATED        |       | DATE          |
| LAST_DDL_TIME  |       | DATE          |
| TIMESTAMP      |       | VARCHAR2(19)  |
| STATUS         |       | VARCHAR2(7)   |
| TEMPORARY      |       | VARCHAR2(1)   |
| GENERATED      |       | VARCHAR2(1)   |
| SECONDARY      |       | VARCHAR2(1)   |

```
SQL> select owner,count(*) from t group by owner;
```

| OWNER  | COUNT(*) |
|--------|----------|
| OUTLN  | 7        |
| PUBLIC | 1623     |
| SYS    | 4042     |
| SYSTEM | 404      |
| WMSYS  | 131      |

```
SQL> create index i_owner on t(owner);
```

Index created.

```
SQL> analyze table t compute statistics for table
```

- 2        for all indexes
- 3        for all indexed columns;

Table analyzed.

```
SQL> set autotrace traceonly explain
```

```
SQL> alter session set db_file_multiblock_read_count=16;
```

```
Session altered.
```

```
SQL> select * from t where owner='SYSTEM';
```

```
Execution Plan
```

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=7 Card=404 Bytes=34744)
1    0    TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=7 Card=404 Bytes=34744)
2    1      INDEX (RANGE SCAN) OF 'I_OWNER' (NON-UNIQUE) (Cost=1 Card=404)
```

```
SQL> alter session set db_file_multiblock_read_count=32;
```

```
Session altered.
```

```
SQL> select * from t where owner='SYSTEM';
```

```
Execution Plan
```

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=6 Card=404 Bytes=34744)
1    0    TABLE ACCESS (FULL) OF 'T' (Cost=6 Card=404 Bytes=34744)
```

可以注意到，当增大 `db_file_multiblock_read_count` 参数时，全表扫描的成本降低，Oracle 在后面的执行计划中选择了全表扫描。所以，当修改这个初始化参数时，必须认识到，很多 SQL 的执行计划可能由此改变。

#### 4. `db_file_multiblock_read_count` 与系统的 IO 能力

`db_file_multiblock_read_count` 的设置受到 OS 最大 IO 能力的影响，也就是说，如果系统的硬件 I/O 能力有限，即使设置再大的 `db_file_multiblock_read_count` 也是没有用的。理论上，最大 `db_file_multiblock_read_count` 和系统 IO 能力应该有如下关系：

$$\text{Max}(\text{db\_file\_multiblock\_read\_count}) = \text{MaxOsIOSize} / \text{db\_block\_size}$$

当然这个 `Max(db_file_multiblock_read_count)` 还受 Oracle 的限制。`SSTIOMAX` 是 Oracle 的内部参数或常数，用以限制单次 IO 读写操作的最大数据传输量。这个参数是固定的，不能被修改。所以 `db_file_multiblock_read_count` 参数的设置就会受到 `SSTIOMAX`

的限制：

```
db_block_size * db_file_multiblock_read_count <= SSTIOMAX
```

另外一个限制是最大 `db_file_multiblock_read_count` 值不能超过 `db_block_buffers/4`。

也可以通过 `db_file_multiblock_read_count` 来测试 Oracle 在不同系统下，单次 IO 最大所能读取得数据量：

```
$ sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 10.1.0.2.0 - Production on Wed Aug 11 23:43:52 2004
```

```
Copyright (c) 1982, 2004, Oracle. All rights reserved.
```

```
Connected to:
```

```
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - 64bit Production
```

```
With the Partitioning, OLAP and Data Mining options
```

```
SYS AS SYSDBA on 11-AUG-04 >show parameter read_count
```

| NAME                          | TYPE    | VALUE |
|-------------------------------|---------|-------|
| db_file_multiblock_read_count | integer | 16    |

```
SYS AS SYSDBA on 11-AUG-04 >create tablespace dfmbrc
```

```
2 datafile '/opt/oracle/oradata/eygle/dfmbrc.dbf'
```

```
3 size 20M extent management local uniform size 2M;
```

```
Tablespace created.
```

```
SYS AS SYSDBA on 11-AUG-04 >create table t tablespace dfmbrc as select * from dba_objects;
```

```
Table created.
```

```
SYS AS SYSDBA on 11-AUG-04 >insert into t select * from t;
```

```
9149 rows created.
```

```
SYS AS SYSDBA on 11-AUG-04 >/
```

```
18298 rows created.
```

SYS AS SYSDBA on 11-AUG-04 >/

36596 rows created.

SYS AS SYSDBA on 11-AUG-04 >commit;

Commit complete.

SYS AS SYSDBA on 11-AUG-04 >alter session set db\_file\_multiblock\_read\_count=1000;

Session altered.

SYS AS SYSDBA on 12-AUG-04 >show parameter read\_count

| NAME                          | TYPE    | VALUE |
|-------------------------------|---------|-------|
| db_file_multiblock_read_count | integer | 128   |

SYS AS SYSDBA on 11-AUG-04 >alter session set events '10046 trace name context forever,level 12';

Session altered.

SYS AS SYSDBA on 11-AUG-04 >alter system flush buffer\_cache;

System altered.

SYS AS SYSDBA on 11-AUG-04 >select count(\*) from t;

| COUNT(*) |
|----------|
| 73192    |

SYS AS SYSDBA on 12-AUG-04 >@gettrace

| TRACE_FILE_NAME |
|-----------------|
|-----------------|

```
/opt/oracle/soft/eygle_ora_24432.trc
```

```
$ cat /opt/oracle/soft/eygle_ora_24432.trc|grep sca
```

```
WAIT #26: nam='db file scattered read' ela= 18267 p1=10 p2=10 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 8836 p1=10 p2=138 p3=127
```

```
WAIT #26: nam='db file scattered read' ela= 8923 p1=10 p2=265 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 8853 p1=10 p2=393 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 8985 p1=10 p2=521 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 8997 p1=10 p2=649 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 9096 p1=10 p2=777 p3=128
```

```
WAIT #26: nam='db file scattered read' ela= 583 p1=10 p2=905 p3=12
```

```
$
```

可以看到,在以上测试平台中,Oracle 最多每次 IO 能够读取 128 个 Block,由于 block\_size 为 8KB,也就是每次最多读取了 1MB 数据。

系统平台如下:

```
$ uname -a
```

```
SunOS billing 5.8 Generic_108528-23 sun4u sparc SUNW,Ultra-4
```

这里以介绍 10046 事件为主,不再过多讨论 db\_file\_multiblock\_read\_count 的内容。

## 5. 小结

SQL\_TRACE/10046 事件是 Oracle 提供的非常强大的工具,应该深入了解、掌握并且熟练运用它。希望本文能够帮助大家了解这个事件,而怎样使用它去解决问题,深入了解 Oracle,还有待大家进一步的探索。

## 9.4 使用物化视图进行翻页性能调整

物化视图从 Oracle 8i 被引入到数据库中,最初被作为数据仓库/决策支持系统的工具,是概要管理的一部分。物化试图通过预计算或汇总构建自己的独立存储,从而可以极大地提高相关处理的性能,通过查询重写(Query Rewrite)功能,Oracle 可以自动对 SQL 进行改写以最大程度地发挥物化视图的作用。

物化视图是典型的通过存储空间换取性能的方式,通过物化视图,Oracle 可以:

- 有效地减少逻辑读取。
- 减少写操作——通过消除排序及聚集实现。
- 减少 CPU 的消耗——无需实时进行复杂运算。
- 显著提高相应速度。

这些是物化视图的主要特点。本节试图通过一个具体案例的应用,说明物化视图在优化

中的应用。

#### 9.4.1 系统环境

操作系统为 Linux AD2.1，数据库版本如下：

```
SQL> select * from v$version;

BANNER
-----
Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
PL/SQL Release 9.2.0.4.0 - Production
CORE      9.2.0.3.0      Production
TNS for Linux: Version 9.2.0.4.0 - Production
NLSRTL Version 9.2.0.4.0 - Production
```

#### 9.4.2 问题描述

数据库系统出现如下错误：

```
Mon Dec  6 16:51:44 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:51:55 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:52:51 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:52:52 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:52:54 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:52:55 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
Mon Dec  6 16:52:56 2004
ORA-1652: unable to extend temp segment by 128 in tablespace      TEMP2
```

临时表空间不能扩展，前台应用出现异常，研发人员请求协助，开始介入进行诊断。

#### 9.4.3 捕获排序 SQL 语句

首先尝试捕获引发排序的 SQL 语句，使用以下代码 getsort.sql:

```

SELECT          /*+ rule */
      DISTINCT a.SID, a.process, a.serial#,
      TO_CHAR (a.logon_time, 'YYYYMMDD HH24:MI:SS') LOGON, a.osuser,
      TABLESPACE, b.sql_text
      FROM v$session a, v$sql b, v$sort_usage c
      WHERE a.sql_address = b.address(+) AND a.sql_address = c.sqladdr
/

```

获得以下主要排序语句：

```

SQL> @getsort

          SID  PROCESS                SERIAL#  LOGON                OSUSER
TABLESPACE
-----
SQL_TEXT
-----

          15                24965 20041207 16:38:01 oracle                TEMP2
select count(1) from HW_User4Love u where u.numIntention<>99 and u.numGender=:1

          21                49757 20041207 16:33:28 oracle                TEMP2
select * from (select t.*, rownum i from (select u.numUserId,u.vc2UserName,u.numUserType,u.numRank,
u.numGender,u.numAge,
u.numDistrict,u.vc2District,u.numLooking,u.numPersonality,u.numAbility,u.numZodiac,u.numExperience
from HW_User4Love u where u.numIntention<>99 order by u.numUserType desc, u.numRank desc, u.numUserId
desc) t where rownum<=260) where i>240

          30                65414 20041207 16:34:04 oracle                TEMP2
select * from (select t.*, rownum i from (select u.numUserId,u.vc2UserName,u.numUserType,u.numRank,
u.numGender,u.numAge,
u.numDistrict,u.vc2District,u.numLooking,u.numPersonality,u.numAbility,u.numZodiac,u.numExperience
from HW_User4Love u where u.numIntention<>99 order by u.numUserType desc, u.numRank desc, u.numUserId
desc) t where rownum<=40) where i>20

```

大量类似的 SQL 在占用大量的排序空间，确定是这些 SQL 引起的临时表空间过量使用。这些 SQL 需要研究和优化。

#### 9.4.4 确定典型问题 SQL

主要问题 SQL，显然是一个翻页查询程序：

```

SELECT *
FROM (SELECT t.*, ROWNUM i
      FROM (SELECT u.numuserid, u.vc2username, u.numusertype, u.numrank,
                  u.numgender, u.numage, u.numdistrict, u.vc2district,
                  u.numlooking, u.numpersonality, u.numability,
                  u.numzodiac, u.numexperience
               FROM hw_user4love u
               WHERE u.numintention <> 99 AND u.numgender = :1
               ORDER BY u.numusertype DESC, u.numrank DESC, u.numuserid DESC) t
      WHERE ROWNUM <= 40)
WHERE i > 20

```

查看该 SQL 语句的执行计划:

```
SQL> @hawa
```

20 rows selected.

Elapsed: 00:01:23.92

Execution Plan

```

-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=31692 Card=40 Bytes=7680)
1    0      VIEW (Cost=31692 Card=40 Bytes=7680)
2    1        COUNT (STOPKEY)
3    2          VIEW (Cost=31692 Card=582622 Bytes=104289338)
4    3            SORT (ORDER BY STOPKEY) (Cost=31692 Card=582622 Bytes=30296344)
5    4              HASH JOIN (Cost=7435 Card=582622 Bytes=30296344)
6    5                HASH JOIN (Cost=4991 Card=583296 Bytes=20415360)
7    6                  TABLE ACCESS (FULL) OF 'HW_USERPROFILE' (Cost=1752
Card=583296 Bytes=11082624)
8    6                    TABLE ACCESS (FULL) OF 'HW_USER' (Cost=1799 Card=1380038
Bytes=22080608)
9    5                      TABLE ACCESS (FULL) OF 'HW_USERSCORE' (Cost=506 Card=1332871
Bytes=22658807)

```

Statistics

```

0 recursive calls
255 db block gets
44760 consistent gets
70299 physical reads
0 redo size
2246 bytes sent via SQL*Net to client
514 bytes received via SQL*Net from client
3 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
20 rows processed

```

发现该 SQL 调用了 3 个底层表，执行全表扫描，逻辑读高达 44760。这里 HW\_User4Love 实际上是一个视图，查询其创建语句：

```
SQL> select text from dba_views where view_name=upper('HW_User4Love');
```

```
TEXT
```

```
-----
select
```

```

    u.numUserId as numUserId,
    u.vc2UserName as vc2UserName,
    u.numUserType as numUserType,
    u.vc2MobileNumber as vc2MobileNumber,
    u.numMobileStatus as numMobileStatus,
    u.vc2Mail as vc2Mail,
    u.numMailStatus as numMailStatus,
    s.numRank as numRank,
    s.numExperience as numExperience,
    p.numGender as numGender,
    p.datBirthday as datBirthday,
    p.numAge,
    p.numZodiac,
    p.numDistrict as numDistrict,
    p.vc2District as vc2District,
    p.numHeight as numHeight,
    p.numSmoke as numSmoke,
    p.numDrink as numDrink,

```

```

    p.chrInterests as chrInterests,
    p.numStatus as numIntention,
    s.numLooking as numLooking,
    s.numPersonality as numPersonality,
    s.numAbility as numAbility
from
    HW_User u,
    HW_UserProfile p,
    HW_UserScore s
where
    u.numUserId = p.numUserId and u.numUserId = s.numUserId and s.numExperience > 100
with read only

Elapsed: 00:00:00.64
SQL>

```

而 3 个底层表都有大量记录：

```

SQL> select count(*) from hw_user;

COUNT(*)
-----
1378484

SQL> select count(*) from hw_userprofile;

COUNT(*)
-----
1378470

SQL> select count(*) from hw_userscore;

COUNT(*)
-----
1378498

```

这 3 个表的全表扫描对数据库的性能产生了巨大的冲击。进一步确认发现，在这个结果集中符合视图查询条件的记录只有少量：

```
SQL> select count(*) from hw_user4love;
```

```
COUNT(*)
```

```
-----
```

```
234975
```

显然每次通过视图全表扫描 3 个底层表是产生性能问题的主要原因。

#### 9.4.5 选择解决办法

结合业务逻辑，考虑创建物化视图，通过物化视图的中间存储消除不必要的全表扫描。创建物化视图如下：

```
CREATE MATERIALIZED VIEW HW_User4Love
    BUILD IMMEDIATE
    REFRESH COMPLETE START WITH SYSDATE
    NEXT trunc(sysdate+1) +4/24
    ENABLE QUERY REWRITE
    AS
select
    u.numUserId as numUserId,
    u.vc2UserName as vc2UserName,
    u.numUserType as numUserType,
    u.vc2MobileNumber as vc2MobileNumber,
    u.numMobileStatus as numMobileStatus,
    u.vc2Mail as vc2Mail,
    u.numMailStatus as numMailStatus,
    s.numRank as numRank,
    s.numExperience as numExperience,
    p.numGender as numGender,
    p.datBirthday as datBirthday,
    p.numAge,
    p.numZodiac,
    p.numDistrict as numDistrict,
    p.vc2District as vc2District,
    p.numHeight as numHeight,
    p.numSmoke as numSmoke,
    p.numDrink as numDrink,
    p.chrInterests as chrInterests,
```

```

        p.numStatus as numIntention,
        s.numLooking as numLooking,
        s.numPersonality as numPersonality,
        s.numAbility as numAbility
from
    HW_User u,
    HW_UserProfile p,
    HW_UserScore s
where
    u.numUserId = p.numUserId and u.numUserId = s.numUserId and s.numExperience > 100

```

### 注 意

必须充分考虑用户的业务需求是否允许足够的刷新闻隔，我定义的是每日刷新一次，在凌晨 4 点进行一次完全刷新。

然后再来看类似查询：

```

SQL> @hawa

20 rows selected.

Elapsed: 00:00:01.03

Execution Plan
-----
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2381 Card=40 Bytes=7680)
   1   0      VIEW (Cost=2381 Card=40 Bytes=7680)
   2   1          COUNT (STOPKEY)
   3   2          VIEW (Cost=2381 Card=102802 Bytes=18401558)
   4   3              SORT (ORDER BY STOPKEY) (Cost=2381 Card=102802 Bytes=4523288)
   5   4                  TABLE ACCESS (FULL) OF 'HW_USER4LOVE' (Cost=337 Card=102802
Bytes=4523288)

Statistics
-----
          0  recursive calls
          0  db block gets
       3446  consistent gets
       2048  physical reads

```

```

0 redo size
2246 bytes sent via SQL*Net to client
514 bytes received via SQL*Net from client
3 SQL*Net roundtrips to/from client
1 sorts (memory)
0 sorts (disk)
20 rows processed

```

注意到，现在这个查询在 1 秒左右即可执行完毕，逻辑读从原来的 44760 减少到现在的 3446，性能大大提高。

### 提 示

有效地降低逻辑读是 SQL 优化的基本原则之一。

#### 9.4.6 进一步的调整优化

注意到这里仍然对 HW\_USER4LOVE 物化视图执行了全表扫描，可以通过对其创建索引来进一步优化。

在原 SQL 语句中包含：

```
order by u.numUserType desc, u.numRank desc, u.numUserId desc
```

正是这个 order by 子句导致了排序，可以通过创建降序索引消除这个排序：

```
SQL> create index idx_desc on HW_USER4LOVE (numUserType desc, numRank desc, numUserId desc);
```

Index created.

Elapsed: 00:00:04.48

### 注 意

降序索引本质上是基于函数的索引，只有在 CBO 下才能被用到。

```
Connected to Oracle9i Enterprise Edition Release 9.2.0.4.0
```

```
Connected as hawa
```

```
SQL> create table t as select * from dba_users;
```

Table created

```
SQL> create index idx_username_desc on t(username desc);
```

Index created

```
SQL> select index_name,table_name,INDEX_TYPE from user_indexes where table_name='T';
```

| INDEX_NAME | TABLE_NAME | INDEX_TYPE |
|------------|------------|------------|
| -----      | -----      | -----      |

```

IDX_USERNAME_DESC          T          FUNCTION-BASED NORMAL
SQL> select column_name,column_position,descend from user_ind_columns
      2  where table_name='T';
COLUMN_NAME      COLUMN_POSITION DESCEND
-----
SYS_NC00013$          1          DESC
    
```

### 注 意

降序索引以及 FBI 的定义可以从 DBA\_IND\_EXPRESSIONS 或 USER\_IND\_EXPRESSIONS 视图中获得。

```
SQL> select * from user_ind_expressions where table_name='T';
```

```

INDEX_NAME  TABLE_NAME COLUMN_EXPRESSION      COLUMN_POSITION
-----
IDX_USERNAME_DESC  T          "USERNAME"          1
    
```

再次执行原查询语句：

```
SQL> @hawa
```

20 rows selected.

Elapsed: 00:00:00.22

#### Execution Plan

```

-----
      0      SELECT STATEMENT Optimizer=CHOOSE (Cost=826 Card=40 Bytes=7680)
      1      0      VIEW (Cost=826 Card=40 Bytes=7680)
      2      1      COUNT (STOPKEY)
      3      2      VIEW (Cost=826 Card=102802 Bytes=18401558)
      4      3      TABLE ACCESS (BY INDEX ROWID) OF 'HW_USER4LOVE' (Cost=826
Card=102802 Bytes=4523288)
      5      4      INDEX (FULL SCAN) OF 'IDX_DESC' (NON-UNIQUE) (Cost=26 Card=234975)
    
```

#### Statistics

```

-----
      0  recursive calls
      0  db block gets
    
```

```

88 consistent gets
    2 physical reads
    0 redo size
2246 bytes sent via SQL*Net to client
514 bytes received via SQL*Net from client
    3 SQL*Net roundtrips to/from client
    0 sorts (memory)
    0 sorts (disk)
    20 rows processed

```

现在 Oracle 通过 Index Full Scan 来执行以上查询，逻辑读下降到了 88，而且排序被彻底消除了。问题至此算是有了一个较好的解决。

#### 9.4.7 小结

Oracle 的物化视图功能强大，使用起来也有很多需要注意的地方，最重要的是，用户的业务逻辑是否允许。所以怎样使用它，还取决于用户对 Oracle 以及自己业务的了解。

关于物化视图的查询重写功能，Thomas Kyte 在他的书中有详细的描述，在笔者的站点上（[www.eygle.com](http://www.eygle.com)）上也有详细描述，这里不再赘述。

### 9.5 一次横跨两岸的问题诊断

最近，通过邮件的方式协助一位朋友解决了一个性能问题，其过程和判断很有参考价值，简单收录在这里，供大家参考，值得高兴的是，网络是没有边界的。

#### 9.5.1 第一封求助邮件

收到这位朋友的第一封邮件时，他是这样描述的。

拜读您在网络上所写 Oracle 诊断案例——SGA 与 Swap 之二，相信您一定可以协助我解决所碰到的问题。我负责的一个系统出现了很高的 iowait，仅知应该是 Oracle 造成。

只要 Oracle 开启，就会造成 iowait 升高，而且执行速度非常缓慢，另外，这是今天才发生的，之前也曾发生一次，但是把 SGA 调小后就恢复正常。但是隔了一个多月，现在发生的问题却再怎么调整还是无法恢复原有的效能。Top 结果如下：

```

last pid: 20888; load averages:  2.13,  2.50,  2.50
349 processes: 346 sleeping, 1 zombie, 2 on cpu
CPU states: 53.7% idle, 18.6% user, 10.7% kernel, 16.9% iowait,  0.0% swap
Memory: 64G real, 46G free, 10G swap in use, 51G swap free

```

| PID | USERNAME | THR | PRI | NICE | SIZE | RES | STATE | TIME | CPU | COMMAND |
|-----|----------|-----|-----|------|------|-----|-------|------|-----|---------|
|-----|----------|-----|-----|------|------|-----|-------|------|-----|---------|

|             |    |    |   |       |              |      |       |            |
|-------------|----|----|---|-------|--------------|------|-------|------------|
| 3589 snims  | 9  | 59 | 0 | 47M   | 36M sleep    | 0:23 | 0.69% | PmTom      |
| 2569 oracle | 1  | 59 | 0 | 8599M | 8557M sleep  | 1:04 | 0.61% | oracle     |
| 3444 snims  | 1  | 50 | 0 | 1520K | 1248K sleep  | 3:08 | 0.57% | logControl |
| 2571 oracle | 1  | 56 | 0 | 8599M | 8557M sleep  | 1:04 | 0.55% | oracle     |
| 2573 oracle | 1  | 59 | 0 | 8599M | 8557M sleep  | 1:00 | 0.55% | oracle     |
| 2567 oracle | 1  | 56 | 0 | 8599M | 8557M sleep  | 1:09 | 0.52% | oracle     |
| 10907 snims | 6  | 59 | 0 | 27M   | 16M sleep    | 0:12 | 0.47% | PmGW       |
| 3599 snims  | 1  | 59 | 0 | 8599M | 8560M sleep  | 0:59 | 0.45% | oracle     |
| 26045 snims | 1  | 59 | 0 | 2920K | 1840K cpu/18 | 0:23 | 0.32% | top        |
| 13836 snims | 1  | 59 | 0 | 2952K | 1856K sleep  | 0:46 | 0.31% | top        |
| 1094 oracle | 15 | 59 | 0 | 8605M | 8554M sleep  | 1:49 | 0.27% | oracle     |
| 3443 snims  | 7  | 59 | 0 | 33M   | 21M sleep    | 1:17 | 0.25% | AlarmGW    |
| 3361 snims  | 1  | 59 | 0 | 8599M | 8559M sleep  | 9:18 | 0.17% | oracle     |
| 7440 snims  | 1  | 59 | 0 | 2352K | 1864K sleep  | 0:05 | 0.16% | iPT        |

透过 sar 可以看到 Oracle 所在的硬盘 busy 是 99%。

```
->sar -d 2 3
SunOS tps2snims11 5.8 Generic_117350-05 sun4u 04/10/06
15:56:02 device %busy avque r+w/s blks/s avwait avserv
15:56:04 md0 10 0.1 20 327 0.0 5.0
          ssd13 99 1.9 108 1501 0.0 18.0
          ssd13,a 0 0.0 0 0 0.0 0.0
          ssd13,b 0 0.0 0 0 0.0 0.0
          ssd13,c 0 0.0 0 0 0.0 0.0
          ssd13,d 99 1.9 108 1501 0.0 18.0
          ssd13,e 0 0.0 0 0 0.0 0.0
```

而 SGA 显示结果如下：

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
JServer Release 9.2.0.4.0 - Production

SQL> show sga;

Total System Global Area 8936995304 bytes
Fixed Size 743912 bytes
Variable Size 2097152000 bytes
Database Buffers 6828326912 bytes
```

|              |                |
|--------------|----------------|
| Redo Buffers | 10772480 bytes |
|--------------|----------------|

alert.log 会一直出现如下错误信息:

```

Creating archive destination LOG_ARCHIVE_DEST_1: '/opt/oracle/oradata/arc/430714142_1_7965.arc'
Mon Apr 10 09:36:46 2006
ARC0: Completed archiving log 4 thread 1 sequence 7965
Mon Apr 10 09:40:29 2006
Thread 1 cannot allocate new log, sequence 7967
Checkpoint not complete
Current log# 5 seq# 7966 mem# 0: /opt/oracle/oradata/SNIMS/redo05a.log
Current log# 5 seq# 7966 mem# 1: /opt/oradata/redo05b.log
Mon Apr 10 09:44:30 2006
Thread 1 advanced to log sequence 7967
Current log# 2 seq# 7967 mem# 0: /opt/oracle/oradata/SNIMS/redo02a.log
Current log# 2 seq# 7967 mem# 1: /opt/oradata/redo02b.log
Mon Apr 10 09:44:30 2006
ARC1: Evaluating archive log 5 thread 1 sequence 7966
ARC1: Beginning to archive log 5 thread 1 sequence 7966
Creating archive destination LOG_ARCHIVE_DEST_2: 'SNIMS_TPS1SNIMS11'
Creating archive destination LOG_ARCHIVE_DEST_1: '/opt/oracle/oradata/arc/430714142_1_7966.arc'
Mon Apr 10 09:44:51 2006
ARC1: Completed archiving log 5 thread 1 sequence 7966
Mon Apr 10 09:45:46 2006
Thread 1 cannot allocate new log, sequence 7968
Checkpoint not complete
Current log# 2 seq# 7967 mem# 0: /opt/oracle/oradata/SNIMS/redo02a.log
Current log# 2 seq# 7967 mem# 1: /opt/oradata/redo02b.log
Mon Apr 10 09:50:26 2006
Thread 1 advanced to log sequence 7968
Current log# 3 seq# 7968 mem# 0: /opt/oracle/oradata/SNIMS/redo03a.log
Current log# 3 seq# 7968 mem# 1: /opt/oradata/redo03b.log
Mon Apr 10 09:50:26 2006
ARC0: Evaluating archive log 2 thread 1 sequence 7967
ARC0: Beginning to archive log 2 thread 1 sequence 7967
Creating archive destination LOG_ARCHIVE_DEST_2: 'SNIMS_TPS1SNIMS11'
Creating archive destination LOG_ARCHIVE_DEST_1: '/opt/oracle/oradata/arc/430714142_1_7967.arc'
Mon Apr 10 09:50:45 2006

```

```

ARC0: Completed archiving   log 2 thread 1 sequence 7967
Mon Apr 10 09:51:38 2006
Thread 1 cannot allocate new log, sequence 7969
Checkpoint not complete
  Current log# 3 seq# 7968 mem# 0: /opt/oracle/oradata/SNIMS/redo03a.log
  Current log# 3 seq# 7968 mem# 1: /opt/oradata/redo03b.log
Mon Apr 10 09:59:13 2006
Thread 1 advanced to log sequence 7969
  Current log# 1 seq# 7969 mem# 0: /opt/oracle/oradata/SNIMS/redo01a.log
  Current log# 1 seq# 7969 mem# 1: /opt/oradata/redo01b.log
Mon Apr 10 09:59:13 2006
ARC1: Evaluating archive    log 3 thread 1 sequence 7968
ARC1: Beginning to archive log 3 thread 1 sequence 7968
Creating archive destination LOG_ARCHIVE_DEST_2: 'SNIMS_TPS1SNIMS11'
Creating archive destination LOG_ARCHIVE_DEST_1: '/opt/oracle/oradata/arc/430714142_1_7968.arc'
Mon Apr 10 09:59:32 2006
ARC1: Completed archiving   log 3 thread 1 sequence 7968
Mon Apr 10 10:00:28 2006
Thread 1 cannot allocate new log, sequence 7970
Checkpoint not complete
  Current log# 1 seq# 7969 mem# 0: /opt/oracle/oradata/SNIMS/redo01a.log
  Current log# 1 seq# 7969 mem# 1: /opt/oradata/redo01b.log

```

而 v\$log 信息如下:

```

SQL> select * from v$log;

```

| GROUP# | THREAD# | SEQUENCE# | BYTES     | MEMBERS | ARC | STATUS   | FIRST_CHANGE# | FIRST_TIM |
|--------|---------|-----------|-----------|---------|-----|----------|---------------|-----------|
| 1      | 1       | 8004      | 52428800  | 2       | YES | INACTIVE | 541861070     | 10-APR-06 |
| 2      | 1       | 8002      | 52428800  | 2       | YES | INACTIVE | 541764248     | 10-APR-06 |
| 3      | 1       | 8003      | 52428800  | 2       | YES | INACTIVE | 541766356     | 10-APR-06 |
| 4      | 1       | 8005      | 104857600 | 2       | YES | INACTIVE | 541897189     | 10-APR-06 |

```

5                                1                8006    104857600                2 NO    CURRENT
541923086 10-APR-06

```

同时/etc/system 相关信息如下:

```

forceload: misc/obpsym
set nopanicdebug = 1
set TS:ts_sleep_promote=1
set shmsys:shminfo_shmmax=10737418240
set shmsys:shminfo_shmmin=100
set shmsys:shminfo_shmmni=100
set shmsys:shminfo_shmseg=100
set semsys:seminfo_semmni=4096
set semsys:seminfo_semmsl=256
set semsys:seminfo_semmns=4096
set semsys:seminfo_semopm=100
set semsys:seminfo_sevmx=32767

```

综合以上状态, 不知道您是否能够了解我们的问题?

从这封邮件的信息, 可以做出以下一些判断。

(1) Checkpoint not complete 提示。说明系统的 I/O 写出存在问题, 这可能是由于数据库过于繁忙, 生成日志过多, 也有可能是因为存储的 I/O 能力存在问题。

(2) sar 的输出。数据文件存放路径 99% 的 I/O Wait, 进一步说明 I/O 负荷沉重, 系统经历 I/O 等待。

(3) v\$log 输出来自不同时段。在 v\$log 输出中, 多数日志组处于 INACTIVE 状态, 所以与 alert 文件显然来自不同时段。

### 9.5.2 第一次回复

我这样回复他。

从 v\$log 情况来看, 你的 Checkpoint not complete 问题应该是间发的, 并非频繁。能否提供一段连续的 iostat 采样信息, 让我看一下 io 状况。

另外, 根据你的状况, 极有可能是应用存在问题。在问题出现时, 请查询 v\$session\_wait 信息, 给我参考。

### 9.5.3 进一步信息提供

网友回信, 提供了 iostat 信息:

```

->iostat -xtcz 2 10

                                extended device statistics                                tty                cpu
device      r/s    w/s    kr/s    kw/s wait actv   svc_t   %w   %b   tin tout   us sy wt id

```

|                            |     |      |      |       |      |      |       |    |    |     |      |     |    |    |    |
|----------------------------|-----|------|------|-------|------|------|-------|----|----|-----|------|-----|----|----|----|
| md0                        | 0.0 | 45.4 | 0.0  | 363.3 | 0.0  | 0.3  | 6.1   | 0  | 28 | 3   | 1023 | 5   | 5  | 17 | 72 |
| md10                       | 0.0 | 45.4 | 0.0  | 363.3 | 0.0  | 0.3  | 5.5   | 0  | 25 |     |      |     |    |    |    |
| md20                       | 0.0 | 45.9 | 0.0  | 367.3 | 0.0  | 0.2  | 3.6   | 0  | 17 |     |      |     |    |    |    |
| sd0                        | 0.0 | 45.4 | 0.0  | 363.3 | 0.0  | 0.2  | 5.5   | 0  | 25 |     |      |     |    |    |    |
| sd1                        | 0.0 | 45.9 | 0.0  | 367.3 | 0.0  | 0.2  | 3.6   | 0  | 17 |     |      |     |    |    |    |
| ssd13                      | 0.0 | 83.3 | 0.0  | 669.9 | 0.0  | 1.2  | 14.1  | 0  | 99 |     |      |     |    |    |    |
| extended device statistics |     |      |      |       |      |      |       |    |    | tty |      | cpu |    |    |    |
| device                     | r/s | w/s  | kr/s | kw/s  | wait | actv | svc_t | %w | %b | tin | tout | us  | sy | wt | id |
| md0                        | 0.0 | 51.0 | 0.0  | 408.0 | 0.0  | 0.3  | 6.0   | 0  | 31 | 3   | 692  | 3   | 9  | 16 | 71 |
| md10                       | 0.0 | 51.0 | 0.0  | 408.0 | 0.0  | 0.2  | 4.7   | 0  | 24 |     |      |     |    |    |    |
| md20                       | 0.0 | 50.5 | 0.0  | 404.0 | 0.0  | 0.2  | 4.1   | 0  | 21 |     |      |     |    |    |    |
| sd0                        | 0.0 | 51.0 | 0.0  | 408.0 | 0.0  | 0.2  | 4.7   | 0  | 24 |     |      |     |    |    |    |
| sd1                        | 0.0 | 50.5 | 0.0  | 404.0 | 0.0  | 0.2  | 4.1   | 0  | 20 |     |      |     |    |    |    |
| ssd13                      | 0.0 | 81.0 | 0.0  | 647.5 | 0.0  | 1.1  | 13.1  | 0  | 99 |     |      |     |    |    |    |
| extended device statistics |     |      |      |       |      |      |       |    |    | tty |      | cpu |    |    |    |
| device                     | r/s | w/s  | kr/s | kw/s  | wait | actv | svc_t | %w | %b | tin | tout | us  | sy | wt | id |
| md0                        | 0.0 | 1.0  | 0.0  | 8.0   | 0.0  | 0.0  | 13.5  | 0  | 1  | 3   | 735  | 4   | 2  | 13 | 82 |
| md10                       | 0.0 | 1.0  | 0.0  | 8.0   | 0.0  | 0.0  | 13.3  | 0  | 1  |     |      |     |    |    |    |
| md20                       | 0.0 | 1.0  | 0.0  | 8.0   | 0.0  | 0.0  | 10.7  | 0  | 1  |     |      |     |    |    |    |
| sd0                        | 0.0 | 1.5  | 0.0  | 8.2   | 0.0  | 0.0  | 12.4  | 0  | 2  |     |      |     |    |    |    |
| sd1                        | 0.0 | 1.5  | 0.0  | 8.2   | 0.0  | 0.0  | 10.6  | 0  | 2  |     |      |     |    |    |    |
| ssd13                      | 0.0 | 90.0 | 0.0  | 767.2 | 0.0  | 1.1  | 12.7  | 0  | 99 |     |      |     |    |    |    |

在这个输出中，注意到在磁盘 I/O 繁忙度为 99% 时，写出能力却仅有 767.2k/s 左右，这个 I/O 能力是很低的。

I/O 能力低下是造成系统性能问题的主要原因。而 I/O 能力低下又可能有几方面的原因，第一可能是硬件本身的限制；第二可能是硬件异常导致 I/O 能力受限。

还有 v\$session\_wait 查询输出，摘录部分内容：

|                                     |              |
|-------------------------------------|--------------|
| SQL> select * from v\$session_wait; |              |
| SID                                 | SEQ# EVENT   |
| -----                               | -----        |
| 31                                  | 4634 enqueue |
| 44                                  | 4578 enqueue |
| 206                                 | 206 enqueue  |
| 204                                 | 5631 enqueue |
| 195                                 | 195 enqueue  |
| 141                                 | 209 enqueue  |

```

129      180 enqueue
125      175 enqueue
59      4571 enqueue
25      460 buffer busy waits
151     107 buffer busy waits
191    1217 buffer busy waits
140    3321 buffer busy waits
127     439 buffer busy waits
126     763 buffer busy waits
82     502 buffer busy waits
39     274 log file switch (checkpoint incomplete)
100    4972 log file switch (checkpoint incomplete)
197      64 log file switch (checkpoint incomplete)
187   41946 log file switch (checkpoint incomplete)
156     337 log file switch (checkpoint incomplete)
107     317 log file switch (checkpoint incomplete)
26      52 db file scattered read
61     570 db file scattered read
215    537 db file scattered read
120    512 db file scattered read
172    477 db file scattered read
138     31 db file scattered read
123    409 db file scattered read
102    449 db file scattered read
2     57783 db file parallel write

.....
127 rows selected.

```

注意到和 I/O 紧密相关的等待事件这里存在：

- db file scattered read 意味着可能存在全表扫描，占用大量的 I/O。
- log file switch (checkpoint incomplete) 意味着检查点完成过慢，导致日志无法切换。而这些因素显然最终都和系统的 I/O 能力直接相关。

#### 9.5.4 进一步的诊断

获知其存储设备为 Sun StorEdge T3 Array，我这样回复他。

(1) 其实根本问题还在于 I/O，检查点不能完成说明 DBWR 的性能太差，写出的过慢。写出慢是因为 IO 存在瓶颈，从你的 iostat 信息来看，在 io busy 为 99% 左右时，写入才达到

500k/s 左右，这个太慢了，正常应该有 1~2M/s 的速度。

(2) 应该找到全表扫描的语句进行相应优化，减少 I/O 竞争和使用。

(3) 这样大规模的并发 Redo，感觉是由大批量的数据更改操作导致的，你应该尝试找出这样的 SQL，看是否存在问题，能否优化，从而在根本上解决问题。

可以在问题出现时多做几次 Statspack 采样，基本应该可以从报告中发现问题所在。当然，能在当时出现问题时到数据库中抓取 SQL 是最好的。

(4) 结合另外一个和 Redo 相关的诊断案例，T3 可能存在硬件故障，导致了性能低下，所以也应该检查影响问题。

### 9.5.5 最后的问题定位

最后这位朋友回复邮件。

您说的很正确，经过这几天的状况比对，以及与另外一台备份主机做比对，我认为有两个原因。

(1) T3 出了问题，以致于 IO 效率大幅降低，我们试了另外一台“正常”表现的主机，发现做同一动作时，“正常”主机其 IO 写入值可以高到 23000k/s，而有问题的 T3 其 IO 值最高写入值仅达 1000k/s。这个已经请硬件厂商查找原因。

(2) 我们其中一个表格，竟然已经成长到了近 2000 万笔，原因是某项定期清除数据库之程序出了错误，造成数据量过大。而且很不幸地，很多查询均是做全表扫描，只要碰上处理这个表格的数据，就会造成恶性循环。在后来的 alert log 中，竟然看到了其中数笔查询都花了 25000 多秒，还造成了以下问题：

```
ORA-01555 caused by SQL statement below (Query Duration=25889 sec, SCN: 0x0000.21fb3c10):
```

最后发现是硬件问题后，就请硬件厂商检查，发现 T3 的 UPS 电池已经坏了，4 个坏了 3 个，而 UPS 电池会严重影响硬盘写入的效率，从 write back 变成 write through，所以硬盘效率会低落，更换电池后已经恢复正常。

### 9.5.6 小结

简要地收录这个案例，只是为了给大家提供一个处理问题的思路，从症状入手，结合种种蛛丝马迹，解决数据库问题并非困难。

## 9.6 总结

Oracle 数据库在管理的过程中，可能会遇到各种各样的问题，解决这些问题，不仅要求具备扎实的基础知识，而且要求我们能够把书本中学到的知识灵活运用到实践中去。所以大家在学习的过程中要认真思考，多做实验，把书上的东西切实变成自己的知识，这样才能够在实际工作中得心应手，应付自如。



## 后 记

感谢你，我的朋友们，如果你已经读完本书，并且有所收获，那将是最愿意听到的好消息。

这本书可以说是我 Oracle 生涯的一个阶段性总结，我的学习方法，我在学习和工作中遇到的疑难和案例，都在本书中进行了选择性的收录，并作了详细的分析和讲解，我的思考、我的思路，在本书中展露无疑。

也许你看到的一个简简单单的案例，在实际生产中就曾经是一个导致巨大损失的灾难；也许你看到的一句轻轻松松的 SQL 语句，在实际生产中却曾是困扰良久的问题。书中几乎每个案例都来自不同的公司、不同类型的生产环境，有着不同的故事。

很多朋友经常会询问我学习的方法，其实答案也许很简单，多学习、多思考，多到论坛提问、参与讨论，甚或只是旁观，就可以极大地开阔视野、扩展思路，助益学习。如果说有捷径的话，也许只需要两个字：勤奋。

我就是在这样的自我学习和问题解决过程中走过来的，很高兴能通过本书再经历了这样一段历程，而且这一次，还有了这么多朋友的陪伴。

算起来，接触 Oracle 已经有 5 年之久，在北京的漂泊也已经超过了 3 年。时间总是流逝得飞快。而对于 ITPUB 论坛，我已经是一个注册了 6 个年头的老会员，从开始对 Oracle 技术一无所知，到现在有所知之，ITPUB 一直是我最好的朋友。也许现在回首还太早，可是将来有一天，我想，我们都会无比怀念那风轻云淡、山高水长的花样年华。

当然还有你们，我的可能素未谋面的、可能相交已久的读者朋友们，也许将来有一天你们还会记得，在这样一个夏日里，有这样一本书曾经陪伴你走过一段学习的旅程。

人生就是如此奇妙。

最后还要感谢你们，熟悉的或者不熟悉的读者朋友们，你们使得这本书有了它的存在价值，并且因为有了你们的支持以及督促，我们才不敢懈怠，不断努力，走向卓越。

Eygle

2006 年 6 月 14 日 于北京