Oracle 数据库性能优化

盖国强 冯春培 叶梁 冯大辉 编著

- 如何学习Oracle
- DBA优化之路
- Statspack高级调整
- 使用SQL_TRACE/10046事件进行数据库诊断
- 表空间的存储管理与优化技术
- 关于Freelists和Freelist Groups的研究
- 深度分析数据库的热点块问题
- Shared Pool原理及性能分析
- 捕获问题SQL解决过多CPU消耗问题
- Web分页与优化技术

人民邮电出版社
POSTS & TELECOM PRESS

Oracle 数据库性能优化

盖国强 冯春培 叶梁 冯大辉 编著

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

Oracle 数据库性能优化 / 盖国强等编著．—北京：人民邮电出版社，2005.6

ISBN 7-115-13438-3

Ⅰ．O… 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（2005）第 048495 号

内 容 提 要

本书面向实际应用，从多个角度出发，对 Oracle 优化中的很多关键问题进行了深入全面的探讨，涵盖了 Oracle 优化的各个技术层面，从内存优化、IO 规划及优化，到 SQL 优化调整，以较为完整的体系阐述了 Oracle 的优化技术。

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，更主要的是，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。

Oracle 数据库性能优化

◆ 编　　著　盖国强　冯春培　叶　梁　冯大辉
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
读者热线　010-67132692
北京密云春雷印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：31.5
字数：763 千字　　2005 年 6 月第 1 版
印数：1 – 4 000 册　　2005 年 6 月北京第 1 次印刷

ISBN 7-115-13438-3/TP・4674

定价：65.00 元

前　言

关于本书

全球著名的 IT 及电信行业市场咨询和顾问机构 IDC 公司的统计数据（2005 年 3 月）表明，2004 年 Oracle 数据库市场年增长率为 14.5%，且以 41.3%的市场占有率居市场领先地位。与此同时，Oracle 数据库在国内也得到了进一步的发展和普及。

应用的普及势必对数据库的管理及优化提出了更高的要求，而 Oracle 数据库的性能优化一直是数据库管理中的重要环节，也是最复杂的内容之一。基于此，我们组织编写了本书。

作为国内著名 Oracle 技术论坛，ITPUB（http://www.itpub.net）一直致力于推动各种形式的技术互动、促进经验交流和提供多种共享资源，以期帮助更多的朋友学习并掌握 Oracle 技术。2004 年，ITPUB 首次编写自己的技术图书《Oracle 数据库 DBA 专题技术精粹》，得到了广大读者的支持与鼓励。

本书是 ITPUB 推出的第二本技术图书，由活跃在 ITPUB 上的一群资深 Oracle 技术人员组稿，历时一年多的努力而成，其主要内容都是来自于实践经验的提炼和总结，具有高度的实用性及参考性。

本书特点

本书内容来自于实践，又高于实践，不但分析实际工作中与性能相关的问题及原理，还提供了详细的操作步骤，兼顾深度与广度，具有很强的实战性和可操作性。

本书对 Oracle 优化中的很多关键问题进行了深入的探讨，其探索程度及实用性在目前市场上的同类书籍中还是比较罕见的。不管是一个初学者还是想深入学习 Oracle 优化技术的专业人士，都能在这本书中找到自己感兴趣的部分。

本书作者

本书集中了 ITPUB 的技术优势，是集体创作的结晶，由 ITPUB 四大技术版主盖国强、冯春培、叶梁和冯大辉担当主编，Richard J.Niemiec、吕学勇、陈吉平、段凌、何小栋、李轶楠、汪海、谢中辉、邢海捷、杨廷琨、

叶宇、张乐奕、陈宇红、诸超、张大鹏、张春宏、张权（排名不分先后）等参与了本书的编写。他们都是来自各行各业的资深数据库管理人员，具有丰富的实践经验。同时感谢他们为此书所付出的努力和心血。

值得一提的是，在本书即将完稿时，我们收到了 Richard J. Niemiec 先生委托吕学勇先生带来的文章“ Statspack 高级调整 ”，现在把这篇文章收入本书，以便更多的读者可以从中受益；我们也欣喜地看到 DBA 渐无国界，希望国内的数据库从业者能够渐渐走向国际。

本书结构

本书分为 5 篇，共 30 章，具体结构划分如下。

- 第一篇：优化工具篇，包括第 1～6 章。这部分内容对 Oracle 数据库优化中常用的工具进行介绍，并辅以案例说明。
- 第二篇：存储优化篇，包括第 7～12 章。这部分内容主要对于存储及 IO 优化相关问题进行探讨。
- 第三篇：内存调整篇，包括第 13～17 章。这部分内容对于内存结构、原理以及优化调整进行了分析。
- 第四篇：诊断案例篇，包括第 18～24 章。这部分包含了实践中经常遇到的问题及案例诊断，通过具体案例介绍解决问题的方法及思路。
- 第五篇：SQL 优化及其他篇，包括第 25～32 章。这部分内容对于实践中棘手的 SQL 优化问题及相关问题进行了探讨。

本书的读者对象

本书适用于具有一定数据库基础的数据库从业人员，尤其适用于 Oracle 数据库管理和开发人员，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．为了简便，本书在不影响读者阅读的前提下采用了简称，例如，“ Oracle 9i ”简称“ 9i ”，“ Oracle 8i ”简称“ 8i ”等。

2．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，本书提供了大量的参考链接，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。因为这些链接地址会因时间而有所变动或调整，所以在此说明，这些链接信息仅供参考，本书无法保证所有的这些信息是长期有效的。

3．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。

4．广大读者如有好的建议或在学习本书中遇到疑难问题，欢迎到网站(http://www.itpub.net)进行探讨，也可发电子邮件联系我们(magazine@itpub.net)。

编 者

2005.4

目　录

第一篇　优化工具篇

第五篇 SQL 优化及其他

第一篇
优化工具篇

本篇共分 6 章，主要内容如下：

第 1 章　DBA 优化之路

第 2 章　Statspack 高级调整

第 3 章　Statspack 使用的几个误区

第 4 章　TKPROF 工具使用简介

第 5 章　使用 Oracle 的等待事件检测性能瓶颈

第 6 章　使用 SQL_TRACE/10046 事件进行数据库诊断

第1章　DBA优化之路

编者按：怎样学习 Oracle，怎样着手优化数据库，这一直是一个反复被提及和讨论的问题，本章并非介绍具体的方法论，而是一篇过来人的经验之谈，希望大家能够从本章的建议以及推荐中，找到一条更为平坦的优化之路。

也许你经过一翻努力，终于有了份 DBA 的工作，忐忑不安地坐在计算机旁，激动得手心冒汗，却不知如何去调整、优化数据库；面对突如其来的故障，电话响个不停，老板虎视眈眈地站在身旁，不知你此时是否能静下心来？

可能你读了许多数据库管理、调优、备份、恢复、PL/SQL 开发方面的书，也可能做了很多故障排除的实验，可当故障真正降临时，却显得那么突然，一点先兆也没有，麻烦总是从你意想不到的地方出现。

数据库系统本身永远是值得注意的麻烦制造者：数不清的 Bug、对象失效、磁盘碎片、索引重建，以及很多没有顾及到的突发事件等。没有数据库开发经验的程序员也是不可忽视的麻烦制造者：编写性能不佳的 SQL 以及创建一些性能较差的存储对象。最可怕的麻烦制造者是谁呢？正是来源于 DBA 本身，对数据库一个微小的修改，或许就将导致一场灾难。

作为一名 DBA，应该怎样学习 Oracle 和着手优化数据库呢？

1.1　学习的建议

对一个初级 DBA 而言，有关 Oracle 体系结构的概念非常重要，如果想比较透彻地理解这些概念，必须做大量的实验，“书上得来终觉浅，绝知些事要躬行”，但千万不要在生产环境中进行实验。如果想从麻烦制造者成长为一个麻烦终结者，只顾自己埋头苦学是不够的，毕竟在生产环境与学习环境中遇到的故障是很有限的，你需要在相关论坛上阅读大量的帖子，从网友的经验与教训中汲取营养，才能尽快拓展发现与解决问题的能力。

独立学习与思考是 DBA 快速成长的关键。许多新手发现系统出现问题或未知的现象，第一时间就是去咨询资深 DBA，其实这并不是好习惯。越是没有经验越是应该尽量对问题进行分析与推断，如果实在没有头绪的话，可以在 Google 或相关的论坛上进行搜索和求助，网络上总会有许多意想不到的惊喜，相信 99%的问题在网络上都可以找到答案，关键是如何找到它们。

一开始，不要对 Oracle Internal 的东西费心费神，打好基础才是关键。PL/SQL 开发知识必不可少，应了解体系结构、备份、恢复等方面的内容；努力提高系统调优及 SQL 优化的技能。当知识累积到一定的层次时，对于 Oracle Internal 的东西自然而然就可以领会。

良好的沟通能力有助于更快地解决问题。很多时候，问题可能已经解决，却不知为什么会产生这种问题，这时可以咨询一下项目负责人或相关程序员，尽量把问题的根源搞清楚，如果某个问题没有从根本上解决，那么这个问题必然会卷土重来。

需要为项目组的程序开发人员提供统一的《数据库开发规范》。如果可能，可以对程序员进行数据库设计（建模）及 SQL 优化方面的培训，尽量让性能不佳的 SQL 胎死腹中。要尽快融入项目组，理解业务系统的需求及发展趋势，通过对数据库结构及业务系统的掌握，为后期产品数据库的结构优化与 SQL 优化打下基础。

需要注意的是，并不是 Oracle 方方面面的知识都需要熟记在心，应该有选择地深入研究 Oracle 技术的某个方面，这样才能突破泛泛之境。不要太在意安装与配置 DataGuard、RAC 等，这些只是雕虫小技而已。piner 网友整理了关于 Oracle 的 FAQ（http://www.itpub.net/180363.html），无论新手还是资深 DBA，都可以从中受益。

1.2 工具推荐

“工欲善其事，必先利其器”，DBA 必须为自己及程序员搭建快捷顺心的工作环境。在 Linux 平台上，SQL*Plus 是不具有历史命令回调功能的，那么如何搭建具有历史命令回调功能的 SQL*Plus 呢？Fenng 网友的帖子（http://www.DBAnotes.net/Oracle/uniread-howto.htm），详细地介绍了其配置过程。

还有就是安装 SQL*Plus 的 Help 及 SQL 语法的 Help，具体方法可以参考这个帖子：http://www.cnoug.org/viewthread.php?tid=1710。在 Oracle 9i 以后的版本中，SQL*Plus 的 Help 是默认安装的，而 SQL 语法的 Help 必须自己安装，遗憾的是，目前在网上只能找到 Oracle 8.0.5 SQL 语法的 Help，不过这些足以应付 DBA 的日常工作。

Oracle 的官方文档是学习 Oracle 的必备读物，内容详实而准确，可以在 Oracle 站点（http://tahiti.oracle.com）找到这些文档，如果觉得 HTML/PDF 格式文档浏览不够方便，ITPUB 网友雪狼（Snowywolf）已经把这些文档编译为.chm 格式，大家可以在他的个人网站（http://www.snowywolf.net/chm.htm）上找到这些内容。

1.3 关于操作系统方面的建议

对 Linux/UNIX 应该有相当的基础。理解 RAID、RAW、LVM、OCFS、ASM 等与存储相关的概念；能够安装 Oracle 软件及打补丁；能够安装 Apache、PHP 等软件；可以熟练使用 Linux/UNIX 常用的命令 rpm、cpio、tar、ftp、top、vmstat、iostat、sar、netstat、ifconfig、crontab、chmod、chown、lsof、ntsysv、mv、rm、vi、mkdir 和 rmdir 等。

关于 Linux/UNIX 的问题，可以到 Linux 相关论坛 http://www.chinaunix.com 、http://linuxforum.net 等寻找答案。

1.4　关于 Oracle 初始化参数的调整

深刻理解 Oracle 初始化参数的含义是 DBA 必不可少的功课，但不能把调整初始化参数作为提高数据库性能的救命稻草，不合适的初始化参数设置必将带来性能上的下降，甚至引发数据丢失的危险。也不要以使用隐藏参数为荣，隐藏参数只是不得已而为之的手段，做事要未雨绸缪，这样才能在系统出现故障时坦然面对。

没有任何公式可以满足 SGA（System Global Area）调整的需要，OLAP（Online Analytical Processing）应用与 OLTP（Online Transaction Processing）应用初始化参数的调整是有很大区别的。初始化参数需要经过多次调整，才能达到比较和谐的效果，具体可以参考一下网友 biti_rainy 关于 SGA 调整的帖子（http://www.itpub.net/137600.html）。在 32bit 的操作系统中，SGA 有 1.7GB 的限制，如果想在 32bit 的操作系统上突破 1.7GB 的限制，必须使用特殊的手段，网友 coolyl 的帖子（http://www.itpub.net/124424.html）针对各个操作系统平台进行了总结。

如果有足够的内存，会不会将 SGA 设置得足够大？有位朋友的物理内存为 16GB，将 SGA 设置到 9GB 左右，实际上他的数据库才 13GB，数据库几乎全被缓存到 SGA 中，但操作员还是报怨系统很慢。Statspack 报表中的部分信息如图 1-1、1-2 所示。

Load Profile

	Per Second	Per Transaction
Redo size:	6,679.72	15,071.00
Logical reads:	80,724.11	182,132.26
Block changes:	23.75	53.58
Physical reads:	0.70	1.57

图 1-1　Load Profile

Instance Efficiency Percentages (Target 100%)

Buffer Nowait %:	100.00	Redo NoWait %:	100.00
Buffer Hit %:	100.00	In-memory Sort %:	100.00
Library Hit %:	99.74	Soft Parse %:	99.93
Execute to Parse %:	31.87	Latch Hit %:	99.87
Parse CPU to Parse Elapsd %:	95.57	% Non-Parse CPU:	98.71

图 1-2　Instance Efficiency Percentages

由于 SGA 足够得大，数据库中 Buffer Hit 是 100%，其他的各项指标也非常好，但在 Load Profile 项中可以看到 Logical reads 比较突出。然后在 Statspack 报表的 Top SQL 部分，发现 SQL 的单次逻辑读也比较大，通过优化 Top SQL 部分列出的 SQL，业务系统的速度明显提高；将 SGA 从 9GB 调到 3GB，通过观察 Statspack 的报表，各项指针并没有显示上升，Top 5 Event 也没有什么变化。再将 SGA 从 3GB 调到 1GB 时，虽然 Load Profile 及 Instance Efficiency Percentages 的各项指标没有明显变化，但 Top 5 Event 中的等待事件的计数猛增，同时 Tablespace IO Stats/File IO Stats for DB 部分的 Read 计数也大幅增加。

从这也可以看出，运转良好的数据库，它的数据量与 SGA 有一定的比例关系。如果在数据量一定的情况下，SGA 大大超过那个比例，系统的性能并不会有很大的提高，随着业务及数据量的增加，数据库的性能肯定是慢慢恶化，如果性能调优的速度赶不上系统恶化的速度，那么系统运行的速度就会让操作员无法忍受。

如何确定 Buffer Cache 中 KEEP 池与 DEFAULT 池的分配？如果想使用 DEFAULT 池与 KEEP 池，那么在 9i 中需要指定 db_cache_size 及 db_keep_cache_size 参数的值，如何为它们分配一块合适的内存呢？可以根据 Statspack 报表来确定热表及热索引。如何利用 Statspack 确定热表及热索引呢？可以参考 http://blog.itpub.net/post/96/14353。确定了热表及热索引就可以计算出这些热表所需要的内存，也就确定了 db_keep_cache_size 参数的值。利用 alter table &table_name storage(buffer_pool keep)可以将收集到的热表缓存到 KEEP 池，再收集一些 Statspack 的快照，产生一个 Statspack 报表，DEFAULT 池与 KEEP 池的效能如图 1-3 所示。

P	Number of Buffers	Cache Hit %	Buffer Gets	Physical Reads	Physical Writes	Free Buffer Waits	Write Complete Waits	Buffer Busy Waits
D	238,238	99.9	310,676,566	369,492	265,827	0	0	175
K	18,018	100.0	324,891,645	2,032	36,149	0	0	17

图 1-3 DEFAULT 池与 KEEP 池的效能

K 所标识是 KEEP 池的效能，D 所标识的是 DEFAULT 池的效能。如果 K 的 Cache Hit %小于 95%，说明 KEEP 池分配不足；如果 D 显示的 Cache Hit %小于 95%，说明 DEFAULT 池分配不足，如果 K 显示的 Buffers Hit 是 100%，那么可以将更多的热表放入到 KEEP 池中，然后经过一段时间的调整，相信可以将 DEFAULT 池与 KEEP 池调到一个比较合适的程度。

1.5 关于 Statspack 的若干建议

Statspack 可以找出系统性能瓶颈所在，但仅此而已。引起性能瓶颈的根本原因可能已经被收集到，也可能没有收集到，定时取样就会产生这样的结果。快照收集间隔过长可能会导致性能不佳的原因收集不到，快照收集间隔太短将会影响系统性能，所以关于 Statspack 快照收集的间隔最好根据系统负载情况而定，通常在 30 分钟左右。

在 Statspack 快照收集到的数据中，有些 SQL 语句太长，在 Statspack 报表中就显示不完整，因为在 Statspack 报表中，通常收集的 SQL 语句只显示 5 行；如果想定制 Statspack 报表中 SQL 语句的显示行数，可以通过修改 $ORACLE_HOME/rdbms/admin/sprepins.sql 档中的 num_rows_per_hash 值，来达到要求。

Statspack 可以显示过去某段时间数据库的运行状态，以及预测将来一段时间数据库性能变化趋势（初始化参数没有重大调整及业务没有急剧变化的情况下），但也不要对 Statspack 抱太大希望，因为想从 Statspack 报表中找出系统瓶颈，也不是一件容易的事，它要求 DBA 必须有扎实的理论基础与丰富的实践经验。

当 DBA 在 Oracle 性能调优方面的知识累积得越多，在 Statspack 报表中可以挖掘的问题就越多。通过对 Statspack 的报表进行分析，DBA 可以对初始化参数进一步微调，也可以对 Top SQL 中列出的性能不佳的 SQL 语句进行优化。

对 Statspack 比较感兴趣的朋友，可以参考 eygle 网友关于 Statspack 的系列精彩帖子（http://www.eygle.com/Statspack/Statspack_list.htm），内容包括 Statspack 的安装、配置等；也可参考本人关于 Statspack 使用的帖子（http://blog.itpub.net/post/96/14353）：如何在 Linux 上用 crontab 定时产生 Statspack 的报表等技巧。

通过对 Statspack 报表的研究，可以大大提高 DBA 在系统调优方面的能力，Statspack 不仅可以对当前系统的运行进行诊断，还可以对系统将来的运行状态进行预测。随着基于等待事件的系统调优方法的流行，Statspack 已经成为 DBA 不可缺少的助手，同时 Statspack 报表也成为 DBA 之间进行交流的有效方法之一。

如果需要制做 Statspack 的性能趋势报表，一般可以用 Excel 来做，但是比较麻烦。本人写了一款专门制做 Statspack 报表的工具，不仅可以更快地制作出漂亮的报表，而且可以对相关的知识进行管理（http://www.cnoug.org/viewthread.php?tid=20115）。

推荐一本关于 Statspack 的书《Oracle 9i STATSPACK 高性能调整》，其详细地介绍了 Statspack 优化的方方面面（系统、网络、数据库等），对于初学者来说绝对是值得一读的，关于该书的“内容介绍”可以参考 http://blog.itpub.net/post/96/14763。通过这本书的学习，读者将会对 Statspack 的体系结构有非常清晰的理解。

biti_rainy 网友有个关于 Statspack 诊断负载突增的帖子（http://blog.itpub.net/post/330/2230），使本人受益颇深，原来 Statspack 的报表竟然可以这样运用，不知不觉就养成了每晚读当天 Statspack 报表的习惯。每次读 Statspack 的报表，若发现数据库有什么细微变化时，嗅觉必须非常敏锐，不要轻易放过任何“小事情”。对于异常的等待事件（平时不出现）或负载突变的情况，最好能追根求源，将瓶颈消弭于无形。

1.6　关于 logmnr 在调优中的运用

一直以来，logmnr 都不是调优所推荐的工具，它主要用于安全审计方面。其实在查找系统瓶颈上，logmnr 的优势可谓得天独厚，但这需要 DBA 有足够的耐心。通过对在线日志或归档日志的审查，可以清楚地知道 Oracle 数据库在过去某段时间内做了什么，至于这样做是不是合理？DBA 必须自己去判断，logmnr 并不能告诉什么是不合理的。

有一次，生产库负载突然上升了很多（Top 上看到的），从平时的 2 上升到 5，从 Statspack 报表中也看不出什么原因，此时系统的 iowait 一直很高，很多客户投诉系统较慢，我发现日志切换非常频繁，就用 logmnr 对日志进行分析，方法如下：

```
sql>exec dbms_logmnr_d.build('esal.ora','/home/Oracle/logmnr');
sql>exec dbms_logmnr.add_logfile('&log_file',dbms_logmnr.new);
sql>exec dbms_logmnr.add_logfile('&log_file',dbms_logmnr.addfile);
sql>exec dbms_logmnr.start_logmnr('/home/Oracle/logmnr/esal.ora');
sql>exec dbms_logmnr.end_logmnr();
```

通过挖掘日志发现，某表上有大量的 insert 操作，每个 insert 操作都有一个 commit 操作，一看就知道是程序方面出了问题。本来在数据批量装入时，计划每 500 条 commit 一次，由于程序员忘了处理这一问题，结果每个 insert 操作都执行了 commit 操作，造成系统的 I/O 瓶颈，当程序员把程序修正后，系统的负载很快就恢复正常了。

在 B/S 结构的应用中，在 session 连接时用 dbms_application_info.set_client_info 设置 session 的 client_info，这样在用 logmnr 进行日志挖掘时，就可以知道是哪个页面执行了这个操作，诊断范围就比较小。在 C/S 结构的应用中，通常每个 Client 连接后，都可能需要很久才断开 session，在客户每打开某个业务模块时，最好用 dbms_application_info.set_client_info 设置该 session 的 client_info，缩小故障诊断范围。

1.7 关于 materialized view 在调优中的运用

在 OLAP 环境中，mview 是以空间换时间的一种有效手段，它可以带来更少的物理读/写，更少的 CPU 时间，更快的响应速度，它不适合高端的 OLTP 环境，如果创建 mview 基表的事务非常多，那么 mview 的刷新将会对系统造成一定的压力。在 OLTP 环境中，规模较大的报表也适合使用 mview 来提高查询性能。《expert one-on-one Oracle》的第 13 章专门讲述 mview 的运用，关于本书的详细信息可以访问 http://www.itpub.net/224536.html。

根据本人测试 mview 的情况，可以看出 mview 在 Oracle 10.1.0.2/10.1.0.3 上还没有成熟，针对这两个版本的测试结果是不相同的，具体可以参考 http://blog.itpub.net/post/96/7535。还有就是，在分区表上创建 fast refresh 的 mveiw 后，分区维护一定要小心，如果对分区进行分割（alter table &table_name split partition…），该分区表上的 mview 将不能被 fast refresh，所有针对该分区表的事务将失败，具体可以参见 http://blog.itpub.net/post/96/3809。

1.8 关于 Stored Outline 在 SQL 优化中的运用

Stored Outline 是为了维持 SQL 执行计划稳定性而推出的功能，主要适用于无法对程序的源代码进行修改等情况，为了保证产品数据库的良好运行，人为调整某些特定 SQL 的执行计划，比较适合使用 bind variable 的 SQL。

关于 Stored Outline 的使用，可以参考 http://blog.itpub.net/post/96/1548。曾对 Stored Outline 抱有厚望，但在实际运用中却发现 Outline 并不是那么好，当 SQL 使用 bind variable 的情况下用 Stored Outline 来稳定计划会更合适一些，否则针对某个 SQL 的每个文本常量定制一个 Stored Outline 就有些不可思议了。

当初始化参数 cursor_sharing=EXACT 时，若查询条件是文本常量（where id=2/3/4）等，那么在这种情况下，就不用 Stored Outline 对该类型的 SQL 进行执行计划的稳定，除非对该 SQL 用 bind variable，或将 cursor_sharing=SIMILAR（或 FORCE），关于 cursor_sharing 的取值及使用，可以参考 http://blog.itpub.net/post/96/8936。

1.9 用 dbms_profiler 调优存储过程

dbms_profiler package 主要用于 PL/SQL Block、Stored Procedure、Stored Function 的性能优化，程序员或 DBA 需要对开发的各种存储对象进行测试，通过 dbms_profiler package 可以找出存储对象中性能不佳的地方，然后再进行改进。dbms_profiler package 不仅可用于开发阶段对存储对象进行性能测试，也可用来测试产品数据库。如何安装和使用 dbms_profiler package 呢？需要用 SYS 用户或有 SYSDBA 权限的用户登录系统，执行以下脚本文件（Linux 平台，Oracle 9i）。

创建相关 packages：

```
sql>@?/rdbms/admin/profload
```

创建相关 tables：

```
sql>@?/rdbms/admin/proftab
```

创建相关 views 及 prof_report_utilities package：

```
sql>@?/plsql/demo/profrep
```

需要注意的是，Oracle 10.1.0.2/10.1.0.3 在 plsql/demo 目录内没有发布 profrep.sql 及 profsum.sql，想使用 10GB 来调试 PL/SQL block 或其他存储对象，只能从 9i 的 plsql/demo 目录内去拷贝 profrep.sql 及 profsum.sql，以下是 dbms_profiler 包的示例，关于 profsum.sql 用法可以参考 http://blog.itpub.net/post/96/16572。

以下是 dbms_profiler 测试存储过程的一段 PL/SQL 代码，使用时不需要做什么修改。其中 start_profiler 和 stop_profiler 必须成对出现，rollup_run 过程计算被测试存储过程总的使用时间，prof_report_utilities.print_run 输出详细执行时间清单。

```
declare
  v_run number;
begin
  dbms_profiler.start_profiler(run_number=>v_run);
  &procedure_name;--输入你要测试的过程名称
  dbms_profiler.stop_profiler;
  dbms_profiler.rollup_run(v_run);
  prof_report_utilities.print_run(v_run);
end;
/
```

1.10　优化前的准备工作

一个初级 DBA 成长为一个资深 DBA，这其中最重要的是对数据库的兴趣。调优要求 DBA 拥有丰富的基础知识，从系统配置、数据库设计、业务处理、SQL 编写等，每一项都不是一两天可以熟悉和掌握的。

但这并不意味着只有资深 DBA 才可以做优化，只要愿意，你就可以踏上数据库优化之旅。初级 DBA 与资深 DBA 可能在优化的方法、效果上稍有不同，但只要有精亦求精的决心，那就有可能与资深 DBA 一样，做到殊途同归或同途同归。不断的努力测试、累积经验，是初级 DBA 成为资深 DBA 的必经之路。

读一些优化基础之类的书是必要的，如果需要买些参考书，笔者推荐《Oracle 9i 性能调整》，虽然翻译得不是很好，但该书的内容确实不错，涉及索引知识、性能相关参数、实用工具、SQL Hints、UNIX 监控等内容。关于该书的“内容介绍”可以参阅 http://blog.itpub.net/post/96/14758。

初级 DBA 要有较好的 PL/SQL 编写基础，虽然 DBA 不一定要亲自写存储对象或 PL/SQL 代码，但必须能够看懂程序员或其他 DBA 编写的存储对象。好的 PL/SQL 编写能力不仅使你在数据库开发方面游刃有余，而且也可指导程序员不要将存储对象写得一塌糊涂。如果你对 PL/SQL 还不是很熟，最好按 Oracle 的在线文档（http://tahiti.oracle.com）进行一些练习；同时还要养成良好的 PL/SQL 编码习惯，具体可参考笔者的转贴（http://blog.itpub.net/post/96/13123）。

不要怕麻烦，要经常做测试。关注一些 Oracle 相关论坛上关于 SQL 调优案例的帖子，然后自己也照着做一遍，消化吸收别人的经验，长此以往地积思广益，在真正做优化时就可以避免犯一些常识性错误。初级 DBA 需要理解及掌握关于优化的重点内容，包括优化器 RBO/CBO、执行成本（Cost）及执行计划（Plan）、优化提示（Hints）等。

了解 v$sql、v$sqlarea 字典，可以从这些字典表中识别出性能较差的 SQL，并了解该 SQL 所实现的业务，分析研究能否通过优化业务流程来达到优化 SQL 性能的目的。要与程序员多沟通，很多 SQL 优化方案是双方努力交流的结果。

了解 SQL 涉及表的数据分布、增长、索引情况。了解数据的分布为索引创建提供依据，比如创建 B-Tree Index 还是 Bitmap Index；了解数据增长情况为优化表结构提供依据，比如是否需要创建分区表等；如果是分区表，确认分区表上的索引类型，是 Global Index 还是 Local Index；确定需要创建的索引，并为索引指定一个比较好记的名称。

1.11 如何对 SQL 进行调整及优化

初始化参数调整与 SQL 优化是最能体现 DBA 智能与价值的工作。通常在 Statspack 的 Top 5 Wait Event 主要是由性能不佳的 SQL 引起的，磁盘排序及 temp 表空间暴涨等多半也与 SQL 有关，不排除创建索引与重建索引时引起 temp 表空间暴涨的问题，但这方面的原因应该由 DBA 负责，最好把为大表创建索引或重建索引的工作安排在系统空闲时。

性能不佳的 SQL 是如何产生的呢？这里面的问题比较复杂。不良的数据库结构必将导致不良的 SQL；还有就是由于程序员有限的 SQL 编写技能而引起。不要奢望程序员是编写 SQL 的专家，在最短时间内完成项目才是程序员最关心的，所以程序员通常不会太关注 SQL 的性能，即是关心也是很有限的。

对程序员进行适当的关于 SQL 优化的培训，提高他们对 SQL 性能的重视程度，针对系统中出现的案例进行分析和讲解，这样程序员潜意识中就会注意避免很多低级的错误。要多与程序员交流，尽量引导程序员提出他们在数据库方面的疑问，并给予他们指导性的意见及解决方案。

对初级 DBA 而言，通常都很有兴趣对系统参数或 SQL 进行调优，却不知如何动手。首先在初始化系统参数方面，初级 DBA 自身一定要对这些内容有所理解，遇到困难时也可以请教资深 DBA（有一定经验的 DBA 都可以把初始化参数调到一个比较合适的程度）。但是不要对初始化系统参数在性能提高方面抱有太大的希望，比如 db_cache_size/shared_pool_size 大一些或小一些对系统性能的影响都不是很大。也可以根据 Statspack 的报表进行分析，对初始化参数进行微调，多少是能带来一些好处的。而在 SQL 调优方面，就必须能够收集并找出性能不佳的 SQL。

如何收集并找出性能不佳的 SQL 呢？通常要综合多项性能指针来进行判断，如 SQL 的响应时间（Response Time）、逻辑读（Consistent Gets）、物理读（Physical Reads）、结果集行数（Resultset Size）等。要根据系统的硬件配置情况（因为好的硬件配置确实可以延缓系统瓶颈的爆发），设置恰当的收集的阀值，然后从 v$sql、v$sqlarea、v$sqltext_new_withlines 字典表中把符合条件的 SQL 查询出来（也可以修改 Statspack 收集的阀值）：

```
set lines 99
col sql_text format a81
col bgets_per format 99999999.9
set long 99999999999
set pagesize 9999
select address,hash_value,disk_reads,elapsed_time/1000000 as
```

```
    "elapsd_time(s)",cpu_time/1000000 as "cpu_time(s)",
          buffer_gets/executions bgets_per,first_load_time,sql_text
    from v$sql where executions > 0
    and (disk_reads/executions > 500 or buffer_gets/executions > 20000);
```

上面的这个查询主要将 Physical Reads > 500 及 Consistent Gets > 20000 的 SQL 语句找了出来，可以对响应时间也进行限制，通常 Consistent Gets 较大或 Physical Reads 较大的 SQL，它的 Response Time 也必然会比较长。当收集到符合条件的 SQL 后，就要动手对它进行优化，那该如何进行优化呢？

首先，对 SQL 的语法进行分析，剔除冗余的、错误的查询条件（有可能是程序员手误），以及修正一些常识性错误，如 to_char(log_time,'yyyy-mm-dd')='2004-01-03'会导致 log_time 字段的索引未被使用等错误。这些分析和修正工作不会花费很多工夫，但 SQL 的性能却可以因此得到极大的提高。

其次，对 SQL 涉及表的结构、查询字段、连接字段的数据分布情况等进行分析，特别是复杂的 SQL，要检查它们是否有更佳的连接路线，连接字段是否有合适的索引，索引的选择性如何等。如果 SQL 的结果集较大，那响应时间长一些也是正常的。

最后，尝试用不同的 hints 改变表的驱动次序及 SQL 的执行计划，hints 分归档的 hints 与未归档的 hints，无论是归档的还是未归档的 hints，都应尽量少用。关于所有的 hints 列表可以参考 http://www.adp-gmbh.ch/ora/sql/hints.html，关于 hints 的具体用法可以参考 Oracle 在线文档（http://tahiti.oracle.com）。

如何在 SQL 执行时产生执行计划呢？在 SQL*Plus 上输入 set autot on 就可以让 SQL*Plus 产生比较详细的执行计划；set autot off 可以让 SQL*Plus 取消产生执行计划；set autot traceonly 可以让 SQL*Plus 显示 SQL 影响的行数、执行计划、执行的统计信息和不输出结果集；set autot on exp 可以让 SQL*Plus 输出执行后的结果集及执行计划；set autot on stat 可以让 SQL*Plus 输出执行后的结果集及统计信息。

```
    set autot[race] {off|on|trace[only]}[exp[lain]] [stat[istics]]
```

至于 explain plan 语句，它也可以对 SQL 的执行计划进行解释。它不仅可以解释使用文本常量的 SQL，也可以对使用 bind var 的 SQL 进行解释，它必须使用 dbms_xplan package 的 display 函数将执行计划显示出来，如下所示（具体可参见 http://blog.itpub.net/post/96/9781）。

```
    explain plan [set statement_id = &item_id] for &sql;
    select * from table(dbms_xplan.display);
```

如何对性能不佳的 SQL 进行优化？这对任何一个 DBA 都具有挑战性。在这个环节上，DBA 必须掌握如何查看 SQL 的执行计划，并对返回结果有一定的了解。如果是新手，可以借助一些 SQL 优化工具进行调优，可借助的工具有 LECCO SQL Expert 及 Quest TOAD，鉴于新手对工具的理解还存在着一些问题，本人为 LECCO SQL Expert 写了中文图解，详细信息可以参考以下链接。

- SQL Expert 教程 http://www.cnoug.org/viewthread.php?tid=22327
- Quest TOAD 教程 http://www.cnoug.org/viewthread.php?tid=3242

任何工具都是比较低智能的，如果你觉得 LECCO 或 TOAD 比较顺手，千万别沉溺其中，它们只是一个拐杖而已，你必须超越它，否则你的价值就值得怀疑。针对 SQL 的优化，必须自己多动手测试，而且也要博览群书，从别人的经验中激发灵感。

关于 SQL 调优的细节很多，不可能一一列举，具体环境必须以执行计划为准，通过对 SQL 的理解，上升到对数据库结构的合理性进行揣测。合理的数据库结构将对 SQL 的性能有较大的改

善。在有些情况下，修改了数据库结构，并不需要在程序上也进行相应的改动，比如将大表进行分区、创建 mview 等。

1.12 表结构优化实例

在系统调优行为中，最容易实现的就是找出性能不佳的 SQL，然后再通过各种手段对 SQL 进行优化，其实这已经是下下之策，虽然可以解决燃眉之急，却不是治根治源的良方，优化数据库结构才是关键。有丰富设计经验的数据库设计师，通常都能够考虑到将来业务快速发展所产生的数据激增，并能针对该种情况设计出一套应对海量数据的方案，那么数据库存在性能瓶颈或性能不佳 SQL 的可能就会大大降低。

如果新接手一个系统的维护，有可能就会遇到许多失败的设计，这时就不能按哪儿疼医哪儿的办法来处理这种问题，必须找出问题所在。根据经常查询的性能不佳的报表，规划出行之有效的整改方案，才能解脱被动挨打的局面。

表结构优化通常伴随着业务实现的整改以及相当大的程序量，所以编写整改方案时必须要全面细致、有说服力，这样才能获得项目组的支持。小的整改方案只涉及到一个表，大的整改方案可能会涉及很多表，涉及的表越多，遇到的阻力就越大。并不一定是海量数据的表要进行调整，只要该表设计不合理，且访问的次数特别频繁，那就是整改的对象。以下链接提供了一个小的整改案例以供参考：http://blog.itpub.net/post/96/14657。

1.13 如何对 session 进行跟踪

跟踪 session 的活动，Oracle 提供了很多种手段，不仅可以对当前连接的 session 进行跟踪，也可以对其他用户的 session 进行跟踪；通过对 trace 档的分析，不仅可以掌握该 session 的活动，也可以找出这个 session 中的瓶颈所在，对 session 的跟踪是 DBA 进行系统调优、故障诊断的常用方法。

对当前会话的活动进行跟踪及停止跟踪：

```
alter session set sql_trace=true/false
```

对任意的 session 进行跟踪及停止跟踪：

```
exec dbms_system.set_sql_trace_in_session(&sid,&serial#,&sql_trace);
alter session set events '&event trace name context forever,level &level';
alter session set events '&event trace name context off';
exec dbms_system.set_ev(&sid,&serial#,&event_10046,&level_12,'');
oradebug event 10046 trace name context forever,level 12
```

利用 Event、SQL Trace 工具等可以收集 SQL 的性能状态数据并把这些数据记录到跟踪文件中，这个跟踪文件提供了许多有用的信息，如解析次数、执行次数、CPU 使用时间、物理读、逻辑读等，这些信息是判断 SQL 性能优劣的依据。user_dump_dest 参数说明了生成跟踪文件的目录，设置 SQL Trace 首先要在 init&sid.ora 中设定 timed_statistics 为 true，这样才能得到那些重要的时间信息，由于 SQL Trace 生成的 trace 文件读起来很困难，最好用 TKPROF 工具对其进行解释，TKPROF 有许多参数，具体可以参考 http://tahiti.oracle.com。

1.14　基于等待事件的性能诊断方法

等待事件（Wait Event）是 Oracle 核心代码的一个命名部分，有两种类型的等待事件：空闲事件（Idle Event）与非空闲事件（Non-Idle Event），空闲事件指 Oracle 正在等待某种工作，常见的空闲等待事件有 client message、null event、pipe get、pmon/smon timer、rdbms rpc message 及 SQL*Net 等；非空闲等待事件有 buffer busy waits、db file scattered read、db file sequential read、enqueue、free buffer waits、latch free、log file sync、log file paralle write 等。

一旦熟悉了系统的等待事件，就能够把握问题的关键，并能够采用相应的方法去处理阻塞系统的瓶颈，一定不要随意地进行优化，否则一波未息一波又起。可以通过 v$system_event 视图获取系统总的等待情况，然后通过 v$session_event 视图查看系统中 session 的等待情况，最后通过 v$session_wait 视图定位瓶颈对象。v$session_wait 视图是会话级的，它包含 session 的实时信息，最重要的是，它显示了等待事件与相应资源的详细信息，可确定出产生瓶颈的类型及其对象。

v$session_wait 视图中的 p1、p2、p3 表示等待事件的具体含义。如果 Wait Event 是 db file scattered read，那么 p1=file_id/p2=block_id/p3=blocks，然后通过 DBA_extents 即可确定出热点对象。如果是 latch free 的话，那么 p2 为闩锁号，它指向 v$latch。

```
    --求等待事件及其对应的 latch
    col event format a32
    col name format a32
    select sid,event,p1 as file_id, p2 as "block_id/latch", p3 as blocks,l.name
      from v$session_wait sw,v$latch l
    where event not like '%SQL%' and event not like '%rdbms%'
    and event not like '%mon%' and sw.p2 = l.latch#(+);
    --求等待事件及其热点对象
    col owner format a18
    col segment_name format a32
    col segment_type format a32
    select owner,segment_name,segment_type
      from DBA_extents
    where file_id = &file_id and &block_id between block_id
    and block_id + &blocks - 1;
    --综合以上两条 SQL，同时显示 latch 及热点对象（速度较慢）
    select sw.sid,event,l.name,de.segment_name
      from v$session_wait sw,v$latch l,DBA_extents de
    where event not like '%SQL%' and event not like '%rdbms%'
    and event not like '%mon%' and sw.p2 = l.latch#(+) and sw.p1 = de.file_id(+) and p2 between
de.block_id and de.block_id + de.blocks - 1;
    --如果是非空闲等待事件，通过等待会话的 SID 可以求出该会话在执行的 SQL
    select sql_text
     from v$sqltext_with_newlines st,v$session se
    where st.address=se.sql_address and st.hash_value=se.sql_hash_value
    and se.sid =&wait_sid order by piece;
```

通过等待事件找出系统中消耗资源较严重的 SQL，是 DBA 进行系统诊断的手段之一。只是过程稍嫌繁琐，由于 session 是动态的、瞬息万变、不可捕获，当你想捕获时，该 session 可能已经

释放，但这种捕获很有针对性；也可以通过对 v$sql 或 v$sqlarea 进行过滤，找出存在性能问题的 SQL。长时间地对 v$sql 进行监控，并对捕获的 SQL 进行优化处理，可以在很大程度上解决系统的性能问题。

1.15 基于资源限制的性能诊断方法

如果想用 resource limit 功能，就必须设置初始化参数 resource_limit=true，当然也可以指定相关的 resource_manager_plan 参数以实现更细致地管理资源。针对某个用户的资源限制，可以通过用户默认的 profile 来实现，也可以为该用户创建新的 profile，关于 create profile 的语法，可以参见 http://tahiti.oracle.com 中的相关文档。

如果不想创建 profile，可以使用默认的 profile，用户在创建时如果没有指定 profile，将使用默认的 profile。如果新创建了 profile，可以用 alter user &user_name profile &profile_name 的方法为用户指定新的 profile。必须清楚 profile 中的每一项的含义及导致的后果，否则设置不当，就会面临被客户投诉的待遇，通过 DBA_profiles 字典可以查出所有 profile 的设置信息，如何设置 profile 的限制与取消限制呢？具体如下。

```
alter profile default limit logical_reads_per_call 300000;
alter profile default limit logical_reads_per_call unlimited;
```

上面第一个语句表示设置 SQL 最大的逻辑读不能超过 300 000，第二个语句表示取消逻辑读的限制。假如设置了 deafult profile 中的 logical_reads_per_call，则表示每个 SQL 的最大逻辑读不能超过该参数设置的值，如果超过限制，session 将会报 02395（ORA-02395: exceeded call limit on IO usage）号错误。如果设置 logical_reads_per_session 的值，则表示每个 session 中所有 SQL 的逻辑读总计不能超过该值。

除了设置逻辑读之外，还可以设置 cpu_per_call，表示每个 SQL 调用在 CPU 时间方面的限制，单位是 1/100 秒；cpu_per_session 表示每个 session 可以使用的 CPU 时间，单位是 1/100 秒；connect_time/idle_time 单位是分钟。

```
sql>select * from DBA_profiles;
PROFILE  RESOURCE_NAME              RESOURCE LIMIT
---------------------------------- -------- ----------
DEFAULT  COMPOSITE_LIMIT            KERNEL   UNLIMITED
DEFAULT  SESSIONS_PER_USER          KERNEL   UNLIMITED
DEFAULT  CPU_PER_SESSION            KERNEL   UNLIMITED
DEFAULT  CPU_PER_CALL               KERNEL   UNLIMITED
DEFAULT  LOGICAL_READS_PER_SESSION  KERNEL   UNLIMITED
DEFAULT  LOGICAL_READS_PER_CALL     KERNEL   29999999
DEFAULT  IDLE_TIME                  KERNEL   UNLIMITED
DEFAULT  CONNECT_TIME               KERNEL   UNLIMITED
DEFAULT  PRIVATE_SGA                KERNEL   UNLIMITED
DEFAULT  FAILED_LOGIN_ATTEMPTS      PASSWORD 3
```

通常要设置的是 logical_reads_per_call/cpu_per_call，它们实际上是对 SQL 的逻辑读及响应时间进行了限制，如果超过这个限制，Oracle 将会终止该 SQL 的执行，前台页面可以把产生错误的 SQL 发到 DBA 的邮箱，DBA 几乎不费什么力气就可以收到性能较差的 SQL。如果 logical_reads_per_call 的值设置太小，会导致正常的查询被取消。

1.16　如何减少共享池的碎片

在 SGA 中，由 shared_pool_size 参数控制共享池的大小，共享池主要是由 SQL Area、Library Cache 等部分组成，每部分所耗用的内存可以通过 v$SGAstat 字典表查看。共享池碎片产生的原因很多，大概有以下方面的原因：

⑴ 不良的数据库设计，导致数据库对象众多。

⑵ 非 bind var 的 SQL，导致共享池过度交换。

⑶ 由 cursor_sharing 参数及 histogram 引起。

⑷ 项目组没有统一的 SQL 编写规范。

关于第（⑴）个原因，处理起来难度较大，风险也高，因为涉及到修改数据库结构，需要修改的地方就比较多，这样做首先要取得公司的支持，然后对数据库进行良构设计，消除结构上不合理的地方。例如，某公司的业务系统是从 MySQL 迁移到 Oracle 中的，有一些很重要的产品表以前是这样设计的 game_card_gm*xxxx*，由于最初产品不是很多，所以矛盾并不突出，但随着业务的扩展，产品表从几十个增加到几百个，问题就随之而来，每个表有 6 个索引，仅产品表相关的对象就上千个（含每个产品表的 sequence），庞大的对象体系必然增加了共享池（SQL Area、Library Cache 等）的消耗，如果数据库结构重新设计，只需要 1 个表、6 个索引、1 个序列就可以满足要求，并可以极大地减少 shared_pool 的争用。

关于第（⑵）个原因，非 bind var 的 SQL 导致共享池中的 SQL 被 aged out，被 aged out 的 SQL 大致可分为两种，可重用的和不可重用的，如 DDL（insert）、DCL（update）和 DML（delete）等，这些 SQL 的 99%是不可以重用的，aged out 后几乎不会被再次 reload。但 select 语句是可以重用的，如果大量的 select 被换进换出就会影响数据库的性能。

Library Cache Activity for DB: ESAL Instance: esal Snaps: 1855 -1869
->"Pct Misses" should be very low

Namespace	Get Requests	Pct Miss	Pin Requests	Pct Miss	Reloads	Invali-dations
BODY	59,807	0.1	59,811	0.2	27	0
CLUSTER	4,429	2.9	5,791	4.4	0	0
INDEX	3,684,268	0.0	1,849,315	0.0	0	0
SQL AREA	7,204,345	25.3	65,947,785	6.1	203,925	9,580
TABLE/PROCEDURE	14,662,989	0.1	20,785,992	0.5	41,058	0
TRIGGER	449,858	0.0	449,858	0.0	95	0

图 1-4　Library Cache Activity

如图 1-4 所示（图 1-4 从 Statspack 报表摘取），SQL Area 被 Reloads 共 20 万次以上，如果 Statspack 报表中关于 Library Cache Activity 的统计项中 SQL Area 的 Reloads 较大，需要考虑对 SQL 使用 bind var。如果业务数据增加较快，被 aged out 的 insert 肯定多，所以对该类型的业务最好使用 bind var。bind var 本质就是通过减少共享池的使用来减少共享池的碎片。

关于 cursor_sharing 参数及 histogram 引起共享池碎片的问题，biti_rainy 网友已有论述，如果感兴趣的话可以访问 http://blog.itpub.net/biti_rainy。

如果项目组没有统一的 SQL 编写规范，程序员在 SQL 编写方面会有很大的随意性，结果导致很多可以共享游标的 SQL 因程序员的失误被重新解析，所以成熟的项目组都有完善的开发规范，

用于减少人为的失误。

在这方面，本人的意见是，编写 SQL 时，组成 SQL 的所有字符单元都用小写字符，每个 SQL 的关键词与非关键词之间只保留一个空格，select 的 field list 与 where 子句中出现的查询字段必须按它们在表中出现的字段顺序为准，select 的 field list 中各字段以逗号（英文“,”）隔开，不必保留空格，如图 1-5 所示。

```
SQL>select id,name,age from test4;
--在 select 列表字段最好以它在表中创建时的顺序出现

SQL>select id,name,age
      from test4 where id = 9 or name ='00009' or age = 1;
--在 where 子句中字段最好以它在表中创建时的顺序出现

SQL>select id,age name
      from test4 where id = 9 and name ='00009' and age = 1;
--这种情况就会给其他人造成混淆，结果就会出现不应有的硬解析

SQL>SELECT id,name,age,
     FROM test4 where id = 9 or name='oooo9' or age=1;
--这个地方应该注意的是关键字必须小写。

SQL> insert into test4(id,name,age) values (:id,:name,:age);
SQL> insert into test4(id,name) values(:id,:name);
--这两个 SQL 中的后者与前者的字段列表不符，通常的写法如下:
SQL>insert into test4(id,name,age) values (:id,:name,:age)
--这个地方出现的: age 就在绑定时传入 null 或其他可以默认的值即可
```

图 1-5　SQL 编写示例

虽然这些约定很简单（见图 1-5），只要所有的程序员都按规范去写，必定可以减少共享池的碎片。熟悉 Oracle 的程序员可以在很大程度上减少共享池碎片，有丰富数据库设计经验的数据库设计师也可以避免数据库对象的膨胀。

尽量在应用软件设计时对 SQL 使用 bind variable，因为 cursor_sharing 控制 SQL 使用变量绑定时，有可能产生一些不可预知的 Bug。常用的开发工具都支持 SQL 的 bind variable（变量绑定），如 Delphi、PHP 等，无论是程序员还是 DBA，都需要知道变量绑定对系统的影响。

如何检验 SQL 使用 bind var 改进后的效果呢？一种方法就是看如图 1-4 所示的 SQL Area 的 Reloads 情况，如果将大量的 SQL 改成使用 bind var，那 SQL Area 部分的 Reloads 值必将有所下降；另一种方法是看如图 1-6（图 1-6 从 Statspack 报表中摘取）所示的 Soft Parse 情况，如果将大量的 SQL 改为使用 bind var 的语法，那 Soft Parse 的值必将有所提升。

```
Instance Efficiency Percentages (Target 100%)

            Buffer Nowait %:  100.00       Redo NoWait %:   99.99
            Buffer  Hit   %:   99.95    In-memory Sort %:  100.00
            Library Hit   %:   95.89       Soft Parse %:   93.94
        Execute to Parse %:    1.24         Latch Hit %:   99.70
Parse CPU to Parse Elapsd %:   75.62     % Non-Parse CPU:   71.61
```

图 1-6　关键能指标

1.17　监控表及索引的意义

除了 logmnr 及 audit 之外，还有什么方法可以监控对表的操作呢？大家可以试试监控表与监控索引。表监控记录主要针对表的一些操作，如 truncate、delete、update、insert 等；索引监控主要记录索引什么时间被使用过。通过表与索引的监控，就可以大概确定该表与索引是否是热对象，并可以确认那些从来没有访问过的索引。

监控/取消监控表的具体代码如下：

```
alter table &table_name monitoring;
alter table &table_name nomonitoring;
```

监控/取消监控索引的具体代码如下：

```
alter index &index_name monitoring usage;
alter index &index_name nomonitoring usage;
```

如果想查看表的监控效果，可以查询 dba_tab_modifications 字典；如果想查看索引监控的效果，可以查询 v$object_usage 字典；如果表与索引数量较多，最好不要对所有表与索引进行监控，这样会增加系统的负载。

1.18　通过优化 SQL 消除 temp 表空间膨胀

经常有人问 temp 表空间暴涨的问题，以及如何回收临时表空间，由于版本不同，方法显然也是多种多样，但有很多方法都是治标不治本，只有深刻理解 temp 表空间快速耗用的原因，才能从根本上解决 temp ts 的问题。

是什么在使用 temp 表空间？索引创建、重建索引、order by、group by、distinct、union、intersect、minus、sort merge joins、analyze 等操作，包括 sort area 设置不当和异常，都会引起 temp 暴涨。

在处理以上操作时，DBA 需要加倍关注 temp 的使用情况，v$sort_segment 字典记载 temp 表空间比较详细的使用情况，而 v$sort_usage 字典将会告诉 DBA 是谁在做什么，具体的 SQL 还需要稍做加工，才可以得出更详细的结果。

```
select se.username,p.spid,su.blocks*8192/1024/1024 Space,sql_text
   from v$sort_usage su,v$session se,v$sql s,v$process p
where su.session_addr=se.saddr and s.hash_value=su.sqlhash
   and s.address=su.sqladdr and p.addr = se.paddr
order by se.username,se.sid;
```

其实大多数情况下 disk sort 都会在几秒内结束，如果在 sort 操作的若干秒内，刚好就捕获了该 SQL，那应该是巧合。即知道某个 SQL 将发生 disk sort 操作，当想捕获它们时，发现它们已经 sort 完了，排序完毕后 sort segment 会被释放。但很多时间，会遇到临时段没有被释放，temp 表空间几乎满的状况，这时该如何处理呢？

把 MetaLink 上推荐的方法收集整理如下。

⑴ 重启实例

重启实例时，SMON 进程会释放临时段，不过在很多时侯，实例是不允许 Shutdown，所以这种方法应用的机会不多；如果实例在重启后 sort 段没有被释放，那么这种情况就需要慎重对待，

目前还没有什么好的解决办法。

（2）修改参数（仅适用于 DMT 管理的表空间）

```
SQL>alter tablespace temp increase 1;
SQL>alter tablespace temp increase 0;
```

（3）合并碎片（仅适用于 DMT 管理的表空间）

```
SQL>alter tablespace temp coalesce;
```

（4）诊断事件（n=ts#+1,ts#可从 ts#中取 temp 的 ts#）

```
SQL>alter session set events 'immediate trace name DROP_SEGMENTS level n'
```

（5）重建 temp file

```
SQL>alter database tempfile '......' drop;
SQL>alter tablespace temp add tempfile '......' size ....m;
```

可以说，以上的方法都是治标不治本的，因为 temp 增长过快显然是由于 disk sort 过多造成的，造成 disk sort 的原因虽然很多，比如 sort area 较小等，这种情况是比较容易处理的，通过 Statspack 可以为 sort area 分配一个合适的值，通过增加 sort area 的值，确保 In-memory Sort 在 99%以上。

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          Buffer Nowait %:  100.00      Redo NoWait %:    99.99
          Buffer  Hit   %:  99.36    In-memory Sort %:   100.00
          Library Hit   %:  99.87        Soft Parse %:    99.84
       Execute to Parse %:   1.17          Latch Hit %:    99.96
Parse CPU to Parse Elapsd %:   92.00    % Non-Parse CPU:    94.59
```

如果 disk sort 是异常引起的，需要将该 sort 的进程 kill 掉，前面给的 SQL 中已经可以查出该 session 的 SPID，只需要用 “ kill -9 spid ” 命令即可；analyze 与 create index 等操作都可以安排在系统空闲时间，姑且不谈；其他的 disk sort 原因大多是 SQL 语句造成的，所以调优 SQL 才是治标治本的关键。

下面介绍一下 sort area 的分配。专用服务器 sort area 在 PGA（Program Global Area）中分配；共享服务器 sort area 在 UGA（User Global Area，UGA 在 Shared Pool 中分配）中分配。在 9i 以前的版本，由 sort_area_size 决定 sort area 的分配。在 9i 及以后的版本，当 workarea_size_policy = auto 时，由 pga_aggregate_target 参数决定 sort area 的大小，这时的 sort area 应该是 PGA 总内存的 5%；当 workarea_size_policy = manual 时，sort area 的大小由 sort_area_size 参数决定。无论是哪个版本，如果 sort area 设置得过小，In-memory Sort 率较低，那么 temp 表空间肯定会增长得很快，如果设置得较高，在 C/S 结构中将会导致内存消耗严重（长连接较多）。

由于 SMON 进程每隔 5 分钟要对不再使用的 sort segment 进行回收，如果不想让 SMON 回收 sort segment 的话，可以使用以下两个 event 写入初始化参数文件，然后重启实例，如果磁盘排序较多，那么很快就会涨暴磁盘……

```
event="10061 trace name context forever, level 10"  //禁止回收排序段
event="10269 trace name context forever, level 10"  //禁止合并碎片
```

合理地设置 PGA 或 sort_area_size 可以消除大部分的 dist sort，那其他的 disk sort 该如何处理呢？从 disk sort 的起因看，索引/分析/异常引起的 disk sort 应该是很少的；通常 SQL 中 distinct、union、group by 和 order by 以及 merge sort join 等引起的也较少；如果 SQL 进行了 disk sort，它的逻辑读、物理读和执行时间等都是比较突出的，所以可以对 v$sqlarea 或 v$sql 字典表进行过

滤，长期地监控数据库，相信可以把这些害群之马都找出来，然后想办法优化它们。从根本上降低 temp 表空间膨胀的方法有 2 个：设置合理的 PGA 或 sort_area_size；优化引起 disk sort 的 SQL。

1.19　理解 compress 选项在优化上的作用

关于表的 compress 属性，数据变动不频繁的 table、index 等对象可以启用数据 compress 属性而减少对 disk space 的需求。通过下面两个例子可以看出，compress 属性可以大幅缩减对 disk space 的占用。

```
SQL> create table compress_1 compress as select * from dba_objects;
SQL> create table compress_2 as select * from dba_objects;
SQL> exec show_space('compress_1','auto');
Total Blocks.....................64
Total Bytes.....................524288
Unused Blocks.....................6
Unused Bytes.....................49152
Last Used Ext FileId...............5
Last Used Ext BlockId..............544
Last Used Block...................2
SQL> exec show_space('compress_2','auto');
Total Blocks.....................256
Total Bytes.....................2097152
Unused Blocks.....................110
Unused Bytes.....................901120
Last Used Ext FileId...............5
Last Used Ext BlockId..............11656
Last Used Block...................18
```

好像 show_space 过程来自以下链接 http://asktom.Oracle.com，对 dbms_space 包的调用更好地将 table/index 的存储信息展示出来，这个帖子（http://www.itpub.net/239697.html）详细地讨论了 show_space 过程的使用方法，而且还有些网友对这个过程进行了改进，希望对大家有所帮助。

1.20　关于在线重定义 table 的建议

在线重定义表是 Oracle 9i 的新功能，可以在系统繁忙的情况下，对表进行重组，消除表中的碎片，不仅可以减少 SQL 的逻辑读及物理读，还可以防止产生修改表结构时的 DDL 锁（Library Cache Pin/Library Cache Lock）等。piner 网友写了一个关于在线重定义表的帖子（http://www.cnoug.org/viewthread.php?tid=1680），它非常详细地介绍了在线重定义表的过程，这里就不多说了，只简单地介绍一下本人使用在线重定义表的经验与教训。

表经过重定义以后，发现以前有一些允许为 null 的字段被 dbms_redefinition package 更改为 not null，结果有一些 insert 报错，因为 SQL 没有为 not null 的字段提供数据。还有一个非常重要的问题：在线重定义过程中，字段的默认值可能会失踪，结果也会造成与上面字段的属性由 null 改成 not null 一样的后果。所以使用在线重定义表时一定要小心，确信在线重定义表的过程没有改变不希望更改的部分。

1.21 关于分区表在数据库设计时的建议

如果表中预期的数据量较大而且该表的访问比较频繁，那么通常该表都需要考虑使用分区表，确定使用分区表后，还要确定什么类型的分区（如 Range Partition、Hash Partition、List Partition 等）和分区区间大小等。随着存储设备价格的下降，RAID 技术渐渐地在一些较低档的服务器上使用，分区表在性能上的好处越来越少，但它在数据库维护方面的好处就越来越明显起来。分区表的好处如下。

- 高可用性：表某分区故障而不能使用，表的其他分区仍然可以使用。
- 维护方便：独立管理每个分区比管理单个大表要轻松得多。
- 改善性能：对大表的查询、增加、修改等操作可分解到表的分区来并行执行。

分区的创建最好与程序编写有某种默契，如果创建分区表，例如是按自然月份定义分区的，那么程序处理查询时，应尽量控制查询在单个分区内完成，否则该分区表并不能提高该查询的性能，具体可参考本人的博客（http://blog.itpub.net/post/96/11476），只有在结合分区键的查询时，分区表才能发挥其性能。关于分区表常用的维护操作可参见帖子 http://www.itpub.net/221061.html。

1.22 关于 DataGuard 在高可用方面的建议

DataGuard 是 Oracle 推出的一种高可用性方案（8i 及以前的版本称为 Standby），从 Oracle 9i 开始，DataGuard 支持逻辑备用库，关于 DataGuard（Physical Standby Database）的配置，可以参考 piner 网友的帖子（http://www.cnoug.org/viewthread.php?tid=22640）。

Physical Standby Database 在主节点与备用节点间通过日志同步来保证数据的同步，备用节点作为主节点的备份，可以实现快速切换与灾难性恢复，物理备用库最大限度地保障数据安全，减少系统停机时间；Logical Standby Database 在主节点与备用节点间也通过日志同步来保证数据的同步，备用节点在日志应用的同时，可以提供查询等操作。

DataGuard 不仅在数据安全、高可用等方面有很突出的意义，同时也可以减轻主生产库的压力，即可以承担部分报表的任务；如果有足够的机器，建议主生产库配置一个 Physical Standby Database 和一个 Logical Standby Database，如果只有一台机器的话，那就配置成 Physical Standby Database 为好。

众所周知，Physical Standby Database 通常在 managed recover 状态是不能进行数据查询的，如果想提供数据查询，Database 就必须处于 open read only 状态；如果 Database 在 open read only 状态，当然不可以再进行实时归档恢复，但在 open read only 状态下，并不妨碍主生产库将归档日志源源不断地传送到备用库，通常大的报表对当天的统计并不是十分关注，所以可以把传送到备用库的归档日志安排到凌晨进行恢复。

通过 crontab 在每天凌晨 3:00 点钟开始运行备用库进行恢复的 shell，然后在凌晨 5:00 点钟用 crontab 把 managed recover 模式取消，再将数据库打开到 open read only 状态，这样 Physical Standby Database 在高可用方面稍有降低的情况下，可以同时提供一部分报表业务，这样将在很大程度上减轻主生产库的压力。对 Logical Standby Database 进行切换的 shell 放在本人的博客

(http://blog.itpub.net/post/96/13603) 上。

作者简介

谢中辉，曾任 ITPUB MS SQL Server 版版主，现任 ITPUB Oracle 开发版版主。

曾经任职于广州大型港资纺织漂染企业，开发过基于 Oracle 的 ERP 系统。在数据库建模、备份恢复、系统调优、SQL 优化、数据迁移、PL/SQL 开发等方面有丰富的经验。目前任职于某大型电子商务网站，负责数据库的故障诊断处理、异构数据迁移、新业务实现规划、系统调优与 SQL 优化等工作。

2 第2章 Statspack高级调整

2005年2月初在美国芝加哥的郊区，我有幸会晤了著名Oracle数据库大师、任The Ultimate Software Consultants（TUSC）顾问公司CEO的Richard J. Niemiec先生，他在数据库性能调整和优化方面有着极深的造诣和举世瞩目的成就，同时他著述颇丰，其得意之作《Oracle Performance Tuning Tips and Techniques》一书在美国几年来连续高居性能调整类书目畅销之首。

亲聆大师教诲，是我多年来的心愿。我抓紧宝贵的时间向他请教了所面对的、也是中国市场上常见的RAC、10g和Oracle绑定Veritas等方面的问题，而他在回答时所表现出来的对Oracle产品的深刻理解、入手分析具体问题的广阔思路和理论联系实际的丰富经验，让我一领世界级大师之风范。临别之际，大师嘱咐我向中国同行们致意，并委托我把他的心爱之作——《Statspack高级调整》带给大家。

仔细读来，此文既具深度又不失广度，短短几页纸却覆盖了实际工作中的大部分内容，使我颇有得来恨晚之感。所以，我仔细地读，仔细地译，仔细地整理。经与Richard越洋商讨并获他同意后，我又重写了本章中关于Latch基本概念和工作方式的部分内容。

行文至此，希望各位同行喜欢这篇译文并在工作中有所应用和发挥，那将是对大师最好的告慰，也将是我这位“使者”最高兴的事情！

译者　吕学勇

以下是Richard J. Niemiec 先生对本译文的评价：

“I want to give a heartfelt thanks to Lu Xueyong for translating this article. His ability in Oracle has taken the original article and made it even better. His tremendous Oracle skills are only exceeded by his character and dedication to helping share the knowledge in China. Thank you Lu for your tremendous contribution and dedication to character! TUSC has always been a company in the U.S. that has been dedicated to sharing the knowledge as well as a dedication to doing work with character, I've been lucky to work with Lu who is dedicated to the same in China!”

2.1 Statspack 高级调整译文

在 Oracle 9i（及 8.1.6）中，如果允许我随意挑选自己喜爱的、在系统监督和性能问题定位方面的工具，但同时又限制最多只能选两个，那么我的首选一定是 Statspack 和 Enterprise Manager。Statspack 代替了 Oracle 先前各版本中所提供的 UTLBSTAT/UTLESTAT 脚本并对这两个脚本的功能进行了重要扩充。下面的行文中我将着重谈谈以 Statspack 施行 Waits 调整时一些高难问题的解决办法。着笔写出这些问题使我感觉颇费心血，初学的人读起来可能会感觉到困难和一头雾水，所以我建议有兴趣的初学者还是从阅读 Oracle 的相关文档入手为宜。

2.1.1 Top 5 Wait Events

当试图快捷地找到并消除系统瓶颈时，Top 5 Wait Events 可能就是整个 Statspack 报告中最能披露问题的一部分内容了。这部分内容将列出 Top 5 的等待事件、全体等待事件和后台等待事件，而识别其中的主要等待事件往往可以帮助解决系统调整方面的燃眉之急。如果 TIMED_STATISTICS 已经设置为 true，则等待事件会按照等待时间的长短来排序；如果没有，则它们会按照等待发生的次数来排序。

从下面的列表中可以发现大量有关读单一数据块的等待（DB File Sequential Reads）和有关 Latches 的等待（Latch Free），同时可以发现很严重的写数据文件和写日志文件的等待以及日志文件的竞争。为了定位和解决主要的问题，往往需要仔细研究 Statspack 其他段落中所提供的更为详细的报表。

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                        Wait    % Total
Event                                        Waits     Time (cs)   Wt Time
---------------------------------------- ------------ ------------ -------
db file sequential read                 18,977,104    22,379,571      82.29
latch free                               4,016,773     2,598,496       9.55
log file sync                            1,057,224       733,490       2.70
log file parallel write                  1,054,006       503,695       1.85
db file parallel write                   1,221,755       404,230       1.49
```

现在列出一些最为常见的问题、对它们的解释以及通常可行的解决方法。

- DB File Scattered Read

这通常表明存在着与全表扫描相关的等待。在内存中进行的全表扫描一般是零散地、而非连续地被分布到缓冲区的各个部分。如果此项数值很大则说明可能有索引丢失或被抑制索引的存在。不少情况下我们其实更希望全表扫描，因为它往往比索引扫描效率更高，所以当发现此类等待事件时，应该首先确认是否真的需要全表扫描。在表很小的情况下则可以尝试把它们整个地装入到内存缓冲区中，以避免反复地从磁盘上读取它们的内容。

- DB File Sequential Read

这是与单数据块读（譬如索引读）相关的指标。巨大的值暗示着可能存在低效的表连接顺序或者选择性差的索引扫描。在一个调整得很好的多事务系统中该项数值通常很大，所以最好把这项等待指标同 Statspack 报告中出现的其他众所熟知的问题（如低效 SQL）关联起来研究，譬如说检查索引扫描是否必要，又譬如说检查多表连接中的连接顺序，等等。参数 DB_CACHE_SIZE

的值是这项等待出现多少的决定性因素。哈希连接所造成的问题将出现在 PGA 内存中，在“ 疯狂吞噬 ” 大量内存的同时导致按顺序读取的严重等待或直接读/写的等待。

- Free Buffer Waits

此一项表明系统因目前没有可用的内存缓冲区而等待。在全部 SQL 语句都调整完好的情况下，这一项等待告诉您应该去增大 DB_BUFFER_CACHE。该项等待也可能是由于当前 SQL 语句的选择性太差而造成的：内存缓冲区被大批索引块占满，系统一时找不到处理该语句所必需的更多内存。大量 DML（增、删或改）或者 DBWR 写速度不够快也可能造成该项等待，内存缓冲区内可能充斥着大量重复的内容，造成严重低效。为解决这类问题，可以考虑加速增量检查点、使用更多 DBWR 进程或者增加物理磁盘的数目。

- Buffer Busy Wait

此类对缓冲区的等待是由于该缓冲区的非共享工作方式，或者是由于该缓冲区正在通过其他会话读取数据块中的内容所致。缓冲区忙等待不应超过 1%的额度。请检查 Statspack 的缓冲区等待的部分（或者检查 v$waitstat 视图），看看等待是否发生于段头上。若是，则应增加该段的 Freelist Groups 或者增大 PCTFREE 和 PCTUSED 之间的差。如果等待发生于 undo header 上，则可以通过增加回滚段来解决问题；如果等待发生于 undo block 上，则可以考虑减少驱动此一致性读的表上面的数据密度，或者增大 DB_CACHE_SIZE。如果等待发生于 data block 上，则可以把该块上的数据搬到另一数据块上以避免“ 热点 ”，增大表的 Freelists 或者采用本地管理的表空间（LMT’s）。如果等待发生于 index block 上，则可以重建索引、将索引分区或者采用反向关键字索引。为避免与数据块相关的缓冲区忙等待，可以转而采用较小的 blocksize：每一块里现在只能容纳较少的记录，该块也就不会再像以前那样“ 热 ” 了。当执行一条 DML（增、删或改）时，Oracle 数据库会向数据块里写信息，其中包括对该数据块状态“ 感兴趣 ” 的用户信息（Interested Transaction List，简称 ITL）。为减少在这一部分上的等待，可以增大 initrans 参数，从而在数据块中为更多的 ITL 槽预留空间；还可以增大表的 PCTFREE 参数，这样，在 initrans 没有为足够多 ITL 槽预留空间的情况下，更多 ITL 槽（受 maxtrans 的封顶限制）可以被动态地分配给空间。

- Latch Free

Latches 是一种低级的同步锁机制，用以维持某些访询和执行操作的顺序。Oracle 服务器通过 enqueue 来达成对重做线程、表和事务一类对象的并发使用，而通过 Latch 来达成对 System Global Area（SGA）中的共享内存结构的保护。Latch 速度快而成本低，往往通过单个的内存单元来实现。大部分的 Latch 是互斥型的，它们可以赋予某些单一进程写的权限。共享型 Latch 允许对某内存结构的并发读操作。当请求得到 Latch 但它已被其他进程占住时，将产生一条 Latch free miss 的记录。大多数 Latch 问题关联到不使用绑定变量（Library Cache Latch）、重做日志生成中问题（Redo Allocation Latch）、内存缓冲区竞争问题（Cache Buffers LRU Chain）和内存缓冲区中“ 过热 ” 的数据块（Cache Buffers Chain）。也有一些 Latch Waits 是由于软件错误而造成的，如果您怀疑自己遇到了这样的错误，则可以到 MetaLink 上查看错误报告（oracle.com/support）。当 Latch 错失率超过 0.5%时，就应当去仔细研究研究了。我本人将为近期的 Oracle Magazine 撰文详谈 Latch 等待的问题，因为这个题目确实需要专门的一篇文章才能讲得比较清楚。

- Enqueue

enqueue 是一种保护共享资源的锁，锁可以保护如记录里面数据一类的共享资源，防止两个

人同时修改该记录。enqueue 结构包含有一种先进先出（FIFO）的队列机制，这一点上有别于 9i 版之前 Oracle Latch 的非先进先出机制。enqueue 等待通常是就 ST enqueue、HW enqueue 和 TX4 enqueue 而言。ST enqueue 在数据字典管理方式表空间的物理空间分配和管理上发挥作用。当某些以数据字典管理的表空间不断出现问题时，可以转而采用本地管理的表空间 （LMT），或者预先分配扩展，再或者至少使下一扩展增大一些。HW enqueue 同段的高水位线一起使用，手工分配扩展可以避免其上的等待。TX4 是各种 enqueue 等待中所最为常见的，它的出现通常是由下述三种问题之一所造成的结果。第一种，惟一索引的重复，您需要 commit/rollback 以释放 enqueue。第二种，多个并发的对同一位图索引片的修改。因为一个位图索引片内可能有多个 rowids，当多位用户试图修改同一片时，就需要 commit/rollback 来释放 enqueue。第三种，也是最可能发生的一种，多用户同时修改相同的数据块，而如果已经没有空闲的 ITL 槽，则会引发一数据块级的锁。运用增大 initrans 或 maxtrans 以容纳更多 ITL 槽的方法，可以解决这个问题，增大表的 PCTFREE 也可以解决这个问题。现在来谈一谈 TM 锁——一种行级锁。如果有外键，则务必要为它们创建索引以避免这种常见的锁的麻烦。

- Log Buffer Space

这种等待发生在以比 LGWR 把日志缓冲区的内容写进重做日志文件更快的速度来写日志缓冲区的情况下，或者发生在日志切换太慢的情况下。为解决此问题，可以增大重做日志文件、增大日志缓冲区，或者使用更快的磁盘。

- Log File Switch

所有关于提交的请求都在等待“ logfile switch（archiving needed）”或者“ logfile switch（chkpt. Incomplete）”。此时应当确认用于归档的磁盘是否已经满了或者是否太慢。由于 I/O 的关系 DBWR 可能太慢，就可能得增加一些大容量的重做日志。不过，如果 DBWR 才是问题所在，那就得转而考虑增加数据库的写进程数目。

- Log File Sync

用户提交或者回滚时，LGWR 将本会话的 redo 从日志缓冲区迅速地写进日志文件中，用户进程必须等待写的成功完成。为缩短等的时间，可以每次提交更多的记录（譬如说，不是逐条记录地提交，而是每 50 条一批地提交）。可以考虑把重做日志放在快速的磁盘上，或者把不同的重做日志放在不同的物理磁盘上，而这些都是为了减小归档对 LGWR 的影响。尽量不要采用 RAID 5，因为对于有大量写操作的应用来说 RAID 5 太慢了。应当尽量采用文件系统直接 I/O 或者裸设备，裸设备写起来确实快。

- Idle Events

在等待列表的最下面会看到一些空闲等待事件，您可以忽略这些事件。空闲事件通常被列在每一段落的最下部分，如发往/来自客户端的 SQL*Net 消息和其他后台相关的定时信息，等等。空闲事件被放在 stats$idle_event 表里。

2.1.2 等待时间快捷参考

把等待问题和可能的解决方法总结如表 2-1 所示。

表 2-1　　等待问题及可能解决方法

等待问题	可能的解决方法
Sequential Read	表明有很多索引读——调整代码（特别是表连接部分）
Scattered Read	表明有很多全表扫描——调整代码、将小表放入内存
Free Buffer	增大 DB_CACHE_SIZE、加速检查点和调整代码
Buffer Busy	段头——增加 freelists 或者 freelist groups
Buffer Busy	数据块——分离“热点”数据、采用反向关键字索引、采用小的数据块
Buffer Busy	数据块——增大 initrans 和 maxtrans
Buffer Busy	undo header——增加回滚段
Buffer Busy	undo block——增加提交频度、增大回滚段
Latch Free	研究 Latch 细节（可以参考下文）
Enqueue - ST	使用本地表空间或者预先分配大扩展
Enqueue - HW	预先分配扩展于高水位线之上
Enqueue - TX4	增大表或索引的 initrans 和 maxtrans
Enqueue - TM	为外键建立索引，查看应用程序中的表锁
Log Buffer Space	增大日志缓冲区，重做日志放在快速磁盘上
Log File Switch	归档设备太慢或者太满，增加或者扩大重做日志
Log File Sync	每次提交更多记录、更快的存放重做日志的磁盘、裸设备
Idle Event	忽略

常见的空闲事件包括以下：

- dispatcher timer（共享服务器空闲事件）；
- lock manager wait for remote message（RAC 空闲事件）；
- pipe get（用户进程空闲事件）；
- pmon timer（后台进程空闲事件）；
- PX Idle Wait（并行查询空闲事件）；
- PX Deq Credit: need buffer（并行查询空闲事件）；
- PX Deq Credit: send blkd（并行查询空闲事件）；
- rdbms ipc message（后台进程空闲事件）；
- smon timer（后台进程空闲事件）；
- SQL*Net message from client（用户进程空闲事件）；
- virtual circuit status（共享服务器空闲事件）。

2.2　关于 Latch

关于 Latch 究竟是什么、它的用法和所遇到的问题等，前面的 Latch Free 一部分中已经做了基本的介绍，这里不再重复。

有两种基本的 Latch 请求方式：“ Willing-To-Wait ”和“ No-Wait ”。在前一种方式下，如果被

请求的 Latch 忙，则请求得到 Latch 的那个进程，比如说进程 A，将在旋转一阵后重新请求得到 Latch，如是循环直到重复_SPIN_COUNT 所确定的值那么多次为止。在单 CPU 系统中，_SPIN_COUNT 被设置为 1，因为在没有另外 CPU 释放占据着 Latch 的进程的情况下，让请求得到 Latch 的进程空转，除了耗费一些 CPU 资源外，并看不出其他的意义。在 CPU 数>1 的情况下，_SPIN_COUNT 的缺省值为 2000。

如果在整个等待循环之后，Latch 仍然被其他进程占据着，进程 A 就会让出 CPU 并进入睡眠状态，而一次睡眠对应于 Latch Free 等待事件中的一次等待。参数_MAX_EXPONENTIAL_SLEEP 决定了进程 A 被获准再次请求得到 Latch 前睡眠状态的最长时间，该参数的缺省值是 200 厘秒（1 厘秒 = 1/100 秒）。初次的睡眠时间是 1 厘秒，第二次时增长 1 倍到 2 厘秒，而且在从第二次算起的接下来的各次请求中，每 2 次循环后睡眠时间就会加倍，形成 1、2、2、4、4、8、8……的睡眠时间模式，直至达到_MAX_EXPONENTIAL_SLEEP 所确定的上限。

在两种情况下一个进程，比如说进程 B，可按“ No-Wait ”的方式请求得到 Latch，在此仅介绍看起来比较简单的一种。以 Redo Copy Latch 为例，因为同种的 Latch 不止一个，进程 B 在请求得到 Redo Copy Latch 时就会按“ No-Wait ”的方式开始：在一个 Latch 已经被占用的情况下，进程 B 并不会等着再次请求它，而是转而立即去请求得到另一个 Latch。仅仅是在请求过所有这类 Latch 而都没有能够得到的情况下，才会由服务器进程确定这种 Latch 中的一个让进程 B 去等待。“ No-Wait ”Latches 为 v$latch 视图中的 immediate_gets、immediate_misses 列和 Statspack 报告中与 Latch 相关的部分生成信息。

定位了可能有问题的和 Latch 相关的等待事件后，在 Statspack 报告中的 Latch Activity 部分将可以得到关于这些 Latch 的更具体的信息。Get Requests、Pct Get Miss 和 Avg Slps/Miss（关于睡眠和错失）是针对“ Willing-To-Wait ”Latches 请求的统计，而 NoWait Requests 和 Pct NoWait Miss 则是针对“ No-Wait ”Latches 请求的。相对于两种 Latch 请求的 Pct Miss 都应该接近于 0.0。

Statspack 报告中的 Latch Sleep Breakdown 部分提供了有关 Latch 的详细信息：请求这些 Latch 的进程正在旋转或睡眠。

分析 Latch 问题时，v$latch 视图十分有帮助，v$latchholder、v$latchname 和 v$latch_children 等视图也都很有帮助。

查看上述 Statspack 报告中关于 Latch 的几节或者查询 v$latch 视图，可以了解到有多少进程不得不等待（Latch Miss）或睡眠（Latch Sleep）以及它们不得不睡眠的次数。下面提供的是 Statspack 报告中 Latch Activity 段落的一部分，其中的库缓存 Latch 显然有问题（库缓存 Latch 是“ Willing-To-Wait ”Latch 的一例）：

```
                                  Pct        Avg        Wait         Pct        Get
                                  Get       Slps        Time      NoWait     NoWait
    Latch                     Requests       Miss       /Miss         (s)
                              Requests       Miss
------------------------ ------------- ------ ------ ------ ------------ ------
KCL freelist latch               4,924        0.0                              0
cache buffer handles           968,992        0.0         0.0                  0
cache buffers chains       761,708,539        0.0         0.4   21,519,841     0.0
cache buffers lru chain      8,111,269        0.1         0.8   19,834,466     0.1
library cache               67,602,665        2.2         2.0      213,590     0.8
redo allocation             12,446,986         0.         0.0            0
redo copy                          320        0.0                10,335,430     0.1
user lock                        1,973        0.3         1.2            0
```

当 Statspack 报告中，等待事件一节里的“ Latch Free ”项显示很高的数值时，在报告中关于 Latch 的段落内一定可以发现需要研究的问题。下面的内容将帮助您查看这些有关 Latch 的问题。

- Library Cache and Shared Pool

库缓存 Latch 为对库中对象的访问请求排队，每当执行 SQL 或 PL/SQL 存储过程、包、函数和触发器时，这个 Latch 即被用到。Parse 操作中此 Latch 也会被频繁使用。Oracle 8i 是以一个共享池 Latch 来保护库缓存的内存分配，但自 9i 始，增加了 7 个子 Latch。共享池太小或 SQL 语句不能重用时，会发生“ Shared Pool ”、“ Library Cache Pin ”或“ Library Cache ” Latches 的竞争。SQL 语句不可再用往往是因为未使用绑定变量，共享池里随处可见很相像但又不完全一样的 SQL 语句，而增加池的大小只会把 Latch 的问题搞得更糟。可以设置初始化参数 CURSOR_SHARING=FORCE（9i 中大同小异）以减少绑定变量未使用的问题。然而，共享池和库缓存的 Latch 问题在后者对需处理的大批 SQL 语句而言设置太小、需要分配更多内存的情况下也会发生。欲将 SQL 或 PL/SQL 语句装入内存而首先释放部分空间的操作会互斥地占据 Latch，令其他用户等待。可以通过增大共享池来减缓此种竞争，还可以通过使用 DBMS_Shared_Pool.Keep 存储过程在内存中固定大的 SQL 或 PL/SQL 语句来解决这个问题。

- Redo Copy

重做拷贝 Latch 的数量在缺省情况下是 2*CPU_COUNT，但可通过 _LOG_SIMULTANEOUS_COPIES 的初始化参数来重新设置。增大此参数可以帮助减缓对重做拷贝 Latch 的竞争。重做拷贝 Latch 用来从 PGA 向重做日志缓冲区拷贝重做记录。

- Redo Allocation

对 Redo Allocation Latch（分配重做日志缓冲区中的空间）的竞争可以通过选用 NOLOGGING 的选项来减缓，该选项可以减轻日志缓冲区的负荷。另外，应当避免不必要的提交。

- Row Cache Objects

“ Row Cache Objects ”Latch 的竞争通常意味着数据字典的竞争，它同时也可能是过度 parsing 依赖于公共同义词 SQL 语句的症状。增大共享池一般可以解决此问题。这个办法往往用来解决库缓存 Latch 的问题，而在那个问题解决好了的前提下，“ Row Cache Objects ” Latch 的竞争通常根本就不会成为一个问题。

- Cache Buffers Chains

扫描 SGA 中的内存缓冲区时需要内存缓冲区链 Latch。内存缓冲区中“ 过热 ”的数据块（通常正在被访问）造成了内存缓冲区链 Latch 的问题，“ 过热 ”的数据块同时也可能是调整差的 SQL 语句所导致的症状。“ 过热 ”的记录创建了“ 过热 ”的块，也就导致了块里其他记录的问题和映射到该块地址上的 hash 链的问题。为了定位“ 热块 ”，可以查询 v$latch_children 视图来确定块地址，并且通过该视图与 x$bh 视图的连接来确定此 Latch 保护下的所有数据块（此将确定受“ 热块 ”影响的全部数据块）。以在 x$bh 视图里找到的 file#和 dbablk，可以进而查询 dba_extents 以确定受影响的对象。如果“ 热块 ”是在某索引上，那么转而采用反向关键字索引即可把连续排放的记录搬到其他数据块中，它们也就不再会被“ 热块 ”一连串地锁住了。如果“ 热块 ”是索引的“ 根 ”块，那么反向关键字索引就帮不上忙了，把_DB_BLOCK_HASH_BUCKETS 设置成比缓冲区数（DB_CACHE_SIZE/DB_BLOCK_SIZE）的 2 倍大的最小质数的值通常能够清除此问题的干扰。Oracle 9i 之前，此参数有一个缺省值，但导致了这个 Latch 的非常严重的竞争问题。在 Oracle 9i 中，此参数被正确地设置为一个质数。

- Cache Buffers LRU Chain

扫描全部内存缓冲区块的 LRU（最近最少使用）链时要用到内存缓冲区 LRU 链 Latch。太小的内存缓冲区、过大的内存缓冲区吞吐量、过多的内存中进行的排序操作、DBWR 速度跟不上工作负载等全都可能成为导致内存缓冲区 LRU 链 Latch 严重竞争的罪魁祸首：请调整导致过量逻辑读的查询！可以增大初始化参数 DB_BLOCK_LRU_LATCHES 以得到多个 LRU Latches，如此即可减少竞争。一般说来，非 SMP（对称多处理器）系统仅需要惟一的 LRU Latch，而在 SMP 系统的情况下，Oracle 自动地把 LRU Latch 的数目设置成 1/2 CPU 数。对于一个数据库写进程，需要分配给它至少一个 LRU Latch，如果需要增加数据库写进程的数目，则不能忘记同时增加 LRU Latch 的数目。

把 Latch 问题及其可能的解决方法总结如表 2-2 所示。

表 2-2　　Latch 问题及可能解决问题

Latch 问题	可能的解决方法
Library Cache	使用绑定变量，调整 SHARED_POOL_SIZE
Shared Pool	使用绑定变量，调整 SHARED_POOL_SIZE
Redo Allocation	最小化 redo 生成并避免不必要的提交
Redo Copy	增大_LOG_SIMULTANEOUS_COPIES
Row Cache Objects	增大共享池
Cache Buffers Chain	_DB_BLOCK_HASH_BUCKETS 应被增大或变为质数
Cache Buffers LRU Chain	设置 DB_BLOCK_LRU_LATCHES 或者使用多个缓冲区池

任何命中率低于 99%的 Latch 都应该仔细检查。在过去，有些 Latch 问题是由于软件错误而造成的，所以应该记得到 MetaLink 上去查看同 Latch 相关的议题。

本章详细讲述了一些最为常见的 Latch 问题，包括内存缓冲区链、重做拷贝、库缓存和内存缓冲区 LRU 链等的 Latch 问题。

参考信息

1. Steve Adams. Oracle 8i Internal Services for Waits, Latches, Locks, and Memory. O'Reilly UK,2003
2. Oracle Doc ID: 61998.1, 39017.1
3. Connie Dialeris, Graham Wood. Performance Tuning with Statspack White Paper, 2000
4. Notes from Richard Powell, Cecilia Gervasio, Russell Green and Patrick Tearle
5. Statspack checklist; Kevin Loney, Randy Swanson, Bob Yingst, 2002
6. Rich Niemiec, IOUG Masters Tuning Class, 2002
7. Richard J.Niemiec. Oracle Performance Tuning Tips and Techniques. McGraw-Hill. 1999
8. Richard J.Niemiec. Oracle 9i Performance Tuning Tips and Techniques. McGraw-Hill. 2003

作者简介

Richard J.Niemiec, TUSC(The Ultimate Software Consultants)公司 CEO, 著有《Oracle Performance Tuning Tips and Techniques》一书。

译者简介

吕学勇，20 世纪 80 年代在美国攻读组合算法，80 年代后期起在 AT&T 贝尔实验室、芝加哥期货交易中心等地从事关系型数据库方面的工作，1996 年起专职任 Oracle DBA/开发员以及后来的高级顾问。2001 年回国服务，现任 UT 斯达康公司高级顾问。

第3章　Statspack使用的几个误区

编者按：Statspack是数据库优化中经常用到的工具，《Oracle数据库DBA专题技术精粹》一书中已包含了《Statspack使用指南》一文，大家可以从ITPUB上找到相关的电子文档作为参考。本章从一些常见的使用误区入手，对Statspack的使用进行了独到的阐述。

Oracle提供的Statspack工具已经成为许多DBA性能调整的好帮手。每个DBA可能都或多或少用过这个工具。Statspack收集V$视图的历史信息，并将其存入到单独的数据库表中，每次收集的信息称为一个快照（snapshot），通过对比快照间的数据给出相应的报告。Statspack是随数据库软件分发的，不需要额外付费。Statspack 收集的信息比较全面，给DBA提供了详实的数据库信息。

虽然 Statspack 有着很多优点，但是也有其局限性——不能给出数据库性能的问题所在，需要人工对数据进行分析，Statspack也不能给出解决问题的办法，DBA必须能够根据给出的报告提供相应的解决方法。尽管现在网络上关于 Statspack 的文档已经很多了，但是在交流中发现很多朋友对这个工具的运用还存在着问题。下面就其中比较容易出问题的几个方面进行分析和讨论。

3.1　以命中率为主衡量性能问题

在Instance Efficiency Percentages一节列出了很多比率。在Oracle 7.x 时代的DBA可能对DBHR（Data Buffer Hit Ratio）最感兴趣了。很多人一看到Buffer Hit的值比较低，第一个反应就是加大Buffer Cache的值。经过“调优”之后再查看数据库的Buffer Cache命中率，发现命中率上来了，可是用户依然报告数据库很慢。

要知道如果过于强调“比率（Ratio）”，那么就意味着会损失很多有价值的信息。DBHR也是这样。一个Buffer Cache命中率为20%的库性能很糟糕么？答案是：不一定。如果Oracle数据库是一个数据仓库系统，那么20%的系统可能性能很好。相反，DBHR为99.9%的数据库性能会很好么？答案还是不一定，如果因为某些特定的语句生成了大量的逻辑I/O（LIO）调用，那么DBHR或许看起来很“不错”。对于这种情况，来看一个例子：

```
                                                  CPU     Elapsd
     Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
```

```
-------------- ------------ -------------- ------ -------- --------- ----------
   15,221,088       4311       35,30.7   85.9   109.86    108.13 3159590556
    SELECT COUNT (*) FROM (SELECT f2.y FROM foo1 f1, foo2 f2 WHERE f1.x = f2.x)
```

这是从一份 Statspack 报表的 SQL ordered by Gets 一节中摘出的内容（为了节省篇幅，这里只列出最主要的一句 SQL）。在这个例子中，这条 select 语句在两个快照间隔期间执行了 4311 次，一共有 Buffer Gets（LIO）15,221,088 次，平均每次执行产生了 3530.7 个（15,221,088/ 4311 的结果）LIO，占总的 Buffer Gets 的 85.9%。可以看出在两个快照间隔中，这条语句是必须注意的。

在系统中通过设定 Event 10046 对该语句进行 Trace 分析，得到的分析数据如下：

```
select count(*)
from (
 select /*+ NO_MERGE ORDERED FULL(f1) INDEX(f2 ix1) USE_NL(f2) */ f2.y
 from foo1 f1, foo2 f2
 where f1.x = f2.x )

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         2     34.59     33.76         0     440025         0          1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total         4     34.59     33.76         0     440025         0          1

Misses in library cache during parse: 0
Optimizer goal: FIRST_ROWS
Parsing user id: STAT

Rows     Row Source Operation
-------  ---------------------------------------------------
     1  SORT AGGREGATE (cr=440025 r=0 w=0 time=33763363 us)
  1209   VIEW  (cr=440025 r=0 w=0 time=33762090 us)
  1209    NESTED LOOPS  (cr=440025 r=0 w=0 time=33759692 us)
  10000    TABLE ACCESS FULL FOO1 (cr=24 r=0 w=0 time=19901 us)
  1209      TABLE ACCESS FULL FOO2 (cr=440001 r=0 w=0 time=33708307 us)
```

其中 query 加上 current 是 LIO 的数值。这个 SQL 中，共产生了 440025（440025+0）的 LIO。从上面的 Trace 文件中可以知道，这个 SQL 没产生 PIO。

在这里，有意强制性地让该语句走嵌套循环的执行计划，有意引导“增加”Buffer Gets 数值。就这个例子来说，如果以更高的频率运行这个 SQL，很快数据库的 Buffer 命中率就会提升到 95% 甚至更高，但是这能代表数据库性能良好么？很明显，不能！可以看到，该语句造成了过高的 LIO，而过高的 LIO 又导致系统竞争加大，从而会导致数据库性能严重下降。这和优化数据库的初衷是南辕北辙的。

同样，看看在合理的执行计划下该语句的开销：

```
select count(*)
from (
 select /*+ FULL(f2) FULL(f1) */ f2.y
 from foo1 f1, foo2 f2
 where f1.x = f2.x )

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       2     0.13      0.12         0         67         0          1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       4     0.13      0.12         0         67         0          1

Misses in library cache during parse: 0
Optimizer goal: FIRST_ROWS
Parsing user id: SYS

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=67 r=0 w=0 time=122848 us)
   1209   HASH JOIN  (cr=67 r=0 w=0 time=121743 us)
  10000    TABLE ACCESS FULL FOO1 (cr=23 r=0 w=0 time=10827 us)
  20000    TABLE ACCESS FULL FOO2 (cr=44 r=0 w=0 time=21011 us)
```

经过优化之后查询走了 HASH JOIN 链接方式。LIO 只有 67（query+current= 67+0=67）。

注　意

逻辑 I/O（LIO）与物理 I/O（PIO）

LIO（Logical I/O）是指数据库核心对 Buffer Cache 中的数据块的读取操作。如果要获取的对象在 Buffer Cache 中不存在，则 LIO 可能会产生 PIO。v$sesstat 视图中的 db block gets 和 consistent gets 给出关于特定会话的 LIO 的汇总信息。consistent gets 代表对特定版本或时间的 block 的访问 select；db block gets 代表对最新的或当前的 block 的访问，通常用于 insert、update 和 delete 操作。

PIO（Physical I/O）不能从 Buffer Cache 中得到的块读取操作，可能是因为块不存在或者是因为是跳过 Buffer Cache 的直接 I/O 类型的 I/O 操作。可以简单地理解为 PIO 就是磁盘读取。可以从 v$sesstat 中得到名为“Physical Reads”的统计信息。

3.2　快照的采样时间间隔问题

Statspack 的报告实际上也就是对比两个快照（也就是数据库当前状态）得出的结果。

一般情况下，对于一个 OLTP 系统，专家建议生成 Statspack 报告的快照时间间隔为 15～30 分钟。

试想，一个人去医院看病，医生对其测量体温，一般也就是 5～10 分钟左右就可以了，为什么是这么长的时间？因为 5～10 分钟这段时间基本可以近似地得到病人的体温。如果时间过短，可能达不到既定的目的，测到的体温会偏低，时间过长，甚至长达几个小时的话（假设有这种情况），病人可能都昏迷几次了。

对生成 Statspack 报告的快照时间间隔也是这样，如果两个快照时间间隔过短，数据库的一些重要周期性事务可能还没有运行，信息收集就会不完全。如果间隔过长，数据也会有所偏差。假设以下的情况：系统一直正常，但是最近几天有用户反映，在 A 时间段应用程序执行很慢。B 时间段正常，而 A 时间段有一个主要的事务 X 运行（也是用户使用到的事务）。B 时间段有另外

一个比较消耗资源的事务 Y 在运行。A 和 B 时间段的跨度比较大。本来快照如果覆盖 A 时间段内就已经能够收集到比较准确的数据了，但不巧的是，Report 所用的两个 Snap ID 的时间跨度太长，从而把 B 时间段内的统计数据也收集了进来。Statspack 经过比较，“ 认为 ”是事务 Y 对系统产生了主要影响（这也会在报告上体现出来），基于这样的分析，你会认为 Y 才是罪魁祸首，接下来，你不遗余力地对 Y 进行了 Tuning……

问题出现了！调整了 B 之后，用户继续报告，A 时间段内系统不但没有变快，反而变得更慢，甚至不可忍受。这种情况是很危险的，可能会对系统造成不同程度的损害。在比较严格的环境中，这已经构成了一次比较严重的事故。不得不承认，Statspack 的快照的采样时间间隔需要重视。

这是一个 Oracle 9.2 版本的 Statspack 报告：

```
           Snap Id     Snap Time      Sessions Curs/Sess Comment
           ------- ------------------ -------- --------- -------------------
Begin Snap:    1006 09-Nov-04 09:00:02       333       1.8
  End Snap:    1044 09-Nov-04 18:00:02       338       2.4

   Elapsed:            540.00 (mins)
```

从中可以看到快照 333 和快照 338 之间为 540.00（mins）。这么长的时间跨度，即使数据库在一定时间间隔内有问题，在 Statspack 报告中也难以准确体现。

还要注意的是，这里说的时间间隔，是 Begin Snap 和 End Snap 之间的间隔，而不是相邻两个 Snap 之间的间隔。对于 Snap 收集的间隔，建议以不要影响性能为准，收集得过于频繁，会对性能和存储都造成压力。当然，对于专家建议的 15～30 分钟，不能墨守成规，具体的环境下应该加以调整。

3.3 以偏概全

Statspack 从本质上说，是对系统的性能统计数据进行采样，然后进行分析。既然是采样，就会有偏差。如何消除偏差？统计学指出差值随样品个数的增加而降低。所以，只凭借一个 Report 文档就断定数据库的性能问题出在哪里是比较武断的做法（个别情况除外）。如果想得到相对准确的结果，就需要 DBA 创建多个报告（包括不同时间段），再进行对比和分析，这样才会起到很好的效果。

另外，在寻求技术支持时最好能够多提交几份报告，这样将有助于支持人员迅速解决问题。

3.4 关于 TIMED_STATISTICS 参数的设定

虽然这算是一个低级的错误，但是很遗憾，常常看到一些朋友对这个参数的忽略。如果在 TIMED_STATISTICS 的值设置为 false 时进行收集，可以说，得到的信息用处不是很大（我想您不会只想看一些实例名字、初始化参数之类的基本信息吧）。甚至可以说，如果该参数不设置为 true，性能分析无从说起。

建议在实例级设定 TIMED_STATISTICS 参数。如果设定了 TIMED_STATISTICS，那么 Statspack 报告将按照时间对等待事件排序。如果实例级的 TIMED_STATISTICS 为 false，而个别用户在 session

级设定了 TIMED_STATISTICS=true ，则报告中这些个别用户的等待时间会有计时，而其他事件则没有。这会直接影响到 Top 5 的那部分结果，甚至会产生一些很难解释的信息。

3.5　你成了泄密者

Statspack 报告会汇集比较全面的数据库系统信息，如果不对报告加以“ 伪装 ”就随意发布到一些网络技术论坛上寻求支持，无疑给一些黑客以可乘之机。

实例名字、主机名、数据库版本号、数据库用户名字、兼容参数、关键的表名字、文件路径等，尤其是关键的 SQL，这些都是对黑客们或是恶意入侵者有利的参考信息。更为严重的是，商业竞争对手也可能正在对你的数据库信息虎视眈眈。如果不想积极配合这些恶意窥探者，那么就不要把 Statspack 报告公之于众！

关于 Statspack 的安全方面还有一点是，如果为收集 Statspack 而建立的用户已经不再使用，为了避免默认用户名字和密码被猜测到，请执行以下的操作锁定该用户（这里假定用户名为 perfstat）。

```
SQL> ALTER USER perfstat ACCOUNT  LOCK;
```

参考信息

1．Advanced Tuning with Statspack
http://otn.oracle.com/oramag/oracle/03-jan/o13expert.html
2．Performance Tuning with Statspack PartII
http://otn.oracle.com/deploy/performance/pdf/statspack_tuning_otn_new.pdf
3．Performance Tuning with Statspack PartI
http://otn.oracle.com/deploy/performance/pdf/statspack.pdf

作者简介

冯大辉，网名 Fenng。目前关注于如何利用 Oracle 数据库有效地进行企业应用构建。对 Oracle RDBMS Tuning、Trouble Shooting 有浓厚的兴趣。

个人技术站点：http://www.dbanotes.net/

电子邮件：dbanotes@gmail.com

第4章　TKPROF工具使用简介

4.1　TKPROF 工具简介

TKPROF 是一个用于分析 Oracle 跟踪文件并且产生一个更加清晰合理的输出结果的可执行工具。如果一个系统的执行效率比较低，一个比较好的方法是跟踪用户的会话并且使用 TKPROF 工具的排序功能格式化输出，从而找出有问题的 SQL 语句。

TKPROF 命令后面可以带各种类型的排序选项，具体如下：

```
Usage: tkprof tracefile outputfile [explain= ] [table= ]
[print= ] [insert= ] [sys= ] [sort= ]
table=schema.tablename   Use 'schema.tablename' with 'explain='
option.
explain=user/password    Connect to ORACLE and issue EXPLAIN PLAN.
print=integer    List only the first 'integer' SQL statements.
aggregate=yes|no
insert=filename  List SQL statements and data inside INSERT
statements.
sys=no          TKPROF does not list SQL statements run as user SYS.
record=filename  Record non-recursive statements found in the
trace file.
waits=yes|no     Record summary for any wait events found in the
trace file.
sort=option      Set of zero or more of the following sort options:
prscnt  number of times parse was called
prscpu  cpu time parsing
prsela  elapsed time parsing
prsdsk  number of disk reads during parse
prsqry  number of buffers for consistent read during parse
prscu   number of buffers for current read during parse
prsmis  number of misses in library cache during parse
execnt  number of execute was called
execpu  cpu time spent executing
exeela  elapsed time executing
exedsk  number of disk reads during execute
exeqry  number of buffers for consistent read during execute
```

```
execu   number of buffers for current read during execute
exerow  number of rows processed during execute
exemis  number of library cache misses during execute
fchcnt  number of times fetch was called
fchcpu  cpu time spent fetching
fchela  elapsed time fetching
fchdsk  number of disk reads during fetch
fchqry  number of buffers for consistent read during fetch
fchcu   number of buffers for current read during fetch
fchrow  number of rows fetched
userid  userid of user that parsed the cursor
```

其中比较有用的一个排序选项是 fchela，即按照 elapsed time fetching 来对分析的结果排序（记住要设置初始化参数 TIME_STATISTICS=true），生成的.prf 文件将把最消耗时间的 SQL 放在最前面显示。另外一个有用的参数就是 SYS，这个参数设置为 no 可以阻止所有以 SYS 用户执行的 SQL 被显示出来，这样可以减少分析出来的文件的复杂度，便于查看。

首先解释 TKPROF 命令输出文件中各个列的含义。

（1）call：每次 SQL 语句的处理都分成 3 个部分。

- Parse：这步将 SQL 语句转换成执行计划，包括检查是否有正确的授权和所需要用到的表、列以及其他引用到的对象是否存在。
- Execute：这步是真正的由 Oracle 来执行的语句。对于 insert、update、delete 操作，这步会修改数据，对于 select 操作，这步就只是确定选择的记录。
- Fetch：返回查询语句中所获得的记录，这步只有 select 语句会被执行。

（2）count:这个语句被 parse、execute、fetch 的次数。

（3）cpu：这个语句对于所有的 parse、execute、fetch 所消耗的 CPU 的时间，以秒为单位。

（4）elapsed：这个语句所有消耗在 parse、execute、fetch 的总的时间。

（5）disk：从磁盘上的数据文件中物理读取的块的数量。一般来说更想知道的是正在从缓存中读取的数据而不是从磁盘上读取的数据。

（6）query：在一致性读模式下，所有 parse、execute、fetch 所获得的 buffer 的数量。一致性模式的 buffer 是用于给一个长时间运行的事务提供一个一致性读的快照，缓存实际上在头部存储了状态。

（7）current：在 current 模式下所获得的 buffer 的数量。一般在 current 模式下执行 insert、update、delete 操作都会获取 buffer。在 current 模式下，如果在高速缓存区发现了有新的缓存足够给当前的事务使用，则这些 buffer 都会被读入缓存区中。

（8）rows: 所有 SQL 语句返回的记录数目，但是不包括子查询中返回的记录数目。对于 select 语句，返回记录是在 fetch 这步，对于 insert、update、delete 操作，返回记录则是在 execute 这步。

4.2 TKPROF 工具的使用步骤

TKPROF 的使用基本上遵循以下几个步骤：

（1）在数据库级别上设置 TIMED_STATISTICS 为 true。

8i 数据库需要修改 init 文件增加 TIMED_STATISTICS=true，然后重启数据库使其生效。

9i 数据库可以直接修改：

```
SQL> alter system set timed_statistics=false scope=both;
系统已更改。
```

（2）使用各种的工具得到想要查看 session 的 trace。

在 Oracle 中能够得到一个 session 的 trace 的方法很多，这里大致地介绍一下。

方法一：如果需要在数据库级别上设置 trace，可以在 init.ora 文件中加入 sql_trace=true，这样在数据库运行的同时会 trace 所有数据库的活动，一般来说，只有在数据库的所有事务都需要被监控的时候才使用这种方法，使用这种方法会产生大量的 trace 文件，此参数是需要重启数据库才可以生效的。trace 信息收集完成后，记得去掉这个参数，然后重启数据库。

方法二：如果需要在 session 级别上设置 trace，可以在 SQL*Plus 中使用下列语句：

```
SQL> alter session set sql_trace=true;
会话已更改。
```

如果要在 PL/SQL 中对 session 级别设置 trace，可以使用 dbms_session 这个包：

```
SQL>exec dbms_system.set_sql_trace_in_session(sid,
serial#,true);
PL/SQL 过程已成功完成。
```

（3）找到生成的正确 trace 文件，在 init.ora 初始化参数文件中 user_dump_dest 定义的路径下可以找到。

（4）对 trace 文件使用 TKPROF 工具进行分析。

```
tkprof tracefile outfile [explain=user/password] [options...]
```

一般来说，使用 TKPROF 得到的输出文件中包含 3 个部分。

- SQL 语句本身。
- 相关的诊断信息，包括 CPU 时间、总共消耗的时间、读取磁盘数量、逻辑读的数量、以及查询中返回的记录数目等。
- 列出这个语句的执行计划。

4.3　TKPROF 工具如何分析 trace 文件

下面用一个具体的例子来说明如何利用 TKPROF 来分析 trace 文件。

先从 OS 上利用 top 命令找到当前占用 CPU 资源最高的一个进程的 PID 号：14483。

```
load averages:  1.53,  1.37,  1.39 db2    23:11:15
246 processes: 236 sleeping, 1 running, 2 zombie, 4 stopped, 3 on cpu
CPU states: 68.0% idle, 17.3% user,  3.0% kernel, 11.6% iowait,  0.0% swap
Memory: 16.0G real, 3.4G free, 9.7G swap in use, 11.0G swap free
   PID USERNAME THR PR NCE  SIZE   RES STATE   TIME FLTS    CPU COMMAND
 14483 oracle    1 51   0  8.8G  8.8G sleep  77.1H    1  5.69% oracle
 28222 oracle    1 52   0  8.8G  8.8G cpu18  93:55 2742  3.32% oracle
  3722 oracle    1 59   0  8.8G  8.8G sleep 157:41    0  0.45% oracle
 16077 oracle    1 59   0  8.8G  8.8G sleep  17.1H    0  0.34% oracle
 12687 oracle    1 59   0  8.8G  8.8G sleep   0:07    0  0.29% oracle
 17781 oracle    1 49   0  8.8G  8.8G run    91:11    8  0.24% oracle
  2359 oracle    1 59   0  8.8G  8.8G cpu19 524:53    0  0.12% oracle
  6559 oracle    1 59   0  8.8G  8.8G sleep 237:41    0  0.10% oracle
```

```
  2242 oracle    1 59   0  8.8G  8.8G sleep 980:56    0  0.09% oracle
  2356 oracle    1 59   0  8.8G  8.8G sleep 121:31    0  0.08% oracle
 16106 oracle    1 59   0  8.8G  8.8G sleep 168:44    0  0.05% oracle
 11432 oracle    1 49   0 2576K 1680K cpu11   0:11    0  0.05% top
  2333 oracle    1 59   0  8.9G  8.8G sleep 159:03    0  0.05% oracle
  2321 oracle    1 59   0  8.8G  8.8G sleep  78:20    0  0.04% oracle
  2282 oracle    1 59   0  8.8G  8.8G sleep 424:57    0  0.03% oracle
```

然后在数据库中根据 PID 号找到相应的 SID 号和 SERIAL#：

```
SQL> select s.sid,s.serial# from v$session s,v$process p
  2  where s.paddr=p.addr and p.spid='14483';
     SID   SERIAL#
---------- ----------
     101     25695
```

使用 dbms_system.set_sql_trace_in_session 包来对这个 session 进行 trace：

```
SQL> exec DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(101,25695,true);
PL/SQL procedure successfully completed.
```

到 user_dump_dest 定义的路径下查找刚刚生成的 trace 文件，根据时间来排序，可以找到最近的 trace 文件，接着使用 TKPROF 工具对此 trace 文件进行格式化分析，生成分析后的 trace 文件。

```
$tkprof orcl_ora_14483.trc allan.txt explain=system/manager aggregate=yes sys=no waits=yes
sort=fchela
TKPROF: Release 9.2.0.4.0 - Production on Sun Dec 5 22:27:28 2004
Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
```

这里生成的 allan.txt 文件就是最终得到的格式化后的 trace 文件了，然后打开这个文件对其进行分析。

TKPROF 工具输出最后总的统计如下：

```
OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse       20      0.01       0.02          0         58          0          0
Execute  13197      0.81       0.90         17       7436       6316       1484
Fetch    12944     22.86      22.10         20    2205941          0       8972
------- ------  -------- ---------- ---------- ---------- ----------
total    26161     23.68      23.02         37    2213435       6316      10456
Misses in library cache during parse: 14
********************************************************************************
Trace file: orcl_ora_14483.trc
Trace file compatibility: 9.00.01
Sort options: fchela
      1  session in tracefile.
    671  user  SQL statements in trace file.
     20  internal SQL statements in trace file.
    691  SQL statements in trace file.
     42  unique SQL statements in trace file.
     26  SQL statements EXPLAINed using schema:
           SYSTEM.prof$plan_table
           Default table was used.
           Table was created.
           Table was dropped.
  67054  lines in trace file.
```

（1）这个 session 中一共有 691 个 SQL 语句在里面，对于递归的语句一共分析了 20 次，基本上不存在什么问题。

（2）根据分析的数据计算命中率。已知：

```
Logical Reads =Consistent Gets+DB Block Gets
```

其中的 Consistent Gets 可以由 query 列中的 execute 和 fetch 中求出，而 DB Block Gets 则是可以由 current 列中的 execute 和 fetch 中求出，因此，Logical Reads 的值为 7436+2205941+6316+0=2219693。Physical Reads 即是 disk 列的值，为 20+17=37。

此时 Hit Ratio =1-(Physical Reads/Logical Reads) =1-(37/2219693) =99.99%

（3）从统计中，可以看出这个 session 中 fetch 的次数，以及取得的记录的行数，一般来说，都希望用尽可能少的 fetch 来取得尽可能多的记录数，这样证明，要少做工作，fetch 值一般要远远小于取得记录的行数才好。从上面的例子可以看到，fetch 了 12944 次，一共取得了 8972 行记录，这表明有些记录是通过好几次的 fetch 才得到的，因此，这个 session 中的一些 SQL 是需要优化的。

（4）从 query 列的 Parse 中可以看到从 dictionay cache 中读出的数据。从上面的统计中可以看出这个 session 从 Library Cache 中读取了 58 个 block 的数据。

查看生成 trace 文件中最消耗资源的语句，主要根据 cpu 列和 elapsed 列的值来确定消耗资源的多少。

```
UPDATE test set Amount = Amount - :b2
where
TestSequenceID = :b1
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse        0      0.00       0.00          0          0          0          0
Execute    731     22.43      21.52          7       2194        974        731
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------
total      731      0.15       0.18          7       2194        974        731
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 43  (TEST)   (recursive depth: 1)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT   GOAL: CHOOSE
      0   UPDATE OF 'TEST'
      0    INDEX (UNIQUE SCAN) OF 'PK_TEST' (UNIQUE)
```

从上面可以看出这个语句访问了 2194 个 block 来找到需要 update 的行记录。在执行 update 的时候只访问了 974 个 block，一共更新了 731 条记录。

从执行计划中可以看出，update 时是走了索引的，所以获取了 731 条记录访问了 2194 个 block。如果只是获取很少了记录，而又访问了大量的 block，就有可能是这个表上的索引失效或者是查询中没有使用索引，这样就需要对相应的 SQL 进行优化了。

查看生成的 trace 文件，分析过多的语句。

```
select * from ErrorEvent where rownum<1000 and ERRORCOUNT<100 order by EventID
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse       12      3.02       6.04          0        101          0          0
```

```
Execute   12     0.00       0.03        0         0         0          0
Fetch     12     6.41       5.47       88    143290         0         10
------- ------  -------- ---------- ---------- ---------- ----------
total     36     9.43      11.55       88    143391         0       10
Misses in library cache during parse: 11
Optimizer goal: CHOOSE
Parsing user id: 43  (TEST)
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  SORT ORDER BY (cr=445 r=1 w=0 time=18059 us)
      0   COUNT STOPKEY (cr=445 r=1 w=0 time=17987 us)
     0    TABLE ACCESS FULL ERROREVENT (cr=445 r=1 w=0 time=17983 us)
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   SORT (ORDER BY)
      0    COUNT (STOPKEY)
      0     TABLE ACCESS (FULL) OF 'ERROREVENT'
```

可以看出这个查询一共返回了 10 条记录，为了返回这 10 条记录，一共访问了 143290 个 block，由执行计划可以看出来，对 ErrorEvent 表做的都是全表扫描，因此访问的 block 会这么多，而只是为了得到 10 条记录，因此，需要考虑在这个表上建立相关的索引或者是看查询为什么没有用到已经建立好的索引。

这个查询被分析了 12 次，而其中有 11 次都是硬解析，占用了比较高的 CPU 资源，这个肯定不是我们所期望的，这个查询执行的时间也比较长，所以需要对这个查询进行相应的优化，减少分析的次数，尽量使用绑定变量，查看共享池是否设置过小。

以上是对使用 Oracle 数据库的 TKPROF 工具的一些简单介绍，希望能对大家熟练掌握这个工具有所帮助。

作者简介

叶梁，网名 cooly1，现任 ITPUB Oracle 管理版版主。

曾任职于国内某大型软件企业做 Oracle 数据库的技术支持，客户遍及全国各个行业，尤其是电信、政府、金融行业。

现任职于某外资电信企业华北区分公司，从事 DBA 工作，负责华北区 40 多个数据库系统的维护，对大型数据库管理经验丰富。

擅长数据库的维护，对于数据库的安装，调整，备份方面有自己独到的经验。同时也给一些国内的大型企业做过 Oracle 的培训，有一定的培训经验。

曾做过很多大型项目的数据库维护和支持工作，对 Oracle 的维护有丰富的实际经验，善于现场解决问题。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

5

第 5 章　使用 Oracle 的等待事件检测性能瓶颈

在Oracle 数据库系统中，经常会出现一些性能问题，即便是一个性能已经很好的系统，有时也会有瓶颈发生，所以优化是 DBA 经常要面临的一项工作。执行操作的进程可能经常会遇到需要等待资源的情况，例如，进程需要的数据可能已经在 SGA 中，但是也很可能还在磁盘上；执行操作所需要的锁存器（确保单个进程执行特定的内核代码片）可能被另一个需求获得；需要的队列资源（访问 Oracle 的一些资源）已经被别的进程占用；等待用户的输入或者其他情况。瓶颈总是存在的，仅仅是大家能够接受的底线不同而已。会话可能总是在等待着什么，可能是等待着什么的开始，也可能是等待着什么的结束，而 DBA 则可以通过会话等待事件的统计信息了解这个会话到底在等着什么。

以前，判定 Oracle 的性能好坏主要依赖于性能指标（如数据块的命中率等）。现在，这些指标已经不再是惟一的判断方法了，当然，它们本来也不是非常好的方法。在本章中使用的方法是会话等待事件的分析方法，这种方法将能够为 DBA 提供更深入的性能分析。

使用各种比率指标进行 Oracle 调优的方法需要对大量性能指标非常地熟悉，要求 DBA 是各种指标的“ 专家 ”。例如，磁盘排序的数量与排序总量的比值，很明显需要最小化磁盘上的排序，以便这个比率接近于 0。为了优化这个排序，就需要得到 Oracle 在这方面的一些比值，加上足够的使用这些比值的经验，和足够的应用程序经验，再加上应用程序执行的经验，才能够很好地利用这些比值来优化性能。有很多性能方面的文章、说明等都涉及到了如何使用这些比值来调优数据库，但由于这些文档涉及到的大量指标比值大多是非常冗长和复杂的。所以能够真正掌握使用各种比率指标进行 Oracle 性能优化的人并不多。

从 Oracle 7 开始，Oracle 开始提供了会话等待事件信息来帮助 DBA 得到更为准确的数据库性能信息。而在老的版本中，管理员仍然需要借助于相对来说不够有效率的比值来分析性能。Oracle 利用内部触发器记录和显示一些视图得到某些资源（Latch、I/O、内存中的数据或者队列等）但是不能立即得到与进程相关的信息。任何人都可以从 V$这种视图中得到准确的关于 Oracle 进程在等待什么资源之类的准确信息。这种性能视图就是会话等待事件视图：v$system_event、v$session_event 和 v$session_wait。

会话等待事件的一个明显优点是它不再与某个单独的应用相关，也就是说，会话等待事件是独立于应用的。任何一个应用具有很高的全表扫描的等待事件都意味着 I/O 有了问题。当然，实

际的根本原因既可能是糟糕的 SQL 语句，也可能是糟糕的 I/O 系统架构或者其他的原因，但是不论怎样，一定是跟 I/O 相关的问题，那么下一步要做的，就是继续根据其他的信息来判断真正的原因是什么。

因为会话等待事件是与应用无关的，所以它可以被作为很好的性能监测工具的基础依据。基于会话等待事件的性能监测工具将是更准确、更方便、更简单的工具。

5.1 判断等待事件的相关视图

关于等待事件的会话视图不是一个，而是一组，每个视图都有特定的目标，描述了关于系统级或者会话级的详细等待信息。这些视图包括了 v$system_event，v$session_event 和 v$session_wait。关于这些视图，下面将分别介绍。

5.1.1 系统级统计信息 v$system_event

v$system_event 用来查看整个系统级或者说最高级别的整体系统的性能情况。这里面并没有每个会话的相关信息，而是对每个事件总计了从系统启动以来在所有会话中发生过的情况，就像整个实例的统计信息总计一样，这些信息在每次重启系统后都会被重新计数。

v$system_event 有 5 个字段：

- Event：该字段列出了所有发生的事件的名称。可能有很多等待事件发生，最常见的有 latch waits、db file scattered read、db file sequential read、enqueue wait、buffer busy wait、log file parallel write 和 free buffer waits。在这里将会介绍一些常见的等待事件，当然更多的等待事件列表，还是需要去查阅 Oracle 的相关文档。
- Total_waits：从数据库启动到现在这个等待事件总的等待次数。
- Total_timeouts：总的等待超时的次数。不是所有的等待都会超时，有些等待在一段时间之后得到锁需要的资源而结束，有些等待在没有得到资源而超时时将会不断重试，有些等待超时后将不会再等待，而有些等待却会一直等下去而不存在超时。
- Time_waited：总的等待时间，以厘秒（百分之一秒）为单位。这个数值表示了从数据库启动以来某个等待事件在所有会话中（包括了已经结束的和正保持着连接状态的会话）总的等待事件之和。
- Average_wait：平均等待事件，以厘秒（百分之一秒）为单位。表示了从数据库启动以来某个等待事件在所有会话中发生的平均等待时间，也就是 total_waits / time_waited 的值。

为了更清晰地看到等待事件的情况，可以对 v$system_event 一段时期内的信息进行比较，得到这段时间发生最多的等待事件，也就是当前这段时间最主要的等待事件是哪些。

图 5-1 显示了记录在 v$system_event 中自启动以来的系统详细信息。可以看出，从系统启动以来，会话有 69.61%的时间都在等待 Oracle 从磁盘中读取数据。当然，仅仅根据从系统启动以来等待与 I/O（Wait Event：db file scattered read）有关的全表扫描的主要等待事件，是不能确定会话当前的等待是由全表扫描引起的。要判断当前的系统瓶颈，需要的是最近一段时间内的系统性能变化情况，这种一段时期内的数据增量，尤其是在业务繁忙阶段收集的性能变化情况，是更好的性能判断依据。

```
SQL> @syssum.sql

Wait Event                      Time Waited %Time w    Waits %Waited
---------------------------------- ----------- ------- --------- -------
db file scattered read                  1463   69.61    17528   94.96
buffer busy waits                        291   13.84       65    0.35
PL/SQL lock timer                        160    7.60      307    1.66
latch free                               130    6.18      127    0.69
log file parallel write                   37    1.76      269    1.46
db file sequential read                   20    0.96      161    0.87
log file switch completion                 1    0.02        0    0.00
log file sync                              0    0.02        1    0.00
library cache pin                          0    0.01        0    0.00
log file sequential read                   0    0.01        0    0.00
log file single write                      0    0.00        0    0.00
process startup                            0    0.00        0    0.00
db file single write                       0    0.00        0    0.00
LGWR wait for redo copy                    0    0.00        0    0.00
enqueue                                    0    0.00        0    0.00
library cache load lock                    0    0.00        0    0.00
db file parallel write                     0    0.00        2    0.01
direct path write                          0    0.00        0    0.00
```

图 5-1　系统详细信息

图 5-2 则显示了一段时间内的会话等待事件的变化情况，会话在这里并不是主要在等待全表扫描，而是在等待重做日志的写操作（Wait Event： log file parallel write）。

```
SQL> @cr_base.sql

SQL> @in_base.sql

执行用户应用……

SQL> @re_base.sql

Wait Event                       Time Waited %Time w    Waits %Waited
---------------------------------- ----------- ------- -------- -------
log file parallel write                 4.360    73.15      598   72.57
buffer busy waits                       0.970    16.28      189   22.94
db file sequential read                 0.160     2.68        7    0.85
latch free                              0.130     2.18        9    1.09
log file sync                           0.060     1.01        1    0.12
enqueue                                 0.000     0.00        7    0.85
LGWR wait for redo copy                 0.000     0.00        1    0.12
direct path write                       0.000     0.00        1    0.12
db file scattered read                  0.000     0.00        0    0.00
db file single write                    0.000     0.00        0    0.00
direct path read                        0.000     0.00        0    0.00
library cache load lock                 0.000     0.00        0    0.00
library cache pin                       0.000     0.00        0    0.00
log file sequential read                0.000     0.00        0    0.00
log file single write                   0.000     0.00        0    0.00
log file switch completion              0.000     0.00        0    0.00
process startup                         0.000     0.00        0    0.00
```

图 5-2　会话等待事件的变化情况

5.1.2 会话级统计信息 v$session_event

v$session_event 视图显示了与 v$system_event 相类似的信息，只不过这些等待信息是每个等待事件在各个会话中的统计，所以多了关于会话 id 的字段而已。在这个视图中，一个会话重新建立时统计信息将被重置为 0。

这个视图在判断一个会话正在等待什么资源的时候非常有用。例如，假如发现 10 号会话的用户性能有明显问题，那么在用户执行应用前查看一下这个视图，然后让用户执行应用，在执行完成后再次查看这个视图，比较两次查询的结果，看看哪些等待事件是这段时间内最主要的等待事件，那么这些等待事件，就是这个用户应用的瓶颈了。

5.1.3 会话详细性能信息 v$session_wait

v$session_wait 是最复杂、最难理解的会话等待视图。它与 v$system_event 和 v$session_event 有明显的不同，这个视图只包含了当前正处于连接状态的会话的等待事件，而且它里面所包含的字段含义又较为隐晦，比如 P1、P2 等。这个视图并没有存储总计的信息，而是提供了当前已经发生或者正在发生的事件的可用信息。

除了实时显示这些已经发生的等待信息外，v$session_wait 也提供了关于当前正在发生的等待事件信息。例如，它直接描绘了一个正在发生锁存器竞争的会话，此时正在等待那个锁存器。或者，一个会话正在等待对一个数据块的访问，这个块所在的文件编号、块编号、访问了多少数据块这些信息都可以得到。

v$session_wait 视图也显示了详细的时间信息，就像 v$system_event 和 v$session_event 视图一样。然而，因为 v$session_wait 显示的是实时信息，所以它所显示的时间代表的含义要根据等待状态来决定。在下面将会有更详细的解释。

v$session_wait 视图包括以下字段：

- SID：会话的 ID。与 v$session_event 或者 v$session 中的会话 ID 一样。
- SEQ#：会话的内部顺序号。
- EVENT：描述事件的名称。一些常见的等待事件有 latch waits、db file scattered read、db file sequential read、enqueue wait、buffer busy wait 和 free buffer waits 等。
- P[1-3]：这三个参数用来提供关于等待事件更为详细的信息。这三个参数也是指向其他视图的外键字段，参数值与具体的等待事件相关。例如，对于一个锁存器等待，P2 就代表了锁存器编号，可以与 v$latch 关联查询。而对于 db file sequential read（indexed read），P1 则表示了文件编号，可以与 v$filestat 或者 dba_data_files 联合查询，而 P2 则代表了数据块编号，可以与 dba_extents、sys.uet$联合查询。为了使用这些参数，应该尽可能去了解这些参数的含义，在 Oracle 的文档中可以查到这些。在视图中的 P[1-3]TEXT 字段也可以提供一些信息，但是不要太多指望它们。
- P[1-3]RAW：十六进制方式描述的 P[1-3]的值。这些字段很少使用，所以也不需要去关心。
- P[1-3]TEXT：一些参数的名字，只是很简单的描述，用处不大。
- STATE：这个参数是很有用的，它提供了对下面两个字段的解释：WAIT_TIME 和 SECONDS_IN_WAIT。如果没有正确的理解 STATE 的含义或者忽略掉它，很可能在理解 WAIT_TIME 和 SECONDS_IN_WAIT 时引起歧义。STATE 字段有下面 4 种含义：

⑴ Waiting：会话正在等待这个事件。

⑵ Waited unknown time：由于 timed_statistics 被设置为 false，而导致不能得到关于时间的信息。

⑶ Waited short time：表示会话发生了等待，但是等待时间非常小，不超过一个时间单位（one clock tick），所以并没有记录。

⑷ Waited known time：一旦会话等待了资源然后得到了，那么状态将从 waiting 进入 waited known time。

- WAIT_TIME：这个字段的值与 STATE 相关。

⑴ 如果 STATE 字段值为 Waiting，那么 WAIT_TIME 值是无用的。

⑵ 如果 STATE 字段值为 Waited unknown time，那么 WAIT_TIME 值也是无用的。

⑶ 如果 STATE 字段值为 Waited short time，那么 WAIT_TIME 值也是无用的。

⑷ 如果 STATE 字段值为 Waited known time，那么这个就是实际的等待时间，以秒作为时间单位。能够看到这个值并不是很容易，因为如果会话开始等待一个资源，status 将再次变成 waiting，这个字段的值又会变成无用的（bogus）。

- SECONDS_IN_WAIT：这个值同样依赖于 STATE 字段。

⑴ 如果 STATE 字段值为 Waiting，那么这个值是实际等待的时间，以秒作为时间单位。如果看到了一个值，当再次查询的时候将看到不同的值，因为每次看到的都是当前的等待已经持续的时间。如果经常去查询这个视图，那么将会得到一个会话什么时候发生了等待，在等待什么这样的信息。

⑵ 如果 STATE 字段值为 Waited unknown time，那么 SECONDS_IN_WAIT 值也是无用的。

⑶ 如果 STATE 字段值为 Waited short time，那么 SECONDS_IN_WAIT 值也是无用的。

⑷ 如果 STATE 字段值为 Waited known time，那么 SECONDS_IN_WAIT 值也是无用的。

这个视图对于确定发生的等待事件或者等待事件的原因是非常有用的。例如，如果系统中有 500 个会话，那么在 v$session_event 中将显示 500 行会话信息。而对于大多数 DBA 来说，这么多的信息反而会显得模糊而难以分辨真正需要的信息。那么更有价值的信息应该是提供发生等待的会话的信息，此时 DBA 需要做的仅仅是通过这个视图发现哪些会话有明显的等待，从而进一步确定是哪些会话的哪些等待事件。

例如，在图 5-3 中可以看到大多数会话都在等待与 I/O 有关（Wait Event：db file scattered read）的全表扫描。

```
SQL> @seswa.sql

                                                 Num. Sess.
Wait Event                          Waited So Far (sec)    Waiting
----------------------------------- ------------------- ----------
rdbms ipc message                                1,012          5
latch free                                           0          2
smon timer                                         301          1
buffer busy waits                                    0          1
db file scattered read                               0        165
pmon timer                                           0          1

6 rows selected.
```

图 5-3　主要的会话等待事件

那么接下来要做的就是，进一步看看这些特定的会话到底在等待对哪些数据块的访问，如图 5-4 所示。

```
SQL> select sid,event,p1,p2,p3
  2  from v$session_wait
  3  where event like 'db%file%scat%';

  ID Wait Event                                P1       P2    P3
---- ----------------------------------- ------------ --------- -----
   8 db file scattered read                        8      6572    16
  10 db file scattered read                        8      6413    16
```

图 5-4 会话所等待的资源

记住记录在 v$session_event 中的等待事件记录在当前等待的资源（记录在 v$session_wait）没有得到之前是不会更新的。所以如果一个等待事件（例如 SQL*Net message [from, to] client）等待的时间非常长，那么将很明显地看到，在 v$session_wait 中的统计信息将持续增加而相应在 v$session_event 中的信息保持不变。

5.1.4 会话等待事件的相关视图之间的关系

在 Oracle 数据库中，由于事件发生的都很快，所以时间信息是非常重要的。例如，对 SQL*Net message 事件研究发现，有很多重要的时间信息。下面是在这瞬间的一些情况：

- 这里总是有一些与会话相关的等待。如果客户端程序总是挂起，也可能是在内部循环，那么 SQL*Net message from client 就是等待事件。
- 一个会话一次只发生一个等待事件，如果看到了其他的等待事件，那仅仅表示在下一个时间片上发生了等待，在某个时刻只存在一个等待。
- v$session_wait 的 time waited 和 seconds in wait 字段值以秒为单位，而 v$session_event 的 time waited 和 average time waited 字段值以厘秒（百分之一秒）为单位。
- v$session_wait 的等待事件结束后，v$session_event 的统计信息将会发生改变。
- 记录 v$session_wait 的统计信息意义不大，因为这里面的数据是实时变化的。
- 应该记录 v$session_event 的信息，当 v$session_wait 中的等待事件得到所需资源而结束等待时，在 v$session_event 视图中的信息将会增加，最近发生的等待时间信息将增加到原有的统计信息中。
- 当 v$session_wait 里面的等待事件结束时，v$session_wait.seconds_in_wait 的值被复制到 v$session_event.time_waited 字段，而 v$session_event.average_time 字段同时也被修改。

5.2 应该怎么考虑进行优化

如上所述，使用会话等待事件统计信息提供了一种性能判断方法，它有助于发现性能的差异，找到产生竞争冲突的原因。在实际环境中怎样使用这些等待事件视图，怎么执行基本的竞争判断方法在下面将会讲到。性能问题通常可以通过查看系统的性能表现或者收集系统的性能指标来发现。

当发现系统变慢时，首先通过 v$system_event 从系统级进行收集，要注意的是不仅仅要看累计的系统等待事件信息，更为重要的是一段时间内的等待的时间增量，这能更好地描述等待事件发生的严重程度。一般建议每隔一段时间进行一次收集和比较（如半小时进行一次收集）。通过这种方法，可以很快地确定整个系统中的主要瓶颈在哪里。

当然，这里仍然留下了一个问题，那就是，虽然瓶颈已经确定，但是真正瓶颈发生在哪个确切位置仍然是不清楚的，例如，等待发生在具体的哪个锁存器、文件或者对哪个块的访问上，这是由于引起瓶颈的真正对象在系统级不能得到，所以需要进一步的收集分析，通过查看会话级的 v$session_wait 视图的等待事件信息，可以进一步判断出具体引起瓶颈的原因是什么，例如，判断出到底在哪个锁存器上发生了竞争，或者对哪个数据文件的访问引起了 I/O 竞争。如果发现是文件 I/O 竞争，还可以进一步找出，竞争的原因是由于对索引的访问还是大量的全表扫描。如果是块上发生了竞争，也可以进一步确定是对哪个文件的哪个段上的块访问发生了竞争。这些信息对于判断性能瓶颈的原因，选择合适的解决方法是非常有用的。

当再一次看到数据高速缓存区的命中率仅仅为 60%时，先不要着急，看看有没有任何相关的等待事件，如果没有，那么不用担心，因为数据库性能良好，在任何操作执行时都没有与数据高速缓存区相关的等待事件。相反，如果数据高速缓存区命中率达到了 99%，但是却发现这里存在着大量的等待，那么一个不幸的消息是，很高的命中率，反而带来了性能的下降（原因有可能是数据高速缓存区太大，超过了物理内存的大小，带来了大量的分页交换等），当然，还有一个好消息，就是现在可以通过等待事件来判断导致问题的原因是什么了。

5.3　主要等待事件

已经有很多等待事件都有文档描述，应该很容易被理解，然而有些等待事件在 Oracle 的文档中描述得并不是很清楚，这里，将本人对这些等待事件的一些看法介绍一下，为大家提供一些利用等待事件解决竞争问题的思路和方法。首先了解一下等待事件的大致分类，看看哪些是应该关心的等待事件。Oracle 中主要有两类等待事件：空闲等待和非空闲等待。

1. 空闲等待

空闲等待意味着 Oracle 正在等待某种动作的发生。有时 Oracle 的进程在等待，实际上并不是因为忙而等待，而是因为没有事情所以在等待，例如 smon timer，SMON 进程的一些操作是每隔一段时间轮询执行的，在系统不忙的时候，这种操作也不会立即发生，而是等待计时器达到一定的时刻时再执行，这时候就会产生 smon timer 等待事件，而系统实际上并没有性能问题。只有很少的空闲等待事件与性能有关，大多时候这类事情对性能影响不大。

空闲类的等待事件典型有：client message（客户机消息）、null event（null 事件）、pipe get（管道取操作）、SQL*Net message from client（来自客户端的消息）、SQL*Net message to client（发送至客户端的消息）、rdbms ipc message（数据库 ipc 消息）、virtual circuit status（虚拟环路状态信息）、smon timer（smon 计时器）、pmon timer（pmon 计时器）、dispatcher timer（调度器计时器）等。

2. 非空闲等待事件

通常在数据库发生了竞争时就会出现非空闲等待。这种等待大多意味着系统中发生了竞争，

当某种操作发生时，操作所需要的资源正在被其他操作占用，而这种独占的资源不能被后面的操作请求立即得到，操作请求被阻塞而发生了等待。非空闲等待事件主要有以下。

buffer busy waits（数据高速缓存忙等待）、db file scattered read（数据文件离散读）、db file sequential read（数据文件顺序读）、db file parallel write（数据文件并行写）、db file single write（数据文件单次写）、enqueue（队列）、free buffer inspected（空闲数据缓冲区探测）、free buffer waits（空闲缓冲区等待）、latch free（栓空闲等待）、log file parallel write（日志文件并行写）、log file sync（日志文件同步）、log buffer space（日志缓冲区空间分配）、log file single write（日志文件单次写）、log file switch（archiving needed）、log file switch（checkpoint incomplete）、direct path read（直接路径读）、direct path write（直接路径写）、library cache load lock（库高速缓存装载锁）、library cache lock（库高速缓存锁）、library cache pin（库高速缓存执行锁）、timer in sksawat（归档过慢）、transaction（事务阻塞）、undo segment extension（回滚段动态扩展）等。

对性能影响的主要是非空闲等待事件，所以这里主要介绍一下常见的非空闲等待事件的基本含义，如表 5-1 所示。

表 5-1　　非空闲等待事件的基本含义

等待事件	等待事件描述
buffer busy waits	表示在等待对数据高速缓存区的访问，这种等待事件通常出现在会话读取数据到 buffer 中或者修改 buffer 中的数据时，例如 DBWR 正在写一些数据块到数据文件的同时，其他进程需要去读取相应的数据块。同时也可能表示着在表上设置的 free lists 太少，不能支持大量并发的 Insert 操作。在 v$session_wait 视图的 P1 字段值表示相关数据块所在的文件编号，P2 则表示文件上的块编号，通过这些信息与 dba_data_files 和 dba_extents 的联合查询就可以很快定位到发生竞争的存储对象，从而进一步确定问题的根源
db file parallel write	与 DBWR 进程相关的等待，一般都代表了 I/O 能力出现了问题。通常与配置的多个 DBWR 进程或者 DBWR 的 I/O slaves 个数有关，当然也可能意味着在设备上存在着 I/O 竞争
db file scattered read	表示发生了与全表扫描相关的等待。通常意味着全表扫描过多，或者 I/O 能力不足，或者 I/O 竞争
db file sequential read	表示发生了与索引扫描相关的等待。同样意味着 I/O 出现了问题，通常表示 I/O 竞争或者 I/O 需求太多
db file single write	表示在检查点发生时与文件头写操作相关的等待。通常与检查点同步数据文件头时文件号的紊乱有关
direct path read	表示与直接 I/O 读相关的等待。当直接读数据到 PGA 内存时， direct path read 出现。这种类型的读请求典型地作为：排序 IO（当排序不能在内存中完成的时候）、并行 Slave 查询或者预先读请求等。通常这种等待与 I/O 能力或者 I/O 竞争有关
direct path write	同 direct path read，只是操作为写
enqueue	表示与内部队列机制相关的等待，例如对保护内部资源或者组件的锁的请求等，一种并发的保护机制
free buffer inspected	表示在将数据读入数据高速缓存区的时候等待进程找到足够大的内存空间。通常这类等待表示数据高速缓存区偏小

续表

等待事件	等待事件描述
free buffer waits	表示数据高速缓存区缺少内存空间。通常与数据高速缓存区内存太小或者脏数据写出太慢有关。这种情况下，可以考虑增大数据高速缓存区或者通过设置更多地 DBWR 来增加脏数据的写能力
latch free	表示某个锁存器上发生了竞争。首先应该确保已经提供了足够多的 Latch 数，如果仍然发生这种等待事件，那么应该进一步确定是那种锁存器上发生了竞争（在 v$session_wait 上的 P2 字段表示了锁存器的标号），然后再判断是什么引起了这种锁存器竞争。大多数锁存器竞争都不是简单的由锁存器引起的，而是与锁存器相关的组件引起的，所以需要找到具体导致竞争的根本。例如，如果发生了 library cache latch 竞争，那么通常都表示着库高速缓存的配置不合理，或者 SQL 语句书写不合理，带来了大量的硬解析
library cache load lock	表示在将对象装在到库高速缓存时出现了等待。这种事件通常代表着发生了负荷很重的语句重载或者装载，可能由于 SQL 语句没有共享或者共享池区域偏小造成的
library cache lock	表示与访问库高速缓存的多个并发进程相关的等待。通常表示不合理的共享池大小
library cache pin	这个等待事件也与库高速缓存的并发性有关，当库高速缓存中的对象被修改或者被检测的时候发生
log buffer space	表示日志缓冲区出现了空间等待事件。这种等待事件意味着写日志缓冲区的时候得不到相应的内存空间，通常发生在日志缓冲区太小或者 LGWR 进程鞋太慢的时候（可能由其他原因造成）
log file parallel write	表示等待 LGWR 向操作系统请求 I/O 开始直到完成 IO。 在触发 LGWR 写的情况下如 3 秒、1/3、1MB、DBWR 写之前可能发生。这种事件发生通常表示日志文件发生了 I/O 竞争或者文件所在的驱动器较慢
log file single write	表示写日志文件头块时出现了等待。一般都是发生在检查点发生时
log file switch（archiving needed）	由于归档过慢造成日志无法切换而发生的等待。这种等待事件的原因可能比较多，最主要的原因就是归档速度赶不上日志切换的速度。可能的原因包括了重做日志文件太小，重做日志组太少，归档能力太低，归档文件发生了 I/O 竞争，归档进程挂起，或者归档日志文件放在了慢速磁盘设备上等
log file switch（checkpoint incomplete）	表示在日志切换时相应文件上的检查点还没有完成。一般意味着日志文件太小造成日志切换太快或者其他原因（例如 DBWR 没有写完脏数据）造成检查点太慢
log file sync	表示当服务进程发出 commit 或 rollback 命令后，直到 LGWR 完成相关日志写操作这段时间的等待。 如果有多个服务进程同时发出这种命令，LGWR 不能及时完成日志的写操作，就有可能造成这种等待
transaction	表示发生了一个阻塞回滚操作的等待
undo segment extension	表示在等待回滚段的动态扩展。这表示可能事务量过大，同时也意味着可能回滚段的初始大小不是最优，MINEXTENTS 设置得偏小。考虑减少事务，或者使用最小区数更大的回滚段

大多数空闲等待与性能的关系很小，所以在等待事件中看到的空闲等待大多可以忽略，在这里只介绍一种空闲等待事件 SQL*Net message from client，这可能是在一定程度上反映了问题的等待。

SQL*Net Message From Client 等待事件在大多数时候也是可以被忽略的，它表示服务进程在等待客户进程返回一些回应信息，由于客户端、服务端本身的一些正常反应、计算、操作等动作的执行，所以正常情况下，连接交互时这种等待可能会经常发生。但是如果发现这种等待事件堆积得非常多，就需要注意，很有可能是网络传输出现了问题，或者网络带宽发生了竞争，从而导致大量回应信息的堆积，这时就需要进行优化了。

5.4 案例分析

仅仅是知道了一些等待事件，知道了与等待事件相关的性能视图，并不能说明已经能够通过这些信息来帮助解决等待事件反映的性能问题，前面所提到的都是在分析等待事件、优化系统性能时所需要的辅助工具，如何去利用它们来解决这些问题才是更重要的，下面就通过一个简单的案例来说明如何利用这些工具辅助分析问题、发现问题和解决问题。

为了全面地了解和分析系统中发生的等待事件，需要用两种方法收集系统级的相关等待信息，以便准确地分析出真正导致这些等待或者性能问题的原因。

首先要从整个系统的角度进行了解，找到从系统启动以来最主要的等待事件是什么。为了得到系统中最主要的等待事件，查询 v$system_event 动态性能视图，如图 5-5 所示。

```
SQL> @syssum.sql

Wait Event                      Time Waited %Time w    Waits %Waited
---------------------------------- ----------- ------- --------- -------
db file scattered read                  1501   66.89    17703    92.82
buffer busy waits                        309   13.77       78     0.41
PL/SQL lock timer                        189    8.40      354     1.85
latch free                               138    6.14      134     0.70
log file parallel write                   51    2.25      304     1.59
db file sequential read                   29    1.31      179     0.94
enqueue                                   17    0.76        0     0.00
free buffer waits                          7    0.30        1     0.00
log file sync                              1    0.05        1     0.01
write complete waits                       1    0.05        0     0.00
log file switch completion                 1    0.05        0     0.00
log buffer space                           1    0.03        0     0.00
library cache pin                          0    0.01        0     0.00
log file sequential read                   0    0.01        0     0.00
log file single write                      0    0.00        0     0.00
undo segment extension                     0    0.00      314     1.65
LGWR wait for redo copy                    0    0.00        0     0.00
process startup                            0    0.00        0     0.00
db file single write                       0    0.00        0     0.00
library cache load lock                    0    0.00        0     0.00
db file parallel write                     0    0.00        2     0.01
direct path write                          0    0.00        0     0.00
direct path read                           0    0.00        0     0.00
buffer deadlock                            0    0.00        0     0.00
```

图 5-5 v$ system_event 动态性能视图

可以清楚地看到在系统中当前最主要的等待事件是与全表扫描（Wait Event：db file scattered read）相关的等待事件。当然，要注意的是，这里看到的等待事件并不意味着它一定是现在系统性能下降的绝对原因，这里显示的只是表示这种等待事件从系统启动以来发生得最多，所以还需要从另一个方面去收集分析。

除了从整体系统的角度了解发生过的等待事件外，更为重要的是找到引起当前系统性能下降的真正等待事件是什么，这就需要了解在应用执行的这段时间中最主要的等待是什么，查询系统中的等待事件仍然要从 v$system_event 中得到，与上面的查询不同的是，现在需要执行查询两次（业务繁忙期间），并对这两次查询进行比较，得到应用执行的这段时间中最主要的等待事件，如图 5-6 所示。从图 5-6 中可以看出，在这段时间内，最主要的等待事件不再是与全表扫描相关的，而是反映重做日志写能力有问题的 log file parallel write 事件了。

```
SQL> @cr_base.sql

SQL> @in_base.sql

执行用户应用……一段时间之后

SQL> @re_base.sql

Wait Event                      Time Waited %Time w    Waits %Waited
---------------------------------- ----------- ------- -------- -------
log file parallel write              23.350    89.70      2164   96.78
free buffer waits                     1.020     3.92         1    0.04
db file sequential read               0.390     1.50        14    0.63
latch free                            0.080     0.31         5    0.22
log file sync                         0.040     0.15         1    0.04
db file parallel write                0.000     0.00        23    1.03
direct path write                     0.000     0.00         1    0.04
LGWR wait for redo copy               0.000     0.00         0    0.00
buffer busy waits                     0.000     0.00         0    0.00
buffer deadlock                       0.000     0.00         0    0.00
db file scattered read                0.000     0.00         0    0.00
db file single write                  0.000     0.00         0    0.00
direct path read                      0.000     0.00         0    0.00
enqueue                               0.000     0.00         0    0.00
library cache load lock               0.000     0.00         0    0.00
library cache pin                     0.000     0.00         0    0.00
log buffer space                      0.000     0.00         0    0.00
log file sequential read              0.000     0.00         0    0.00
log file single write                 0.000     0.00         0    0.00
log file switch completion            0.000     0.00         0    0.00
process startup                       0.000     0.00         0    0.00
undo segment extension                0.000     0.00         0    0.00
write complete waits                  0.000     0.00         0    0.00

23 rows selected.
```

图 5-6　主要等待事件

为了了解目前这个数据库的重做日志写能力，继续查看当前的重做日志缓冲区大小、日志组

数和组成员大小。日志缓冲区大小 1MB，属于比较合理的大小，一般不会引起等待事件，与上面的重做日志等待事件无关。然而这里却发现只有两个重做日志组，每个组员文件大小只有 500KB，是不是因为这个引起了等待呢。

仔细地分析一下，log file parallel write 等待事件跟什么有关呢？看看前面对事件的描述，这个等待事件表示等待 LGWR 向操作系统请求 I/O 开始直到完成 I/O。这种事件发生通常表示日志文件发生了 I/O 竞争或者文件所在的驱动器较慢。这说明这种等待与日志切换、检查点的执行都没有关系，而是直接反映了 LGWR 的写能力，因此即使日志文件组数过少、文件偏小，也与目前的等待事件没有直接关系，所以增加日志组数、日志文件大小并不会有助于解决现在的性能问题。

那么现在怎么解决这个问题呢？这个问题的直接原因主要跟 LGWR 的写能力有关，但是单纯的 LGWR 进程的写能力不可能像 DBWR 进程那样可以通过多个写进程来提高，所以这时候要考虑的是如何在单个 LGWR 进程的前提下让写的日志量不超过当前的 LGWR 写能力。这个可以从两个方面来考虑，一方面要考虑是否在应用中产生了太多无意义的重做日志，导致日志产生量太大，从而使日志的产生量超出了 LGWR 的写能力，如果是这样，那么考虑通过一些方法限制重做日志的产生。另一方面也要考虑如果日志产生量确定的情况下，如何让 LGWR 进程写日志能够写得更多更快，这主要取决于两个方面，一个是 LGWR 在写日志的时候是否发生了 I/O 竞争，另一方面是重做日志文件所在的磁盘速度是否过低，如果是竞争引起的，移动重做日志文件到其他的磁盘上，如果是磁盘速度引起的，那么选择高速磁盘存放重做日志。

通过前面的分析发现，主要的等待事件是与 LGWR 的写能力相关的，而 LGWR 写的重做日志都是由用户执行的 DML 语句产生的，那么现在就应该进一步分析，搞清楚问题到底是哪些会话中执行的哪条 SQL 语句引起的。

首先需要找到哪些会话产生了大量的 log file parallel write 等待事件，为了找到答案，就需要查看另外一些与会话相关的动态性能视图，会话级的视图有 v$session_event 和 v$session_wait，当然，由于要找的是当前发生了大量 log file parallel write 等待事件的会话，所以真正需要的视图应该是反映了当前会话等待信息的 v$session_wait 视图，通过这个视图可以找到是由哪些会话导致的这个等待。

在这里，将通过查询 v$session_wait 来得到产生 log file parallel write 等待事件最多的那些会话。当然，前面说过这个视图中的 P1、P2 和 P3 字段都是非常有用的字段，不过在这个案例中，对于 log file parallel write 这个事件来说，这几个字段是用不上的。如图 5-7 所示的查询显示了下面这 4 个会话产生了大量的 log file parallel write 等待事件。

```
SQL> @seswt.sql log

 SID Wait Event               Wait Stat W'd So Far （secs） Time W'd （secs）
---- -------------------------- --------  ----------------- ---------------
  13 log file parallel write     WAITING                0               0
  97 log file parallel write     WAITING                0               0
 176 log file parallel write     WAITING                0               0
 122 log file parallel write     WAITING                0               0

4 rows selected.
```

图 5-7 产生主要等待事件的会话

现在已经找到哪些会话是导致 log file parallel write 等待的主要来源，但是还需要更进一步定

位，找到是哪些语句引起的等待。找寻系统中执行过的 SQL 语句一般通过 v$sql_area，从这个视图中可以找到自系统启动以来放在共享池中的所有 SQL 语句，如图 5-8 所示就是按照最消耗资源的顺序排列显示了这些语句。

```
SQL> @sqls1.sql 1000 1000

Stmt Addr         Disk Rds    Buff Gets   Sorts    Runs Body Loads
-------------- ------------ ------------ ------- -------- ----------
2188ED68       276,390,597  278,896,280      1   65,628         1
2187A230            42,435   42,315,278      11,021,345         1
21668D10           189,628      576,659       0       55         1
216DD948            92,147    1,054,969       0   3,267         1
2172F948             3,321    2,424,735       0 377,248         1
219BB408            15,231       70,876       0       48         1
2168C058             4,949        4,373       2        1         1
2168CBB8             4,295        4,387       1        1         1
216978EC             4,245        4,253       1        1         1
2181D5E8             4,179        4,249       1        1         1
2164CBB4             2,862        9,908       1        1         1
2198901C             1,611       91,968       0       48         1
……

1000 rows selected.
```

图 5-8　按资源消耗的顺序排序

现在需要找到的是引起大量 log file parallel write 等待事件的那些 SQL 语句，所以上面的查询对我们来说也是没有用的，那么怎么得到我们所关心的这些语句呢？在前面已经得到了那些产生等待事件的会话信息，那么通过这些会话信息与 v$sqltext 视图进行关联查询，就可以得到相关的 SQL 语句了。从上面的 4 个会话中随便找出一个，看看这个会话执行的哪条语句带来了大量 log file parallel write 等待。例如，取出 13 号会话，执行下面的查询来找到这个会话执行的 SQL 语句的地址信息，如图 5-9 所示。

```
SQL> @Sqlst.sql 13

  SID User Nam CPU（sec） IO Read（k） IO Write（k） SQL Address
----- -------- -------- ----------- ------------  -----------
   13 WEBBER          0          11          5120   2168CBB8

1 rows selected.
```

图 5-9　SQL 语句的地址信息

通过前面的查询，找到了相关 SQL 语句的地址为 2168CBB8，通过这个语句地址可以在 v$sqltext 中进行查询，这样就找到了这条 SQL 语句，如图 5-10 所示。

定位到的这条语句是一条 update 语句，所以会产生大量的重做日志信息。而这条语句比较特殊的地方是 where 条件中的 to_char（STATUS），一般都知道，带有函数或者表达式的 where 语句，字段上的索引是不会被用到的，所以在这里执行的这条语句，将执行全表扫描，这也就解释了为什么存在大量的 db file scattered read 等待了。

```
SQL> @sqls2.sql 2168CBB8

SQL Statement Text
----------------------------------------------------------------
UPDATE HITS SET STATUS=:b1 WHERE to_char（STATUS） = :b2

1 row selected.
```

图 5-10 找到 SQL 语句

现在引起问题的原因看上去已经找到了，但是否真的就这么结束了呢，回顾一下刚才提到的 log file parallel write 等待事件的主要原因：这种事件发生通常表示日志文件发生了 I/O 竞争或者文件所在的驱动器较慢，也可能代表着应用中产生了太多无意义的重做日志，导致日志产生量太大。那么真正的原因是什么呢？

如果收集和分析数据仅仅局限于数据库的这些动态性能视图，那么真正的原因可能并不能真正找到，事实上，在做优化工作的时候，操作系统的信息也是非常重要的一种分析数据。那么来看看有哪些操作系统的信息是值得关心的，DBA 又应该怎样利用操作系统的这些信息来分析性能瓶颈的原因。

关于收集操作系统的工具，实际上系统管理员应该更为熟悉，而 DBA 所要掌握的，仅仅是帮助比较大的一些工具。DBA 所需要关心的性能信息主要包括了 CPU、I/O、内存、SWAP 交换这些，在 UNIX 系统上，常用的收集这些信息的工具有 Top、Sar、iostat、vmstat 等，在这个案例中，就通过其中的 vmstat 来进行收集，并且利用这些收集的信息找到真正导致 log file parallel write 等待事件发生的原因。

如图 5-11 所示的是使用 vmstat 收集的信息，具体的用法就不多讲了。在这里可以看到，系统中 idle 的 CPU 仍然很多（“ id ”列 80%左右），但是关于阻塞队列（“ b ”列是由 I/O 引起的队列）却非常多，这说明在操作系统级已经出现了 I/O 阻塞，而这种系统级的 I/O 阻塞，也就是导致 log file parallel write 等待事件发生的真正原因了。

```
$ vmstat 10 10
…
  procs          memory    swap         io       system     cpu
 r  b  w   swpd   free   buff  cache  si  so   bi    bo   in    cs  us  sy  id
 2  0  0  44736   1480   4300  23972   0  27   124    80 1729   204  17   7  76
 0  2  0  44736   1168   4316  24276   0   0   244    35 2330   215  20  11  69
 1  2  1  44984   1424   3832  24756   0  25    75    31  946   100  69  25   7
 1  4  0  45412   1528   4796  24160  10  47     5   139 1258   280  32  12  57
 1  1  0  45408   1080   4812  24524  13   0    12   107 1050   227  54  15  31
 1  2  0  45708   1468   5740  23552   1  31    31    95 1110   197  10   4  86
 2  5  0  45620   1048   8580  21008  13   0    35   150 1581   320  17   7  77
 0  4  0  45540    736  11348  18408  22   0    93   128 1868   285  13   6  81
 1  2  0  45540    848  13308  16288   0   0   128   153 2344   328  19   7  74
 0  1  0  45528   1400  12412  16624   1   0   114   129 2049   280  19   5  76
 0  3  0  45484   1196  12696  16508   5   1   100   129 1922   283  15   5  80
 1  3  0  45484   1520  11884  16996   0   0   100    92 1643   215  14   5  81
 1  4  0  45484   1496  12392  16512   0   0    45    98 1245   218   9   6  85
 0  2  0  45484    732  11748  17924   0   0    67   114 1542   244  12   6  82
……
```

图 5-11 vmstat 收集的信息

5.5　小结

通过上面的分析，可以看到系统中出现了两种明显的等待事件，一种是与全表扫描相关的 db file scattered read 等待事件，这种事件在整个系统中出现很多；而另一种是 log file parallel write 等待事件，这种事件则是当前主要的性能瓶颈。进一步分析发现，一些会话中的 SQL 语句产生了大量的重做日志信息，从而导致了这种等待事件的发生。通过对 SQL 语句的分析和操作系统信息的收集，最终得到了以下结论：

- 了解应用的需求，确定为什么要使用 to_char 函数屏蔽 status 字段索引的使用。
- 如果必要，调整收集到的 SQL 语句书写，减少全表扫描的发生，从而减少相应的 I/O 操作。
- 考虑在索引上使用 NOLOGGING，减少表中数据维护相应的索引维护产生的重做日志。
- 减少重做日志文件所在磁盘上无关紧要的 I/O 操作，如果必要，将重做日志文件移动到新的磁盘上。
- 从长远考虑，增加重做日志文件组的个数和组员文件的大小以防止其他重做日志相关的等待事件的发生。

5.6　附录

- syssum.sql

```
SELECT EVENT "Wait Event", TIME_WAITED "Time Waited",
      TIME_WAITED / （SELECT SUM（TIME_WAITED）
                                FROM v$system_event） "%Time waited",
      TOTAL_WAITS "Waits",
      TOTAL_WAITS / （SELECT SUM（TOTAL_WAITS）
                               FROM v$system_event） "%Waited"
from v$system_event
order by 3 desc ;
```

- cr_base.sql

```
create table sys_b（
EVENT VARCHAR2（64）,
TIME_WAITED NUMBER,
TOTAL_WAITS NUMBER,）;

create table sys_e（
EVENT VARCHAR2（64）,
TIME_WAITED NUMBER,
TOTAL_WAITS NUMBER,）;
```

- in_base.sql

```
insert into sys_b
select EVENT, TIME_WAITED, TOTAL_WAITS
from v$system_event;
```

- re_base.sql

```
insert into sys_e
select EVENT, TIME_WAITED, TOTAL_WAITS
from v$system_event;

create table sys_dif
as
select  e. EVENT
       e.TIME_WAITED - b.TIME_WAITED  TIME_WAITED,
       e.TOTAL_WAITS - b.TOTAL_WAITS  TOTAL_WAITS
from sys_b b , sys_e e
where b.EVENT=e.EVENT;;

SELECT EVENT "Wait Event", TIME_WAITED "Time Waited",
      TIME_WAITED / (SELECT SUM(TIME_WAITED) FROM sys_dif) "%Time waited",
      TOTAL_WAITS "Waits",
      TOTAL_WAITS / (SELECT SUM(TOTAL_WAITS) FROM sys_dif) "%Waited"
from sys_dif
order by 3 desc ;

drop table sys_dif;
drop table sys_d;
drop table sys_e;
```

- seswa.sql

```
select EVENT "Wait Event",
      count(SECONDS_IN_WAIT) "Waited So Far (sec)",
      count(SID) "Num Sess Waiting"
from v$session_wait
group by EVENT;
```

- seswt.sql

```
SELECT SID, EVENT "Wait Event", STATE "Wait Stat",
      WAIT_TIME "W'd So Far (secs)", SECONDS_IN_WAIT "Time W'd (secs)"
FROM v$session_wait
WHERE EVENT LIKE '&a&'
ORDER BY 5;
```

- sqls1.sql

```
SELECT * FROM
      (SELECT  ADDRESS "Stmt Addr",
            DISK_READS "Disk Rds",
            BUFFER_GETS "Buff Gets",
            SORTS "Sorts",
            EXECUTIONS "Runs",
            LOADS "Body Loads"
      FROM V$SQLAREA
      WHERE DISK_READS > &A
      ORDER BY DISK_READS)
 WHERE ROWNUM < &B;
```

- Sqlst.sql

```
 select cpu.sid "SID", cpu.username "User Name", cpu.value "CPU(sec)",
      reads.value "IO Read(k)", writes.value "IO Write(k)"
 from
```

```
    (select a.sid sid, a.username username, b.name, c.value value,a.serial# serial#
     from v$session a, v$statname b, v$sesstat c
     where a.sid = c.sid and b.statistic# = c.statistic# and b.name ='CPU used by this session' ) cpu,
    (select a.sid, a.username, b.name, c.value value from v$session a, v$statname b, v$sesstat c
     where a.sid = c.sid and b.statistic# = c.statistic#  and b.name ='physical reads' ) reads,
    (select a.sid, a.username, b.name, c.value value from v$session a, v$statname b, v$sesstat c
     where a.sid = c.sid and b.statistic# = c.statistic#  and b.name ='physical writes' ) writes
where cpu.sid = reads.sid and reads.sid = writes.sid and cpu.username is not null;
```

- sqls2.sql

```
select SQL_TEXT "SQL Statement Text"
from v$sqltext
where ADDRESS=&a;
```

作者简介

Ora-600，现任 ITPUB OCP 版块版主，擅长 Oracle DBA 技术、需求分析、系统分析与设计、数据建模、代码开发等，8i/9i OCP 认证获得者。

曾参与国土资源厅国土资源土地管理系统、土地整理预算系统、北京公交一卡通项目客服系统和 ED 卡管理系统、武威地区电力局的商务办公自动化系统和电力 MIS 系统、西北电力设计院商务办公自动化系统等项目，负责 Oracle 数据库后台设计、规划、管理、开发报表系统、培训等各项工作，曾在北京多家培训中心进行 Oracle OCP 课程讲授，并为全国多个行业客户提供技术服务和支持。

6 第6章 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。

本章就 SQL_TRACE/10046 事件的使用做简单探讨，并通过具体案例对 SQL_TRACE 的使用进行说明。

6.1 SQL_TRACE 及 10046 事件的基础介绍

本节将对 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。

6.1.1 SQL_TRACE 说明

先来关注一下 Oracle 官方文档（Oracle9iR2 文档）对 SQL_TRACE 的说明，如表 6-1 所示。

表 6-1 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

注 意

从 Oracle 10g 开始，SQL_TRACE 成为了动态参数。

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。

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，一定不要设置这个参数。如果只是对特定的会话启用跟踪，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使对性能的影响最小化：

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。

在 Oracle 10g 中可以在全局动态启用：

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

```
SQL*Plus: Release 10.1.0.2.0 - Production on Tue Mar 8 23:39:18 2005
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 08-MAR-05 >alter system set sql_trace=true;

System altered.
```

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）、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地调用，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 SET_INT_PARAM_IN_SESSION：

```
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。

6.1.2　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 session 级设置。

1. 在全局设置

在参数文件中增加以下：

```
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.
```

6.1.3 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
```

6.1.4 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
```

6.2　案例分析之一

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。

6.2.1　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。

操作系统为 Sun OS 5.8，数据库版本为 8.1.7。

系统症状：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。

6.2.2　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.
```

6.2.3 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。

6.2.4　登录数据库检查相应表结构

登录数据库，获得表结构信息：

```
SQL> select index_name,table_name,column_name from user_ind_columns
  2  where table_name=upper('categoryarticleassign');

INDEX_NAME                     TABLE_NAME                     COLUMN_NAME
------------------------------ ------------------------------ ---------------
IDX_ARTICLEID                  CATEGORYARTICLEASSIGN          ARTICLEID
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.
```

可以注意到，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。

6.2.5 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。

6.2.6　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。

6.3　案例分析之二

6.3.1　问题描述

很多时候，在进行数据库操作时，比如 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，很多时候无法确定问题出在何处。

本案例就这一类问题提供了一个解决思路及方法，供大家参考。

6.3.2 drop user 出现问题

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 所做的工作可能比我们想象的要复杂得多。

6.3.3 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

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0      0.00       0.00          0          0          0          0
Execute      1      0.00       0.00          0          0          0          0
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.00       0.00          0          0          0          0

Misses in library cache during parse: 0
Misses in library cache during execute: 1
```

```
Optimizer goal: CHOOSE
Parsing user id: SYS
********************************************************************************

drop user wapcomm

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      0      0.00       0.00          0          0          0          0
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        1      0.00       0.00          0          0          0          0

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……

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          4          0
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.00       0.00          0          0          4          0

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: SYS   (recursive depth: 1)

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;

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          2          0
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.00       0.00          0          0          2          0

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 31    (recursive depth: 1)
********************************************************************************

alter session set sql_trace=false


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        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.00       0.00          0          0          0          0

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 错误是由于 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。

6.3.4 问题定位

对于本案例，通过 MetaLink（http://metalink.oracle.com）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。

6.3.5　实际处理

MDSYS.SDO_GEOM_METADATA_TABLE 为 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。

6.3.6 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。这是深入研究学习 Oracle 的必经之途。

6.4 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。

同样的，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。

6.4.1 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> col name for a30
SQL> col p1 for a10
SQL> col p2 for a10
SQL> col p3 for a10
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
```

6.4.2 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 8 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_ora_20912.trc

    SQL> !
     [oracle@eygle rac]$ cat /opt/oracle/admin/rac/udump/rac1_ora_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
```

可以看到此时 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。

6.4.3 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)
 …
 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。

6.4.4　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：

```
Max(db_file_multiblock_read_count) = MaxOsIOsize/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
```

Oracle 所支持的最大 db_file_multiblock_read_count 值为 128。

也可以通过 db_file_multiblock_read_count 来测试 Oracle 在不同系统下，单次 IO 最大所能读取的数据量：

```
$ 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
```

本文以介绍 10046 事件为主，不再过多讨论 db_file_multiblock_read_count 的内容。

6.4.5　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。

参考信息

Eygle：　http://www.eygle.com/case/Use.sql_trace.to.Diagnose.database.htm
http://www.eygle.com/case/sql_trace_1.htm
http://www.eygle.com/case/sql_trace_2.htm

MetaLink：Note:291239.1 What is the value of SSTIOMAX and ......?
Note:131530.1SSTIOMAX AND DB_FILE_MULTIBLOCK_READ_COUNT

作者简介

盖国强，网名 eygle，ITPUB Oracle 管理版版主，ITPUB 论坛超级版主，曾任 ITPUB MS 版主。CSDN eMag Oracle 电子杂志主编。

曾任职于某国家大型企业，服务于烟草行业，开发过基于 Oracle 数据库的大型 ERP 系统，属国家信息产业部重点工程。同时负责 Oracle 数据库管理及优化，并为多家烟草企业提供 Oracle 数据库管理、优化及技术支持。

目前任职于北京某电信增值业务系统提供商企业，首席 DBA，负责数据库业务。管理全国 30 多个数据库系统。项目经验丰富，曾设计规划及支持中国联通增值业务等大型数据库系统。

实践经验丰富，长于数据库诊断、性能调整与 SQL 优化等。对于 Oracle 内部技术具有深入研究。

高级培训讲师，培训经验丰富，曾主讲 ITPUB DBA 培训及 ITPUB 高级性能调整等主要课程。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

可以在 http://www.eygle.com 上找到关于作者的更多信息。

第二篇
存储优化篇

本篇共分 6 章，主要内容如下：

第 7 章　表空间的存储管理与优化技术

第 8 章　关于 Oracle 数据库中行迁移/行链接的问题

第 9 章　HWM 与数据库性能的探讨

第 10 章　调整 I/O 相关的等待

第 11 章　Oracle 在 Solaris 的 VxFS 上的异步 I/O 问题

第 12 章　关于 Freelists 和 Freelist Groups 的研究

第 7 章　表空间的存储管理与优化技术

本章重点介绍了以下几个内容：数据库的逻辑存储结构——表空间（TABLESPACE）、字典管理表空间（DMT）的特性以及相应缺点、字典管理表空间的优化方法、本地管理表空间（LMT）的特性以及相应优点、Oracle 9i 的表空间类型以及相应优化、段自动管理表空间的特点、Oracle 10g 的表空间的特点及相应优化。

7.1　表空间的作用与分类

表空间是数据库中最大的逻辑存储结构，为数据库提供使用空间，其对应的物理结构是数据文件，一个表空间可以包含多个数据文件，但是一个数据文件只能属于一个表空间。表空间所包含的数据文件的大小，也就决定了表空间的大小，所以，表空间也是逻辑结构连接到物理结构的一个纽带。

数据库、表空间、数据文件的关系如图 7-1 所示。

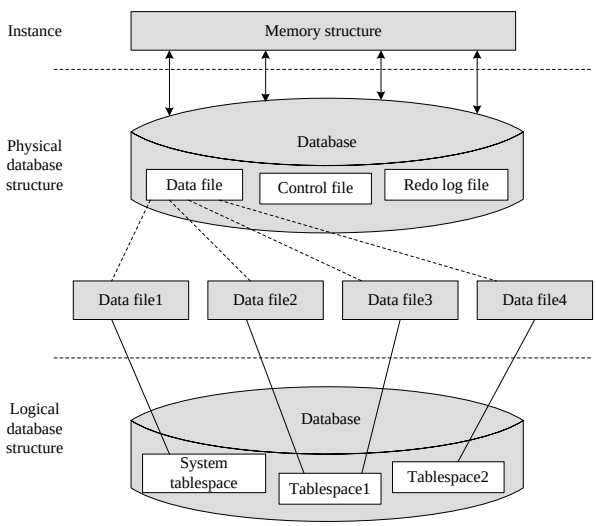

图 7-1　数据库、表空间与数据文件

既然表空间为数据库提供使用空间，它就必须有自己的空间管理办法，在表空间中增加、删除段的时候，数据库就必须跟踪这些段空间的使用。

假定一个新创建的表空间包含了 5 个表，如下所示：

表一……表二……表三……表四……表五……未用空间

当删除表四的时候，就有以下结果：

表一……表二……表三……空闲空间段……表五……未用空间

很明显，Oracle 需要有一个机制来管理表空间中各数据文件的这些分配的或未分配的空间，为了跟踪这些可以使用的空间（包括未分配使用的和可以重复使用的空间），对于每一个空间，必须知道以下情况：

- 这个可用空间位于什么数据文件。
- 这个空间的尺寸是多大。
- 如果它被占用了，是哪一个段占用的这个空间。

在空间的管理方式上，Oracle 推出了 3 种主要的空间管理方式。

一种是 8i 以前的字典管理方式（DMT），这种方式是在数据字典中管理可用空间和未用空间的，Oracle 通过一个递归 SQL 语句到该字典表中请求空间。

另外一种就是 8i 以后的本地管理模式（LMT），本地管理模式完全放弃以前的管理方法，通过在数据文件的头部建立位图区域来管理空间的分配，在一定程度上避免了并发上的冲突；而且本地管理表空间通过存储上统一的空间管理并取消了为独立的段的 NEXT 存储参数，也解决了表空间一直以来头疼的碎片问题。

还有一种自动区段空间管理（ASSM）是 9iR2 推出的一种表空间级别的段空间管理模式，ASSM 表空间是通过将 SEGMENT SPACE MANAGEMENT AUTO 子句添加到本地管理表空间的定义句法里而实现的。通过使用位图数组取代传统单向的链接列表（Freelist），ASSM 表空间会将链接列表的管理自动化，并取消为独立的段指定 PCTUSED、FREELISTS 和 FREELIST GROUPS 存储参数。

注　意

ASSM 表空间一定就是 LMT 表空间。

7.2　字典管理表空间

7.2.1　字典管理表空间的特性

在 8i 以前（不包括 8i），只存在一种表空间的管理模式，这就是字典管理表空间。其主要语法如下：

```
CREATE TABLESPACE 表空间名字
   DATAFILE '数据文件详细信息'  [SIZE INTETER [K|M]
    |[DEFAULT STORAGE]|[PERMANENT|TEMPORARY]]
```

关键字 DEFAULT STORAGE 指明了该表空间的默认存储格式，包含了 INITIAL、NEXT、PCTINCREASE 等相关参数的设置，如果创建在该表空间中的对象不指明存储参数的话，将采

用表空间的默认存储参数；PERMANENT|TEMPORARY 指明了该表空间的类型是永久的还是临时的。

为了确保能保存以上空间分配的信息，Oracle 用了两个数据字典表：UET$（已使用的空间）和 FET$（空闲空间）来保存表空间的空间使用与释放的信息，在涉及到空间分配的时候，Oracle 使用一个递归 SQL 语句到该字典表中请求空间。

7.2.2　字典管理表空间的缺点

字典表空间由于本身设计上的问题，存在以下缺陷。

1. 并发等待

通过查询 UET$或 FET$，可以看到，每个已使用空间段或空闲空间段（不一定是一个 extent，可以是多个 extent）都在该表中对应了一行，如下所示：

```
SQL> select * from UET$;
SEGFILE#  SEGBLOCK#   EXT#     TS#     FILE#   BLOCK#   LENGTH
--------  ---------  ------  ------  --------  -------  ---------
1            127       0        0        1    127         2
1            109       0        0        1    109         4
1            119       0        0        1    119         2
1             42       0        0        1     42         2
1            133       0        0        1    133        10
1             51       0        0        1     51         2
1             69       0        0        1     69        10
……

SQL> select * from  FET$;
      TS#     FILE#    BLOCK#    LENGTH
----------  ----------  ----------  ----------
        1         2      1090        64
        1         2      2626       128
        2         3         2        99
        3         4        82         8
        3         4        74         8
……
```

它的工作方式是当建立一个新的段或者段在表空间中请求新的空间时，Oracle 会通过一个递归 SQL 语句来完成这个工作，移动或增加相应的行到 UET$中，并改变相应 FET$；当删除一个段的时候，Oracle 将移动 UET$中相应的行到 FET$。这个过程的发生是连续的、串行的，极可能发生等待。当并发性很高的时候，将产生数据字典的争用。

而且数据字典表的信息发生改变，同时也使用了在系统表空间里的回滚段，引起空间争用，因为每一个字典改变的操作都需要记入系统回滚段，而且当一个段有几万或者几十万个区间的时候（对应字典表的几十万条记录），不用说管理该字典表的负担增加，就是对该段的 drop 操作都会变得异常困难，甚至导致系统回滚段空间不够而失败。

2. 空间碎片

当段的空间很不连续或表空间有大量的碎片就会引起数据库性能上的下降。因为字典管理表空间没有任何措施可以保证段的所有区间是相邻存储的。当要满足一个空间要求时，数据库不再合并相邻的自由范围（除非别无选择）， 而是寻找表空间中最大的自由范围来使用。这样将逐渐形成越来越多的、离散的、分隔的、较小的自由空间（即碎片）。随着时间推移，基于数据库的应用系统的广泛使用，产生的碎片就会越来越多，这将对数据库有以下两个主要影响：

- 导致系统性能减弱，如上所述，当要满足一个空间要求时，数据库将首先查找当前最大的自由范围，而“最大”自由范围逐渐变小，要找到一个足够大的自由范围已变得越来越困难，从而导致表空间中的速度障碍，使数据库的空间分配愈发远离理想状态。
- 浪费大量的表空间，尽管有一部分自由范围（如表空间的 PCTINCREASE 为非 0）将会被 SMON（系统监控）后台进程周期性地合并，但始终有一部分自由范围无法自动合并，浪费了大量的表空间。而非 0 的 PCTINCREASE 容易导致更多的碎片。

7.2.3 字典管理表空间的优化

从以上一系列操作中，可以看到，UET$记录了任何使用的段的区间，如果区间数太多，将给表 UET$的操作带来一定压力，所以在字典管理的表空间中，一般要求区间数少一点比较好，除非有特殊要求，一般建议在 20 个区间以下。还有一点就是如果发生连续的空间请求，将导致 Oracle 在两个字典表之间的操作等待，对于并发性很高的数据库来说，这是一个高昂的操作，所以，可以采用预分配空间的方式，并不断地监控段的空间使用情况（大小、区间数），这样就可以在很大程度上解决并发处理带来的额外开销。

另一个方面，可以指定在字典管理表空间中的所有段都具有 PCTINCREASE=0 的特性，保证每个区间的大小相等，然后可以设定每个区间的大小等于某一个特定数的整数倍，如等于 n * db_block_size * db_file_multiblock_read_count，这样可以在很大程度上防止表空间的碎片化。

7.3 本地管理表空间

7.3.1 本地管理表空间的特性

在 Oracle 8i 的版本中，Oracle 推出了一种全新的表空间管理方式：本地化管理的表空间。所谓本地化管理，就是指 Oracle 不再利用数据字典表来记录 Oracle 表空间里面的区的使用状况，而是在每个表空间的数据文件的头部加入了一个位图区，在其中记录每个区的使用状况。每当一个区被使用，或者被释放以供重新使用时，Oracle 都会更新数据文件头部的这个记录，来反映这个变化。

本地化管理的表空间的创建过程的主要语法如下：

```
CREATE TABLESPACE 表空间名字
       DATAFILE '数据文件详细信息'
       [EXTENT MANAGEMENT { LOCAL
       {AUTOALLOCATE | UNIFORM [SIZE INTETER [K|M] ] } } ]
```

关键字 EXTENT MANAGEMENT LOCAL 指定这是一个本地化管理的表空间。对于系统表空间，只能在创建数据库的时候指定 EXTENT MANGEMENT LOCAL，因为它是数据库创建时建立的第一个表空间。

在 Oracle 8i 中，字典管理还是默认的管理方式，当选择了 LOCAL 关键字，即表明这是一个本地管理的表空间。当然还可以继续选择更细的管理方式：是自动分配（AUTOALLOCATE）还是统一尺寸（UNIFORM）。若为自动分配，则表明让 Oracle 来决定区块的使用办法；若选择了统一尺寸，则还可以详细指定每个区间（Extent）的大小，若不加指定，则默认每个区使用 1MB 大小。

在表空间的空间管理上，Oracle 将存储信息保存在表空间的头部的位图中，而不是保存在数据字典中。通过这样的方式，在分配或者回收空间的时候，表空间就可以独立地在数据文件头部完成操作而不用与其他对象打交道。

也因为仅仅需要操作数据文件头部几个块，而不用操作数据字典，所以 Oracle 在本地管理的表空间中添加、删除段的时候，效率要比字典管理的表空间快，特别是在并发性很强的空间请求中。

对于在本地管理的表空间内部创建的对象而言，NEXT 扩展子句是过时的，因为由本地管理的表空间会自动管理它们。但是，INITIAL 参数仍然是需要的，因为 Oracle 不可能提前知道初始段加载的大小。

在本地管理的表空间中，如果数据库块尺寸（db_block_size）为 16KB 或 16KB 以下，数据文件头是 64KB 保留空间，若是 32KB 的块尺寸，则保留 128KB，以默认的 8KB 的块大小而言，就是 8 个块用在数据文件头部用于系统消耗，其中的 3～8 块用于记录区间的位图信息，如图 7-2 所示是块大小为 8KB 时数据文件的头部结构。

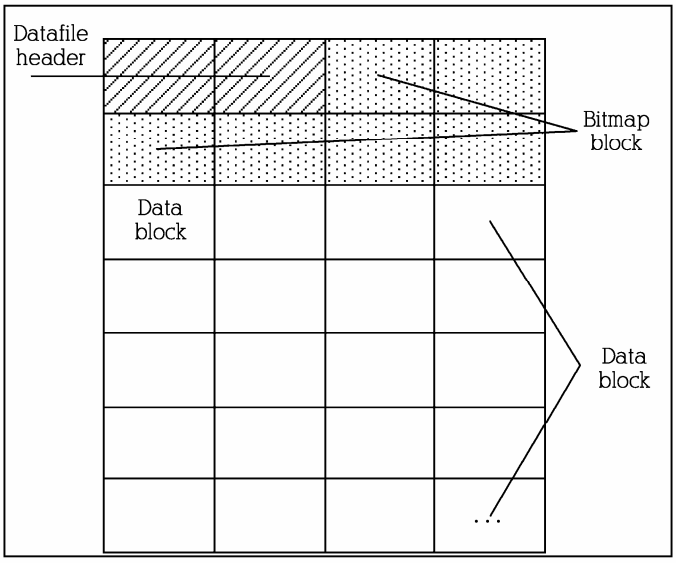

图 7-2　本地管理表空间

7.3.2 管理位图块的内部结构

1. 统一尺寸的本地管理表空间

如果 dump 统一尺寸方式的本地管理表空间包含的数据文件的第三个块，就可以得到类似以下的信息：

```
Start dump data blocks tsn: 5 file#: 5 minblk 3 maxblk 3
buffer tsn: 5 rdba: 0x01400003 (5/3)
scn: 0x0000.202f7c64 seq: 0x01 flg: 0x00 tail: 0x7c641e01
frmt: 0x02 chkval: 0x0000 type: 0x1e=KTFB Bitmapped File Space Bitmap
File Space Bitmap Block:
BitMap Control:
RelFno: 5, BeginBlock: 9, Flag: 0, First: 19, Free: 63469
FFFF070000000000 0000000000000000 0000000000000000 0000000000000000
.....
```

注意其中的 FFFF07 转换为二进制为 1111,1111,1111,1111,0000,0111，注意字节交换，得到 1111,1111,1111,1111,1110,0000。

可以看到，该表空间用 19 个位（bit）来代表共分配的 19 个区间。

2. 自动分配的本地管理表空间

在自动分配的本地管理的表空间中，区间尺寸可能由以下尺寸组成：64KB、1MB、8MB、64MB 甚至是 256MB。但是不管多大，都有一个通用尺寸 64KB，所以 64KB 就是该表空间的一个位标记的大小。例如，同样地 dump 文件头的第三个块：

```
Start dump data blocks tsn: 19 file#: 12 minblk 3 maxblk 3
buffer tsn: 19 rdba: 0x03000003 (12/3)
scn: 0x0000.00f2959b seq: 0x01 flg: 0x00 tail: 0x959b1e01
frmt: 0x02 chkval: 0x0000 type: 0x1e=KTFB Bitmapped File Space Bitmap
File Space Bitmap Block:
BitMap Control:
RelFno: 12, BeginBlock: 9, Flag: 0, First: 800, Free: 62688
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
FFFFFFFF00000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
.....
```

可以看到这里共分配了 800 个位（bit），但是自动分配模式下的位大小并不一定等于 extent 大小，所以不一定是对应 800 个区间，如它可能对应的是 50 个 1MB 大小的区间，如 50*1024=800*64。

7.3.3 本地管理表空间的优点

Oracle 之所以推出了这种新的表空间管理方法，正是因为它的优点。下面就简单总结一下这种表空间组织方法的优点：

(1) 本地化管理的表空间用数据文件的头部记录来管理空闲块，避免了递归的空间管理操作，避免了利用系统回滚段因此带来的性能与空间问题。

（2）本地化管理的表空间避免了在数据字典相应表里面写入空闲空间、已使用空间的信息，从而减少了数据字典表的竞争，提高了空间管理的并发性。

（3）区的本地化管理自动跟踪表空间里的空闲块，减少了手工合并自由空间的需要。

（4）表空间里的区的大小可以选择由 Oracle 系统来决定，或者由数据库管理员指定一个统一的大小，避免了字典表空间一直头疼的碎片问题。

7.4　段自动管理表空间

7.4.1　段自动管理表空间的特性

在 Oracle920 以前，表的剩余空间的管理与分配都是由链接列表（Freelist）来完成的，因为链接列表存在串行的问题因此往往容易引起段头的争用与空间的浪费，而且还需要 DBA 花费大量的精力去管理这些争用并监控表的空间利用。

自动段空间管理（ASSM），它首次出现在 Oracle920 里。有了 ASSM，链接列表被位图数组所取代，它是一个二进制的数组，能够迅速有效地管理存储扩展和剩余区块（free block），因此能够改善分段存储的本质。

看看位图数组是如何实现链接列表的功能的。从使用区段空间管理自动参数创建表空间开始：

```
CREATE TABLESPACE demo
   DATAFILE  '/u01/oracle/demo01.dbf '
   SIZE  5M
   EXTENT MANAGEMENT LOCAL   -- Turn on LMT
   SEGMENT SPACE MANAGEMENT AUTO -- Turn on ASSM;
```

带有 ASSM 的本地管理表空间会略掉任何为 PCTUSED、NEXT 和 FREELISTS 所指定的值。Oracle 9i 会使用位图数组来自动地管理表空间里的对象空间使用。

新的管理机制用位图数组来跟踪或管理每个分配到对象的块，每个块有多少剩余空间根据位图的状态来确定，如>75%、50-75%、25-50%和<25%，也就是说位图其实采用了 4 个状态位来代替以前的 PCTUSED，什么时候该利用该数据块则由设定的 PCTFREE 来决定。

使用 ASSM 的一个巨大优势是，位图数组肯定能够减轻缓冲区忙等待（buffer busy wait）的负担，这个问题在 Oracle 9i 以前的版本里曾是一个严重的问题，因为在没有多个位图数组的时候，每个 Oracle 段在段的头部都曾有一个数据块用于链接列表，用来管理对象所使用的剩余区块，并为任何 SQL 申请的新数据行提供数据块。当数据缓冲内的数据块由于被另一个 DML 事务处理锁定而无法使用的时候，缓冲区忙等待就会发生。当需要将多个任务插入到同一个表格里的时候，这些任务就被强制等待，而 Oracle 会在同时分派剩余的区块，一次一个。

有了 ASSM 之后，Oracle 宣称其显著地提高了 DML 并发操作的性能，因为位图数组的不同部分可以被同时使用，这样就消除了寻找剩余空间的串行化。根据 Oracle 的测试结果，使用位图数组会消除所有分段头部（对资源）的争夺，还能获得超快的并发插入操作。如图 7-3 所示的是 ASSM 表空间一个数据文件的头部结构。

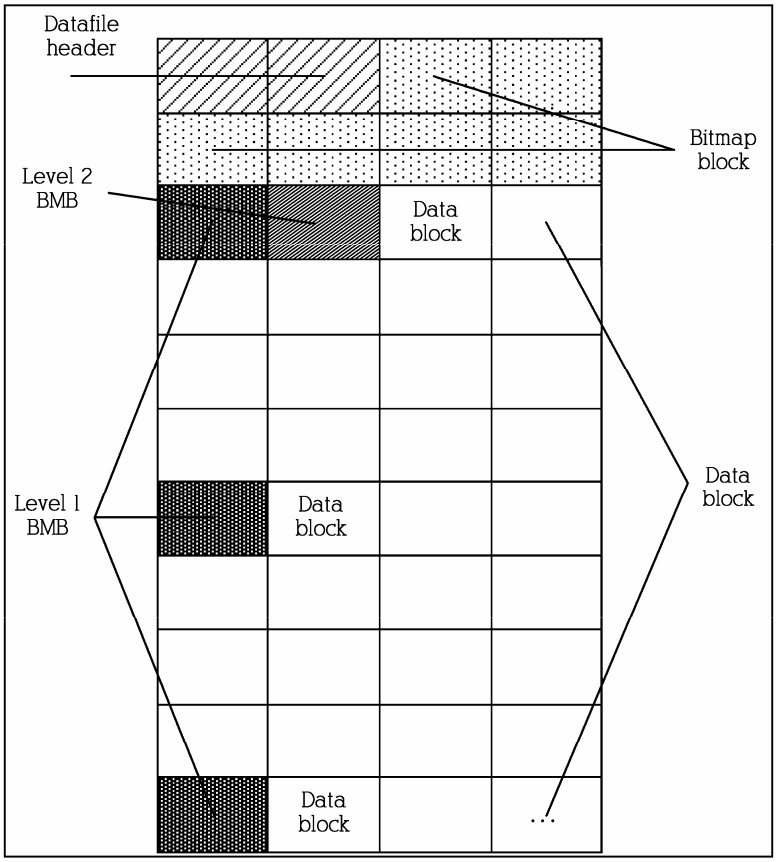

图 7-3 段自动管理表空间

7.4.2 位图管理段内部结构

如果 dump ASSM 表空间的第 10 个块，可以发现以下信息：

```
Start dump data blocks tsn: 6 file#: 7 minblk 10 maxblk 10
buffer tsn: 6 rdba: 0x0680000a (7/10)
scn: 0x0000.00181a39 seq: 0x01 flg: 0x04 tail: 0x1a392101
frmt: 0x02 chkval: 0x2738 type: 0x21=SECOND LEVEL BITMAP BLOCK
Dump of Second Level Bitmap Block
   number: 8      nfree: 8      ffree: 0     pdba:    0x0680000b
  opcode:0
 xid:
  L1 Ranges :
  -------------------------------------------------------
   0x06800009  Free: 5 Inst: 1
   0x06800019  Free: 5 Inst: 1
   0x06800029  Free: 5 Inst: 1
   0x06800039  Free: 5 Inst: 1
   0x06800049  Free: 5 Inst: 1
   0x06800059  Free: 5 Inst: 1
……
```

可以看到，块 10 是一个二级管理位图块，负责记录一级位图块的地址，其中的 9、19、29 等就是记录自由空间的一级位图块的地址，从这里可以看到，每个一级位图块管理 16 个（十六进制 10 个）块的存储信息。如果在并发的处理中，每个一级位图块可以单独地管理或者是分配空间，而不是再由以前的一个链接列表块来进行空间的管理，在实际的情况中，每个一级位图块也不一定是管理 16 个块的空间信息，也有可能是 32 个或者更多。

位图数据的级别可以分为 3 个级别，当存在一个或多个一级位图块（如块 9、19）的时候，将由二级位图（如块 10）块来保存一级位图块的地址，同理，一个二级位图块不够使用而出现多个二级位图块的时候，将由三级位图块来保存二级位图块的地址（由于三级位图块的出现需要很多数据块，所以这里不讨论三级位图块）。整个位图数组的结构形成一个树状结构，这将有利于 Oracle 跟踪所有的位图数据块的位置。

如果 dump 其中的一个一级位图块，如块 39，对应的是十进制的 57，可以发现以下：

```
Start dump data blocks tsn: 6 file#: 7 minblk 57 maxblk 57
buffer tsn: 6 rdba: 0x06800039 (7/57)
scn: 0x0000.0018b7cb seq: 0x04 flg: 0x04 tail: 0xb7cb2004
frmt: 0x02 chkval: 0x27d2 type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
 --------------------------------
  nbits : 4 nranges: 2       parent dba:  0x0680000a   poffset: 3
  unformatted: 8      total: 16       first useful block: 1
  owning instance : 1
  instance ownership changed at 08/19/2003 10:41:42
  Last successful Search 08/19/2003 10:41:42
  Freeness Status:  nf1 1     nf2 0     nf3 0     nf4 6

  Extent Map Block Offset: 4294967295
  First free datablock : 1
  Bitmap block lock opcode 0
  Locker xid:    :  0x0000.000.00000000
     Highwater::  0x06800041  ext#: 6     blk#: 8     ext size: 8
 #blocks in seg. hdr's freelists: 0
 #blocks below: 50
 mapblk  0x00000000  offset: 6
 HWM Flag: HWM Set
 ------------------------------------------------------
 DBA Ranges :
 ------------------------------------------------------
  0x06800039  Length: 8     Offset: 0
  0x06800041  Length: 8     Offset: 8

  0:Metadata   1:FULL   2:FULL   3:75-100% free
  4:75-100% free   5:75-100% free   6:75-100% free   7:0-25% free
  8:unformatted   9:unformatted   10:unformatted   11:unformatted
  12:unformatted   13:unformatted   14:unformatted   15:unformatted
 ------------------------------------------------------
End dump data blocks tsn: 6 file#: 7 minblk 57 maxblk 57
```

可以看到，在位图块 57 管理的 16 个数据块中，1 个位图+2 个 FULL+4 个 75-100% free+1 个 0-25% free+8 个未使用的，共 16 个块，如果在下次的插入或者更新中，位图块 57 将负责

这 16 个数据块的空间的分配与使用以及相应的状态记载，可以想象，除了这 16 个块之外的块的空间管理，将由其他类似块 57 的块来完成，多个位图块并行管理将明显地增加并发处理的能力。

7.4.3 段自动管理表空间的优化

尽管 ASSM 显示出了令人激动的特性并能够简化 Oracle DBA 的工作，但是 Oracle9iR2 的位图分段管理还是有一些局限性的：

- 一旦被 DBA 分配之后，它就无法控制表空间内部的独立表格和索引的存储行为。
- 大型对象不能够使用 ASSM，而且必须为包含有 LOB 数据类型的表格创建分离的表空间。
- 不能够使用 ASSM 创建临时的表空间。这是由排序时临时分段的短暂特性所决定的。
- 只有本地管理的表空间才能够使用位图分段管理。
- 使用超高容量的 DML（如 insert、update 和 delete 等）的时候可能会出现性能上的问题。

正因为 ASSM 还不是足够稳定与完善，所以至少在 9iR2 的版本上，还不建议在生产系统中大规模使用 ASSM 的表空间。

7.5 9i 对表空间的管理优化

7.5.1 自动 undo 管理的表空间

在 Oracle 9i 以前，回滚段全是手工管理与监控的，DBA 需要花费一定的时间去管理与监控回滚段的性能，创建不好或管理不好的回滚段，将引起很大的性能瓶颈。从 9i 开始，为了更好地管理回滚段，Oracle 默认采用自动回滚段管理。

自动回滚段管理可以最大限度地避免 8i 中比较有名的 ORA-01555“ 快照太老 ”的错误，Oracle 9i 通过以下 4 个初试化参数来设置自动回滚段管理：

```
undo_management                  string      AUTO
undo_retention                   integer     10800
undo_suppress_errors             boolean     FALSE
undo_tablespace                  string      UNDOTBS1
```

- undo_management 该参数设置回滚段管理采用自动方式，Oracle 建议采用自动方式，如果不是对数据库非常了解，不要修改该参数。
- undo_retention 该参数设置回滚信息在回滚段中保持的时间，单位是秒，默认 3 个小时，如果数据库的事务量特别大，可以适当地减少该参数值，以避免回滚表空间的膨胀，但是过小的值也将导致 ORA-01555 错误的重现以及 FLASHBACK QUERY 功能的局限。
- undo_suppress_errors 该参数设置不显示某些错误信息，如对系统回滚段的操作将不显示错误，虽然这个操作没有成功。
- undo_tablespace 该参数设置使用自动回滚的表空间，DBA 需要监控该表空间的大小。

自动回滚段的另外一个好处就是可以利用 FLASHBACK QUERY 来查看提交以前的数据或导出当前时间点以前的数据，防止一定程度上的人为错误。

7.5.2　完全本地的临时表空间

9i 默认的本地管理的临时文件，总是处于 NOLOGGING 模式，控制文件也不记录临时文件的位置，由于不记载 redo 信息，所以 9i 的临时数据文件不需要进行备份与恢复，如果发生意外，只需要重新创建一个即可。

另外，由于 9i 默认临时表空间的出现，减少了 9i 以前因为默认临时表空间是系统表空间而导致的表空间碎片问题。

7.6　Oracle 10g 对表空间的优化

SYSAUX 表空间在 Oracle Database 10g 中被引入，作为 SYSTEM 表空间的辅助表空间，这是一个管理及规划上的改进，进一步独立 SYSTEM 表空间，保证其存储及性能。以前版本中很多使用独立表空间或系统表空间的数据库组件，现在在 10g 新增的 SYSAUX 表空间中创建。通过分离这些组件和功能，SYSTEM 表空间的负荷得以减轻。反复创建一些相关对象及组件引起 SYSTEM 表空间的碎片问题将得以避免。而且，由于大量的独立表空间中的对象都被移往了 SYSAUX 表空间，使得 10g 的表空间数目变得很少，对于空间管理，备份与优化都是一个不错的方法。

而且如果 SYSAUX 表空间的不可用，数据库核心功能将保持有效，所以使用 SYSAUX 表空间的特点将会失败或功能受限，使数据库变得更稳定可靠。

7.7　小结

下面对本章的内容进行一下总结：

⑴ 了解字典管理表空间的工作原理，尽量减少空间分配的串行化与表空间的碎片化。

⑵ 了解本地管理表空间的工作原理，尽量使用空间管理的本地化来减少字典管理表空间带来的问题。

⑶ 了解段自动管理表空间的工作原理，理解链接列表的工作原理，理解 ASSM 对于大量并发处理的好处以及相关缺点。

⑷ 了解 9i 新型自动重作表空间的好处以及完全本地管理的临时表空间的优点。

⑸ 了解 10g 新型 SYSAUX 表空间出现的原因以及相应管理上的优化。

7.8　附录

1. 表空间的空间监控

表空间的空间使用其实是一个需要特别注意的问题，因为数据文件不可扩展而导致表空间的空间不够，就可能导致无法写入任何新的数据，甚至导致数据库的停止。以下的语句可以监控表空间的空间利用情况，如果使用了 9i 的完全临时表空间，则加入后半部分用于检测临时表空间。

```
SELECT  D.TABLESPACE_NAME,SPACE  "SUM_SPACE(M)",BLOCKS  SUM_BLOCKS,SPACE-NVL(FREE_SPACE,0)
```

```
"USED_SPACE(M)",
    ROUND((1-NVL(FREE_SPACE,0)/SPACE)*100,2) "USED_RATE(%)",FREE_SPACE "FREE_SPACE(M)"
    FROM
    (SELECT TABLESPACE_NAME,ROUND(SUM(BYTES)/(1024*1024),2) SPACE,SUM(BLOCKS) BLOCKS
    FROM DBA_DATA_FILES
    GROUP BY TABLESPACE_NAME) D,
    (SELECT TABLESPACE_NAME,ROUND(SUM(BYTES)/(1024*1024),2) FREE_SPACE
    FROM DBA_FREE_SPACE
    GROUP BY TABLESPACE_NAME) F
    WHERE  D.TABLESPACE_NAME = F.TABLESPACE_NAME(+)
    --如果采用了完全本地管理的临时表空间，就加入如下部分
    UNION ALL  --if have tempfile
    SELECT D.TABLESPACE_NAME,SPACE "SUM_SPACE(M)",BLOCKS SUM_BLOCKS,
    USED_SPACE "USED_SPACE(M)",ROUND(NVL(USED_SPACE,0)/SPACE*100,2) "USED_RATE(%)",
    NVL(FREE_SPACE,0) "FREE_SPACE(M)"
    FROM
    (SELECT TABLESPACE_NAME,ROUND(SUM(BYTES)/(1024*1024),2) SPACE,SUM(BLOCKS) BLOCKS
    FROM DBA_TEMP_FILES
    GROUP BY TABLESPACE_NAME) D,
    (SELECT TABLESPACE_NAME,ROUND(SUM(BYTES_USED)/(1024*1024),2) USED_SPACE,
    ROUND(SUM(BYTES_FREE)/(1024*1024),2) FREE_SPACE
    FROM V$TEMP_SPACE_HEADER
    GROUP BY TABLESPACE_NAME) F
    WHERE  D.TABLESPACE_NAME = F.TABLESPACE_NAME(+)
```

2. 段的空间利用监控

段的空间与区间的利用，在字典管理的表空间里尤为重要，如果一个对象的区间数太多，不但大大加重了字典表的管理负担与系统回滚段的压力，也严重影响对该段（如表或索引）的性能。该查询也可以看到现在利用的区间与最大区间数的差异，如果该差值已经很小，就需要注意新的空间的分配，避免因为不能分配新的区间而导致新数据的写入错误。

```
    SELECT S.OWNER,S.SEGMENT_NAME,S.SEGMENT_TYPE,S.PARTITION_NAME,
    ROUND(BYTES/(1024*1024),2) "USED_SPACE(M)",
    EXTENTS USED_EXTENTS,S.MAX_EXTENTS,S.BLOCKS ALLOCATED_BLOCKS,
    S.BLOCKS USED_BOLCKS,S.PCT_INCREASE,S.INITIAL_EXTENT,
    S.NEXT_EXTENT/1024 "NEXT_EXTENT(K)",S.TABLESPACE_NAME
    FROM DBA_SEGMENTS S
    WHERE S.OWNER = USER
    ORDER BY Used_Extents DESC
```

作者简介

陈吉平，ITPUB 的 ID 为 piner，现任职于国内某大型电子商务网站，主要负责网站后台数据库维护，擅长备份与恢复，数据库高可用性与系统容灾，对数据库优化也有很深的研究。希望能广交 Oracle 朋友，共同进步。

第 8 章　关于 Oracle 数据库中行迁移/行链接的问题

8.1　行迁移/行链接的简介

在实际工作中，经常会碰到一些 Oracle 数据库性能较低的问题。当然，引起 Oracle 数据库性能较低的原因是多方面的，但可以通过一些正确的设计和有效的诊断来尽量避免一些 Oracle 数据库性能上的问题，Row Migration（行迁移）和 Row Chaining（行链接）就是其中要尽量避免的引起 Oracle 数据库性能低下的潜在问题。通过合理地诊断行迁移/行链接，可以较大幅度地提高 Oracle 数据库的性能。

那究竟什么是行迁移/行链接呢，先从 Oracle 的 block 开始谈起。

操作系统的最小读写操作单元是操作系统的 block，所以当创建一个 Oracle 数据库的时候应该把数据库的 block size 设置成为操作系统的 block size 的整数倍。Oracle block 是 Oracle 数据库中读写操作的最小单元。在 Oracle 9i 之前的 Oracle 数据库版本中，Oracle block 一旦在创建数据库的时候被设定后就无法再更改。为了在创建数据库之前确定一个合理的 Oracle block 的大小，就需要考虑一些因素，如数据库本身的大小以及并发事务的数量等。使用一个合适的 Oracle block 大小对于数据库的调优是非常重要的。Oracle block 的结构如图 8-1 所示。

由图 8-1 可以看出，一个 Oracle block 由 3 个部分组成，分别是数据块头（Header）、自由空间（Free Space）和实际数据（Data）。

- 数据块头：主要包含数据块地址的一些基本信息和段的类型，以及表和包含有数据的实际行的地址。
- 自由空间：是指可以为以后的更新和插入操作分配的空间，其大小受 PCTFREE 和 PCTUSED 两个参数影响。
- 实际数据：是指在行内存储的实际数据。

当创建或者更改任何表和索引的时候，Oracle 在空间控制方面使用两个存储参数：

- PCTFREE:为将来更新已经存在的数据预留空间的百分比。
- PCTUSED:用于为插入一新行数据的最小空间的百分比。这个值决定了块的可用状态。可用的块是可以执行插入的块，不可用状态的块只能执行删除和修改，可用状态的块被放在

Freelist 中。

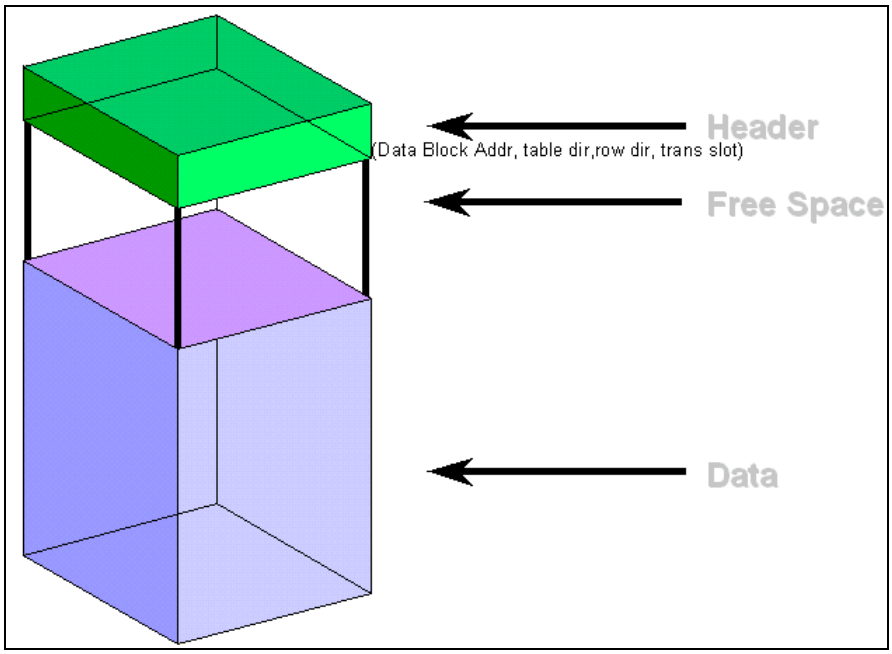

图 8-1 Oracle block 结构

当表中一行的数据不能在一个数据 block 中放入的时候，这个时候就会发生两种情况，一种是行链接，另外一种就是行迁移了。

行链接产生在第一次插入数据时，如果一个 block 不能存放一行记录的情况下。在这种情况下，Oracle 将使用链接一个或者多个在这个段中保留的 block 存储这一行记录，行链接比较容易发生在比较大的行上，例如行上有 LONG、LONG RAW、LOB 等数据类型的字段，这种时候行链接是不可避免地会产生的。

当一行记录初始插入时是可以存储在一个block中的，由于更新操作导致行长增加了，而block的自由空间已经完全满了，这个时候就产生了行迁移。在这种情况下，Oracle 将会把整行数据迁移到一个新的 block 中（假设一个 block 中可以存储下整行数据），Oracle 会保留被迁移行的原始指针指向新的存放行数据的 block，这就意味着被迁移行的 ROW ID 是不会改变的。

当发生了行迁移或者行链接，对这行数据操作的性能就会降低，因为 Oracle 必须要扫描更多的 block 来获得这行数据的信息。

下面举例详细说明行迁移/行链接的产生过程。

先创建一个 PCTFREE 为 20 和 PCTUSED 为 50 的测试表：

```
create table test(
col1 char(20),
col2 number)
storage (
pctfree 20
pctused 50);
```

当插入一条记录时，Oracle 会在 free list 中先去寻找一个自由的块，并且将数据插入到这个自由块中。而在 free list 中存在的自由块是由 PCTFREE 值决定的。初始的空块都是在 free list 中

的，直到块中的自由空间达到 PCTFREE 的值，此块就会从 free list 中移走，而当此块中的使用空间低于 PCTUSED 时，此块又被重新放到 free list 中。

Oracle 使用 free list 机制可以大大地提高性能，对于每次的插入操作，Oracle 只需要查找 free list 就可以了，而不是去查找所有的 block 来寻找自由空间。

假设第一次插入数据使用的一个空的 block，如图 8-2 所示。

假设插入第一条记录的时候占用一个 block 的 10%的空间（除去 block 头占去的大小），如图 8-3 所示，剩余的自由空间 90%大于 PCTFREE 20%，因此这个 block 还将继续为下次的插入操作提供空间。

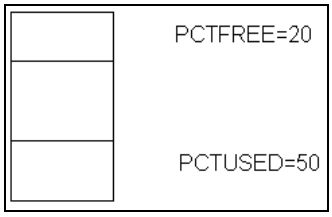

图 8-2　Oracle 空的 block 结构图

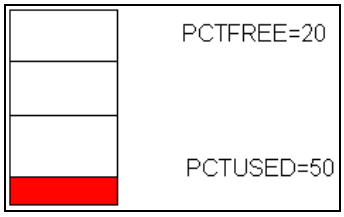

图 8-3　插入 10%后的 Oracle block 结构图

再连续插入 7 条记录，使 block 的剩余自由空间剩下 20%，如图 8-4 所示，此时，这个 block 将要从 free list 中移走，如果再插入记录，Oracle 将再 free list 中寻找下一个空余的 block 去存放后来插入的数据。

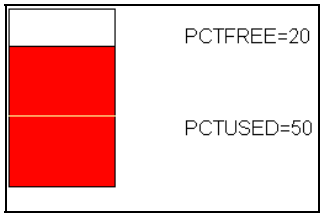

图 8-4　插入 80%后的 Oracle block 结构图

此时如果去更新第一条插入的记录，使其行长增加 15%，Oracle 将会使用这个 block 中剩余的 20%的自由空间来存放此行数据，如果再更新第二条记录，同样地使其行长增加 15%，而此 block 中只剩下 5%的自由空间，不够存放更新的第二条记录，于是 Oracle 会在 free list 中寻找一个有自由空间（10%+15%）的 block 来存放这行记录的 block 去存储，在原来的 block 中保存了指向新的 block 的指针，原来这行记录的 ROW ID 保持不变，这个时候就产生了行迁移。

而当插入一条新记录时，如果一个 blcok 不足以存放下这条记录，Oracle 就会寻找一定数量的 block 一起来容纳这条新的记录，这个时候就产生了行链接，行链接主要产生在 LOB、CLOB、BLOB 和大的 VA 行链接 HAR2 数据类型上。

具体通过下面的一个试验来查看行链接和行迁移是如何产生并在数据文件中体现出来的。

先查看 ALLAN 这个表空间的数据文件号，为了便于测试，这里只建立了一个数据文件。

```
SQL> select file_id from dba_data_files where tablespace_name='ALLAN';
  FILE_ID
----------
        23
```

创建一个测试表 test：

```
SQL> create table test ( x int primary key, a char(2000), b char(2000), c char(2000), d char(2000),
e char(2000) ) tablespace allan;
Table created.
```

因为数据库的 db_block_size 是 8KB，所以创建的表有 5 个字段，每个占 2000 个字节，这样一行记录大约 10KB，就会超过一个 block 的大小了。

然后插入一行记录：

```
SQL> insert into test(x) values (1);
1 row created.
SQL> commit;
Commit complete.
```

查找这行记录所在的 block，并 dump 出来：

```
SQL> select dbms_rowid.rowid_block_number(rowid) from test;
DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)
-----------------------------------
                             34
SQL> alter system dump datafile 23 block 34;
System altered.
```

在 udump 目录下查看 trace 文件的内容如下：

```
Start dump data blocks tsn: 34 file#: 23 minblk 34 maxblk 34
buffer tsn: 34 rdba: 0x05c00022 (23/34)
scn: 0x0000.013943f3 seq: 0x01 flg: 0x02 tail: 0x43f30601
frmt: 0x02 chkval: 0x0000 type: 0x06=trans data
Block header dump:  0x05c00022
 Object id on Block? Y
 seg/obj: 0x3ccd  csc: 0x00.13943ef  itc: 2  flg: O  typ: 1 - DATA
    fsl: 0  fnx: 0x0 ver: 0x01
Itl         Xid               Uba        Flag  Lck       Scn/Fsc
0x01   0x000a.02e.00000ad7  0x00800036.03de.18  --U-    1  fsc 0x0000.013943f3
0x02   0x0000.000.00000000  0x00000000.0000.00  ----    0  fsc 0x0000.00000000
data_block_dump,data header at 0xadb505c
===============
tsiz: 0x1fa0
hsiz: 0x14
pbl: 0x0adb505c
bdba: 0x05c00022
    76543210
flag=--------
ntab=1
nrow=1
frre=-1
fsbo=0x14
fseo=0x1f9a
avsp=0x1f83
tosp=0x1f83
0xe:pti[0]     nrow=1  offs=0
0x12:pri[0]    offs=0x1f9a
block_row_dump:
tab 0, row 0, @0x1f9a
tl: 6 fb: --H-FL-- lb: 0x1  cc: 1
col  0: [ 2]  c1 02
end_of_block_dump
End dump data blocks tsn: 34 file#: 23 minblk 34 maxblk 34
```

对相关的一些信息做一些解释。

- fb：H 是指行记录的头，F 是指行记录的第一列，L 是指行记录的最后一列。
- cc：列的数量。

- nrid：对于行链接或者行迁移来说的下一个 ROW ID 的值。

由上面的 dump 信息，可以看出来当前表 test 是没有行链接或者行迁移的，然后更新 test 表，并重新 dump 出来：

```
SQL> update test set a='test',b='test',c='test',d='test',e='test' where x=1;
1 row updated.
SQL> commit;
Commit complete.
```

此时应该有行迁移/行链接产生了。

```
SQL> alter system dump datafile 23 block 34;
System altered.
```

在 udump 目录下查看 trace 文件的内容如下：

```
Start dump data blocks tsn: 34 file#: 23 minblk 34 maxblk 34
buffer tsn: 34 rdba: 0x05c00022 (23/34)
scn: 0x0000.0139442b seq: 0x01 flg: 0x02 tail: 0x442b0601
frmt: 0x02 chkval: 0x0000 type: 0x06=trans data
Block header dump:  0x05c00022
 Object id on Block? Y
 seg/obj: 0x3ccd  csc: 0x00.1394429  itc: 2  flg: -  typ: 1 - DATA
     fsl: 0  fnx: 0x0 ver: 0x01
 Itl         Xid               Uba        Flag  Lck       Scn/Fsc
0x01   0x000a.02e.00000ad7  0x00800036.03de.18  C---    0  scn 0x0000.013943f3
0x02   0x0004.002.00000ae0  0x0080003b.0441.11  --U-    1  fsc 0x0000.0139442b
data_block_dump,data header at 0xadb505c
===============
tsiz: 0x1fa0
hsiz: 0x14
pbl: 0x0adb505c
bdba: 0x05c00022
    76543210
flag=--------
ntab=1
nrow=1
frre=-1
fsbo=0x14
fseo=0x178a
avsp=0x177c
tosp=0x177c
0xe:pti[0]     nrow=1  offs=0
0x12:pri[0]    offs=0x178a
block_row_dump:
tab 0, row 0, @0x178a
tl: 2064 fb: --H-F--N lb: 0x2  cc: 3
nrid:  0x05c00023.0
col  0: [ 2]  c1 02
col  1: [2000]
 74 65 73 74 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
 ……
col  2: [48]
 74 65 73 74 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

```
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
end_of_block_dump
End dump data blocks tsn: 34 file#: 23 minblk 34 maxblk 34
```

不难看出，nrid 出现了值，指向了下一个 ROW ID，证明刚刚的 update 操作使这行记录产生了行链接或者行迁移了。

8.2 行迁移/行链接的检测方法

通过前面的介绍可以知道，行链接主要是由于数据库的 db_block_size 不够大，对于一些大的字段没法在一个 block 中存储下而产生的。对于行链接，除了增大 db_block_size 之外没有别的任何办法可以避免，但是因为数据库建立后 db_block_size 是不可改变的（在 9i 之前），对于 Oracle 9i 数据库可以对不同的表空间指定不同的 db_block_size，因此行链接的产生几乎是不可避免的，也没有太多可以调整的地方。行迁移则主要是由于更新表的时候，由于表的 PCTFREE 参数设置太小，导致 block 中没有足够的空间去容纳更新后的记录，从而产生了行迁移。对于行迁移来说就非常有调整的必要了，因为这个是可以调整和控制清除的。

如何检测数据库中存在有了行迁移和行链接呢？可以利用 Oracle 数据库自身提供的脚本 utlchain.sql（在$ORACLE_HOME/rdbms/admin 目录下）生成 chained_rows 表，然后利用 ANALYZE TABLE table_name LIST CHAINED ROWS INTO chained_rows 命令逐个分析表，将分析的结果存入 chained_rows 表中。从 utlchain.sql 脚本中，可以看到 chained_rows 的建表脚本，对于分区表、cluster 表都是适用的。然后可以使用拼凑语句的办法生成分析所需要的表的脚本，并执行脚本将具体的分析数据放入 chained_rows 表中。例如，下面是分析一个用户下所有表的脚本：

```
SPOOL list_migation_rows.sql
SET ECHO OFF
SET HEADING OFF
SELECT 'ANALYZE TABLE ' || table_name || ' LIST CHAINED ROWS INTO chained_rows;' FROM
user_tables;
SPOOL OFF
```

然后查询 chained_rows 表，可以具体查看某张表上有多少的行链接和行迁移。

```
SELECT table_name, count(*) from chained_rows GROUP BY table_name;
```

当然，也可以查询 v$sysstat 视图中的“ table fetch continued row ”列来得到当前的行链接和行迁移数量。

```
SELECT name, value FROM v$sysstat WHERE name = 'table fetch continued row';
```

可以使用以下脚本来直接查找存在行链接和行迁移的表，自动完成所有的分析和统计。

```
accept owner prompt " Enter the schema name to check for Row Chaining (RETURN for All): "
prompt
prompt
accept table prompt " Enter the table name to check (RETURN for All tables owned by &owner): "
prompt
prompt
set head off serverout on term on feed off veri off echo off
!clear
prompt
declare
v_owner varchar2(30);
```

```
    v_table varchar2(30);
    v_chains number;
    v_rows number;
    v_count number := 0;
    sql_stmt varchar2(100);
    dynamicCursor INTEGER;
    dummy INTEGER;
    cursor chains is
    select count(*) from chained_rows;
    cursor analyze is
    select owner, table_name
    from sys.dba_tables
    where owner like upper('%&owner%')
    and table_name like upper('%&table%')
    order by table_name;
    begin
    dbms_output.enable(64000);
    open analyze;
    fetch analyze into v_owner, v_table;
    while analyze%FOUND loop
    dynamicCursor := dbms_sql.open_cursor;
    sql_stmt := 'analyze table '||v_owner||'.'||v_table||' list chained rows into chained_rows';
    dbms_sql.parse(dynamicCursor, sql_stmt, dbms_sql.native);
    dummy := dbms_sql.execute(dynamicCursor);
    dbms_sql.close_cursor(dynamicCursor);
    open chains;
    fetch chains into v_chains;
    if (v_chains != 0) then
    if (v_count = 0) then
    dbms_output.put_line(CHR(9)||CHR(9)||CHR(9)||'<<<<< Chained Rows Found >>>>>');
    v_count := 1;
    end if;
    dynamicCursor := dbms_sql.open_cursor;
    sql_stmt := 'Select count(*) v_rows'||' From '||v_owner||'.'||v_table;
    dbms_sql.parse(dynamicCursor, sql_stmt, dbms_sql.native);
    dbms_sql.DEFINE_COLUMN(dynamicCursor, 1, v_rows);
    dummy := dbms_sql.execute(dynamicCursor);
    dummy := dbms_sql.fetch_rows(dynamicCursor);
    dbms_sql.COLUMN_VALUE(dynamicCursor, 1, v_rows);
    dbms_sql.close_cursor(dynamicCursor);
    dbms_output.put_line(v_owner||'.'||v_table);
    dbms_output.put_line(CHR(9)||'---> Has '||v_chains||' Chained Rows and '||v_rows||' Num_Rows
in it!');
    dynamicCursor := dbms_sql.open_cursor;
    sql_stmt := 'truncate table chained_rows';
    dbms_sql.parse(dynamicCursor, sql_stmt, dbms_sql.native);
    dummy := dbms_sql.execute(dynamicCursor);
    dbms_sql.close_cursor(dynamicCursor);
    v_chains := 0;
    end if;
    close chains;
    fetch analyze into v_owner, v_table;
```

```
end loop;
if (v_count = 0) then
dbms_output.put_line('No Chained Rows found in the '||v_owner||' owned Tables!');
end if;
close analyze;
end;
/
set feed on head on
prompt
```

8.3 行迁移/行链接的清除方法

由于对于行链接来说，只能通过增大 db_block_size 来清除，而 db_block_size 在创建了数据库后又是不能改变的，所以这里对行链接的清除就不做过多的叙述了，主要是针对行迁移来谈谈在实际的生产系统中如何对其进行清除。

对于行迁移的清除，一般来说分为两个步骤：第一步，控制行迁移的增长，使其不再增多；第二步，清除以前存在的行迁移。

众所周知，行迁移产生的主要原因是因为表上的 PCTFREE 参数设置过小所致，要实现控制行迁移的增长，就必须设置一个合适的 PCTFREE 参数，否则即使清除了当前的行迁移后马上又会产生很多新的行迁移。当然，这个参数也不是越大越好，如果 PCTFREE 设置得过大，就会导致数据块的利用率低，造成空间的大量浪费，因此必须设置一个合理的 PCTFREE 参数。如何确定一个表上合理的 PCTFREE 参数呢，一般来说有两种方法。

第一种是定量的设定方法，就是利用公式来设定 PCTFREE 的大小。先使用 ANALYZE TABLE table_name ESTIMATE STATISTICS 命令来分析要修改 PCTFREE 的表，然后查看 user_tables 中的 AVG_ROW_LEN 列值，得到一个平均行长 AVG_ROW_LEN1，然后大量地对表操作之后，再次使用上述命令分析表，得到第二个平均行长 AVG_ROW_LEN2，然后运用公式 100 *（AVG_ROW_LEN2-AVG_ROW_LEN1）/（AVG_ROW_LEN2-AVG_ROW_LEN1 + 原始的 AVG_ROW_LEN）得出的结果就是定量计算出来的一个合适的 PCTFREE 值。这种方法因为是定量计算出来的，不一定会很准确，而且因为要分析表，所以对于使用 RBO 执行计划的系统不是很适用。例如，avg_row_len_1=60，avg_row_len_2=70，则平均修改量为 10，PCTFREE 应调整为 100*10/（10 + 60）=16.7% 。

第二种是差分微调的方法，先查询到当前表的 PCTFREE 值，然后监控和调整 PCTFREE 参数，每次增加一点 PCTFREE 的大小，每次增加的比例不要超过 5 个百分点，然后使用 ANALYZE TABLE TABLE_NAME LIST CHAINED ROWS INTO chained_rows 命令分析每次所有的行迁移和行链接的增长情况，对于不同的表采取不同的增长比例，对于行迁移增长得比较快的表 PCTFREE 值就增加得多点，对于增长慢的表就增加得少点，直到表的行迁移基本保持不增长了为止。但是注意不要把 PCTFREE 调得过大，一般在 40%以下就可以了，否则会造成空间的很大浪费和增加数据库访问的 I/O。

当使用上述方法控制住了当前表的行迁移的增长后，就可以开始清除之前表上存在的行迁移了。是否清除掉行迁移，关系到系统的性能是否能够有很大的提高。因此，对于以前存在的行迁移是一定而且必须要清除的。清除已经存在的行迁移有很多方法，但是并不是所有的方法都能适

用所有的情况，例如，表中的记录数多少、表上的关联多少、表上行迁移的数量多少等这些因素都会成为选择清除方法的制约条件，因此，根据表的特点和具体情况的不同，应该采用不同的方法来清除行迁移。下面将逐一介绍各种清除行迁移的方法以及它们各自适用的情况。

1. 传统清除行迁移的方法

具体步骤如下：

(1) 执行$ORACLE_HOME/rdbms/admin 目录下的 utlchain.sql 脚本创建 chained_rows 表。

```
@$ORACLE_HOME/rdbms/admin/utlchain.sql
```

(2) 将存在行迁移的表（此处用 table_name 代替）中的产生行迁移的行的 rowid 放入到 chained_rows 表中。

```
ANALYZE TABLE table_name LIST CHAINED ROWS INTO chained_rows;
```

(3) 将表中的行迁移的 row id 放入临时表中保存。

```
CREATE  TABLE  table_name_temp  AS
SELECT  *  FROM  table_name
WHERE  rowid  IN
 (SELECT  head_rowid  FROM  chained_rows
WHERE  table_name  =  'table_name');
```

(4) 删除原来表中存在的行迁移的记录行。

```
DELETE  table_name
WHERE  rowid  IN
 (SELECT  head_rowid
FROM  chained_rows
WHERE  table_name  =  'table_name');
```

(5) 从临时表中取出且重新把那些被删除了的数据插入到原来的表中，并删除临时表。

```
INSERT  INTO  table_name  SELECT  *  FROM  table_name_temp;
DROP  TABLE  table_name_temp;
```

这种传统的清除 RM 的方法，其优点是执行起来过程比较简单，容易实现。但是这种算法的缺陷是没有考虑到表关联的情况，在大多数数据库中很多表都是和别的表之间有表关联的，有外键的限制，这样就造成在步骤 (3) 中根本无法 delete 掉存在有行迁移的记录行，所以这种方法的适用范围是有限的，它只能适用于表上无任何外键关联的表。由于这种方法在插入和删除数据的时候都没有 disable 掉索引，这样导致时间主要消耗在删除和插入时维持索引树的均衡上了，对于记录数不多的情况，这个时间还比较短，但是如果对于记录数很多的表，这个所消耗的时间就不是能接受的。显然，这种方法在处理大数据量的表的时候是不可取的。

以下是一个具体在生产数据库上清除行迁移的例子，在这之前已经调整过表的 PCTFREE 参数至一个合适的值了：

```
SQL>@$ORACLE_HOME/rdbms/admin/utlchain.sql
Table created.
SQL> ANALYZE TABLE CUSTOMER LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL>SELECT count(*) from chained_rows;
TABLE_NAME                    COUNT(*)
------------------------------ ----------
CUSTOMER                        21306
1 rows selected.
```

查看在 CUSTOMER 表上存在的限制：

```
    SQL>select   CONSTRAINT_NAME,CONSTRAINT_TYPE,TABLE_NAME   from   USER_CONSTRAINTS   where
TABLE_NAME='CUSTOMER';
    CONSTRAINT_NAME             C TABLE_NAME
    ------------------------------ - ------------------------------
    PK_CUSTOMER1                P CUSTOMER
    SQL>select   CONSTRAINT_NAME,CONSTRAINT_TYPE,TABLE_NAME   from   USER_CONSTRAINTS   where
R_CONSTRAINT_NAME='PK_CUSTOMER1';
    no rows selected
    SQL> CREATE  TABLE  CUSTOMER_temp  AS
    SELECT  *  FROM  CUSTOMER   WHERE  rowid  IN
     (SELECT  head_rowid  FROM  chained_rows
    WHERE  table_name  =  'CUSTOMER');
    Table created.
    SQL>select count(*) from CUSTOMER;
      COUNT(*)
    ----------
       338299
    SQL> DELETE  CUSTOMER WHERE rowid  IN
     (SELECT  head_rowid
    FROM  chained_rows
    WHERE  table_name  =  'CUSTOMER');
    21306 rows deleted.
    SQL> INSERT  INTO  CUSTOMER SELECT  *  FROM  CUSTOMER_temp;
    21306 rows created.
    SQL> DROP  TABLE  CUSTOMER_temp;
    Table dropped.
    SQL> commit;
    Commit complete.
    SQL> select count(*) from CUSTOMER;
      COUNT(*)
    ----------
    338299
    SQL> truncate table chained_rows;
    Table truncated.
    SQL> ANALYZE TABLE CUSTOMER LIST CHAINED ROWS INTO chained_rows;
    Table analyzed.
    SQL> select count(*) from chained_rows;
      COUNT(*)
    ----------
           0
```

以上清除 20000 多行行迁移的整个过程在 2 分钟左右完成，而且全部都在联机的状态下完成，基本上不会对业务有什么影响，惟一就是在要清除行迁移的表上不能有对外键的限制，否则就不能采用这个方法进行清除了。

2. 改进的传统清除行迁移的方法

具体步骤如下：

(1) 执行$ORACLE_HOME/rdbms/admin 目录下的 utlchain.sql 脚本创建 chained_rows 表。

(2) 禁用所有其他表上关联到此表上的所有限制。

(3) 将表中的行迁移的 row id 放入临时表中保存。

（4）删除原来表中存在的行迁移的记录行。

（5）从临时表中取出且重新把那些被删除了的数据插入到原来的表中，并删除临时表。

（6）启用所有其他表上关联到此表上的所有限制。

这种算法是对传统算法的一种改进，这种算法考虑到了表之间的关联，还可以灵活地利用 TOAD 工具生成的表关联信息，是一种比较适合于清除行迁移的一种方法。但是因为使用这种方法后来需要重建索引，对于记录数很大的表，比如具有上千万条以上记录的表，就不是很合适了，因为这个重建索引的时间会很长，是线性时间复杂度的，而重建索引会使得索引所在的表被锁定，从而导致插入不了新的记录，重建索引的时间太长导致记录长时间插入不了是会严重影响应用的，甚至导致数据的丢失，因此这是使用这个方法之前必须要考虑到的一个重要因素。对于 8i 以上的版本可以使用 online 的方法来重建索引，这样不会导致锁表，但是会有额外的开销，时间会很长。再者，因为这种方法在插入记录和删除记录都是带着索引的，如果表上的行迁移比较多，这样耗费时间就会比较长，而且占用资源也会比较大，因此只适用于表上行迁移存在得比较少的表。总的来说，这种方法对于表记录太多或者是表上的行迁移太多的情况都不是很适用，比较适合表记录少和表上行迁移都不太多的情况。

以下是一个具体在生产数据库上清除行迁移的例子，在这之前已经调整过表的 PCTFREE 参数至一个合适的值了：

```
    SQL>select index_name,index_type,table_name from user_indexes where table_name='TERMINAL';
    INDEX_NAME                     INDEX_TYPE        TABLE_NAME
    ----------------------------------------------------------------
    INDEX_TERMINAL_TERMINALCODE    NORMAL            TERMINAL
    I_TERMINAL_ID_TYPE             NORMAL            TERMINAL
    I_TERMINAL_OT_OID              NORMAL            TERMINAL
    PK_TERMINAL_ID                 NORMAL            TERMINAL
    UI_TERMINAL_GOODIS_SSN         NORMAL            TERMINAL
    SQL>select    CONSTRAINT_NAME,CONSTRAINT_TYPE,TABLE_NAME  from  USER_CONSTRAINTS  where
R_CONSTRAINT_NAME='PK_TERMINAL_ID';
    CONSTRAINT_NAME             C TABLE_NAME
    ------------------------------ - ------------------------------
    SYS_C003200                R CONN
    SQL>alter table CONN disable constraint SYS_C003200;
    Table altered.
    SQL>CREATE  TABLE  TERMINAL_temp  AS
    SELECT  *  FROM  TERMINAL
    WHERE  rowid  IN
     (SELECT  head_rowid  FROM  chained_rows
    WHERE  table_name  =  'TERMINAL');
    Table created.
    SQL>select count(*) from TERMINAL_temp;
      COUNT(*)
    ----------
         8302
    SQL>DELETE  TERMINAL
    WHERE  rowid  IN
     (SELECT  head_rowid
    FROM  chained_rows
    WHERE  table_name  =  'TERMINAL');
     8302 rows deleted.
```

```
SQL>INSERT INTO TERMINAL SELECT * FROM TERMINAL_temp;
8302 rows created.
SQL>alter table CONN disable constraint SYS_C003200;
Table altered.
SQL>select count(*) from terminal;
  COUNT(*)
----------
647799
SQL>truncate table chained_rows;
Table truncated.
SQL>ANALYZE TABLE TERMINAL LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL>select count(*) from chained_rows;
  COUNT(*)
----------
       0
```

从上面过程中可以看出，对 TERMINAL 这张表的行迁移清除耗时总共不到 5 分钟，总体来说还是比较快的。从本人在生产数据库中清除行迁移的经验来说，这种方法基本适用于大部分存在有行迁移的表。

3. 使用 TOAD 工具清除行迁移的方法

具体步骤如下：

⑴ 备份要清除 RM 的表。

```
RENAME table_name TO table_name_temp;
```

⑵ drop 所有其他表上关联到 table_name 的外键限制。

```
SELECT CONSTRAINT_NAME,CONSTRAINT_TYPE,TABLE_NAME from USER_CONSTRAINTS where R_CONSTRAINT_NAME in (SELECT CONSTRAINT_NAME from USER_CONSTRAINTS where TABLE_NAME='table_name' AND CONSTRAINT_TYPE='P');
ALTER TABLE table_name DROP CONSTRAINT XXXX;（XXXX 为上述的查询结果）
```

⑶ 重建 ⑴ 中被 rename 的表。

```
CREATE TABLE table_name AS SELECT * FROM table_name_temp WHERE 0 = 1;
```

⑷ 重建表中原来的数据。

```
INSERT /*+ APPEND */ INTO table_name SELECT * FROM table_name_temp;
```

⑸ 删除在 table_name_temp 上的索引和关联其他表的外键。

⑹ 在 table_name 上建立和原来一样的索引、主键和所有的外键限制。

⑺ 重新编译相关的存储过程、函数和包。

⑻ 删除表 table_name_temp。

使用这种方法来清除行迁移，全部的代码都是可以由 TOAD 工具来生成的。由于此方法把表上的关联考虑进去了，也是一种考虑比较全面的清除方法，而且在清除过程中重建了表和索引，对于数据库的存储和性能上都有所提高。因为这种方法一开始是 rename 表为临时表，然后重建一个新表出来的，因此需要 2 倍的表空间，因此在操作之前一定要检查要清除的表所在的表空间的 free 空间是否足够。但是这种方法也有一定的缺陷，因为在新表中重新插入原来的数据后需要重建索引和限制，因此在时间和磁盘的空间上都有比较大的开销，而且对于前台的应用可能会有一段时间的中断，当然，这个中断时间主要是消耗在重建索引和重建限制上了，而时间的长短跟需要重建索引和限制的多少及表的记录多少等因素都有关系。对于 7*24 小时要求的系统使用这

种方法清除行迁移不是很合适，因为使用这种方法可能会导致系统有一段时间的停机，如果系统的实时性比较高，这种方法就不是很适用了。

4. 使用 EXP/IMP 工具清除行迁移的方法

具体步骤如下：

(1) 使用 EXP 导出存在有行迁移的表。

(2) 然后 TRUNCATE 原来的表。

(3) IMP 开始导出的表。

(4) (可选) 重建表上所有的索引。

使用这种方法可以不用重建索引，省去了这部分时间，但是完成之后索引的使用效率不会很高，最好是在以后逐步地在线重建索引，这样是可以不需要中断业务的。但是需要考虑的是 IMP 的时候会比较慢，而且会占用比较大的 I/O，应该选择在应用不是很繁忙的时候做这项工作，否则会对应用的正常运行产生较大的影响。对于这种方法还存在一个比较大的弊端，就是在 EXP 表的时候要保证该表是没有数据的更新或者是只读状态，不能对表有插入或者更新操作，否则会导致数据的丢失。

```
SQL> select count(*) from test;
  COUNT(*)
----------
169344
SQL> truncate table chained_rows;
Table truncated.
SQL> analyze table test LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL> select count(*) from chained_rows;
  COUNT(*)
----------
      3294
$ exp allan/allan file=test.dmp tables=test
Export: Release 9.2.0.3.0 - Production on Sun Jun 6 13:50:08 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
Export done in ZHS16GBK character set and AL16UTF16 NCHAR character set
About to export specified tables via Conventional Path ...
. . exporting table                         TEST    169344 rows exported
Export terminated successfully without warnings.
$ sqlplus allan/allan
SQL*Plus: Release 9.2.0.3.0 - Production on Sun Jun 6 13:50:43 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> truncate table test;
Table truncated.
SQL> exit
```

```
Disconnected from 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
$ imp allan/allan file=test.dmp full=y ignore=y buffer=5000000
Import: Release 9.2.0.3.0 - Production on Sun Jun 6 13:51:24 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
Export file created by EXPORT:V09.02.00 via conventional path
import done in ZHS16GBK character set and AL16UTF16 NCHAR character set
. importing ALLAN's objects into ALLAN
. . importing table                      "TEST"    169344 rows imported
Import terminated successfully without warnings.
$ sqlplus allan/allan
SQL*Plus: Release 9.2.0.3.0 - Production on Sun Jun 6 13:52:53 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 count(*) from test;
  COUNT(*)
----------
   169344
SQL> select index_name from user_indexes where table_name='TEST';
INDEX_NAME
------------------------------
OBJ_INDEX
SQL> alter index OBJ_INDEX rebuild online;
Index altered.
SQL> truncate table chained_rows;
Table truncated.
SQL> analyze table test LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL> select count(*) from chained_rows;
  COUNT(*)
----------
       0
```

5. 使用 MOVE 命令来清除行迁移的方法

具体步骤如下：

(1) 查看要清除行迁移的表所在的表空间。

```
select table_name,tablespace_name from user_tables where table_name='table_name';
```

(2) 查看要清除行迁移的表上的具体索引。

```
select index_name,table_name from user_indexes where table_name= ‘table_name’;
```

(3) move 要清除 RM 的表到指定的表空间中去。

```
alter table table_name move tablespace tablespace_name;
```

(4) 重建表上的所有索引。

```
alter index index_name rebuild;
```

这种方法适用于 8i 及其以上的数据库版本，主要是利用数据库的一个 move 命令来实现行迁移的清除，move 命令的实质其实就是 insert … select 的一个过程，在 move 表的过程中需要原来表的大小的 2 倍空间，因为中间过程要保留原来的旧表，新表创建完成后旧表就被删除并释放空间了。move 的时候要注意后面一定要加上表空间参数，所以必须要先知道表所在的表空间；因为 move 表之后需要重建索引，所以之前要确定表上的所有索引。

这种方法对于表记录数很大或者表上索引太多的情况不太适用，因为本身的 move 就会很慢，而且 move 表的时候会要锁定表，时间长了会导致对表的其他操作出现问题，导致数据插入不了，丢失数据；move 表后还要重建索引，索引太多了的话重建的时间也会很长；再者，这个方法也比较消耗资源，因此强烈建议在业务不繁忙的时候执行。

以下是一个具体在生产数据库上清除行迁移的例子，在这之前已经调整过表的 PCTFREE 参数至一个合适的值了：

```
SQL>ANALYZE TABLE SERVICE LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL>SELECT count(*) from chained_rows;
COUNT(*)
----------
   9145
SQL>select table_name,tablespace_name from user_tables where table_name='SERVICE';
TABLE_NAME                     TABLESPACE_NAME
------------------------------ ------------------------------
SERVICE                        DATA
SQL>select index_name,table_name from user_indexes where table_name='SERVICE';
INDEX_NAME                     TABLE_NAME
------------------------------ ------------------------------
I_SERVICE_ACCOUNTNUM           SERVICE
I_SERVICE_DATEACTIVATED        SERVICE
I_SERVICE_SC_S                 SERVICE
I_SERVICE_SERVICECODE          SERVICE
PK_SERVICE_SID                 SERVICE
SQL>select count(*) from SERVICE;
  COUNT(*)
----------
   518718
SQL>alter table SERVICE move tablespace DATA;
Table altered.
SQL>alter index I_SERVICE_ACCOUNTNUM rebuild;
Index altered.
SQL>alter index I_SERVICE_DATEACTIVATED rebuild;
Index altered.
SQL>alter index I_SERVICE_SC_S rebuild;
Index altered.
SQL>alter index I_SERVICE_SERVICECODE rebuild;
Index altered.
SQL>alter index PK_SERVICE_SID rebuild;
Index altered.
SQL>truncate table chained_rows;
Table truncated.
```

```
SQL>ANALYZE TABLE SERVICE LIST CHAINED ROWS INTO chained_rows;
Table analyzed.
SQL>SELECT count(*) from chained_rows;
COUNT(*)
----------
       0
```

利用 move 命令来清除行迁移，执行的命令都相对比较简单。在上面的例子中，清除表 SERVCIE 中的行迁移所耗费的时间大概在 5 分钟左右，其中 move 命令执行的时间不到 2 分钟，也就是说锁表的时间大概不到 2 分钟，对于大多数的应用来说这个问题都是不大的，如果放在系统闲的时候执行，基本上不会对应用产生什么的影响。

6. 对于一些行迁移数量巨大而且表记录数巨大的表的行迁移的清除方法

具体步骤如下：

（1）使用 TOAD 工具或者别的方法获取存在大量行迁移并且表记录很大的表的重建表的 SQL，然后保存为脚本。

（2）使用 rename 命令将原始表重命名为一个备份表，然后删除别的表对原始表上的限制以及原始表上的外键和索引。

（3）利用（1）步骤中生成的脚本重建原始表，以及表上的限制、外键、索引等对象。

（4）然后按表模式导出（2）步骤中备份的表，然后导入到另外的一个临时中转的数据库中，因为表的名字已经改变，所以导入后需要 rename 表为原来的名字，然后重新导出，最后再导入到原来的数据库中。

这种方法主要是用来针对一些数据量比较大，并且表上的行迁移也比较多的表的行迁移清除。对于这些大表的行迁移的清除，正常来说都需要把应用停掉一段较长时间才能够完成，这让人感觉比较头疼，对于 7*24 小时的应用来说，停机的时间越长损失就越大，当然是要尽量地减短停机的时间。

因为表本身比较大，不管怎样做什么操作都是会比较耗费时间和资源的，但是如果应用在某段时间内主要是以插入数据为主，更新数据和删除数据都很少的，就可以考虑采用这么一种方法：先重命名表，然后重新建立一个和原来一样的表，用来保证之后的应用的数据是可以正常插入的，从而使应用不用停掉很久，因为重建一个没有任何数据的表结构的过程是很短暂的，大概只需要几秒钟的时间，而重建好表之后就能保证应用能够正常地写入数据，从而使应用几乎不用停顿，然后把开始重命名的原始表按表模式导出，因为表的名字已经被改变，因此需要一个临时库来导入这些数据，然后重命名为原来的名字，然后按原来的表名导出后再重新导入原始数据库，这样操作起来虽然会比较麻烦，但是却是一种很有效、很实际的方法，速度也很快，导出后导入，因为本身表结构已经建立好了，不需要其他任何的多余操作，而且最关键的是这种方法所需要的停机时间是最短的。

```
SQL>ALTER TABLE USER.PAY RENAME TO PAY_X ;
然后导出 PAY_X 表;
$ exp USER/USER file=PAY_X.dmp tables=PAY_X
SQL>ALTER TABLE USER.BATCHPAYMENTDETAIL DROP CONSTRAINT FK_BATCHPAYMENTDETAIL_OPAYID ;
SQL>ALTER TABLE USER.DEPOSITCLASSIFY DROP CONSTRAINT FK_DEPOSITCLASSIFY2 ;
SQL>ALTER TABLE USER.DEPOSITCREDITLOG DROP CONSTRAINT FK_DEPOSITCREDITLOG2 ;
SQL>ALTER TABLE USER.DEPOSIT DROP CONSTRAINT SYS_C003423 ;
SQL>ALTER TABLE USER.PAY_X DROP CONSTRAINT SYS_C003549 ;
```

```
SQL>DROP INDEX USER.I_PAY_STAFFID ;
SQL>CREATE TABLE USER.PAY
 (
  PAYID          NUMBER(9),
  ACCOUNTNUM         NUMBER(9),
  TOTAL             NUMBER(12,2),
  PREVPAY        NUMBER(12,2),
  PAY           NUMBER(12,2),
  STAFFID           NUMBER(9),
  PROCESSDATE        DATE,
  PAYNO          CHAR(12),
  TYPE              CHAR(2)                 DEFAULT '0',
  PAYMENTMETHOD      CHAR(1)                DEFAULT '0',
  PAYMENTMETHODID    VARCHAR2(20),
  BANKACCOUNT        VARCHAR2(32),
  PAYMENTID          NUMBER(9),
  STATUS            CHAR(1)                DEFAULT '0',
  MEMO              VARCHAR2(255),
  SERVICEID          NUMBER(9),
  CURRENTDEPOSITID    NUMBER(9),
  SHOULDPROCESSDATE   DATE                   DEFAULT sysdate,
  ORIGINALEXPIREDATE  DATE,
  ORIGINALCANCELDATE  DATE,
  EXPIREDATE         DATE,
  CANCELDATE         DATE,
  DEPOSITTYPE        CHAR(1)
)
TABLESPACE USER
PCTUSED    95
PCTFREE    5
INITRANS   1
MAXTRANS   255
STORAGE    (
           INITIAL        7312K
           NEXT           80K
           MINEXTENTS      1
           MAXEXTENTS      2147483645
           PCTINCREASE     0
           FREELISTS       1
           FREELIST GROUPS  1
           BUFFER_POOL     DEFAULT
          )
NOLOGGING
NOCACHE
NOPARALLEL;
SQL>CREATE INDEX USER.I_PAY_STAFFID ON USER.PAY
 (STAFFID)
NOLOGGING
TABLESPACE USER
PCTFREE    5
INITRANS   2
MAXTRANS   255
```

```
STORAGE   (
          INITIAL        1936K
          NEXT           80K
          MINEXTENTS      1
          MAXEXTENTS      2147483645
          PCTINCREASE     0
          FREELISTS       1
          FREELIST GROUPS  1
          BUFFER_POOL     DEFAULT
         )
NOPARALLEL;
SQL>CREATE UNIQUE INDEX USER.PK_PAY_ID ON USER.PAY
 (PAYID)
NOLOGGING
TABLESPACE USER
PCTFREE   5
INITRANS  2
MAXTRANS  255
STORAGE   (
          INITIAL        1120K
          NEXT           80K
          MINEXTENTS      1
          MAXEXTENTS      2147483645
          PCTINCREASE     0
          FREELISTS       1
          FREELIST GROUPS  1
          BUFFER_POOL     DEFAULT
         )
NOPARALLEL;
SQL>ALTER TABLE USER.PAY ADD (
  FOREIGN KEY (STAFFID)
REFERENCES USER.STAFF (STAFFID));
SQL>ALTER TABLE USER.DEPOSITCLASSIFY ADD
 CONSTRAINT FK_DEPOSITCLASSIFY2
 FOREIGN KEY (PAYID)
  REFERENCES USER.PAY (PAYID)  ;
SQL>ALTER TABLE USER.DEPOSITCREDITLOG ADD
 CONSTRAINT FK_DEPOSITCREDITLOG2
 FOREIGN KEY (PAYID)
  REFERENCES USER.PAY (PAYID)  ;
SQL>ALTER FUNCTION "USER"."GENERATEPAYNO" COMPILE ;
SQL>ALTER PROCEDURE "USER"."ENGENDERPRVPAY" COMPILE ;
SQL>ALTER PROCEDURE "USER"."ISAP_ENGENDERPRVPAY" COMPILE ;
SQL>ALTER PROCEDURE "USER"."SPADDCREDITDEPOSIT" COMPILE ;
SQL>ALTER PROCEDURE "USER"."SPADDDEPOSITWITHOUTCARD" COMPILE ;
SQL>ALTER PROCEDURE "USER"."SPADJUSTLWDEPOSIT" COMPILE ;
……
```

然后将导出的表 PAY_X 的dmp 文件导入一个临时的数据库中，然后在临时数据库中将其表名重新命名为 PAY，再按表模式将其导出。

```
imp USER/USER file= PAY_x.dmp tables=PAY ignore=y
SQL>rename PAY_X to PAY;
```

```
$ exp USER/USER file=PAY.dmp tables=PAY
```

最后将这个.dmp 文件导入正式的生产数据库中即可。

以上过程在重建好 PAY 表后，整个应用就恢复正常了，而重命名表后重建表的时间是非常之短的，本人测试的大概是在几分钟之内就可以做完，新数据也就可以插入表了，剩下的工作就是将旧的数据导入数据库，这个工作在时间要求上就没有那么高了，因为应用已经正常了，一般来说利用晚上业务不忙的时候都可以把一张表的数据导入完成的。

以上 6 种清除行迁移的方法各有各的优缺点，分别适用于不同的情况。利用以上几种清除行迁移的方法基本就能完全清除掉系统中存在的行迁移，当然，具体的生产环境中还需要具体问题具体分析，针对不同类型的系统，系统中不同特点的表应采用不同的清除方法，尽量地减少暂停数据库的时间，以保证应用不间断地稳定运行。

作者简介

叶梁，网名 cooly1，现任 ITPUB Oracle 管理版版主。

曾任职于国内某大型软件企业做 Oracle 数据库的技术支持，客户遍及全国各个行业，尤其是电信、政府、金融行业。现任职于某外资电信企业华北区分公司，从事 DBA 工作，负责华北区 40 多个数据库系统的维护，对大型数据库管理经验丰富。

擅长数据库的维护，对于数据库的安装，调整，备份方面有自己独到的经验。同时也给一些国内的大型企业做过 Oracle 培训，有一定的培训经验。

曾做过很多大型项目的数据库维护和支持工作，对 Oracle 的维护有相当多的实际经验，善于现场解决问题。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

第 9 章　HWM 与数据库性能的探讨

本章讨论的是 Oracle 中关于 table 的 HWM 的问题，主要包括什么是 HWM、HWM 是如何移动的、HWM 对于性能的影响、何时应该降低以及如何降低 HWM、其他一些影响 HWM 的操作等内容。

除了特殊注明，本章内容所有的实验都基于：Windows 2000、Oracle 9201、bolck_ size 8KB 的实验环境。

9.1　什么是 HWM

高水标记 HWM（High Water Mark），这个概念在 segment 的存储内容中是比较重要的。简单来说，HWM 就是一个 segment 中已使用和未使用的 block 的分界线。

在 Oracle 的 concept 中对于 HWM 的说明是这样的：在一个 segment 中，HWM 是使用和未使用空间的分界线。当请求新的空闲块，并且现有空闲列表中的块不能满足要求时，HWM 指向的块将被标记为已使用，然后 HWM 将移动指向下一个未使用过的块。

在 Oracle 中，存储数据的最小单元是 block，对于一个 segment（table 或 index），都是由很多个 block 组成的，这些 block 的状态分为已使用和未使用两种，一般来说，在 HWM 之下的 block 都是存储过数据的，如图 9-1 所示。

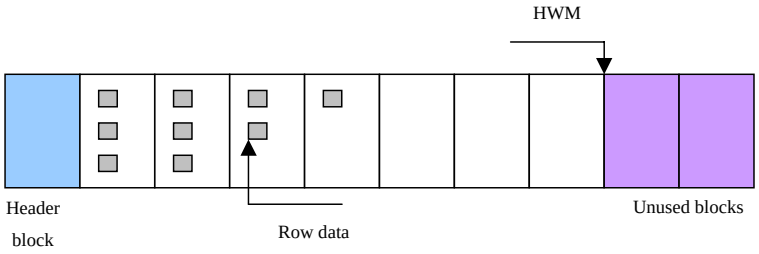

图 9-1　block 的分布

从图 9-1 中，可以很清楚地看到，一个 segment 中的 block 的分布情况。在 HWM 左边的 block 是已使用的，或者说是可以用来存储数据的。而 HWM 右边的 block 是不能用来存储数据的。当

HWM 左边的 block 空间都使用完之后，还有新的数据需要存储，那么会怎样处理呢？这时 Oracle 会向右移动 HWM，即把右边没有使用的 block 移到 HWM 的左边，这时 HWM 左边的 block 就增加了，那么就有新的 block 空间可供使用了。

Oracle 从 9i 开始，推出了一种新的 segment 的空间管理方式，即 ASSM（Auto Segment Space Management）。这种 segment 的空间管理方式和以前的 FLM（Freelist Management）是不一样的，这里简单地介绍一下。

在 FLM 模式下，对于一个 segment 的 HWM 下的所有 block 空间的使用，是通过 Freelist 来管理的，Freelist 位于 segment 的第一个 extent 中。一个 block 何时应该位于 Freelist 之上，取决于 PCTUSED 和 PCTFREE 这样两个参数。基于 Freelist 管理模式和位于 segment header 的情况，如果对一个 segment 进行高并发的频繁的 DML 操作，就不可避免地会出现 header 争用的情况，虽然可以采用增加 Freelists 或 Freelist Group 的方式来缓解这种状况。

那么从 Oracle 92 开始，推出了 ASSM 这样一种全新的 segment 空间管理的方式（又称 Bitmap Managed Segments），Freelist 被位图所取代，使用位图来管理 block 的空间使用状况，并且这些位图块分散在 segment 中。ASSM 管理的 segment 会略掉任何为 PCTUSED、NEXT 和 Freelists 所指定的值。

使用 ASSM 也有一定的局限性：

- ASSM 只能位于 Local Manage 的 tablespace 之上。
- 不能够使用 ASSM 创建临时的 tablespace。
- LOB 对象不能在一个指定进行自动段空间管理的 tablespace 中创建。

以上简单地介绍了 ASSM 和 FLM 的概念和区别，接下来，看看这两种 segment 空间管理模式在 HWM 的处理上有什么不同。

9.2 初始创建的 table 中 HWM 的不同情况

来看 FLM 管理的 table，先创建名为 HWM 的 tablespace，指定非自动段空间管理，extent 大小为 40KB。并在上面创建 table TEST_HWM，PCTFREE 为 40，PCTUSED 为 20。

```
SQL> connect dlinger/dlinger@Oracle9i_dl
连接到:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

SQL> CREATE TABLESPACE HWM
  2  DATAFILE 'D:\ORACLE\ORADATA\ORACLE9I\HWM.dbf'
  3  SIZE 50M uniform size 40K;

表空间已创建。

SQL> select TABLESPACE_NAME,BLOCK_SIZE,EXTENT_MANAGEMENT,
  2  ALLOCATION_TYPE, SEGMENT_SPACE_MANAGEMENT
  3  from dba_tablespaces where TABLESPACE_NAME = 'HWM';
```

```
TABLESPACE_NAME BLOCK_SIZE EXTENT_MANAGEMENT ALLOCATION_TYPE SEGMENT_SPACE_MANAGEMENT
--------------- ---------- ----------------- --------------- ------------------------
HWM                8192 LOCAL           UNIFORM        MANUAL


SQL> alter user dlinger default tablespace hwm;

用户已更改。

SQL> CREATE TABLE TEST_HWM  (ID CHAR(2000) , NAME CHAR(2000) )
  2  STORAGE ( MINEXTENTS 2)  PCTFREE 40 PCTUSED 20;

表已创建。

SQL>select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
     from dba_extents
  2  where segment_name='TEST_HWM' ;

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0         11           11          9          5
         1         11           11         14          5

SQL> alter system dump datafile 11 block 9;

系统已更改。
```

Table TEST_HWM 位于 datafile 11，segment header 为 block 9，dump 出 block 9 来看看：

```
*** 2004-06-09 20:31:26.000
*** SESSION ID:(9.5) 2004-06-09 20:31:26.000
Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
buffer tsn: 14 rdba: 0x02c00009 (11/9)
scn: 0x0000.013e974e seq: 0x01 flg: 0x00 tail: 0x974e1001
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
  Extent Control Header
  -----------------------------------------------------------------
  Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
                last map  0x00000000  #maps: 0     offset: 4128
     Highwater::  0x02c0000a ext#: 0     blk#: 0     ext size: 4
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
                Unlocked
    Map Header:: next  0x00000000  #extents: 2    obj#: 32377  flag: 0x40000000
  Extent Map
  -----------------------------------------------------------------
   0x02c0000a  length: 4
   0x02c0000e  length: 5

  nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 0
  SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
```

从 dump 的 trace 文件中，可以看到 Highwater:: 0x02c0000a，在 FLM 的 segment 下，初始创建的 table，HWM 是从第一个 extent 的第二个 block 开始的。为 segment header 保留一个块。从这里也可以看出来：

```
Extent Map
 -----------------------------------------------------------------
 0x02c0000a  length: 4
```

说明第一个 extent 可用的 block 为 4。

这里看到的结果是在默认 Freelist 为 1 的条件下得到的。在 FLM 下，如果对 segment 设置了 Freelist Groups N，则 HWM 指向第 N＋2 个 block，当 N+2 ＞ initextent 的 block 数时，会返回 ORA-03237 的错误信息，这里 tablespace HWM 的 extent 为 40KB，block_size 为 8KB：

```
SQL>  CREATE TABLE TEST_HWM2  (ID CHAR(2000) , NAME CHAR(2000) )
  2   STORAGE ( MINEXTENTS 2 freelist groups 4)  PCTFREE 40 PCTUSED 20;
 CREATE TABLE TEST_HWM2  (ID NUMBER(10) , NAME CHAR(2000) )
*
ERROR 位于第 1 行:
ORA-03237: 在表空间 (HWM) 无法分配指定大小的初始区
```

在 ASSM 下，情况是怎样的呢？创建名为 ASSM 的 tablespace，指定自动段空间管理，extent 大小为 40KB。并在上面创建 table TEST_HWM1，注意，这里只指定了 PCTFREE 为 40，因为 PCTUSED 在 ASSM 下的 segment 中是无效的。

```
SQL> CREATE TABLESPACE ASSM
  2  DATAFILE 'D:\ORACLE\ORADATA\ORACLE9I\ASSM.dbf'
  3  SIZE 50M uniform size 40K segment space management auto;

表空间已创建。


SQL> select TABLESPACE_NAME,BLOCK_SIZE,EXTENT_MANAGEMENT,
  2  ALLOCATION_TYPE, SEGMENT_SPACE_MANAGEMENT
  3  from dba_tablespaces where TABLESPACE_NAME = 'ASSM';

TABLESPACE_NAME BLOCK_SIZE EXTENT_MANAGEMENT ALLOCATION_TYPE SEGMENT_SPACE_MANAGEMENT
--------------- ---------- ----------------- --------------- ------------------------
ASSM               8192 LOCAL           UNIFORM        AUTO


SQL> CREATE TABLE TEST_HWM1  (ID CHAR(2000), NAME CHAR(2000) )
  2  Tablespace ASSM
  3  STORAGE ( MINEXTENTS 2)  PCTFREE 40;

表已创建。

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2  from dba_extents
  3  where segment_name='TEST_HWM1' ;

EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0         12           12          9          5
         1         12           12         14          5
```

```
SQL>  alter system dump datafile 12 block min 9 block max 11;

系统已更改。
```

在 FMT 下，segment 的第一个 block 是存储 segment header 的，在本例中，ASSM 下，Oracle 使用 segment 的至少前 3 个 block 来存储 segment header。这里，dump 9～11 的 block 信息。再来看一下 dump 的结果和 FMT 下有什么不同：

```
Start dump data blocks tsn: 15 file#: 12 minblk 9 maxblk 11
buffer tsn: 15 rdba: 0x03000009 (12/9)
scn: 0x0000.01ca6d7f seq: 0x02 flg: 0x00 tail: 0x6d7f2002
frmt: 0x02 chkval: 0x0000 type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
--------------------------------
   nbits : 4 nranges: 2       parent dba:  0x0300000a   poffset: 0
   unformatted: 7      total: 10       first useful block: 3
   owning instance : 1
   instance ownership changed at
   Last successful Search
   Freeness Status:  nf1 0     nf2 0     nf3 0     nf4 0

   Extent Map Block Offset: 4294967295
   First free datablock : 3
   Bitmap block lock opcode 3
   Locker xid:    :  0x0004.008.0000713c
      Highwater::  0x0300000c  ext#: 0     blk#: 3     ext size: 5
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  HWM Flag: HWM Set
  ------------------------------------------------------------
  DBA Ranges :
  ------------------------------------------------------------
   0x03000009  Length: 5     Offset: 0
   0x0300000e  Length: 5     Offset: 5

   0:Metadata   1:Metadata   2:Metadata   3:unformatted
   4:unformatted   5:unformatted   6:unformatted   7:unformatted
   8:unformatted   9:unformatted
  ------------------------------------------------------------
buffer tsn: 15 rdba: 0x0300000a (12/10)
scn: 0x0000.01ca6d7e seq: 0x02 flg: 0x00 tail: 0x6d7e2102
frmt: 0x02 chkval: 0x0000 type: 0x21=SECOND LEVEL BITMAP BLOCK
Dump of Second Level Bitmap Block
   number: 1      nfree: 1      ffree: 0     pdba:    0x0300000b
  opcode:0
 xid:
  L1 Ranges :
  ------------------------------------------------------------
   0x03000009  Free: 5 Inst: 1

  ------------------------------------------------------------
```

```
buffer tsn: 15 rdba: 0x0300000b (12/11)
scn: 0x0000.01ca6d80 seq: 0x01 flg: 0x00 tail: 0x6d802301
frmt: 0x02 chkval: 0x0000 type: 0x23=PAGETABLE SEGMENT HEADER
  Extent Control Header
  -----------------------------------------------------------------
  Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 10
              last map  0x00000000  #maps: 0     offset: 2716
     Highwater::  0x0300000c  ext#: 0     blk#: 3     ext size: 5
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
               Unlocked
  -----------------------------------------------------------
  Low HighWater Mark :
     Highwater::  0x0300000c  ext#: 0     blk#: 3     ext size: 5
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  Level 1 BMB for High HWM block: 0x03000009
  Level 1 BMB for Low HWM block: 0x03000009
  -----------------------------------------------------------
  Segment Type: 1 nl2: 1     blksz: 8192   fbsz: 0
  L2 Array start offset:  0x00001434
  First Level 3 BMB:  0x00000000
  L2 Hint for inserts:  0x0300000a
  Last Level 1 BMB:  0x03000009
  Last Level II BMB:  0x0300000a
  Last Level III BMB:  0x00000000
    Map Header:: next  0x00000000  #extents: 2    obj#: 32499  flag: 0x20000000
  Extent Map
  -----------------------------------------------------------------
   0x03000009  length: 5
   0x0300000e  length: 5

  Auxillary Map
  -----------------------------------------------------------
   Extent 0    :  L1 dba:  0x03000009 Data dba:  0x0300000c
   Extent 1    :  L1 dba:  0x03000009 Data dba:  0x0300000e
  -----------------------------------------------------------

   Second Level Bitmap block DBAs
   -----------------------------------------------------------
   DBA 1:   0x0300000a
```

这里可以看到 Highwater:: 0x0300000c，HWM 指向的第一个 extent 的第 4 个 block，也就是说，segment head 保留了 3 个 block。

为什么前面说 Oracle 在 ASSM 的 segment 中至少用前 3 个 block 来存储 segment header 的信息呢？可以创建一个 extent 为 256KB 的 tablespace，然后在上面创建 table，来看一下结果：

```
SQL> create tablespace assm
  2  datafile '/data1/Oracle/oradata/assm01.dbf'
  3  size 10M
```

```
  4  extent management local uniform size 256K
  5  segment space management auto
  6  /

Tablespace created.

SQL> CREATE TABLE TEST_HWM1    (ID CHAR(2000), NAME CHAR(2000) )
  2  Tablespace ASSM
  3  STORAGE ( MINEXTENTS 2)  PCTFREE 40
  4  /

Table created.

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2  from dba_extents
  3  where segment_name='TEST_HWM1'
  4  /

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0          7            7          9         32
         1          7            7         41         32

SQL> alter system dump datafile 7 block min 9 block max 11;

System altered.
```

看 trace 文件其中的一部分内容：

```
Start dump data blocks tsn: 16 file#: 7 minblk 9 maxblk 11
buffer tsn: 16 rdba: 0x01c00009 (7/9)
scn: 0x0000.01444ea9 seq: 0x02 flg: 0x00 tail: 0x4ea92002
frmt: 0x02 chkval: 0x0000 type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
 --------------------------------
   nbits : 4 nranges: 1        parent dba:  0x01c0000b   poffset: 0
   unformatted: 12     total: 16       first useful block: 4
   owning instance : 1
   instance ownership changed at
   Last successful Search
   Freeness Status:  nf1 0     nf2 0     nf3 0     nf4 0

   Extent Map Block Offset: 4294967295
   First free datablock : 4
   Bitmap block lock opcode 0
   Locker xid:    :  0x0000.000.00000000
      Highwater::  0x01c0000d  ext#: 0     blk#: 4     ext size: 32
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  HWM Flag: HWM Set
  ------------------------------------------------------------
  DBA Ranges :
```

```
 -------------------------------------------------------
  0x01c00009  Length: 16    Offset: 0

  0:Metadata   1:Metadata   2:Metadata   3:Metadata
  4:unformatted   5:unformatted   6:unformatted   7:unformatted
  8:unformatted   9:unformatted   10:unformatted   11:unformatted
  12:unformatted   13:unformatted   14:unformatted   15:unformatted
 -------------------------------------------------------
```

可以发现，这里使用了前 4 个 block 来存储 segment header 的内容。

9.3 insert 数据时 HWM 的移动

在 LMT 下，进行 insert 操作：

```
SQL> insert into test_hwm values('1','dlinger');

已创建 1 行。

SQL> alter system dump datafile 11 block 9;

系统已更改。

SQL> insert into test_hwm values('2','dlinger');

已创建 1 行。

SQL> alter system dump datafile 11 block 9;

系统已更改。

SQL> insert into test_hwm values('3','dlinger');

已创建 1 行。

SQL> alter system dump datafile 11 block 9;

系统已更改。

SQL> insert into test_hwm values('4','dlinger');

已创建 1 行。

SQL> alter system dump datafile 11 block 9;

系统已更改。

SQL> insert into test_hwm values('5','dlinger');
已创建 1 行。
SQL>  alter system dump datafile 11 block 9;
```

```
系统已更改。
```

查看_bump_highwater_mark_count 参数：

```
select x.ksppinm name, y.ksppstvl value,
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 '\_%' escape '\' and
x.ksppinm like '%bump_highwater_mark_count%'
order by
translate(x.ksppinm, ' _', ' ');

NAME                      VALUE
------------------------------ --------
_bump_highwater_mark_count        0
```

再看 dump 的结果：

```
*** 2004-06-14 10:46:56.000
Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
buffer tsn: 14 rdba: 0x02c00009 (11/9)
scn: 0x0000.015032ef seq: 0x01 flg: 0x00 tail: 0x32ef1001
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
 Extent Control Header
 -----------------------------------------------------------------
 Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
              last map  0x00000000  #maps: 0     offset: 4128
    Highwater::  0x02c0000b  ext#: 0     blk#: 1     ext size: 4
 #blocks in seg. hdr's freelists: 1
 #blocks below: 1
 mapblk  0x00000000  offset: 0
              Unlocked
   Map Header:: next  0x00000000  #extents: 2    obj#: 32387  flag: 0x40000000
 Extent Map
 -----------------------------------------------------------------
  0x02c0000a  length: 4
  0x02c0000e  length: 5

 nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 1
 SEG LST:: flg: USED  lhd: 0x02c0000a ltl: 0x02c0000a
End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
*** 2004-06-14 10:47:25.000
Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
buffer tsn: 14 rdba: 0x02c00009 (11/9)
scn: 0x0000.01503349 seq: 0x02 flg: 0x00 tail: 0x33491002
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
 Extent Control Header
 -----------------------------------------------------------------
 Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
              last map  0x00000000  #maps: 0     offset: 4128
    Highwater::  0x02c0000c  ext#: 0     blk#: 2     ext size: 4
 #blocks in seg. hdr's freelists: 1
```

```
 #blocks below: 2
 mapblk  0x00000000  offset: 0
              Unlocked
   Map Header:: next  0x00000000  #extents: 2    obj#: 32387  flag: 0x40000000
 Extent Map
 -----------------------------------------------------------------
  0x02c0000a  length: 4
  0x02c0000e  length: 5

 nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 2
 SEG LST:: flg: USED  lhd: 0x02c0000b ltl: 0x02c0000b
End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
*** 2004-06-14 10:47:50.000
Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
buffer tsn: 14 rdba: 0x02c00009 (11/9)
scn: 0x0000.01503350 seq: 0x02 flg: 0x00 tail: 0x33501002
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
 Extent Control Header
 -----------------------------------------------------------------
 Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
             last map  0x00000000  #maps: 0     offset: 4128
    Highwater::  0x02c0000d  ext#: 0     blk#: 3     ext size: 4
 #blocks in seg. hdr's freelists: 1
 #blocks below: 3
 mapblk  0x00000000  offset: 0
              Unlocked
   Map Header:: next  0x00000000  #extents: 2    obj#: 32387  flag: 0x40000000
 Extent Map
 -----------------------------------------------------------------
  0x02c0000a  length: 4
  0x02c0000e  length: 5

 nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 3
 SEG LST:: flg: USED  lhd: 0x02c0000c ltl: 0x02c0000c
End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
*** 2004-06-14 10:48:04.000
Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
buffer tsn: 14 rdba: 0x02c00009 (11/9)
scn: 0x0000.015033a4 seq: 0x02 flg: 0x00 tail: 0x33a41002
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
 Extent Control Header
 -----------------------------------------------------------------
 Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
             last map  0x00000000  #maps: 0     offset: 4128
    Highwater::  0x02c0000e  ext#: 0     blk#: 4     ext size: 4
 #blocks in seg. hdr's freelists: 1
 #blocks below: 4
 mapblk  0x00000000  offset: 0
              Unlocked
   Map Header:: next  0x00000000  #extents: 2    obj#: 32387  flag: 0x40000000
 Extent Map
 -----------------------------------------------------------------
```

```
      0x02c0000a  length: 4
      0x02c0000e  length: 5

     nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 4
     SEG LST:: flg: USED  lhd: 0x02c0000d ltl: 0x02c0000d
   End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
   *** 2004-06-14 10:50:20.000
   Start dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
   buffer tsn: 14 rdba: 0x02c00009 (11/9)
   scn: 0x0000.0150350e seq: 0x03 flg: 0x00 tail: 0x350e1003
   frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
     Extent Control Header
     -----------------------------------------------------------------
     Extent Header:: spare1: 0     spare2: 0     #extents: 2     #blocks: 9
                  last map  0x00000000  #maps: 0     offset: 4128
        Highwater::  0x02c00013  ext#: 1     blk#: 5     ext size: 5
     #blocks in seg. hdr's freelists: 5
     #blocks below: 9
     mapblk  0x00000000  offset: 1
                  Unlocked
       Map Header:: next  0x00000000  #extents: 2    obj#: 32387  flag: 0x40000000
     Extent Map
     -----------------------------------------------------------------
      0x02c0000a  length: 4
      0x02c0000e  length: 5

     nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 5
     SEG LST:: flg: USED  lhd: 0x02c0000e ltl: 0x02c00012
   End dump data blocks tsn: 14 file#: 11 minblk 9 maxblk 9
```

分析一下这个结果：

```
   Highwater:0x02c0000b → Highwater:  0x02c0000c → Highwater:  0x02c0000d → Highwater:
0x02c0000e → Highwater:  0x02c00013
```

当没有设置_bump_highwater_mark_count 时，在前 5 个数据块，HWM 是以 1 为步长移动的；在 5 块以后，HWM 是以 5 为步长移动的。

对于 ASSM 来说，情况又是不一样的。

对于 extents <= 16 blocks 的情况，HWM 移动遵循：第一次移动→ extent blocks – metadata；第二次移动→ extent blocks。

对于 extents > 16 blocks 的情况，HWM 移动遵循：每次移动 32 个 blocks，但是 HWM 包含未格式化的 block，每次格式化 16 个 block 或者 16 – metadata blocks。

在这里只是提出这样的问题让大家注意，而不对 ASSM 的问题进行专门的讨论。

9.4　HWM 对性能的影响

对一个 table 进行 DML 操作，主要是 insert、update、delete 这三种操作。前面已经讨论了，当一个 table 进行了多次 insert 数据时，table 的 HWM 会不停地提升。现在来假设这样一种情况：如果在这期间对这个 table 进行了大量的 delete 操作，这时 table 的 HWM 会不会随着数据量的

减少而下降呢？下面将通过一个实例来说明这个问题：

这里要先引入一个 procedure（转自 tom 的《Oracle 高级专家编程》）：

```
create or replace procedure show_space
 ( p_segname in varchar2,
  p_owner   in varchar2 default user,
  p_type    in varchar2 default 'TABLE',
  p_partition in varchar2 default NULL )
as
   l_total_blocks            number;
   l_total_bytes             number;
   l_unused_blocks           number;
   l_unused_bytes            number;
   l_LastUsedExtFileId        number;
   l_LastUsedExtBlockId      number;
   l_last_used_block          number;
   procedure p( p_label in varchar2, p_num in number )
   is
   begin
      dbms_output.put_line( rpad(p_label,40,'.') ||
                        p_num );
   end;
begin

   dbms_space.unused_space
   ( segment_owner    => p_owner,
     segment_name     => p_segname,
     segment_type     => p_type,
     partition_name   => p_partition,
     total_blocks     => l_total_blocks,
     total_bytes      => l_total_bytes,
     unused_blocks    => l_unused_blocks,
     unused_bytes     => l_unused_bytes,
     last_used_extent_file_id => l_LastUsedExtFileId,
     last_used_extent_block_id => l_LastUsedExtBlockId,
     last_used_block => l_last_used_block );

   p( 'Total Blocks', l_total_blocks );
   p( 'Total Bytes', l_total_bytes );
   p( 'Unused Blocks', l_unused_blocks );
   p( 'Unused Bytes', l_unused_bytes );
   p( 'Last Used Ext FileId', l_LastUsedExtFileId );
   p( 'Last Used Ext BlockId', l_LastUsedExtBlockId );
   p( 'Last Used Block', l_last_used_block );
end;
/
```

通过这个 procedure 显示的结果，可以得到一个 segment 的 HWM 的位置。在 SQL*Plus 中，要看到这个 procedure 显示的结果，就需要设置 set serveroutput on。这里，HWM = total_blocks – Unused Blocks +1。

来看这样一个实验，使用系统视图 all_objects 来创建测试 table MY_OBJECTS，然后 insert

31007 行数据：

```
SQL> create table MY_OBJECTS as
  2  select * from all_objects;

Table created
SQL> select count(*) from MY_OBJECTS;

  COUNT(*)
----------
     31007

SQL> exec show_space(p_segname=>'MY_OBJECTS',p_owner =>'DLINGER',p_type => 'TABLE');
Total Blocks............................425
Total Bytes.............................3481600
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................439
Last Used Block.........................2
```

这时，使用 show_space 来计算 table MY_OBJECTS 的 HWM，这里 HWM=425 − 3 + 1 = 423；现在对 table MY_OBJECTS 进行 delete 操作，删除前 15000 行数据：

```
SQL> delete from MY_OBJECTS where rownum <15000;

已删除 14999 行。

SQL> exec show_space(p_segname => 'MY_OBJECTS',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks............................425
Total Bytes.............................3481600
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................439
Last Used Block.........................2

PL/SQL 过程已成功完成。
```

现在再来观察 HWM 的结果，可以看到，这里 HWM=425 − 3 + 1 = 423。

HWM 的位置并没有发生变化。这说明对 table MY_OBJECTS 删除了 14999 行数据后，并不会改变 HWM 的位置。

那么，HWM 过高会对数据库的性能有什么样的影响呢？这里以全表扫描为例，来讨论 HWM 过高的不良影响。同样，也通过一个实验来看 full table scan 在 delete 前后访问的 block 数量的情况：

```
SQL> set autotrace traceonly
SQL> select count(*) from MY_OBJECTS;

  COUNT(*)
----------
     31007

Statistics
```

```
----------------------------------------------------------
…
      422  physical reads
        0  redo size
      378  bytes sent via SQL*Net to client
      503  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
…
```

这里，通过 Oracle 的 autotrace 来观察 SQL 的执行情况。看看这个 SQL 访问的 block：422 physical reads。

通过 Statistics 的内容，可以看到，在 table MY_OBJECTS 有 31007 行数据的情况下，对 table MY_OBJECTS 进行一次 full table scan，Oracle 需要访问了 422 个 block。

这里，发现 full table scan 时访问的 block 数和 HWM 之下的 block 数量是一致的。

如果删除 table MY_OBJECTS 的一部分数据后，那么对 table MY_OBJECTS 进行一次 full table scan 需要访问的 block 会不会随着数据行数的减少而降低呢？于是 delete 14999 行数据，这时只剩 16008 行数据了：

```
SQL> delete from MY_OBJECTS where rownum<15000;

14999 rows deleted

SQL> commit;

Commit complete

在这里，把 Oracle 先 shutdown，然后在 startup，以便清空 cache 中的数据。

SQL> set autotrace traceonly

SQL> select count(*) from MY_OBJECTS;

  COUNT(*)
----------
     16008

Statistics
----------------------------------------------------------
…
      422  physical reads
        0  redo size
      378  bytes sent via SQL*Net to client
      503  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
…
```

通过上面的 Statistics 的内容，可以看到，table full scan 仍然访问了 422 个 block。

当没有 delete 前 14999 行数据时，全表扫描需要访问 31007 行数据；而当 delete 了 14999 行数据之后，全表扫描实际需要访问的数据行减少了，但是 Oracle 访问的 block 数量并没有减少。这说明进行 table full scan 时，实际上是对 HWM 下所有的 block 进行访问。众所周知，访问的 block 数量越多，代表需要消耗的资源越多。那么，当一个 table 在进行了大量的 delete 操作后，

或者说，当一个 table 在 HWM 之下的 block 上的数据不饱和时，应该考虑采用一些方法来降低该表的 HWM，以减小 table full scan 时需要访问的 block 数量。

9.5　何时应该降低 HWM

table 包含两种空闲的 block：在 HWM 之上的空闲 block 和在 HWM 之下的空闲 block。

1. 在 HWM 之上的空闲 block

运行 analyze table 后，在 HWM 之上的空闲 block 会在 user_tables 的 EMPTY_BLOCKS 中被统计。这些空闲的 blocks 实际上是从来没有存储过数据的，可以用以下命令来释放这些空间：

```
Alter table table_name deallocate unused;
```

2. 在 HWM 之下的空闲 block

当数据插入到一个 block 后，那么 HWM 就移动到这个 block 之上了。然后后续的操作又将这个 block 中的数据删除了，那么，这个 block 实际上是空闲的。但是这些 blocks 位于 HWM 之下，所以是不会出现在 EMPTY_BLOCKS 中的。那么，这样的 block 过多，是会影响性能的，就像前面讨论过的 table full scan 中看到的那样。

同样用系统视图 all_objects 来创建测试 table MY_OBJECTS，然后随意 delete 其中的一部分数据，然后再对 table MY_OBJECTS 进行分析，来观察现在这个 table 的 HWM 之下的数据分布状况。

9.5.1　对于 LMT 下的 FLM

可以用这个方法来了解一个 table 在 HWM 下有多少 blocks 是不包含数据的：

```
SQL> analyze table MY_OBJECTS compute statistics;

Table analyzed
SQL>  select (1- a.num/ b.num_total)*100 as percent from
  2  (select count(distinct substr(rowid,1,15)) num from MY_OBJECTS)a ,
  3  (select BLOCKS - EMPTY_BLOCKS num_total from user_tables where table_name= 'MY_OBJECTS') b;

   PERCENT
----------
24.8606346
```

从上面的结果，可以看到，table MY_OBJECTS 中 HWM 下有 24.86%的 blocks 是不包含数据的。当这个值比较高的时候，可以考虑用一些方法来释放 HWM 下的空闲 blocks 了。注意，这里一定要先对 table 进行分析。

还可以考察这样一个指标：

```
SQL>select NUM_ROWS*AVG_ROW_LEN/
     ((BLOCKS-EMPTY_BLOCKS)*((100-PCT_FREE)/100)*8192)*100 percnt
  2  from dba_tables where table_name = 'MY_OBJECTS';

    PERCNT
----------
```

```
72.1461836
```

这里，可以看到 table MY_OBJECTS 的 blocks 的平均数据充满度为 72%。注意，这里本人的环境下 Oracle 的 block_size 为 8KB，那么在不同的 block_size 下，应该修改上面 SQL 中的 8192 数值。这里计算时已经除去了 PCTFREE 的部分，MY_OBJECTS 的 PCTFREE 为 10，那么 block 的平均数据充满度实际上是 72%×90%=64.8%。

如果table经常进行全表扫描或范围扫描，那么在这个值比较低的时候，也应该考虑合并 HWM 下的 blocks，将空闲的 block 释放。

9.5.2 对于 ASSM

对于 ASSM 的 segment 来说，考察 HWM 下的 blocks 的空间使用状况相对要简单一些。在这里，可以使用这样一个 procedure 来得到 table 的 blocks 使用情况：

```
create or replace procedure show_space_assm(
p_segname in varchar2,
p_owner in varchar2 default user,
p_type in varchar2 default 'TABLE' )
as
l_fs1_bytes number;
l_fs2_bytes number;
l_fs3_bytes number;
l_fs4_bytes number;
l_fs1_blocks number;
l_fs2_blocks number;
l_fs3_blocks number;
l_fs4_blocks number;
l_full_bytes number;
l_full_blocks number;
l_unformatted_bytes number;
l_unformatted_blocks number;
procedure p( p_label in varchar2, p_num in number )
is
begin
dbms_output.put_line( rpad(p_label,40,'.') ||p_num );
end;
begin
dbms_space.space_usage(
segment_owner     => p_owner,
segment_name      => p_segname,
segment_type      => p_type,
fs1_bytes        => l_fs1_bytes,
fs1_blocks       => l_fs1_blocks,
fs2_bytes        => l_fs2_bytes,
fs2_blocks       => l_fs2_blocks,
fs3_bytes        => l_fs3_bytes,
fs3_blocks       => l_fs3_blocks,
fs4_bytes        => l_fs4_bytes,
fs4_blocks       => l_fs4_blocks,
full_bytes       => l_full_bytes,
```

```
    full_blocks       => l_full_blocks,
    unformatted_blocks => l_unformatted_blocks,
    unformatted_bytes  => l_unformatted_bytes);
    p('free space 0-25% Blocks:',l_fs1_blocks);
    p('free space 25-50% Blocks:',l_fs2_blocks);
    p('free space 50-75% Blocks:',l_fs3_blocks);
    p('free space 75-100% Blocks:',l_fs4_blocks);
    p('Full Blocks:',l_full_blocks);
    p('Unformatted blocks:',l_unformatted_blocks);
    end;
    /
```

在 ASSM 下，block 的空间使用分为 free space：0-25%、25-50%、50-75%、70-100%和 Full 这 5 种情况，show_space_assm 会对需要统计的 table 汇总这 5 种类型的 block 数量。

来看 table HWM1 的空间使用情况：

```
SQL> exec show_space_assm('HWM1','DLINGER');
free space 0-25% Blocks:.................0
free space 25-50% Blocks:................1
free space 50-75% Blocks:................0
free space 75-100% Blocks:...............8
Full Blocks:.....................................417
Unformatted blocks:.........................0
```

这个结果显示，对于 table HWM1，full 的 block 有 417 个，free space 为 75-100% block 有 8 个，free space 25-50% block 有 1 个。当 table HWM 下的 blocks 的状态大多为 free space 较高值时，可以考虑合并 HWM 下的 blocks，将空闲的 block 释放，降低 table 的 HWM。

9.6　如何降低 HWM

在 Oracle 8i 以前的版本中，如果需要降低 segment 的 HWM，可以采用两种方法：EXP/IMP 和 CTAS，对这两种方法大家都很熟悉，这里就不做讨论了，下面介绍其他的几种方法。

9.6.1　Move

从 8i 开始，Oracle 开始提供 move 命令。通常使用这个命令将一个 table segment 从一个 tablespace 移动到另一个 tablespace。

move 实际上是在 block 之间物理地 copy 数据，那么，可以通过这种方式来降低 table 的 HWM。本节先通过一个实验来看看 move 是如何移动数据的。

创建 table TEST_HWM：

```
SQL> create table TEST_HWM (id int ,name char(2000)) tablespace hwm;
Table created
```

往 table TEST_HWM 中 insert 以下数据：

```
insert into TEST_HWM values (1,'aa');
insert into TEST_HWM values (2,'bb');
insert into TEST_HWM values (2,'cc');
insert into TEST_HWM values (3,'dd');
insert into TEST_HWM values (4,'ds');
```

```
insert into TEST_HWM values (5,'dss');
insert into TEST_HWM values (6,'dss');
insert into TEST_HWM values (7,'ess');
insert into TEST_HWM values (8,'es');
insert into TEST_HWM values (9,'es');
insert into TEST_HWM values (10,'es');
```

来看看这个 table 的 rowid、block 的 id 和信息：

```
SQL>  select rowid , id,name from TEST_HWM;

ROWID                       ID    NAME
--------------------------        ----------- ------
AAAH7JAALAAAAAUAAA             1    aa
AAAH7JAALAAAAAUAAB             2    bb
AAAH7JAALAAAAAUAAC             2    cc
AAAH7JAALAAAAAVAAA             3    dd
AAAH7JAALAAAAAVAAB             4    ds
AAAH7JAALAAAAAVAAC             5    dss
AAAH7JAALAAAAAWAAA             6    dss
AAAH7JAALAAAAAWAAB             7    ess
AAAH7JAALAAAAAWAAC             8    es
AAAH7JAALAAAAAXAAA             9    es
AAAH7JAALAAAAAXAAB            10    es
SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2  from dba_extents  where segment_name='TEST_HWM' ;

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0         11           11         19          5
```

这里，简单地介绍一下 rowid 的相关知识。rowid 在磁盘上需要 10 个字节的存储空间并使用 18 个字符来显示，它包含下列组件：

- 数据对象编号：每个数据对象如表或索引在创建时分配，并且此编号在数据库中是惟一的。
- 相关文件编号：此编号对于一个表空间中的每个文件是惟一的。
- 块编号：表示包含此行的块在文件中的位置。
- 行编号：标识块头中行目录位置的位置。

在内部数据对象编号需要 32 位，相关文件编号需要 10 位，块编号需要 22 位，行编号需要 16 位，加起来总共是 80 位或 10 个字节，rowid 使用以 64 为基数的编码方案来显示。该方案将 6 个位置用于数据对象编号，3 个位置用于相关文件编号，6 个位置用于块编号，3 个位置用于行编号。以 64 为基数的编码方案使用字符 A～Z、a～z、0～9、+和/共 64 个字符，如下例所示：

```
AAAH7J AAL AAAAAU AAA
```

在本例中，AAAH7J 是数据对象编号，AAL 是相关文件编号，AAAAAU 是块编号，AAA 是行编号。

那么，根据数据的 rowid，可以看出这 11 行数据分布在 AAAAAU、AAAAAV、AAAAAW 和 AAAAAX 这 4 个 block 中。然后从 table TEST_HWM 中 delete 一些数据：

```
delete from TEST_HWM where id = 2;
delete from TEST_HWM where id = 4;
delete from TEST_HWM where id = 3;
```

```
delete from TEST_HWM where id = 7;
delete from TEST_HWM where id = 8;
delete from TEST_HWM where id = 9;
```

再来看看这个 table 的 rowid、block 的 id 和信息：

```
SQL> select rowid , id,name from TEST_HWM;

ROWID                  ID NAME
------------------ ---------- --------- --------------
AAAH7JAALAAAAAUAAA      1  aa
AAAH7JAALAAAAAVAAC      5  dss
AAAH7JAALAAAAAWAAA      6  dss
AAAH7JAALAAAAAXAAB      10  es
SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2   from dba_extents  where segment_name='TEST_HWM' ;

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0         11          11         19          5
```

在这里，可以看到，数据的 rowid 没有发生改变，根据数据的 rowid，可以看出这 4 行数据依然分布在 AAAAAU、AAAAAV、AAAAAW 和 AAAAAX 这 4 个 block 中。接下来对 table TEST_HWM 进行 move 的操作，然后再来观察 rowid 和 blockid 的信息：

```
SQL> alter table TEST_HWM move;
Table altered

SQL> select rowid,id,name from HWM;

ROWID                   ID NAME
------------------ ---------- -------- ---------------
AAAH7NAALAAAANrAAA      1 aa
AAAH7NAALAAAANrAAB      5 dss
AAAH7NAALAAAANrAAC      6 dss
AAAH7NAALAAAANsAAA      10 es

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2   from dba_extents  where segment_name=' TEST_HWM ' ;

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0         11          11        874          5
```

可以看到，对 table TEST_HWM 进行 move 后，该 table 所在的 blockid 发生了改变，那么数据的 rowid 自然也发生了改变。从上面的结果可以看到，现在 table TEST_HWM 的数据分布在 AAAANr 和 AAAANs 2 个 block 中了。但是从这 4 行数据的 rowid 的顺序来看，这 4 行数据在 table 中的存储顺序并没有发生改变。move 是在 block 之间对于数据的物理 copy。

再来看看 move 操作对于 table 的 HWM 的位置有什么变化，同样使用系统视图 all_objects 来创建测试 table my_objects，然后 delete 前 9999 行数据：

```
SQL> create table my_objects tablespace HWM
  2  as select * from all_objects;
SQL> delete from my_objects where rownum<10000;
9999 rows deleted
```

```
SQL> select count(*) from my_objects;
  COUNT(*)
----------
    21015
SQL> exec show_space(p_segname => 'MY_OBJECTS',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks............................425
Total Bytes.............................3481600
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................1294
Last Used Block.........................2
```

这里 HWM=425 − 3 + 1 = 423，然后对 table MY_OBJECTS 进行 move 操作：

```
SQL> alter table MY_OBJECTS move;
表已更改。
SQL> exec show_space(p_segname => 'MY_OBJECTS',p_owner => 'DLINGER',p_type =>'TABLE');
Total Blocks............................290
Total Bytes.............................2375680
Unused Blocks...........................1
Unused Bytes............................8192
Last Used Ext FileId....................11
Last Used Ext BlockId...................1584
Last Used Block.........................4
```

可以看到，table MY_OBJECTS 的 HWM 从 423 移动到 290，table 的 HWM 降低了！

还可以使用别的方法来降低 table 的 HWM，比如 CTAS、insert into 等，那么 move 操作对 redo logo 的写和其他的方式比较起来是相对较少的，在这里就不列出具体的实验结果了，大家有兴趣的可以自己动手来证实一下。

上面讨论了 move 的执行机制和如何使用 move 降低 table 的 HWM，下面将补充说明 move 的另外一些用法，以及使用 move 时要注意的一些问题。

1. Move 的一些用法

以下是 alter table 中 move 子句的完整语法，这里介绍其中的几点：

```
MOVE [ONLINE]
  [segment_attributes_clause]
  [data_segment_compression]
  [index_org_table_clause]
  [ { LOB_storage_clause | varray_col_properties }
    [ { LOB_storage_clause | varray_col_properties } ]...
  ]
  [parallel_clause]
```

可以使用 move 将一个 table 从当前的 tablespace 上移动到另一个 tablespace 上，例如：

```
alter table t move tablespace tablespace_name;
```

还可以用 move 来改变 table 已有的 block 的存储参数，例如：

```
alter table t move storage (initial 30k  next 50k);
```

另外，move 操作也可以用来解决 table 中的行迁移问题。

2. 使用 Move 的一些注意事项

(1) table 上的 index 需要 rebuild

在前面讨论过，move 操作后，数据的 rowid 发生了改变，因为 index 是通过 rowid 来 fetch 数据行的，所以，table 上的 index 是必须要 rebuild 的。

```
SQL>  create index i_my_objects on my_objects (object_id);
Index created

SQL> alter table my_objects move;
Table altered

SQL> select index_name,status from user_indexes where index_name='I_MY_OBJECTS';

INDEX_NAME                     STATUS
------------------------------ --------
I_MY_OBJECTS                   UNUSABLE
```

从这里可以看到，当 table MY_OBJECTS 进行 move 操作后，该 table 上的 index 的状态为 UNUSABLE，这时可以使用 alter index I_MY_OBJECTS rebuild online 的命令，对 index I_MY_OBJECTS 进行在线 rebuild。

(2) Move 时对 table 的锁定

当对 table MY_OBJECTS 进行 move 操作时，查询 v$locked_objects 视图可以发现，table MY_OBJECTS 上加了 exclusive lock。

```
SQL>select OBJECT_ID, SESSION_ID,ORACLE_USERNAME,LOCKED_MODE from v$locked_objects;

 OBJECT_ID SESSION_ID ORACLE_USERNAME    LOCKED_MODE
---------- ---------- ------------------ -----------
     32471          9 DLINGER                    6
SQL> select object_id from user_objects where object_name = 'MY_OBJECTS';

 OBJECT_ID
----------
     32471
```

这就意味着，table 在进行 move 操作时，只能对它进行 select 的操作。反过来说，当一个 session 对 table 进行 DML 操作且没有 commit 时，在另一个 session 中是不能对这个 table 进行 move 操作的，否则 Oracle 会返回这样的错误信息“ ORA-00054：资源正忙，要求指定 NOWAIT ”。

(3) 关于 Move 时空间使用的问题

当使用 alter table move 来降低 table 的 HWM 时，有一点是需要注意的，这时，当前的 tablespace 中需要有 1 倍于 table 的空闲空间以供使用：

```
SQL> CREATE TABLESPACE TEST1
  2  DATAFILE 'D:\ORACLE\ORADATA\ORACLE9I\TEST1.dbf' SIZE 5M
  3  UNIFORM SIZE 128K ;

SQL> create table my_objects tablespace test1 as select * from all_objects;
表已创建。

SQL> select bytes/1024/1024 from user_segments where segment_name='MY_OBJECTS';
```

```
BYTES/1024/1024
---------------
       3.125

SQL> alter table MY_OBJECTS move;

alter table MY_OBJECTS move
       *
ERROR 位于第 1 行:
ORA-01652: 无法通过 16（在表空间 TEST1 中）扩展 temp 段

SQL> ALTER DATABASE
  2  DATAFILE 'D:\ORACLE\ORADATA\ORACLE9I\TEST1.DBF' RESIZE  7M;

数据库已更改。
SQL> alter table MY_OBJECTS move;
表已更改。
```

9.6.2 DBMS_REDEFINITION

这个包是从 Oracle 9i 开始引入的，用来做 table 的联机重组和重定义。可以通过这种方法在线地重组 table，来移动 table 中的数据，降低 HWM，修改 table 的存储参数、分区等。

这个操作要求 table 上有一个主键，并要求预先创建一个带有要求修改的存储参数的 table，以便保存重新组织后的数据。保存重新组织的数据的 table 叫临时表，它只在重新组织期间被使用，在操作完成后可以被删除。

使用 DBMS_REDEFINITION 包需要如下权限：

```
create any table;
alter any table;
drop any table;
lock any table;
select any table;
```

在 DBMS_REDEFINITION 上执行操作。使用 DBMS_REDEFINITION 重组 table 一般是这样几个步骤：

（1）使用 DBMS_REDEFINITION.CAN_REDEF_TABLE()验证所选择的 table 能够被重建。

（2）创建空的临时表，确保这个临时表定义了主键。

（3）使用 DBMS_REDEFINITION.START_REDEF_TABLE()进行 table 的重组。

（4）在临时表上创建触发器、索引和约束，一般来说，这些对象与源表中的是一致的，但是名称必须不同。同时要确保所创建的所有外键约束不可用。在重组结束时，所有这些对象将替换定义在源表上的对象。

（5）使用 DBMS_REDEFINITION.FINISH_REDEF_TABLE()完成重组的过程。在这期间，源表将会 lock 较短的时间。

（6）删除临时表。

在这里，只是简单地介绍如何使用 DBMS_REDEFINITION 对 table 进行在线重组和重定义，关于这个 package 具体的使用方法和使用上的限制，可以查阅 Oracle 的官方文档（http://tahiti.Oracle.com/）。

9.6.3　Shrink

从 10g 开始，Oracle 开始提供 shrink 命令，假如表空间中支持自动段空间管理（ASSM），就可以使用这个特性缩小段，即降低 HWM。这里需要强调一点，10g 的这个新特性，仅对 ASSM 表空间有效，否则会报 “ ORA-10635: Invalid segment or tablespace type ”。

在 9.4 节中已经讨论过，如何考察在 ASSM 下 table 是否需要回收浪费的空间，下面将讨论如何对一个 ASSM 的 segment 回收浪费的空间。

同样，用系统视图 all_objects 在 tablespace ASSM 上创建测试表 my_objects（本小节的实验环境为 Oracle 10.1.0.2）：

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

BANNER
----------------------------------------------------------------
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Prod
PL/SQL Release 10.1.0.2.0 - Production
CORE           10.1.0.2.0    Production

TNS for 32-bit Windows: Version 10.1.0.2.0 - Production
NLSRTL Version 10.1.0.2.0 - Production

SQL> select TABLESPACE_NAME,BLOCK_SIZE,EXTENT_MANAGEMENT,
  2  ALLOCATION_TYPE, SEGMENT_SPACE_MANAGEMENT
  3  from dba_tablespaces where TABLESPACE_NAME = 'ASSM';

TABLESPACE_NAME  BLOCK_SIZE EXTENT_MANAGEMENT ALLOCATION_TYPE SEGMENT_SPACE_MANAGEMENT
---------------- ---------- ----------------- --------------- ------------------------
ASSM                 8192 LOCAL           UNIFORM        AUTO

SQL> create table my_objects tablespace assm
  2  as select * from all_objects;
Table created
```

然后随机地从 table MY_OBJECTS 中删除一部分数据：

```
SQL> select count(*) from my_objects;
  COUNT(*)
----------
    47828
SQL> delete from my_objects where object_name like '%C%';
16950 rows deleted

SQL> delete from my_objects where object_name like '%U%';
4503 rows deleted

SQL> delete from my_objects where object_name like '%A%';
6739 rows deleted
```

现在使用 show_space 和 show_space_assm 来看看 my_objects 的数据存储状况：

```
SQL> exec show_space('MY_OBJECTS','DLINGER');
Total Blocks.............................680
Total Bytes.............................5570560
```

```
Unused Blocks.....................1
Unused Bytes.....................8192
Last Used Ext FileId.....................6
Last Used Ext BlockId.....................793
Last Used Block.....................4

PL/SQL 过程已成功完成。

SQL> exec show_space_assm('MY_OBJECTS','DLINGER');
free space 0-25% Blocks:.....................0
free space 25-50% Blocks:.....................205
free space 50-75% Blocks:.....................180
free space 75-100% Blocks:.....................229
Full Blocks:.....................45
Unformatted blocks:.....................0

PL/SQL 过程已成功完成。
```

这里，table my_objects 的 HWM 下有 679 个 block，其中，free space 为 25-50%的 block 有 205 个，free space 为 50-75%的 block 有 180 个，free space 为 75-100%的 block 有 229 个，Full 的 block 只有 45 个，这种情况下，需要对这个 table 的现有数据行进行重组。

要使用 ASSM 上的 shrink，首先需要使该表支持行移动，可以用这样的命令来完成：

```
alter table my_objects enable row movement;
```

现在，可以使用以下命令来降低 my_objects 的 HWM 并回收空间：

```
alter table bookings shrink space;
```

具体地看一下实验的结果：

```
SQL> alter table my_objects enable row movement;
表已更改。

SQL> alter table my_objects shrink space;
表已更改。

SQL> exec show_space('MY_OBJECTS','DLINGER');
Total Blocks.....................265
Total Bytes.....................2170880
Unused Blocks.....................2
Unused Bytes.....................16384
Last Used Ext FileId.....................6
Last Used Ext BlockId.....................308
Last Used Block.....................3

PL/SQL 过程已成功完成。

SQL> exec show_space_assm('MY_OBJECTS','DLINGER');
free space 0-25% Blocks:.....................0
free space 25-50% Blocks:.....................1
free space 50-75% Blocks:.....................0
free space 75-100% Blocks:.....................0
Full Blocks:.....................249
Unformatted blocks:.....................0
```

```
PL/SQL 过程已成功完成。
```

在执行完 shrink 命令后，可以看到，table my_objects 的 HWM 现在降到了 264 的位置，而且 HWM 下的 block 的空间使用状况为：Full 的 block 有 249 个，free space 为 25-50% block 只有 1 个。

接下来讨论一下 shrink 的实现机制，同样使用讨论 move 机制的那个实验来观察。

```
SQL> create table TEST_HWM (id int ,name char(2000)) tablespace ASSM;

Table created
```

往 table TEST_HWM 中插入如下的数据：

```
insert into TEST_HWM values (1,'aa');
insert into TEST_HWM values (2,'bb');
insert into TEST_HWM values (2,'cc');
insert into TEST_HWM values (3,'dd');
insert into TEST_HWM values (4,'ds');
insert into TEST_HWM values (5,'dss');
insert into TEST_HWM values (6,'dss');
insert into TEST_HWM values (7,'ess');
insert into TEST_HWM values (8,'es');
insert into TEST_HWM values (9,'es');
insert into TEST_HWM values (10,'es');
```

来看看这个 table 的 rowid、block 的 id 和信息：

```
SQL>  select rowid , id,name from TEST_HWM;

ROWID                  ID NAME
------------------ ---------- ----- ---------
AAANhqAAGAAAAFHAAA       1 aa
AAANhqAAGAAAAFHAAB       2 bb
AAANhqAAGAAAAFHAAC       2 cc
AAANhqAAGAAAAFIAAA        3 dd
AAANhqAAGAAAAFIAAB        4 ds
AAANhqAAGAAAAFIAAC        5 dss
AAANhqAAGAAAAFJAAA        6 dss
AAANhqAAGAAAAFJAAB        7 ess
AAANhqAAGAAAAFJAAC        8 es
AAANhqAAGAAAAFKAAA        9 es
AAANhqAAGAAAAFKAAB        10 es

11 rows selected

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2  from dba_extents  where segment_name='TEST_HWM' ;

EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
        0          6            6        324          5
        1          6            6        329          5
```

然后从 table TEST_HWM 中删除一些数据：

```
delete from TEST_HWM where id = 2;
delete from TEST_HWM where id = 4;
delete from TEST_HWM where id = 3;
```

```
delete from TEST_HWM where id = 7;
delete from TEST_HWM where id = 8;
```

观察 table TEST_HWM 的 rowid 和 blockid 的信息：

```
SQL> select rowid , id,name from TEST_HWM;

ROWID                  ID NAME
------------------ ---------- ----- --------
AAANhqAAGAAAAFHAAA      1 aa
AAANhqAAGAAAAFIAAC      5 dss
AAANhqAAGAAAAFJAAA      6 dss
AAANhqAAGAAAAFKAAA      9 es
AAANhqAAGAAAAFKAAB      10 es

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2   from dba_extents  where segment_name='TEST_HWM' ;

EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
       0         6           6       324         5
       1         6           6       329         5
```

从以上的信息，可以看到，在 table TEST_HWM 中，剩下的数据是分布在 AAAAFH、AAAAFI、AAAAFJ 和 AAAAFK 这样 4 个连续的 block 中。

```
SQL> exec show_space_assm('TEST_HWM','DLINGER');
free space 0-25% Blocks:.................0
free space 25-50% Blocks:................1
free space 50-75% Blocks:................3
free space 75-100% Blocks:...............3
Full Blocks:.............................0
Unformatted blocks:......................0
```

通过 show_space_assm 可以看到目前这 4 个 block 的空间使用状况，AAAAFH、AAAAFI、AAAAFJ 上各有一行数据，可以猜测 free space 为 50-75%的 3 个 block 是这三个 block，那么 free space 为 25-50%的 1 个 block 就是 AAAAFK 了，剩下 free space 为 75-100%的 3 个 block，是 HWM 下已格式化的尚未使用的 block（关于 ASSM 下 HWM 的移动，上文已经详细地讨论过了，在 extent 不大于 16 个 block 时，是以一个 extent 为单位来移动的）。

然后，对 table my_objects 执行 shrink 的操作：

```
SQL> alter table test_hwm enable row movement;

Table altered

SQL> alter table test_hwm shrink space;

Table altered

SQL> select rowid ,id,name from TEST_HWM;

ROWID                   ID NAME
------------------ ---------- ------ -----------
AAANhqAAGAAAAFHAAA     1 aa
AAANhqAAGAAAAFHAAB     10 es
```

```
AAANhqAAGAAAAFHAAD     9 es
AAANhqAAGAAAAFIAAC      5 dss
AAANhqAAGAAAAFJAAA      6 dss

SQL> select EXTENT_ID,FILE_ID,RELATIVE_FNO,BLOCK_ID,BLOCKS
  2  from dba_extents  where segment_name='TEST_HWM' ;

 EXTENT_ID    FILE_ID RELATIVE_FNO   BLOCK_ID    BLOCKS
---------- ---------- ------------ ---------- ----------
         0          6            6        324          5
         1          6            6        329          5
```

当执行了 shrink 操作后，有意思的现象出现了。来看看 Oracle 是如何移动行数据的，这里的情况和 move 已经不太一样了。在 move 操作的时候，所有行的 rowid 都发生了变化，table 所位于的 block 的区域也发生了变化，但是所有行物理存储的顺序都没有发生变化，所以得到的结论是，Oracle 以 block 为单位，进行了 block 间的数据 copy。那么 shrink 后，可以发现，部分行数据的 rowid 发生了变化，同时，部分行数据的物理存储的顺序也发生了变化，而 table 所位于的 block 的区域却没有变化，这就说明，shrink 只移动了 table 其中一部分的行数据来完成释放空间，而且，这个过程是在 table 当前所使用的 block 中完成的。

那么 Oracle 移动行数据的具体过程是怎样的呢？根据这样的实验结果，可以来猜测一下。

Oracle 是以行为单位来移动数据的。Oracle 从当前 table 存储的最后一行数据开始移动，从当前 table 最先使用的 block 开始搜索空间，所以，在 shrink 之前，rownum=10 的那行数据（10,es），被移动到 block AAAAFH 上，写到（1,aa）这行数据的后面，所以（10,es）的 rownum 和 rowid 同时发生改变。然后是（9,es）这行数据，重复上述过程。这是 Oracle 从后向前移动行数据所大致遵循的规则，那么移动行数据的具体算法是比较复杂的，包括向 ASSM 的 table 中 insert 数据时使用 block 的顺序的算法也是比较复杂的，大家有兴趣的可以自己深入研究，在这里不多做讨论。

还可以在 shrink table 的同时 shrink 这个 table 上的 index：

```
alter table my_objects shrink space cascade;
```

同样地，这个操作只有当 table 上的 index 也是 ASSM 时才能使用。

下面讨论一下关于使用 shrink 的几个问题。

1. Shrink 后 index 是否需要 rebuild

因为 shrink 的操作也会改变行数据的 rowid，那么，如果 table 上有 index 时，shrink table 后 index 会不会变为 UNUSABLE 呢？来看下面的实验，同样构建 my_objects 的测试表：

```
create table my_objects  tablespace ASSM as select * from all_objects where rownum<20000;
create index i_my_objects on my_objects (object_id);
delete from my_objects where object_name like '%C%';
delete from my_objects where object_name like '%U%';
```

现在来 shrink table my_objects：

```
SQL> alter table my_objects enable row movement;

Table altered

SQL> alter table my_objects shrink space;

Table altered
```

```
SQL> select index_name,status from user_indexes where index_name='I_MY_OBJECTS';

INDEX_NAME                     STATUS
------------------------------ --------
I_MY_OBJECTS                    VALID
```

可以发现，table my_objects 上的 index 的状态为 VALID，估计 shrink 在移动行数据时，也一起维护了 index 上相应行的数据 rowid 的信息。可以认为，这是对于 move 操作后需要 rebuild index 的改进。但是如果一个 table 上的 index 数量较多，因为维护 index 的成本是比较高的，所以 shrink 过程中用来维护 index 的成本也会比较高。

2. Shrink 时对 table 的 lock

在对 table 进行 shrink 时，会对 table 进行怎样的锁定呢？当对 table MY_OBJECTS 进行 shrink 操作时，查询 v$locked_objects 视图可以发现，table MY_OBJECTS 上加了 row-X (SX) 的 lock：

```
SQL>select OBJECT_ID, SESSION_ID,ORACLE_USERNAME,LOCKED_MODE from v$locked_objects;

 OBJECT_ID SESSION_ID ORACLE_USERNAME    LOCKED_MODE
---------- ---------- ------------------ -----------
     55422        153 DLINGER                      3
SQL> select object_id from user_objects where object_name = 'MY_OBJECTS';

 OBJECT_ID
----------
     55422
```

那么，当 table 在进行 shrink 时，对 table 是可以进行 DML 操作的。

3. Shrink 对空间的要求

前面已经讨论了 shrink 移动数据的机制，既然 Oracle 是从后向前移动行数据，那么，shrink 的操作就不会像 move 一样，shrink 不需要使用额外的空闲空间。

9.6.4 小结

在这一部分中介绍了 3 种降低 table HWM 的方法，那么在实际的环境中，是选择 move 还是 shrink，可以针对这几项的特性，结合系统的具体情况，再做出选择。

9.7 其他几种会移动 HWM 的操作

还有几种操作是会改变 HWM 的，如 insert append、truncate 等。还有一些方法也可以用来降低 HWM，如 exp/imp 等，在这里不做讨论。

9.7.1 Insert Append

当使用 insert /*+ append */ into 向一个 table 中插入数据时，Oracle 不会在 HWM 以下寻找

空间，而是直接移动 HWM，从 EMPTY_BLOCKS 中获得要使用的 block 空间，来满足这一操作对 blocks 的需要。

下面来看一个实验：

```
SQL> create table hwm as select * from all_objects;

Table created

SQL> select count(*) from hwm;

  COUNT(*)
----------
     31009
SQL> delete from hwm;

31009 rows deleted
SQL> exec show_space(p_segname => 'HWM',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks............................425
Total Bytes.............................3481600
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................439
Last Used Block.........................2
```

往表 HWM 中先插入 31009 条数据，然后再 delete 掉所有的数据。前面讨论过，delete 操作不会降低 HWM，所以这时的 HWM = 425 − 3 + 1 = 423。

下面来比较一下 insert 和 insert append 的不同结果。先使用 insert into 向表 HWM 中插入 1000 行数据，结果 HWM 没有移动。

```
SQL> insert into hwm select * from all_objects where rownum<1000;

999 rows inserted

SQL> commit;

Commit complete
SQL>  exec show_space(p_segname => 'HWM',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks............................425
Total Bytes.............................3481600
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................439
Last Used Block.........................2
```

然后 delete 掉所有的数据，再用 insert append 来做同样的操作。可以看到，使用 append 提示后，结果就不一样了。

```
SQL> delete from hwm;

999 rows deleted

SQL> commit;
```

```
Commit complete

SQL> insert /*+ append */ into hwm select * from all_objects where rownum<1000;

999 rows inserted

SQL> commit;

Commit complete
SQL>  exec show_space(p_segname => 'HWM',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks............................440
Total Bytes.............................3604480
Unused Blocks...........................3
Unused Bytes............................24576
Last Used Ext FileId....................11
Last Used Ext BlockId...................459
Last Used Block.........................2
```

可以发现，往 HWM 中插入同样的 999 行数据，使用 insert append，此时 HWM = 440 − 3 + 1 = 438 ，HWM 从 423 移动到了 438！

再来比较一下 insert 和 insert append 的性能，对 HWM 插入同样的 10000 条数据。构建表 T：

```
SQL> create table t as select * from all_objects;

Table created
SQL> insert /*+ append */ into t select * from t;
31010 rows inserted
SQL> commit;
Commit complete
SQL>  insert /*+ append */ into t select * from t;
62020 rows inserted
SQL> commit;
Commit complete
SQL> select count(*) from t;

  COUNT(*)
----------
    124040
```

对 HWM 插入数据：

```
SQL> set timing on
SQL> insert into hwm select * from t;

124040 rows inserted

已用时间:  00: 00: 02.93
SQL> commit;

Commit complete

已用时间:  00: 00: 00.20
SQL> insert /*+ append */  into hwm select * from t;
```

```
124040 行 rows inserted

已用时间:  00: 00: 01.02
SQL> commit;

Commit complete

已用时间:  00: 00: 00.30
```

当使用 insert 来插入 124040 行数据时，使用了 2.93 秒；而使用 insert append 插入 124040 行数据时，只使用了 1.02 秒。

在这里，提一下使用 insert append 需要注意的一个问题：当使用 insert append 时，Oracle 会生成表级的独占锁。

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

      SID STATISTIC#     VALUE
---------- ---------- ----------
       13         0         1

SQL> insert /*+ append */ into hwm select * from all_objects where rownum<1000;
999 rows inserted  --我们在这里不作 commit

--在另一个 session 中执行:
QL> select * from v$mystat where rownum <2;

      SID STATISTIC#     VALUE
---------- ---------- ----------
       10         0         1

SQL> insert into hwm select * from all_objects where rownum<10;
--这个 session 出现等待
```

现在观察 v$lock：

```
SQL> select SID,TYPE,ID1,ID2,LMODE,REQUEST,BLOCK from v$lock;

SID TYPE       ID1       ID2     LMODE    REQUEST     BLOCK
---------- ---- ---------- ---------- ---------- ---------- ----------
……
10 TM       32398         0         0         3         0
13 TX       65579     22477         6         0         0
13 TM       32398         0         6         0         1
                                                  --13 阻塞了一个 process
SQL> select object_name from user_objects where object_id = '32398';

OBJECT_NAME
--------------------------------------------------------------------------------
HWM
```

session 13 在 HWM 上加上了 exclusive 的 TM 锁，这时 session 13 blocking 了 session 10。这里是在 LMT 下的 segment 中做的测试。在 ASSM 中 append 锁表的情况同样存在（直到 Oracle10g 的 ASSM 中依然如此）。

9.7.2 Truncate

讨论 truncate table，一般是和 delete from table 做比较。前面，已经讨论过 delete 不会降低 HWM 的问题，这里再来看一下 truncate 的情况：

```
SQL> exec show_space(p_segname => 'HWM',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks.............................3380
Total Bytes..............................27688960
Unused Blocks............................18
Unused Bytes.............................147456
Last Used Ext FileId.....................11
Last Used Ext BlockId....................5069
Last Used Block..........................2

PL/SQL 过程已成功完成。
--这里 HWM = 3380 - 18 + 1= 3363
SQL> truncate table HWM;

表已截掉。

SQL> exec show_space(p_segname => 'HWM',p_owner => 'DLINGER',p_type => 'TABLE');
Total Blocks.............................5
Total Bytes..............................40960
Unused Blocks............................4
Unused Bytes.............................32768
Last Used Ext FileId.....................11
Last Used Ext BlockId....................19
Last Used Block..........................1

PL/SQL 过程已成功完成。
--执行 truncate 后 HWM = 5 - 4 + 1 = 2
```

可以发现，truncate table 之后，HWM 又回到了 9.1 节中的 segment 初始化状态下 HWM 的位置。

作者简介

段凌，ITPUB 入门及认证版版主，ITPUB ID dlinger，又称“凌波微步”。

一个 Oracle 的初学者，曾任职某国内大型软件公司数据库设计师，主要负责产品和项目的多数据效率优化，现任职某著名外企 SR.Database Engineer。

“也没什么特别擅长的，对于数据库，我惟一执著的，是自己的兴趣而已，需要走的路，也还很长”。

第 10 章　调整 I/O 相关的等待

本章主要介绍了如何在出现 I/O 竞争等待时对 Oracle 数据库进行优化。对 Oracle 数据库进行调整优化，基本上最终都可以归结到 I/O 调整上，因此，了解如何来优化 Oracle 数据库的 I/O 对于一个 DBA 来说就显得十分重要了。

10.1　Oracle 数据库 I/O 相关竞争等待简介

当 Oracle 数据库出现与 I/O 相关的竞争等待时，通常都会引起 Oracle 数据库的性能下降，一般可以通过以下 3 种方法来查看 Oracle 数据库是否存在 I/O 相关的竞争等待：

- Statpack 报告的 Top 5 Wait Events 部分中主要都是 I/O 相关的等待事件。
- 数据库等待事件的 SQL 语句跟踪中主要都是 I/O 相关的等待事件的限制。
- 操作系统工具显示存储数据库文件的存储磁盘有非常高的利用率。

如果发现数据库存在 I/O 竞争，那就必须要通过各种方法来调整优化 Oracle 数据库。在调优数据库的过程中，其中一个重要的步骤就是对响应时间的分析，看看数据库消耗的时间究竟是消耗在什么上面了。对于 Oracle 数据库来说，响应时间的分析可以通过以下公式来计算：

```
Response Time = Service Time + Wait Time
```

其中 Service Time 是指“ CPU used by this session ”的统计时间；Wait Time 是指所有消耗在等待事件上的总的时间。

如果使用性能调整工具（如 Statpack）来调整数据库时，评测的则是所有响应时间中各个部分的相对影响，并且应该根据消耗的时间的多少来调整影响最严重的部分。

因为等待事件有很多，所以还需要去判定哪些是真的很重要的等待事件，很多调优工具如 Statpack 都会列出最重要的等待事件，Statpack 工具的报告中的重要的等待事件都是包含在 Top 5 Wait Events 部分中。因为这些工具都已经把重要的等待事件全部列出来了，因此就很容易处理这些已经列出来的等待事件而不必再去首先评估所有响应时间的影响。在某些情况下，Service Time 会比 Wait Time 显得更加重要（如 CPU 使用率），此时等待事件产生的影响就显得不是那么重要了，重点调整的目标应该放在 Service Time 上。因此，应该先比较在 Top 5 Wait Events 部分中的“ CPU used by this session ”所占用的时间，然后直接调整最消耗时间的等待事件。在 Oracle 9i 的 release2

的版本以后，Top 5 Wait Events 部分变成了 Top 5 Timed Events，Service Time 也由“ CPU used by this session ”变成了“ CPU time ”来衡量，这也就意味着可以更加精确地判断在响应时间中的等待事件的影响，从而调整最需要优化的部分。

下面举一个例子来具体说明为什么在调整数据库性能的时候必须同时查看 Service Time 和 Wait Time，因为如果不同时查看这两个方面的信息，就往往容易走入调整的误区。

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~
                                        Wait          % Total Event
                                        Waits       Time (cs)    Wt Time
------------------------------------ ------------ ------------ -------
direct path read                         9,590     15,546      86.10 db
file scattered read                      6,105      1,262       6.99 latch
free                                     2,036      1,047       5.80 log file
sync                                       107        131        .73
db file parallel write                      40         69        .38
```

上面是 Statpack 收集了大约 30 分钟的 Top 5 Wait Events 部分的信息，如果基于上面给出的列表，很容易发现 direct path read 的 wait 很高，并且会试图调整这个等待事件，但是这样做就没有考虑到 Service Time。再来看看在这个 Statpack 中关于 Service Time 的统计：

```
Statistic                             Total   per Second    per Trans
------------------------------- ------------- ------------ ------------
CPU used by this session            429,648       238.7       257.4
```

可以大致地计算一下响应时间：

```
Wait Time = 15,546×100% / 86.10% = 18,056 cs
Service Time = 429,648 cs
Response Time = 429,648 + 18,056 = 44,7704 cs
```

接着来计算一下响应时间中各个部分的比例：

```
CPU time           = 95.97%
direct path read        =  3.47%
db file scattered read    =  0.28%
latch free            =  0.23%
log file sync          =  0.03%
db file parallel write    =  0.02%
```

从上面的计算中可以明显地看出来，I/O 相关的等待事件所消耗的时间在整个响应时间中占的比例并不大，只不过是很小的一部分，而相对来说 Service Time 所消耗的时间远远大于 Wait Time，因此，应该直接调整的是 Service Time（CPU 的使用率）而不是 I/O 相关的等待事件，因此，在调优数据库的时候要尽量避免走入这种误区。

10.2 Oracle 数据库 I/O 相关竞争等待的处理方法

接下来具体看看对于出现的 I/O 问题的一些处理方法。

在使用 Statpack 这类工具分析数据库的响应时间后，如果数据库的性能主要是被一些 I/O 相关的等待事件所限制住了，那么可以针对这种情况可以采用处理 I/O 问题的一些方法，下面对这些方法的一些概念和基本原理进行简单的阐述说明。

1. 优化 Oracle 数据库的 SQL 语句来减少数据库对 I/O 的需求

如果数据库没有运行任何用户的 SQL 的话，一般来说只会产生很少的磁盘 I/O 或者几乎没有磁盘 I/O，基本上来说数据库产生 I/O 的最终原因都是直接或者间接的由于用户执行 SQL 语句导致的。这也就意味着可以控制单个 SQL 语句避免其产生大量的 I/O 来减少整个数据库对磁盘 I/O 的需求，通过优化 SQL 语句改变其执行计划以便让其产生尽可能少的 I/O。一般典型的存在问题的情况仅仅只是很少的几个 SQL 语句，但是由于其相应的执行计划不理想，会导致产生大量的物理磁盘 I/O，从而使得整个数据库的性能非常之差。因此，让用户执行的 SQL 语句优化产生比较好的执行计划来减少磁盘 I/O 是一种非常行之有效的方法。

2. 调整实例的初始化参数来减少数据库的 I/O 需求

一般来说可以通过两种途径来减少数据库的 I/O 需求。

一种途径是通过内存缓存来减少 I/O。数据库的 I/O 分为两种，一种是实际读取了数据文件的物理 I/O，一种是从缓存中读取数据的逻辑 I/O，可以通过使用一定数量的内存缓存来减少物理 I/O（例如高速缓存区、日志缓存区以及各种排序区等）。适当的增大高速缓存区，可以有更多的缓存供给数据库的进程使用，从缓存中读取所需要的数据，这样产生的 I/O 都是逻辑 I/O，而不是直接从物理磁盘上读取数据，减少了物理 I/O 的产生。设置一个适当的排序区，可以减少在排序操作中读取临时表空间数据文件所在磁盘的次数，而尽可能多的使用缓存中的排序区来排序。其他的缓存区的工作原理基本都是一致的，都是通过使用缓存来减少读取物理磁盘的次数来降低 I/O。

另外一种途径是调整一次读取多个 block 的大小，单独的一次多个 block 读取的操作的大小是由实例的初始化参数 db_file_multiblock_read_count 来控制的。如果执行比较大的 I/O 操作，一次读取的多个 block 大小越大，所需要的时间就会越短。例如，操作系统的一次能够传输的最大 I/O 大小是 8MB，一次性请求传输 50MB 的数据会比请求 5 次每次只是请求 10MB 的速度快得多。但是，当一次读取的多个 block 的大小超过了操作系统一次能够传输的最大的 I/O 大小，这个差别就基本不明显了。如一次性传输 100M 和一次性传输 1G 的大小在时间上基本上没有什么差别了，效率差不多了。因为整个 I/O 的所消耗的时间分为 I/O Setup Time 和 I/O Transfer Time 两个主要部分，I/O Setup Time 可以看成是 I/O 的寻道所消耗的时间，I/O Transfer Time 可以看成是 I/O 传输所消耗的时间。当一次读取的多个 block 的大小比较小的时候，读取的一定数量的 block 就使得读取次数就会比较多，每次 I/O 读取都要先寻道，并且寻道时间在这个时候所用的时间会占到整个 I/O 完成时间的绝大部分，导致整个 I/O 完成所消耗的时间也就会比较多了。而 I/O 传输一定数量的 block 的时间相对固定，不管传输的次数多少，基本上变化不大，因此影响 I/O 读取时间长短的主要就是取决于 I/O Setup Time 所花费的时间。

因此，在配置数据库初始化参数的时候，根据操作系统的 I/O 吞吐能力都会设置的一次读取多个 Block 的大小尽量多，以减少读取 I/O 的次数。

3. 在操作系统级别上优化 I/O

在操作系统级别上优化磁盘的 I/O，以提高 I/O 的吞吐量，如果操作系统支持异步 I/O，尽量去使用异步 I/O；还可以使用高级文件系统的一些特性，例如直接 I/O 读取，忽略掉操作系统的文件缓存，也就是平时所说的使用裸设备。还有一种可行的方法是增大每次传输的最大 I/O 大小的限制，以便每次能够传输的 I/O 尽可能的大。

4. 通过使用 RAID、SAN、NAS 来平衡数据库的 I/O

RAID 是 Redundent Array of Independent Disks 的缩写，直译为“廉价冗余磁盘阵列”，也简称为“磁盘阵列”。RAID 的优点是传输速率高并且可以提供容错功能。在 RAID 中，可以让很多磁盘驱动器同时传输数据，而这些磁盘驱动器在逻辑上又是一个磁盘驱动器，所以使用 RAID 可以达到单个磁盘驱动器几倍、几十倍甚至上百倍的速率。因为普通磁盘驱动器无法提供容错功能，如果不包括写在磁盘上的 CRC（循环冗余校验）码的话。RAID 容错是建立在每个磁盘驱动器的硬件容错功能之上的，所以它提供更高的安全性。RAID 分为以下几个级别：

- RAID 0：RAID 0 并不是真正的 RAID 结构，没有数据冗余。RAID 0 连续地分割数据并并行地读/写于多个磁盘上。因此具有很高的数据传输率，但 RAID 0 在提高性能的同时，并没有提供数据可靠性，如果一个磁盘失效，将影响整个数据。因此 RAID 0 不可应用于需要数据高可用性的关键应用。
- RAID 1：RAID 1 通过数据镜像实现数据冗余，在两对分离的磁盘上产生互为备份的数据。RAID 1 可以提高读的性能，当原始数据繁忙时，可直接从镜像拷贝中读取数据。RAID 1 是磁盘阵列中费用最高的，但提供了最高的数据可用率。当一个磁盘失效，系统可以自动地交换到镜像磁盘上，而不需要重组失效的数据。
- RAID 2：从概念上讲，RAID 2 同 RAID 3 类似，两者都是将数据条块化分布于不同的硬盘上，条块单位为位或字节。然而 RAID 2 使用称为“加重平均纠错码”的编码技术来提供错误检查及恢复。这种编码技术需要多个磁盘存放检查及恢复信息，使得 RAID 2 技术实施更复杂，因此，在商业环境中很少使用。
- RAID 3：不同于 RAID 2，RAID 3 使用单块磁盘存放奇偶校验信息。如果一块磁盘失效，奇偶盘及其他数据盘可以重新产生数据。如果奇偶盘失效，则不影响数据使用。RAID 3 对于大量的连续数据可提供很好的传输率，但对于随机数据，奇偶盘会成为写操作的瓶颈。
- RAID 4：同 RAID 2 和 RAID 3 一样，RAID 4 和 RAID 5 也同样将数据条块化并分布于不同的磁盘上，但条块单位为块或记录。RAID 4 使用一块磁盘作为奇偶校验盘，每次写操作都需要访问奇偶盘，成为写操作的瓶颈，在商业应用中很少使用。
- RAID 5：RAID 5 没有单独指定的奇偶盘，而是交叉地存取数据及奇偶校验信息于所有磁盘上。在 RAID 5 上，读/写指针可同时对阵列设备进行操作，提供了更高的数据流量。RAID 5 更适合于小数据块、随机读写的数据。RAID 3 与 RAID 5 相比，重要的区别在于 RAID 3 每进行一次数据传输，需涉及到所有的阵列盘。而对于 RAID 5 来说，大部分数据传输只对一块磁盘操作，可进行并行操作。在 RAID 5 中有“写损失”，即每一次写操作，将产生 4 个实际的读/写操作，其中两次读旧的数据及奇偶信息，两次写新的数据及奇偶信息。
- RAID 6：RAID 6 与 RAID 5 相比，增加了第二个独立的奇偶校验信息块。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高。即使两块磁盘同时失效，也不会影响数据的使用。但需要分配给奇偶校验信息更大的磁盘空间，相对于 RAID 5 有更大的“写损失”。RAID 6 的写性能非常差，较差的性能和复杂的实施使得 RAID 6 很少使用。
- SAN（Storage Area Network，存储局域网）：是独立于服务器网络系统之外几乎拥有无限存储能力的高速存储网络，这种网络采用高速的光纤通道作为传输媒体，以 FC（Fiber Channel，光通道）+ SCSI（Small Computer System Interface，小型计算机系统接口）的应用协议作为存储访问协议，将存储子系统网络化，实现了真正高速共享存储的目标。一个完整的 SAN 包括：支持

SAN 的主机设备、支持 SAN 的储存设备、用于连接 SAN 的连接设备、支持 SAN 的管理软件和支持 SAN 的服务。

- NAS（Network Attached Storage，网络附加存储设备）：是一种专业的网络文件存储及文件备份设备，或称为网络直联存储设备、网络磁盘阵列。NAS 是基于 LAN 的，按照 TCP/IP 协议进行通信，面向消息传递，以文件的 I/O 方式进行数据传输。在 LAN 环境下，NAS 已经完全可以实现异构平台之间的数据级共享，比如 NT、UNIX 等平台的共享。一个 NAS 包括处理器、文件服务管理模块和多个的硬盘驱动器用于数据的存储。NAS 可以应用在任何网络环境当中。主服务器和客户端可以非常方便地在 NAS 上存取任意格式的文件，包括 SMB 格式（Windows）、NFS 格式（UNIX、Linux）和 CIFS 格式等。NAS 系统可以根据服务器或者客户端计算机发出的指令完成对内在文件的管理。NAS 是在 RAID 的基础上增加了存储操作系统，因此，NAS 的数据能由异类平台共享。

因此，利用 RAID、SAN、NAS 的技术在多个物理磁盘之间平衡数据库的 I/O，尽量避免数据库产生 I/O 竞争的瓶颈。

5. 手工分配数据文件到不同的文件系统、控制器和物理设备来重新调整数据库 I/O

如果数据库目前的存储设备不算太好，那么采用这种方法是一个不错的选择。这样可以让所有的磁盘得到充分的利用，不至于出现某些磁盘的 I/O 过于太高，而某些磁盘就根本没有被使用的情况，使得在配置较低的情况下得到一个比较好的数据库性能。

需要注意的一点是对于大部分数据库来说，一些 I/O 是一直会存在的。如果上述的方法都尝试过但是数据库的 I/O 性能还是没有达到预定的要求，可以尝试删除数据库中一些不用的旧数据或者使用性能更好的硬件设施。

10.3　Oracle 数据库 I/O 相关的等待事件和相应的解决方法

下面总结了在 Oracle 数据库中最经常出现的一些 I/O 相关的等待事件。

数据文件 I/O 相关的等待事件包括以下：

- db file sequential read
- db file scattered read
- db file parallel read
- direct path read
- direct path write
- direct path read (lob)
- direct path write (lob)

控制文件 I/O 相关的等待事件包括以下：

- control file parallel write
- control file sequential read
- control file single write

重做日志文件 I/O 相关的等待事件包括以下：

- log file parallel write

- log file sync
- log file sequential read
- log file single write
- switch logfile command
- log file switch completion
- log file switch (clearing log file)
- log file switch (checkpoint incomplete)
- log switch/archive
- log file switch (archiving needed)

高速缓存区 I/O 相关的等待事件包括以下：

- db file parallel write
- db file single write
- write complete waits
- free buffer waits

下面来对这些 I/O 相关的等待事件进行具体的说明并提供相应的处理方法。

10.3.1 数据文件相关的 I/O 等待事件

1. db file sequential read 等待事件

这个是非常常见的 I/O 相关的等待事件。在大多数的情况下读取一个索引数据的 block 或者通过索引读取数据的一个 block 的时候都会去要读取相应的数据文件头的 block。在早期的版本中会从磁盘中的排序段读取多个 block 到高速缓存区的连续的缓存中。

在 v$session_wait 这个视图里面，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 ABSOLUTE 文件号，P2 代表 Oracle 从这个文件中开始读取的 block 号，P3 代表 Oracle 从这个文件开始读取的 block 号后读取的 block 数量，通常这个值为 1，表明是单个 block 被读取，如果这个值大于 1，则是读取了多个 block，这种多 block 读取常常出现在早期的 Oracle 版本中从临时段中读取数据的时候。

如果这个等待事件在整个等待时间中占主要的部分，可以采用以下的几种方法来调整数据库。

方法一：从 Statpack 报告中的 “ SQL ordered by Reads ” 部分或者从 v$sql 视图中找出读取物理磁盘 I/O 最多的几个 SQL 语句，优化这些 SQL 语句以减少对 I/O 的读取需求。

如果有 Index Range Scans，但是却使用了不该用的索引，就会导致访问更多的 block，这个时候应该强迫使用一个可选择的索引，使访问同样的数据尽可能少地访问索引块，减少物理 I/O 的读取 ;如果索引的碎片比较多，那么每个 block 存储的索引数据就比较少，这样需要访问的 block 就多，这个时候一般来说最好把索引 rebuild，减少索引的碎片；如果被使用的索引存在一个很大的 Clustering Factor，那么对于每个索引 block 获取相应的记录时就要访问更多表的 block，这个时候可以使用特殊的索引列排序来重建表的所有记录，这样可以大大地减少 Clustering Factor，例如，一个表有 A、B、C、D 和 E 5 个列，索引建立在 A 和 C 上，这样可以使用以下语句来重建表：

```
CREATE TABLE TABLE_NAME AS SELECT * FROM old ORDER BY A,C;
```

此外，还可以通过使用分区索引来减少索引 block 和表 block 的读取。

方法二：如果不存在有问题的执行计划导致读取过多的物理 I/O 的特殊 SQL 语句，那么可能存在以下的情况：

数据文件所在的磁盘存在大量的活动，导致其 I/O 性能很差。这种情况下可以通过查看 Statpack 报告中的“ File I/O Statistics ”部分或者 v$filestat 视图找出热点的磁盘，然后将在这些磁盘上的数据文件移动到那些使用了条带集、RAID 等能实现 I/O 负载均衡的磁盘上去。

使用如下的查询语句可以得到各个数据文件的 I/O 分布：

```
    select d.name name, f.phyrds, f.phyblkrd, f.phywrts, f.phyblkwrt, f.readtim, f.writetim from
v$filestat f, v$datafile d where f.file# = d.file# order by f.phyrds desc, f.phywrts desc;
```

从 Oracle 9.2.0 开始，可以从 v$segment_statistics 视图中找出物理读取最多的索引段或者是表段，通过查看这些数据，可以清楚详细地看到这些段是否可以使用重建或者分区的方法来减少所使用的 I/O。如果 Statpack 设置的 level 为 7 就会在报告中产生“ Segment Statistics ”的信息。

```
    SQL> select distinct statistic_name from v$segment_statistics;
    STATISTIC_NAME
    ----------------------------------------------------------------
    ITL waits
    buffer busy waits
    db block changes
    global cache cr blocks served
    global cache current blocks served
    logical reads
    physical reads
    physical reads direct
    physical writes
    physical writes direct
    row lock waits
    11 rows selected.
```

从上面的查询可以看到相应的统计名称，使用下面的查询语句就能得到读取物理 I/O 最多的段：

```
    select object_name,object_type,statistic_name,value
    from v$segment_statistics
    where statistic_name='physical reads'
    order by value desc;
```

方法三：如果不存在有问题的执行计划导致读取过多的物理 I/O 的特殊 SQL 语句，磁盘的 I/O 也分布得很均匀，这种时候可以考虑增大高速缓存区。对于 Oracle 8i 来说可以增大初始化参数 DB_BLOCK_BUFFERS，让 Statpack 中的 Buffer Cache 的命中率达到一个满意值。对于 Oracle 9i 来说则可以使用 Buffer Cache Advisory 工具来调整 Buffer Cache；对于热点的段可以使用多缓冲池，将热点的索引和表放入到 KEEP Buffer Pool 中去，尽量让其在缓冲中被读取，减少 I/O。

2. db file scattered read 等待事件

这也是一个非常常见的等待事件。当 Oracle 从磁盘上读取多个 block 到不连续的高速缓存区的缓存中就会发生这个等待事件，Oracle 一次能够读取的最多的 block 数量是由初始化参数 DB_FILE_MULTIBLOCK_READ_COUNT 来决定，这个等待事件一般伴随着全表扫描或者 Fast Full Index 扫描一起出现。

在 v$session_wait 这个视图里面，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 ABSOLUTE 文件号，P2 代表 Oracle 从这个文件中开始读取的 block 号，P3 代表 Oracle 从这个文件开始读取的 block 号后读取的 block 数量。

如果这个等待事件在整个等待时间中占了比较大的比重，可以以下的几种方法来调整 Oracle 数据库。

方法一：找出执行全表扫描或者 Fast Full Index 扫描的 SQL 语句，判断这些扫描是否是必要的，是否导致了比较差的执行计划，如果是，则需要调整这些 SQL 语句。

从 Oracle 9i 开始提供了一个视图 v$sql_plan，可以很快地帮助我们找到那些全表扫描或者 Fast Full Index 扫描的 SQL 语句，这个视图会自动忽略关于数据字典的 SQL 语句。

查找全表扫描的 SQL 语句可以使用以下语句：

```
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 语句可以使用以下语句：

```
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;
```

如果是 Oracle 8i 的数据库，可以从 v$session_event 视图中找到关于这个等待事件的进程 SID，然后根据 SID 来跟踪相应的会话的 SQL。

```
select sid,event from v$session_event where event='db file sequential read'
```

或者可以查看物理读取最多的 SQL 语句的执行计划，看是否里面包含了全表扫描和 Fast Full Index 扫描。可以通过以下语句来查找物理读取最多的 SQL 语句：

```
select sql_text from (
select * from v$sqlarea
order by disk_reads)
where rownum<=10;
```

方法二：有时候在执行计划很好的情况下也会出现多 block 扫描的情况，这时可以通过调整 Oracle 数据库的多 block 的 I/O，来设置一个合理的 Oracle 初始化参数 DB_FILE_MULTIblock_READ_COUNT，使得尽量满足以下的公式：

```
DB_BLOCK_SIZE × DB_FILE_MULTIBLOCK_READ_COUNT = max_io_size of system
```

DB_FILE_MULTIBLOCK_READ_COUNT 是指在全表扫描中一次能够读取的最多的 block 数量，这个值受操作系统每次能够读写最大的 I/O 限制，如果设置的值按照上面的公式计算超过了操作系统每次的最大读写能力，则会默认为 max_io_size/db_block_size。例如 DB_FILE_MULTIBLOCK_READ_COUNT 设置为 32，DB_BLOCK_SIZE 为 8KB，这样每次全表扫描的时候能读取 256KB 的表数据，从而大大地提高了整体查询的性能。设置这个参数也不是越大越好的，设置这个参数之前应该要先了解应用的类型，如果是 OLTP 类型的应用，一般来说全表扫描较少，这个时候设定比较大的 DB_FILE_MULTIBLOCK_READ_COUNT 反而会降低 Oracle 数据库的性能，因此 CBO 在某些情况下会因为多 block 读取导致 COST 比较低从而错误地选用全表扫描。

此外，还可以通过对表和索引使用分区、将缓存区的 LRU 末端的全表扫描和 Fast Full Index 扫描的 block 放入到 KEEP 缓存池中等方法来调整这个等待事件。

3. db file parallel read 等待事件

当 Oracle 从多个数据文件中并行读取多个 block 到内存的不连续缓冲中（高速缓存区或者是 PGA）时可能就会出现这个等待事件。这种并行读取一般出现在恢复操作中或者是从缓冲中预取数据达到最优化（而不是多次从单个 block 中读取）。这个事件表明会话正在并行执行多个读取的需求。

在 v$session_wait 这个视图里面，这个等待事件有 3 个参数 P1、P2、P3，其中 P1 代表有多少个文件被读取所请求，P2 代表总共有多少个 block 被请求，P3 代表总共有多少次请求。

如果在等待时间中这个等待事件占的比重比较大，可以按照处理 db file sequential read 等待事件的方法来处理这个事件。

4. direct path read 等待事件

这个等待事件一般出现在 Oracle 将数据直接读入到 PGA 内存中（而不是高速缓存区），如果系统使用了异步 I/O，那么 Oracle 可以一边提交请求一边同时继续处理请求，这样能加速 I/O 请求的结果并会出现 direct path read 等待直到请求 I/O 完成。如果没有使用异步 I/O，I/O 请求会被阻塞直到之前的 I/O 请求完成后，但是此时不会出现 I/O 等待，会话稍后重新恢复并加速 I/O 请求的完成，此时就会出现 direct path read 等待。因此，对于这个等待事件容易产生两方面的误解：一是认为等待的总的数量不能反映出 I/O 请求的数量，二是消耗在这个等待事件上的总的时间不能反映出实际的等待时间。这类型的读取请求主要是用于不在内存中排序的 I/O、并行查询以及预读操作（提前请求一个进程即将使用的 block）。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表等待 I/O 读取请求的文件的 ABSOLUTE 文件号，P2 代表等待 I/O 读取请求的第一个 block 号，P3 代表总共有多少个连续的 block 被请求读取。

这个等待事件的等待时间是指等待 block 直到显著的 I/O 请求完成的时间。值得注意的是对于异步 I/O 来说等待时间并不是 I/O 本身所耗费的时间，因为此时等待并没有开始，而且在这个等待事件中 Oracle 并不会出现超时的现象。在 DSS 类型的系统中，执行大量批处理操作的过程中出现这个等待事件属于很正常的现象，然而如果在 OLTP 类型的系统中大量出现这个等待事件则是表明 Oracle 数据库存在着问题，需要调整。

如果在等待时间中这个等待事件占的比重比较大，那么可以从以下几个方面来调整：

- 如果是等待的文件是临时表空间的数据文件，那么需要查看是否存在大量不合理的磁盘排序，优化相应的存在问题的 SQL 语句。如果是 Oracle 9i 可以考虑使用自动 SQL 执行内存管理，Oracle 8i 的话可以手工地调整各种排序区。
- 尽量减少 I/O 请求的次数，通过设置初始化参数 DB_FILE_DIRECT_IO_COUNT，使得满足：

```
DB_BLOCK_SIZE × DB_FILE_DIRECT_IO_COUNT = max_io_size of system
```

在 Oracle 8i 中默认这个值为 64 个 block；在 Oracle 9i 中可以设置隐含参数 _DB_FILE_DIRECT_IO_COUNT，参数的值也变成了 bytes 而不是 block 数量了，默认值也变成了 1MB。

- 确认异步 I/O 是否配置正确，使用异步 I/O 不会减少这个等待事件的等待时间但是却可以减少会话所消耗的时间。
- 检查是否存在 I/O 消耗很严重的 SQL 语句，如果存在，尝试优化 SQL 语句减少 I/O 的消耗。

- 最后确认一下是否达到了磁盘的 I/O 极限，如果是，则需要考虑更换性能更好的硬件设备。

5. direct path write 等待事件

direct path write 是允许一个会话让一个 I/O 写请求入队列的同时处理操作系统的 I/O。如果会话想确认明显的写是否已经完成就会出现这个等待事件。因为会话需要空的缓存和空的槽位（等待之前的 I/O 释放），或者是会话需要确认所有的写操作都已经完成。如果没有使用异步 I/O，I/O 请求会被阻塞直到之前的 I/O 请求完成后，但是此时不会出现 I/O 等待，会话稍后重新恢复并加速 I/O 请求的完成，此时就会出现 direct path write 等待。因此，对于这个等待事件容易产生两方面的误解：一是认为等待的总的数量不能反映出 I/O 请求的数量，二是消耗在这个等待事件上的总的时间不能反映出实际的等待时间。这类型的写请求主要是用于直接装载数据的操作（create table as select）、并行的 DML 操作、不在内存中排序的 I/O 以及写入没有 cache 的 LOB 段操作。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表等待 I/O 读取请求的文件的 ABSOLUTE 文件号，P2 代表等待 I/O 读取请求的第一个 block 号，P3 代表总共有多少个连续的 block 被请求读取。

这个等待事件的等待时间是指等待 block 直到明显的 I/O 请求完成的时间。通常来说，如果不是存在特殊的 JOB，一般是不会出现这个等待事件，如果在等待时间中这个等待事件占的比重比较大，那么可以从以下几个方面来调整：

- 如果是等待的文件是临时表空间的文件，那么需要查看是否存在大量不合理的磁盘排序，优化相应的存在问题的 SQL 语句。如果是 Oracle 9i 可以考虑使用自动 SQL 执行内存管理，Oracle 8i 的话可以手工地调整各种排序区。
- 尽量减少 I/O 请求的次数，通过设置初始化参数 DB_FILE_DIRECT_IO_COUNT，使得满足：

```
DB_BLOCK_SIZE × DB_FILE_DIRECT_IO_COUNT = max_io_size of system
```

在 Oracle 8i 中默认这个值为 64 个 block；在 Oracle 9i 中可以设置隐含参数 _DB_FILE_DIRECT_IO_COUNT，参数的值也变成了 bytes 而不是 block 数量了，默认值也变成了 1MB。

- 确认异步 I/O 是否配置正确，异步 I/O 不会减少这个等待事件的等待时间但是却可以减少会话所消耗的时间。
- 检查是否存在 I/O 消耗很严重的 SQL 语句，如果存在，尝试优化 SQL 语句减少 I/O 的消耗。
- 最后确认一下是否达到了磁盘的 I/O 极限，如果是，则需要考虑更换更好的硬件设备。

6. direct path read/write (LOB) 等待事件

这个等待事件是从 Oracle 8.1.7 开始出现的，表明在等待直接路径读取访问一个 LOB 对象。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表等待 I/O 读取请求的文件的 ABSOLUTE 文件号，P2 代表等待 I/O 读取请求的第一个数据 block 地址，P3 代表总共有多少个连续的 block 被请求读取。

对于那些没有 cache 的 LOB 对象，强烈建议将其所在的数据文件放置在存在缓存的磁盘上（如文件系统），这样使得直接读取操作能够受益于那些非 Oracle 的 cache，加快读取的速度。

10.3.2　控制文件相关 I/O 等待事件

1. control file parallel write 等待事件

这个等待事件表明服务器进程在更新所有的控制文件时等待 I/O 的完成。因为控制文件所在磁盘的 I/O 过高引起无法完成对所有控制文件的物理写入，写入控制文件的这个会话会拥有 CF 队列，因此其他的会话都会在这个队列中等待。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，这 3 个参数都设置为同样的值，代表控制文件对 I/O 的请求数量。当 Oracle 更新控制文件时是同时更新所有控制文件并写入同样的信息。

如果在等待时间中这个等待事件占的比重比较大，可以从以下几个方面来调整：

- 在确保控制文件不会同时都丢失的前提下，将控制文件的数量减小到最少。
- 如果系统支持异步 I/O，则推荐尽量使用异步 I/O，这样可以实现真正并行地写入控制文件。
- 将控制文件移动到负载比较低，速度比较快的磁盘上去。

2. control file sequential read 等待事件

读取控制文件的时候遇到 I/O 等待就会出现这个等待事件，例如备份控制文件的时候、读取 block 头部都会引起这个等待事件，等待的时间就是消耗在读取控制文件上的时间。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表正在读取的控制文件号，通过下面的 SQL 语句可以知道究竟具体是哪个控制文被读取：

```
SELECT * FROM X$KCCCF WHERE INDX = <file#>;
```

P2 代表开始读取的控制文件 block 号，它的 block 大小和操作系统的 block 大小一样，通常来说是 512KB，也有些 UNIX 的是 1MB 或者 2MB，P3 代表会话要读取 block 的数量。一般来说使用参数 P1 和 P2 来查询 block，当然也可以包括参数 P3，但是那样最终就变成了一个多 block 读取，因此一般都忽略参数 P3。

如果这个等待事件等待的时间比较长，则需要检查控制文件所在的磁盘是否很繁忙，如果是，将控制文件移动到负载比较低、速度比较快的磁盘上去。如果系统支持异步 I/O，则启用异步 I/O。对于并行服务器来说，如果这种等待比较多，会造成整个数据库性能下降，因为并行服务器之间的一些同步是通过控制文件来实现的。

3. control file single write 等待事件

这个等待事件出现在写控制文件的共享信息到磁盘的时候，这是个自动操作，并且通过一个实例来保护，如果是并行的数据库服务器，那么对于并行服务器来说也只能有一个实例能够执行这个操作。这个事件的等待事件就是写操作所消耗的时间。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表正在读取的控制文件号，通过下面的 SQL 语句可以知道究竟是具体是哪个控制文被读取：

```
SELECT * FROM X$KCCCF WHERE INDX = <file#>;
```

P2 代表开始读取的控制文件 block 号，它的 block 大小和操作系统的 block 大小一样，通常

来说是 512KB，也有些 UNIX 的是 1MB 或者 2MB，P3 代表会话要读取 block 的数量。一般来说使用参数 P1 和 P2 来查询 block，当然也可以包括参数 P3，但是那样最终就变成了一个多 block 读取，因此一般都忽略参数 P3。

尽管这个事件是 single write，事实上也会出现多 block 写的情况，即 P3>1。使用参数 P1 和 P2 来查询检测 block 而不用去考虑 P3 的值。

如果这个等待事件等待的时间比较长，则需要检查控制文件所在的磁盘是否很繁忙，如果是，将控制文件移动到负载比较低、速度比较快的磁盘上去。如果系统支持异步 I/O，则启用异步 I/O。对于并行服务器来说，如果这种等待比较多，会造成整个数据库性能下降，因为并行服务器之间的一些同步是通过控制文件来实现的。

10.3.3 重做日志文件相关的等待事件

1. log file parallel write 等待事件

这个等待事件出现在当 LGWR 后台进程从日志缓冲区写日志信息到磁盘上的重做日志文件的时候。只有启用了异步 I/O 的时候 LGWR 进程才会并行写当前日志组内的重做日志文件，否则 LGWR 只会循环顺序逐个地写当前日志组重做日志文件。LGWR 进程不得不等待当前日志组所有的重做日志文件成员全部写完，因此，决定这个等待事件的等待时间长短的主要因素是重做日志文件所在磁盘的 I/O 读写速度。

如果是当前 LGWR 进程写的速度不够快而导致了这个等待事件，可以通过查看一些和重做日志相关的统计值来判定当前的 LGWR 进程是否效率很低，具体的可以查看 redo writes、redo blocks written、redo write time、redo wastage、redo size 统计值，这些都是和 LGWR 进程性能直接相关的一些统计值。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表正在被写入的重做日志文件组中的重做日志文件号，P2 代表需要写入重做日志组中每个重做日志文件的重做日志 block 数量，P3 代表 I/O 请求的次数，需要被写入的 block 会被分成多次分别请求。

如果这个等待事件占用的等待时间比较多，可以从以下几个方面来进行调整：

- 对能使用 UNRECOVERABLE/NOLOGGING 的操作尽量使用这两个选项来减少重做日志的产生。
- 在保证不会同时丢失重做日志文件的前提下尽量减少重做日志组中的成员的个数，减少每次写重做日志组文件的时间。
- 除非在备份的情况下，否则不要在将表空间置于热备的模式下，因为表空间处于热备的模式下会产生更多的重做日志文件。
- 对于使用 LogMiner、Logical Standby 或者 Streams，在能够满足要求功能的前提下，尽量使用最低级别的追加日志以减少重做日志的产生。
- 尽量将同一个日志组内的重做日志文件分散到不同的硬盘上，减少并行写重做日志文件时产生的 I/O 竞争。
- 不要将重做日志文件放置在 RAID 5 的磁盘上，最好使用裸设备来存放重做日志文件。
- 如果设置了归档模式，不要将归档日志的目的地设置为存放重做日志的磁盘上，避免引起 I/O 竞争。

2. log file sync 等待事件

这个等待事件是指等待 Oracle 的前台的 commit 和 rollback 操作进程完成，有时候这个等待事件也会包括等待 LGWR 进程把一个会话事务的日志记录信息从日志缓冲区中写入到磁盘上的重做日志文件中。因此，当前台进程在等待这个事件的时候，LGWR 进程同时也在等待事件 log file parallel write。理解什么造成这个等待事件的关键在于对比这个等待事件和 log file parallel write 等待事件的平均等待时间：如果它们的等待时间差不多，那么就是重做日志文件的 I/O 引起了这个等待事件，则需要调整重做日志文件的 I/O，这个在之后会有详细的讲述。如果 log file parallel write 等待事件的平均等待时间明显小于 log file sync 等待事件的等待时间，那么就是一些其他写日志的机制在 commit 和 rollback 操作时引起了等待，而不是 I/O 引起的等待，例如重做日志文件的 latch 的竞争，会伴随着出现 latch free 或者 LGWR wait for redo copy 等待事件。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表在日志缓冲区中需要被写入到重做日志文件中的缓存的数量，写入的同时会确认事务是否已经被提交，并且保留提交信息到实例意外中断之前，因此必须等待 LGWR 将 P1 数量的缓存写入重做日志文件为止。P2 和 P3 属于无用的参数。

如果这个等待事件在整个等待时间中占了比较大的比重，可以从以下三个方面来调整这个等待事件：

- 调整 LGWR 进程使其具有更好的磁盘 I/O 吞吐量，例如不要将日志文件放置在 RAID 5 的磁盘上。
- 如果存在很多执行时间很短的事务，可以考虑将这些事务集合成一个批处理事务以减少提交的次数，因为每次提交都需要确认相关的日志写入重做日志文件，因此使用批处理事务来减少提交的次数是一种非常行之有效的减少 I/O 的方法。
- 查看是否一些操作可以安全地使用 NOLOGGING 或者 UNRECOVERABLE 选项，这样可以减少日志的产生。

3. log file sequential read 等待事件

这个等待事件是指等待读取重做日志文件中的日志记录，等待的时间就是耗费在完成整个读取日志记录的物理 I/O 操作的时间。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表一个日志组里面所有日志文件的相对 sequence 号，P2 代表日志文件在指定物理块大小的偏移量，P3 代表读取 block 的数量，如果 P3 的值为 1，一般来说都是在读取日志文件头。

4. log file single write 等待事件

这个等待事件是指等待写重做日志文件操作完成，常常是在等待写重做日志文件头，例如在增加一个新的重做日志组成员时，Oracle 数据库就会往这个重做日志文件头写入相应的 sequence 号。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表正在被写入的重做日志文件组的组号，P2 代表日志文件在指定物理块大小的偏移量，P3 代表写入 block 的数量。

因为 single write 通常都是在写或者重写日志文件头的时候出现，因此开始的 block 号总是为

1。一般如果出现这个等待事件，应该对重做日志文件尽量使用裸设备，避免将多个日志文件放在同一个磁盘上，减少产生 I/O 竞争的可能。

5. switch logfile command 等待事件

这个等待事件是指执行日志文件切换命令的时候等待日志文件切换完成，Oracle 数据库会每隔 5 秒钟就检测一次是否超时。

如果出现这个等待事件，表明花费了很长的时间去切换重做日志文件，此时需要检查数据库的告警日志文件以查看 Oracle 后台进程 LGWR 是否正常工作。

6. log file switch completion 等待事件

这个等待事件是指由于当前重做日志文件已经被写满了而 Oracle 后台进程 LGWR 需要完成写完当前重做日志文件并且要打开一个新的重做日志文件而导致的重做日志文件切换的等待，或者是其他请求需要切换重做日志文件导致等待。

如果当前的重做日志写满了，这个时候 Oracle 数据库就需要切换重做日志文件来提供足够的磁盘空间给重做日志写日志缓存。但是由于一些其他的进程也同样可以引起重做日志的切换，Oracle 数据库不会同时去切换重做日志两次，因此，就出现了这个等待事件，在 Oracle 数据库早期的版本中还有 log_file_switch_checkpoint_incomplete、log_file_switch_archiving_needed、log_file_switch_clearing_log_file 的等待事件。

7. log file switch（checkpoint incomplete）等待事件

这个等待事件是指由于当前重做日志的检查点没有及时地完成而导致重做日志文件无法切换到下一个日志文件引起的日志文件切换的等待。

调整这个等待事件的方法一般是加速检查点的完成，可以通过减小 Buffer Cache 缓冲区或者增加更多的 DBWR 进程、调整相关检查点的初始化参数等方法来达到相应的效果。

8. log file switch（archiving needed）等待事件

这个等待事件是指当前的重做日志文件准备切换到下一重做日志文件，但是当前重做日志文件因为没有被归档而导致等待，这个等待事件只出现于采用了归档方式的 Oracle 数据库中。

如果出现这个等待事件，首先应该查看 Oracle 数据库的告警日志文件，看是否因为写归档日志文件错误导致归档进程停止，其次，可以增加归档进程的数量或者将归档日志文件存放到 I/O 速度比较快的磁盘上，还可以通过增大和增加重做日志文件的大小和数量来给予归档更多的时间。

10.3.4 高速缓存区相关的 I/O 等待事件

1. db file parallel write 等待事件

这个等待事件是指 Oracle 后台进程 DBWR 等待一个并行写入文件或者是 block 的完成，等待会一直持续到这个并行写入操作完成。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表 Oracle 正在写入的数据文件的数量，P2 代表操作将会写入多少的 block 数量，P3 在 Oracle 9i release2 版本

之前代表总共有多少 block 的 I/O 请求，等于 P2 的值；在 Oracle 9i release2 版本之后则代表等待 I/O 完成的超时的时间，单位是百分之一秒。

这个等待事件即使在总的等待时间中占的比例比较大也不会对用户的会话产生很大的影响，只有当用户的会话显示存在大量的等待时间消耗在“write complete waits”或者是“free buffer waits”上时才会影响到用户的会话，较明显的影响是这个写操作的等待会影响到读取同一个磁盘上数据的用户会话的 I/O。

2. db file single write 等待事件

这个等待事件通常是表明在等待写入数据到数据文件头。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表 Oracle 正在写入的数据文件的文件号：

```
SELECT * FROM v$datafile WHERE file# = <file#>;
```

P2 代表 Oracle 正在写入的 block 号，如果 block 号不是 1，则可以通过如下查询查出 Oracle 正在写入的对象是什么：

```
SELECT  segment_name ,  segment_type ,
owner , tablespace_name
FROM  sys.dba_extents
WHERE  file_id = <file#>
AND  <block#>
BETWEEN block_id AND block_id + blocks -1;
```

P3 代表 Oracle 写入 file#的数据文件中从 block#开始写入的 block 的数量。

Oracle 数据文件的文件头一般来说都是 block1，操作系统指定的文件头是 block0，如果 block 号大于 1，则表明 Oracle 正在写入的是一个对象而不是文件头。

3. write complete waits 等待事件

这个等待事件表明 Oracle 的会话在等待写入缓存，一般都是缓存的正常老化或者是实例之间的互相调用引起的。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表 Oracle 正在写入的数据文件的文件号，P2 代表 Oracle 正在写入的 block 号，可以通过如下查询查出 Oracle 正在写入的对象是什么：

```
SELECT  segment_name ,  segment_type ,owner , tablespace_name
FROM  sys.dba_extents
WHERE  file_id = <file#>
AND  <block#> BETWEEN block_id AND block_id + blocks -1;
```

P3 代表产生这个等待事件原因的 id 号，具体的 id 号所代表的原因如下：

- 1022 无。
- 1027 在写入过程中的 buffer，最多等待 1 秒后会重新扫描 cache。
- 1029 试图应用改变到正在写入中的 block，会一直等到 block 可用，每次等待的时间都是 1 秒。
- 1030 无。
- 1031 无。
- 1033 无。

- 1034 实例间的交叉写：一个实例企图去修改一个 block，而另外一个实例想去获得此 block 的状态，这个实例产生最多 1 秒钟的等待。
- 1035 与 1034 一样，但是产生的原因是由于数据库出于热备的状态下。

当 Oracle 的后台进程 DBWR 获取可以写入的缓存并标记这些缓存为正在写入的状态，接着这些被收集的缓存中的数据将会被写入磁盘上的数据文件中，当所有的 I/O 完成后将清除在原来那些被标记的缓存上的标记，这个等待事件出现意味着 Oracle 想获取的 buffer 已经被标记为正在写入的状态，只有等标记被清除才能获取到相应的 buffer。在 Oracle 7.2 以前的版本中，只有当批处理中所有的 buffer 都被写入磁盘后标记才被清除，在这之后的版本，每个 buffer 写入磁盘后就将清除在这个 buffer 上的标记。

增加更多的 Oracle 后台 DBWR 进程或者是采用异步 I/O 都将能减少这个等待事件的产生。

4. free buffer waits 等待事件

这个等待事件出现的原因比较多，大致可以分为以下几种：

- 当一个数据文件从只读状态变成为可读写状态的时候，所有的 buffer gets 全部都被挂起了，就可能出现这个等待事件。已经存在的所有 buffer 都必须要失效，因为它们没有链接到 lock elements（OPS/RAC 环境下时需要）。因此，只有当这些 buffers 失效完成后才能够被分配给数据库块地址。
- 当 Oracle 数据库需要从系统全局区（SGA）中读取一个 buffer 给一致性读（CR）操作，只读操作或者是用于任何恢复模式中的操作时，也可能出现这个等待事件，此时可以加速 Oracle 后台的 DBWR 进程来获得较多的空闲 buffer。
- 在检查了“ free buffers inspected ”之后也会出现这个等待事件，如果没有找到空闲的 buffer，Oracle 会等待 1 秒钟后继续试图去获取空闲的 buffer。

在 v$session_wait 视图中，这个等待事件有 3 个参数 P1、P2 和 P3，其中 P1 代表 Oracle 读取 buffer 而引起等待的数据文件的文件号，P2 代表数据文件中读取 buffer 的 block 的号，P3 代表要读取 buffer 的缓存中的 block（7.3.X 以上的版本）。

一般来说，这个等待事件都是由于 Oracle 的后台进程 DBWR 不能及时地将 buffer 写完到磁盘上的数据文件中而引起的，尽量将 I/O 平均分配到各个磁盘上，减少出现某个磁盘上 I/O 负载很高而引起 DBWR 进程写入慢的情况，可以通过操作系统上的 I/O 监控工具或者查询 v$filestat 视图来获取相应的数据：

```
SELECT name, phyrds, phywrts
FROM v$filestat a, v$datafile b
WHERE a.file# = b.file#;
```

还可以通过查看数据文件上是否存在全表扫描来判断：

```
SELECT name, phyrds, phyblkrd, phywrts
FROM v$filestat a, v$datafile b
WHERE a.file# = b.file#
and phyrds!= phyblkrd;
```

需要注意在应用中要避免漏建了索引，否则会引起 I/O 大幅度的增加，导致不必要的磁盘扫描，如果有多块硬盘来存储 Oracle 的数据文件，尽量使用操作系统的条带化软件来分布 Oracle 的数据文件使得 I/O 分配均匀。此外，大量的磁盘排序会导致存在很多的脏缓存需要写完，因此，临时表空间中的数据文件最好能分配到不同的磁盘上，避免同一个磁盘上的 I/O 竞争。还有如果

排序的 block 的检查点没有完成，将会存在于正常的缓存写批处理中，如果缓存写批处理中全部都被排序块给占满了，那其他的脏数据块就没法被写入，而导致前台的应用不得不等待分配空闲的 buffer。对于 Oracle 9i 之后的版本，因为排序使用的块通常都是来自临时表空间文件，不会进入到缓存中，因此，由于大量排序引起的这种等待在 9i 中基本上就不会存在了。

了解了在 Oracle 数据库 I/O 性能或者是响应时间低下时该如何去调整和优化数据库之后，还有一点很重要的需要注意的是，无论是何种情况，都应该先检查操作系统上的日志文件，因为如果是本身在操作系统级别上出现了 I/O 问题，那不管如何调整 Oracle 数据库都是徒劳的，所以必须首先要保证在操作系统级别上 I/O 不存在问题，然后再去 Oracle 数据库中具体地检查问题产生的原因。

10.4　小结

不管用何种方法去解决 Oracle 数据库的 I/O 性能问题，关键都是先找出产生 I/O 性能问题的根本原因，然后想各种各样的办法去解决产生的问题就可以达到优化数据库的目的了。以上所谈到的都是关于 Oracle 数据库 I/O 调整优化的一些基本概念和方法，希望能起到一个抛砖引玉的作用，以便大家能够更好地深入理解 Oracle 数据库 I/O 性能方面的知识。

作者简介

叶梁，网名 cooly1，现任 ITPUB Oracle 管理版版主。

曾任职于国内某大型软件企业做 Oracle 数据库的技术支持，客户遍及全国各个行业，尤其是电信、政府、金融行业。现任职于某外资电信企业华北区分公司，从事 DBA 工作，负责华北区 40 多个数据库系统的维护，对大型数据库管理经验丰富。

擅长数据库的维护，对于数据库的安装，调整，备份方面有自己独到的经验。同时也给一些国内的大型企业做过 Oracle 的培训，有一定的培训经验。

曾做过很多大型项目的数据库维护和支持工作，对 Oracle 的维护有相当多的实际经验，善于现场解决问题。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

第 11 章　Oracle 在 Solaris 的 VxFS 上的异步 I/O 问题

11.1　VxFS 文件系统的简介

VxFS 文件系统是 Veritas 公司推出的一种高性能、高可用性的文件系统，一般用在数据中心。它是一种基于扩展的文件系统，能够让应用程序读取和写入大的连续块，适用于 OLTP 系统和 DSS 系统。

Oracle 数据库在 Solaris 操作系统上的 VxFS 文件系统中是可以实现异步 I/O 的，那 Oracle 数据库在 VxFS 文件系统中究竟该不该使用异步 I/O？如何去判断 Oracle 数据库是否真正实现了异步 I/O？下面就这几个问题来具体地看看 Oracle 数据库在 VxFS 文件系统上的异步 I/O。

11.2　VxFS 文件系统上如何启用异步 I/O

首先要知道 Solaris 操作系统上哪些磁盘上的文件系统是 vxfs 格式的，可以使用以下的命令来进行查看：

```
Df -F vxfs
/opt/oracle/db02   (/dev/vx/dsk/ipasdg/db02_vol):55665072 blocks   869766 files
/opt/oracle/db03   (/dev/vx/dsk/ipasdg/db03_vol):41688928 blocks   651380 files
/opt/oracle/db04   (/dev/vx/dsk/ipasdg/db04_vol):41688928 blocks   651380 files
/opt/oracle/arch   (/dev/vx/dsk/ipasdg/arch_vol):164632064 blocks  2572348 files
/backup           (/dev/vx/dsk/ipasdg/backup_vol):314529872 blocks  4914519 files
```

如果想在 VxFS 上面使用异步 I/O，必须首先安装一个叫做 Quick I/O 的模块，并且要启用 Quick I/O，这个模块是需要向 Veritas 公司单独购买 License 的。默认 VxFS 文件系统 mount 的时候是启用 Quick I/O 的，如果在 mount 的时候指定了-o noqio 的选项，那么 Quick I/O 是被禁用的。

想查看一个文件系统上是否采用了 Quick I/O，使用 fsadmin、fstype 这些命令是无法查看的，而且/etc/mnttab、/etc/vfstab 这些文件也没有记录相关的信息。这里介绍一种可以查看文件是否是 Quick I/O 文件的方法。

ls –al 可以列出所有文件，包括 Quick I/O 文件和它的链接。

```
$ ls -al d* .d*
-rw-r--r-- 1 <oracle> dba 104890368 Oct 2 13:42 .dbfile
lrwxrwxrwx 1 <oracle> dba 17 Oct 2 13:42 dbfile -> \ .dbfile::cdev:vxfs:
```

ls –lL 可以显示是否 Quick I/O 被成功安装和启用。

```
$ ls -lL dbfile
crw-r--r-- 1 <oracle> dba 45, 1 Oct 2 13:42 dbfile
```

这里第一个字符 c，表明这是一个裸字符设备文件，如果没有这个字符则表明 Quick I/O 没有正确安装或者是没有一个合法的 License Key。

确认文件系统启用了 Quick I/O 后，就可以给 Oracle 配置异步 I/O 了，在 Oracle 的初始化参数中配置 DISK_ASYNCH_IO = TRUE，然后重启数据库使其生效。

因为启用了 Quick I/O 后，在 OS 级别上是消除了缓冲的，所以在启用了 Quick I/O 后，数据库的 Buffer Cache 是应该需要增加的。

11.3 如何检测在 VxFS 文件系统上是否支持异步 I/O

对于 Solaris 操作系统可以使用下面的一段源代码来检测系统是否支持异步 I/O：

```
/*
* Quick kaio test. Read 1k bytes from a file using async I/O.
* To compile:
* cc -o aio aio.c -laio
* To run:
* aio file_name
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/aio.h>
#define BSIZE 1024
main(int argc, char *argv[])
{
aio_result_t res;
char buf[BSIZE];
int fd;
if ((fd=open(argv[1], O_RDONLY)) == -1) {
perror("open");
exit(-1);
}
aioread(fd, buf, BSIZE, 0L, SEEK_SET, &res);
aiowait(0);
if (res.aio_return == BSIZE) {
printf("aio succeeded\n");
close(fd);
exit(0);
}
perror("aio");
}
```

然后使用 root 用户编译：

```
# cc -o aio aio.c -laio
```

这样就可以用来检测系统是否支持异步 I/O 了。

对于 VxFS 上的裸设备，不需要启用 Quick I/O 就可以直接支持异步 I/O：

```
# truss -t kaio,lwp_create ./aio /dev/rdsk/c0t0d0s1
kaio(5, 0xFFBEF640, 0x00000000, 0xFF21FB68, 0x00000000, 0xFFBEF648, 0x00000000) = 0
lwp_create(0xFFBEF640, 0, 0xFF21FF5C)         = 2
lwp_create()   (returning as new lwp ...)     = 0
kaio(AIOREAD, 3, 0xFFBEF9C0, 1024, 0, 0xFFBEFDC0) = 0
kaio(AIOWAIT, 0x00000000)                  = 4290706880
aio succeeded
```

从上面的测试中可以看出，“ aio succeeded ”表明裸设备上的异步 I/O 操作是成功的，并且异步 I/O 的读写是通过 AIOREAD 和 AIOWRITE 来实现的。

11.4　如何查看 VxFS 文件系统上异步 I/O 的性能

如何查看采用了异步 I/O 的 VxFS 文件系统的性能呢？可以通过 sar 命令进行简单地观察，主要查看%usr、%sys、%wio、%idle 这几列的值。一般可以明显地看到，CPU 消耗在等待 I/O 上的时间比不采用异步 I/O 之前有明显的减少。

先来看没有使用异步 I/O 的情况：

```
          %usr    %sys    %wio   %idle
20:05:13      2      23       1      74
20:05:23      2      24       1      73
20:05:33      2      24       1      73
Response time = 4 min 22 secs .
Oracle stats .
Statistic                                 Total   per Second    per Trans
--------------------------------- ---------------- ------------ ------------
CPU used by this session                 25,188        99.6        25.2
CPU used when call started               25,188        99.6        25.2
```

再来看使用了异步 I/O 的情况：

```
           %usr    %sys    %wio   %idle
19:53:37     17       9       0      74
19:53:42     16       8       2      74
19:53:47     16       7       2      75
19:53:57     17       7       0      75
Response time = 37 secs .
Oracle stats . Statistic                        Total   per Second    per Trans
---------------------------------- ---------------- ------------ ------------
CPU used by this session                        2,119        96.3         2.1
CPU used when call started                      2,119        96.3         2.1
```

从上面的比较数据不难看出，使用了异步 I/O 后的响应时间大大缩短，从原来的 4 分多钟减少到 37 秒，CPU 的使用率也大大降低，Oracle 通过调用 Solaris 上的异步 I/O 库 AIOREAD 和 AIOWRITE 来实现异步 I/O 读写。但是，如果在没有启用 Quick I/O 的 VxFS 文件系统上设置了 Oracle 的异步 I/O，Oracle 的性能不会提高而且会变得极其低下，造成 CPU 的时间绝大多数消耗在 I/O 等待上。当然，如果系统本来就不是很繁忙，I/O 不是很多，CPU 又足够得多，也可能体现不出来

这种问题。这个问题是 Oracle 上的一个 Bug。因此，如果没有使用 Quick I/O 的话，在 VxFS 文件系统上还是不建议使用异步 I/O 的。

做了一个测试，具体如下。

在没有启用 Quick I/O 的 VxFS 文件系统上，设置数据库的初始化参数 DISK_ASYNCH_IO = TRUE，然后让数据库正常地写入数据，在 OS 上跟踪 Oracle 后台的 DBWR 进程，查看其 trace，就可以发现如果没有启用 Quick I/O，即使数据库设置了异步 I/O，也是无法在 OS 级别上实现异步 I/O 的。

```
ps -ef |grep ora_
oracle 16813    1  0   Nov 03 ?       0:02 ora_pmon_ORCL
oracle 16819    1  0   Nov 03 ?       0:53 ora_ckpt_ORCL
oracle 16831    1  0   Nov 03 ?       0:00 ora_d000_ORCL
oracle 16823    1  0   Nov 03 ?       2:31 ora_smon_ORCL
oracle 16815    1  0   Nov 03 ?      52:41 ora_dbw0_ORCL
oracle 16825    1  0   Nov 03 ?       0:01 ora_reco_ORCL
oracle 16827    1  0   Nov 03 ?       0:02 ora_cjq0_ORCL
oracle 16829    1  0   Nov 03 ?       0:00 ora_s000_ORCL
oracle 16817    1  0   Nov 03 ?      96:24 ora_lgwr_ORCL
oracle 16835    1  0   Nov 03 ?       4:11 ora_arc1_ORCL
oracle 16833    1  0   Nov 03 ?       4:14 ora_arc0_ORCL

truss -fl -p 16815
16815/1:       lwp_cond_signal(0xFFFFFFFF7CB8FF70)           = 0
16815/26:    lwp_cond_wait(0xFFFFFFFF7CB8FF70, 0xFFFFFFFF7CB8FF80, 0x00000000) = 0
16815/1:       kaio(AIOWAIT,0xFFFFFFFF7CB7DF70)   Err#22 EINVAL
16815/26:    pread64(408, "1B02\0\0\080\0 2\0\0D09D".., 8192, 102400) = 8192
16815/1:     kaio(AIOWAIT,0xFFFFFFFF7CB7DF70)   Err#22 EINVAL           = 0
16815/26:       kaio(AIONOTIFY, 27977120) = 0
16815/1 :     kaio(AIOREAD, 408, 0x01B98DE0, 2048, 110592, 0x01A8486C) Err#48 ENOTSUP
```

注意到调用 kaio(AIOREAD,…)的时候返回了一个 OS 的错误 errno = 48# ENOTUP，这表明没有启用 Quick I/O 的 VxFS 文件系统是不支持异步 I/O 的。但这并不是一个应用级别的错误，而是 Solaris 异步 I/O 库调用的，产生这个错误表明文件系统并不支持核心的异步 I/O，并且使用了一个同步进程调用 pread 取代了，只是在应用级别上模拟异步 I/O 而已。

11.5 如何转换 VxFS 文件系统上的数据文件为支持异步 I/O 的数据文件

一般来说，要 Oracle 数据库使用具有 Quick I/O 的数据文件，就应该预先分配具有 Quick I/O 特性的数据文件，然后将数据文件加入到相应的 Oracle 数据库表空间中去，可以使用 qiomkfile 命令预先分配具有 Quick I/O 特性的数据文件：

```
/usr/sbin/qiomkfile -s 500M /oradata/test.dbf
```

qiomkfile 这个命令在 Veritas 的 Quick I/O 包中提供，上述命令将会在 VxFS 文件系统的 /oradata 目录下建立两个文件，一个是.test.dbf，另一个是链接文件 test.dbf，其指向.test.dbf，Oracle 数据库中就可以使用这个链接文件 test.dbf 做为表空间的数据文件。

那如何将已有的 VxFS 文件系统上的 Oracle 数据库转换成启用了 Quick I/O 的 Oracle 数据库

呢？Veritas Database Edition for Oracle 提供了两个脚本文件用来转换，一个是 getdbfiles.sh，另一个是 mkqio.sh，两个脚本都存放在/opt/VRTSordba/bin 目录下。前提是在转换之前，数据库的数据文件必须是分布在 VxFS 文件系统上。

getdbfiles.sh 这个脚本是用来从 Oracle 数据库的系统表中得到所有数据文件的名字和位置等相关信息，该脚本必须要用 Oracle 用户来运行，得到的信息存储在一个叫 mkqio.dat 文件中。

mkqio.sh 这个脚本是用来处理 mkqio.dat 文件中包含的所有数据文件，并将它们转换成 Quick I/O 上的数据文件，这个脚本也最好用 Oracle 用户来执行，以避免出现一些权限问题，在运行完 getdbfiles.sh 脚本后，必须要先完全关闭数据库后才能执行 mkqio.sh 脚本。如果转换的时候出现什么问题，就可以使用 mkqio.sh -u 将 Quick I/O 上的文件转换回普通 VxFS 上的文件，需要注意的是这个脚本只能用于转换 VxFS 上的数据库文件，如果数据文件本身并不是建立在 VxFS 文件系统上，那么运行了 getdbfiles.sh 后，必须手工编辑 mkqio.dat 文件去掉那些不是在 VxFS 文件系统上的数据文件。

还有一种简单的方法同样可以把 VxFS 上的数据文件转换为 Quick I/O 的数据文件，通过上文已经知道，建立具有 Quick I/O 的数据文件时会生成一个实际的文件和一个链接文件，于是就可以通过以下步骤来把非 Quick I/O 的数据文件转换为 Quick I/O 的数据文件。首先正常地关闭 Oracle 数据库，然后对那些非 Quick I/O 的数据文件分别执行以下两个命令：

```
mv < datafile> .< datafile>
ln -s .< datafile>::cdev:vxfs: < datafile>
```

这样就将那些非 Quick I/O 的数据文件转换为了具有 Quick I/O 的数据文件。

以上是对 Solaris 上的 VxFS 文件系统上的 Oracle 数据库使用异步 I/O 的初探，当然还可以通过操作系统上对 I/O 分析的一些其他方法进行更加详细深入的研究，因为这些内容不在此篇文章论述范围之内，这里就不详细分析了，有兴趣的朋友可以深入研究一下，从操作系统上异步 I/O 的库文件定义入手，深入发掘操作系统 Solaris 上异步 I/O 的内部机制。

作者简介

叶梁，网名 cooly1，现任 ITPUB Oracle 管理版版主。

曾任职于国内某大型软件企业做 Oracle 数据库的技术支持，客户遍及全国各个行业，尤其是电信、政府、金融行业。现任职于某外资电信企业华北区分公司，从事 DBA 工作，负责华北区 40 多个数据库系统的维护，对大型数据库管理经验丰富。

擅长数据库的维护，对于数据库的安装，调整，备份方面有自己独到的经验。同时也给一些国内的大型企业做过 Oracle 的培训，有一定的培训经验。

曾做过很多大型项目的数据库维护和支持工作，对 Oracle 的维护有相当多的实际经验，善于现场解决问题。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

第12章　关于Freelists和Freelist Groups的研究

本章在于探讨 Freelists 和 Freelist Groups 的作用、存取机制、争用诊断和优化方法，同时通过理论和测试来推翻一些存在了很久的错误观点。本章的读者应该具有较深入的 Oracle 知识，对于一般的开发人员这篇文章可能并没有太多的帮助。

12.1　什么是 Freelists

众所皆知，Oracle 数据库的读取单位是数据块（block），而一个 block 是否允许被写入数据是基于一定的空闲度，这就是大家知道的 PCTFREE 和 PCTUSED 存储参数设置。

假设 PCTFREE=10，PCTUSED=40，这就表明当一个 block 的空间使用率达到了 90%（100-pctfree）时，这个 block 就不再允许被用于新增数据（insert），而保留下来的这 10%空间则被预留为行更新（update）所可能需要的空间扩展，此时这个 block 就从 Freelist 上被摘走了（实际上还有另外一种情况，就是当块剩余空间不足以插入一条记录同时该块的使用率已经超过了 PCTUSED 定义的值并且该块位于 Freelist header 处时，该块也会从 Freelist 上被摘走，术语称为 UNLINK）。当有数据删除（delete）的时候，只有该 block 中的数据被删除到一定的程度，该块才会重新被加入到 Freelists 中，而这个程度就是 PCTUSED 参数定义的数值，如在这个例子中，只有块中的数据降低到 40%以下时，该块才被允许重新用于新增数据。

通过上面的描述，可以知道所谓 Freelists，就是一个指定了所有可以用于 insert 操作的数据块的列表。存在于这个列表中的数据块才能用于 insert 操作，一旦一个数据块无法用于 insert（达到了 PCTFREE 参数指定的限度）则立刻从这个列表中被摘除。Freelists 的作用就在于管理高水位标志（HWM）以下的空闲空间。

注　意

Freelists 只是管理高水位标志以下的空闲空间，而实际上一个 segment 可用的空闲空间包括两种类型：

1. 已经分配给这个 segment 但是从来未被使用过的位于高水位标志之上的 blocks。
2. 位于高水位标志之下，被链接在 Freelists 上的 blocks。

至于 Freelist Groups 的概念和作用，在后续的章节中会进行解释。

12.2 Freelists 是否已经过时

随着 Oracle 9i 的推出，对于空闲块的管理变得更加智能和有效率了。在 LMT（Locally Managed Tablespaces）中如果指定了 ASSM（Automatic Segment Space Management），那么对于任何 PCTUSED、Freelists、Freelist Groups 存储参数的指定都将被忽略。创建 ASSM 表空间的方法如下：

```
CREATE TABLESPACE lmtbsb DATAFILE '/u02/oracle/data/lmtbsb01.dbf' SIZE 50M
   EXTENT MANAGEMENT LOCAL
   SEGMENT SPACE MANAGEMENT AUTO;
```

ASSM 得益于使用位图（Bitmaps）来管理段中的空闲块，至于具体是如何管理的，那又是另外一篇文章了。

就此意义上来说，对于 Freelists 的探讨确实可能已经有些过时了，但是首先并不是所有的数据库现在都已经升级到了 Oracle 9i，甚至在最需要调整的一些大型应用上往往都由于业务的稳定性而不愿意承担升级新版本的风险；其次即使是新的应用使用了 Oracle 9i 数据库，如果数据库管理员在创建表空间的时候没有明确指定 SEGMENT SPACE MANAGEMENT AUTO，那么默认情况下仍然会使用 Freelists 和 Freelist Groups 来管理 free block。

所以，在仍然存在有大量 Oracle 8i 数据库和非自动段空间管理表空间的现在，对于 Freelists 的研究仍然具有很实际的意义，而由于默认的 Freelists 和 Freelist Groups 又都只有 1，所以恰恰是高负载的应用中最需要调整（Tuning）的部分之一。

12.3 Freelists 存储在哪里

Freelists 存储在每个 segment 的 header block 中，可以通过 dump 来得到更详细的信息。dump 在研究 Oracle 的内部机制时通常都扮演着很重要的角色。

假设创建一个表空间 TS_TEST，此表空间是非自动段空间管理的，然后在该表空间中创建 T_MANUAL、T_MANUAL_FREE2 和 T_MANUAL_FREEGROUP2 三张表。这三张表的 Freelists 和 Freelist Groups 设置如下。

```
SQL> select SEGMENT_NAME,SEGMENT_TYPE,FREELISTS,FREELIST_GROUPS from USER_SEGMENTS where
TABLESPACE_NAME='TS_TEST';

SEGMENT_NAME        SEGMENT_TYPE       FREELISTS FREELIST_GROUPS
------------------- ------------------ ---------- ---------------
T_MANUAL             TABLE                     1               1
T_MANUAL_FREE2       TABLE                     2               1
T_MANUAL_FREEGROUP2  TABLE                     4               2
```

可以参照下面的方法对 segment header block 进行 dump 操作。

首先从数据字典中得到存储这个 segment 的文件号和此 segment 的第一个 block 号（也就是 segment header block）。

```
SQL> select FILE_ID,BLOCK_ID from dba_extents where segment_name='T_MANUAL';
```

```
  FILE_ID   BLOCK_ID
---------- ----------
       7         9
```

使用 dump 命令转储这个 block 的内容，转储的结果将保存在初始化参数 user_dump_dest 指定的目录中。

```
SQL> alter system dump datafile 7 block 9;

System altered
```

查看 user_dump_dest 目录中的相应 trace 文件，可以看到包含以下几行：

```
frmt: 0x02 chkval: 0x0000 type: 0x10=DATA SEGMENT HEADER – UNLIMITED
```

表示这个 block 正是 segment header block。

```
#blocks in seg. hdr's Freelists: 2
#blocks below: 2
```

表示位于 Freelist 中的数据块有 2 个，在高水位标志（HWM）下的数据块也有 2 个。

```
SEG LST:: flg: USED   lhd: 0x01c0000a ltl: 0x01c0000b
```

由于 dump 的是 TS_MANUAL 表的 header block，而这张表的 Freelists=1，所以在 dump 文件中看到只有一个 seg lst，这个 Freelist 被称为 Segment Free List 或者 Master Free List，每个 segment 都至少有一个而且只有一个 Master Free List（当然是在非自动段空间管理类型下）。

- flg（flag）表示该 Freelist 是否被使用。
- lhd（list header）表示位于该 list 中的第一个可用 block 的 dba（data block address）。
- ltl（list tail）表示位于该 list 中的最后一个可用 block 的 dba，这个 block 必定位于 HWM 之下。

此时可以发现 Freelists 只是记录了这个 segment 中空闲块的第一个块地址和最后一个块地址，在第一个空闲块的块头处（block header）记录了它之后的下一个空闲块的地址，而下一个空闲块又记录了再下一个空闲块的地址，由此依次记录，一直到最后一个空闲块，如图 12-1 所示。Oracle 通过这种链表的方式实现了 Freelists 对于空闲块的管理。

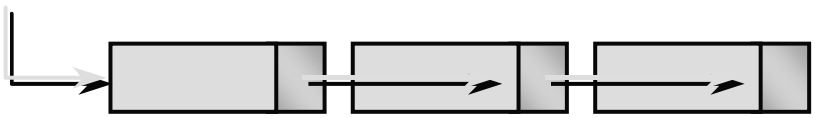

图 12-1　Freelists 链表

注　意

每次当一个 block 被加入到 free list 中时，该 block 会被放置在 free list 的链表头部。

同样可以 dump 第一个空闲块来验证上面的链表说法。

比如在 lhd 部分记录的 dba 是 0x01c0000a，这是一个十六进制的数，首先转化为十进制，于是得到 29360138。然后通过 Oracle 提供的两个函数将块地址转化为可以供使用的文件号和块号，以便于进行 dump 操作。

```
SQL> select dbms_utility.data_block_address_file(29360138) from dual;

DBMS_UTILITY.DATA_BLOCK_ADDRES
------------------------------
```

```
                          7

SQL> select dbms_utility.data_block_address_block(29360138) from dual;

DBMS_UTILITY.DATA_BLOCK_ADDRES
------------------------------
                         10
```

现在已经得到第一个空闲块是 7 号文件的 10 号块。用前面提到的转储命令 dump 这个块的内容，可以找到下面的内容：

```
fnx: 0x1c0000b
```

表示下一个可用的块地址是 0x1c0000b，在这里的例子中，这个块正好是可用的最后一个块（segment header block 中的 lhd 部分），可以再次 dump 这个 0x1c0000b 块，同样查看转储的结果，找到下面的内容：

```
fnx: 0x0
```

0x0 表示下面没有可用的空闲块了，也就是表明这是 Freelists 中的最后一个空闲块。

注 意

读者的测试可能得到跟我不一样的转储内容，这是正常的。

12.4 有多少种 free list

1. Master Free List 或者 Segment Free List

Master Free List 简称为 MFL，在 segment 被创建的时候自动生成，如果在创建 segment 时没有指定 Freelists 参数，或者指定 Freelists=1，都是生成这个 MFL。MFL 对于每个 segment 来说有且只有一个（如果指定 Freelists>1，产生的就是不是 MFL，这一点将在 Process Free List 部分解释）。MFL 相当于一个空闲空间池，当一个 segment 被创建时的初始化 block 以及以后动态分配的新 block 都链接到 MFL 中，这个池中的所有空闲块是被所有进程共享的，对于该 segment 有 insert 操作的所有进程都可能会去读取这个 free list，这样当有多个进程要同时 insert 数据时，就可能出现在 MFL 上的争用（MFL 在一个时间只能允许一个进程取得空闲块，当然，其实进程从 MFL 上读取空闲块的操作并不是简单地需要多少就取多少，取得以后就直接向块中插入数据，实际上的过程要更复杂一些，这个过程在“进程请求空闲块的过程”部分会有详细描述）。由此，推出了 Freelist Groups 的概念，设置 Freelist Groups 参数大于 1 就是设置了多个 MFL，这样就缓解了对于 MFL 的争用。有关 Freelist Groups 更详细的内容将在“Super Master Free List”部分进行描述。

2. Process Free List

如果进程必须直接从 MFL 中读取空闲块，那么对于 MFL 的争用由 Freelist Groups 参数解决（设置多个 MFL），但是显然还有另外一个思路就是尽量不让进程去直接读取 MFL，没有需求自然就无所谓争用。由此引入了另外一个级别的 free list，这就是 Process Free List，简称为 PFL。当指定存储参数 Freelists>1 的时候，生成的就是 PFL。

前面说过 MFL 是在 segment 被创建的时候自动生成的，所以无论是不是有 PFL，对于每个

segment 来说都仍然存在 1 个 MFL。也就是如果定义 Freelists 等于 2 的话，那么在 segment header block 中将总共存在 3 个 Freelist，其中 1 个是 MFL，另外 2 个是 PFL。

这一点同样可以通过 dump 转储信息来验证。

```
SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
SEG LST:: flg: USED  lhd: 0x01c0008b ltl: 0x01c0008b
```

后面两个 free list 是 PFL，而前面一个则是 MFL。

顾名思义，既然命名为 Process Free List，那么显然位于这个级别的 free list 中的空闲块只能被一个进程读取。想象一下，如果当前系统对于某张表最多会有 10 个进程同时做 insert 操作，那么设置 Freelists=10，将能尽量满足每个进程都能够使用专属于自己的 free list，无疑通过这样的手段可以缓解 free list 的争用。

一个进程到底会使用哪个 PFL，Oracle 内部的算法是：

```
(P % NFL) + 1
```

其中 P 表示 DML 操作进程的 Process ID，可以从 v$process.pid 字段中取得。NFL 表示 Freelists 存储参数定义的 PFL 数量。

可能会有疑问，如果是这样，多个 MFL 有存在的必要吗？只需要设置多个 PFL 不就可以了吗？然而事实并非如此，在后面的“进程请求空闲块的过程”内容中会解释这个问题。

3. Transaction Free List

在 Oracle 中事务（transaction）是一个重要的概念，每次 DML 操作，事务的开始都是自动的，可以通过 commit 或者 rollback 来标志一个事务的结束。一个进程（或者说一个用户会话）有自己的 PFL，然后一个进程可能会执行很多的事务，于是又出现了这个级别的 free list，这就是 Transaction Free List，简称为 TFL。

TFL 是动态产生的，只有当 DML 语句（比如 delete 或者 update）使 block 占用量降到 PCTUSED 参数指定值之下时才会生成 TFL，一个 TFL 只属于一个事务，而一个事务也只会有一个 TFL，一个事务没有提交之前，此事务的 TFL 上的空闲块不会被其他事务使用。但是可以立刻被本事务使用（此时这些空闲块被称为 previously freed blocks）。

每个 segment 最少可以有 16 个 TFL，同时只要有需求就会动态增加 TFL 数量，除非达到了 segment header block size 的限制。当没有空间允许新的事务得到自己的 TFL 时，这个事务就必须等待其他的事务提交并释放 TFL。等待哪个事务的算法是：

```
(P % NFL)
```

其中 P 表示 DML 操作进程的 Process ID，可以从 v$process.pid 字段中取得。NFL 表示当前的 TFL 总数量。

通过 dump 转储数据块信息，可以看到类似于下面的内容：

```
XCT LST:: flg: USED  lhd: 0x01c0008c ltl: 0x01c0008a xid: 0x0008.01f.000003d2
```

其中 xid 表示 transaction id，关于 transaction id 的格式和表示的意义，有兴趣的读者可以查看其他的资料。

4. Super Master Free List 或者 Segment Master Free List

这个级别的 free list 只有在设置了多个 Freelist Groups 时才会出现。

当设置 Freelist Group>1，就会产生 freelist group block，这些 block 紧跟在 segment header

block 之后，假设设置了 storage（Freelists 4 Freelist Groups 2），那么该 segment 的第一个块是 segment header block，第 2、3 个块则都是 freelist group block。

首先在 segment header block 中存在 1 个 free list，这个 free list 就被称为 Super Master Free List 或者 Segment Master Free List。

而在每个 freelist group block 中又都存着 1 个 MFL，还存在 4 个 PFL，每个 freelist group block 块的剩余空间则全部留给 TFL 使用。

在单个 instance 中进程选择 Freelist Group 的算法是：

```
(P % NFB) + 1
```

其中 P 表示 DML 操作进程的 Process ID，可以从 v$process.pid 字段中取得。NFB 表示 Freelist Groups 参数定义的 Freelist Groups 数量。

而在 RAC 环境中的算法则更加复杂，这里就不作讨论了。

查看 freelist group block 的转储文件可以看到类似于下面的内容：

```
frmt: 0x02 chkval: 0x0000 type: 0x16=DATA SEGMENT FREE LIST BLOCK WITH FREE BLOCK COUNT
  blocks in free list = 5 ccnt = 0
  SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
  SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
  SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
  SEG LST:: flg: USED   lhd: 0x01c00116 ltl: 0x01c0011a
  SEG LST:: flg: UNUSED lhd: 0x00000000 ltl: 0x00000000
```

12.5 进程请求空闲块的过程

下面通过对两个事务的描述来说明进程请求空闲块的整个过程。

（1）事务 T1 删除了表 T 中的一些数据，释放了这个 segment 的 block 10 中的一些空间。并且使 block 的已用空间降到 PCTUSED 参数值以下，因此在 segment header block 中产生了一个 T1 的 TFL，而 block 10 被 link 到这个 TFL 中。

（2）事务 T2 想要插入一些数据到表 T 中。但是由于 T1 没有提交，所以 block 10 并不能被 T2 使用，而假定 T2 没有作过释放空间（delete 或者 update）的操作，所以 T2 也没有自己的 TFL。

❶ T2 开始查找自己的 PFL，尝试找到可以使用的空闲块（术语称为 WALK）。假设在 T2 的 PFL 上有三个 block：block 11、block12、block13，但是都没有足够的空间满足 T2 的需求。

❷ 假定 block 11 的已用空间超过了 PCTUSED 参数值，而又由于无法满足 T2 需求，所以从 T2 的 PFL 上被摘除（术语称为 UNLINK），同时 free list header 变为下一个 block，也就是 block 12（术语称为 EXCHANGED），同样 block 12 不满足需求，又 exchange 到 block 13，可惜的是 block 13 也同样不满足需求，于是 T2 的 PFL 上就没有可使用的空闲块了。

这个步骤的后台思想是当一个块在搜索空闲块的过程中失败，那么就不应该把这个块再放在 free list header 处。

注 意

dead block 的出现通常是由于设置了过高的 PCTUSED 参数所致。假设设置 PCTUSED=90，那么如果由于 update 等原因使一个 block 中的数据占用量降低了 90%以下，这个 block 就立刻

被重新 link 到 free list 中，但是很可能这一点儿空间根本就不允许 insert 一条记录，又因为此时的 block 占用量仍然位于 PCTUSED 参数值之下，所以即使这个 block 不满足插入一条记录的条件，也仍然被放置在 free list header 处。当进程需要 free block 的时候，会先查找这些块（默认最多查找 5 个块），如果都不满足 insert 的需求，才尝试提升 HWM。

❸ T2 将 PFL 上的所有块都检查过，并且发现没有可用的，此时就停止查询 PFL。

❹ Oracle 尝试从 MFL 中移动空闲块到 PFL 中（术语称为 Free Lists Merge）。移动的块数是一个常量——5 个块。如果移动成功，那么从❶步骤重新开始。

注　意

此时 Oracle 不会去检查其他 PFL，即使在其他的 PFL 上可能会有空闲块。

❺ 如果上一步中在 MFL 中没有找到可用的块，此时 Oracle 尝试从其他已经提交的事务的 TFL 中获取 block。扫描 header block 中的 TFL entries，查看是否又已经提交了并且链接有空闲块的 TFL。如果找到，所有的空闲块都转移到 MFL 中。然后从❹步骤重新开始。

❻ 如果上一步失败，Oracle 尝试提升高水位标志（bump up HWM）。为了减少在 segment header block 上的争用，Oracle 每次只提升 m 个 block。m 的算法如下：

- 1：如果 HWM<=4 并且位于第一个初始化的 extent 上。
- 5：未设置隐含参数_bump_highwater_mark_count 时。
- min(_bump_highwater_mark_count * (PFL number+1) , unused blocks in the extent)：设置了隐含参数_bump_highwater_mark_count 时。

❼ 如果上一步失败（HWM 之上没有分配了但还未使用的 block），则分配新的 extent，新分配的在只有一个 MFL 情况下转移到 MFL 上，然后从❹步骤重新开始。在有多个 PFL（Freelists>1）的情况下，则直接转移到申请空间的进程所拥有的 PFL 上，然后从❶步骤重新开始。

（3）事务 T1 准备向表 T 中插入新的记录。首先，它会检索自己的 TFL 上是否有空闲空间。由于 block 10 有可用空间，并且空间满足新记录的需求，所以 block 10 将会被使用。

总结一下上面描述的过程。

当重新 insert 数据或者发生 row migration 的时候，会从 TFL header 处开始使用已经释放了的空闲 block。如果 TFL 中没有 previously freed blocks，那么就从 PFL 中寻找，因为可能定义了多个 Freelist，所以到底从哪个 free list 中找，算法是前面描述过的(P % NFL) + 1。如果在 PFL 中找不到或者根本就没有 PFL（segment 只有一个 Freelist 的情况），那么再从 MFL 中找。仍然找不到的话，返回去再查其他会话的 TFL，判断其中的事务是否已经 commit，如果已经 commit 了，则把这个 TFL 的 flag 标志为 unused，同时把位于这个 list 中的所有 block 合并到 MFL 中。

如果经过上面的步骤，在任何一个级别的 free list 中都没有找到可用的空闲块，此时 segment 就要提升 HWM，如果提升 HWM 失败，则分配下一个 extent，新分配的 block 被 link 到 MFL 或者 PFL 中，重新开始上面的查找步骤。

当一个块由于 DML 操作而被重新链接到 free list 中是被放置在 MFL 中的，如果有 PFL 存在，那么在使用前，根据数据请求量，每次最多转移 5 个 block 到相应的 PFL 中，也就是在存在多个 Freelist 的情况下，一次空间的请求总是读取 TFL 或者 PFL，而不会直接从 MFL 中读取。

现在可以回答前面的那个问题了。

多个 MFL 有存在的必要吗？只需要设置多个 PFL 不就可以了吗？

从这个地方可以看到，虽然多个 free list 可以缓解多个并发的 session 同时更新一个 segment 时对于 data block 的争用，但是如果只有一个 Main Free List，那么在从 Main Free List 把 block 转移到 Process Free List 的这个环节上仍然会出现争用，此时就是多个 Freelist Group 发挥效果的时候了。所以多个 Freelist Group 不仅对于 RAC 环境有效，对于单个 instance 也是可以缓解一定的资源争用情况。

但是如果有多个 Freelist Group，就不可避免地会产生空间浪费的副作用，在某些特殊场合下甚至会让一个 segment 急速地增大，为什么会这样，可以参看后面关于 Freelist Groups 的那个比喻。

可以通过转储 segment header block 来查询 HWM 的位置，然后再查询转储 freelist group blocks 来查询 link 在每个 free list 上的空闲块，如果发现 HWM 的位置很高却又有大量的空闲块，就可能表示这个 segment 经历了不正常的 extent 扩展。

12.6 块在 free list 间的移动

下面以图 12-2 来说明这个算法及过程：

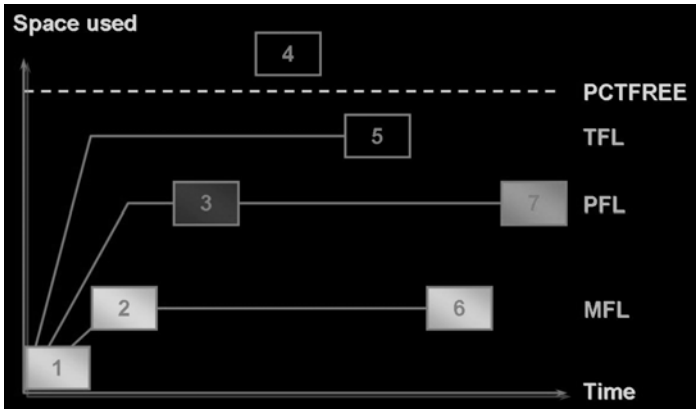

图 12-2 块在 free list 间的移动

(1) block 被分配给一个 segment，但是却不存在于 free list 中，因为这个 block 还位于 HWM 之上。

(2) 当 block 降到 HWM 之下时（bump up HWM），这个块就进入 MFL 或者一个 Free List Group 的 MFL 中。

(3) 如果设置了 Freelists>1，也就是存在多个 PFL 的情况下，block 会进入 PFL。有利于缓解对于 MFL 的争用。

(4) 当 block 的可用空间低于 PCTFREE 定义的值，block 从 free list 中删除。此 block 不再被允许进行 insert 操作。

(5) 当一个事务进行了 delete 或者 update 操作，致使 block 的已用空间低于 PCTUSED 定义的值，该块进入这个事务的 TFL，除非事务 commit，否则该块只能被此事务使用。

(6) 在寻找 free block 时，空闲块会从 TFL 转移到 MFL。

(7) 当用户进程需要空闲块时，空闲块又从 MFL 转移到 PFL。

12.7　关于 free list 将导致大量空间浪费的误解

一直以来，有观点认为多个 Freelist 也可能会导致大量空间浪费，其实这是一个误解。至少从测试和某些具有权威性的文档来看，多个 Freelist 没有太多的空间浪费。

回顾一下一个事务请求空间的整个过程，如图 12-3 所示。

（1）一个 session 需要空间了，先在自己的 TFL 上找，这是在找 previously freed blocks。

（2）找不到，找自己的 PFL。

（3）再找不到，找 MFL，如果找到，移动最多 5 个块到自己的 PFL 上，然后重新查找 PFL。

（4）如果 MFL 中仍然找不到，返回去找其他会话的 TFL，看看有没有 commit 了的事务，如果有，将这个 TFL 中的 free block 转移到 MFL 上（这个时候其他事务的 free block 就能被这个事务使用了），然后重复（3）步。

（5）所有的 TFL 上都没有空闲块，查找 Super Master Free List，如果找到，将最多 5 个块转移到 MFL 上，然后重复第（3）步。

（6）如果找不到，移动 HWM（bump up HWM），如果成功，那么直接将块转移到自己的 PFL 上，重新查找 PFL。

（7）如果移动 HWM 不成功，那么分配 extent，也就是分配新的 block，重复（6）步。

（8）分配 extent 不成功，最终报错，没有可用的空间。

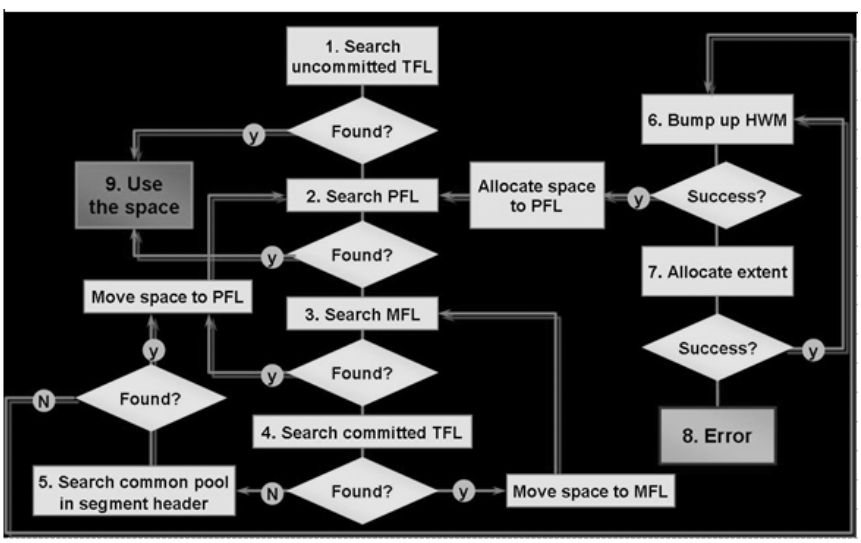

图 12-3　Free List Searching

可见多个 Freelist 只是多个 Process Freelist 而已，而 Main Freelist 始终只有 1 个，又因为 Main Freelist 是共享的，所有在同一个 Freelist Groups 中的 process 都可以从这个 Main Freelist 中取得 free block，而每次取得的空闲块数也最多只有 5 个块，那么也就是一个 PFL 下不会链接过多的空闲块，同时当一个事务删除了大量的数据并且 commit 之后，这些空闲块是会归还给 MFL，从而可以被其他事务使用。所以即使是设置了多个 Freelist，也并不会存在多少浪费空间的问题。当然不论如何，设置了过多的 Freelist，终究是会浪费一些 free block（即使可能只是浪费 HWM 之下

的几十个 block），而且还有一个坏处就是会减少可以分给 TFL 使用的 free list entry，所以比较合适的设置是将 Freelists 设置为最多可能同时操作该 segment 的会话数。

12.8 关于 Freelists 和 Freelist Groups 的一个比喻

对于 Freelist 和 Freelist Group，Tom 在《Effective Oracle by Design》中有个比喻，非常形象。想象有一台饮水机和源源不断的急需喝水的人们，一台饮水机就代表一个 free list，而一个想喝水的人就是一个准备向 segment 中插入数据的会话。如果只有一个饮水机，那么所有想喝水的人都必须要排成一队，然后前一个喝完了下一个才能喝，这就产生了争用。设想一下现在放置 10 个饮水机，很明显人们可以排到 10 个队伍中的任何一个队伍里，毫无疑问效率将大大提高。这时瓶颈又出现了，就是如果一个饮水机里的水被喝完了，就得给这个饮水机加水，此时如果只有一个加水员（这就是一个 Freelist Group），那么加水的速度可能就会跟不上了，添加 Freelist Group 就是增加加水员，增加到 2 个，每个人负责 5 个饮水机（体现在参数上，是 Freelists 5 Freelist Groups 2），那么效率又提升了。

对于空间浪费的负面影响，可以继续设想一下，来了一个十分能喝水的人，他守住一个饮水机不停地喝，喝完了，引水员就加水，又喝完了，又加，即使是其他的 9 个饮水机里都是满满的水也没用，因为这个人一旦从某个送水员管辖下的饮水机中开始喝水，就不会换到另外一个送水员管理的饮水机上去（一个会话一旦从某个 Freelist Group 的 Freelist 中开始读取空闲块，就不会再使用其他 Freelist Group 中的 Freelist，即使其他的 Freelist 中还有很多的空闲块）。

12.9 与 Freelists 和 Freelist Groups 相关的等待事件

v$system_event 视图中记录了从数据库启动以来所有的等待事件信息，通过这个视图可以比较清楚地知道数据库经历的最严重的等待是哪些。

但是一个等待事件可能会是各种各样的原因导致的，所以在检查 v$system_event 视图的同时还需要一些其他视图的辅助，才可以更准确地定位问题的所在，比如可能还需要去查看 v$session_wait、v$waitstat 等视图。

跟 Freelists 和 Freelists Groups 相关的等待事件中，比较主要的是 Buffer Busy Waits 事件和 Enqueue 事件。

1. Buffer Busy Waits 事件

当一个进程需要一个数据块，但是发现这个数据块正在被其他进程读入 Buffer Cache，或者说这个数据块虽然已经存在于 Buffer Cache 中，但是却处于一个无法共享的状态，此时就经历 buffer busy waits 等待事件。

通过检查 v$system_event 视图，可以确认系统是否正在遭受严重的 buffer busy waits 事件（TIME_WAITED 字段的值过大，表示等待总时长比较显著），但是这还不足以判断问题到底出在什么地方，此时就需要再去检查 v$waitstat 视图。这个视图中每一行都记录了系统经历的不同类别 block 的等待时间。通常如果发现 segment header 类别或者 data block 类别的等待时间显著，那么就很可能是 Freelists 争用。

如果能够明白这两种类别的 block 等待产生的原因，自然就有相应的解决方法了。

首先解释一下为什么会产生 segment header 部分的等待。前文描述过如果没有使用多个 free list group，那么所有的 free lists 都是位于 segment header block 中的。一个进程对于 free list 的操作将持有独占锁，如果多个进程同时请求操作 free list，那么就会产生争用，而这个争用在 v$waitstat 视图中就表现为 segment header 上的等待。

那么为什么会产生争用，这又有两种原因。

一种是同时要求操作 free list 的进程确实过多，这种情况下需要增加 free list。

另外一种情况则是实际上同时操作的进程并不多，但是操作 free list 的频度过高，为什么会频繁操作，因为不停地有 block 从 free list 中被摘掉同时又不停有新的 free block 被放入 free list 中，这通常表示设置了不正确的 PCTFREE 和 PCTUSED 参数值，比如 PCTUSED 值和 PCTFREE 值设置得过近，这种情况下盲目增大 free list 可能并没有什么效果，需要做得是设置正确的 PCTUSED 和 PCTFREE 值。

再来解释为什么会产生 data block 部分的等待。因为每个进程对于 free block 的请求都是从 free list header 处开始的，如果 free list 较少，那么无疑就增加了不同进程取得的 free block 是同一个块的可能性。多个进程又同时去更新一个 block，这就产生了 data block 的等待，这也是常说的热点块问题。由于热点块问题，同时在等待事件中可能比较显著的还会有 latch free（Cache Buffers Chains Latches），latch 又是 Oracle 内部的另外一种锁定机制了，本章不再深入解释。

增加 free list 可以缓解这个问题，因为每个 free block 只会属于一个 free list，给不同的进程分配不同的 free list 就减少了并发进程同时访问一个数据块的风险。

另外设法将被频繁并发访问的 segment 上的数据分布在更多的 block 中也可以缓解这个问题，比如可以增加 PCTFREE 值，以扩大数据分布。但是这就引出了另外一个问题，如何确定哪些数据块是热点块，只有知道了是哪些数据块，才可能知道这些块是属于哪个 segment，才能有的放矢地调整相应 segment 的存储参数或者甚至将整个 segment 转移到另外的数据文件或另外的存储介质上。

由此引入另外一个视图 v$session_wait。这个视图中存储的是一个会话实时的等待事件，如果只是偶尔去检查一次，可能正好捕捉不到 buffer busy waits 事件，所以应该通过计划任务（Oracle JOB 或者操作系统级别的 cron）定时地检查这个视图，将检查结果存入另外的表中，建议可以在系统繁忙期每隔 5 秒或 10 秒运行一次下面的 SQL，注意取样的总时间不能过长，因为这个 SQL 对于系统还是有影响的。

```
SELECT /*+rule*/s.username, e.owner, e.segment_name, p1 "File#", p2 "Block#"
FROM v$session s, v$session_wait w, dba_extents e
WHERE w.event='buffer busy waits'
AND s.sid = w.sid
AND e.file_id = p1
AND p2 between e.block_id and e.block_id +(e.blocks - 1);
```

返回的结果中，username 表示读取这个块的用户名，owner 表示拥有热点块所属 segment 的用户名，segment_name 表示热点块所属的 segment 名称，File#表示热点块所处的数据文件号，Block#表示热点块号。

注　意

如果使用 Oracle 10g，那么新增的性能视图 v$session_wait_history 将自动保留最近的 10 个会话等待事件的历史信息，通过这个视图可以更简单地判断问题。

如果设置了多个 Freelist Group，那么在 v$waitstat 视图中还可能会发现 free list 类别的等待，这个等待表示在 Freelist Group blocks 上产生了争用。注意，只有当设置了多个 Freelist Groups 时，才可能出现这个类别的 buffer busy waits 等待事件。至于调优的方法可以参看上面的调整手段。

2. Enqueue 事件

enqueue 也是一种锁（lock），只不过是 Oracle 用来管理共享内存结构的锁定机制，通过 enqueue 来保证对于数据库资源的串行存取。

当在 v$system_event 中或者 Statspack 报告中发现了严重的 enqueue 等待，首先需要确定的是系统在经历什么类型的 enqueue 等待。

如果使用了 Statspack，那么可以查看报告相应部分的 enqueue 等待详细信息，或者也可以通过以下方法来确定 enqueue 类型。

运行下面的 SQL，得到 enqueue 等待的相应记录。

```
SELECT username, event, p1, p2, p3
FROM v$session s, v$session_wait w
WHERE event='enqueue'
AND s.sid = w.sid;
```

在返回结果中，p1 表示 enqueue 的类型和模式，p2 和 p3 根据 p1 的不同有不同的含义。

注 意

在 Oracle 10g 之前，由于各种 enqueue 类型都显示为同样的 enqueue 等待，所以 p2 和 p3 的含义并不能从公开的文档中获得，而在 10g 中各种 enqueue 都显示为单独的 enqueue 等待，所以可以从 v$event_name 视图的 PARAMETER2 和 PARAMETER3 字段中轻松获得每种不同 enqueue 的 p2 和 p3 含义。

p1 的返回值是十进制，转化为十六进制以后，低 2 位表示模式，高 2 位表示类型。现在假设 p1=1213661190，看一下如何通过这个数字得到 enqueue 的信息。

十进制 1213661190 转化为十六进制：4857 0006。

低 2 位：0006 → 模式（mode）是 6，表示为 Exclusive。

高 2 位：4857 → 将 4857 分成两个单字节，分别是 48 和 57。

将 48 和 57 转化为十进制：分别是 72 和 87。

ASCII 为 72 的字符是 H，ASCII 为 87 的字符是 W。

```
SQL> select chr(72)||chr(87) "enqueue" from dual;

enqueue
-------
HW
```

由此知道了现在经历的是 HW enqueue 等待，并且是独占模式（mode = Exclusive）。

也可以简单地使用下面的 SQL 直接获得 enqueue 的类型和模式。

```
select sid,
      event,
      p1,
      p1raw,
      chr(bitand(P1, -16777216) / 16777215) ||
      chr(bitand(P1, 16711680) / 65535) "TYPE",
```

```
        mod(P1, 16) "MODE"
  from v$session_wait
 where event = 'enqueue';
```

常见的跟 Freelists 争用相关的 enqueue 等待是 HW enqueue。

当给 segment 分配新的 extent 时，将放置 HW enqueue，如果有多个进程同时请求 insert 数据并且需要分配新的 extent 提升 HWM 时就会产生 HW enqueue 争用。通常发生在设置了多个 free list group 的情况下。

如果为了避免 Freelists 争用，而增加 Freelist Group，确实可以减少 buffer busy waits 事件，也会减少所有类别的 block 上的等待(从 v$waitstat 视图中可以得知)，但是却会增加 HW enqueue 的等待事件。

因为一个进程使用了一个 Freelist Group 之后，即使其他的 Freelist Group 中有大量的空闲空间（比如一个进程刚刚删除了大量的数据，从而在自己的 TFL 上腾出了很多空闲块），这个进程也不会去检索。所以如果这个进程正好需要 insert 大量数据，那么就很可能会提升 HWM，也就会放置 HW enqueue，那么如果还有其他的进程也需要 insert 数据，就不可避免地经历 HW enqueue 等待。正是由于多个 Freelist Group 使提升 HWM 的几率变大，所以也就会导致 HW enqueue 等待事件的增多。

参考信息

1．Ixora: Data Blocks and Freelists

2．Metalink Note:1029850.6,Freelists and Freelist Groups

3．Metalink Note:157250.1,Freelist Management With Oracle 8i

4．Thomas Kyte. Effective Oracle by Design. McGraw—Hill Osborne Media,2003

5．Richmond Shee,Kirtikumar Deshpande,K Gopalakrishnan. Oracle Wait Interface: A Practical Guide to Performance Diagnostics & Tuning

6．ITPUB 讨论: http://www.itpub.net/319420.html

作者简介

张乐奕，网名 kamus，曾任 ITPUB Oracle 认证版版主，现任 ITPUB Oracle 管理版版主。

曾任职于北京某大型软件公司，首席 DBA，主要负责证券行业的全国十数处核心交易系统数据库管理及维护工作。现任职于某外资电信企业北京分公司，高级 DBA，主要负责 3G 相关业务。

热切关注 Oracle 技术和其他相关技术，出没于各大数据库技术论坛，目前是中国最大的 Oracle 技术论坛 www.itpub.net 的数据库管理版版主。

阅读更多技术文章和随笔可以登录个人 blog（http://blog.dbform.com）。

第三篇
内存调整篇

本篇共分 5 章，主要内容如下：

第 13 章　自动 PGA 管理——原理及优化

第 14 章　32bit Oracle SGA 扩展原理和 SGA 与 PGA 的制约关系

第 15 章　KEEP 池和 RECYCLE 池

第 16 章　深度分析数据库的热点块问题

第 17 章　Shared Pool 原理及性能分析

13 第13章 自动PGA管理——原理及优化

在一个复杂的数据库系统中会同时运行着许多 SQL 语句，这些语句当中会进行 sort、hash_join 等很多操作，它们都必须分配一些内存进行这些操作，在 Oracle 的 dedicated server（当使用 shared server 时，workarea 将在 SGA 的 large pool 中分配，同时 PGA 自动内存管理将被忽略，所有 workarea 内存管理还是来自于 sort_area_size、hash_area_size 等参数的设置）环境中，这些内存将会被分配到 PGA（Program Global Area）中，如果 PGA 的内存分配管理不当将会导致系统内存不足，操作系统将会频繁 page in/out，会降低数据库的整体性能，所以正确管理 PGA 内存分配是一件相当重要的事情。

在 Oracle 9i 以前的版本中，可以通过手工修改 sort_area_size、hash_area_size 等值来控制 PGA 的使用率，使用这种分配方法会存在一个弊端，因为数据库中存在成千上万条不同的 SQL 语句，它们之中有的需要很大的内存，有的需要很少，如果指定一个大的 sort_area_size 或 hash_area_size，虽然保证了某些语句的执行速度，但是 PGA 总体内存将会过度分配，严重时会导致操作系统内存紧缺，降低整个数据库的性能。如果 sort_area_size 或 hash_area_size 设置的太小，那么某些语句将会执行过长的时间，影响应用程序的响应速度。所以如何手工设置合适的 PGA 参数在 Oracle 9i 推出之前一直是很多 DBA 心头的痛。不过这种情况在 Oracle 9i 推出后已经得到了比较好的解决，9i 的 PGA 自动管理功能很好地解决了这个问题。

13.1 什么是 PGA 内存自动管理

Oracle 8i 或更早版本中用 sort_area_size 和 hash_area_size 参数来指定每个 session 可用的最大排序和 hash 内存大小。

在 Oracle 9i 中，用 pga_aggregate_target 参数来指定所有 session 一共使用最大 PGA 内存的上限。这个参数可以被动态地更改，赋值范围从 10MB ～ 4096GB-1 byte。

9i 里还提供了 WORKAREA_SIZE_POLICY 参数用于开关 PGA 内存自动管理功能。

- auto：自动管理。
- manual：手工管理。

当 WORKAREA_SIZE_POLICY 设置为 manual 时，PGA 的内存分配还是使用 sort_area_size 等参数来分配，设置为 auto 时，sort_area_size 等参数将被忽略。

注 意

在 9i 中 WORKAREA_SIZE_POLICY 默认被设置为 auto。

在 OLAP 和 DSS 系统中通常会存在很多相当复杂的语句，它们会进行多表关联，处理很大的数据量，会存在很多 sort 和 hash join，这些语句将会需要很多内存空间去执行。当这些内存分配后，Oracle 进程会在其中进行排序或构造 hash 表。一般来说，分配的内存越多 SQL 语句就会执行得越快。通常把这段执行的区域叫做 work area，把能进行完全内存操作的 work area 大小叫做 optimal（cache）size。与之对应的还有 onepass size 和 multipass size。

- optimal：所有操作都在内存中进行。
- onepass：使用最小写磁盘操作,大部分在内存中进行。
- multipass：当 workarea 太小的话将会发生大量磁盘操作，性能急剧下降。

这里 sort 和 hash join 还是有点区别，当进行 sort 操作的话，分配的内存介于 optimal 与 onepass 之间，语句的响应时间是不会随着内存的增加而减少的。但是如果是 hash join，那么一旦增加内存，它的响应时间将会随之减少，这与 sort、hash join 的内部执行方式有关，有兴趣的读者可以参考 sort、hash join 的相关文档。

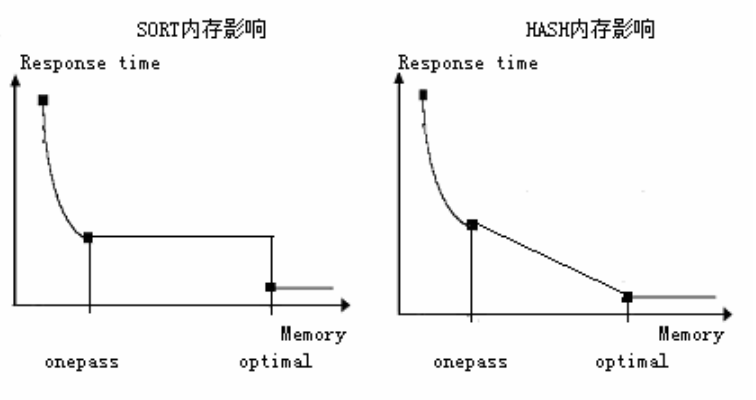

图 13-1 sort 和 hash 的内存影响

从图 13-1 中，也可以发现 PGA 内存大小对响应时间还是影响很大的，所以应该尽量让它们都在内存中进行，一个典型的 OLTP 环境需要：

```
workarea execution - optimal >= 90%
workarea execution - multiplass = 0%
```

现在了解了 PGA 内存分配的重要性，也知道 PGA 自动管理的优点，接下来深入到 PGA 自动管理里面看看内部是怎么实现的。

PGA 自动管理是采用 feedback loop 算法来管理内存分配的，如图 13-2 所示。

在图 13-2 中，当一条语句开始执行，它会通过 Local Memory Manager 注册一个 Active Workarea Profile，Profile 里面包括了这个 Workarea 的一系列属性，它的类型（sort、hash-join 等），它当前执行需要的 minimum，one-pass and optimal 内存大小，它的并行度，它当前在使用的内存等。由于 Profile 经常被更新，所以所有 Active Profiles 基本可以反映出当前内存需要和当前在使用的内存。有了这些 Profile 信息，后台进程 Global Memory Manager 就可以计算出全局的 memory

bound，Global Memory Manager 每隔 3 秒将会更新一下 memory bound，Local Memory Manager 得到 memory bound 后将会计算出每个 Active Statement 所要分配的内存大小，在这里被称为 expect size。然后每个 Active Statement 将会在自己所分配到的 expect size 内存中进行操作。

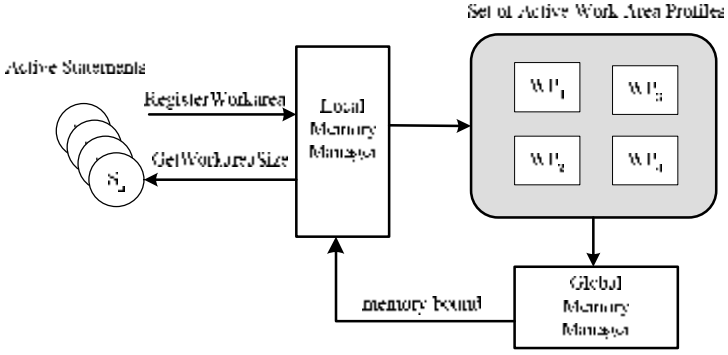

图 13-2　管理内存分配

图 13-2　管理内存分配

计算 expect size 的时候还需要符合以下 5 个规则。

- 规则 1：expected size 不能小于 minimum memory 需求。
- 规则 2：expected size 不能大于 optimal memory 需求。
- 规则 3：如果 bound 介于 minimum 和 optimal 之间，那么将会使用 bound 值作为 expect size。sort 操作除外，因为 sort 操作并不会从多余内存中得到利益，上面也已经提到，所以当 sort 操作时将会取 onepass 作为 expect size。
- 规则 4：如果这个 workarea 是并行执行的，那么它的 expect size 和它的并行度成正比。
- 规则 5：最后，expect size 不能超过 5%pga_aggregate_target 大小或 100MB，并行操作下不能超过 30%pga_aggregate_target 的大小。

这里面 SQL memory target、global memory bound、dirft management 都运用了一系列机制来保证 PGA 自动内存管理的稳定性，在这里不再详述。

现在大家都了解了 PGA 自动内存管理的原理，下面就做一些实验来验证以上的理论。

```
SQL> alter system set pga_aggregate_target=10M;

System altered.

SQL> alter system set workarea_size_policy=auto;

System altered.

执行一个排序

select distinct * from a where rownum<500000;

Elapsed: 00:01:00.5

*View SQL workarea
SQL>
SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
```

```
      last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
 WHERE l.hash_value = a.hash_value
   AND sql_text = 'select distinct * from a where rownum<500000';

SQL_TEXT                                   OPERATION_TYPE
------------------------------------------------- ----------------
select distinct * from a where rownum<500000    GROUP BY (SORT)

POLICY LAST_MEMORY_USED/1024/1024   LAST_EXECUTION LAST_TEMPSEG_SIZE
------ --------------------------- -------------- -----------------
AUTO   .5                       30 PASSES     12288000
```

可以看到这里使用到了 0.5MB 的内存，再把 pga_aggregate_target 改为 40MB：

```
SQL> alter system set pga_aggregate_target=40M;

System altered.

SQL> select distinct * from a where rownum<500000;

Elapsed: 00:00:38.36

SQL>
SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
      last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
 WHERE l.hash_value = a.hash_value
   AND sql_text = 'select distinct * from a where rownum<500000';

SQL_TEXT                                   OPERATION_TYPE
------------------------------------------------- ----------------
select distinct * from a where rownum<500000    GROUP BY (SORT)

POLICY LAST_MEMORY_USED/1024/1024   LAST_EXECUTION LAST_TEMPSEG_SIZE
------ --------------------------- -------------- -----------------
AUTO   2                        1 PASSES      12288000
```

可以看到内存使用达到 2MB，也正好是 5% pga_aggregate_target，这里证明了 expect size 不能超过 5% pga_aggregate_target 大小的理论。

再把 pga_aggregate_target 改成 10GB：

```
SQL> alter system set pga_aggregate_target=10G;

System altered.

SQL> select distinct * from a where rownum<500000;

Elapsed: 00:00:30.23

SQL>
SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
      last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
```

```
 WHERE l.hash_value = a.hash_value
   AND sql_text = 'select distinct * from a where rownum<500000';

SQL_TEXT                                    OPERATION_TYPE
------------------------------------------------ ----------------
select distinct * from a where rownum<500000    GROUP BY (SORT)

POLICY LAST_MEMORY_USED/1024/1024   LAST_EXECUTION LAST_TEMPSEG_SIZE
------ ---------------------------- -------------- -----------------
AUTO   19.35                        OPTIMAL
```

可以看到这时候使用了 19.35MB 内存，所有排序操作都在内存中完成，语句的执行时间也从 00:01:00.5 减少到了 00:00:30.23。

上面还提到了 expect size 不能超过 100MB，来看一下是否能测试出来：

```
SQL>
SELECT NAME, VALUE / 1024 / 1024 / 1024
  FROM v$parameter
 WHERE NAME = 'pga_aggregate_target';

NAME                           VALUE/1024/1024/1024
---------------------------------- --------------------
pga_aggregate_target                            10

SQL> select distinct * from a;

5000000 rows selected

Elapsed: 00:09:35.42

SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
       last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
 WHERE l.hash_value = a.hash_value AND sql_text = 'select distinct * from a';

SQL_TEXT                  OPERATION_TYPE    POLICY
--------------------------- ----------------- --------
select distinct * from a    GROUP BY (SORT)   AUTO

LAST_MEMORY_USED/1024/1024   LAST_EXECUTION   LAST_TEMPSEG_SIZE
---------------------------- ---------------- -------------------
92.356732                 1 PASS         129138688
```

为什么不能用到 5% pga_aggregate_target=0.05*10GB=500MB 的内存呢?这是由于有隐藏参数_pga_max_size 来控制，每个 session 只能用到一半_pga_max_size 值大小的内存，_pga_max_size 默认的大小为 200MB。再来看看改了_pga_max_size 的效果如何：

```
SQL> alter system set "_pga_max_size"=500M;

System altered

SQL> select distinct * from a;

5000000 rows selected
```

```
Elapsed: 00:08:55.23

SQL>  SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
      last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
 WHERE l.hash_value = a.hash_value AND sql_text = 'select distinct * from a';

SQL_TEXT                  OPERATION_TYPE    POLICY
------------------------- ----------------- --------
select distinct * from a    GROUP BY (SORT)   AUTO

LAST_MEMORY_USED/1024/1024   LAST_EXECUTION   LAST_TEMPSEG_SIZE
---------------------------- ---------------- -------------------
198.56346                 OPTIMAL
```

可以看到 198.56346 的内存被使用并且全部在内存中执行。再来看一下并发情况下 PGA 自动内存管理的效果，首先看一下 20 个并发的情况：

```
SQL> select value/1024/1024 from v$parameter
where name = 'pga_aggregate_target';

VALUE/1024/1024
---------------
           50
SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
      last_execution, last_tempseg_size
  FROM v$sql l, v$sql_workarea a
 WHERE l.hash_value = a.hash_value AND sql_text = ' select distinct * from a where
rownum<300000';

SQL_TEXT                                          OPERATION_TYPE
------------------------------------------------- ----------------
select distinct * from a where rownum<300000       GROUP BY (SORT)

POLICY   LAST_MEMORY_USED/1024/1024   LAST_EXECUTION  LAST_TEMPSEG_SIZE
-------- ---------------------------- --------------- -------------------
AUTO    2.328674                 1 PASS         6995868
```

这时用了 2.328674/50=4.6% pga_aggregate_target 大小。换成 40 个并发的情况：

```
SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024 / 1024,
      last_execution, last_tempseg_size
      FROM v$sql l, v$sql_workarea a
      WHERE l.hash_value = a.hash_value AND
      sql_text = ' select distinct * from a where rownum<300000';

SQL_TEXT                                          OPERATION_TYPE
------------------------------------------------- ----------------
select distinct * from a where rownum<300000        GROUP BY (SORT)

POLICY   LAST_MEMORY_USED/1024/1024   LAST_EXECUTION  LAST_TEMPSEG_SIZE
-------- ---------------------------- --------------- -------------------
AUTO    .52145                   15 PASSES      7995392
```

可以看到 0.52145/50=0.010429=1%，由于没有足够的内存让 40 个进程进行 onepass 操作，所以这里每个进程只用到了 1%左右的内存来进行 multipass 操作。

再来看看如果 100 个并发的情况会怎么样：

```
SQL> SELECT sql_text, operation_type, POLICY, last_memory_used / 1024,
      last_execution, last_tempseg_size
      FROM v$sql l, v$sql_workarea a
      WHERE l.hash_value = a.hash_value
      AND sql_text = ' select distinct * from a where rownum<300000';

SQL_TEXT                                        OPERATION_TYPE
----------------------------------------------- ----------------
select distinct * from a where rownum<300000      GROUP BY (SORT)

POLICY   LAST_MEMORY_USED/1024   LAST_EXECUTION   LAST_TEMPSEG_SIZE
-------- ----------------------- ---------------- -------------------
AUTO     595                  105 PASSES      11018240
```

注　意

100 个并发的情况下已经出现 overalloc 的情况了，每个进程使用了 0.6MB，100 个进程已经使用超过 pga_aggregate_target 的值了，这是因为 memory bound 每 3 秒才由 Global Memory Management 更新一次，发布的 memory bound 还未及时反映到内存使用情况导致。但是这也说明一种情况，就是 pga_aggregate_target 是可以被突破的，虽然这种情况很少见。也可以从 v$pgastat 中看到 overalloc 的信息。

```
SQL> select * from v$pgastat where NAME  like 'over%';
NAME                                             VALUE UNIT
------------------------------------------------------  -----------
over allocation count                              15483
```

13.2　PGA Advice 功能

自从 9i release 2 开始，Oracle 提供了 PGA advice 功能，主要通过 2 张视图 v$pga_target_advice 和 v$pga_target_advice_histogram 实现建议功能。下面看一下这两张视图的结构，如表 13-1、13-2 所示。

表 13-1　　v$pga_target_advice 视图结构

列　名	数 据 类 型	字 段 描 述
PGA_TARGET_FOR_ESTIMATE	NUMBER	估计的 pga_aggregate_target 大小（bytes）
PGA_TARGET_FACTOR	NUMBER	PGA_TARGET_FOR_ESTIMATE 除以当前 pga_aggregate_target 得出的值
ADVICE_STATUS	VARCHAR2(3)	指示 advice 开启或关闭
BYTES_PROCESSED	NUMBER	所有 workarea 操作的数据量（bytes）
ESTD_EXTRA_BYTES_RW	NUMBER	估计设置 pga_aggregate_target 为 PGA_TARGET_FOR_ESTIMATE 时读写 byte 数

续表

列　　名	数 据 类 型	字 段 描 述
ESTD_PGA_CACHE_HIT_PERCENTAGE	NUMBER	估计内存命中率
ESTD_OVERALLOC_COUNT	NUMBER	估计 overalloc 次数

表 13-2　　v$pga_target_advice_histogram 视图结构

列　　名	数 据 类 型	字 段 描 述
PGA_TARGET_FOR_ESTIMATE	NUMBER	估计的 pga_aggregate_target 大小（bytes）
PGA_TARGET_FACTOR	NUMBER	PGA_TARGET_FOR_ESTIMATE 除以当前 pga_aggregate_target 得出的值
ADVICE_STATUS	VARCHAR2(3)	指示 advice 开启或关闭
LOW_OPTIMAL_SIZE	NUMBER	Histogram 区间内 Optimal 下限（bytes）
HIGH_OPTIMAL_SIZE	NUMBER	Histogram 区间内 Optimal 上限（bytes）
ESTD_OPTIMAL_EXECUTIONS	NUMBER	Histogram 区间内估计 optimal 次数
ESTD_ONEPASS_EXECUTIONS	NUMBER	Histogram 区间内估计 onepass 次数
ESTD_MULTIPASSES_EXECUTIONS	NUMBER	Histogram 区间内估计 multipass 次数
ESTD_TOTAL_EXECUTIONS	NUMBER	Histogram 区间内估计执行总次数
IGNORED_WORKAREAS_COUNT	NUMBER	Histogram 区间内由于内存和 CPU 限制而被 advice 忽略的 workarea 数

有了这两张视图，就可以很清楚地知道如何设置 pga_aggregate_target 比较好，来看下面的例子。

```
SQL> SELECT *  FROM v$pga_target_advice;

134217728   .125 ON  1.3903E+12  8.6153E+10   94  32014
268435456    .25 ON  1.3903E+12  9871660032   99      0
536870912     .5 ON  1.3903E+12  4168520704  100      0
805306368    .75 ON  1.3903E+12  3538477056  100      0
1073741824     1 ON  1.3903E+12  2298456064  100      0
1288489984   1.2 ON  1.3903E+12  2256093184  100      0
1503238144   1.4 ON  1.3903E+12  2188313600  100      0
1717986304   1.6 ON  1.3903E+12  2188313600  100      0
1932734464   1.8 ON  1.3903E+12  2188313600  100      0
2147483648     2 ON  1.3903E+12  2188313600  100      0
3221225472     3 ON  1.3903E+12  2188313600  100      0
4294967296     4 ON  1.3903E+12  2188313600  100      0
6442450944     6 ON  1.3903E+12  2188313600  100      0
8589934592     8 ON  1.3903E+12  2188313600  100      0
```

看到 PGA_TARGET_FACTOR 从 0.125 一直到 8，PGA 的大小从 134217728 bytes 到 8589934592 bytes，当前的 PGA 大小为 1073741824 bytes，当 PGA 设置为 134217728 bytes 时内存命中率为 94%，268435456 bytes 时为 99%，536870912 bytes 时就到 100%了，当然这是一个近似值，实际上从 ESTD_EXTRA_BYTES_RW 列可以看出来一直到 pga_aggregate_target 增加到 8GB，还是会有一些 disk read/write，这是因为数据库中存在着某些比较大的排序操作所致。

v$pga_target_advice_histogram 视图里有更清楚的分区间的统计数据，由于数据比较多，不在这里列出，有兴趣的朋友可以直接查询一下 v$pga_target_advice_histogram。这两个视图提供了直观数据来调整 PGA 的设置，从此调整 PGA 设置就不必像 9i Release 2 以前版本那么困难了。

13.3　自动 PGA 内存管理相关初始化参数

最后来看一下自动 PGA 管理的相关参数，如表 13-3 所示：

```
SELECT a.ksppinm, a.ksppdesc, b.ksppstvl
  FROM x$ksppi a, x$ksppsv b
 WHERE a.indx = b.indx AND a.ksppinm LIKE '_smm%';
```

表 13-3　　自动 PGA 内存管理的相关参数

参　数　名	参 数 描 述	默　认　值
pga_aggregate_target	合计 PGA 的目标大小	Init.ora 或 spfile 中设置
workarea_size_policy	Workarea 管理策略（MANUAL/AUTO）	AUTO
_smm_advice_enabled	如果为 TRUE，开启 v$pga_advice	TRUE
_smm_trace	关闭或开启跟踪 SQL memory manager	0
_smm_min_size	Auto 模式下最小的 workarea 空间	1024
_smm_max_size	Auto 模式下最大的 workarea 空间（非并行）	5% pga_aggregate_target
_smm_px_max_size	Auto 模式下最大的 workarea 空间（并行）	30% pga_aggregate_target
_smm_bound	覆盖自动计算的 Global Memory bound	0

作者简介

汪海，现任职于国内某大型电子商务网站，专职 Oracle DBA。作者于 2001 年毕业于杭州电子工业学院，毕业后在软件公司做数据库相关开发工作。接触并掌握了多种数据库系统，现专注于 Oracle 的研究，对备份恢复、SQL 优化、Internal 都有相当兴趣。

14 第 14 章　32bit Oracle SGA 扩展原理和 SGA 与 PGA 的制约关系

32bit Oracle 由于位数限制，使得 Oracle 进程只能访问 4GB（2^{32}）以下的虚拟内存地址，在很多时候这是一个很让人头疼的问题，因为空着许多内存而不能使用，而默认情况下 SGA 不能超过 1.7GB。比如 Linux 下有 8GB 内存，却有部分空着不能用。这个时候就要考虑怎样扩展 Oracle 的 SGA。

14.1　如何识别 32bit 的 Oracle

如何识别 32bit 的 Oracle 呢？可以通过以下查询得到。

```
sqlplus> 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
```

如果是 64bit Oracle，在查询结果中一定会显示 64bit 字样，若没有出现，则一定是 32bit Oracle。当然，在 OS 上通过 file oracle 也能看到，例如：

```
 [oracle@ocn2 bin]$ cd $ORACLE_HOME/bin
 [oracle@ocn2 bin]$ file oracle
oracle: setuid setgid ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked
(uses shared libs), not stripped
 [oracle@ocn2 bin]$
```

在某些 OS 上，比如 AIX 上，64bit Oracle 会正常显示信息，32bit 则不正常显示。

在确认了 32bit Oracle 之后，通常情况下 OS 进程只能访问 4GB 以下的空间，Redhat Linux AS 2.1 或者 AS 3.0 版本例外，它们可以提供 VLM（Very Large Memory）功能支持，使得通过转换可以使用 36bit 来标志内存地址，那么就是 2 的 36 次方，理论上最大可支持 64GB 内存访问。这在 Oracle 中，则是通过将内存当作文件来访问的，虚拟一个/dev/shm 的文件系统，这个文件系统是

完全由内存组成的，这样将突破 4GB 的限制。那回过来看看，既然进程可以访问 4GB 以下内存，为何通常 SGA 又是 1.7GB 呢?

14.2 为何存在 1.7GB 的限制

在 OS 中，规定了一个进程在应用程序中，能访问的虚拟内存空间为 0～3GB，而 3GB～4GB 这段虚拟地址空间是保留给 Kernel 使用的。要注意这里强调的是虚拟地址空间，并没有说物理地址空间，也就是说，假设有 8GB 的内存，这 0～3GB 的虚拟地址空间可能出现在 8GB 内存的 3GB～8GB 部分内存段，并不是说是物理内存的 0～3GB 段。而在这 0～3GB 的虚拟地址中，Oracle 又是如何来使用的呢，这是固定好了地址的。

```
++++++++++++++++    4GB
+                         +
+                         +
+                         +
++++++++++++++++    3GB（Kernel）
+                         +
+                         +
+                         +
++++++++++++++++    2GB（Process Stack）
+                         +
+                         +
++++++++++++++++    1.25GB（SGA 起点）
++++++++++++++++    1GB（Oracle 共享库装载起点）
+                         +
+                         +
+                         +
++++++++++++++++    0GB（Oracle Program 装载起点）
```

在这段虚拟地址的分配中，1.25GB 是 SGA 的起点，而进程的私有空间的分配（stack 部分）却是从靠近 3GB 处开始的。也就是实际上 SGA 和进程私有空间是共用了 1.25GB～3GB 这部分的，由于进程私有空间特别小，通常习惯性地认为 SGA 可以达到 1.7GB。进程私有空间有 0.05GB 足够了。从 Oracle 启动开始，或者从任何一用户进程登录开始，所有虚拟地址的分配都已经固定好了，只有用户私有空间还可以扩展。来看一下数据库启动后 PMON 进程（实际上任何一个进程都是一样的）的虚拟地址分配情况。由于一台机器上跑着 2 个数据库，所以来看其中一个，先看看数据库 SGA 的相关信息。

```
[root@ocnsb1 root]# su - oracle
[oracle@ocnsb1 oracle]$ sqlplus "/ as sysdba"
SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jul 26 11:37:23 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, Real Application Clusters, OLAP and Oracle Data Mining options
JServer Release 9.2.0.4.0 - Production

select">sys@OCN>select INSTANCE_NAME   from   v$instance  ;
```

```
INSTANCE_NAME
----------------
roocn1
sys@OCN>show  sga
Total System Global Area  437327188 bytes
Fixed Size                 451924 bytes
Variable Size            301989888 bytes
Database Buffers         134217728 bytes
Redo Buffers              667648 bytes
sys@OCN>
exit
Disconnected from Oracle9i Enterprise Edition Release 9.2.0.4.0 - Production
With the Partitioning, Real Application Clusters, OLAP and Oracle Data Mining options
JServer Release 9.2.0.4.0 - Production
[oracle@ocnsb1 oracle]$ ipcs
------ Shared Memory Segments --------
key       shmid     owner     perms     bytes     nattch    status
0x73a32bdc 131072    oracle    640       457179136  50
0x84cc76ac 163841    oracle    640       1379926016 90
------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status
0x8df96364 622592    oracle    640       64
0x53609d64 753665    oracle    640       504
------ Message Queues --------
key       msqid     owner     perms     used-bytes   messages
 [oracle@ocnsb1 oracle]$
```

这里的共享内存段只有一个，并且就是 SGA 的大小（shmid 为 131072）。这是因为 shnmax 设置太大的缘故。

```
[oracle@ocn2 kernel]$ more  /proc/sys/kernel/shmmax
3221225472
[oracle@ocn2 kernel]$
```

接下来看看 PMON 信息，首先要找到 PMON 进程号，然后去/proc/pid/maps 中看该进程的虚拟地址分配信息。

```
[oracle@ocnsb1 oracle]$ ps -ef|grep pmon
oracle   13655    1  0 Jul24 ?       00:00:00 ora_pmon_roocn1
oracle   13926    1  0 Jul24 ?       00:00:00 ora_pmon_ocn1
oracle   31435 31092  0 11:51 pts/3   00:00:00 grep pmon
[oracle@ocnsb1 oracle]$
 [oracle@ocnsb1 oracle]$ more /proc/13655/maps
08048000-0a4ba000 r-xp 00000000 08:05 681621    /opt/oracle/products/9.2.0/bin/oracle
0a4ba000-0ad54000 rw-p 02471000 08:05 681621    /opt/oracle/products/9.2.0/bin/oracle
0ad54000-0ae07000 rwxp 00000000 00:00 0
```

这部分是 Oracle Program（可执行代码）装载信息，可以看到空间使用了 0～0ae07000，这部分大小不足 256MB。

```
40000000-40016000 r-xp 00000000 08:02 448102    /lib/ld-2.2.4.so
```

这是 Oracle 共享库装载的起点，0x40000000 正好是 1GB。

```
40016000-40017000 rw-p 00015000 08:02 448102    /lib/ld-2.2.4.so
40017000-40018000 rw-p 00000000 00:00 0
40018000-40019000 r-xp 00000000 08:05 308464    /opt/oracle/products/9.2.0/lib/libodmd9.so
```

```
40019000-4001a000 rw-p 00000000 08:05 308464    /opt/oracle/products/9.2.0/lib/libodmd9.so
4001a000-40026000 r-xp 00000000 08:05 308345    /opt/oracle/products/9.2.0/lib/libskgxp9.so
40026000-4002a000 rw-p 0000b000 08:05 308345    /opt/oracle/products/9.2.0/lib/libskgxp9.so
4002a000-40038000 r-xp 00000000 08:05 308461    /opt/oracle/products/9.2.0/lib/libskgxn9.so
40038000-40039000 rw-p 0000d000 08:05 308461    /opt/oracle/products/9.2.0/lib/libskgxn9.so
40039000-4004d000 rw-p 00000000 00:00 0
4004d000-4032c000 r-xp 00000000 08:05 308455    /opt/oracle/products/9.2.0/lib/libjox9.so
4032c000-4043c000 rw-p 002de000 08:05 308455    /opt/oracle/products/9.2.0/lib/libjox9.so
4043c000-4043e000 rw-p 00000000 00:00 0
4043e000-40441000 r-xp 00000000 08:02 448115    /lib/libdl-2.2.4.so
40441000-40442000 rw-p 00002000 08:02 448115    /lib/libdl-2.2.4.so
40442000-40443000 rw-p 00000000 00:00 0
40443000-40465000 r-xp 00000000 08:02 448117    /lib/libm-2.2.4.so
40465000-40466000 rw-p 00021000 08:02 448117    /lib/libm-2.2.4.so
40466000-40475000 r-xp 00000000 08:02 448147    /lib/libpthread-0.9.so
40475000-4047d000 rw-p 0000e000 08:02 448147    /lib/libpthread-0.9.so
4047d000-40490000 r-xp 00000000 08:02 448120    /lib/libnsl-2.2.4.so
40490000-40491000 rw-p 00012000 08:02 448120    /lib/libnsl-2.2.4.so
40491000-40493000 rw-p 00000000 00:00 0
40493000-40494000 r-xp 00000000 08:02 352330    /usr/lib/libaio.so.1
40494000-40495000 rw-p 00000000 08:02 352330    /usr/lib/libaio.so.1
40495000-405ca000 r-xp 00000000 08:02 448111    /lib/libc-2.2.4.so
405ca000-405cf000 rw-p 00134000 08:02 448111    /lib/libc-2.2.4.so
405cf000-405d3000 rw-p 00000000 00:00 0
405d3000-405d4000 r-xp 00000000 08:02 146106    /lib/libredhat-kernel.so.1.0.1
405d4000-405d5000 rw-p 00000000 08:02 146106    /lib/libredhat-kernel.so.1.0.1
405d5000-405f9000 rw-p 00000000 00:00 0
405fa000-40604000 r-xp 00000000 08:02 448136    /lib/libnss_files-2.2.4.so
40604000-40605000 rw-p 00009000 08:02 448136    /lib/libnss_files-2.2.4.so
40605000-40685000 rw-p 00000000 08:02 69445     /dev/zero
40685000-406c6000 rw-p 00000000 00:00 0
```

共享库消耗了不到 20MB 的空间。

```
50000000-6b000000 rw-s 00000000 00:04 131072    /SYSV73a32bdc (deleted)
```

这是 SGA 的起点，0x50000000 表示 1.25GB。

```
6b000000-6b001000 r--s 1b000000 00:04 131072    /SYSV73a32bdc (deleted)
6b001000-6b0a2000 rw-s 1b001000 00:04 131072    /SYSV73a32bdc (deleted)
6b0a2000-6b0a3000 r--s 1b0a2000 00:04 131072    /SYSV73a32bdc (deleted)
6b0a3000-6b400000 rw-s 1b0a3000 00:04 131072    /SYSV73a32bdc (deleted)
```

SGA 虚拟空间分配到这里，通过计算十六进制数，正好和 SGA 大小吻合，131072 就是在 ipcs 查看时的 shmid。

```
bffe5000-bffee000 rwxp ffff8000 00:00 0
bfff0000-bfff1000 r-xs 00000000 08:02 69304     /dev/vsys
```

由于 0xc0000000 正好是 3GB（十六进制数 c=12，4*3=12，0x40000000 表示 1GB），则这里表示进程私有空间的分配的起点。查看 Oracle 任何一个用户登录进程也将发现这样的虚拟地址分配。在这里很容易看出来，Oracle Program 和共享内存库所占用的空间很小，没有必要给那么大，实际上，Oracle Program 给 256MB 足够安全，而共享库给 50MB 也足够安全了，也就是从理论上来讲，可以把 Oracle Program 所需要压缩在 0x10000000 以下，共享库所需要内存压缩在 0x12000000 以下，这样 SGA 的起点就可以提升到 0x12000000（0.3G）。而原来是从 0x5000000（1.25G）开始的，只有大

约 1.7GB 分配给 SGA，现在从 0.3GB 开始分配 SGA 则可以接近 2.7GB，比如分配 2.65GB 内存给 SGA。要实现这个功能，需要重新编译 Oracle Program，降低共享库虚拟内存分配的地址和 SGA 的分配起点位置。0x4000000 这个共享库装载的起点，是由进程的 mapped_base 来决定的。

```
[oracle@ocnsb1 oracle]$ more /proc/13655/mapped_base
1073741824
```

这个大小是 1GB，则意味着共享库的装载从虚拟地址的 1GB 位置开始，如果要降低这个地址，需要在 Oracle 启动之前，也就是用 root 用户把将启动 Oracle 的进程的 mapped_base 降低到 256MB，这样 Oracle 启动之后产生的进程都将继承这个值。

```
su - root echo 268435456 > /proc//mapped_base
```

当然也可以通过一些 shell 来实现 Oracle 用户登录之后自动降低 mapped_base 的功能，这个在 Google 上就能找到了。

SGA 起点从 1.25GB 降低到 0.3GB 则需要重新编译 Oracle Program。必须要强调的是，SGA 的起点是和共享库的起点 mapped_ase 相关的，SGA 的起点至少得大于共享库的起点 0.05GB 以上才是安全的，否则数据库将不能启动或者崩溃。

```
关闭 oracle
  su – oracle
  cd $ORACLE_HOME/rdbms/lib
  修改共享库装载地址的文件定义
  genksms -s 0x12000000 > ksms.s
  编译好目标文件
 make -f ins_rdbms.mk ksms。o
重新编译 oracle 可执行文件
 make -f ins_rdbms.mk ioracle
```

至于 Redhat Linux AS 2.1 以上版本，Oracle VLM 的使用也是比较简单的，Google 上很容易找到的。

在这里要指出一个问题，也是大家在实践中遇到的一个问题，那就是，若 SGA 分配得很大，但没有使用 VLM，几乎很靠近 3GB 的时候，大约只留下 20MB 左右。这样当一个进程进行 hash join，由于 pga_aggregate_target 设置为 1GB，Oracle 默认单个进程使用 PGA 可以达到 pga_aggregate_target * 5% = 50MB，则使得在进行 hash join 的时候出错：

```
ORA-04030: out of process memory when trying to allocate 254476 bytes (hash-join subh,kllcqas:kllsltba)
```

调整 pga_aggregate_target 减小到 400MB，则该查询执行成功。因为没有使用 VLM 的情况下单个进程的内存分配空间必须在 3GB 以下，而 PGA 的分配也属于这个范畴。如果使用 VLM 则 PGA 已经被分配到 4GB 以上部分的虚拟地址，不再有这个问题。在此不再对 VLM 进行过多的阐述，因为使用也比较简单，从原理上来讲就是通过 OS 扩展 32bit 到 36bit，Oracle 使用文件来管理内存，并支持进程访问 4GB 以上部分的虚拟内存。Linux 上这种用法得到推广的根本原因是因为其 64bit Oracle 很少被使用，其他如 Sun OS/HP UNIX/AIX 等都广泛使用 64bit Oracle 了，这些方法也就失去价值了。

下面来介绍 32bit 下 SGA 与 PGA 之间的制约关系。

14.3　32bit 下 SGA 与 PGA 之间的制约关系

在本人的测试中，由于降低了 SGA 的起点（降低 SGA 起点和本章没有实质上的关系，仅仅

因为本人的测试数据库正好是被降低了 SGA 起点而已），从 0x5000000 降低到 0x42000000，也就是说，当没有降低的时候，SGA 的理论极限值是约小于 1.75GB，降低起点后，SGA 的极限值大约是 1.95GB。

```
sys@OCN> show sga
Total System Global Area 2064719084 bytes
Fixed Size                453868 bytes
Variable Size           385875968 bytes
Database Buffers        1677721600 bytes
Redo Buffers              667648 bytes
sys@OCN>
```

这时我的 SGA_MAX_SIZE 的大小为 2064719084。

```
sys@OCN> show parameters sga_max_size
NAME                                 TYPE       VALUE
------------------------------------ ----------- ------------------------------
sga_max_size                          big integer 2064719084
```

实际上，这个时候的 SGA_MAX_SIZE 比真实的 SGA 设置要大一些，这是手工调大的。下面来观察单个数据库进程的 maps。

```
 [oracle@ocnsb2 oracle]$ ps -ef|grep pmon
oracle   18346    1  0 12:16 ?       00:00:00 ora_pmon_roocn
oracle   18535 18493  0 13:04 pts/0   00:00:00 grep pmon

[oracle@ocnsb2 oracle]$ more  /proc/18346/maps
08048000-0a4b9000 r-xp 00000000 08:05 324868    /opt/oracle/products/9.2.0/bin/oracle
0a4b9000-0ad53000 rw-p 02470000 08:05 324868    /opt/oracle/products/9.2.0/bin/oracle
0ad53000-0adec000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 08:02 448102    /lib/ld-2.2.4.so
40016000-40017000 rw-p 00015000 08:02 448102    /lib/ld-2.2.4.so
40017000-40018000 rw-p 00000000 00:00 0
40018000-40019000 r-xp 00000000 08:05 1038540   /opt/oracle/products/9.2.0/lib/libodmd9.so
40019000-4001a000 rw-p 00000000 08:05 1038540   /opt/oracle/products/9.2.0/lib/libodmd9.so
4001a000-40026000 r-xp 00000000 08:05 1038423   /opt/oracle/products/9.2.0/lib/libskgxp9.so
40026000-4002a000 rw-p 0000b000 08:05 1038423   /opt/oracle/products/9.2.0/lib/libskgxp9.so
4002a000-40038000 r-xp 00000000 08:05 1038537   /opt/oracle/products/9.2.0/lib/libskgxn9.so
40038000-40039000 rw-p 0000d000 08:05 1038537   /opt/oracle/products/9.2.0/lib/libskgxn9.so
40039000-4004d000 rw-p 00000000 00:00 0
4004d000-4032c000 r-xp 00000000 08:05 1038531   /opt/oracle/products/9.2.0/lib/libjox9.so
4032c000-4043c000 rw-p 002de000 08:05 1038531   /opt/oracle/products/9.2.0/lib/libjox9.so
4043c000-4043e000 rw-p 00000000 00:00 0
4043e000-40441000 r-xp 00000000 08:02 448115    /lib/libdl-2.2.4.so
40441000-40442000 rw-p 00002000 08:02 448115    /lib/libdl-2.2.4.so
40442000-40443000 rw-p 00000000 00:00 0
40443000-40465000 r-xp 00000000 08:02 448117    /lib/libm-2.2.4.so
40465000-40466000 rw-p 00021000 08:02 448117    /lib/libm-2.2.4.so
40466000-40475000 r-xp 00000000 08:02 448147    /lib/libpthread-0.9.so
40475000-4047d000 rw-p 0000e000 08:02 448147    /lib/libpthread-0.9.so
4047d000-40490000 r-xp 00000000 08:02 448120    /lib/libnsl-2.2.4.so
40490000-40491000 rw-p 00012000 08:02 448120    /lib/libnsl-2.2.4.so
40491000-40493000 rw-p 00000000 00:00 0
40493000-405c8000 r-xp 00000000 08:02 448111    /lib/libc-2.2.4.so
405c8000-405cd000 rw-p 00134000 08:02 448111    /lib/libc-2.2.4.so
```

```
    405cd000-405f4000 rw-p 00000000 00:00 0
    405f5000-405ff000 r-xp 00000000 08:02 448136    /lib/libnss_files-2.2.4.so
    405ff000-40600000 rw-p 00009000 08:02 448136    /lib/libnss_files-2.2.4.so
    40600000-40680000 rw-p 00000000 08:02 69445     /dev/zero
    40680000-406c1000 rw-p 00000000 00:00 0
    42000000-be400000 rw-B 00000000 00:04 524288    /SYSV0676004c (deleted)
    bffe4000-bffee000 rwxp ffff7000 00:00 0
    0a4b9000-0ad53000 rw-p 02470000 08:05 324868    /opt/oracle/products/9.2.0/bin/oracle
    0ad53000-0adec000 rwxp 00000000 00:00 0
    40000000-40016000 r-xp 00000000 08:02 448102    /lib/ld-2.2.4.so
    40016000-40017000 rw-p 00015000 08:02 448102    /lib/ld-2.2.4.so
    40017000-40018000 rw-p 00000000 00:00 0
    40018000-40019000 r-xp 00000000 08:05 1038540   /opt/oracle/products/9.2.0/lib/libodmd9.so
    40019000-4001a000 rw-p 00000000 08:05 1038540   /opt/oracle/products/9.2.0/lib/libodmd9.so
    4001a000-40026000 r-xp 00000000 08:05 1038423   /opt/oracle/products/9.2.0/lib/libskgxp9.so
    40026000-4002a000 rw-p 0000b000 08:05 1038423   /opt/oracle/products/9.2.0/lib/libskgxp9.so
    4002a000-40038000 r-xp 00000000 08:05 1038537   /opt/oracle/products/9.2.0/lib/libskgxn9.so
    40038000-40039000 rw-p 0000d000 08:05 1038537   /opt/oracle/products/9.2.0/lib/libskgxn9.so
    40039000-4004d000 rw-p 00000000 00:00 0
    4004d000-4032c000 r-xp 00000000 08:05 1038531   /opt/oracle/products/9.2.0/lib/libjox9.so
    4032c000-4043c000 rw-p 002de000 08:05 1038531   /opt/oracle/products/9.2.0/lib/libjox9.so
    4043c000-4043e000 rw-p 00000000 00:00 0
    4043e000-40441000 r-xp 00000000 08:02 448115    /lib/libdl-2.2.4.so
    40441000-40442000 rw-p 00002000 08:02 448115    /lib/libdl-2.2.4.so
    40442000-40443000 rw-p 00000000 00:00 0
    40443000-40465000 r-xp 00000000 08:02 448117    /lib/libm-2.2.4.so
    40465000-40466000 rw-p 00021000 08:02 448117    /lib/libm-2.2.4.so
    40466000-40475000 r-xp 00000000 08:02 448147    /lib/libpthread-0.9.so
    40475000-4047d000 rw-p 0000e000 08:02 448147    /lib/libpthread-0.9.so
    4047d000-40490000 r-xp 00000000 08:02 448120    /lib/libnsl-2.2.4.so
    40490000-40491000 rw-p 00012000 08:02 448120    /lib/libnsl-2.2.4.so
    40491000-40493000 rw-p 00000000 00:00 0
    40493000-405c8000 r-xp 00000000 08:02 448111    /lib/libc-2.2.4.so
    405c8000-405cd000 rw-p 00134000 08:02 448111    /lib/libc-2.2.4.so
    405cd000-405f4000 rw-p 00000000 00:00 0
    405f5000-405ff000 r-xp 00000000 08:02 448136    /lib/libnss_files-2.2.4.so
    405ff000-40600000 rw-p 00009000 08:02 448136    /lib/libnss_files-2.2.4.so
    40600000-40680000 rw-p 00000000 08:02 69445     /dev/zero
    40680000-406c1000 rw-p 00000000 00:00 0
    42000000-be400000 rw-B 00000000 00:04 524288    /SYSV0676004c (deleted)
```

这里可以看出单个进程中根据 SGA_MAX_SIZE 预分配了 0x42000000～0xbe400000 这段虚拟内存空间，虽然实际 SGA 并没有这么大，但是正是这段空间的预留，为 9i 中动态增大 SGA 提供了可能。也就是说进程能访问的虚拟地址空间是登录数据库时决定的，这样即使 SGA 真正地发生了变化，只要在 SGA_MAX_SIZE 之内，该进程都可以访问这些虚拟内存空间。

```
    sys@OCN>select                          (to_number('be400000','xxxxxxxxxx')            -
to_number('42000000','xxxxxxxxxx'))/1024/1024 from dual;

     (TO_NUMBER('BE400000','XXXXXXXXXX')-TO_NUMBER('42000000','XXXXXXXXXX'))/1024/1024
    ---------------------------------------------------------------------------
1988

    sys@OCN>
```

```
 bffe4000-bffee000 rwxp ffff7000 00:00 0
[oracle@ocnsb2 oracle]$
```

从这里可以看出进程的私有内存分配的顶点，0xbe400000～0xbffe4000 属于 Oracle 进程可使用的 PGA 的空间大小。

```
sys@OCN> select to_number('bffe4000','xxxxxxxxxx') - to_number('be400000','xxxxxxxxxx')
from dual;
TO_NUMBER('BFFE4000','XXXXXXXXXX')-TO_NUMBER('BE400000','XXXXXXXXXX')
---------------------------------------------------------------------
                                                             29245440
```

从这里可以看出 PGA 能使用的空间大约是 29245440 字节，下面来做一个测试。

```
 sys@OCN> show parameters pga
NAME                                 TYPE       VALUE
-------------------------------- ----------- ------------------------------
pga_aggregate_target              big integer 524288000
```

在 Oracle 中，单个进程的 PGA 的使用遵循原则是小于 min（pga_aggregate_target*5%, 100MB），在这样的设置下，执行对两个大表做 hash join 的查询，结果是成功完成。然后修改 pga_aggregate_target = 1024MB。

```
sys@OCN> alter system set pga_aggregate_target = 1024m;
System altered。
  sys@OCN> show parameters pga
NAME                                 TYPE       VALUE
-------------------------------- ----------- ------------------------------
pga_aggregate_target              big integer 1073741824
```

当再运行这个查询的过程中，同时观察该进程的 PGA 使用量。在 session 1 中执行以下查询：

```
1* select * from company c,member m where c。admin_member_id = m。login_id
alibaba@OCN> /
ERROR:
ORA-04030: out of process memory when trying to allocate 254476 bytes (hash-join
subh,kllcqas:kllsltba)

no rows selected

Execution Plan
```

在 session2 中观察结果（不停地执行下面的查询以观察变化）：

```
sys@OCN>  select spid,USERNAME,PGA_USED_MEM ,PGA_ALLOC_MEM,PGA_FREEABLE_MEM, PGA_MAX_MEM
from v$process;
SPID        USERNAME                PGA_USED_MEM PGA_ALLOC_MEM PGA_FREEABLE_MEM PGA_MAX_MEM
----------- ---------------------- ------------ ------------- --------------- -----------
                                              0             0               0           0
18346       oracle                       147425        476981               0      476981
18348       oracle                       244313        570985               0      570985
18350       oracle                      1304737       1635765               0     1635765
18354       oracle                      2945957       3300193               0     3300193
18356       oracle                       200089        534729               0      534729
18358       oracle                      4368569       4741969               0     4741969
18360       oracle                       151577        551357               0      551357
18362       oracle                       165841        510101               0      510101
18364       oracle                       156625        485493               0      485493
18366       oracle                       157009        485493               0      485493
```

```
18368       oracle                4362441      4748549            0      4748549
18370       oracle                4362441      4715245            0      4715245
18388       oracle                 191285       790553       196608       790553
18406       oracle                29041865     30095501            0     30095501
```

可以发现，执行 hash join 的进程的 PGA 最大达到 29041865 之后，就报出错误 4030 而返回了，基本吻合前面关于 SGA_MAX_SIZE 与进程私有空间虚拟地址的顶点之间的计算。

作者简介

冯春培，网络 ID biti_rainy，曾任 ITPUB Oracle 开发版版主，现任 ITPUB Oracle 管理版版主和超级版主。有丰富的 Oracle 实践经验，对数据库的体系结构、备份恢复、SQL 优化、数据库整体性能优化、Oracle Internal 都有深入研究。

开发出身，对数据库应用设计中如何正确地应用 Oracle 特性以扬长避短具有深刻理解。曾于某电信集成公司负责计费系统的开发，然后成为某系统集成公司的 DBA，再辗转在香港一家跨国公司珠海研发中心担任技术负责人（公司主要产品就是 SQL 与数据库优化工具，产品主要销往欧洲和北美），此后成为自由职业者，独立为客户提供 Oracle 数据库的技术服务和高级性能调整等方面的培训，同时提供 ITPUB 在华南和华东地区的培训。

目前服务于国内某大型电子商务网站，维护系统数据库并提供开发支持。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

更多个人作品：http://blog.itpub.net/biti_rainy

第15章　KEEP池和RECYCLE池

本章先简单介绍了 Buffer Cache 的作用，然后着重说明 KEEP 池和 RECYCLE 池，并结合事例说明了它们的特点和适用情况。

15.1　Oracle 的数据缓冲池

在 Oracle 进行处理过程中，代价最昂贵的就是物理 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。因此，优化 Oracle 的一个重要目标就是尽可能地降低物理 I/O 操作。

Oracle 的 Buffer Cache 用于缓存从磁盘中读取的数据，当 Oracle 需要查找某些信息时，首先会在 Buffer Cache 中寻找，如果找到了，则直接将结果返回。如果找不到，则需要对磁盘进行扫描，Oracle 将在从磁盘扫描得到的结果返回的同时，会把这些数据在 Buffer Cache 中也保留一份，如果下次需要访问相同的数据时，则不需要再次扫描磁盘，而是直接从 Buffer Cache 中读取即可。

当 Buffer Cache 存储空间已满，但是又有新的数据需要存放在 Buffer Cache 中，这时 Oracle 会将这部分新的数据替换掉 Buffer Cache 中的一部分数据。Oracle 通过 LRU 算法（最近最少使用算法）来确定哪些数据要被清除出 Buffer Cache。

Buffer Cache 中的内存由两个链表组成，Write List（写链表）和 Least Recent Used List（最近最少使用链表）。写链表包含那些还没有被写到磁盘上的脏的缓冲。LRU 链表包括空闲缓冲区、目前正在使用的 Pinned Buffer 和还没有移到写链表中的脏缓冲区（Dirty Buffer）。如果 Oracle 访问了缓冲区中的数据，则会把这部分缓冲移到 LRU 链表的 MRU 端（Most Recent Used）。当 Oracle 需要寻找空闲缓冲时，它会从 LRU 链表的 LRU 端开始寻找。在寻找过程中，如果发现了脏的缓冲，就把它移到写链表中。当 Oracle 找到一个空闲缓冲时，就会停止搜索，如果搜索缓冲数量超过了阈值的限制还一直找不到空闲缓冲，则会停止搜索，启动 DBW0 后端进程将一些脏的缓冲写到磁盘上。

如果用户执行的是全表扫描的操作，这些操作产生的数据缓冲不会放到 LRU 的 MRU 端，而是放到 LRU 端。因为 Oracle 认为全表扫描得到的数据只是暂时的需要，这些数据以后被重用的机会很少，应该快速地清除出缓冲区，把空间留给其他更常用的数据。可以在表的级别上改变这种处理方式。在建表的时候指定 Cache 语句会使得这张表全表扫描得到的数据也放到 LRU 链表的

MRU 端。

在调整 SGA 大小时，应当配置合适的缓冲大小，将尽可能多的数据放到内存中，从而减少物理磁盘的扫描。在 9i 中，可以通过调整初始化参数 db_cache_size 的值来修改数据缓冲区的大小。在 8i 中，数据缓冲大小= db_block_size*db_block_buffers，通过调整 db_block_buffers 的大小可以调整数据缓冲区的大小。

15.2 KEEP 池和 RECYCLE 池

如果内存足够大，可以容纳所有的数据，则访问任何数据都可以从内存中直接获得，那么效率肯定是最高的。但是在实际应用当中，经常是数据库的大小达到了几百个 GB 甚至几个 TB，而 Oracle 的可用内存只有几个 GB 大小。缓存区中缓存的数据只能占到整个数据库数据的很小一部分，因此，这就要求必须合理地分配内存的使用。

如果可使用的内存空间比较小，导致数据缓冲区的命中率较低，则可以通过配置 KEEP 池和 RECYCLE 池，把性质不同的表分离到不同的数据缓冲区，以提高命中率，降低批操作对正常访问的影响。

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）来决定。如果在建表或修改表的时候指定 STORAGE（BUFFER_POOL KEEP）或者 STROAGE（BUFFER_POOL RECYCLE）语句，就设置这张表使用 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 8i 中，只能修改 pfile 文件，然后重启数据库，才能使对这两个参数的修改生效。在 9i 中，可以动态修改，不过不能直接增加这两个缓冲区的大小，必须先缩小 SGA 中其他内存缓冲区的大小，然后再增加 db_keep_cache_size 和 db_recycle_cache_size 的值，如下所示：

```
SQL> show parameter cache_size
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------
db_16k_cache_size                    big integer 0
db_2k_cache_size                     big integer 0
db_32k_cache_size                    big integer 0
db_4k_cache_size                     big integer 0
db_8k_cache_size                     big integer 16777216
db_cache_size                        big integer 109051904
db_keep_cache_size                   big integer 0
db_recycle_cache_size                big integer 0

SQL> alter system set db_cache_size = 80m;
系统已更改。

SQL> alter system set db_keep_cache_size = 12m;
系统已更改。

SQL> alter system set db_recycle_cache_size = 8m;
系统已更改。
```

```
SQL> show parameter cache_size
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------
db_16k_cache_size                     big integer 0
db_2k_cache_size                      big integer 0
db_32k_cache_size                     big integer 0
db_4k_cache_size                      big integer 0
db_8k_cache_size                      big integer 16777216
db_cache_size                        big integer 83886080
db_keep_cache_size                    big integer 16777216
db_recycle_cache_size                 big integer 8388608
```

15.2.1　KEEP 池

KEEP 池用来缓存那些经常会被访问的表。KEEP 池使用的缓冲区独立于 DEFAULT 池，因此把最经常使用的表缓存到单独的缓冲区中，使得数据库的其他操作，如执行大量批操作也不会影响到这些在 KEEP 缓冲区中的表，保证访问这些最常用的表的数据时，都可以从内存中直接获得。

```
SQL> COL NAME FORMAT A30
SQL> COL VALUE FORMAT A30
SQL> SHOW PARAMETER DB_KEEP_CACHE_SIZE
NAME                                 TYPE        VALUE
------------------------------------ ----------- --------------------------
db_keep_cache_size                    big integer 16777216

SQL> CREATE TABLE TEST_DEFAULT (COL NUMBER(3)) STORAGE(BUFFER_POOL DEFAULT);
表已创建。

SQL> CREATE TABLE TEST_KEEP (COL NUMBER(3)) STORAGE(BUFFER_POOL KEEP);
表已创建。

SQL> INSERT INTO TEST_DEFAULT VALUES (1);
已创建 1 行。

SQL> INSERT INTO TEST_KEEP VALUES (1);
已创建 1 行。

SQL> COMMIT;
提交完成。

SQL> SET AUTOT ON STAT
SQL> SELECT * FROM TEST_DEFAULT;
      COL
----------
         1

Statistics
----------------------------------------------------------
          0  recursive calls
```

```
         0  db block gets
        21  consistent gets
         0  physical reads
         0  redo size
       371  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)
         1  rows processed

SQL> SELECT * FROM TEST_KEEP;
      COL
----------
         1

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
        21  consistent gets
         0  physical reads
         0  redo size
       371  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)
         1  rows processed

SQL> SHOW PARAMETER DB_CACHE_SIZE
NAME                                 TYPE       VALUE
------------------------------------ ----------- ------------------------------
db_cache_size                         big integer 83886080

SQL> CREATE TABLE TEST_EAT_MEMORY
  2  (
  3  COL1 VARCHAR2(4000),
  4  COL2 VARCHAR2(4000),
  5  COL3 VARCHAR2(4000),
  6  COL4 VARCHAR2(4000),
  7  COL5 VARCHAR2(4000),
  8  COL6 VARCHAR2(4000),
  9  COL7 VARCHAR2(4000),
 10  COL8 VARCHAR2(4000),
 11  COL9 VARCHAR2(4000),
 12  COL10 VARCHAR2(4000)
 13  )
 14  STORAGE(BUFFER_POOL DEFAULT);
表已创建。

SQL> INSERT INTO TEST_EAT_MEMORY
```

```
  2  SELECT RPAD('1', 4000, '1'), RPAD('2', 4000, '2'), RPAD('3', 4000, '3'),
  3  RPAD('4', 4000, '4'), RPAD('5', 4000, '5'),
  4  RPAD('6', 4000, '6'), RPAD('7', 4000, '7'),
  5  RPAD('8', 4000, '8'),  RPAD('9', 4000, '9'),
  6  RPAD('0', 4000, '0')  FROM ALL_OBJECTS
  7  WHERE ROWNUM < 2501;
已创建 2500 行。

Statistics
----------------------------------------------------------
      2043  recursive calls
     49593  db block gets
     17311  consistent gets
        60  physical reads
 104517904  redo size
       616  bytes sent via SQL*Net to client
       798  bytes received via SQL*Net from client
         3  SQL*Net roundtrips to/from client
        14  sorts (memory)
         0  sorts (disk)
      2500  rows processed

SQL> COMMIT;
提交完成。

SQL> SELECT * FROM TEST_DEFAULT;
       COL
----------
         1

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
        21  consistent gets
        16  physical reads
         0  redo size
       371  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)
         1  rows processed

SQL> SELECT * FROM TEST_KEEP;
       COL
----------
         1

Statistics
----------------------------------------------------------
         0  recursive calls
```

```
          0  db block gets
         21  consistent gets
          0  physical reads
          0  redo size
        371  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)
          1  rows processed
```

在上面的例子中，建立了两张表。TEST_DEFAULT 指定默认的 DEFAULT 池，TEST_KEEP 指定了 KEEP 池。分别插入一条数据，然后把 AUTOTRACE 的统计信息打开，对这两张表进行查询。由于刚刚执行了 insert 语句，这两条数据都存放在各自的缓冲区中，因此查询的物理读（Physical Reads）为 0。接着查看 db_cache_size 的值，发现大小在 80MB 左右。下面构造一个较大的批操作。在这个批操作中，插入的数据大小约为 10×4KB×2500=100MB，通过产生的 redo size 的大小也可以看出，大约插入了 100MB 的数据，这个大小已经超出了数据缓冲区（DEFAULT 池）的大小。执行完批操作以后，对两张表再次进行查询。结果很明显，由于 KEEP 和 DEFAULT 池彼此独立，对 TEST_KEEP 的查询的物理读仍然为 0，而对 TEST_DEFAULT 的查询则包含了 16 个物理读。

上面的例子可以看出，使用 KEEP 池可以保证那些指定 KEEP 池的表不受其他表的影响。用户可以通过查询 V$BH 视图来找到经常被使用的表，根据表的使用频繁程度来确定是否指定 KEEP 池。

```
SQL> COL OBJECT_NAME FORMAT A30
SQL> SELECT O.OBJECT_NAME, COUNT(*)
  2  FROM DBA_OBJECTS O, V$BH BH
  3  WHERE O.OBJECT_ID = BH.OBJD
  4  AND O.OWNER !='SYS'
  5  GROUP BY O.OBJECT_NAME
  6  HAVING COUNT(*) > 100
  7  ORDER BY COUNT(*) DESC;
OBJECT_NAME                    COUNT(*)
------------------------------ ----------
ORD_ORDER_ITEM                  18180
CON_LIST_ITEM                  16178
ORD_ORDER                      15018
ORD_HIT_COMM                    5392
CAT_AREA_MEDICARE               3987
CAT_REGION_MERCHANDISE          2737
CAT_ORG                          497
SMS_QUEUE                        371
CON_LIST_ITEM_SHARE              252
USR_USER                         195
```

确定好使用 KEEP 池的表以后，可以根据这些表的实际大小之和来计算 KEEP 缓冲区的大小。由于这个大小可以比较准确地计算出来，因此可以对这些表使用 Cache，以保证即使采用全表扫描得到的数据也可以被缓冲。

通常情况下，并不追求 KEEP 池的命中率达到 100%，如果命中率为 100%，则说明给 KEEP

池的内存空间过大，有部分内存被浪费。即使 KEEP 池中缓存的都是最经常访问的表，这些访问操作也不大可能访问到表中的所有数据。因此，可以适当地减少 KEEP 池的内存分配，使 KEEP 池的命中率维持在接近 100%的数值。将这部分内存分配给其他缓冲区，可以得到更高的效率。

可以采用下面的 SQL 语句来计算 KEEP 池的命中率：

```
SQL> SELECT NAME, PHYSICAL_READS, DB_BLOCK_GETS, CONSISTENT_GETS,
  2  1 - (PHYSICAL_READS / (DB_BLOCK_GETS + CONSISTENT_GETS)) "Hit Ratio"
  3  FROM V$BUFFER_POOL_STATISTICS
  4  WHERE NAME = 'KEEP';
NAME             PHYSICAL_READS DB_BLOCK_GETS CONSISTENT_GETS  Hit Ratio
---------------- -------------- ------------- --------------- ----------
KEEP                          0            33              66          1
```

15.2.2　RECYCLE 池

RECYCLE 池用来缓存那些不希望保存在内存中的表。例如，很少进行扫描或访问的表。如果应用程序以一种随机的方式访问一张比较大的表，这些被缓冲的数据在被清除出内存之前，很少会有机会再次被访问。这些数据存放在缓冲区中，不仅会浪费内存空间，而且可能把其他一些有可能被访问的数据清除出去。这些数据没有必要保存在缓冲区中，可以通过使用 RECYCLE 池来避免这些数据对其他数据的影响。

调整参数 db_recycle_cache_size 的大小来设置 RECYCLE 池。一般来说，不需要给 RECYCLE 池很大的内存空间，因为 RECYCLE 池中的数据没有什么被缓存的价值。设置较小的缓冲区可以将更多的内存留给 KEEP 池和 DEFAULT 池。但是，如果缓冲区太小的话，数据可能在事务结束之前就从内存中被清除了，这会导致额外的性能问题。

由于在 9i 中，缓冲区的大小可以动态调整，配合对表的缓冲区的修改，可以在执行批操作前将表改为 RECYCLE 池，这样可以将批操作对系统缓存命中率的影响降至最低，具体如下：

```
SQL> SELECT * FROM TEST_DEFAULT;
      COL
----------
       1

Statistics
----------------------------------------------------------
        0  recursive calls
        0  db block gets
       21  consistent gets
        0  physical reads
        0  redo size
      371  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)
        1  rows processed

SQL> TRUNCATE TABLE TEST_EAT_MEMORY;
```

```
表已截掉。

SQL> ALTER TABLE TEST_EAT_MEMORY STORAGE (BUFFER_POOL RECYCLE);
表已更改。

SQL> INSERT INTO TEST_EAT_MEMORY
  2  SELECT RPAD('1', 4000, '1'), RPAD('2', 4000, '2'), RPAD('3', 4000, '3'),
  3  RPAD('4', 4000, '4'), RPAD('5', 4000, '5'),
  4  RPAD('6', 4000, '6'), RPAD('7', 4000, '7'),
  5  RPAD('8', 4000, '8'),  RPAD('9', 4000, '9'),
  6  RPAD('0', 4000, '0')  FROM ALL_OBJECTS
  7  WHERE ROWNUM < 2501;
已创建 2500 行。

Statistics
----------------------------------------------------------
      1568  recursive calls
     49596  db block gets
     17021  consistent gets
         9  physical reads
  106567004  redo size
       624  bytes sent via SQL*Net to client
       798  bytes received via SQL*Net from client
         3  SQL*Net roundtrips to/from client
         6  sorts (memory)
         0  sorts (disk)
      2500  rows processed

SQL> COMMIT;
提交完成。

SQL> SELECT * FROM TEST_DEFAULT;
       COL
----------
         1

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
        21  consistent gets
         0  physical reads
         0  redo size
       371  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)
         1  rows processed
```

在这个例子中，将进行批操作的表改为 RECYCLE 池，在批操作执行完成后，发现 TEST_DEFAULT 表的数据仍然可以在 DEFAULT 池中找到。这种方法屏蔽了批操作对系统的影响。

15.3　小结

对于大多数的系统而言，使用 DEFAULT 池就够了。但是如果内存空间相对较小，或者对系统中表的使用情况有比较清晰的认识，则可以通过配置 KEEP 池和 RECYCLE 池来细化内存的分配，提高数据缓冲区的命中率，降低批操作对系统的影响。

虽然 KEEP 池、RECYCLE 池和 DEFAULT 池用来缓存不同类型的数据，但是它们的机制是相同的，它们采用的都是 LRU 算法。如果 KEEP 池分配的内存不足，那么也会有部分数据被清除出内存；如果 RECYCLE 池的内存分配足够，也可以保证其中的数据全部缓存。从本质上讲，KEEP 池和 RECYCLE 池并没有什么区别，只是名字不同罢了。

另外，如果给 KEEP 池或 RECYCLE 分配的内存大小不合适，不但不会提高性能，而且会造成性能的下降。以 KEEP 池为例，内存分配小了，这些经常被访问的数据就会有部分被清除出内存，导致命中率的降低。如果内存分配过大，则导致 DEFAULT 池内存要相应地减少，DEFAULT 池不仅包括用户部分数据，而且也包括数据字典的缓冲。因此，DEFAULT 池内存的不足，必然导致整个系统性能的下降。而且，由于真实环境中，所有的表的大小都处于变化之中，因此，需要经常对不同缓冲区的命中率进行检查，并随时调整缓冲区的大小以满足数据不断变化的需要。

参考信息

1．Oracle 9i Database Concepts
http://tahiti.oracle.com
2．Oracle 9i Database Performance Tuning Guide and Reference
http://tahiti.oracle.com

作者简介

杨廷琨，由于偶然的机会，接触了 Oracle，并对 Oracle 产生了较大的兴趣，自毕业以来一直从事 Oracle 开发和管理工作。2001 年 12 月加入 ITPUB，很喜欢这里的氛围，经常出入 Oracle 管理和 Oracle 开发版块，目前任 Oracle 数据库管理版版主。

任职于某电子商务公司。对 Oracle 数据库的安装、优化和 SQL 的调优积累了一定的经验。希望能和大家一起学习探讨 Oracle 技术。

E-mail:yangtingkun@itpub.net

第 16 章　深度分析数据库的热点块问题

16.1　热点块的定义

数据库的热点块，简单来讲，就是在极短的时间内对少量数据块进行了过于频繁的访问。定义看起来是很简单的，但实际在数据库中，要去观察或者确定热点块的问题，却不是那么简单了。要深刻地理解数据库是怎么通过一些数据特征来表示热点块的，就需要了解一些数据库在这方面处理机制的特性。

16.2　数据缓冲区的结构

当开始查询的时候，进程首先去数据缓冲区中查找是否存在查询所需要的数据块，如果没有，就去磁盘上把数据块读到内存中来。在这个过程中，涉及到数据缓冲区中 LRU 链的管理（8i 开始以接触点计数为标准衡量 buffer 冷热从而决定 buffer 是在 LRU 的冷端还是热端），关于这部分内容，从 Oracle Concepts 中就能得到详尽的文档，所以这里不准备去论述这部分内容，这也不是本章的重点。下面要讨论的重点是，到底进程是如何快速定位到自己所想要的 block 的，或者如何快速确定想要的 block 不在内存中而去进行物理读的。

随着硬件的发展，内存越来越大，Cache Buffer 也越来越大，用户如何才能在大容量的内存中迅速定位到自己想要的 block？总不能去所有 buffer 中遍历吧！在此数据库引入了 hash 的概念（Oracle 中快速定位信息总是通过 hash 算法来得到的，比如快速定位 SQL 是否在 shared pool size 中存在就是通过 hash value 来实现的，也就是说 shared pool size 中的对象也是通过 hash table 来管理的），了解数据结构的基本知识就可以知道，hash 的一大重要功能就是快速地查找。举个最简单的例子，假设有一个 hash table 就是一个二维数组 a[200][100]，现在有 1000 个无序数字，要从这 1000 个数字里面查找某个值是否存在，或者说当接收到某个数字的时候必须判断是否已经存在，当然，可以遍历这 1000 个数字，但这样的效率就很低。但现在考虑这样一种方法，那就是把 1000 个数字除以 200，根据其余数，放在 a[200][100]里面（假设相同余数的最大数量不超过 100），余数就是数组的下标。这样，平均来说一个数组 a[i]里面可能有 5 个左右的数字。当要去判别一个数字是否存在的时候，对这个数字除以 200（这就是一个最简单的 hash 算法），再把余数 i 作

为下标去数组 a[i]中查找，大约进行 5 次查找就能判别其是否已经存在，这样通过开辟内存空间 a[200][100]来换取了时间（当然 hash 算法的选取和 hash table 的大小是一个很关键的问题）。

明白了基本的 hash 原理之后，再来看 Oracle 的 block 管理。数据库为这些 block 也开辟了 hash table，假设是 a，则在一维上的数量是由参数_db_block_hash_buckets 来决定的，也就是存在 hash table a[_db_block_hash_buckets]，从 Oracle 8i 开始，_db_block_hash_buckets= db_block_buffers*2。而一个 block 被放到哪个 buckets 里面，则是由 block 的文件编号、块号做 hash 算法来决定的（x$bh.dbarfl、x$bh.dbablk 对应 block 的文件属于表空间中的相关编号和 block 在文件中的编号，x$bh 是所有 Cache Buffer 的 header 信息，通过表格的形式可以查询），而 bucket 里面就存放了这些 buffers 的地址。这样当要访问数据时，可以获得 segment 的 extent（可以通过 dba_extents 查看到，详细的信息来源这里不做探讨），自然知道要访问的文件编号和 block 编号，根据文件和 block 编号可以通过 hash 算法计算出 hash bucket，然后就可以去 hash bucket 里面寻找 block 对应的 buffer。

除此之外，为了维护对这些 block 的访问和更改，Oracle 还提供了一种 latch 来保护这些 block。因为要避免不同的进程随意地径直并发修改和访问这些 block，否则就很可能会破坏 block 的结构。latch 是数据库内部提供的一种维护内部结构的一种低级锁，latch 的生存周期极短（微秒以下级别），进程加 latch 后快速地进行某个访问或者修改动作，然后释放 latch（关于 latch 不作过多的阐述，那可能需要相当的篇幅才能阐述清楚）。这种 latch 数量是通过参数_db_block_hash_latches 来定义的，一个 latch 对应地保护了多个 buckets。从 8i 开始，这个参数的 default 规则如下。

当 cache buffers 少于 2052 buffers。则

```
_db_block_hash_latches = power(2,trunc(log(2, db_block_buffers - 4) - 1))
```

当 cache buffers 多于 131075 buffers，则

```
_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，则

```
_db_block_hash_latches = 1024
```

通过这个规则可以看出，一个 latch 大约可以维护 128 个左右的 buffers。由于 latch 使得对 block 的操作串行化（9i 中有改进，读与读可以并行，但读与写、写与写依然要串行），就很容易想到，如果大量进程对相同的 block 进程进行操作，必然在这些 latch 上造成竞争，也就是说必然形成 latch 的等待。这在宏观上就表现为系统级的等待。明白了这些原理，就为后面在数据库中的诊断奠定了基础。

16.3 如何确定热点对象

如果经常关注 Statspack 报告，就会发现有时候出现 Cache Buffer Chains 的等待。这个 Cache Buffer Chains 就是_db_block_hash_latches 所定义的 latch 的总称，通过查询 v$latch 也可得到：

```
sys@OCN>select latch#,name,gets,misses,sleeps from v$latch where name like 'cache
buffer%';

   LATCH# NAME                          GETS     MISSES    SLEEPS
---------- ------------------------------ ---------- ---------- ----------
        93 cache buffers lru chain              54360446     21025       238
        98 cache buffers chains               6760354603   1680007     27085
```

```
        99 cache buffer handles                    554532        6        0
```

在这个查询结果里，可以看到记录了数据库启动以来的所有 Cache Buffer Chains 的 latch 的状况，GETS 表示总共有这么多次请求，MISSES 表示请求失败的次数（第一次加锁不成功），而 sleeps 表示 miss 后第一次 spin 过程中依然不成功而休眠的次数，通过 SLEEPS 可以大体知道数据库中 latch 的竞争是否严重，这也间接地表征了热点块的问题是否严重。由于 v$latch 是一个聚合信息，用户并不能获得哪些块可能存在频繁访问，那就要来看另一个 view 信息，那就是 v$latch_children，v$latch_children.addr 记录的就是这个 latch 的地址。

```
sys@OCN>select addr,LATCH#,CHILD#,gets,misses,sleeps from v$latch_children
 2  where name = 'cache buffers chains'  and rownum < 21;

ADDR       LATCH#    CHILD#    GETS      MISSES    SLEEPS
-------- ---------- ---------- ---------- ---------- ----------
91B23B74        98      1024   10365583      3957        33
91B23374        98      1023    5458174       964        25
91B22B74        98      1022    4855668       868        15
91B22374        98      1021    5767706       923        22
91B21B74        98      1020    5607116       934        31
91B21374        98      1019    9389325      1111        25
91B20B74        98      1018    5060207       994        31
91B20374        98      1017   18204581      1145        18
91B1FB74        98      1016    7157081       920        23
91B1F374        98      1015    4660774       922        22
91B1EB74        98      1014    6954644       976        32
91B1E374        98      1013    4881891       970        19
91B1DB74        98      1012    5371135       971        28
91B1D374        98      1011    5154497       990        26
91B1CB74        98      1010    5013796       936        18
91B1C374        98      1009    5667446       939        25
91B1BB74        98      1008    4673421       883        14
91B1B374        98      1007    4589646       986        17
91B1AB74        98      1006   10380781      1020        20
91B1A374        98      1005    5142009      1110        19

20 rows selected.
```

到此，可以根据 v$latch_child.addr 关联到对应的 x$bh.hladdr（这是 buffer header 中记录的当前 buffer 所处的 latch 地址），通过 x$bh 可以获得块的文件编号和 block 编号。

```
sys@OCN>select dbarfil,dbablk
   from x$bh
  where hladdr in
   (select addr
   from (select addr
      from v$latch_children
      order by sleeps desc)
        where rownum < 11);

  DBARFIL    DBABLK
---------- ----------
       4      6498
      40     14915
```

```
      15     65564
      28     34909
      40     17987
       1     24554
       8     21404
      39     29669
      28     46173
      28     48221

……
```

由此就打通了 Cache Buffers Chains 和具体 block 之间的关系，那再继续下来，知道了 block，就需要知道究竟是哪些 segment。这个可以通过 dba_extents 来获得，具体如下：

```
select distinct a.owner,a.segment_name from
dba_extents a,
(select dbarfil,dbablk
from x$bh
where hladdr in
  (select addr
  from (select addr
     from v$latch_children
     order by sleeps desc)
     where rownum < 11)) b
where a.RELATIVE_FNO = b.dbarfil
and a.BLOCK_ID <= b.dbablk and a.block_id + a.blocks > b.dbablk;


OWNER                     SEGMENT_NAME               SEGMENT_TYPE
------------------------ ------------------------  ------------------
ALIBABA                   BIZ_SEARCHER               TABLE
ALIBABA                   CMNTY_USER_MESSAGE         TABLE
ALIBABA                   CMNTY_VISITOR_INFO_PK      INDEX
ALIBABA                   COMPANY_AMID_IND           INDEX
ALIBABA                   COMPANY_DRAFT              TABLE
ALIBABA                   FEEDBACK_POST              TABLE
ALIBABA                   IM_BLACKLIST_PK            INDEX
ALIBABA                   IM_GROUP                   TABLE
ALIBABA                   IM_GROUP_LID_IND           INDEX
ALIBABA                   MEMBER                     TABLE
ALIBABA                   MEMBER_PK                  INDEX
ALIBABA                   MLOG$_SAMPLE               TABLE

……
```

还有另外一种方式：

```
select object_name
from dba_objects
where data_object_id in
(select obj
from x$bh
where hladdr in
  (select addr
  from (select addr
```

```
      from v$latch_children
      order by sleeps desc)
      where rownum < 11)) ;


OBJECT_NAME
-----------------------------------
I_CCOL2
RESOURCE_PLAN$
DUAL
FGA_LOG$
AV_TRANSACTION
COMPANY_DRAFT
MEMBER
SAMPLE
SAMPLE_GROUP
VERTICAL_COMPONENT
MEMBER_PK
SAMPLE_GROUP_PK
IM_BLACKLIST_PK
IM_CONTACT
IM_GROUP
CMNTY_USER_MESSAGE
CMNTY_VISITOR_INFO_PK
IM_OFFLINEMSG_TID_IND
OFFER
OFFER_PK
OFFER_EMAIL_IND
OFFER_DRAFT
CMNTY_USER_MESSAGE_TD_BSM_IND
CMNTY_MESSAGE_NUM_PK
BIZ_EXPRESS_MEMBER_ID_IND

……
```

到这里基本能找到热点块对应的对象。但实际上还有另外一个途径来获取这些信息，那就是与 x$bh.tch 相关的一种方法。从 8i 版本开始，Oracle 提供了接触点（touch count）来作为 block 是冷热的标志，在满足一定条件的情况下，block 被进程访问一次 touch count 就增加 1，到某个标准之后被移动到 LRU 热端（关于 touch count 在这里不做详细介绍）。那在短时间内从某种意义上讲，touch count 大的 block 可能暗示着在当前某个周期内被访问次数比较多。

```
select distinct a.owner,a.segment_name,a.segment_type from
dba_extents a,
(select dbarfil,dbablk
from (select dbarfil,dbablk
   from x$bh order by tch desc) where rownum < 11) b
where a.RELATIVE_FNO = b.dbarfil
and a.BLOCK_ID <= b.dbablk and a.block_id + a.blocks > b.dbablk;

OWNER                      SEGMENT_NAME                SEGMENT_TYPE
-------------------------  --------------------  ---------------------
ALIBABA                     CMNTY_USER_MESSAGE            TABLE
```

```
ALIBABA                    MEMBER_PK                      INDEX
ALIBABA                    OFFER_DRAFT_GMDFY_IND          INDEX
```

同上面一样还有以下方法：

```
select object_name
from dba_objects
where data_object_id in
(select obj
from (select obj
   from x$bh order by tch desc) where rownum < 11) ;
OBJECT_NAME
--------------------------------------------------
DUAL
MEMBER_PK
SAMPLE_GROUP_PK
CMNTY_USER_MESSAGE_TD_BSM_IND
OFFER_DRAFT_MID_GMDFY_IND
OFFER_MID_GPOST_IND
OFFER_DRAFT_PK
MEMBER_GLLOGIN_IND
OFFER_MID_STAT_GEXPIRE_IND
SAMPLE_MID_STAT_IND

10 rows selected.
```

到这里，寻找热点块和热点对象的工作算是完成了，但还并没有解决问题。

16.4 热点问题的解决

热点块和热点对象都找到了，但是该怎么来解决这个问题呢？一般来说，热点块会导致 Cache Buffers Chains 竞争等待，但并不是说 Cache Buffer Chains 一定是因为热点块而起，在特别情况下有可能是因为 latch 数量的问题而导致的，也就是一个 latch 管理的 buffers 数量太多而导致竞争激烈。但是 latch 数量一般是不会轻易去设置的，这是 Oracle 的隐藏参数。

实际上最有效的办法，是从优化 SQL 入手，不良的 SQL 往往带来大量的不必要的访问，这是造成热点块的根源。比如本该通过全表扫描的查询却走了索引的 range scan，这样将带来大量的对块的重复访问，从而形成热点问题。或者不当地走了 nested loops 的表连接，也可能对非驱动表造成大量的重复访问。那么在这个时候，目标就是找出这些 SQL 并尝试优化。根据 Statspack 报告中的 SQL 列表，如果是通过 dba_extents 确定的热点对象而不是通过 dba_objects 确定的，则可以通过查找出的热点 segment 转换为对应的表，对于非分区的索引，index_name 就是 segment_name，通过 dba_indexes 很容易就找到对应的 table_name，对于分区表和分区索引，也能通过 dba_tab_partition 和 dba_ind_partitions 找到 segment 和 table 的对应关系。再通过这些 table 到 Statspack 报告中去找相关的 SQL。

```
select sql_text
from stats$sqltext a,
(select distinct a.owner,a.segment_name,a.segment_type from
dba_extents a,
(select dbarfil,dbablk
```

```
from (select dbarfil,dbablk
  from x$bh order by tch desc) where rownum < 11) b
where a.RELATIVE_FNO = b.dbarfil
and a.BLOCK_ID <= b.dbablk and a.block_id + a.blocks > b.dbablk) b
where a.sql_text like '%'||b.segment_name||'%' and b.segment_type = 'TABLE'
order by  a.hash_value,a.address,a.piece;

SQL_TEXT
----------------------------------------------------------------
SELECT SEQ_SMS_TRANSACTION.nextval FROM DUAL
SELECT SEQ_BIZ_EXPRESS.nextval FROM DUAL
SELECT bizgroup.seq_grp_post.NextVal FROM DUAL
SELECT SEQ_SAMPLE.nextval FROM DUAL
SELECT bizgroup.seq_grp_user.NextVal FROM DUAL
SELECT SEQ_BIZ_SEARCHER.nextval FROM DUAL
SELECT SEQ_OFFER_DRAFT.nextval FROM DUAL
select seq_Company_Draft.NextVal from DUAL
SELECT SEQ_SAMPLE_GROUP.nextval FROM DUAL
SELECT SEQ_CMNTY_USER_MESSAGE.nextval FROM DUAL
SELECT SYSDATE FROM DUAL
select seq_News_Forum.NextVal from DUAL
SELECT SEQ_SMS_USER.nextval FROM DUAL
select seq_Biz_Member.NextVal from DUAL
select seq_Pymt_Managing.NextVal from DUAL
E= '+08:00' NLS_DUAL_CURRENCY = '$' NLS_TIME_FORMAT = 'HH.MI.SSX
SELECT SEQ_COMPANY_DRAFT.nextval FROM DUAL
SELECT 1 FROM DUAL
select seq_offer_draft.NextVal from DUAL
select seq_Biz_Express_Category.NextVal from DUAL

20 rows selected.
```

当然这里是从 Statspack 搜集的 stats$sqltext 中去找的（可以在 Statspack 的文本报告中去找），实际上，可以直接在当前数据库中的 v$sqlarea 或者 v$sqltext 里面去找到这些 SQL，然后来尝试优化。

```
select sql_text
from v$sqltext a,
(select distinct a.owner,a.segment_name,a.segment_type from
dba_extents a,
(select dbarfil,dbablk
from (select dbarfil,dbablk
  from x$bh order by tch desc) where rownum < 11) b
where a.RELATIVE_FNO = b.dbarfil
and a.BLOCK_ID <= b.dbablk and a.block_id + a.blocks > b.dbablk) b
where a.sql_text like '%'||b.segment_name||'%' and b.segment_type = 'TABLE'
order by  a.hash_value,a.address,a.piece;
SQL_TEXT
----------------------------------------------------------------
SELECT NULL FROM DUAL FOR UPDATE NOWAIT
SELECT SEQ_SMS_TRANSACTION.nextval FROM DUAL
SELECT SEQ_BIZ_EXPRESS.nextval FROM DUAL
SELECT SEQ_IM_GROUP.nextval FROM DUAL
```

```
SELECT SEQ_SAMPLE.nextval FROM DUAL
='DD-MON-RR HH.MI.SSXFF AM TZR' NLS_DUAL_CURRENCY='$' NLS_COMP='
SELECT SEQ_BIZ_SEARCHER.nextval FROM DUAL
SELECT SEQ_OFFER_DRAFT.nextval FROM DUAL
SELECT SEQ_SAMPLE_GROUP.nextval FROM DUAL
DD-MON-RR HH.MI.SSXFF AM TZR' NLS_DUAL_CURRENCY='$' NLS_COMP='BI
SELECT SEQ_CMNTY_USER_MESSAGE.nextval FROM DUAL
SELECT SYSDATE FROM DUAL
SELECT SEQ_SMS_USER.nextval FROM DUAL
IMESTAMP_TZ_FORMAT='DD-MON-RR HH.MI.SSXFF AM TZR' NLS_DUAL_CURRE
SELECT SEQ_COMPANY_DRAFT.nextval FROM DUAL
SELECT 1 FROM DUAL
SELECT USER FROM DUAL
SELECT DECODE('A','A','1','2') FROM DUAL

18 rows selected.
```

当然对于热点的表或者索引来说，如果小的话，除了优化 SQL 外，还可以考虑 cache 在内存中，这样可能降低物理读，提高 SQL 运行速度（这并不会减少 Cache Buffer Chains 的访问次数），对于序列，可以对序列多设置一些 cache。如果是并行服务器环境中的索引对象，并且这个索引是系列递增类型，可以考虑反向索引（关于反向索引这里就不过多地做介绍了）。

16.5 热点块的其他相关症状

在数据库中还可能存在一些其他方面的热点块症状，通过 v$waitstat 的等待可以看出一些端倪，v$waitstat 是根据数据缓冲区中各种 block 的类型（x$bh.class）而分类统计的等待状况。

```
sys@OCN>select * from v$waitstat;

CLASS                    COUNT        TIME
------------------ ---------- ----------
data block             1726977    452542
sort block                   0         0
save undo block              0         0
segment header              40        11
save undo header             0         0
free list                    0         0
extent map                   0         0
1st level bmb              611       112
2nd level bmb               42        13
3rd level bmb                0         0
bitmap block                 0         0
bitmap index block           0         0
file header block           13        92
unused                       0         0
system undo header         111        28
system undo block            7         0
undo header               2765       187
undo block                 633       156
```

比如在 ASSM 表空间出现之前，由于 freelist 的存在，如果表经常被并发的进程 DML，则可能

存在大量的 data block 的等待，或者有 freelist 的等待。那么这个时候发现这样的 segment 之后需要考虑增加 freelist 数量。再比如经常发生长时间的 DML 的表被频繁地访问，这样将会造成过多的对回滚段中块的访问，这时可能 undo block 的等待会比较多。那么可能需要控制 DML 的时间长度或者想办法从应用程序入手来解决问题。如果是 undo header 的等待比较多，没使用 undo tablespace 之前，可能需要考虑增加回滚段的数量。

16.6　小结

本章从热点块的原理入手，详细地由 Oracle 的内部结构特征开始介绍了热点块的产生和表现特征，进而阐述了诊断热点对象和找出造成热点对象的 SQL 的方法，并从解决热点问题的角度提供了思路。

作者简介

冯春培，网络 ID biti_rainy，曾任 ITPUB Oracle 开发版版主，现任 ITPUB Oracle 管理版版主和超级版主。有丰富的 Oracle 实践经验，对数据库的体系结构、备份恢复、SQL 优化、数据库整体性能优化、Oracle Internal 都有深入研究。

开发出身，对数据库应用设计中如何正确地应用 Oracle 特性以扬长避短具有深刻的理解。曾于某电信集成公司负责计费系统的开发，然后成为某系统集成公司的 DBA，再辗转在香港一家跨国公司珠海研发中心担任技术负责人（公司主要产品就是 SQL 与数据库优化工具,产品主要销往欧洲和北美），此后成为自由职业者，独立为客户提供 Oracle 数据库的技术服务和高级性能调整等方面的培训，同时提供 ITPUB 在华南和华东地区的培训。

目前服务于国内某大型电子商务网站，维护系统数据库并提供开发支持。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

更多个人作品：http://blog.itpub.net/biti_rainy

第 17 章　Shared Pool 原理及性能分析

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。

本章就 Shared Pool 的原理及性能问题进行深入探讨，并通过几个具体案例介绍相关性能问题的诊断及解决方法。

17.1　Shared Pool 的基本原理

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 进程负责解析和优化所有 SQL 和 PL/SQL 代码。典型的，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在于：实现共享。如果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，并且客户端要不停的获取 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，可是在性能优化中会发现，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，就应该有意识地提高自己的代码水平，以期减少数据库的压力。这也应该是对开发人员的最初和最基本的要求。

17.2　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，这并不是一句话就能予以定论的，下面试图通过一些内部分析，从内部原理来回答这个问题。

17.2.1 基本知识

下面用到了 Shared Pool 的转储，所以首先了解一下与其相关的命令。

可以通过以下命令转储 Shared Pool 共享内存的内容：

```
SQL> alter session set events 'immediate trace name heapdump level 2';
Session altered.

本测试中引用的两个 trace 文件:

9i:

SQL> @gettrcname

TRACE_FILE_NAME
------------------------------------------------------------------------------
/opt/oracle/admin/hsjf/udump/hsjf_ora_24983.trc

8i:

SQL> @gettrcname

TRACE_FILE_NAME
------------------------------------------------------------------------------
/usr/oracle8/admin/guess/udump/guess_ora_22038.trc
```

其中 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。

转储的内容被记录在一个 trace 文件中，这个 trace 文件可以在 undmp 目录下找到。

为了比较 Oracle 8i 以及 Oracle 9i 的不同，文中引用了两个版本的跟踪文件。

这里，gettrcname.sql 是用来获取 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
/
```

有兴趣的朋友也可以在我的网站（http://www.eygle.com）找到相关脚本及更多的详细说明。

17.2.2　Shared Pool 的 Free List 管理

Shared Pool 通过 Free Lists 管理 free 内存块（Chunk），free 内存块按不同 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 17-1 对 Shared Pool 的 Free List 管理进行说明：

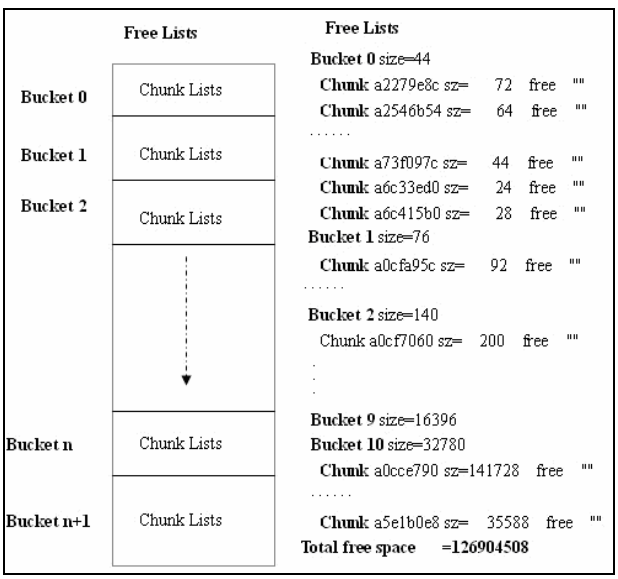

图 17-1　Shared Pool 自由列表

在 Oracle 8i 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）：

```
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

本章所引用的所有 trace 文件都可以在本人的个人网站（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。

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 Share Pool 碎片过多。

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 的机制在这里不过多介绍，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 的竞争。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及资源，Latch 的争用此时就会成为一个非常严重的问题。

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。这就是大家曾经听说过的：过大的 Shared_Pool 会带来管理上的负担。

在 Oracle 9i 中，Oracle 改写了 Shared Pool 管理的算法，来看一下 Oracle 9i 中的处理方式：

```
 [oracle@jumper oracle]$ sqlplus "/ as sysdba"
SQL*Plus: Release 9.2.0.3.0 - Production on Wed Aug 18 22:13:07 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> 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
```

观察以上输出可以看到，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 bytes 递增。
- Bucket 200～249 容纳 size 以 64 bytes 递增。

从 Bucket 249 开始，Oracle 各 Bucket 步长进一步增加：

- Bucket 249：4012～4107 = 96
- Bucket 250：4108～8203 = 4096
- Bucket 251：8204～16395 = 8192
- Bucket 252：16396～32779 = 16384

- Bucket 253：32780～65547 = 32768
- Bucket 254：>=65548

对比 Oracle 8i，在 Oracle 9i 中 Shared Pool 管理最为显著的变化是，对于数量众多的 Chunk，Oracle 增加了更多的 Bucket 来管理。

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。

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 1 个增加到现在的 7 个。如果系统有 4 个或 4 个以上的 CPU，并且 shared_pool_size 大于 250MB，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 "_kghdsidx_count"=2 scope=spfile;

System altered.

SQL> startup force;
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.
Database opened.

SQL> col KSPINM for a20
SQL> col KSPPSTVL for a20
SQL> select a.ksppinm, b.ksppstvl
  2  from   x$ksppi a, x$ksppsv b
  3  where  a.indx = b.indx and   a.ksppinm = '_kghdsidx_count';

KSPPINM                              KSPPSTVL
```

```
---------------------------------------------------------------- --------------------
_kghdsidx_count                          2

SQL>

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：3316003，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 中出现的麻烦。

在论坛上经常看到很多人对于 Shared_Pool 的建议一直就是 200MB～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。

另外，在这里，想提一下 Buffer Cache 的管理方式，Buffer Cache 的管理也涉及两个重要的 Latch：Cache Buffers LRU Chain 和 Cache Buffers Chains。

其中 Cache Buffers LRU Chain 用于管理 Buffer Cache 中内存块的分配和使用，并按照 LRU 算法进行老化，当新数据需要读到 Buffer Cache 中时，该 Latch 需要被持有以寻找和锁定可用的内存块。而 Cache Buffers Chains 用于 Buffre Cache 中的数据访问（pined），当需要访问数据时，该 Latch 需要被持有。

可以看到，过于频繁的数据访问几乎肯定会引起 Cache Buffers Chains 的竞争，也就是通常所说的热点块的竞争。Oracle 通过两个数据结构来管理这部分内存：Hash Buckets 和 Hash Latchs。

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：

```
next_prime (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*db_block_buffers，而_db_block_hash_latchs 的数量也发生了变化：

当 cache buffers 少于 2052 buffers，则

```
_db_block_hash_latches = power(2,trunc(log(2, db_block_buffers - 4) - 1))
```

当 cache buffers 多于 131075 buffers，则

```
_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，则

```
_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_latchs 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。

下面通过图 17-2 来简要描述这个变化（图中省略了一些内容）。

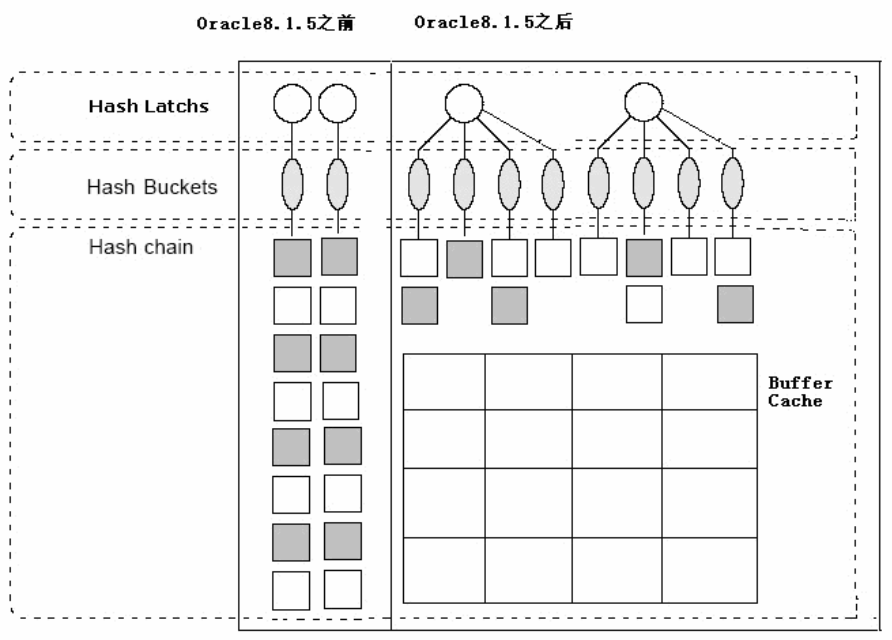

图 17-2 Buffer Cache 管理示意图

Buffer Cache 的这一系列改进和 Shared Pool 的改进极为类似，通过减少每个 Bucket 管理的 Block 的数量，从而提高了管理效率，降低了闩锁的竞争。

关于这部分内容，biti_rainy 网友在他的文章“ 深度分析数据库的热点块问题 ”中有详细阐述，在此不再更多涉及。

17.2.3 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。

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
```

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。

- recr

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。

- freeabl

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，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 空间分配和分割。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$KSMSP 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 的状态：

```
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。

17.3 诊断和解决 ORA-04031 错误

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。关于 ORA-04031 错误，网络上相关的参考文档已经很多，这里只做简要说明。

17.3.1 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 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 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".
```

17.3.2　内存泄露

在 Oracle9iR2 之前的很多 ORA-04031 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 所致（也就是在下面列表中可以看到的 Bug：1397603）。

表 17-1 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 **17-1**　　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

续表

BUG 号	描　　述	解 决 方 法	修正版本
<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

17.3.3 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 biti_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)。

17.3.4　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。

17.3.5　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，不建议设置得更低。

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 并且 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。

17.3.6　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 脚本创建所需的对象。

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。

17.3.7　模拟 ORA-04031 错误

Oracle9iR2 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$sgastat
        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_before-l_after),'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               3927884
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$sgastat
    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
```

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。

17.4　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。

锁定主要有三种模式：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 有三种模式：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 的报告中，可能看到以下等待事件：

```
Top 5 Wait Events
~~~~~~~~~~~~~~~~~                                       Wait    % Total
Event                                            Waits   Time (cs)   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 都是用户所关心的，接下来就研究一下这几个等待事件。

17.4.1　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

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

SQL*Plus: Release 9.2.0.3.0 - Production on Mon Sep 6 14:16:57 2004

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

Connected to an idle instance.

SQL> startup
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.
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：

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

SQL*Plus: Release 9.2.0.3.0 - Production on Mon Sep 6 16:13:43 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> exec calling
```

此时 calling 开始执行。

session 2：

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

SQL*Plus: Release 9.2.0.3.0 - Production on Mon Sep 6 16:14:16 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> 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    0    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    0    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    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$KGLOB 视图就可以得到对象的具体信息：

注　意

X$KGLOB 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$KGLOB
where KGLHDADR ='52D6730C'
/


ADDR  KGLHDADR 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 pining 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.KGLPNSES,b.KGLPNHDL,
    b.kGLPNLCK, b.KGLPNMOD, b.KGLPNREQ
    from v$session a,x$kglpn b
    where a.saddr=b.kglpnuse and b.kglpnhdl = '52D6730C' and b.KGLPNMOD<>0
    /

      SID USERNAME    PROGRAM         ADDR     KGLPNADR KGLPNUSE KGLPNSES 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         P2 P2RAW
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, kglhdpar, 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.kglpnses, b.kglpnhdl, b.kglpnlck, b.kglpnmod, b.kglpnreq
  FROM v$session a, x$kglpn 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$session a, x$kglpn 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_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-llll=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-llll=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。

17.4.2 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：

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

```
SQL*Plus: Release 9.2.0.3.0 - Production on Tue Sep 7 17:05:25 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> 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                    0 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。

17.5　诊断案例一

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
         2 pmon timer                             300 0000012C
         1 rdbms ipc message                      300 0000012C
         4 rdbms ipc message                      300 0000012C
         6 rdbms ipc message                   180000 0002BF20

       SID EVENT                                 P1 P1RAW
---------- ----------------------------------------------------------------
        18 rdbms ipc message                    6000 00001770
       102 rdbms ipc message                    6000 00001770
       311 rdbms ipc message                    6000 00001770
       194 rdbms ipc message                    6000 00001770
       178 rdbms ipc message                    6000 00001770
         3 log file parallel write                 1 00000001
        13 log file sync                        2705 00000A91
        16 log file sync                        2699 00000A8B
       104 log file sync                        2699 00000A8B
       308 log file sync                        2694 00000A86
       262 log file sync                        2705 00000A91

       SID EVENT                                 P1 P1RAW
---------- ----------------------------------------------------------------
       172 log file sync                        2689 00000A81
       169 log file sync                        2705 00000A91
       108 log file sync                        2694 00000A86
        38 log file sync                        2707 00000A93
        34 db file scattered read                 63 0000003F
         5 smon timer                            300 0000012C
        27 SQL*Net message to client         1413697536 54435000
        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

       SID EVENT                          P1 P1RAW
---------- ----------------------------------------------------------
       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 Latche 的数量受一个隐含参数_kgl_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：

⑴ 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[parse count（total）-parse count（hard）] / 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_SCHEM
A_ID,ADDRESS,HASH_VALUE
from v$sqlarea where version_count >1000;
  2

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,AREAID,IFIDDEST,IFIDSRC,
ADDRSRC
,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,PLANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,SMLEN,SM
CONT) 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 scotth 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,AREAID,IFIDDEST,IFIDSRC,
ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,PLANID,FEETYPE,FEEVALUE,DATACODING,FLAGS,S
MLEN,SMCONT) values

(:b0,:b1,:b2,:b3,:b4,:b5,:b6,:b7,:b8,:b9,:b10,:b11,:b12,:b13,:b14,:b15,:b16,:b17,:b18,:b19,:b
20,:b21,:b22)
  hash=ece9cab2 timestamp=09-09-2004 12:51:29
  namespace=CRSR flags=RON/TIM/PN0/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
        4 c1589040  bfd17a18 c35f1388
        5 c1589040  bfd17a34 c77401bc
        6 c1589040  bfd17a50 c4092838
        7 c1589040  bfddb310 c6cd5258
        8 c1589040  bfddb32c c63c6650
        9 c1589040  bfddb348 c7e4e3d0
       10 c1589040  bfddb364 c4c4b110
       11 c1589040  bfddb380 c5950348
       12 c1589040  bfddb39c c6c33aa4
```

```
          13 c1589040  bfddb3b8 c672b0bc
    ........................................
    .....ignore losts of child cursor here.....
    ........................................
         7001 bf595bc8  c641fba0 c6467890
         7002 bf595bc8  c641fbbc c3417168
         7003 bf595bc8  c641fbd8 c3417bb0
         7004 bf595bc8  c641fbf4 c2fdccbc
         7005 bf595bc8  c641fc10 c7f7ca50
         7006 bf595bc8  c641fc2c c7f508ec
         7007 bf595bc8  c641fc48 c268d8d8
         7008 c641fcb8  c641fc64 bec61ed8
         7009 c641fcb8  c641fc80 c4a6cc5c
         7010 c641fcb8  c641fc9c c1a8aa34
         7011 c641fcb8  c0ae4ea0 c0ae4ddc
         7012 c641fcb8  c0ae4ebc bd55fe60
         7013 c641fcb8  c0ae4ed8 c226914c
         7014 c641fcb8  c0ae4ef4 c51dd2e0
         7015 c641fcb8  c0ae4f10 c480c468
         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_ID,PARSING_SCHEMA_ID,ADDRE
SS,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
             2       9160 CHOOSE         238150              36                36 C82AF1C8 3974744754
             3       9127 CHOOSE         238150              36                36 C82AF1C8 3974744754
             4       8109 CHOOSE         238150              36                36 C82AF1C8 3974744754
             5       4386 CHOOSE         238150              36                36 C82AF1C8 3974744754
             6       4913 CHOOSE         238150              36                36 C82AF1C8 3974744754
             7       3764 CHOOSE         238150              36                36 C82AF1C8 3974744754
             8       3287 CHOOSE         238150              36                36 C82AF1C8 3974744754
             9       3156 CHOOSE         238150              36                36 C82AF1C8 3974744754
    .....
          7015          1 CHOOSE         238150              36                36 C82AF1C8 3974744754
```

```
          7016         1 CHOOSE         238150            36              36 C82AF1C8 3974744754
          7017         0 CHOOSE         238150            36              36 C82AF1C8 3974744754

    CHILD_NUMBER EXECUTIONS OPTIMIZER_ OPTIMIZER_COST PARSING_USER_ID PARSING_SCHEMA_ID ADDRESS
HASH_VALUE
    ------------ ---------- --------- ------------ --------------- ---------------- ------ ----------
          7018       9396 NONE                              0               0 C82AF1C8 3974744754
          7019       5008 CHOOSE         237913            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_ 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_ 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：

```
    _sqlexec_progression_cost = 0
```

这个参数的具体含义为：SQL execution progression monitoring cost threshold，即 SQL 执行进度监控成本阀值。

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。

_sqlexec_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_sqlexec_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) valu
es (: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,i
fiddest,servicecode,planid,feetype,feevalue,smcont,submittimes,submitdate,submit
time,respdate,resptime,reptdate,repttime,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,M
SGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,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)

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)

insert into sms_statusrept (reptdate,addruser,msgid_gw,repttime,statustype,msgid
_stus,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,msgt
ime,ifiddest,msqkey,servicecode,planid,feetype,feevalue,smcont,submittimes,submi
tdate,submittime,msgstate_resp,errorcode_resp,msgstate_rept,errorcode_rept) valu
es (: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,M
SGSTATUS,AREAID,IFIDDEST,IFIDSRC,ADDRSRC,ADDRDEST,ADDRFEE,ADDRUSER,SERVICECODE,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,repttime,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。

17.6　诊断案例二

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$KGLOB
      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.kglpnses, 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

      SID USERNAME
---------- ------------------------------
PROGRAM                                          ADDR    KGLPNADR KGLPNUSE
------------------------------------------------ -------- -------- --------
KGLPNSES KGLPNHDL KGLPNLCK   KGLPNMOD   KGLPNREQ
-------- -------- -------- ---------- ----------

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.iftype = 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_dsql := 'truncate table iptt_pm_all';
 EXECUTE IMMEDIATE v_dsql;
```

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。

17.7　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。

参考信息

Eygle	关于 Shared Pool 深入探讨的系列文章 http://www.eygle.com/internal/shared_pool-4.htm 关于绑定变量的 Peeking http://www.eygle.com/sql/Peeking.of.User-Defined.Bind.Variables.htm
Biti_rainy	深度分析数据库的热点块问题 http://blog.csdn.net/biti_rainy/archive/2004/07/06/learn_oracle_20040706_1.aspx 关于 cursor_sharing 的研究 http://blog.itpub.net/post/330/1648
Steve Adams	关于_db_block_hash_buckets 等的论述 http://www.ixora.com.au/q+a/0010/13050341.htm http://www.ixora.com.au/newsletter/2000_11.htm http://www.ixora.com.au/newsletter/2000_07.htm
MetaLink	Note: 22908.1 What are Latches and What Causes Latch Contention Note: 146599.1 Diagnosing and Resolving Error ORA-04031 Note: 122793.1 How to find the session holding a library cache lock Note: 1020008.6 SCRIPT FULLY DECODED LOCKING SCRIPT Note: 34579.1 WAITEVENT: "library cache pin" Reference Note

作者简介

盖国强，网名 eygle，ITPUB Oracle 管理版版主，ITPUB 论坛超级版主，曾任 ITPUB MS 版主。CSDN eMag Oracle 电子杂志主编。

曾任职于某国家大型企业，服务于烟草行业，开发过基于 Oracle 数据库的大型 ERP 系统，属国家信息产业部重点工程。同时负责 Oracle 数据库管理及优化，并为多家烟草企业提供 Oracle 数据库管理、优化及技术支持。

目前任职于北京某电信增值业务系统提供商企业，首席 DBA，负责数据库业务。管理全国 30 多个数据库系统。项目经验丰富，曾设计规划及支持中国联通增值业务等大型数据库系统。

实践经验丰富，长于数据库诊断、性能调整与 SQL 优化等。对于 Oracle 内部技术具有深入研究。

高级培训讲师，培训经验丰富，曾主讲 ITPUB DBA 培训及 ITPUB 高级性能调整等主要课程。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

可以在 http://www.eygle.com 上找到关于作者的更多信息。

第四篇
诊断案例篇

本篇共分 7 章，主要内容如下：

18 第 18 章　一次性能调整过程总结

本章通过总结笔者对生产数据库的一次性能调整过程，简要介绍了数据库调整与优化的通用过程以及在特定情况的调整方法。

18.1　系统环境

生产数据库的系统环境如下。

- 平台：Windows 2000 Server SP3
- 数据库：Oracle 9.0.1.1.1
- 应用：典型的 OLTP 应用

18.2　基本的调优过程

在高峰时段，系统高负载的情况下用 Statspack 每小时做一次系统的快照。注意 Statspack 报告中的 Top 5 Wait Events。Statspack 调优工具的具体用法可以参考 eygle 写的 Statspace 使用指南（http://www.itpub.net/117182.html）。

下面简要介绍几种常见等待事件的产生原因以及通常的解决方法。

18.2.1　db file scattered read

1. 产生原因

该等待时间在数据库会话等待多块 IO 读取结束的时候产生，比如全表扫描和快速索引扫描。数据库同时读取初始化参数 DB_FILE_MULTIBLOCK_READ_COUNT 指定的块数并把这些数据块离散地分布在数据库缓冲区中。

2. 诊断方法

在会话级，查询视图 v$session_wait 时如果有该事件存在，那么该视图中的 P1 表示文件编号，P2 表示块编号，P3 表示读取的块数。在实例级，通过查询视图 v$filestat 并指定条件

BLKS_READ/READS > 1（大于 1 表示有多个多块读取正在发生）可以判断出数据库正在读取哪些文件。

通过下面的查询可以搜索出哪些会话正在执行全表、快速索引扫描：

```
SELECT sid, total_waits, time_waited
   FROM v$session_event
  WHERE event='db file scattered read'
    and total_waits>0
  ORDER BY 3,2;
```

3. 通常的解决方法

常用的解决方法包括以下：

(1) 在合适的字段上建立索引把表的访问方式从全表扫描变为索引扫描可以有效地降低物理 IO。

(2) 对于大表，在合适的字段，比如年月、地区编码上建立分区把全表扫描变成分区扫描以减少物理 IO。

(3) 把需要经常扫描的数据库表放在 KEEP 池同样会有效地降低物理 IO。

18.2.2 db file sequential read

1. 产生原因

该等待事件通常意味着一次 I/O 读取请求的结束。该等待事件与 db file scattered read 的区别请参考 www.itpub.net 的帖子以及 biti_rainy 网友的 blog。一次 sequential 读取通常是单个块的读取，但偶尔的在读取多块的时候也会看到 sequential 读取。

2. 诊断方法

在会话级，查询视图 v$session_wait 时如果有该事件存在，那么该视图中的 P1 表示文件编号 FILE#，P2 表示块编号 BLOCK#，P3 表示读取的块数 BLOCKS，也就是数据库在 FILE#上第 BLOCK# 开始读取的数据库块数。在实例级，通过 Statspack 报告中的 FILE IO 和 TABLESPACE IO 部分可以判断出读取最频繁的表空间或数据文件。

通过下面的查询可以搜索出哪些会话正在执行顺序读取：

```
 SELECT sid, total_waits, time_waited
   FROM v$session_event
  WHERE event='db file sequential read'
    and total_waits>0
  ORDER BY 3,2;
```

3. 通常的解决方法

常用的解决方法包括以下：

(1) 块读取通常是不可避免的，但可以通过最小化不必要的 IO 以避免不必要的块读取。

(2) 检查使用了不合适的索引扫描的 SQL，优化 SQL 语句。

(3) 可能的话，加大 SGA 中的数据库缓存池以缓冲更多的数据库表块和索引块。

(4) 重新组织数据库表，比如使用 export、import 或者 CTAS 以使表有更好的聚集，关于数据库表的聚集因子 cluster_factor，可以参考 biti_rainy 网友的 blog。

（5）判断表是否适合分区以减少需要查询的数据块总数。

18.2.3　Enqueue

1. 产生原因

enqueues 是在读取各种不同的数据库资源时在其上产生的锁，该等待意味着在访问同样的数据库资源时需要等待其他会话已经获取的锁。

2. 诊断方法

在会话级，查询视图 v$session_wait 时如果有该事件存在，那么该视图中的 P1 表示锁类型和模式；P2 表示锁 ID1，以十进制表示的 enqueue 名称的 ID1；P3 表示锁 ID2，以十进制表示的 enqueue 名称的 ID2。

实际的等待时间依赖于锁的类型，在等待超时后数据库会检查已获取锁的会话，如果会话仍然存活那么其他会话会继续等待。

通过查询视图 v$lock 可以获取锁的相关信息，其中 TYPE 列表示锁类型，ID1 表示锁的 ID1，ID2 表示锁的 ID2：

```
SELECT TYPE,ID1,ID2,DECODE(request,0,'Holder: ','Waiter: ')||sid session
  FROM V$LOCK
  WHERE (id1, id2, type) IN
  (SELECT id1, id2, type FROM V$LOCK WHERE request>0)
  ORDER BY id1, request;
```

在实例级，通过查询视图 v$sysstat 中的 enqueue waits 统计信息和 Statspack 报告中的 Statistics 部分可以诊断该事件实际发生的等待次数。通过查询视图 v$enqueue_stat 可以诊断是哪些 enqueues 导致了等待：

```
SELECT     eq_type "Lock",
          total_req# "Gets",
          total_wait# "Waits",
          cum_wait_time
   FROM V$enqueue_stat
WHERE Total_wait# > 0;
```

3. 通常的解决方法

基于不同的锁类型有不同的解决方法。经常发生的等待类型为：

- TX Transaction Lock：通常是由于应用的原因造成锁等待。
- TM DML Enqueue：通常是由于应用原因，部分是因为在外键约束上没有建立索引而导致的。
- ST Space Management Enqueue：通常是由于过多的空间管理（比如因为 extent 过小而导致 extent 的频繁分配/大量的排序再导致临时段的分配）而产生的。

18.2.4　Latch Free

1. 产生原因

该等待事件意味着进程正在等待其他进程已持有的 latch。

2. 诊断方法

在会话级，查询视图 v$session_wait 时如果有该事件存在，那么该视图中的 P1 表示 latch 地址，也就是进程正在等待的 latch 地址。P2 表示 latch 编号，对应于视图 v$latchname 中的 latch#。P3 表示为了获得该 latch 而尝试的次数。

在实例级，Statspack 报告有关于 latch 活动的信息，这些信息同样可以通过查询视图 v$latch 获得：

```
SELECT latch#, name, gets, misses, sleeps
   FROM v$latch
  WHERE sleeps>0;
```

对于常见的 latch 等待，通常的解决方法如下。

- Share Pool Latch：在 OLTP 应用中应该更多地使用绑定变量以减少该 latch 的等待。
- Library Cache Latch：同样需要通过优化 SQL 语句，使用绑定变量减少该 latch 的等待。

回到本例，下面是 StatsPack 报告中的 Top 5 Timed Events 信息：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                 % Total
Event                                          Waits    Time (s) Ela Time
---------------------------------------------- ------------ ----------- --------
db file scattered read                     5,439,418     92,452
db file sequential read                    3,535,182     63,612
enqueue                                        2,377      6,937
PX Deq: Txn Recovery Start                     1,075      2,201
async disk IO                                 74,763        851
        -------------------------------------------------------------
Wait Events for DB: SIMIS  Instance: simis  Snaps: 141 -142
-> s  - second
-> cs - centisecond -    100th of a second
-> ms - millisecond -   1000th of a second
-> us - microsecond - 1000000th of a second
-> ordered by wait time desc, waits desc (idle events last)
```

通过 Statspack 报告，基本上可以把问题定位在数据库执行了太多的全表扫描以及存在太多的锁等待。当然，有可能是由于表、索引统计信息的不准确而导致 CBO 选择了错误的执行计划。鉴于在该数据库出现性能问题之前没有经常性地定时收集数据库对象的统计信息，因此决定首先在中午客户停止办理业务的空闲时候使用 dbms_stats 包收集统计信息。而且由于用户每天晚上 11 点到第二天上午 9 点都不会有业务，故在晚上 11 点启动定时任务执行过程：

```
dbms_stats. gather_schema_stats(OWNNAME=>'<SchemaName>', DEGREE=>4,CASCADE=>TRUE);
```

收集相关 schema 的统计信息以保证 CBO 拥有合适的统计信息来产生合适的执行计划。如果认为当前 SQL 语句的执行计划是最优的，可以使用 Oracle 提供的执行计划稳定性特性，以使 SQL 的执行计划在变化的环境中仍然保持稳定，更多的内容可以参考 biti_rainy 的 Blog。

抽取 Statspack 报告中的 SQL ordered by Gets for DB、SQL ordered by Reads for DB 部分并对其中的 SQL 语句进行调整。同时可以通过对用户会话设置跟踪来定位有问题的 SQL 语句，用过程 dbms_support.start_trace_in_session 对用户会话设置跟踪，相应地用过程 dbms_support.stop_trace_in_session 取消对用户会话的跟踪。dbms_support 工具包可以通过在数据库服务器上执行以下脚本生成：

```
$ORACLE_HOME/rdbms/admin/dbmssupp.sql(Unix 平台)
```

```
%ORACLE_HOME%\rdbms\admin\dbmssupp.sql(Windows 平台)
```

使用实用工具 TKPROF 格式化跟踪文件：

```
tkprof <trace file name> <output file name> sys=no sort=prsdsk，exedsk，fchdsk
```

通过 TKPROF 格式化后，输出文件中的一部分，如下所示：

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        0     0.00       0.00          0          0          0          0
Execute     17     0.01       0.01          0          0          0          0
Fetch       17     0.01       0.20         17         68          0         17
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       34     0.03       0.22         17         68          0         17
```

其中有以下几个列：

- call：数据库处理 SQL 语句的各个阶段，分为 Parse、Execute 和 Fetch。
- count：SQL 语句的执行时间（以秒为单位）。
- cpu：使用的 CPU 时间。
- elapsed：CPU 时间加上 OS 执行上下文切换、服务中断等所花费的时间。
- disk：物理读取 Oracle 块的数量。
- query：以一致的方式从 SGA 中读取 Oracle 块的数量。
- current：需要读取的数据块已经在 SGA 中存在时该值加 1。
- rows：处理的行数。

通常来说 query＋current 等于 SQL 语句总的逻辑 I/O。

通过分析格式化后的输出文件获得其中效率低下的 SQL 语句，然后就可以有针对性地对这些 SQL 语句进行调整了。

一般可以分析 SQL 语句中的 where 子句部分，在表上加上合适的索引来消除全表扫描。注意调整初始化参数 db_file_multiblock_read_count 和 optimizer_index_cost_adj，这两个参数影响 CBO 选择的执行计划，如果希望获取最快的响应速度以及使 CBO 更倾向选择使用索引，可以加提示/*+ first_rows*/。关于这两个初始化参数以及 FIRST_ROWS 优化器模式的更多内容请参考第 26 章 “ CBO 成本计算初探 ”。

经上述调整后，基本上消除了系统中大部分长时间的操作等待，应用程序明显加快。但客户抱怨在应用程序中查询某个报表的时候非常慢，一般都需要 2 个小时以上。对应用程序源码分析后，发现该报表的 SQL 是执行一个嵌套视图查询，该嵌套视图的结构大致如下：

```
create or replace view vw_v1
as
select c1,c2,c3,c4
from t1
/
create or replace view vw_v2
as
select c1,c4,sum(c2),sum(c3)
from vw_v1
group by c1,c4
/
create or replace view vw_v3
as
select *from vw_v2
```

```
union
select *from vw_other
/
```

客户端查询的时候执行以下：

```
select *
from vw_v3
where c1 = :b1
/
```

其中 t1 表是个分区表，目前大概有 4000 万行，每月大概增长 1000 万行。对 SQL 语句分析后认为性能问题主要是由于视图 vw_v2 中对 t1 表的“ group by c1,c4 ”操作造成的，使用 Oracle 的包调整如下：

```
create or replace view vw_v2
as select c1,sum(c2),sum(c3)
from vw_v1
where c1 = pg_arg.sf_get_c1()
group by c1,c4
/
```

其中 pg_arg 是自定义的 PL/SQL 包，主要的代码如下：

```
-- 包定义
CREATE OR REPLACE PACKAGE pg_arg IS
-- 其他变量
-- 对应 c1 列
v_c1     CHAR (10);
-- 其他过程/函数
…
-- 获取 c1 的函数
FUNCTION sf_get_c1 RETURN CHAR;
-- 设置 c1 的过程
PROCEDURE sp_set_c1 (p_c1 IN CHAR);
   PRAGMA restrict_references (sf_get_c1, WNDS, WNPS);
END;
/
-- 包体
CREATE OR REPLACE PACKAGE BODY pg_arg
IS
-- 其他函数和过程
……
   -- 取得 c1 的函数
   FUNCTION sf_get_c1
     RETURN CHAR
   IS
   BEGIN
     RETURN v_c1;
   END;
-- 设置 c1 的过程
   PROCEDURE sp_set_c1 (p_c1 IN CHAR)
   IS
   BEGIN
     v_c1 := p_c1;
   END;
```

```
END;
/
```

在执行该查询前首先用 pg_arg.sp_set_c1(c1)过程设置 c1 参数，从而达到在 group by 前加入 where c1 = pg_arg.sf_get_c1()条件而限定 group by 的记录数。通过这样的调整，就大大提高了查询的性能，将该报表的查询耗时减少到 10 分钟以内。

用户反映在打印一张报表的时候，如果数据量不大（400 人以内）的情况 1～2 分钟就可以把结果查询出来，但如果数据量在 1000 人以上时，查询就变得很慢，需要 10 分钟以上。检查报表的 SQL 语句后发现 SQL 语句带有 order by 子句，初步估计是由于 sort_area_size 参数设置过少而导致大数据量排序的情况下在临时表空间（而不是内存）中执行排序，从而引起查询性能低下。检查初始化参数后发现 PGA 是手工管理，鉴于数据库版本是 9.0.1.1.1，出于稳定性的考虑决定不使用 PGA 自动管理，通过把 sort_area_size 从 512KB 变为 2MB 并重启数据库后发现在大数据量情况下报表的查询速度得到了明显提高。值得一提的是，由于该报表的生成一天只有 1～2 次，不定时的 Statspack 报告检查不出这种“额外”的情况。

用户仍在抱怨从 Excel 报表倒盘到数据库（主要是插入、删除操作）时非常缓慢，200 人的数据量需要 2 个小时才能导入到数据库中。首先试着在 SQL*Plus 里面执行插入操作，观察到在性能最差的时候大概要 30 秒才能插入一条记录。通过

```
alter table modify partition <partition name> freelists <number>;
```

增大空闲列表的数目后感觉对性能的提高微乎其微。同时在插入的时候加/*+ append*/提示，对性能的提高也是很有限，原来 2 个小时的操作在 1 小时 50 分钟完成，基本上没有明显的性能提高。对分区表的结构分析后，发现在该表上建立了全局分区索引，而该全局分区索引中的各个分区所在的表空间均为同一个表空间。删除该全局分区索引并建立普通全局索引后 insert 语句在 SQL*Plus 中可以即时响应。关于全局分区索引和全局普通索引的更多内容请参考 Oracle 文档“Oracle Concepts”。

为了进一步提升插入性能，在插入数据的时候使用批量插入的方式进行插入，批量提取批量插入的更多内容，详细内容可以参考笔者的另一篇文章“关于 Update 语句的调整”。

首先建立临时表，其目的只是为了初始化 Nested Table：

```
prompt -- 临时表,插入 6 条记录
drop table temp_table
/
create table temp_table
 (ny char(6))
/
insert into temp_table values('000000');
insert into temp_table values('000000');
insert into temp_table values('000000');
insert into temp_table values('000000');
insert into temp_table values('000000');
insert into temp_table values('000000');
commit;
-- 相关的存储过程如下:
create or replace sp_insert is
……
   /*********************************
先定义 Nested Table 变量
然后把数据一次性 fetch 出来
```

```
最后用 forall 一次性插入到表中
*********************************/
   TYPE t_jzny IS TABLE OF char(6); -- 数组类型
   b_ny      t_ny;               -- Nested Table 变量
   v_rowcount   NUMBER;           -- 半年的记录数(取值 1-6)
l_jzny      DATE;              -- 日期型变量
   CURSOR cur_bulk
   IS
      SELECT ny
        FROM <table>
WHERE ROWNUM <= 6;-- 6 表示半年内的 6 条记录一次批量插入
……
-- 初始化数组
   OPEN cur_bulk;

   FETCH cur_bulk
   BULK COLLECT INTO b_ny;

   CLOSE cur_bulk;
-- 执行其他业务操作
     ……
     -- 获取半年内的记录数
v_rowcount := 0;
WHILE l_jzny <= TO_DATE (v_jzny, 'yyyymm')
LOOP
   -- 把年月加入到数组中
   b_ny (  v_rowcount + 1) := TO_CHAR (l_jzny, 'yyyymm');
   -- 计数加 1
   v_rowcount :=   v_rowcount + 1;
   -- 处理下一个月份
   l_jzny := ADD_MONTHS (l_jzny, 1);
END LOOP;
    -- 批量插入
FORALL v_i IN 1 .. v_rowcount
   Insert into <table_name>
    ……
ny,
……
   values
   (
   ……
   b_ny(v_i),
   ……
   )
--------------------------

  End;
/
```

经以上调整后，200 人左右的倒盘操作可以从原来的 2 个小时提高到 20 分钟以内，收到立竿见影的效果。

18.3　小结

性能调整一般是通过 Statspack 生成报告，配合使用 dbms_support 包跟踪用户会话并用 TKPROF 分析生成的跟踪文件来定位效率低下的 SQL 语句。对于查询语句的调整，可以加入合适的索引，添加合适的 Hint 来调整查询语句。对于 DML 语句的调整，可以通过 9i 提供的跟踪索引使用的特性：

```
alter index <index_name> monitoring usage;
```

来获取没有使用的索引并删除这些索引以提高 DML 语句的性能，特别地，对于 insert 操作可以通过加入/*+ append*/提示、批量插入等手段进行调整。

参考信息

1．oracle 文档
http://tahiti.oracle.com

2．eygle，启用 AutoTrace 功能
http://www.eygle.com/faq/AutoTrace.htm

3．eygle，如何获得跟踪文件名称
http://www.eygle.com/faq/How.To.Get.Tracefile.Name.htm

4．eygle，使用 DBMS_SUPPORT 包
http://www.eygle.com/faq/DBMS_SUPPORT.htm

5．eygle，Statspack 安装配置使用说明
http://www.itpub.net/117182.html

6．biti_rainy，what is the db file sequential read ？
http://blog.itpub.net/post/330/3102

7．biti_rainy，what is the clustering_factor ？
http://blog.itpub.net/post/330/2970

8．biti_rainy，关于执行计划的稳定性
http://blog.itpub.net/post/330/1611

作者简介

何小栋，网名 husthxd，毕业于华中理工大学，一直从事政府行业软件的开发和实施，但时常“不务正业”地研究 Oracle 数据库。擅长后台开发、系统分析与设计，目前沉迷于 XP。希望能结交广大喜欢 Oracle 和 Java 的朋友，一起交流，共同进步。

E-mail: husthxd@itput.net

MSN: husthxd@hotmail.com

Blog: blog.itpub.net/husthxd

第 19 章　电信业 Oracle 优化手记

在电信业的超大型数据库里，挑战无处不在。数据沉淀带来的“数据黑洞”问题，表里的记录条数以亿为单位，临时表空间和 rollback 表空间都比别人的整个数据库都大，这的确需要随需应变。本章总结了对这些巨无霸进行性能优化的几个方面。

19.1　一条 SQL 语句要运行 2 年怎么办

一不小心，写出的 SQL 需要运行 2 年，怎么办？本节以一个简单的例子，描述了优化的过程。

在普通数据量下，处理数据的办法比较完善，不过，这些办法在海量数据下，情况就变得完全不同了。是否遇到这样的情形：在一张大表中，做一个普通的 DML 操作，或者建一个索引，看看预计完成时间，居然是个天文数字。

下面来看一下，处理海量数据的一个典型情形。超大量的数据处理起来有什么特点？为什么处理海量数据时，平时一直有效的方法会行不通？两种处理方法有什么适用范围？

本例的具体要求：更新 t1 表，记录数 4161 万条，通过表中的 d2_pk_id 字段与表 d2 的主键 d2_pk_id 关联，以 d2 的 value1 更新 t1 表的 value1，d2 的记录数为 684 万条。相关字段有索引，并经过充分优化，并有采集统计数据。

从初步的试验中可以发现，如果单纯对某些记录定位，是很快的。如果更新几百万数据，花的时间也很少，速度比预期值要快。如果直接写一句 update 的 SQL，那么长时间运行后报经典错误“快照过旧”。

这证明，只要索引创建适当，定位不是瓶颈。小数据量的 DML 操作，性能也相当好。不过，如果全表 4000 万条记录都要更新，速度就会显著慢了下来，最后也无法完成。即使加大回滚段的长度、增加 undo_retention、保留足够的系统资源，再次运行，问题也依旧无法解决。

于是，尝试了以下几个方法，从不同的方面进行优化，以提高速度。

方法一：写 PL/SQL，开 cursor。

```
     UPDATE t1
SET value1  = rec1.value1
      WHERE d2_pk_id = rec1.d2_pk_id;
```

结果：该过程运行了 5 小时 20 分钟，尚未完成，报 snapshot too old 错误退出。这是最传统的做法，显然行不通。

方法二：用 loop 循环，分批操作。

```
UPDATE t1
   SET t1.value1 =
         (SELECT value1
            FROM d2
           WHERE d2_pk_id = t1.d2_pk_id
            )
 WHERE t1.d2_pk_id >= i AND t1.d2_pk_id <= (i + 10000);
```

结果：1 小时 24 分完成约 1500 万条，总时间为 4 小时 40 分。通过分批操作减小了系统资源的消耗，虽然能够完成任务，但是速度不能令人满意。由于有多处需要类似的操作，就大大影响了业务系统的运作，因而此方法也不能接受。

方法三：在方法二的基础上优化 SQL。

```
update ( select t1.value1 a1, d2.value1 b1
       from t1 , d2
       where d2.d2_pk_id>=1000000000 and d2.d2_pk_id<1003000000
         and t1.d2_pk_id>=1000000000 and t1.d2_pk_id<1003000000
         and t1.d2_pk_id = d2.d2_pk_id)
set a1 = b1;
```

结果：1 小时 12 分钟完成约 1200 万条，大约 4 小时 30 分完成全部 4000 多万条数据。这种方法通过虚拟一张表，来进行更新操作，可以使执行计划最优，并且不受干扰。经在进一步的测试发现，数据量在 1000 万以上时，方法三比方法二要快一点，但是不显著；数据量在 1000 万以下时，方法三比方法二要明显得快，并且如果这两个表本身结构很复杂的话，那么方法三比方法二的效率要高很多。

经过这 3 个方法的测试后，发现通过优化表、优化索引和优化 SQL，确实能提高一定的性能，提高处理效率。但是，如果业务需求非常苛刻，deadline 压得很紧，要在非常有限的时间里完成任务，怎么办呢？

再从基本面上回顾一下，看看有什么优化工作可以做。

对于这条 update 语句，由于 value1 字段上不存在索引，也就不会有索引字段的更新。而 update 的时候，之前的 value1 字段的值会被写入 redo 和 undo，之后的值还会被记录在 redo 中和 data buffer 中。

在大批量数据的 DML 操作时，与其他 DML 操作相比，例如，在 undo 中需要记录 rowid 的 select，update 其实是一种相当昂贵的操作，除了本身的数据操作，还会伴随着相关的大量系统操作。

再看看这个 update 任务。如果任务不成功了，还可以再做一次，update 之前是什么值也不需要留意，因为无论原来是什么值，都会使用另外一张表的数据来无条件地覆盖表 t1 的这个字段。因此，恢复并不重要。

所以说，不记录 redo 信息和 undo 信息是可以接受的。

而通常来讲，Oracle 会把对表 t1 的所有操作都生成 redo log 和 undo log，当出现故障，如实例错误、电源掉电等导致操作不成功时，就可以通过联机重做日志、undo 信息等进行恢复。

因此可以通过设置表的 NOLOGGING 属性来控制这个表的重做信息和 undo 信息的生成。经过初步的测试，可以看到 NOLOGGING 可以减少 log 的生成，而且，当把表设为 NOLOGGING 后，并且使用/*+ append */的 insert 时，速度是最快的，这个时候 Oracle 只会生成最低限度的必须的 redo log，而没有一点 undo 信息。

原来的任务是 update，现在发现可能要采用 insert 才能达到最快的速度。update 怎么用 insert 代替呢？因此，就有了方法四。

方法四：具体实现步骤如下。

(1) 新建表。

```
create table t2 as select * from t1 where rownum<1;
```

(2) 把新建的表置为 NOLOGGING。

```
alter table t2 nologging;
```

(3) 插入数据。

```
insert /*+ append */ into t2 a
 (
col1,
col2,
col3,
col4,
col5,
col6,
col7,
col8,
value1
)
select
s.col1,
s.col2,
s.col3,
s.col4,
s.col5,
s.col6,
s.col7,
s.col8,
s.value1
from  t1 b, d2
where b.d2_pk_id=d2.d2_pk_id
```

结果：23 分钟，完成全表 4100 万条记录！快是情理之中，快这么多是意料之外。

(4) 收尾工作。

现在已经有 t1 和 t2 两张表了，t1 是更新前的旧表，t2 是更新后的新表。要使用新数据，可以把旧表改名或者删除，把新表改成需要的名字。当完成这些动作后，这个更新工作就算成功完成了。柳暗花明又一村的结局无疑充满了戏剧效果，而强劲的速度不但带来了目瞪口呆，也带来了深思。

总结一下，处理超大量数据的特点就是慢，既指数据处理慢，又指优化的过程缓慢。调整一个索引、更新一个字段，都是非常花费时间的事情，并且很难精确地预测优化成果。

不过，分区和基础理论能够很好地帮助我们。

19.2　优化的传统定律和新时尚

下面有几条曾经非常流行的优化定律，这些定律现在过时了吗？

- index 和表同一个表空间。
- 定期重建索引。
- 裸设备应该取代文件系统。
- 初始参数设置：cursor_sharing=similar。
- 初始参数设置：fast=true 。

19.2.1 index 和表同一个表空间（过时）

“ 把索引和表分开表空间放，这样可以减少磁盘的争用，优化 Oracle 的性能。”这是经常听到的一句金科玉律，无懈可击。Oracle 的优化指南上也这样写着，即使在 Oracle9iR2 的文档上也是这样强调的。

不过，让我们对这条规则了解得更清晰一些。

在这条守则最有效的时候，运行 Oracle 的服务器往往是挂着多个 18GB 的 SCSI 硬盘，如果 DBA 申请要求增加硬盘，就又会得到一堆新的 18GB 或者是 36GB 的硬盘。现在，增加空间意味着又一个 1T 的阵列挂在某一个逻辑卷下面。

在这条守则最有效的时候，RAID 还用得不多，磁盘设备盘直接挂在服务器下。现在，striping 已经非常普遍。并且，细化到每个存储厂商对各自硬件储存产品的参数设置，例如条状的宽度和深度，都有各自独特的推荐。

因此，虽然 I/O 优化的核心没有改变，依然是打散 I/O，消除 I/O 热点，均衡 I/O 请求。但是过去和现在两种截然不同的环境，导致了对待表空间的规划，有着截然不同的做法。

过去的最佳做法，就是手工创建不同的表空间，每个表空间由若干个数据文件组成，再把数据文件放到不同的磁盘设备上。因此索引和表分开表空间存放，就会使数据 striping 到不同的磁盘设备上，I/O 性能就会提高。

现在的最佳做法，就是让 RAID 阵列去完成这些底层的工作。由 RAID 自动把数据 striping 到不同的磁盘设备上，每次有 I/O 请求时，就由 RAID 控制器和 RAID 内的所有设备交互，多个设备同时处理这些请求。

可以这样理解，单纯地分开表空间而不分开磁盘设备是没有效果的。只要把索引和表实质上分散了，分布到不同的磁盘设备上，就是成功的 I/O 优化，而不必强求把索引和表放在不同的表空间。

直接地说，两者在不在同一个表空间并不重要，重要的是它们实质上是不是分布在多个磁盘设备上。

更进一步说，如果在索引和表放在同一个表空间的情况下，发现出现了 I/O 瓶颈，本质不在这两者要不要分开表空间，而在于这个表空间的物理规划有问题，I/O 不够均衡。

那是否可以不需要表空间的规划呢？把系统里所有的表和索引都放在同一个表空间里呢？不是。越大型的数据库，可维护性，例如备份，对表空间的要求就越多，也越苛刻。所以，今天的表空间的规划首先要满足管理维护的需求，而性能方面的要求就可以交给 RAID 控制器来完成。也就是说，表空间的规划从以前注重性能方面的考虑转为注重可维护性的考虑。

同一模块、同一周期或者同一来源的表、索引可能有相同的维护需求，那么就可以把它们划分到同一个表空间里。具体来说，典型的情形是：一个月的数据就是一个表空间，里面包含了表和索引，这个表空间包含多个数据文件，分布在多个磁盘设备上。

此外，有一个小细节，在 Oracle9iR2 中，为一个表空间新增数据文件，表空间内的原有数据的储存位置不会有变动。而在 10g 中，新增数据文件后，原有数据会在数据文件之间自动均衡。因此，在 9iR2 及更早的版本中，早期的规划更重要一些，并且新增数据文件后可以调整一下数据的分布以达到均衡 I/O 的效果。

新时尚 >>

只要表空间物理规划合理，属性相近的索引和表可以放在同一个表空间内。

当然，把表和索引都放在 system 表空间里是不可饶恕的。

19.2.2　定期重建索引（过时）

定期重建索引并不能解决任何问题。在重建索引之前，总是会期望进行重建后，索引的性能会比之前好，索引的空间会节省很多。

也许会遇到这样的情况，系统长期运转后，对某个表的操纵性能会下降，但这并不一定是索引的错，重建索引未必能提高这个表的操纵性能。这可能是执行计划变了，也可能是 extent 太多，还有很多原因，如果只顾着重建索引，反而会耽误寻找真正导致缓慢的原因。

系统长期运转后，索引会不断增长，可能会长得很大很胖，甚至比表本身还大，但是重建索引并不能帮索引减肥。

如果周期性地重建索引有效，那么 Oracle 肯定会有一项新功能：Background Intelligent Rebuild Index Service。

不过，与此相对的是，定期收集表和索引的统计数据依然非常有效。

新时尚 >>

把每天定时重建索引的任务停掉吧。

19.2.3　裸设备应该取代文件系统（过时）

优化 Oracle 的性能，确实可以通过使用裸设备来实现。

在 UNIX 系统上，通常的做法是在磁盘分区创建 UNIX 文件系统，然后就可以使用。不过，文件系统并不是必须的，磁盘设备上没有一个文件系统也可以使用。裸设备（Raw Device）指不含有文件系统的磁盘分区。

因此，对于一个要运行 Oracle 的 UNIX 系统，使用磁盘有两个选择，要么不建文件系统，作为裸设备使用磁盘设备。要么创建文件系统，通过操作系统来使用磁盘设备。

从原理上来说，裸设备的 I/O 操作肯定比文件系统快。因为创建文件系统后，对磁盘设备的使用就会由操作系统调度，也就是说，每次对这些磁盘设备有读或者写的请求时，操作系统都需要处理并消耗一定的系统资源。而裸设备就可以避免这样的开销，I/O 请求可以跳过操作系统直接对磁盘操作，这样速度就快了。Oracle 也会因此而受益，I/O 会更快，Oracle 的性能也就更好。

但是，裸设备的使用不方便。由于操作系统不管理裸设备，所以，所有的标准 UNIX 文件操作命令都不能使用，例如 ls、cp、mv 等常用命令，很多备份和恢复的软件、产品、功能也不能使

用，这无疑增加了很多管理的困难。因此，对于裸设备和文件系统的对决来说，常常就是要性能还是要易维护性。

在以前，裸设备带来的 Oracle 性能的提高是显著的，面对性能的苛刻要求，易维护性只好被牺牲。今天，情况有所改变。硬盘速度提高了，普通的 IDE 家用硬盘比昔日的 SCSI 顶级硬盘速度还快，磁盘控制器变快了，缓存变快了，操作系统的逻辑卷管理器变快了，这些导致裸设备的性能并不比文件系统快多少。

因此，现在的选择并没有很明确的偏向。选择裸设备和文件系统都行，并不需要屈服于性能或者易维护性中的任何一方面。

新时尚 >>

如果熟悉裸设备并愿意使用裸设备，那么就可以享受裸设备带来的性能提高，虽然不像以前提高得那么多。如果不熟悉裸设备并担心会影响维护效率，那么也不用担心没有使用裸设备会给 Oracle 系统带来 I/O 瓶颈。

另外，追求性能极限的 DBA 可以尝试一下 reiserfs4 文件系统（还有 FreeBSD 的 softupdate），虽然这种文件系统似乎还没有被用于 Oracle 生产系统，可靠性还没有得到验证，不过，看上去性能相当不错。

19.2.4 初始参数设置 cursor_sharing=similar（不一定有效）

cursor_sharing=similar 曾经是一个无敌的参数，一个 DBA 在开始工作的第一天，就可以宣称，在增加了这个参数后，Soft Parse 提高了 10%，Oracle 的性能有了很大提高。但是，今天，使用这个参数需要格外的小心。

cursor_sharing 参数改变的是当一条 SQL 到达 Shared Pool 时，Oracle 所采取的动作。如果是设定了 similar，而 Shared Pool 里又恰好已经缓存并分析了另一句 SQL，这两句 SQL 基本一致、仅有字面值不同时，Oracle 会采用 Shared Pool 里所缓存的已经分析的代码，包括执行计划，避免再次分析并缓存 SQL 来达到节省系统资源、提高处理效率的目的。如果设定了 exact 就只有当两条 SQL 完全一致时，才会使用 Shared Pool 里的代码和执行计划。

（1）现在，DSS/DW 系统比过去任何时候都更普遍，在这些系统里，运行 SQL 的强度会比 OLTP 低很多，但是每条 SQL 语句的平均运行时间要长得多，全表扫描相当频繁，索引的创建和使用都需要严格监控和管理。这种时候，使用 cursor_sharing=similar，减少 Shared Pool 的重解释，节省的时间很少，意义就不那么大了。

（2）现在，索引比以往任何时候都多。而并不是所有的索引都被严格地优化，有很多时候，针对某一些表可能发生的操作是如此之多，以至于以前那种“把索引针对所有可能的操作都检查并优化一遍，以适合所有的情况”的做法成为不可能。

如果设定了 cursor_sharing=similar，Oracle 是动态地用字面值替换赋值变量，以便采用共享 Shared Pool 中已缓存的代码。这个动态替换会选择那些从语法上看不会改变执行计划的内容来进行替换。所以使用 cursor_sharing=similar 从原理上看是不会影响执行计划的。

但是，实际上，由于索引的影响，仅有字面值不同的两条 SQL 很可能会产生不同的执行计划，尤其是那些字面值分布范围很大，分布浓度很不均匀的情况。因此，使用了 cursor_sharing 的参数后并不能保证 Shared Pool 里缓存的那条 SQL 对新的这条 SQL 会是合适的，所以结果是有可能

更快，也有可能更慢。

这样的情况不但存在于 OLTP 系统也存在于 DSS/DW 系统。

（3）要留意大压力下的异常情况。在大压力的环境下，如果是一直使用 exact，并且运行很正常，那么改成 cursor_sharing=similar 后，就需要非常仔细地进行监控。最保险的做法是从项目一开始，就使用 cursor_sharing=similar，从项目一开始，就培训开发人员使用绑定变量，从项目一开始，就要使用完全的数据量做完整的压力测试。

新时尚 >>

不要迷信 cursor_sharing=similar 这个参数，无论如何，在生产环境应用前，一定要测试、测试、再测试。

如果准备条件充分，默认的 cursor_sharing=exact 也会工作得不错。

19.2.5　初始参数设置 fast=true（有效）

那些在 initSID.ora 里设了这个参数的 DBA 可以继续使用这个参数。这个未公开的只流传于少数 DBA guru 之间的参数会使你的 Oracle 像按了 Turbo 键一样，获得一定的性能提升而不会有副作用。

从目前的情况来看，在 9.2.0.1～9.2.0.5、10.1.0.1～10.1.0.3 等版本中，这个参数都依然有效，并且，Oracle 的 Roadmap 并没有显示，在未来几年的版本升级中会取消这个参数，如果想用的话就放心用好了。

新时尚 >>

以前一直在用的可以继续使用。

19.3　联机重做日志的优化

联机重做日志的优化常常被忽视，其实它的性能和可靠性对系统影响巨大，为什么不关注一下联机重做日志呢？

不可否认的是，监视联机重做日志是有些困难，或者说没有很方便的途径，相关的视图也非常有限，不过，在 Statspack 和操作系统的 I/O 监测工具甚至 alertlog 里都能够找到一些很有用的指标。

总的来说，关于联机重做日志的调整，主要和以下两方面相关。

19.3.1　联机重做日志组内创建多个成员

对于 Oracle 来说，有两种文件非常重要，一种是联机重做日志，另一种是控制文件，尤其是联机重做日志。Oracle 的重做机制是 Server Process 把 redo 信息写到 redo log buffer 后，再由 Log Writer 把 redo log buffer 的内容写到 redo log 上。当前的日志组写满了就会切换到下一个日

志组。

于是，如果联机重做日志丢失了，或者被破坏了，已经提交的事务就有可能丢失。

一般来说，系统都会使用带校验或镜像的机制的 RAID 阵列来加强数据的可靠性。但是如果一个重做日志组里只有一个日志，那么系统并不是任何时候都得到完全的保护。一旦某个存放有重做日志的磁盘设备亮了黄灯，即使不是红灯，也意味着这个设备上的日志就有可能有问题了，例如一下子坏了两块硬盘，或 RAID 的控制器出现了问题。

因此，要更好地保护联机重做日志，除了依赖良好的 RAID 规划外，为每个日志组都创建多个日志成员，还是非常有必要的。

19.3.2 加大 redo log 的容量

为了在提高恢复速度和提高性能之间找到平衡点，Oracle 推荐的重做日志切换的时间是一小时 3 次，也就是 20 分钟一次。

太少的日志切换会给可恢复性带来难度，过多的日志切换将严重影响系统性能，因为每次的日志切换都会引起检查点，而检查点要完成很多事情，例如，DBW0 后台进程把 database buffer cache 中的数据写到数据文件中，LGWR 后台进程把 redo log buffer 中的数据写到联机重做日志中，CKPT 后台进程把控制文件和数据文件的头部更新等。

可见，检查点的 I/O 操作非常频繁。如果前一个检查点还没有完成，下一个检查点就被启动，那么前一个检查点无论做到哪一步，都将会被放弃，重新从下一个检查点开始工作。所以如果发生了这样的情况，一个检查点未完成，那么就意味着很多系统资源被额外的浪费。

所以，重做日志切换需要保持在每小时 2～4 次以内，如果每小时超过 5 次，就要引起注意，如果超过 10 次，就需要调整了。如果每小时发生了 300 多次，那么这段时间里，系统都会忙于日志切换、完成检查点以及相关的 I/O 操作上了，而无法迅速处理其他事务，这样的系统是无论如何也快不起来的。

Oracle 9i 默认的联机重做日志的大小是 100MB，三组，每组一个成员，这对大系统来说是远远不够的。如果在 alertSID.log 里，频繁地出现日志切换和检查点未完成的标记，那么就证明用户需要调整联机重做日志。至于日志具体需要多大的容量，这取决于运行的事务的大小和频繁程度，在上面提到的那个每小时 300 多次日志切换的例子中，联机重做日志的大小容量调到了每个 2GB 才符合要求，达到每小时 3 次的正常水平。每个日志 2GB 甚至 3GB～4GB，可能超出了原来的 I/O 规划，又或者有人会担心这么大的 log 会不会难以操作，不用担心，测试是最好的朋友，事实是最有说服力的，并且可以使用 Statspack 监控系统的运行状况。

关于联机重做日志的调整主要有这两个方面，至于具体的监控参数和相关的视图、工具，可以参考有关的文档。有一个好习惯要提一下：在生产系统上实施前，要记得做测试。

作者简介

叶宇，在中国电信广州研发中心工作，OCP 9i，熟悉 BI/CPM/KPI，擅长应用新技术满足业务需求。

E-mail: yeahy@itpub.net

第 20 章　一次诊断和解决 CPU 利用率高的问题分析

Oracle 数据库经常会遇到 CPU 利用率很高的情况，这种时候大都是数据库中存在着严重性能低下的 SQL 语句，这种 SQL 语句大大地消耗了 CPU 资源，导致整个系统性能低下。当然，引起严重性能低下的 SQL 语句的原因是多方面的，具体的原因要具体地来分析，下面通过一个实际的案例来说明如何来诊断和解决 CPU 利用率高的这类问题。

本案例的诊断背景如下。

- 操作系统：Solairs 8
- 数据库：Oracle 9.2.0.4
- 问题描述：现场工程师汇报数据库非常慢，几乎所有应用操作均无法正常进行。

20.1　问题的具体描述

首先登录主机，执行 top 命令发现 CPU 资源几乎消耗殆尽，存在很多占用 CPU 很高的进程，但是内存和 I/O 的占用率都不高，具体情况如下：

```
last pid: 26136;  load averages:  8.89,  8.91,  8.12
216 processes: 204 sleeping, 8 running, 4 on cpu
CPU states:  0.6% idle, 97.3% user,  1.8% kernel,  0.2% iowait,  0.0% swap
Memory: 8192M real, 1166M free, 14M swap in use, 8179M swap free
PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
25725 oracle    1  50    0 4550M 4508M cpu2   12:23 11.23% oracle
25774 oracle    1  41    0 4550M 4508M run    14:25 10.66% oracle
26016 oracle    1  31    0 4550M 4508M run     5:41 10.37% oracle
26010 oracle    1  41    0 4550M 4508M run     4:40  9.81% oracle
26014 oracle    1  51    0 4550M 4506M cpu6    4:19  9.76% oracle
25873 oracle    1  41    0 4550M 4508M run    12:10  9.45% oracle
25723 oracle    1  50    0 4550M 4508M run    15:09  9.40% oracle
26121 oracle    1  41    0 4550M 4506M cpu0    1:13  9.28% oracle
25745 oracle    1  41    0 4551M 4512M run     9:33  9.28% oracle
26136 oracle    1  41    0 4550M 4506M run     0:06  5.61% oracle
```

```
  409 root     15  59    0 7168K 7008K sleep 173.1H  0.52% picld
25653 oracle    1  59    0 4550M 4508M sleep   1:01  0.46% oracle
25565 oracle    1  59    0 4550M 4508M sleep   0:07  0.24% oracle
25703 oracle    1  59    0 4550M 4506M sleep   0:08  0.13% oracle
25701 oracle    1  59    0 4550M 4509M sleep   0:23  0.10% oracle
```

20.2 问题的详细诊断解决过程

于是先查看数据库的告警日志 ALERT 文件，并没有发现有什么错误存在，日志显示数据库运行正常，排除数据库本身存在问题。

然后查看这些占用 CPU 资源很高的 Oracle 进程究竟是在做什么操作，使用如下 SQL 语句：

```
select sql_text,spid,v$session.program,process  from
v$sqlarea,v$session,v$process
where v$sqlarea.address=v$session.sql_address
and v$sqlarea.hash_value=v$session.sql_hash_value
and v$session.paddr=v$process.addr
and v$process.spid in (PID);
```

用 top 中占用 CPU 很高的进程的 PID 替换脚本中的 PID,得到相应的 Oracle 进程所执行的 SQL 语句，发现占用 CPU 资源很高的进程都是执行同一个 SQL 语句：

```
SELECT   d.domainname,d.mswitchdomainid,   a.SERVICEID,a.SERVICECODE,a.USERTYPE,a.STATUS,
a.NOTIFYSTATUS,to_char(a.DATECREATED,'yyyy-mm-dd   hh24:mi:ss')   DATECREATED,VIPFLAG,STATUS2,
CUSTOMERTYPE,CUSTOMERID  FROM service a, gatewayloc b, subbureaunumber c, mswitchdomain d   WHERE
b.mswitchdomainid = d.mswitchdomainid and b.gatewaysn = c.gatewaysn  AND a.ServiceCode like
c.code||'%' and a.serviceSpecID=1 and a.status!='4' and a.status!='10'  and a.servicecode like
'010987654321%' and SubsidiaryID=999999999
```

基本上可以肯定是这个 SQL 引起了系统 CPU 资源大量被占用，那究竟是什么原因造成这个 SQL 这么大量地占用 CPU 资源呢，先来看看数据库的进程等待事件都有些什么：

```
SQL> select sid,event,p1,p1text from v$session_wait;
      SID EVENT      P1 P1TEXT
---------- ---------------------------------------------------------------
       12 latch free  4.3982E+12 address
       36 latch free  4.3982E+12 address
       37 latch free  4.3982E+12 address
       84 latch free  4.3982E+12 address
      102 latch free  4.3982E+12 address
      101 latch free  4.3982E+12 address
       85 latch free  4.3982E+12 address
       41 latch free  4.3982E+12 address
      106 latch free  4.3982E+12 address
      155 latch free  4.3982E+12 address
      151 latch free  4.3982E+12 address
      149 latch free  4.3982E+12 address
      147 latch free  4.3982E+12 address
        1 pmon timer  300 duration
```

从上面的查询可以看出，大都是 latch free 的等待事件，然后接着查一下这些 latch 的等待都是什么进程产生的：

```
SQL> select spid from v$process where addr in
```

```
    (select paddr from v$session where sid in(84,102,101,106,155,151));
SPID
------------
25774
26010
25873
25725
26014
26016
```

由此看出 latch free 这个等待事件导致了上面的那个 SQL 语句在等待，占用了大量的 CPU 资源。下面来看看究竟主要是哪种类型的 latch 等待，根据下面的 SQL 语句：

```
SQL> SELECT latch#, name, gets, misses, sleeps
    FROM v$latch
    WHERE sleeps>0
    ORDER BY sleeps;
LATCH#  NAME                      GETS    MISSES     SLEEPS
---------- ----------------------------------------------------------------
   15   messages                   96876      20         1
  159   library cache pin allocation   407322     43         1
  132   dml lock allocation          194533     213        2
    4   session allocation           304897     48         3
  115   redo allocation              238031     286        4
   17   enqueue hash chains          277510     85         5
    7   session idle bit            2727264    314        16
  158   library cache pin            3881788    5586       58
  156   shared pool                 2771629    6184       662
  157   library cache                5637573    25246      801
   98   cache buffers chains         1722750424  758400     109837
```

由上面的查询可以看出最主要的 latch 等待是 cache buffers chains，这个 latch 等待表明数据库存在单独的 block 竞争，下面来看这个 latch 存在的子 latch 及其对应的类型：

```
SQL> SELECT addr, latch#, gets, misses, sleeps
    FROM v$latch_children
    WHERE sleeps>0
    and latch# = 98
    ORDER BY sleeps desc;
ADDR              LATCH#      GETS    MISSES    SLEEPS
---------------- ---------- ---------- ---------- ----------
000004000A3DFD10        98   10840661     82891       389
000004000A698C70        98    159510        2       244
0000040009B21738        98  104269771     34926       209
0000040009B227A8        98  107604659     35697       185
000004000A3E0D70        98    5447601     18922       156
000004000A6C2BD0        98     853375        7       134
0000040009B24888        98   85538409     25752       106
000004000A36B250        98    1083351       199        96
000004000A79EC70        98     257970        64        35
000004000A356AD0        98    1184810       160        34
……
```

接着来查看 sleep 较多的子 latch 都对应哪些对象：

```
SQL> select distinct a.owner,a.segment_name,a.segment_type from
```

```
    dba_extents a,
 (select dbarfil,dbablk
from x$bh
where hladdr in
    (select addr
    from (select addr
    from v$latch_children
    order by sleeps desc)
    where rownum < 5)) b
    where a.RELATIVE_FNO = b.dbarfil
    and a.BLOCK_ID <= b.dbablk and a.block_id + a.blocks > b.dbablk;
OWNER                SEGMENT_NAME                SEGMENT_TYPE
------------------------------------------------------------------------------
TEST                I_SERVICE_SERVICESPECID           INDEX
TEST                I_SERVICE_SUBSIDIARYID            INDEX
TEST                SERVICE                          TABLE
TEST                MSWITCHDOMAIN                     TABLE
TEST                I_SERVICE_SC_S                    INDEX
TEST                PK_MSWITCHDOMAIN                  INDEX
TEST                GATEWAYLOC                       TABLE
……
```

可以看到在开始的那个 SQL 语句中的几个对象都有包括在内，于是来看看开始的那个 SQL 的执行计划：

```
SQL> set autotrace trace explain
SQL>SELECT                                              d.domainname,d.mswitchdomainid,
a.SERVICEID,a.SERVICECODE,a.USERTYPE,a.STATUS,a.NOTIFYSTATUS,to_char(a.DATECREATED,'yyyy-mm-d
d hh24:mi:ss') DATECREATED,VIPFLAG,STATUS2,CUSTOMERTYPE,CUSTOMERID  FROM service a, gatewayloc
b, subbureaunumber c, mswitchdomain d   WHERE b.mswitchdomainid = d.mswitchdomainid and b.gatewaysn
= c.gatewaysn  AND a.ServiceCode like c.code||'%' and a.serviceSpecID=1 and a.status!='4' and
a.status!='10'  and a.servicecode like '010987654321%' and SubsidiaryID=999999999;
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
   1    0   NESTED LOOPS
   2    1     NESTED LOOPS
   3    2       NESTED LOOPS
   4    3         TABLE ACCESS (FULL) OF 'SUBBUREAUNUMBER'
   5    3         TABLE ACCESS (BY INDEX ROWID) OF 'GATEWAYLOC'
   6    5           INDEX (UNIQUE SCAN) OF 'PK_GATEWAYLOC' (UNIQUE)
   7    2       TABLE ACCESS (BY INDEX ROWID) OF 'MSWITCHDOMAIN'
   8    7         INDEX (UNIQUE SCAN) OF 'PK_MSWITCHDOMAIN' (UNIQUE)
   9    1     TABLE ACCESS (BY INDEX ROWID) OF 'SERVICE'
  10    9       AND-EQUAL
  11   10         INDEX (RANGE SCAN) OF 'I_SERVICE_SERVICESPECID' (NON
         -UNIQUE)
  12   10         INDEX (RANGE SCAN) OF 'I_SERVICE_SUBSIDIARYID' (NON-
         UNIQUE)
```

根据开始查到的引起 latch free 等待中的对象和 SQL 语句的执行计划，觉得 SERVICE 表上的索引有问题，似乎存在了过多的扫描，于是将同样的 SQL 语句在其他的同样的数据库上执行一下，查看相应的执行计划：

```
SQL> set autotrace trace explain
SQL>SELECT                                                    d.domainname,d.mswitchdomainid,
a.SERVICEID,a.SERVICECODE,a.USERTYPE,a.STATUS,a.NOTIFYSTATUS,to_char(a.DATECREATED,'yyyy-mm-d
d hh24:mi:ss') DATECREATED,VIPFLAG,STATUS2,CUSTOMERTYPE,CUSTOMERID  FROM service a, gatewayloc
b, subbureaunumber c, mswitchdomain d   WHERE b.mswitchdomainid = d.mswitchdomainid and b.gatewaysn
= c.gatewaysn  AND a.ServiceCode like c.code||'%' and a.serviceSpecID=1 and a.status!='4' and
a.status!='10'  and a.servicecode like '010987654321%' and SubsidiaryID=999999999;
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
   1   0   TABLE ACCESS (BY INDEX ROWID) OF 'SERVICE'
   2   1     NESTED LOOPS
   3   2       NESTED LOOPS
   4   3         NESTED LOOPS
   5   4           TABLE ACCESS (FULL) OF 'SUBBUREAUNUMBER'
   6   4           TABLE ACCESS (BY INDEX ROWID) OF 'GATEWAYLOC'
   7   6             INDEX (UNIQUE SCAN) OF 'PK_GATEWAYLOC' (UNIQUE)
   8   3         TABLE ACCESS (BY INDEX ROWID) OF 'MSWITCHDOMAIN'
   9   8           INDEX (UNIQUE SCAN) OF 'PK_MSWITCHDOMAIN' (UNIQUE)
  10   2       INDEX (RANGE SCAN) OF 'I_SERVICE_SC_S' (NON-UNIQUE)
```

对比两个执行计划，发现索引 I_SERVICE_SERVICESPECID 和 I_SERVICE_SUBSIDIARYID 是不应该走的，于是又对比两个地方的 SERVICE 表上的索引个数：

```
SQL> select index_name from user_indexes where table_name='SERVICE';
INDEX_NAME
------------------------------
I_SERVICE_ACCOUNTNUM
I_SERVICE_CID
I_SERVICE_DATEACTIVATED
I_SERVICE_PRICEPLANID
I_SERVICE_SC_S
I_SERVICE_SERVICECODE
I_SERVICE_SERVICESPECID
I_SERVICE_SUBSIDIARYID
PK_SERVICE_SID
SQL> select index_name from user_indexes where table_name='SERVICE';
INDEX_NAME
------------------------------
I_SERVICE_ACCOUNTNUM
I_SERVICE_CID
I_SERVICE_DATEACTIVATED
I_SERVICE_SC_S
I_SERVICE_SERVICECODE
PK_SERVICE_SID
```

发现存在问题的数据库中的 SERVICE 表上不知怎么多出了 I_SERVICE_PRICEPLANID、I_SERVICE_SERVICESPECID 和 I_SERVICE_SUBSIDIARYID 三个索引，而这些索引就是导致了开始那个 SQL 语句用了不该用的索引，引起 latch free 等待和 CPU 占用很高的罪魁祸首，于是删除了这三个索引，重新执行相应的 SQL 语句，很快就得出了结果，CPU 的利用率也马上下降为正常了，观察结果如下：

```
last pid: 26387;  load averages:  1.61, 1.38, 1.21
```

```
195 processes: 194 sleeping, 1 on cpu
CPU states: 96.2% idle,  1.6% user,  1.7% kernel,  0.5% iowait,  0.0% swap
Memory: 8192M real, 1183M free, 14M swap in use, 8179M swap free
PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
26383 oracle    1  59    0 4550M 4506M sleep   0:12  4.52% oracle
  409 root     15  59    0 7168K 7008K sleep 173.1H  0.53% picld
25653 oracle    1  59    0 4550M 4508M sleep   2:12  0.48% oracle
26384 root      1  59    0 2800K 1912K cpu2    0:00  0.21% top-3.5b8-sun4u
25569 oracle    1  59    0 4550M 4508M sleep   0:12  0.09% oracle
25717 oracle    1  59    0 4550M 4507M sleep   0:07  0.05% oracle
25571 oracle    1  59    0 4550M 4507M sleep   0:10  0.04% oracle
25681 oracle    1  59    0 4550M 4508M sleep   0:10  0.04% oracle
25544 oracle    1  58    0 4554M 4501M sleep   0:14  0.03% oracle
25703 oracle    1  59    0 4550M 4506M sleep   0:23  0.03% oracle
25679 oracle    1  59    0 4550M 4508M sleep   0:09  0.03% oracle
25649 oracle    1  59    0 4550M 4506M sleep   0:05  0.03% oracle
25697 oracle    1  59    0 4550M 4506M sleep   0:02  0.03% oracle
25625 oracle    1  59    0 4550M 4509M sleep   0:01  0.03% oracle
26012 oracle    1  59    0 4551M 4509M sleep   0:12  0.02% oracle
```

至此，问题得到解决。

20.3 小结

对于 CPU 利用率过高的情况，如果是 SQL 语句性能比较低下引起的基本上都可以按照这个思路来诊断和解决问题，当然具体问题还得具体分析，解决问题的方法也有很多种，只要最终能达到解决问题的目的就可以了，这里不过是抛砖引玉一下。

作者简介

叶梁，网名 cooly1，现任 ITPUB Oracle 管理版版主。

曾任职于国内某大型软件企业做 Oracle 数据库的技术支持，客户遍及全国各个行业，尤其是电信、政府、金融行业。现任职于某外资电信企业华北区分公司，从事 DBA 工作，负责华北区 40 多个数据库系统的维护，对大型数据库管理经验丰富。

擅长数据库的维护，对于数据库的安装，调整，备份方面有自己独到的经验。同时也给一些国内的大型企业做过 Oracle 的培训，有一定的培训经验。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

曾做过很多大型项目的数据库维护和支持工作，对 Oracle 的维护有相当多的实际经验，善于现场解决问题。

第 21 章　一次异常内存消耗问题的诊断及解决

21.1　问题发现

一大早，接到通知说现网数据库（RAC 中的一个节点）出现异常，现象是大量的业务拥堵，系统可用内存很低，经常徘徊在 100MB 左右，随着业务高峰期的时间就要到了，眼看系统的内存已经处在极其危险的状态。

21.2　解决过程

下面对该问题进行详细的分析和诊断。

21.2.1　环境介绍

硬件和软件环境如下：

- HP 7410，16GB 内存，8 个 CPU，存储是 HP XP128
- OS 版本，HPUX 11.11 + MC 11.15
- 2 节点 RAC，Oracle 版本 9.2.0.4

21.2.2　问题现象

首先看一下进程的内存情况（HPUX 11.11）：

```
oracle@cs_db02:/oracle > vmstat 2 10
       procs          memory               page                            faults      cpu
    r    b    w     avm    free   re   at    pi   po   fr  de   sr     in     sy    cs  us sy id
    8    5    0 2659897   54987  127   13     0    0    0   0   23  17401 114389  6145  21  6 73
    8    5    0 2659897   54365    4    0     4    0    0   0    0  16032 126259  7605  35  5 60
    8    5    0 2659897   54232    2    0     3    0    0   0    0  16043 139904  7705  38 11 51
```

```
   2   5    0  2664805   53921    0    0    32    0   0   0    0  16252 141213  7843  33  6 60
   2   5    0  2664805   56209    0    0    20    0   0   0    0  15828 136485  7592  30  6 65
   2   5    0  2664805   55701    0    0    56    0   0   0    0  16330 137069  7952  38  7 54
   2   5    0  2664805   55685    0    0    35    0   0   0    0  16819 141478  8098  32  6 62
   2   5    0  2664805   53749    0    0    22    0   0   0    0  16855 139303  8003  34  7 59
   7   1    0  2620255   51536  172    0    41    0   0   0    0  16521 136507  8093  44  6 50
   7   1    0  2620255   51323  109    0    28    0   0   0    0  17859 151458  8623  43  7 50
oracle@cs_db02:/oracle >
```

一般来说，系统的 page out（po）可以有一些，但如果 page in（pi）很高，并间歇性地还在逐渐上升，这就要引起注意。总得来说，正常系统的 paging 可以有一些，但不能太多。

而这里，可以看见运行队列（r）很高，系统可用的空闲内存极少，并且伴随大量的 page in，已经出现了明显的内存不足。同时也可以看到用户进程的 CPU 使用率（us）几乎没有异常，并且它们的 CPU 使用率都在合理的范围内（抛除 wait io 意味的 CPU 使用率）。

再使用 top 观察是否有大量的极其耗费 CPU 的进程在操作：

```
oracle@cs_db02:/oracle > top

System: cs_db02                                  Fri Mar  4 12:14:47 2005
Load averages: 0.56, 0.57, 0.58
966 processes: 951 sleeping, 15 running
Cpu states:
CPU   LOAD   USER   NICE    SYS   IDLE  BLOCK  SWAIT   INTR   SSYS
 0    0.65  30.0%   0.0%   5.7%  64.3%   0.0%   0.0%   0.0%   0.0%
 1    0.77  95.7%   0.0%   1.0%   3.4%   0.0%   0.0%   0.0%   0.0%
 2    0.64  24.9%   0.0%  11.6%  63.5%   0.0%   0.0%   0.0%   0.0%
 3    0.40  36.5%   0.0%   7.3%  56.2%   0.0%   0.0%   0.0%   0.0%
 4    0.59  27.2%   0.0%   7.7%  65.1%   0.0%   0.0%   0.0%   0.0%
 5    0.54  26.4%   0.0%   7.3%  66.3%   0.0%   0.0%   0.0%   0.0%
 6    0.60  17.8%   0.0%  11.4%  70.8%   0.0%   0.0%   0.0%   0.0%
 7    0.29  36.3%   0.0%   3.6%  60.2%   0.0%   0.0%   0.0%   0.0%
---   ----  -----  -----  -----  -----  -----  -----  -----  -----
avg   0.56  36.9%   0.0%   6.9%  56.2%   0.0%   0.0%   0.0%   0.0%

Memory: 14829952K (8998976K) real, 20328344K (10419100K) virtual, 227036K free  Page# 1/35

CPU TTY  PID USERNAME PRI NI   SIZE    RES STATE    TIME %WCPU  %CPU COMMAND
 7  ?  8946 oracle   241 20  7328M  2148K run     13:56 99.14 98.97 ora_j002_csmisc2
 4  ? 11551 oracle   241 20  7344M 14384K run    148:31 22.84 22.80 oraclecsmisc2
 0  ? 22315 oracle   221 20  7358M  5668K run    294:01 18.22 18.19 ora_lms0_csmisc2
 3  ? 22317 oracle   154 20  7358M  5668K sleep  307:13 17.89 17.86 ora_lms1_csmisc2
 1  ?  9108 oracle   241 20  7328M  2820K run      0:06 17.31 17.28 oraclecsmisc2
 0  ?  9106 oracle   154 20  7328M  2928K sleep    0:22 10.87 10.85 oraclecsmisc2
 1  ?  7044 oracle   154 20  7333M  5720K sleep    0:45  7.25  7.24 oraclecsmisc2
 2  ? 11591 oracle   154 20  7328M   864K sleep  183:59  7.25  7.24 oraclecsmisc2
 4  ?  7235 root     152 20 43172K  7236K run      4:03  7.04  7.02 nco_m_ssmagent
 6  ? 11687 oracle   156 20  7328M   852K sleep   86:09  6.27  6.26 oraclecsmisc2
 4  ?  3650 oracle   154 20  7328M   816K sleep    7:11  5.54  5.53 oraclecsmisc2
 0  ? 11384 oracle   154 20  7328M   844K sleep   79:19  5.47  5.46 oraclecsmisc2
 5  ?  8772 oracle   154 20  7328M  2420K sleep    1:00  5.02  5.01 oraclecsmisc2
 3  ? 10083 oracle   154 20  7328M  1052K sleep   71:55  5.00  4.99 oraclecsmisc2
 3  ? 22335 oracle   156 20  7335M  2604K sleep   67:37  4.94  4.94 ora_lgwr_csmisc2
```

```
 2   ?  8836 root     -16 10 39044K 18820K run      0:36  3.02  3.01 midaemon
 4   ?  3225 oracle   154 20  7328M   844K sleep    3:55  2.95  2.95 oraclecsmisc2
 6   ? 22339 oracle   148 20  7328M  1560K sleep   47:11  2.60  2.60 ora_smon_csmisc2
 6   ?    0 root     127 20    32K     0K sleep   48:24  2.34  2.34 swapper
 2   ?  5115 oracle   154 20  7332M  2072K sleep    4:06  1.73  1.72 oraclecsmisc2
 2   ?  7607 oracle   154 20  7353M  1548K sleep    1:39  1.67  1.67 oraclecsmisc2
 4   ? 22313 oracle   154 20  7334M  3992K sleep   22:38  1.65  1.65 ora_lmd0_csmisc2
 3   ?  4824 oracle   154 20  7332M  3160K sleep    3:12  1.53  1.53 oraclecsmisc2
 3   ?  4896 oracle   154 20  7332M  3168K sleep    4:05  1.50  1.50 oraclecsmisc2
 1   ?  9697 oracle   154 20  7332M  3460K sleep    4:06  1.50  1.50 oraclecsmisc2
 5   ?  9733 oracle   154 20  7332M  2672K sleep    4:25  1.39  1.39 oraclecsmisc2
 4   ? 11487 oracle   154 20  7328M  1064K sleep    9:46  1.34  1.34 oraclecsmisc2
oracle@cs_db02:/oracle >
```

21.2.3　对比分析

这里可以清楚地看到空闲内存只有 200MB，并且几乎每个进程都伴随有很高的 wait io。于是想到对比以前的系统性能记录：

```
oracle@cs_db02:/oracle > sar -u 2 10

HP-UX cs_db02 B.11.11 U 9000/800   02/08/05

18:32:24    %usr    %sys    %wio   %idle
18:32:26      34       5      19      43
18:32:28      39       7      18      37
18:32:30      37       6      15      42
18:32:32      31       5      12      51
18:32:34      29       4       9      57
18:32:36      28       4      10      57
18:32:38      31       4      14      52
18:32:40      35       5      12      48
18:32:42      39       3      10      49
18:32:44      42       5      16      37

Average       35       5      14      47
oracle@cs_db02:/oracle >
```

使用 sar -u 命令可以看到，那时的系统状态是非常良好的，CPU 的 idle（%idle）平均值在 47%，而系统的 wait io（%wio）平均值在 14%，这是一个比较理想的状态。

再来看系统正常时候的内存使用情况：

```
oracle@cs_db02:/oracle > vmstat 2 10
        procs          memory                page                              faults       cpu
    r    b    w     avm    free   re   at   pi   po   fr   de   sr     in     sy    cs  us sy id
    5    2    0  930013 1008560  127    9    0    0    0    0    0  17382 114792  6110  21  6 73
    6    2    0  988914 1007728  208   40    0    0    0    0    0  15703 128093  7582  47  7 46
    6    2    0  988914 1007665  203   50    0    0    0    0    0  15603 127244  7496  42  6 52
    6    2    0  988914 1007664  213   63    0    0    0    0    0  15444 126853  7647  48  6 45
    6    2    0  988914 1007552  136   40    0    0    0    0    0  15592 133625  7417  39  4 56
    6    2    0  988914 1007265  163   54    0    0    0    0    0  15733 140186  7161  40  5 54
    4    1    0 1004118 1007248  104   34    0    0    0    0    0  15356 137336  6836  44  5 51
```

```
   4   1   0  1004118  1006640  130   46   1   0   0   0   0  15461 138396  6926  38  5 57
   4   1   0  1004118  1006641  163   59   1   0   0   0   0  16366 148071  7617  42  6 51
   4   1   0  1004118  1006640  104   37   1   0   0   0   0  15761 136805  7334  40  4 55
oracle@cs_db02:/oracle >
```

可以看到，虽然也存在有一些等待队列，但是系统的可用内存在 4GB 左右（free*page size=1008560*4096=4131061760 byte）。系统的几乎没有 page in。

情急之下，请业务人员停掉一个应用，以便系统稳定在 free memory 在 200MB 左右，然后开始进一步定位问题的原因。

现在，检查一下当前的 Oracle 进程数目，是否由于今天的业务太多，或者连接的用户太多而造成压力呢？

```
oracle@cs_db02:/oracle > ps -ef | grep oracle| wc -l
842
oracle@cs_db02:/oracle >
```

这个数值和平时相比，是一个非常正常的值，系统曾经在并发连接数目达到 1200 以上仍然正常工作，因此排除了业务过量导致系统资源不足的可能性。

现在需要尽快找到入手的线索，既然内存不足，存在大量的 page in 操作，那么看一下交换分区的使用状态：

```
oracle@cs_db02:/oracle > swapinfo
           Kb      Kb      Kb   PCT  START/      Kb
TYPE      AVAIL    USED    FREE  USED   LIMIT RESERVE  PRI  NAME
dev    32768000 4058864 28709136   12%      0       -    1  /dev/vg00/lvol2
reserve       - 15057520 -15057520
memory  12954600 1303496 11651104   10%
oracle@cs_db02:/oracle >
```

可以看见，一个正常的系统，交换分区的使用应该在 5%以内，这里交换区使用率已经达到了 12%，远远超过这个一般 HP 系统的常规值。

21.2.4 假设和分析

联系所有上面的现象，已经基本可以断定，系统今天的现象绝对不是由于某些单纯的用户进程引起的，应该有几种可能：

（1）Oracle 的 SGA 设置不合理。

（2）Oracle 的 PGA 使用有异常。

（3）有可能某些僵死的进程不能正常地释放内存。

（4）有某些人为的操作造成了内存泄漏。

（5）遇到了 HP 或者 Oracle 的某些 Bug。

逐步检查上述原因，继续寻找线索。首先，考虑到既然操作系统上连接的 Oracle 用户进程数目正常，会不会 Oracle 的 SGA 设置不合理呢？

```
SQL> show sga

Total System Global Area 7391385520 bytes
Fixed Size                747440 bytes
Variable Size          3070230528 bytes
Database Buffers       4294967296 bytes
```

```
Redo Buffers            25440256 bytes
SQL>
```

使用 show sga 可以清楚地看见，Oracle 的 SGA 区仅仅占用了物理内存 7.3GB 左右，其中，Data Buffer 使用了 4GB。对于这是一个有着 16GB 物理内存的 HP 7410 来说，专门地作为 Oracle 数据库服务器，并且没有任何的用户应用跑在上面，这个设置显然没有什么异常（在正常情况下，甚至曾经预计将再将 1GB～2GB 的系统空闲内存分给 Oracle 的 Data Buffer 来使用）。

排除了 SGA 设置过大的可能性，接下来看看，用户进程的时候是否有什么异常现象？这里新建立一个连接，在不做任何操作的情况下，看看一个空连接对系统 PGA 的使用情况，以观察是否有系统的 Bug 导致每个 Oracle 用户进程过度消耗内存的可能性：

```
SQL> select distinct sid from v$mystat;

      SID
----------
      206

Elapsed: 00:00:00.01
SQL> l
  1  select b.PGA_USED_MEM PGA_USED_MEM, b.PGA_ALLOC_MEM PGA_ALLOC_MEM,
  2  b.PGA_FREEABLE_MEM PGA_FREEABLE_MEM,b.PGA_MAX_MEM PGA_MAX_MEM
  3  from v$session a, v$process b
  4* where a.PADDR=b.ADDR and a.sid ='&sid'
SQL> /
Enter value for sid: 206
old   4: where a.PADDR=b.ADDR and a.sid ='&sid'
new   4: where a.PADDR=b.ADDR and a.sid ='206'

PGA_USED_MEM PGA_ALLOC_MEM PGA_FREEABLE_MEM PGA_MAX_MEM
------------ ------------- ---------------- -----------
     330201       612345              0    11884537
Elapsed: 00:00:00.02
```

很明显，一个空的进程只需要 300KB 的内存，这是非常正常的状态。现在再观察，系统总的 PGA 的使用情况，看看有什么发现：

```
SQL> select sum(PGA_USED_MEM) from v$process;

SUM(PGA_USED_MEM)
-----------------
       561546135

Elapsed: 00:00:00.08
SQL>
SQL> select sum(PGA_USED_MEM) from v$process;

SUM(PGA_USED_MEM)
-----------------
       409176749

Elapsed: 00:00:00.00
```

这里观察到，系统总的 PGA 的使用基本上稳定在 400MB～700MB 之间（系统设置的 PGA 的

最大值是 1GB），与往常相比也是一个非常合理的状态。

21.2.5 找到根源

到这里，可以做个简单的加减法计算，总的物理内存 16GB，减去 Oracle SGA 占用的 7.3GB，再减去 PGA 占用的 400MB～700MB（每时每刻根据使用情况在变化，最大为 1GB），再减去系统使用的和系统上显示的空闲内存，还应该有至少 6 个多 GB 的内存去了哪里？

这台主机只用于数据库服务器，因为没有运行任何应用的情况下，所以就不存在某些应用服务异常占用过量共享内存段而导致内存不足（这样的情况是存在的），那么我在这里做个大胆的假设：有某些异常的进程，异常地使用了将近 6 个多 GB 的共享内存段。如果可以找到哪个共享内存段被谁异常使用，就应该可以找到上述问题的答案。

按照这个假设，使用 ipcs 观察共享内存段的使用情况：

```
oracle@cs_db02:/oracle/app/oracle/admin/csmisc/udump > ipcs -mb
IPC status from /dev/kmem as of Fri Mar  4 15:57:17 2005
T     ID     KEY        MODE       OWNER    GROUP  SEGSZ
Shared Memory:
m      0 0x411c5552 --rw-rw-rw-     root     root    348
m      1 0x4e0c0002 --rw-rw-rw-     root     root  61760
m      2 0x412040ba --rw-rw-rw-     root     root   8192
m   15875 0x5e1003c9 --rw-------     root     root    512
m  260612 0x00000000 D-rw-r-----    oracle      dba 6267330560
m 8103429 0x00000000 --rw-r-----    oracle      dba 6442450944
m      6 0x06347849 --rw-rw-rw-     root      dba  77384
m      7 0xffffffff --rw-r--rw-     root      dba  22908
m 6940680 0x320fb498 --rw-r-----    oracle      dba 982507520
oracle@cs_db02:/oracle/app/oracle/admin/csmisc/udump >
```

注意到确实有一个异常的共享内存段（Share Memory ID 为 260612），而它刚好也占用了 6.2GB 左右的内存。这是一个 Oracle 用户占用的共享内存段，由于状态为 D 的共享内存段本身就是没有正常使用的内存段，所以以为使用 ipcrm –m 260612 删除这个共享内存段，就应该可以解决问题，但是，当时上述做法的结果是系统报告找不到这个 ID。

再看一下共享内存的详细使用信息：

```
oracle@cs_db02:/oracle/app/oracle/product/920/bin > ipcs -ma
IPC status from /dev/kmem as of Fri Mar  4 16:25:19 2005
T      ID     KEY        MODE   CREATOR CGROUP NATTCH  SEGSZ  CPID  LPID   ATIME    DTIME
CTIME
Shared Memory:
m      0 0x411c5552 --rw-rw-rw-   root root   0    348   528   528  4:35:46  4:35:46  4:35:39
m      1 0x4e0c0002 --rw-rw-rw-   root root   2  61760   528   530  4:36:40  4:35:46  4:35:39
m      2 0x412040ba --rw-rw-rw-   root root   2   8192   528   530  4:36:40  4:35:39  4:35:39
m   15875 0x5e1003c9 --rw-------   root root   1    512  1323  1323  4:37:10 no-entry  4:37:10
m  260612 0x00000000 D-rw-r----- oracle dba  1 6267330560 15316 17931 2:27:03  2:29:53 2:29:41
m 8103429 0x00000000 --rw-r----- oracle dba 738 6442450944 22305 17897 16:24:24 16:25:15
2:53:58
m      6 0x06347849 --rw-rw-rw-   root  dba   0  77384 12170 18503 16:21:41 16:21:41 10:20:38
m      7 0xffffffff --rw-r--rw-   root  dba   0  22908 12169 12169 10:20:40 10:20:40 10:20:40
m 6940680 0x320fb498 --rw-r----- oracle dba 738 982507520 22305 17897 16:24:24 16:25:15 2:53:58
```

```
m 6833161 0x0c6629c9 --rw-r-----  root  dba   2 17602880 18503 18502 16:21:41 no-entry 16:21:41
oracle@cs_db02:/oracle/app/oracl
```

这里有一个误解，CPID 是指创建这个共享内存段的进程号，创建时间是 2:29，LPID 是最后一次获取和使用这个共享内存段的进程号，使用时间是 2:29。因为 Oracle 在创建了这个段之后这个进程的使命也就完成了，可以简单地理解为类似一个 sh 脚本执行完成后正常退出的过程。同理，最后一次使用这个共享内存段的 LPID 值也是这个道理。所以，要试图通过这两个进程号来查找进程信息或者 kill 这两个进程时，是不会有什么结果的（kill 这两个进程时，系统却报告说找不到这个进程）。

这里观察到两个正常的 Oracle 共享内存段的创建时间似乎都在 2:53。联想起几天前由于修改业务而曾经重新启动过数据库，那么这里再做一个假设：这块内存被 Oracle 系统进程异常占用，或者是数据库 Shutdown 时没有正常释放所有的共享内存段。换个角度说，在数据库 Shutdown 时，操作系统没有正常地收回所有的共享内存段。

21.2.6　解决问题

现在使用 shminfo（需要使用 root 权限）查看一下当前到底哪个进程在使用这个共享内存段：

```
cs_db02#[/]./shminfo -s 260612
libp4 (7.0): Opening /stand/vmunix /dev/kmem

Loading symbols from /stand/vmunix
shminfo (3.6)

Shmid 260612:
struct shmid_ds at 0xf468e8
Pseudo vas at 0x594fd900
Pseudo pregion at 0x624a1d00
Shared region at 0x788a53c0
Segment at 0xd8d3400.0xc000000040000000
Segment allocated out of "Global 64-bit quadrant 4"
Processes using this segment:
proc=0x6d635040 (pid 17927 "oracle"): vas=0x61faee00, SHMEM preg=0x7dc32180
cs_db02#[/]
```

现在很清楚地看到，进程 17927 正在使用该共享内存段。接着，再使用 ps 查看 17927 的进程信息：

```
cs_db02#[/]ps -ef|grep 17927
   root 18107 18038  1 16:13:25 pts/tn   0:00 grep 17927
  oracle 17927    1  0  Feb  6  ?      93:09 ora_diag_csmisc2
cs_db02#[/]
```

注意进程名称是 ora_diag_csmisc2，这是一个后台诊断进程（Diagnosability Daemon），一般出现在 OPS 和 RAC 系统中，多用在实例 crash 前生长大量的系统跟踪文件和 SYSTEM DUMP 信息（生成的所有进程跟踪信息一般存放在 bdump 目录下一个以故障时间点命名的 core 目录中）。

通过该进程的时间可以看到，这是 2 月 6 日创建的进程，事实上，我在 3 月 3 日晚上重启了数据库，说明这个进程的创建时间和上面的共享内存段的创建时间和最后使用时间存在着出入。由于是现网数据库，所以在确定删除该进程前，我特意对比了一下 RAC 中的另一个正常节点：

```
cs_db02#[/]ps -ef |grep ora_diag_csmisc
```

```
    root 19117 18855  0 16:36:24 pts/to   0:00 grep ora_diag_csmisc
  oracle 22309    1  0  Mar  3  ?       0:19 ora_diag_csmisc2
  oracle 17927    1  1  Feb  6  ?      93:09 ora_diag_csmisc2
cs_db02#[/]

oracle@cs_db01:/oracle > ps -ef | grep ora_diag_csmisc
  oracle  3494  3457  0 16:37:11 pts/te   0:00 grep ora_diag_csmisc
  oracle 14718    1  1  Mar  3  ?       6:20 ora_diag_csmisc1
oracle@cs_db01:/oracle >
```

经过对比，发现节点 1（正常的节点）只有一个 3 月 3 日创建的后台诊断进程，而节点 2（异常的节点）存在 2 个后台诊断进程，一个是 3 月 3 日创建的，一个是 2 月 6 日创建的。

至此，已经可以确定，进程 17927 就是异常占用 6GB 以上的异常 Oracle 后台（系统）进程，可以删除，于是 kill 它：

```
cs_db02#[/]kill 17927
cs_db02#[/]ps -ef |grep ora_diag_csmisc
  oracle 22309    1  0  Mar  3  ?       0:19 ora_diag_csmisc2
    root 19661 18855  2 16:45:53 pts/to   0:00 grep ora_diag_csmisc
cs_db02#[/]
```

现在，再来看看系统性能：

```
cs_db02#[/]vmstat 2 5
       procs          memory                page                          faults      cpu
   r    b    w     avm    free   re   at   pi   po   fr   de    sr     in     sy    cs  us sy id
   2    0    0  933372 1544205  127   13    0    0    0    0   31  17401 114427  6148  21  6 73
   2    0    0  933372 1544193    8    0    4    0    0    0    0  12975  95961  5599  15  4 81
   2    0    0  933372 1543328    4    0    3    0    0    0    0  12989  94187  5522  14  4 82
   2    0    0  933372 1542944    2    0    2    0    0    0    0  13600  96823  5848  14  5 81
   2    0    0  933372 1540620    1    0    6    0    0    0    0  13216  90131  5740  22  5 73
cs_db02#[/]
```

几乎立竿见影，删除这个进程后，系统的运行队列（r）几乎没有了，阻塞队列（b）也一下子消失了，操作系统的 page in 导致的页交换骤减到一个非常正常的可接受值，系统的可用内存升高到 free*page size=1544205*4096=6325063680 byte 左右。

再次使用 top，可以看到：

```
cs_db02#[/]top


……
Memory: 7744752K (2796296K) real, 13421176K (4155264K) virtual, 6105600K free  Page# 1/36
……
```

系统的可用内存确实已经升高到 6GB 左右了。

再来看看 CPU 的使用：

```
cs_db02#[/]sar -u 2 10

HP-UX cs_db02 B.11.11 U 9000/800   03/04/05

16:39:31   %usr   %sys   %wio   %idle
16:39:33     15      8     20      56
16:39:35     14      7     20      59
16:39:37     12      4     18      66
16:39:39     20      5     22      54
```

```
16:39:41     17      8      12      63
16:39:43     11      4      13      73
16:39:45     15      4      17      64
16:39:47     13      4      13      70
16:39:49     17      9      22      51
16:39:51     14      5      15      65

Average      15      6      17      62
cs_db02#[/]
```

可以看到，CPU 的 idle 平均值（%idle）在 62%，而系统的 wait io（%wio）平均值在 17%，这已经是一个非常理想的状态，即系统性能已恢复正常。

21.3　小结

总结下来，解决这个问题的关键其实就在于如何释放掉那个操作系统不能正常回收而浪费的（Oracle 异常占用的）6GB 的内存。虽然，可以通过 shminfo 找到进程号杀死进程，也可以通过重启主机解决（当然，很少有人选择后者，尤其是现网数据库），但是继续跟踪这个问题的话，就能发现这是一个 HP MC 11.15 的 Bug，需要通过打 PATCH PHSS_28926 来解决这种内存不能正常回收的问题。

最后请记住，在一个安装在 HP MC 11.15 环境下的 Oracle RAC 系统中，要想正常使用数据库，必须至少安装 3 个相应的 HP MC 11.15 的 Patch：PHSS_28926、PHSS_30370 和 PHSS_29096。

作者简介

张大鹏，网名 Lunar2000（Lunar），ITPUB 资深会员。现任职于某大型外资企业，服务于电信增值业务，从事专职 DBA 工作。主要负责 Oracle 数据库日常管理，包括备份和恢复，性能优化，故障诊断等。实践经验丰富，长于数据库故障诊断、性能优化和备份恢复。

22

第 22 章　如何捕获问题 SQL 解决过度 CPU 消耗问题

本章通过一个实际业务系统性能调整的案例，给出了一个常见 CPU 消耗问题的诊断方法。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。

本案例平台为 UNIX，所以不可避免地应用了一些 UNIX 下的常用工具来辅助诊断，如 vmstat、top 等。本章对这些工具只做了简单介绍，详细信息可以参考 UNIX 系统中的 Man Page。

本案例的诊断背景如下。

- 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。

22.1　检查当前情况

登录数据库主机，检查当前情况。

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  1  1  0 4294967196 0 0 -84 -5 -145
 131 0 0 5368072 1518360 56 691 0 2 2 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  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 3570 8534 3316 97  3  0
 134 0 0 5373616 1520512 127 1078 0 2 2 0 0 0  1  0  0 3838 9584 3623 96  4  0
 136 0 0 5369392 1518496 107 924 0 5 5 0 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  0 3358 7944 3119 97  3  0
 129 0 0 5358648 1511712 189 1236 0 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
 128 0 0 5346848 1507704 99 823 0 0 0 0  0  0  0  3  0 3189 9048 3074 96  4  0
 125 0 0 5341248 1504704 80 843 0 2 2 0  0  0  6  1  0 3563 9514 3314 95  5  0
```

```
 133 0 0 5332744 1501112 79 798 0 0 0 0  0  0  0  1  0 3218 8805 2902 97  3  0
 129 0 0 5325384 1497368 107 643 0 2 2 0 0  0  1  4  0 3184 8297 2879 96  4  0
 126 0 0 5363144 1514320 81 753 0 0 0 0  0  0  0  0  0 2533 7409 2164 97  3  0
 136 0 0 5355624 1510512 169 566 786 0 0 0 0 0 0  1  0 3002 8600 2810 96  4  0
 130 1 0 5351448 1502936 267 580 1821 0 0 0 0 0 0 0  0 3126 7812 2900 96  4  0
 129 0 0 5347256 1499568 155 913 2 2 2 0 0  0  0  1  0 2225 8076 1941 98  2  0
 116 0 0 5338192 1495400 177 1162 0 0 0 0 0 0  0  1  0 1947 7781 1639 97  3  0
```

对于 vmstat 的用法及输出，简要说明一下，vmstat 是 UNIX 平台上一个常用的工具，可以帮助查看系统内存及 CPU 使用的情况。

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。

22.2 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。

在本案例中，观察进程 CPU 耗用情况，没有发现明显过高使用 CPU 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。

22.3　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。

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。

22.4　登录数据库

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
     ……
       161 buffer busy waits                    17 file#
       195 buffer busy waits                    17 file#
       311 buffer busy waits                    17 file#
       314 buffer busy waits                    17 file#
       205 buffer busy waits                    17 file#
       269 buffer busy waits                    17 file#
       200 buffer busy waits                    17 file#
       164 buffer busy waits                    17 file#
```

```
      SID EVENT                          P1 P1TEXT
-------- ---------------------- ---------- ------------------------------------------
      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#
      227 db file sequential read          17 file#
      336 db file sequential read          17 file#

      SID EVENT                          P1 P1TEXT
-------- ---------------------- ---------- ------------------------------------------
      333 db file sequential read          17 file#
     ……
      330 db file scattered read           17 file#
      310 db file scattered read           17 file#
      302 db file scattered read           17 file#
      299 db file scattered read           17 file#
       89 db file scattered read           17 file#
        5 smon timer                      300 sleep time

      SID EVENT                          P1 P1TEXT
-------- ---------------------- ---------- ------------------------------------------
       20 SQL*Net message to client     1952673792 driver id
      103 SQL*Net message to client     1650815232 driver id
     ……
      148 SQL*Net more data from client  1952673792 driver id
      291 SQL*Net more data from client  1952673792 driver id

244 rows selected.
```

简要说明一下 v$session_wait 视图，该视图记录了当前数据库系统中活动 session 正在等待的资源。

```
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>代表以十进制定义的参数值；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，可以通过 v$event_name 视图来查看这些参数的定义：

```
SQL> desc v$event_name
 Name         Null?    Type
 ----------- -------- ------------------------------------------
 EVENT#                NUMBER
 NAME                  VARCHAR2(64)
 PARAMETER1            VARCHAR2(64)
 PARAMETER2            VARCHAR2(64)
 PARAMETER3            VARCHAR2(64)
```

对于本案例，可以发现存在大量 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。

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，就可能需要检查这些数据表是否已进行了正确的设置。

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。

关于常见的等待事件，可以在以下网址找到更为详细的说明：http://www.eygle.com/statspack/statspack12.htm。

22.5　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就应该使用绑定变量，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.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=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，而这里有 22 万条记录，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)
……
 VC2NOTES                                 VARCHAR2(1000)
 INTINFOTYPE                     NOT NULL NUMBER(38)
 VC2PRIZEFLAG                             VARCHAR2(1)
 VC2DESC                                  VARCHAR2(1000)
```

22.6　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)
```

22.7 观察系统状况

可以看到原大量等待消失：

```
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

      SID EVENT                          P1 P1TEXT
---------- ------------------------------ ---------- -----------------------------------
      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 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
```

```
 20 0 0 5421792 1503488 38 434 136 0 0 0 0  0  0  2  0 2931 7795 2622 91  9  0
 23 1 0 5416080 1500632 95 734 56 0 0 0  0  0  0  0  0 2949 8057 2598 89 11  0
 26 0 0 5412016 1498480 210 1170 21 5 5 0 0 0  2  1  0 3301 9647 3116 90 10  0
 25 0 0 5394912 1490160 242 1606 56 0 0 0 0 0  0  1  0 3133 9318 2850 89 11  0
 40 0 0 5390200 1488112 162 1393 66 0 0 0 0 0  0  0  0 2848 9080 2502 90 10  0
 40 0 0 5377120 1481792 136 1180 120 2 2 0 0 0 1  1  0 2846 9099 2593 92  8  0
 36 0 0 5363216 1475168 134 1169 53 0 0 0 0 0  3  2  0 2871 7989 2621 88 12  0
 39 0 0 5348936 1469160 157 1448 210 0 0 0 0 0 0  0  0 3660 10062 3480 88 12 0
 35 0 0 5344552 1466472 7 15 56 0  0  0  0  0  0  0  0 2885 7663 2635 92  8  0
 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
 40 1 0 5290040 1439208 315 3894 8 2 2 0 0  0  1  6  0 3631 13414 5206 61 9 30
 0 1 0 5237192 1414744 731 6749 45 0 0 0 0  0  2  7  0 3264 13558 4941 52 14 34
 0 0 0 5163632 1380608 747 6585 10 0 0 0 0  0  0  1  0 2617 12291 3901 46 12 41
 1 0 0 5090224 1348152 712 6079 29 0 0 0 0  0  0  6  0 2825 12416 4178 50 12 39
 1 0 0 5023672 1317296 714 6183 24 0 0 0 0  0  0  5  0 3166 12424 4745 47 13 40
 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  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 2298 3930 3324 35  2 63
 2 0 0 4926088 1270824 34 560 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  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 2309 3908 3321 35  1 64
 0 0 0 5562048 1577000 16 234 0 0  0  0  0  0  0  0  0 2507 5187 3433 35  8 57
 0 0 0 5665672 1623848 252 1896 8 2 2 0  0  0  1  0  0 2091 6548 2939 34  5 61
 0 0 0 5654752 1618208 5 173 16 0  0  0  0  0  0  0  0 2226 4218 3051 35  4 60
 0 0 0 5727024 1651120 28 254 0 0  0  0  0  0  0  0  0 2126 4224 2982 38  2 60
 0 0 0 5723184 1648880 93 562 8 2  2  0  0  0  1  1  0 2371 5140 3432 38  3 59
 0 0 0 5730744 1652512 7 177 26 2  2  0  0  0  1  0  0 2465 4442 3575 36  3 61
```

至此，此问题得以解决。

22.8　性能何以提高

回答这个问题似乎是多余的，在这里重申一点：有效的降低 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.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  ;

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
    14279  bytes sent via SQL*Net to client
     2222  bytes received via SQL*Net from client
       25  SQL*Net roundtrips to/from client
        2  sorts (memory)
        0  sorts (disk)
      352  rows processed
```

(2) 使用索引的性能

```
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;

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
    14279  bytes sent via SQL*Net to client
     2222  bytes received via SQL*Net from client
```

```
          25  SQL*Net roundtrips to/from client
           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。

22.9　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，就必须能够快速地诊断并解决各种复杂问题。

作者简介

盖国强，网名 eygle，ITPUB Oracle 管理版版主，ITPUB 论坛超级版主，曾任 ITPUB MS 版主。CSDN eMag Oracle 电子杂志主编。

曾任职于某国家大型企业，服务于烟草行业，开发过基于 Oracle 数据库的大型 ERP 系统，属国家信息产业部重点工程。同时负责 Oracle 数据库管理及优化，并为多家烟草企业提供 Oracle 数据库管理、优化及技术支持。

目前任职于北京某电信增值业务系统提供商企业，首席 DBA，负责数据库业务。管理全国 30 多个数据库系统。项目经验丰富，曾设计规划及支持中国联通增值业务等大型数据库系统。

实践经验丰富，长于数据库诊断、性能调整与 SQL 优化等。对于 Oracle 内部技术具有深入研究。

高级培训讲师，培训经验丰富，曾主讲 ITPUB DBA 培训及 ITPUB 高级性能调整等主要课程。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

可以在 http://www.eygle.com 上找到关于作者的更多信息。

第 23 章　一条 SQL 导致数据库整体性能下降的诊断及解决

本章介绍了数据库出现性能问题后分析诊断以及解决的整个流程，在这个过程中使用了 Statspack 报告，着重从等待事件、SQL、10053 event 入手，并利用了 Standby Database 来做对比分析从而找到问题的根本。

23.1　现象

早上一来，数据库 load 就比往常高了许多。想想数据库惟一的变化是昨天早上我曾经重新分析过数据库对象。发现数据库 load 很高，首先看 top 发现没有特别异常的进程，在数据库中实时抓取正在运行的 SQL 也没发现异常（通常运行时间非常短的 SQL 是不能被抓取到的）。询问相关应用程序人员，确认最近应用程序没有变动。检查应用程序日志发现今天早上跟往常一样，也没有过多的登录和操作。基本上可以圈定问题是出在数据库服务器本身上。

23.2　诊断与解决

这时候还没有办法确定到底是哪个应用的哪个查询有问题，因为数百个进程的几十台 Server 连着，我不能去及时地追踪。打算等到 10 点过后，抽取 8/9/10 高峰期的整点的 Statspack 报告出来，跟上星期的这个时间产生的报告对比看看。

Statspack 报告对比：

```
report I:
Event                                          Waits    Time (s) Ela Time
---------------------------------------- ------------ ----------- --------
CPU time                                         2,341      42.60
db file sequential read                        387,534       2,255    41.04
global cache cr request                        745,170         231     4.21
log file sync                                   98,041         229     4.17
log file parallel write                         96,264         158     2.88
```

```
report II:
Event                                          Waits      Time (s) Ela Time
---------------------------------------- ------------ ----------- --------
db file sequential read                      346,851       1,606    47.60
CPU time                                       1,175      34.83
global cache cr request                      731,368         206     6.10
log file sync                                 90,556          91     2.71
db file scattered read                        37,746          90     2.66
```

通过对比报告，发现 CPU Time 今天 1 小时内增加了大约 1200 秒（2,341-1,175）。这是一个重大的变化，很显然有两种可能：

（1）今天过多地执行了某些 SQL。

（2）某些 SQL 的执行计划发生变化导致 CPU 使用过多。

接下来对比 SQL 部分内容：

```
Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- ---------
17,899,606      47,667         375.5   55.6  1161.27   1170.22 3481369999
   Module: /home/oracle/AlitalkSrv/config/../../AlitalkSrv/
   SELECT login_id, to_char(gmt_create, 'YYYY-MM-DD HH24:MI:SS')
     from IM_BlackList    where black_id = :b1
```

发现这条 SQL 出现在了今天报告的前列，而以往的报告中该 SQL 根本不排在 Buffer Gets 的前面位置，显然这条 SQL 消耗了大约 1161.27 秒 CPU Time，检查原来的报告：

```
Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
107,937      47,128          2.3    0.8    7.39     6.94 3481369999
   Module: /home/oracle/AlitalkSrv/config/../../AlitalkSrv/
   SELECT login_id, to_char(gmt_create, 'YYYY-MM-DD HH24:MI:SS')
     from IM_BlackList    where black_id = :b1
```

可以发现只消耗了 7.39 秒的 CPU Time。到这个时候基本可以断定，是由于这个 SQL 没有走索引而走了全表扫描。但是为什么会走全表扫描呢，这是一个问题，接下来要检查表的索引：

```
SQL> select index_name,column_name from user_ind_columns where table_name =
'IM_BLACKLIST';

IM_BLACKLIST_PK             LOGIN_ID
IM_BLACKLIST_PK             BLACK_ID
IM_BLACKLIST_LID_IND         BLACK_ID
```

很显然存在着 BLACK_ID 的单独索引，应该正常使用才对。于是在生产库上执行这个 SQL 一看，却发现走了全表扫描。为此我到一个 2 天前的 Standby 的 open read only 状态数据库上查询了一下该索引字段的 histogram（这个时候昨天早上分析对象的日志还没有被应用过去）：

```
sys@OCN> select COLUMN_NAME ,ENDPOINT_NUMBER, ENDPOINT_VALUE , ENDPOINT_ACTUAL_VALUE from
dba_histograms
  2  where table_name = 'IM_BLACKLIST' and column_name = 'BLACK_ID';

COLUMN_NAME      ENDPOINT_NUMBER ENDPOINT_VALUE    ENDPOINT_ACTUAL_VALUE
------------------------------ --------------- --------------
```

```
BLACK_ID                    0           2.5031E+35
BLACK_ID                    1           2.6065E+35
BLACK_ID                    2           2.8661E+35
BLACK_ID                    3           5.0579E+35
BLACK_ID                    4           5.0585E+35
BLACK_ID                    5           5.0585E+35
BLACK_ID                    6           5.0589E+35
BLACK_ID                    7           5.0601E+35
BLACK_ID                    8           5.1082E+35
BLACK_ID                    9           5.1119E+35
BLACK_ID                   10           5.1615E+35
BLACK_ID                   11           5.1616E+35
BLACK_ID                   12           5.1628E+35
BLACK_ID                   13           5.1646E+35
BLACK_ID                   14           5.2121E+35
BLACK_ID                   15           5.2133E+35
BLACK_ID                   16           5.2155E+35
BLACK_ID                   17           5.2662E+35
BLACK_ID                   18           5.3169E+35
BLACK_ID                   19           5.3193E+35
BLACK_ID                   20           5.3686E+35
BLACK_ID                   21           5.3719E+35
BLACK_ID                   22           5.4198E+35
BLACK_ID                   23           5.4206E+35
BLACK_ID                   24           5.4214E+35
BLACK_ID                   25           5.4224E+35
BLACK_ID                   26           5.4238E+35
BLACK_ID                   27           5.4246E+35
BLACK_ID                   28           5.4743E+35
BLACK_ID                   29           5.5244E+35
BLACK_ID                   30           5.5252E+35
BLACK_ID                   31           5.5252E+35
BLACK_ID                   32           5.5272E+35
BLACK_ID                   33           5.5277E+35
BLACK_ID                   34           5.5285E+35
BLACK_ID                   35           5.5763E+35
BLACK_ID                   36           5.6274E+35
BLACK_ID                   37           5.6291E+35
BLACK_ID                   38           5.6291E+35
BLACK_ID                   39           5.6291E+35
BLACK_ID                   40           5.6291E+35
BLACK_ID                   42           5.6311E+35
BLACK_ID                   43           5.6794E+35
BLACK_ID                   44           5.6810E+35
BLACK_ID                   45           5.6842E+35
BLACK_ID                   46           5.7351E+35
BLACK_ID                   47           5.8359E+35
BLACK_ID                   48           5.8887E+35
BLACK_ID                   49           5.8921E+35
BLACK_ID                   50           5.9430E+35
BLACK_ID                   51           5.9913E+35
BLACK_ID                   52           5.9923E+35
```

```
BLACK_ID                              53            5.9923E+35
BLACK_ID                              54            5.9931E+35
BLACK_ID                              55            5.9947E+35
BLACK_ID                              56            5.9959E+35
BLACK_ID                              57            6.0428E+35
BLACK_ID                              58            6.0457E+35
BLACK_ID                              59            6.0477E+35
BLACK_ID                              60            6.0479E+35
BLACK_ID                              61            6.1986E+35
BLACK_ID                              62            6.1986E+35
BLACK_ID                              63            6.1994E+35
BLACK_ID                              64            6.2024E+35
BLACK_ID                              65            6.2037E+35
BLACK_ID                              66            6.2521E+35
BLACK_ID                              67            6.2546E+35
BLACK_ID                              68            6.3033E+35
BLACK_ID                              69            6.3053E+35
BLACK_ID                              70            6.3069E+35
BLACK_ID                              71            6.3553E+35
BLACK_ID                              72            6.3558E+35
BLACK_ID                              73            6.3562E+35
BLACK_ID                              74            6.3580E+35
BLACK_ID                              75            1.1051E+36
```

然后对比了一下当前的 histograms：

```
COLUMN_NAME      ENDPOINT_NUMBER ENDPOINT_VALUE    ENDPOINT_ACTUAL_VALUE
------------------------------ --------------- --------------
BLACK_ID                                0            1.6715E+35
BLACK_ID                                1            2.5558E+35
BLACK_ID                                2            2.7619E+35
BLACK_ID                                3            2.9185E+35
BLACK_ID                                4            5.0579E+35
BLACK_ID                                5            5.0589E+35
BLACK_ID                                6            5.0601E+35
BLACK_ID                                7            5.1100E+35
BLACK_ID                                8            5.1601E+35
BLACK_ID                                9            5.1615E+35
BLACK_ID                               10            5.1624E+35
BLACK_ID                               11            5.1628E+35
BLACK_ID                               12            5.1642E+35
BLACK_ID                               13            5.2121E+35
BLACK_ID                               14            5.2131E+35
BLACK_ID                               15            5.2155E+35
BLACK_ID                               16            5.2676E+35
BLACK_ID                               17            5.3175E+35
BLACK_ID                               18            5.3684E+35
BLACK_ID                               19            5.3727E+35
BLACK_ID                               20            5.4197E+35
BLACK_ID                               21            5.4200E+35
BLACK_ID                               22            5.4217E+35
BLACK_ID                               23            5.4238E+35
BLACK_ID                               24            5.4244E+35
```

```
BLACK_ID                25              5.4755E+35
BLACK_ID                26              5.5252E+35
BLACK_ID                27              5.5252E+35
BLACK_ID                28              5.5252E+35
BLACK_ID                29              5.5283E+35
BLACK_ID                30              5.5771E+35
BLACK_ID                31              5.6282E+35
BLACK_ID                32              5.6291E+35
BLACK_ID                33              5.6291E+35
BLACK_ID                34              5.6291E+35
BLACK_ID                35              5.6299E+35
BLACK_ID                36              5.6315E+35
BLACK_ID                37              5.6794E+35
BLACK_ID                39              5.6816E+35
BLACK_ID                40              5.6842E+35
BLACK_ID                41              5.7838E+35
BLACK_ID                42              5.8877E+35
BLACK_ID                43              5.8917E+35
BLACK_ID                44              5.9406E+35
BLACK_ID                45              5.9909E+35
BLACK_ID                46              5.9923E+35
BLACK_ID                47              5.9923E+35
BLACK_ID                48              5.9946E+35
BLACK_ID                49              5.9950E+35
BLACK_ID                50              5.9960E+35
BLACK_ID                51              5.9960E+35
BLACK_ID                52              5.9960E+35
BLACK_ID                53              5.9960E+35
BLACK_ID                54              5.9960E+35
BLACK_ID                55              5.9960E+35
BLACK_ID                56              5.9960E+35
BLACK_ID                57              6.0436E+35
BLACK_ID                58              6.0451E+35
BLACK_ID                59              6.0471E+35
BLACK_ID                60              6.1986E+35
BLACK_ID                61              6.1998E+35
BLACK_ID                62              6.2014E+35
BLACK_ID                63              6.2037E+35
BLACK_ID                64              6.2521E+35
BLACK_ID                65              6.2544E+35
BLACK_ID                66              6.3024E+35
BLACK_ID                67              6.3041E+35
BLACK_ID                68              6.3053E+35
BLACK_ID                69              6.3073E+35
BLACK_ID                70              6.3558E+35
BLACK_ID                71              6.3558E+35
BLACK_ID                72              6.3558E+35
BLACK_ID                73              6.3558E+35
BLACK_ID                74              6.3580E+35
BLACK_ID                75              1.1160E+36
```

发现原来的 histograms 值分布比较均匀，而昨天分析后的值分布就有一些地方是集中的，参

考上面 75 行中的第 50～56 行部分。于是再做个 10053 event 对比，在分析之前的 Standby Database 上，执行：

```
alter session set events '10053 trace name context forever';
```

然后执行相关的 SQL 再去看 trace 文件：

```
Table stats   Table: IM_BLACKLIST  Alias: IM_BLACKLIST
  TOTAL ::  CDN: 57477  NBLKS:  374  AVG_ROW_LEN:  38
-- Index stats
  INDEX NAME: IM_BLACKLIST_LID_IND  COL#: 2
    TOTAL ::  LVLS: 1   #LB: 219  #DK: 17181  LB/K: 1  DB/K: 2  CLUF: 44331
  INDEX NAME: IM_BLACKLIST_PK  COL#: 1 2
    TOTAL ::  LVLS: 1   #LB: 304  #DK: 57477  LB/K: 1  DB/K: 1  CLUF: 55141
_OPTIMIZER_PERCENT_PARALLEL = 0
*************************************
SINGLE TABLE ACCESS PATH
Column:   BLACK_ID  Col#: 2     Table: IM_BLACKLIST  Alias: IM_BLACKLIST
    NDV: 17181    NULLS: 0        DENS: 5.8204e-05
    NO HISTOGRAM: #BKT: 1 #VAL: 2
  TABLE: IM_BLACKLIST    ORIG CDN: 57477  ROUNDED CDN: 3  CMPTD CDN: 3
  Access path: tsc  Resc:  38  Resp:  38
  Access path: index (equal)
     Index: IM_BLACKLIST_LID_IND
  TABLE: IM_BLACKLIST
     RSC_CPU: 0   RSC_IO: 4
  IX_SEL:  0.0000e+00  TB_SEL:  5.8204e-05
  Skip scan: ss-sel 0  andv 27259
    ss cost 27259
    table io scan cost 38
  Access path: index (no sta/stp keys)
     Index: IM_BLACKLIST_PK
  TABLE: IM_BLACKLIST
     RSC_CPU: 0   RSC_IO: 309
  IX_SEL:  1.0000e+00  TB_SEL:  5.8204e-05
  BEST_CST: 4.00  PATH: 4  Degree:  1
*************************************
OPTIMIZER STATISTICS AND COMPUTATIONS
*************************************
GENERAL PLANS
**********************
Join order[1]: IM_BLACKLIST [IM_BLACKLIST]
Best so far: TABLE#: 0  CST:         4  CDN:         3  BYTES:        75
Final:
  CST: 4  CDN: 3  RSC: 4  RSP: 4  BYTES: 75
  IO-RSC: 4  IO-RSP: 4  CPU-RSC: 0  CPU-RSP: 0
```

在分析之后的当前数据库中，再做一个 10053 trace，得到如下内容：

```
SINGLE TABLE ACCESS PATH
Column:   BLACK_ID  Col#: 2     Table: IM_BLACKLIST  Alias: IM_BLACKLIST
    NDV: 17069    NULLS: 0        DENS: 1.4470e-03
    HEIGHT BALANCED HISTOGRAM: #BKT: 75 #VAL: 75
  TABLE: IM_BLACKLIST    ORIG CDN: 57267  ROUNDED CDN: 83  CMPTD CDN: 83
```

```
  Access path: tsc  Resc:  38  Resp:  38
  Access path: index (equal)
     Index: IM_BLACKLIST_LID_IND
  TABLE: IM_BLACKLIST
     RSC_CPU: 0   RSC_IO: 65
  IX_SEL:  0.0000e+00  TB_SEL:  1.4470e-03
  Skip scan: ss-sel 0  andv 27151
    ss cost 27151
    table io scan cost 38
  Access path: index (no sta/stp keys)
     Index: IM_BLACKLIST_PK
  TABLE: IM_BLACKLIST
     RSC_CPU: 0   RSC_IO: 384
  IX_SEL:  1.0000e+00  TB_SEL:  1.4470e-03
  BEST_CST: 38.00  PATH: 2  Degree:  1
**************************************
OPTIMIZER STATISTICS AND COMPUTATIONS
**************************************
GENERAL PLANS
**********************
Join order[1]: IM_BLACKLIST [IM_BLACKLIST]
Best so far: TABLE#: 0  CST:        38  CDN:        83  BYTES:      2407
Final:
  CST: 38  CDN: 83  RSC: 38  RSP: 38  BYTES: 2407
  IO-RSC: 38  IO-RSP: 38  CPU-RSC: 0  CPU-RSP: 0
```

注意上面加框部分，发现分析之前和之后全表扫描 cost 都是 38，但是分析之后的根据索引扫描却成为了 65，而分析之前是 4。很显然是由于这个查询导致昨天早上分析之后走了全表扫描。于是再对表进行分析，只不过这次没有分析索引字段，而是：

```
analyze table im_blacklist compute  statistics;
```

这样分析之后，dbms_histograms 中的信息如下：

```
COLUMN_NAME      ENDPOINT_NUMBER ENDPOINT_VALUE    ENDPOINT_ACTUAL_VALUE
---------------- --------------- ------------------------------
GMT_CREATE                     0             2452842.68
GMT_MODIFIED                   0             2452842.68
LOGIN_ID                       0             2.5021E+35
BLACK_ID                       0             1.6715E+35
GMT_CREATE                     1             2453269.44
GMT_MODIFIED                   1             2453269.44
LOGIN_ID                       1             6.3594E+35
BLACK_ID                       1             1.1160E+36
```

再执行该 SQL，就走了索引，从而使得数据库的 load 降了下来。

分析这整个过程，我无法知道 Oracle 的走索引 cost 65 是怎么计算出来的，当然这是跟 histograms 有关，但计算方法我却是不清楚的。

这条 SQL 是 bind var，却走了全表扫描，这是由于 9iR2 数据库在对 bind var 的 SQL 进行第一次硬解析时去 histograms 中窥视了数据分布从而根据 cost 选择了 FTS。所以后面继续执行的 SQL，不论是否该走索引，都走了 FTS。这是 9iR2 这个版本特性的弊病。也就是说，这有偶然性因素的存在。但是对于这个表，在做了分析（不分析索引字段）之后不存在 histograms，则 SQL

无论如何都走了索引扫描。

作者简介

冯春培，网络 ID biti_rainy，曾任 ITPUB Oracle 开发版版主，现任 ITPUB Oracle 管理版版主和超级版主。有丰富的 Oracle 实践经验，对数据库的体系结构、备份恢复、SQL 优化、数据库整体性能优化、Oracle Internal 都有深入研究。

开发出身，对数据库应用设计中如何正确地应用 Oracle 特性以扬长避短具有深刻理解。曾于某电信集成公司负责计费系统的开发，然后成为某系统集成公司的 DBA，再辗转在香港一家跨国公司珠海研发中心担任技术负责人（公司主要产品就是 SQL 与数据库优化工具，产品主要销往欧洲和北美），此后成为自由职业者，独立为客户提供 Oracle 数据库的技术服务和高级性能调整等方面的培训，同时提供 ITPUB 在华南和华东地区的培训。

目前服务于国内某大型电子商务网站，维护系统数据库并提供开发支持。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

更多个人作品：http://blog.itpub.net/biti_rainy

第 24 章　Library Cache Lock 成因和解决方法的探讨

在执行一些 DML、DDL，甚至在 desc tablename 等操作的时候，会话可能会出现 hang 住的情况；有的时候，当使用 create or replace procedure/function 等语句修改 Procedure 和 Function 的时候，会话也可能会 hang 住，这是为什么呢？

当出现上述情况时，可以通过 v$session_wait 视图查询进程的等待事件，当然，这种情况下，该 session 的等待事件通常都是 “ Library Cache Lock ”。

24.1　几个相关的概念

为了彻底搞清楚会话 hang 住的原委，首先从几个概念说起。

24.1.1　什么是库高速缓存（Library Cache）

库高速缓存，也称为共享 SQL 区，用来存储 SQL 语句和 PL/SQL 块。它使用 LRU 算法管理缓冲区。当用户发出一条 SQL 语句，如果可以在库缓冲区中找到该语句的分析树和执行计划，那么 Oracle 就会使用这些执行计划而不需要再进行 SQL 语句的分析和解析，这也就是通常所说的软解析。如果总是可以在内存中找到用户发出的 SQL 语句，并且可以找到这些语句的分析树和执行计划，那么系统就可以节省大量的资源。

库高速缓存中包含：

- SQL 语句（文本）和 PL/SQL 块
- 分析树（语句的已编译版本）
- 执行计划（执行语句时要采取的步骤）

其中，PL/SQL 块包括过程、函数、程序包、触发器和匿名 PL/SQL 块。

24.1.2　一个 SQL 语句的处理流程

SQL 语句的处理需要 4 个阶段，即分析 (Parse)、定义 (Define)、绑定 (Bind)、执行 (Execute)，

如果是查询语句（select）还要有获取数据（Fetch）的阶段：

（1）分析阶段：SQL 语句从用户进程传递到服务器进程（Server Process），服务器进程根据每个字符的 ASCII 值对语句进行散列（hash），然后在共享池中搜索 SQL 语句的现有副本（根据传递到共享池中库缓存内对应的某个位置的地址）。如果该 SQL 语句在散列地址中不存在（在得到散列内存地址后，服务器进程查看是否已经存在一条相符的语句），则需要进行硬分析，即服务器进程对 SQL 语句进行语法检查、对象解析和权限检查，执行成功后建立分析树和 SQL 语句的执行计划。

（2）定义阶段：用户进程和服务器进程交换 SQL 语句中应用到的各个列的数据类型信息。这里需要用 NET8i（Oracle 8i 及其以后的版本）在用户进程和服务器进程之间进行数据类型解析，即解决了由于字符集不同，在两端将数据映射为相应的表示。如果是 Oracle 8 版本使用 NET8，如果是 Oracle 8 以前的版本，使用 SQL*Net。

（3）绑定阶段：处理 SQL 语句中引用的绑定变量（如:b1、:v1）。绑定变量的使用对“ 重用 SQL ”（也叫共享 SQL）有很大的好处，它可以减少共享池资源的争用。

（4）执行阶段：对于 DML 操作（insert、update、delete），服务器进程（Server Process）应用执行计划并根据需要从文件（数据段文件、索引段文件、回滚段文件）中读取数据块，在内存中进行数据处理，然后返回给用户进程（User Process）操作成功或失败的信号；对于查询操作（select），服务器进程标识选定的行，准备检索数据。

（5）获取数据阶段：这个阶段只有查询语句（select）会有，服务器进程根据需要把相应的数据块读入到数据库缓冲区高速缓存（DB Buffer）中，并应用执行计划把行返回给用户进程。在获取数据（Fetch）阶段服务器进程会根据需要对数据行进行排序。

24.1.3 硬分析（Hard Parse）

这个过程会完成对 SQL 语句进行语法检查，执行数据字典查找来验证表和列的定义，获取对象的分析锁以便在语句的分析过程中保持对象的定义不变，检查用户访问引用方案对象的权限，确定语句的最佳执行计划，将语句和执行计划载入共享的 SQL 区。

通常 SQL 语句在第一次执行时需要进行硬分析。

如果一条 SQL 语句在此执行前，其引用到的对象结构经过更改（alter）等操作，或者应用到的对象的统计数据集经过重新分析（analyze）了，那么该 SQL 语句在内存中被标记为 INVALID，因此执行此 SQL 语句的服务器进程就必须根据最新的对象结构和统计数据集重新建立分析树和执行计划，即执行硬分析。

24.1.4 软分析（Soft Parse）

如果服务器进程根据 hash 地址在库缓存中找到了相符的 SQL 语句，以及它的分析树和执行计划，就进行软分析（不需要重新建立分析树和执行计划）。

24.1.5 分析树

分析树实际上是以树的形式重新格式化和结构化 SQL 语句。

24.1.6　执行计划

执行计划是从分析树中推导出来的检索数据的方法。

了解了关于库高速缓存（Library Cache）的相关概念后，可以看到 SQL 语句在第一次执行时需要进行硬分析，即在 hash 地址中找不到 SQL 语句，服务器进程不得不执行更多的递归 SQL 来建立一棵分析树和一个执行计划。如果可以跳过建立分析树和执行计划的过程（即执行软分析），就可以节省大量的资源（如 CPU 利用率），进而减少内部资源的争用（库缓冲区 Latch 的等待）。硬分析要花时间和资源规划如何执行语句，而软分析着重于执行语句，这就是硬分析需要“ 软化 ”的原因。

通常在设置共享池尺寸的时候，建议给共享池一个初始尺寸，然后需要根据系统测试，再确定系统在存在共享池尺寸导致的性能瓶颈时是否要进一步调整共享池。当已经确定需要调整共享池时，一般主要是关心库高速缓存，因为 Oracle 算法倾向于把字典数据放在内存的时间较长，而库高速缓存数据放在内存的时间较短。

24.2　了解 Library Cache Lock

下面介绍一下几种容易引起 Library Cache Lock 的情况和防患的方法。

24.2.1　几种容易引起 Library Cache Lock 的情况

现在回过头看看最初的问题，当在执行一些 DML、DDL，甚至 desc tablename 等操作时，会话就 hang 住了，比如，当会话 1（session 1）在对一个表执行 DML 或者 DDL，与此同时还有另一个会话，姑且称之为会话 2（session 2），这个会话 2 也在对这个表执行 DDL（如 alter table）操作，当会话 2 的完成需要很长时间时（以操作的具体的数据量而定），会话 1 就会 hang 住。

还有一种情况，当使用 create or replace procedure/function 等语句修改 Procedure 和 Function 时，会话也会 hang 住，比如，当会话 1（session 1）在修改一个 package，与此同时还有另一个会话——会话 2（session 2），如果这个会话 2 正在执行会话 1 所修改的 package 中的 Procedure 或者 Function，会话 1 就会 hang 住。

这时，当上述两种情况发生时，使用 v$session_wait 视图，就会发现大量的 Library Cache Lock 等待事件。这个事件或者引起数据库性能严重下降，或者使进程挂起。

上述问题通常发生在大量使用数据库存储 PL/SQL 块的并发应用程序中。为了预防该问题的发生，在对 Package/Procedure/Function/View 进行编译和分析时，就需要先确定此时没有其他人正在编译和分析相同的对象，即确保没有其他人也在此时改变这些需要重定义（drop 和 recreate）的对象的定义。因为编译或分析（parse）package 或 procedure 或 function 或 view 时，Oracle 需要先取得“ Library Cache Lock ”和“ Library Cache Pin ”以保证在编译或分析（parse）期间没有 session 正使用此 Object。

24.2.2 几种防患的方法

1. 检查谁正在使用某个对象

这里推荐一种方法来发现当前谁正在使用某个 Package/Procedure/Function/View：

```
SQL> create or replace procedure who_is_using(obj_name varchar2) is
  2  begin
  3   dbms_output.enable(1000000);
  4     for i in (SELECT distinct b.username，b.sid
  5              FROM SYS.x$kglpn a，v$session b，SYS.x$kglob c
  6              WHERE a.KGLPNUSE = b.saddr
  7                and upper(c.KGLNAOBJ)  like upper(OBJ_NAME)
  8                and a.KGLPNHDL = c.KGLHDADR) loop
  9     dbms_output.put_line('('||to_char(i.sid)||') - '||i.username);
 10     end loop;
 11  end;
 12  /

Procedure created.

SQL> execute who_is_using('STANDARD%');
 (9) - SYS

PL/SQL procedure successfully completed.
```

2. 维护和操作中的注意事项

除了上述方法以外，在对 PL/SQL 存储过程中经常引用到的 Object 进行修改，授权，收回授权时必须非常小心。比如说，在当对用户的权限进行管理即进行“ grant ”、“ revoke ”时，通过角色来对最终用户进行授权或收回授权，而不要用显性的方式即直接对最终用户授权或收回授权，从而避免产生 Library Cache Pin 和 Library Cache Lock 等问题；在对数据库对象做 DDL 时尽量安排在非业务高峰期来做等。

3. 其他防患措施

另外，可以在数据库中设置参数“ _SQLEXEC_PROGRESSION_COST=0 ”以避免 session 因等待 Library Cache Pin 和 Library Cache Lock 而导致系统性能严重下降或者挂起。

实际上，解决这些问题大多要依靠应用程序的开发和维护，应用程序开发商应该考虑到某些方案的决策可能会给应用程序的伸缩性和性能带来负面影响。

24.3 解决问题的方法

这里用一个实际的案例来讲述，当发生因为 Library Cache Lock 而导致进程挂起的解决方法。

24.3.1　使用 X$KGLLK 和 systemstate 事件解决问题

1. 问题描述

一个下午，忽然接到应用人员的报告说“任何对表 CSNOZ629926699966 的操作都会 hang，包括 desc CSNOZ629926699966”。

例如：

```
ora9i@cs_dc02:/ora9i > sqlplus pubuser/pubuser

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jan 10 10:11:06 2005

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 and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production

SQL> conn pubuser/pubuser
Connected.
SQL> desc CSNOZ629926699966

……

这个进程 hang 了

……
```

询问了一下之前有无特别的操作，业务人员说很久以前执行了脚本，但是该脚本运行很久都没有结果，然后他就退出了会话（异常退出），再之后，就出现了上面的情况。他执行的脚本内容如下：

```
$ cat CSNOZ629926699966.sh
#!/bin/sh
sqlplus pubuser/pubuser@csmisc << EOF  #use your username/password

create table CSNOZ629926699966 as select * from CSNOZ62992266cs
where  mid  not  in  (  select  mid  from  pubuser.SUBSCRIPTION_BAK_200412@newdb  where servid='020999011964' and status in ('A'，'B'，'S'));

exit;
$
$
```

2. 检查 LOCK 信息

首先看看是否存在某个锁的 equeue 信息，导致了上述问题：

```
ora9i@cs_dc02:/ora9i > sqlplus "/ as sysdba"
```

```
SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jan 10 10:19:13 2005

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 and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production

SQL> select * from v$lock where block=1;

no rows selected
```

结果，看到目前没有锁的阻塞信息。接下来，查找对象加锁的情况，看看是否可以发现一些线索：

```
SQL> select xidusn， object_id， session_id， locked_mode from v$locked_object;
……
 XIDUSN  OBJECT_ID SESSION_ID LOCKED_MODE
---------- ---------- ---------- -----------
   14        18        37         3
……
```

果然，通过查找 v$locked_object，发现了一个可疑的会话 SID 37，它始终持有上面的锁，进一步查看被加锁的对象的信息，发现居然是 “ obj$ ”：

```
SQL> select object_name，owner，object_type from dba_objects where object_id=18;

……

OBJECT_NAME                    OWNER                          OBJECT_TYPE
------------------------------ ------------------------------ ------------------
OBJ$                          SYS                           TABLE

……
```

奇怪怎么一直有这个锁？

初步猜测是由于 SID 为 37 的会话执行了上面的 DDL 语句，并在语句未完成前异常退出，造成了所有访问那个（DDL 语句中涉及到的）对象的进程都 hang 住了。

3. 检查等待事件

接下来，看看有什么等待事件：

```
SQL> select event， p1，p2，sid from v$session_wait
   where event not like 'SQL*%' and event not like 'rdbms%';

EVENT                                                P1          P2       SID
---------------------------------------------- ---------- ------------- -----------
pmon timer                                          300           0          1
ges remote message                                   32           0          4
gcs remote message                                   64           0          5
gcs remote message                                   64           0          7
smon timer                                          300           0         19
```

```
library cache lock                              1.3835E+19   1.3835E+19        30
wakeup time manager                                      0            0        22

7 rows selected.
```

注意到 Library Cache Lock 等待非常显著。P1 是句柄地址（Handle Address），也就是 Library Cache Lock 发生的地址。P2 是一个状态对象，在这里，它表示在对象上加载的锁的地址（Lock Address）。P1 和 P2 都是科学计数法表示的十进制数。

这些信息再次证实了上面的猜测，SID 37 阻塞了 SID 30。

4. 设置 10046 事件

接下来，为了了解这两个会话（session）的详细信息，需要找出这两个可疑进程的 SID 和 SERIAL#，然后对它们设置 10046 事件，设置方法如下：

```
SQL> select sid，serial# from v$session where sid in (30，37);

      SID    SERIAL#
---------- ----------
       30     24167
       37      2707

SQL> exec dbms_system.set_ev(30，24167，10046，12，'');

PL/SQL procedure successfully completed.

SQL> exec dbms_system.set_ev(37，2707，10046，12，'');

PL/SQL procedure successfully completed.
```

跟踪期间再次测试一下，看看有没有其他线索。再新开一个会话，找出其 SID、SERIAL#和 SPID 等信息：

```
ora9i@cs_dc01:/ora9i > sqlplus pubuser/pubuser

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jan 10 11:36:25 2005

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 and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production

SQL> select distinct sid from v$mystat;

      SID
----------
       33

SQL> select sid，serial# from v$session where sid=33;

      SID    SERIAL#
```

```
---------- ----------
       33      6639

SQL> SELECT SPID，PID FROM V$PROCESS WHERE ADDR=(SELECT PADDR FROM V$SESSION WHERE SID=37);

SPID             PID
------------ ----------
20552             26

SQL> SELECT SPID，PID FROM V$PROCESS WHERE ADDR=(SELECT PADDR FROM V$SESSION WHERE SID=30);

SPID             PID
------------ ----------
22580             28

SQL> show parameter dump

NAME                                 TYPE       VALUE
------------------------------------ ----------- ------------------------------
background_core_dump                  string     partial
background_dump_dest                  string     /ora9i/app/oracle/admin/csmisc
                                                 /bdump
core_dump_dest                       string     /ora9i/app/oracle/admin/csmisc
                                                 /cdump
max_dump_file_size                    string     UNLIMITED
shadow_core_dump                     string     partial
user_dump_dest                       string     /ora9i/app/oracle/admin/csmisc
                                                 /udump
SQL>
```

然后，再尝试对 CSNOZ629926699966 表进行操作，看看结果如何：

```
SQL> desc CSNOZ629926699966

Hang…
```

可以看到，又 hang 住了。于是中断这个操作（CTRL+C）：

```
SQL> desc CSNOZ629926699966
ERROR:
ORA-01013: user requested cancel of current operation

SQL> select tname from tab where tname='CSNOZ629926699966';

no rows selected.
```

查看 PUBUSER 用户下的这个表，居然不存在！

进一步证实了前面的猜测，也就是说会话 37 阻塞了其他所有操作表 CSNOZ629926699966 的会话，造成了进程的 hang，当然，包括上面的 SID 30 和 SID 33 的 DDL 语句。

现在，结束 10046 的事件跟踪：

```
SQL> exec dbms_system.set_ev(30，24167，0，0，'');

PL/SQL procedure successfully completed.

SQL> exec dbms_system.set_ev(37，2707，0，0，'');
```

```
PL/SQL procedure successfully completed.
```

根据上面记录的信息，可以知道这两个会话产生的跟踪信息分别如下。

SID 为 30 的会话，产生的跟踪文件为：

```
/ora9i/app/oracle/admin/csmisc/udump/csmisc2_ora_22580.trc
```

SID 为 37 的会话，产生的跟踪文件为：

```
/ora9i/app/oracle/admin/csmisc/udump/csmisc2_ora_20552.trc
```

接下来，仔细研究一下这两个文件的 trace 文件，可以看到 SID 为 30 的会话，产生的跟踪文件（csmisc2_ora_22580.trc）的主要内容是：

```
/ora9i/app/oracle/admin/csmisc/udump/csmisc2_ora_22580.trc
Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
With the Partitioning and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production
ORACLE_HOME = /ora9i/app/oracle/product/920
System name: HP-UX
Node name: cs_dc02
Release: B.11.11
Version: U
Machine: 9000/800
Instance name: csmisc2
Redo thread mounted by this instance: 2
Oracle process number: 28
Unix process pid: 22580，image: oracle@cs_dc02 (TNS V1-V3)

*** 2005-01-10 11:31:49.416
*** SESSION ID:(30.24167) 2005-01-10 11:31:49.354
WAIT #0: nam='library cache lock' ela= 507258 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 505686 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507678 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507595 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507880 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507317 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507703 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507683 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 508265 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507100 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507684 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 505889 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507731 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507650 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507604 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301
WAIT #0: nam='library cache lock' ela= 507698 p1=-4611686013547141416 p2=-4611686013691716064 p3=1301

……
```

可以看到 SID 30 的跟踪文件中的等待事件就是在 v$session_wait 视图中看到的 Library Cache Lock。再看看 SID 为 37 的会话，产生的跟踪文件（csmisc2_ora_20552.trc）的主要内容如下：

```
Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
With the Partitioning and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production
```

```
ORACLE_HOME = /ora9i/app/oracle/product/920
System name: HP-UX
Node name: cs_dc02
Release: B.11.11
Version: U
Machine: 9000/800
Instance name: csmisc2
Redo thread mounted by this instance: 2
Oracle process number: 26
Unix process pid: 20552, image: oracle@cs_dc02 (TNS V1-V3)

*** 2005-01-10 11:33:22.702
*** SESSION ID:(37.2707) 2005-01-10 11:33:22.690
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:35:07.452
WAIT #1: nam='SQL*Net message from dblink' ela= 102293555 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 3 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:36:55.980
WAIT #1: nam='SQL*Net message from dblink' ela= 105969709 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:39:05.416
WAIT #1: nam='SQL*Net message from dblink' ela= 126390826 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:41:12.878
WAIT #1: nam='SQL*Net message from dblink' ela= 124461520 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:43:01.285
WAIT #1: nam='SQL*Net message from dblink' ela= 105859385 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
*** 2005-01-10 11:44:48.200
WAIT #1: nam='SQL*Net message from dblink' ela= 104397696 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0

……
```

5. 使用 systemstate 事件

现在来 dump 系统状态（systemstate），看看更详细的信息。

首先简单地介绍一下 event systemstate。很多人把 systemstate 事件理解为 dump 发生的那一刻的系统内所有进程的信息，这是个错误的概念，事实上，转储 system state 产生的跟踪文件是从 dump 开始到 dump 任务完成之间这段时间内的系统内所有进程的信息。dump systemstate 产生的跟踪文件包含了系统中所有进程的进程状态等信息。每个进程对应跟踪文件中的一段内容，反映该进程的状态信息，包括进程信息、会话信息、enqueues 信息（主要是 lock 的信息）、缓冲区的信息和该进程在 SGA 区中持有的（held）对象的状态等信息。

那么通常在什么情况下使用 systemstate 比较合适呢？Oracle 推荐使用 systemstate 事件的几种情况是：

- 数据库 hang 住了。
- 数据库很慢。

- 进程正在 hang。
- 数据库出现某些错误。
- 资源争用。

dump systemstate 的语法为：

```
ALTER SESSION SET EVENTS 'immediate trace name systemstate level 10';
```

也可以使用 ORADEBUG 实现这个功能：

```
ORADEBUG DUMP SYSTEMSTATE level 10
```

如果希望在数据库发生某种错误时触发 systemstate 事件，可以在参数文件（spfile 或者 pfile）中设置 event 参数，例如，当系统发生死锁（出现 ORA-00060 错误）时 dump systemstate：

```
event = "60 trace name systemstate level 10"
```

现在，使用上面的方法 dump 系统状态来看：

```
SQL> ALTER SESSION SET EVENTS 'IMMEDIATE TRACE NAME SYSTEMSTATE LEVEL 8';

Session altered.

SQL> host
ora9i@cs_dc02:/ora9i >cd /ora9i/app/oracle/admin/csmisc/udump
ora9i@cs_dc02:/ora9i/app/oracle/admin/csmisc/udump > ll -ctl
-rw-r-----  1 ora9i     dba      1070863 Jan 10 13:02 csmisc2_ora_22580.trc
-rw-r-----  1 ora9i     dba      1345368 Jan 10 13:01 csmisc2_ora_22568.trc
-rwxrwxrwx  1 ora9i     dba        44114 Jan 10 12:52 ass1015.awk
-rw-r-----  1 ora9i     dba       407133 Jan  7 15:10 csmisc2_ora_2708.trc
-rw-r-----  1 ora9i     dba          640 Jan  7 14:48 csmisc2_ora_835.trc
-rw-r-----  1 ora9i     dba         1590 Dec 30 22:50 csmisc2_ora_16244.trc
-rw-r-----  1 ora9i     dba      1308403 Dec 30 22:44 csmisc2_ora_16033.trc
-rw-r-----  1 ora9i     dba          616 Dec 28 14:16 csmisc2_ora_2176.trc
-rw-r-----  1 ora9i     dba          644 Dec  8 18:22 csmisc2_ora_21083.trc
ora9i@cs_dc02:/ora9i/app/oracle/admin/csmisc/udump >
ora9i@cs_dc02:/ora9i/app/oracle/admin/csmisc/udump > mailx -s "22568" zhangdp@aspire-tech.com
< csmisc2_ora_22568.trc
```

这个跟踪文件很大（因为它包含了所有进程的信息），那么从哪里开始看起呢？

首先，通过在跟踪文件中查找字符串“ waiting for 'library cache lock' ”，可以找到被阻塞进程的信息：

```
PROCESS 28: ----------------被阻塞的 Oracle 进程，这里 PROCESS 28 对应了 V$PROCESS 中的 PID 的值，
    也就是说可以根据这一信息在 V$PROCESS 和 V$SESSION 找到被阻塞的会话的信息
  ----------------------------------------
  SO: c000000109c83bf0, type: 2, owner: 0000000000000000, flag: INIT/-/-/0x00
  (process) Oracle pid=28, calls cur/top: c00000010b277890/c00000010b277890, flag: (0) -
          int error: 0, call error: 0, sess error: 0, txn error 0
  (post info) last post received: 17 24 6
            last post received-location: ksusig
            last process to post me: c000000109c840f8 25 0
            last post sent: 0 0 15
            last post sent-location: ksasnd
            last process posted by me: c000000109c7ff90 1 6
    (latch info) wait_event=0 bits=0
    Process Group: DEFAULT, pseudo proc: c000000109eefda0
    O/S info: user: ora9i, term: pts/th, ospid: 22580
```

```
        ----------------该进程的操作系统进程号，对应于 V$PROCESS 中的 SPID
        OSD pid info: Unix process pid: 22580， image: oracle@cs_dc02 (TNS V1-V3)
        ----------------------------------------
        SO: c000000109f02c68， type: 4， owner: c000000109c83bf0， flag: INIT/-/-/0x00
        (session) trans: 0000000000000000， creator: c000000109c83bf0， flag: (100041) USR/-
BSY/-/-/-/-/-
                DID: 0002-001C-00000192， short-term DID: 0000-0000-00000000
                txn branch: 0000000000000000
                oct: 0， prv: 0， sql: c00000011f8ea068， psql: c00000011f8ea068， user: 50/PUBUSER
        O/S info: user: ora9i， term: ， ospid: 22536， machine: cs_dc02
                program: sqlplus@cs_dc02 (TNS V1-V3)
        application name: SQL*Plus， hash value=3669949024
        waiting for 'library cache lock' blocking sess=0x0 seq=18589 wait_time=0
                  handle   address=c000000122e2a6d8  ，     lock   address=c00000011a449e20  ，
100*mode+namespace=515

    ……

        SO: c00000010b277890， type: 3， owner: c000000109c83bf0， flag: INIT/-/-/0x00
        (call) sess: cur c000000109f02c68， rec 0， usr c000000109f02c68; depth: 0
          ----------------------------------------
          SO: c00000011a449e20， type: 51， owner: c00000010b277890， flag: INIT/-/-/0x00
          LIBRARY OBJECT LOCK: lock=c00000011a449e20 handle=c000000122e2a6d8 request=S
          call pin=0000000000000000 session pin=0000000000000000
          htl=c00000011a449e90[c00000011a4bc350，c00000011a4bc350] htb=c00000011a4bc350
          user=c000000109f02c68 session=c000000109f02c68 count=0 flags=[00] savepoint=463
          the rest of the object was already dumped

    ……
```

注意下面的信息：

```
    ……

    waiting for 'library cache lock' blocking sess=0x0 seq=18589 wait_time=0
    handle address=c000000122e2a6d8， lock address=c00000011a449e20， 100*mode+namespace=515

    ……
```

这段信息说明 Oracle PID 为 28 的进程（PROCESS 28），正在等待 Library Cache Lock，通过“ handle address=c000000122e2a6d8 ”可以找到阻塞它的会话的 Oracle PID 信息。

同时，还注意到下面这段信息：

```
      LIBRARY OBJECT LOCK: lock=c00000011a449e20 handle=c000000122e2a6d8 request=S
      call pin=0000000000000000 session pin=0000000000000000
      htl=c00000011a449e90[c00000011a4bc350，c00000011a4bc350] htb=c00000011a4bc350
      user=c000000109f02c68 session=c000000109f02c68 count=0 flags=[00] savepoint=463
```

这里就是阻塞 PROCESS 28 进程的会话的信息。

简单地记住这个依据的要点是：waiting session 的 handle address 值对应于 blocking session 的 handle 的值。

回过头来，看看这个 handle address 的值，它对应于上面在 v$session_wait 视图中看到的 P1 和 P2 的值：

```
    SQL> select to_number('C000000122E2A6D8', 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX') from dual;
```

```
TO_NUMBER('C000000122E2A6D8', 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX')
------------------------------------------------------------
                                                 1.3835E+19
```

这个值就对应于在 v$session_wait 中看到的 Library Cache Lock 的值。

6. 介绍 X$KGLLK

这里，首先简单地介绍一下 X$KGLLK，这个基表保存了库缓存中对象的锁的信息，它对于解决这类问题特别有用，其名称的含义如下：

[K]ernel Layer
[G]eneric Layer
[L]ibrary Cache Manager （defined and mapped from kqlf）
Object Locks
X$KGLLK - Object [L]oc[K]s

KGLNAOBJ 列包含了在 Library Cache 中的对象上执行命令的语句的前 80 个字符（其实从这里也可以大大缩小范围了）。X$KGLLK.KGLLKUSE 和 X$KGLLK.KGLLKSES 对应于跟踪文件中的 owner 的值 X$KGLLK.KGLLKADR。

X$KGLLK.KGLLKHDL 对应于跟踪文件中的 handle 的值（handle=C000000122E2A6D8），也就是 Library Cache Lock 的地址。

X$KGLLK.KGLLKPNS 对应于跟踪文件中的 session pin 的值。X$KGLLK.KGLLKSPN 对应于跟踪文件中的 SAVEPOINT 的值。

接下来，就根据 c000000122e2a6d8 地址，在 X$KGLLK 中查找当前在 Library Cache 中相应的锁信息：

```
SQL> l
  1  select INST_ID, USER_NAME, KGLNAOBJ, KGLLKSNM, KGLLKUSE, KGLLKSES, KGLLKMOD, KGLLKREQ, KGLLKPNS,
KGLLKHDL
  2* from X$KGLLK where KGLLKHDL = 'C000000122E2A6D8' order by KGLLKSNM, KGLNAOBJ
SQL> /

   INST_ID USER_NAME  KGLNAOBJ   KGLLKSNM KGLLKUSE  KGLLKSES KGLLKMOD  KGLLKREQ KGLLKPNS  KGLLKHDL
---------- --------- ------------- ------ -------- -------- -------- -------- --------- ----------
         2 PUBUSER   CSNOZ629926699966   30 C000000109F02C68 C000000109F02C68  0  2 00  C000000122E2A6D8
         2 PUBUSER   CSNOZ629926699966   37 C000000108C99E28 C000000108C99E28  3  0 00  C000000122E2A6D8
```

7. 解决问题

按照 Oracle 推荐的结束一个 session 的方法，现在应该使用 “ alter system kill session ” 命令 kill 掉 SID 37，得到了 ORA-00031 输出：

```
SQL> alter system kill session '37, 2707';

alter system kill session '37, 2707'
*
ERROR at line 1:
ORA-00031: session marked for kill

SQL>
```

于是，检查 sessiion 37 的状态：

```
SQL> set linesize 150
SQL> col program for a50
SQL> select sid，serial#，status，username，program from v$session where sid=37;

      SID    SERIAL# STATUS   USERNAME                    PROGRAM
--------- --------- ------- ------------------       ------------------------------
37           2707    KILLED   PUBUSER                     sqlplus@cs_dc02 (TNS V1-V3)

SQL>
```

再次证实了最初的想法：有人在执行了某个需要运行很久的 DDL（多数是语句效率低，当然不排除遭遇 Bug 的可能），然后没等语句结束就异常退出了会话。

这个例子中，在上面的跟踪文件已经找到了该会话对应的操作系统进程（SPID），如果在其他情况下，如何找到这种状态为 “ KILLED ” 的操作系统进程号（SPID）呢？下面给出了一个方法，可以参考：

```
SQL> l
  1  SELECT s.username，s.status，
  2  x.ADDR，x.KSLLAPSC，x.KSLLAPSN，x.KSLLASPO，x.KSLLID1R，x.KSLLRTYP，
  3  decode(bitand (x.ksuprflg，2)，0，null，1)
  4  FROM x$ksupr x，v$session s
  5  WHERE s.paddr(+)=x.addr
  6  and bitand(ksspaflg，1)!=0
  7* and s.sid=37
SQL> /

USERNAME                 STATUS   ADDR             KSLLAPSC   KSLLAPSN KSLLASPO      KSLLID1R KS D
------------------------ -------- ------------ ---------- ---------- ---------- --------- -- -
PUBUSER                  KILLED   C000000109C831E0       41       15 16243             17
```

x$ksupr.ADDR 列的值对应了 v$process 中的 ADDR 的值，知道了这个 SPID 的地址，找到这个操作系统进程（SPID）就简单了，例如：

```
SQL> select spid，pid from v$process where addr='C000000109C831E0';

SPID             PID
------------ ----------
20552             26

SQL>
```

现在，只需要在操作系统上 kill 操作系统进程 20552 就可以了：

```
ora9i@cs_dc02:/ora9i > ps -ef | grep 20552
   ora9i 20552    1  0  Jan  8  ?       0:01 oraclecsmisc2 (LOCAL=NO)
   ora9i 14742 14740  0 17:19:02 pts/ti    0:00 grep 20552
ora9i@cs_dc02:/ora9i > kill -9 20552
ora9i@cs_dc02:/ora9i > ps -ef | grep 20552
   ora9i 14966 14964  0 17:40:01 pts/ti    0:00 grep 20552
ora9i@cs_dc02:/ora9i >
```

再来检查一下 session 37 的信息，可以看到这个会话是真的被 kill 掉了：

```
ora9i@cs_dc02:/ora9i > exit

SQL> select sid，serial#，status，username，program from v$session where sid=37;
```

```
no rows selected

SQL> l
  1  SELECT s.username，s.status，
  2  x.ADDR，x.KSLLAPSC，x.KSLLAPSN，x.KSLLASPO，x.KSLLID1R，x.KSLLRTYP，
  3  decode(bitand (x.ksuprflg，2)，0，null，1)
  4  FROM x$ksupr x，v$session s
  5  WHERE s.paddr(+)=x.addr
  6  and bitand(ksspaflg，1)!=0
  7* and s.sid=37
SQL> /

no rows selected
```

8. 再次测试

现在，问题解决了，回到刚才 hang 住的会话，它已经恢复了正常操作，并且已经得到了“ ORA-04043: object CSNOZ629926699966 does not exist ”这个正常的信息：

```
SQL> desc CSNOZ629926699966

 ERROR:
ORA-04043: object CSNOZ629926699966 does not exist

SQL>
```

进一步测试一下，再开一个会话，执行 desc CSNOZ629926699966 试试看：

```
ora9i@cs_dc02:/ora9i > sqlplus pubuser/pubuser

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Jan 10 17:42:16 2005

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 and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production

SQL> set timing on
SQL> desc CSNOZ629926699966
ERROR:
ORA-04043: object CSNOZ629926699966 does not exist
```

很明显，当发出命令“ desc CSNOZ629926699966 ”的时候，可以看到系统立刻返回了“ ORA-04043: object CSNOZ629926699966 does not exist ”信息，问题就此解决了。

24.3.2　使用 v$session 和 systemstate 事件解决问题

上一个例子主要借助了 X$KGLLK 基表和 Event systemstate 来解决问题，那么如果不了解

X$KGLLK 基表，或者忘记了如何使用它，那也不要紧张，这里再介绍一种常规的方法。

1. 从 system state 的转储信息中找 waiting session

从 system state 的转储信息中，通过查找字符串 “ waiting for 'library cache lock ”，可以查看到被阻塞进程的信息，具体如下：

```
    PROCESS 28:  ----------------被阻塞的 Oracle 进程，这里 PROCESS 28 对应了 V$PROCESS 中的 PID 的值，
                 也就是说我们可以根据这一信息在 V$PROCESS 和 V$SESSION 找到被阻塞的会话的信息
      ----------------------------------------
      SO: c000000109c83bf0, type: 2, owner: 0000000000000000, flag: INIT/-/-/0x00
      (process) Oracle pid=28, calls cur/top: c00000010b277890/c00000010b277890, flag: (0) -
              int error: 0, call error: 0, sess error: 0, txn error 0
      (post info) last post received: 17 24 6
                 last post received-location: ksusig
                 last post received-location: ksusig
                 last process to post me: c000000109c840f8 25 0
                 last post sent: 0 0 15
                 last post sent-location: ksasnd
                 last process posted by me: c000000109c7ff90 1 6
        (latch info) wait_event=0 bits=0
        Process Group: DEFAULT, pseudo proc: c000000109eefda0
        O/S info: user: ora9i, term: pts/th, ospid: 22580
        ----------------该进程的操作系统进程号，对应于 V$PROCESS 中的 SPID
        OSD pid info: Unix process pid: 22580, image: oracle@cs_dc02 (TNS V1-V3)
        ----------------------------------------
        SO: c000000109f02c68, type: 4, owner: c000000109c83bf0, flag: INIT/-/-/0x00
        (session) trans: 0000000000000000, creator: c000000109c83bf0, flag: (100041) USR/-
BSY/-/-/-/-/-
                 DID: 0002-001C-00000192, short-term DID: 0000-0000-00000000
                 txn branch: 0000000000000000
                 oct: 0, prv: 0, sql: c00000011f8ea068, psql: c00000011f8ea068, user: 50/PUBUSER
        O/S info: user: ora9i, term: , ospid: 22536, machine: cs_dc02
                 program: sqlplus@cs_dc02 (TNS V1-V3)
        application name: SQL*Plus, hash value=3669949024
        waiting for 'library cache lock' blocking sess=0x0 seq=18589 wait_time=0
                  handle   address=c000000122e2a6d8  ,      lock   address=c00000011a449e20  ,
100*mode+namespace=515

    ……

        SO: c00000010b277890, type: 3, owner: c000000109c83bf0, flag: INIT/-/-/0x00
        (call) sess: cur c000000109f02c68, rec 0, usr c000000109f02c68; depth: 0
          ----------------------------------------
          SO: c00000011a449e20, type: 51, owner: c00000010b277890, flag: INIT/-/-/0x00
          LIBRARY OBJECT LOCK: lock=c00000011a449e20 handle=c000000122e2a6d8 request=S
          call pin=0000000000000000 session pin=0000000000000000
          htl=c00000011a449e90[c00000011a4bc350,c00000011a4bc350] htb=c00000011a4bc350
          user=c000000109f02c68 session=c000000109f02c68 count=0 flags=[00] savepoint=463
          the rest of the object was already dumped

    ……
```

在这里已经知道了 PROCESS 28 当前正在等待 Library Cache Lock。“ handle

address=c000000122e2a6d8 ”表示的就是正持有 PROCESS 28 进程所等待的 Library Cache 中的地址。

2. 从 system state 的转储信息中找 blocking session

现在继续在跟踪文件中查找包含“ handle=c000000122e2a6d8 ”字符串的 Oracle PROCESS，也就是查找 blocking session 的信息，发现信息如下：

```
PROCESS 26: --------------阻塞其他会话的 Oracle 进程，这里 PROCESS 26 对应了 V$PROCESS 中的 PID 的值
  ----------------------------------------
  SO: c000000109c831e0, type: 2, owner: 0000000000000000, flag: INIT/-/-/0x00
  (process) Oracle pid=26, calls cur/top: c00000010b2774d0/c00000010b2774d0, flag: (0) -
          int error: 0, call error: 0, sess error: 0, txn error 0
  (post info) last post received: 17 24 6
            last post received-location: ksusig
            last post received-location: ksusig
            last process to post me: c000000109c840f8 25 0
            last post sent: 751404 0 15
            last post sent-location: ksasnd
            last process posted by me: c000000109c836e8 1 6
    (latch info) wait_event=0 bits=0
    Process Group: DEFAULT, pseudo proc: c000000109eefda0
    O/S info: user: ora9i, term: UNKNOWN, ospid: 20552
    OSD pid info: Unix process pid: 20552, image: oracle@cs_dc02 (TNS V1-V3)
    ----------------------------------------
    SO: c0000001180b9510, type: 8, owner: c000000109c831e0, flag: INIT/-/-/0x00
    (FOB) flags=2 fib ptr=162e1b48 incno=0 pending i/o cnt=0
    ----------------------------------------
    SO: c0000001180b9458, type: 8, owner: c000000109c831e0, flag: INIT/-/-/0x00
    (FOB) flags=2 fib ptr=162deb18 incno=0 pending i/o cnt=0
    ----------------------------------------
    SO: c0000001180b8230, type: 8, owner: c000000109c831e0, flag: INIT/-/-/0x00
    (FOB) flags=2 fib ptr=162de848 incno=0 pending i/o cnt=0
    ----------------------------------------
    SO: c0000001180b7b00, type: 8, owner: c000000109c831e0, flag: INIT/-/-/0x00
    (FOB) flags=2 fib ptr=162de578 incno=0 pending i/o cnt=0
```

注意下面的信息：

```
     ----------------------------------------
    SO: c000000108c99e28, type: 4, owner: c000000109c831e0, flag: INIT/-/-/0x00
   (session) trans: c0000001169403c0, creator: c000000109c831e0, flag: (100041) USR/-
BSY/-/-/-/-/-
            DID: 0002-001A-0000007D, short-term DID: 0000-0000-00000000
            txn branch: c00000011b825e18
            oct: 0, prv: 0, sql: 800003fb0005f7b0, psql: c00000011fbe3f98, user: 50/PUBUSER
    O/S info: user: report16, term: , ospid: 20550, machine: cs_dc02
            program: sqlplus@cs_dc02 (TNS V1-V3)
    application name: SQL*Plus, hash value=3669949024
    waiting for 'SQL*Net message from dblink' blocking sess=0x0 seq=3319 wait_time=0
              driver id=28444553, #bytes=1, =0
              -------------------
```

SO: c000000108c99e28：c000000108c99e28 对应的就是 v$session 中的 SADDR 的值，通过这

个信息就可以找到 blocking session 的 SID 等信息。

而下面的就是进程信息：

```
    O/S info: user: report16, term: , ospid: 20550, machine: cs_dc02
            program: sqlplus@cs_dc02 (TNS V1-V3)
    application name: SQL*Plus, hash value=3669949024
```

再看这个 session 的等待信息：

```
    waiting for 'SQL*Net message from dblink' blocking sess=0x0 seq=3319 wait_time=0
```

其中，#bytes 表示一个 Server Process 通过 database link 发送给另一个 Server Process 的字节数（bytes）。driver id 是一个十进制数，需要把它转化为十六进制数，然后就会发现它对应于通过 event 10046 中的相应信息：

```
*** 2005-01-10 11:44:48.200
WAIT #1: nam='SQL*Net message from dblink' ela= 104397696 p1=675562835 p2=1 p3=0
WAIT #1: nam='SQL*Net message to dblink' ela= 4 p1=675562835 p2=1 p3=0
```

```
SQL> select to_char(675562835,'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX') from dual;

TO_CHAR(675562835,'XXXXXXXXXXXXXX
--------------------------------
                        28444553
```

继续在 dump 文件中找线索，很快，发现了下面的信息：

```
    temporary object counter: 0
      ----------------------------------------
      SO: c00000011a4496b0, type: 51, owner: c000000108c99e28, flag: INIT/-/-/0x00
      LIBRARY OBJECT LOCK: lock=c00000011a4496b0 handle=c00000012029f968 mode=N
      call pin=0000000000000000 session pin=c00000011a44ad70
      htl=c00000011a449720[c00000011a4baa78,c00000011a4baa78] htb=c00000011a4baa78
      user=c000000108c99e28 session=c000000108c99e28 count=1 flags=[00] savepoint=173
      LIBRARY OBJECT HANDLE: handle=c00000012029f968
      namespace=CRSR flags=RON/KGHP/PN0/[10010000]

……
        ----------------------------------------
        SO: c00000011a44a150, type: 51, owner: c0000001169403c0, flag: INIT/-/-/0x00
        LIBRARY OBJECT LOCK: lock=c00000011a44a150 handle=c000000122e2a6d8 mode=X
        call pin=0000000000000000 session pin=0000000000000000
        htl=c00000011a44a1c0[c00000011a4bb328,c00000011a4bb328] htb=c00000011a4bb328
        user=c000000108c99e28 session=c000000108c99e28 count=1 flags=[00] savepoint=179
       LIBRARY OBJECT HANDLE: handle=c000000122e2a6d8
        name=PUBUSER.CSNOZ629926699966
        hash=eddf82b5 timestamp=01-08-2005 13:00:18 previous=NULL
        namespace=TABL/PRCD/TYPE flags=KGHP/TIM/PTM/SML/[02000000]
        kkkk-dddd-llll=0000-0709-0001 lock=X pin=X latch#=3
        lwt=c000000122e2a708[c00000011a449e40        ,        c00000011a449e40]
ltm=c000000122e2a718[c000000122e2a718,c000000122e2a718]
        pwt=c000000122e2a738[c000000122e2a738        ,        c000000122e2a738]
ptm=c000000122e2a7c8[c000000122e2a7c8,c000000122e2a7c8]
        ref=c000000122e2a6e8[c000000122e2a6e8,
    c000000122e2a6e8] lnd=c000000122e2a7e0[c000000122e2a7e0,c000000122e2a7e0]
          LOCK INSTANCE LOCK: id=LBcafc8485d0949f81
```

```
              PIN INSTANCE LOCK: id=NBcafc8485d0949f81 mode=X release=F flags=[00]
              LIBRARY OBJECT: object=c000000122e12f70
              type=TABL flags=EXS/LOC/CRT[0015] pflags= [00] status=VALD load=0
              DATA BLOCKS:
              data#    heap  pointer status pins change
              ----- -------- -------- ------ ---- ------
                 0 c000000122e2a618 c000000122e13118 I/P/A    0 INSERT
                 3 c000000122e13178       0 -/P/-    1 NONE
                 8 c000000122e12c30 c000000122febdb8 I/P/A    1 UPDATE
                 9 c000000122e13090       0 -/P/-    1 NONE
                10 c000000122e12ce0 c000000122acbc70 I/P/A    1 UPDATE
          ----------------------------------------
    ……
```

下面来解释一下其中的重要信息：

- SO: c00000011a44a150， type: 51， owner: c0000001169403c0， flag: INIT/-/-/0x00

X$KGLLK.KGLLKADR 对应于 SO（SO: c00000011a44a150），X$KGLLK.KGLLKUSE 和 x$kgllk.KGLLKSES 对应于 owner 的值（owner: c0000001169403c0）。

- LIBRARY OBJECT LOCK: lock=c00000011a44a150 handle=c000000122e2a6d8

X$KGLLK.KGLLKADR 对应于 SO 和 lock 的值（SO: c00000011a44a150，lock=c00000011a44a150），X$KGLLK.KGLLKHDL 对应于 handle 的值（handle=c000000122e2a6d8）。

- call pin=0000000000000000 session pin=0000000000000000

X$KGLLK.KGLLKPNS 对应于 session pin 的值（session pin=0000000000000000）。

- user=c000000108c99e28 session=c000000108c99e28 count=1 flags=[00] savepoint=179

user 和 session 的值分别对应着 X$KGLLK.KGLLKUSE 和 X$KGLLK.KGLLKSES，也对应于 v$session 中阻塞其他会话的 SADDR。X$KGLLK.KGLLKSPN 对应于 SAVEPOINT 的值（savepoint=179）。

已经了解阻塞进程和等待进程的信息，根据上述两个 Oracle 进程号（Oracle PID），就可以找到它们的会话信息和操作系统进程信息：

```
SQL> select spid，pid，addr from v$process where pid in (26，28);

SPID             PID ADDR
------------ ---------- ----------------
20552             26 C000000109C831E0
22580             28 C000000109C83BF0
```

这里，看到“C000000109C831E0”表示阻塞其他会话的 Oracle 进程的进程地址，“C000000109C83BF0”表示被阻塞的 Oracle 进程的进程地址。

3. 使用 v$session 发现会话信息

下面使用 v$session 视图来进一步证实上述信息：

```
SQL>col username for a20
SQL> col osuser for a20
SQL> col machine for a20
SQL> l
  1  select sid，serial#，username，osuser，machine，to_char(logon_time，'yyyy/mm/dd hh24:mi:ss')
LogonTime
  2* from v$session where paddr in ( select addr from v$process where spid ='&spid')
SQL> /
```

```
Enter value for spid: 20552 ----------- 阻塞其他会话的 Oracle 进程

old  2: from v$session where paddr in ( select addr from v$process where spid ='&spid')
new  2: from v$session where paddr in ( select addr from v$process where spid ='20552')

     SID   SERIAL# USERNAME          OSUSER           MACHINE          LOGONTIME
-------- ------- ------------- ----------------- ---------------- -------------------
      37     2707 PUBUSER          report16         cs_dc02          2005/01/08 13:00:17

SQL> /
Enter value for spid: 22580 ----------- 被阻塞的 Oracle 进程

old  2: from v$session where paddr in ( select addr from v$process where spid ='&spid')
new  2: from v$session where paddr in ( select addr from v$process where spid ='22580')

     SID   SERIAL# USERNAME          OSUSER           MACHINE          LOGONTIME
-------- ------- --------------- ----------------- ----------------- -------------------
      30    24167 PUBUSER          ora9i            cs_dc02          2005/01/10 10:20:31


SQL> select sid，saddr，paddr，username，status，OSUSER from v$session where sid in (37，30);

     SID SADDR         PADDR         USERNAME          STATUS   OSUSER
---------- ---------------- ---------------- ------------------- -------- --------------------
      30 C000000109F02C68 C000000109C83BF0 PUBUSER          ACTIVE   ora9i
      37 C000000108C99E28 C000000109C831E0 PUBUSER          ACTIVE   report16
```

这里，已经再次证实了刚刚在 dump 文件中看到的信息，很明显，report16 用户（操作系统用户）的会话（session）37 阻塞了 ora9i 用户（操作系统用户）的会话（session）30。

4. 解决问题

现在，问题已经水落石出了，解决方法和方法 1 中的一样（在操作系统中直接 kill 掉相应的操作系统进程）：

```
ora9i@cs_dc02:/ora9i > ps -ef | grep 20552
  ora9i 20552    1  0  Jan  8  ?       0:01 oraclecsmisc2 (LOCAL=NO)
  ora9i 14742 14740  0 17:19:02 pts/ti   0:00 grep 20552
ora9i@cs_dc02:/ora9i > kill -9 20552
ora9i@cs_dc02:/ora9i > ps -ef | grep 20552
  ora9i 14966 14964  0 17:40:01 pts/ti   0:00 grep 20552
ora9i@cs_dc02:/ora9i >
```

5. 重温一下当时锁的信息

出于研究的目的，这里使用 X$KGLLK 来进一步了解上述两个会话（SID 30 和 SID 37）所有已经持有锁的相关信息：

```
SQL> set linesize 150
SQL> set pages 10000
SQL> select * from v$lock where sid in (37，30);

ADDR           KADDR          SID TY    ID1    ID2    LMODE    REQUEST     CTIME    BLOCK
```

```
---------- ---------- --------- -- ------- ----- ------- ---------- ---------- ----------
C0000001169403C0 C000000116940538     37 TX    917507    26579      6      0  180478     2
C00000011676DAE0 C00000011676DB08     37 TM        18        0      3      0  180478     2
C00000010B30C4E8 C00000010B30C508     37 XR         4        0      2      0  180369     2
C00000010B30C460 C00000010B30C480     37 DX        21        0      1      0      68     0

SQL>
```

不难看出，会话 37 阻塞了其他会话。现在，再进一步看看会话 37 当前在哪些对象上加了锁：

```
SQL> select object_name，object_id from dba_objects where object_id in ('917507'，'18'，'4'，
'21') order by object_id;

OBJECT_NAME                    OBJECT_ID
------------------------------ ----------
TAB$                                4
OBJ$                               18
COL$                               21

SQL> /

OBJECT_NAME                    OBJECT_ID
------------------------------ ----------
TAB$                                4
OBJ$                               18
COL$                               21
```

接下来，再着重看看 SID 为 37 的会话在 Library Cache 中请求和持有对象锁的详细信息：

```
SQL> col KGLNAOBJ for a30
SQL> col USER_NAME for a10
SQL> l
  1  select INST_ID，USER_NAME，KGLNAOBJ，KGLLKSNM，KGLLKUSE，KGLLKSES，KGLLKMOD，KGLLKREQ
  2* from x$kgllk where KGLLKSNM  = 37
SQL> /

   INST_ID USER_NAME  KGLNAOBJ    KGLLKSNM KGLLKUSE    KGLLKSES     KGLLKMOD   KGLLKREQ
-------- ------- ------------- -------- ------------ ------------ --------- ----------
       2 PUBUSER    DBMS_OUTPUT                  37 C000000108C99E28 C000000108C99E28  1  0
       2 PUBUSER    DBMS_OUTPUT                  37 C000000108C99E28 C000000108C99E28  1  0
       2 PUBUSER    DBMS_STANDARD                37 C000000108C99E28 C000000108C99E28  1  0
       2 PUBUSER    PUBUSER                      37 C000000108C99E28 C000000108C99E28  1  0
       2 PUBUSER    SELECT MINOR_VERSION  FROM SY  37 C000000108C99E28 C000000108C99E28 1  0
                    S.CDC_SYSTEM$

       2 PUBUSER   SELECT MINOR_VERSION   FROM SY  37 C000000108C99E28 C000000108C99E28 1  0
                    S.CDC_SYSTEM$

       2 PUBUSER    DBMS_CDC_PUBLISH             37 C000000108C99E28 C000000108C99E28  1   0
       2 PUBUSER    DBMS_CDC_PUBLISH             37 C000000108C99E28 C000000108C99E28  1   0
       2 PUBUSER    CSNOZ629926699966           37 C000000108C99E28 C000000108C99E28   3   0
       2 PUBUSER    DBMS_APPLICATION_INFO        37 C000000108C99E28 C000000108C99E28  1   0
       2 PUBUSER    DBMS_APPLICATION_INFO        37 C000000108C99E28 C000000108C99E28  1   0
       2 PUBUSER    DATABASE                    37 C000000108C99E28 C000000108C99E28   1   0
```

```
12 rows selected.
```

再看看 SID 为 30 的会话在 Library Cache 中请求和持有对象锁的详细信息：

```
SQL> select INST_ID, USER_NAME, KGLNAOBJ, KGLLKSNM, KGLLKUSE, KGLLKSES, KGLLKMOD, KGLLKREQ
  2  from x$kgllk where KGLLKSNM  = 30
  3  /

   INST_ID USER_NAME  KGLNAOBJ     KGLLKSNM KGLLKUSE  KGLLKSES         KGLLKMOD   KGLLKREQ
-------- --------- ----------- ---------- --------- --------------- ---------- ----------
       2 PUBUSER   PUBUSER                    30 C000000109F02C68 C000000109F02C68     1   0
       2 PUBUSER   CSNOZ629926699966          30 C000000109F02C68 C000000109F02C68     0   2
       2 PUBUSER   DBMS_APPLICATION_INFO      30 C000000109F02C68 C000000109F02C68     1   0
       2 PUBUSER   DBMS_APPLICATION_INFO      30 C000000109F02C68 C000000109F02C68     1   0
       2 PUBUSER   DATABASE                   30 C000000109F02C68 C000000109F02C68     1   0
```

其中：

(1) KGLNAOBJ 列包含了在 Library Cache 中的对象上执行命令的语句的前 80 个字符，其实从这里也就可以大大缩小范围了。

(2) KGLLKSES 对应于 v$session 中的 SADDR 列的值。

(3) KGLLKSNM 对应于 v$session 中的 SID（Session ID)。

(4) KGLLKHDL 的值与方法 1 中跟踪文件中的 handle address 的值对应。

(5) KGLLKPNS 的值对应于方法 1 中跟踪文件中的 session pin 的值。

24.4 小结

至此，已经了解了 Library Cache Lock 的产生原因及防范方法，并知道了如果出现这个问题应该如何解决，文中给出了详细的分析步骤和解决方法（本章中给出的 SQL 语句请先测试后再使用)。

参考信息

MetaLink：Note:122793.1 HOW TO FIND THE SESSION HOLDING A LIBRARY CACHE LOCK

作者简介

张春宏（第一作者)，任职于某国家安全部门，若干年来一直负责公安警卫信息化建设；公安金盾工程资深专家；在部级大型数据库设计以及应用方面有雄厚的经验，长于数据库诊断、性能调整与 SQL 优化等。对于 Oracle 内部技术具有一定的研究。

张大鹏（第二作者)，网名 Lunar2000（Lunar)，ITPUB 资深会员。现任职于某大型外资企业，服务于电信增值业务，从事专职 DBA 工作。主要负责 Oracle 数据库日常管理，包括备份和恢复，性能优化，故障诊断等。实践经验丰富，长于数据库故障诊断、性能优化和备份恢复。

第五篇
SQL 优化及其他

本篇共分 8 章，主要内容如下：

25

第 25 章　Oracle 数据库优化之索引（Index）简介

如果要大幅度地提升数据库的工作能力，必须对服务器的硬件设施进行升级。据 IT 行业的统计显示，数据库能力的提升，70%是取决于服务器硬件的优劣。在剩下可以人为调优的 30%里面，才是数据库管理人员一展身手的空间。

硬件与管理人员对系统优化的提升来讲，很多人认为管理人员的技能对系统更为重要，占系统优化比重的 90%之多。关于这方面的讨论，可以打一个比方：如果硬件是一座房子的话，应用系统就好比是家具与装修。假若房子已经给定的话，管理人员的工作只是怎样调整摆放家具的位置，让这座房子看起来更顺眼更舒服。硬件设施，对于应用软件来讲，如果设施过分好，是浪费了资源，但总好过房子太小，家具太多，难以摆放的窘境。很多的时候，数据管理员和系统管理员都没有办法对硬件的配置有决定性的权利。他们所可以做的工作，就是在可能优化的空间内实现最大限度的优化。

业界对人为可以提升的这部分空间的划分也进行了调查，结果如图 25-1 所示。

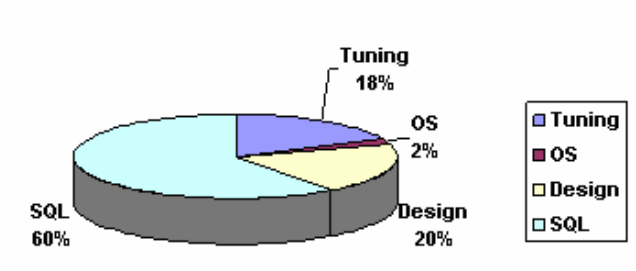

图 25-1　优化空间的划分

从图 25-1 中可以看到，可以人为调整的部分中，60%的问题是发生在 SQL 部分的；20%的问题发生在数据库的设计部分。这两个部分都是由开发人员来完成的。真正作为数据库管理人员的 DBA，对数据库的调优调整，最大只能做到人为可调整的份额中的 18%，占整个系统调整的 5.4%而已。系统管理员所占的份额更少，只是人为系统调整的 2%，占整个系统的 0.6%。

所以业界的戏言，系统所需的最佳状态，是这样来完成的：

```
系统的最佳状态 = 金钱 + 优秀开发人员 + 优秀的 DBA
```

对于技术人员无法控制的项目预算来说，对服务器的硬件选择，只能提供参考的意见，因此，系统的状态，只能这样完成：

```
系统状态 =  优秀开发人员 + 优秀的 DBA
```

而对于目前很多运行状态不佳的数据库的状态，只能形容成：

```
系统状态 = 开发人员提交项目时的状态
```

造成开发人员提交程序中存在 SQL 以及设计方面问题的原因很多。例如，一些开发人员没有受到正规计算机专业的训练，不了解计算机算法的处理过程；或者有一些开发人员，只是写标准的程序，适用于所有数据库的 SQL，而不是针对 Oracle 数据库的特点开发出能应用在 Oracle 数据库上的最优 SQL 语句。另外，对于开发部门提交的程序，即使测试部门做了足够的调试，还是存在很多的问题。这些程序没有在真正的生产环境上进行试运行，很多问题没有暴露出来。即使是在真正的生产环境下进行过 30 天试运行，才正式进入生产模式的数据库也会存在类似的问题，例如，对试运行中极少用到的程序，或随着数据量不断增加才显现出来的性能问题，都是 DBA 日常需要面对的问题。对于 SQL 的程序问题，还可以通过对 SQL 的调优来解决，而存在于数据库设计上面的问题，则很难解决，通常是动一发而牵全身。

因此对一个 DBA 来讲，日常的工作，除了要做维护数据库的调优工作（那 18%的部分），很大的一部分工作，是要帮助开发人员调整正运行在应用程序中的 SQL 程序。

很多的时候，不仅本公司开发部门人员提交的 SQL 程序存在着问题，应用程序提供商提交的 SQL 程序也存在一些问题。本来以为像 PeopleSoft 这样大的厂商提交的程序，应该是经过完全测试的，可以完全信赖。但最近一次碰到的问题，让我忙碌了两个星期之久的，居然就是 PeopleSoft 提交的年度报表的一个包，里面居然有 3 个 dead loops。每次运行起来，三四个小时之后，rollback segment 肯定会爆掉，因为产生的 rollback segment 的大小，已经超出相关表中数据总和的数倍了。最后不得不把这个包切成一段一段运行，让本来几分钟就能完成的工作，从开始到结束运行了 2 天之久。

做了几年的 DBA 下来，感觉到，需要复杂技巧及高深的知识来解决所遇到的问题的情形并不多，相对来说，在对基本概念有清晰认识的情形下，解决问题的切入口通常都会比较准确。对于写给别人看的技术文章，我一向倾向于介绍一点比较基础的本质的东西，希望能帮助到更多的人。本章所涉及的有关 Oracle 索引（index）方面的介绍，就是很基础很本质的内容。

25.1 索引的作用

索引（Index），顾名思义，是以最快的路径找到所需要的数据。在 SQL 调优上，索引的使用，是在进行查询操作时避免对所查询的表进行 full table scan 的简便方法。在对数据进行 update 或者 delete 操作时，使用索引也同样可以提高性能。

索引的另一个功用，是对数据完整性的强制控制。众所皆知，在两种情形下，系统会自动为表的列建立索引：

(1) 当表的列被指定为 primary key 时。

(2) 当表的列有 unique constraint 时。

当一个表中有 foreign key 存在时，通常也建议这个列使用索引。当对表进行插入数据的操作时，对该插入的数据是否与现有数据重复的核对工作，即是对 PK 和 UK 的列的索引核对。当 FK

所在列存在索引时，如果对其母表插入或者更新数据，锁定的不是该子表，而是该子表 FK 列上面的索引。但是 Oracle 从 9i 版本开始逐步改变了一些运行的方式。

25.2　索引管理的常见问题

下面列出一些索引管理方面的常见问题，供大家参考。

1. 是否最近做过数据库的分析

对于使用 Oracle Cost Based Optimizer（CBO）的应用程序，对于日常使用的数据库，是建议每天都对整个数据库做分析的，当然这是基于数据库不是很大，一般的分析可以在 30～60 分钟左右完成。如果是很大的数据库，每天对只有数据变动很大的表进行分析即可，对于静态不变动的表，并不需要每天都进行分析。

Oracle version 7.x 的版本中，Oracle 提供了两种表分析的方法：

- ANALYZE
- DBMS_UTILITY

Oracle 从其版本 8.1.5 开始，建议使用的是 DBMS_STATS 包。值得注意的是，很多 DBA 反映说 DBMS_STATS 包有很多 Bugs。我的环境目前仍然使用 ANALYZE 和 DBMS_UTILITY。

如果每天定时运行了数据库的分析工作，当出现性能问题时，是否就可以不用考虑有关表的统计数据是否有问题？

其实还是要核对一下表和索引最近一次做过分析的时间。如果当天恰好某个表做过 drop/import，那个表就没有分析过的数据存在于系统表中。如果有恰好这个表是某个大的查询所需的相关表，这种情形下多半就会很糟糕。如果不确定是否有人动过数据库，核对一下做过表分析的时间，多半有利无害的。

2. 索引的数量

许多程序员认为，所有在 where 语句中的列，都应该建立索引。这种想法是不对的。过多建立索引，与建立索引太少都会极大地影响到数据库的性能。

为什么过多的建立索引会影响性能呢？

当对表中的数据进行 insert/update/delete 的操作，索引中的数据也会随之更新。假如不是一个只读表的话，索引对查询，以及更新数据的提高性能，或者降低性能，就需要一个平衡点。而这个平衡点是很难去把握的。记得多年之前读过的资料，Oracle 是建议，索引的列不要超过表的列数的 40%。由此话题，引申出的另一个问题就是，什么时候需要建索引，建立什么样的索引对提高数据库的性能最为有利，这就是一个相当复杂的问题了。

假使真的整个表都需要索引的话，不妨考虑一下能否 IOT（Indexed Organized Table）。IOT 适用于 lookup 数据，通过 PK 进行搜索，以强制数据按照 PK 排序来提供更好的 clustering 因子，缩小 range scan 方面的花费。因此，IOT 也有很多限制，比方表中数据最好是静止的，建议表的列不要太多等。

3. 是否使用了正确的索引种类

索引有很多种，不是所有的数据库都支持所有种类的索引。常用的有以下几种。

- B-Tree：Oracle 数据库中用得最普遍的索引还是 B-Tree。关于 B-Tree 的描述出现在几乎所有关于数据结构与算法的书籍中。
- FBI（Function Based Index）：当 where 语句中使用 function 时，B-tree 索引不能使用，变更的方法就是使用 FBI 索引。对于经常更新的表来说，更新 FBI，相对来说也是一件很昂贵的事情。同时，Rule Based Optimizer 不支持 FBI 索引。
- Bitmap：Bitmap 索引不是所有的数据库都支持，Oracle 数据库也只是在 CBO 的模式下才支持 Bitmap 索引。Bitmap 索引适用于静态读取的表，并且索引所在的列是低基数。数据仓库的环境，是 Bitmap 索引发挥功用的最佳环境。

4. 索引不被使用的几种情况

很多时候，表的正确的列建立了正确的索引，但是某些 queries 使用的列上面的索引，并没有被使用，查询的速度是难以接受的慢。Oracle 解释说，有很多种情形导致 Oracle 的优化器 Optimizer 无法使用索引。这个时候，就需要修改 SQL 来强制 SQL 语句使用索引。现在仅仅知道下面的几种情况下，索引是不被使用的，所影响到的表必须进行全表检索（full table scan）。

这里使用表 scott.emp 来做实验，除了原来 empno 上面的 primary key 之外，在 comm、mgr、job 三个列上面也建立了索引：

```
SQL> select table_name, index_name, column_name from user_ind_columns where table_name='EMP';

TABLE_NAME     INDEX_NAME     COLUMN_NAME
-------------- -------------- ---------------
EMP            PK_EMP         EMPNO
EMP            COMM_TST       COMM
EMP            MGR_TST        MGR
EMP            JOB_TST        JOB

SQL> DESC EMP
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
 DEPTNO                                             NUMBER(2)

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> set autotrace on;
```

(1) 当对同一个表中的两个列 (empno 和 mgr) 进行比较的情形下，索引 (pk_emp 和 mgr_tst) 有时不会被使用：

```
SQL> select * from emp where empno<mgr;

     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

7 rows selected.

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=8 Bytes=296
         )

   1    0   TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=8 Bytes=
         296)

Statistics
----------------------------------------------------------
         1  recursive calls
         0  db block gets
         8  consistent gets
         0  physical reads
         0  redo size
      1043  bytes sent via SQL*Net to client
       500  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
```

```
         0  sorts (memory)
         0  sorts (disk)
         7  rows processed
```

(2) Null 值。一般情形下，索引中并不存在 Null 值。如果 where 语句中出现 is null 或者 is not null 时，索引就不能被使用（comm._tst 没有被使用）。

```
SQL> select * from emp where comm is not null;

    EMPNO ENAME     JOB            MGR HIREDATE        SAL      COMM    DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
     7499 ALLEN     SALESMAN       7698 20-FEB-81      1600        300        30
     7521 WARD      SALESMAN       7698 22-FEB-81      1250        500        30
     7654 MARTIN    SALESMAN       7698 28-SEP-81      1250       1400        30
     7844 TURNER    SALESMAN       7698 08-SEP-81      1500          0        30


Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=4 Bytes=148
         )

   1    0   TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=4 Bytes=
         148)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
         8  consistent gets
         0  physical reads
         0  redo size
       902  bytes sent via SQL*Net to client
       500  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         4  rows processed
```

(3) 当 where 语句中存在有 not function 时，比如 not in、not exist、column <> value、column1 > value 或 column2 < value 等情形下，索引不能被使用。

```
SQL> select * from emp where comm <> 1000;

    EMPNO ENAME     JOB            MGR HIREDATE        SAL      COMM    DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
     7499 ALLEN     SALESMAN       7698 20-FEB-81      1600        300        30
     7521 WARD      SALESMAN       7698 22-FEB-81      1250        500        30
     7654 MARTIN    SALESMAN       7698 28-SEP-81      1250       1400        30
     7844 TURNER    SALESMAN       7698 08-SEP-81      1500          0        30


Execution Plan
```

```
-------------------------------------------------------
  0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=3 Bytes=111
        )

  1   0   TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=3 Bytes=
        111)

Statistics
-------------------------------------------------------
        1  recursive calls
        0  db block gets
        8  consistent gets
        6  physical reads
        0  redo size
      902  bytes sent via SQL*Net to client
      500  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        0  sorts (memory)
        0  sorts (disk)
        4  rows processed
```

（4）当使用了 single-row function 时，如 nvl、to_char、lower 等，索引不能被使用。

```
SQL> select ename, nvl(comm, 0) from emp;

ENAME     NVL(COMM,0)
---------- -----------
SMITH              0
ALLEN            300
WARD             500
JONES              0
MARTIN          1400
BLAKE              0
CLARK              0
SCOTT              0
KING               0
TURNER             0
ADAMS              0
JAMES              0
FORD               0
MILLER             0

14 rows selected.


Execution Plan
-------------------------------------------------------
  0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=14 Bytes=11
        2)

  1   0   TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=14 Bytes
        =112)
```

```
Statistics
----------------------------------------------------------
        1  recursive calls
        0  db block gets
        8  consistent gets
        0  physical reads
        0  redo size
      623  bytes sent via SQL*Net to client
      500  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        0  sorts (memory)
        0  sorts (disk)
       14  rows processed
```

（5）当使用通配符号%或者_作为查询字符串的第一个字符时，例如，在语句“ where name like ‘%xxxx’ ”的情形下，索引也无法使用（对于这种情况，现在很多数据库都支持所谓的“ 全文检索索引 ”，可以很好地解决这个问题）。但是如果查询字符串的第一个字确定，例如“ where name like ‘a%’ ”这样，则可以使用索引。

```
SQL> select ename, job from emp where job like '%C%';

ENAME     JOB
---------- ---------
SMITH     CLERK
ADAMS     CLERK
JAMES     CLERK
MILLER    CLERK


Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1 Bytes=14)
   1    0   TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=3 Card=1 Bytes=
         14)

Statistics
----------------------------------------------------------
        1  recursive calls
        0  db block gets
        8  consistent gets
        0  physical reads
        0  redo size
      497  bytes sent via SQL*Net to client
      500  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        0  sorts (memory)
        0  sorts (disk)
        4  rows processed

SQL> select ename, job from emp where job like 'C%';

ENAME     JOB
```

```
---------- ---------
SMITH      CLERK
ADAMS      CLERK
JAMES      CLERK
MILLER     CLERK


Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=3 Bytes=42)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=2 Car
         d=3 Bytes=42)

   2    1    INDEX (RANGE SCAN) OF 'JOB_TST' (INDEX) (Cost=1 Card=3)

Statistics
----------------------------------------------------------
         1  recursive calls
         0  db block gets
         4  consistent gets
         1  physical reads
         0  redo size
       497  bytes sent via SQL*Net to client
       500  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         4  rows processed
```

5. 索引重建

所谓的索引重建，一般都是指对 B-Tree 索引进行重建。对于更新频繁的表，数据的插入时，依附在相应 PCTFREE List 上面的索引块（block）的后面，如果这个块所含的数据达到了 PTCFREE 规定，就导致索引数据所在的块分裂成两个，到了一定的程度，可能 B-Tree 索引会加深一层（split、move down to next level）。当表 delete 掉了很多数据时，这些数据在索引中所占的空间，却不能被释放出来重新使用。

对于什么样的情形，必须对表进行重建，很早以前，Oracle 曾经建议过，在以下两种情形下，索引需要重建：

- 当索引的层数超过 4 层时。
- 索引中被删除的数据超过了该索引数据总和的 20%（因为这个数据是从索引分析之后的数据演算而来，在很多情形下，对表和索引分析的结果，强制了 CBO 使用这些结果进行 SQL 的优化计算）。

```
SQL> analyze index abcd validate structure;
SQL> select (del_lf_rows_len/lf_rows_len)*100 as index_usage
       From index_stats
       Where index_name = 'ABCD';
```

在最近的资料中，Oracle 不再给出重建索引的标准了，似乎都随 DBA 的意愿来决定是否重建索引。

对所管理的数据库，我采用一种最简单的方法判断索引是否需要重建。我所管理的数据库，索引都集中在 2 个到 3 个表空间（tablespace）。我使用 EM 中的表空间管理工具 Tablespace Map 对表空间进行分析，如果索引出现了红旗的标志，我就重建索引。到目前为止这个方法还是很省时省力。

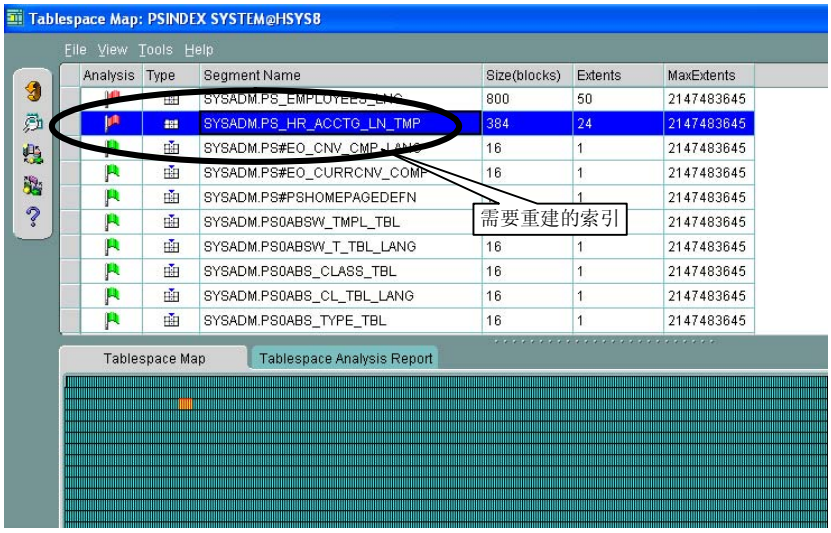

图 25-2　需要重建的索引

25.3　索引的管理

接下来，还有几个在实际操作中需要注意的问题。

1. 怎样发现表的索引太少

这个问题好像很简单。如果 explain plan 上面显示了 full table scan，而且相关的列没有索引，而且又需要建立索引，那么就请建立一个。

如果是不常用的查询，或者一次性的查询，建立一个索引如果需要半小时，却可以把该查询运行的时间缩短 2 个小时，或者 3 个小时，或者更多，可以考虑先建立一个临时的索引，运行完之后再删除。

2. 怎样发现表的索引太多

这个问题就麻烦得多了，通常要对表逐个检查才可能发现问题。对于一个只有三四个表的数据库，或许逐个表的查一遍不难，对于一个有两三百个表的数据库，仔细地过滤一次也可以做到。对于一个 ERP 环境的数据库来说则几乎是不可能的任务了。新版的 PeopleSoft Finanical 数据库，表的数量高达 7000 多个，索引也达到了 5000 的数量，索引太多是普遍存在的问题。但是对于这种厂商提供的套装软体，DBA 几乎是不能做任何改动的。

因此发现表的索引太多而引起的性能问题，通常只是对自己公司开发的应用程序，或者当用户又无法忍耐的性能问题的时候，才有可能去寻找问题的根本，而解决的方案，也是要得到开发

人员的认同之后，共同确定改进的办法。通常来说，DBA 是无法了解许多应用程序的细节的，所以绝对不能轻易地删除任何他人建立的 Objects，即使 DBA 有多么确定这个 Object 是没有用处的。

发现表的索引太多的方法，就是把一个表的所有索引都列出来。发现是否有重复建立的索引，是否有同功能的索引，然后把其删除。

Oracle 数据库自 9i 的版本开始，增加了监测索引是否被使用了的功能，让数据库管理员能很容易地发现从来没有使用过，或者很少使用过的索引。这个功能是下面的 SQL 命令：

```
Alter index <index_name> monitoring usage;
Alter index <index_name> nomonitoring usage;
```

注意 Index Monitoring 的 Bug，不能 Monitor Sys 的索引，Oracle 在版本 9.2.0.5 之后宣布修正了这个问题。另外，显示没有用到的索引，如果是 Unique Index，可能也不能 Drop，有些是为了 Unique Constraint 需要的。

Oracle 建议当需要监测索引的使用情况时才使用它。虽然监测不会引起什么大的性能问题，但是每次使用索引监测的时候，explain plan 中索引的相关部分都是要重复运行一次的，而不能重复使用已经生成的 explain plan。

3. 重建索引

自从 Oracle 支持 rebuild online 的功能之后，一般小一点的索引，在任何时间进行 rebuild online 已经没有什么问题了。其主要方面还是在如何判断索引是否需要重建（前面已经讨论过了）。

对于比较大的索引，还是建议在非繁忙时刻进行。rebuild 比 rebuild online 的运行时间稍微要快一点。

4. EZSQL by John H. Dorlon：一个小巧的 Oracle 数据库管理工具

（感谢 itpub.net 的超级版主 easyfree 游立新先生提供 EZSQL 工具）

EZSQL 这个小工具，是 Quest 公司著名的管理工具 TOAD 的前身。在 Quest 公司向 EZSQL 的发明人 John Dorlon 买断 EZSQL，并邀请他进入了 Quest 公司继续开发 TOAD 之前，EZSQL 都是免费下载使用的。EZSQL 是一个很小的工具，一共只有 1.57 MB 的大小，却是麻雀虽小，五脏俱全。其中监测索引使用状态的 GUI 界面更是简单容易、使用方便，如图 25-3 所示。

在写这篇文章的时候，我对网络进行了一下搜索，发现在 oradb.net 上面有两个版本可以下载，地址如下。

- http://www.oradb.net/book/Ezsql2.3.15.6677.rar
- http://www.oradb.net/book/19818ezsqlexe.zip

5. DB Expert for Oracle by LECCO

DB Expert for Oracle 在调整、监测索引方面也有很强的功能。但是这个工具是需要付费的。另外，LECCO 公司已经被 Quest 公司收购合并，不知道这个软件目前的升级情况，以及购买的费用。

DB Export for Oracle 的索引监测功能如图 25-4 所示。

DB Export for Oracle 的索引调优功能如图 25-5 所示。

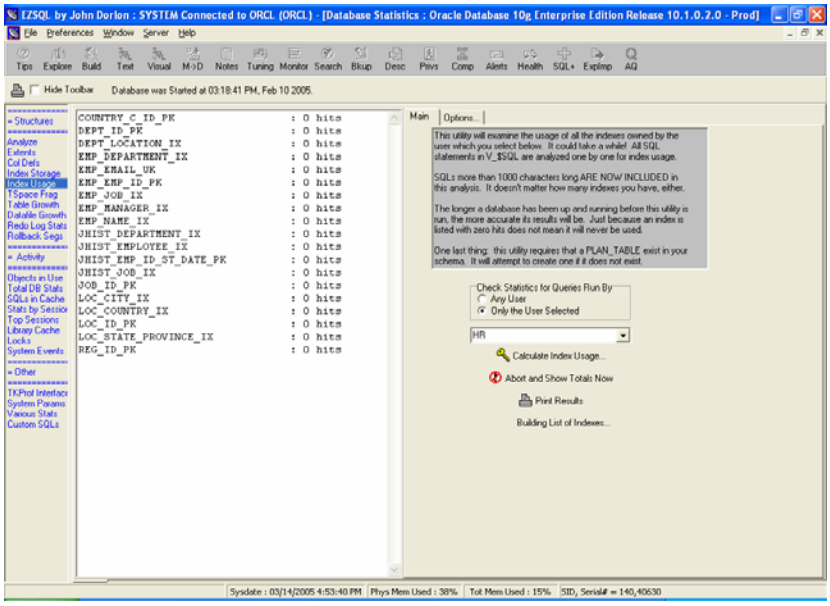

图 25-3　EZSQL by John H. Dorlon 界面

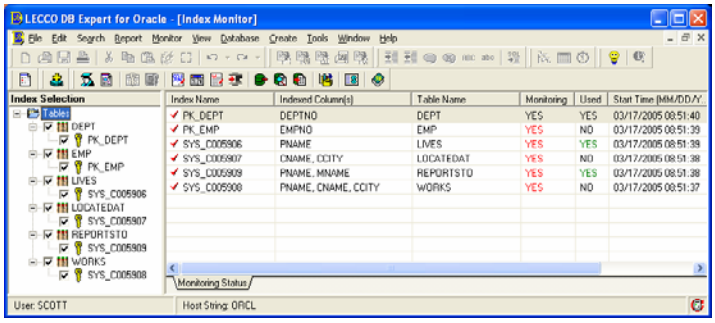

图 25-4　索引监测

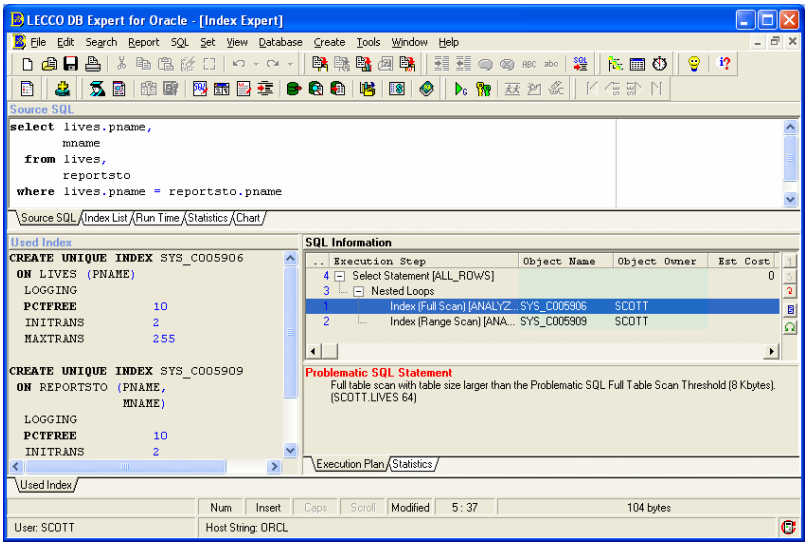

图 25-5　索引调优

25.4　一些索引管理的脚本

1. 列出需要重建的索引（脚本来源：MetaLink）

```
REM ============================================================
REM
REM                     rebuild_indx.sql
REM
REM  Copyright (c) Oracle Software, 1998 - 2000
REM
REM  Author  : Jurgen Schelfhout
REM
REM  The sample program in this article is provided for educational
REM  purposes only and is NOT supported by Oracle Support Services.
REM  It has been tested internally, however, and works as documented.
REM  We do not guarantee that it will work for you, so be sure to test
REM  it in your environment before relying on it.
REM
REM  This script will analyze all the indexes for a given schema
REM  or for a subset of schema's. After this the dynamic view
REM  index_stats is consulted to see if an index is a good
REM  candidate for a rebuild or for a bitmap index.
REM
REM  Database Version : 7.3.X and above.
REM
REM ============================================================

prompt
ACCEPT spoolfile CHAR prompt 'Output-file : ';
ACCEPT schema CHAR prompt 'Schema name (% allowed) : ';
prompt
prompt
prompt Rebuild the index when :
prompt   - deleted entries represent 20% or more of the current entries
prompt   - the index depth is more then 4 levels.
prompt Possible candidate for bitmap index :
prompt   - when distinctiveness is more than 99%
prompt
spool &spoolfile

set serveroutput on
set verify off
declare
 c_name       INTEGER;
 ignore       INTEGER;
 height       index_stats.height%TYPE := 0;
 lf_rows      index_stats.lf_rows%TYPE := 0;
 del_lf_rows   index_stats.del_lf_rows%TYPE := 0;
 distinct_keys index_stats.distinct_keys%TYPE := 0;
```

```
     cursor c_indx is
      select owner, table_name, index_name
      from dba_indexes
      where owner like upper('&schema')
        and owner not in ('SYS','SYSTEM');
    begin
     dbms_output.enable (1000000);
     dbms_output.put_line ('Owner    Index Name      % Deleted Entries Blevel Distinctiveness');
     dbms_output.put_line ('-------- -------------- ----------------- ------ ---------------');

     c_name := DBMS_SQL.OPEN_CURSOR;
     for r_indx in c_indx loop
      DBMS_SQL.PARSE(c_name,'analyze index ' || r_indx.owner || '.' ||
                  r_indx.index_name || ' validate structure',DBMS_SQL.NATIVE);
      ignore := DBMS_SQL.EXECUTE(c_name);

      select HEIGHT, decode (LF_ROWS,0,1,LF_ROWS), DEL_LF_ROWS,
            decode (DISTINCT_KEYS,0,1,DISTINCT_KEYS)
            into height, lf_rows, del_lf_rows, distinct_keys
      from index_stats;
    --
    -- Index is considered as candidate for rebuild when :
    --   - when deleted entries represent 20% or more of the current entries
    --   - when the index depth is more then 4 levels.(height starts counting from 1 so > 5)
    -- Index is (possible) candidate for a bitmap index when :
    --   - distinctiveness is more than 99%
    --
      if ( height > 5 ) OR ( (del_lf_rows/lf_rows) > 0.2 ) then
        dbms_output.put_line (rpad(r_indx.owner,16,' ') || rpad(r_indx.index_name,40,' ') ||
                          lpad(round((del_lf_rows/lf_rows)*100,3),17,' ') ||
                          lpad(height-1,7,'                         ')                         ||
lpad(round((lf_rows-distinct_keys)*100/lf_rows,3),16,' '));
      end if;

     end loop;
     DBMS_SQL.CLOSE_CURSOR(c_name);
    end;
    /

    spool off
    set verify on
```

2. 将索引从一个表空间移到另一个表空间

```
    alter index rebuild tablspace <目标表空间名字> [online];
```

3. 监测某一个索引的使用状况

```
    --*************************************************************************
    --  Copyright 2004 by Rampant TechPress Inc.
    --  Free for non-commercial use!
    --  To license, e-mail info@rampant.cc
```

```
--********************************************************************************
col c1 heading 'Begin|Interval|time' format a20
col c2 heading 'Search Columns'     format 999
col c3 heading 'Invocation|Count'   format 99,999,999

break on c1 ship 2

accept idxname char prompt 'Enter Index Name: '

ttile 'Invocation Counts for index|&idxname'

select
to_char(sn.begin_interval_time, 'yy-mm-dd hh24') c1,
p.search_columns                                 c2,
count(*)                                        c3
from
dba_hist_snapshot  sn,
dba_hist_sql_plan   p,
dba_hist_sqlstat   st
where
st.sql_id = p.sql_id
and
sn.snap_id = st.snap_id
and
p.object_name = '&idxname'
group by
begin_interval_time, search_columns;
```

作者简介

陈宇红，网名 snowhite2000、snow、白雪红尘，ITPUB 数据库管理版现任版主。一直担任某外资企业生产系统的 DBA，参与了多个项目的计划与实施。现担任某外企 PeopleSoft ERP 系统 DBA，对生产环境的 Oracle 产品实施与应用、MS SQL Server 等数据库的生产环境系统的管理有较多的经验。

诸超，网名 chao_ping，ITPUB 数据库管理版现任版主，活跃于 ITPUB 数据库管理版、UNIX 系统管理版、网易 Oracle 数据库版及其他数据库 BBS。曾经担任北京某知名企业 SAP ERP 系统应用 DBA，现就职于上海，担任国内某知名大型网站 DBA。对企业 ERP 系统的实施，超大型数据库的运行和管理、性能调整、数据库的高可用性、SQL 优化和 UNIX 操作系统等方面都有十分深入的研究和丰富的经验。

26

第 26 章　CBO 成本计算初探

本章初步探讨了 CBO 成本计算的基本原理，并简要介绍了初始化参数 optimizer_index_cost_adj 和 db_file_multiblock_read_count 对 CBO 成本计算的影响。

数据库版本：Oracle 9.0.1

操作系统：Windows 2000

```
system@TEST> select *from v$version;
BANNER
----------------------------------------------------------------
Oracle9i Enterprise Edition Release 9.0.1.1.1 - Production
PL/SQL Release 9.0.1.1.1 - Production
CORE    9.0.1.1.1      Production
TNS for 32-bit Windows: Version 9.0.1.1.0 - Production
NLSRTL Version 9.0.1.1.1 – Production
```

26.1　建立测试数据

下面是本章使用的测试数据，在 test 用户下建有 3 张表：test1、test2 和 test3。这 3 张表的 n1 列上均建有索引。

```
system@TEST> conn test/test@test
已连接。
test@TEST> -- 建立执行计划表
test@TEST> @%ORACLE_HOME%\rdbms\admin\utlxplan.sql
表已创建
test@TEST>
test@TEST> -- 建立测试表
test@TEST> -- 表 1,2 除索引列外有其他列,表 3 只有索引列
test@TEST> drop table test1
  2  /
表已丢弃。
test@TEST> create table test1
  2  (
  3  n1    number(10),
  4  c1    char(100)
```

```
  5 )
  6 /
表已创建。
test@TEST> drop table test2
  2 /
表已丢弃。
test@TEST> create table test2
  2 (
  3 n1   number(10),
  4 c1   char(100)
  5 )
  6 /

表已创建。
test@TEST> drop table test3
  2 /
表已丢弃。

test@TEST> create table test3
  2 (
  3 n1   number(10)
  4 )
  5 /
表已创建。
test@TEST> -- 插入 test1 测试数据
test@TEST> begin
  2   for i in 1..5000 loop
  3     insert into test1 values(i,'test');
  4   end loop;
  5 end;
  6 /
PL/SQL 过程已成功完成。
test@TEST> declare
  2   i number;
  3 begin
  4   i := 1;
  5   for j in 1..5000 loop
  6      i := mod(j,250);
  7      insert into test2 values(i,'test');
  8      insert into test3 values(i);
  9   end loop;
 10 end;
 11 /
PL/SQL 过程已成功完成。
test@TEST> commit
  2 /
提交完成。
test@TEST> -- 建立索引
test@TEST> create index idx_test1_n1 on test1(n1)
  2 /
索引已创建。
test@TEST> create index idx_test2_n1 on test2(n1)
```

```
  2  /
索引已创建。
test@TEST> create index idx_test3_n1 on test3(n1)
  2  /
索引已创建。
test@TEST> analyze table test1 compute statistics
  2  /
表已分析。

test@TEST> analyze table test2 compute statistics
  2  /
表已分析。
test@TEST> analyze table test3 compute statistics
  2  /
表已分析。
```

26.2　CBO 计算成本原理初探

CBO 的成本主要由物理 I/O 组成，基于以往的经验，公式大致为：

```
IO + CPU/1000 + Net I/O*1.5
```

IO 表示物理 I/O 请求，CPU 表示逻辑 I/O 请求，Net I/O 表示通过数据库链接访问远程数据库的逻辑 I/O 请求。CBO 会尝试计算所有可能执行计划的物理 I/O，选择只需要最小物理 I/O 的计划。

下面通过几个简单的例子，初步探究 CBO 是如何计算成本的。

以下为表统计信息和索引统计信息：

```
test@TEST> select table_name,blocks,num_rows
  2  from user_tables
  3  /

TABLE_NAME                     BLOCKS   NUM_ROWS
------------------------------ ---------- ----------
PLAN_TABLE
TEST1                            158      5000
TEST2                            158      5000
TEST3                             20      5000

test@TEST> select
  2  table_name ,
  3  num_rows   ,
  4  avg_leaf_blocks_per_key    l_blocks,
  5  avg_data_blocks_per_key    d_blocks,
  6  clustering_factor         cl_fac
  7  from user_indexes
  8  /

TABLE_NAME                   NUM_ROWS   L_BLOCKS   D_BLOCKS    CL_FAC
------------------------------ ---------- ---------- ---------- ----------
TEST1                          5000        1         1       157
TEST2                          5000        1        20      5000
```

```
TEST3                          5000         1        15      3875
```

各列的粗略解释：

- avg_leaf_blocks_per_key：每个索引值的平均叶块数目。
- avg_data_blocks_per_key：每个索引值的平均数据块数目。
- clustering_factor：B 树叶块和表数据之间的关系称为 clustering_factor。索引叶子块指向的数据块越多，该参数值越小，在范围扫描使用索引的性能越好。如果该值与表的块数相接近，表示表行依次按索引排序，如果该值与表的行数接近，表示表行不是按索引排序。该值很高的索引通常在范围扫描中不使用。关于 clustering_factor 的更多解释可以参考本章“参考信息”中 biti_rainy 的文章。

```
test@TEST> set autotrace trace exp
test@TEST> select *from test1 where n1 = 100
  2  /
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=103)
   1   0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST1' (Cost=2 Card=1 By
         tes=103)
   2   1     INDEX (RANGE SCAN) OF 'IDX_TEST1_N1' (NON-UNIQUE) (Cost=
         1 Card=1)
```

Cost 计算大致如下：从 t1 的索引统计信息中得知，idx_test1_n1 的 l_blocks 和 d_blocks 均为 1，Cost =1+1=2，索引叶块物理读取 Cost 为 1，数据块物理读取 Cost 为 1。

```
test@TEST> select *from test2 where n1 = 100
  2  /

已选择 20 行。
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=21 Card=20 Bytes=206
         0)
   1   0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST2' (Cost=21 Card=20
         Bytes=2060)
   2   1     INDEX (RANGE SCAN) OF 'IDX_TEST2_N1' (NON-UNIQUE) (Cost=
         1 Card=20)
```

Cost 计算大致如下：从 t2 的索引统计信息中可以看出，idx_test2_n1 的 l_blocks 为 1，d_blocks 为 20，Cost =1+20 =21，其中索引叶块物理读取 Cost 为 1，数据块物理读取 Cost 为 20。

```
test@TEST>
test@TEST> select *from test3 where n1 = 100
  2  /
已选择 20 行。
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=20 Bytes=60)
   1   0   INDEX (RANGE SCAN) OF 'IDX_TEST3_N1' (NON-UNIQUE) (Cost=1
         Card=20 Bytes=60)
```

Cost 计算大致如下：从 t3 的索引统计信息，idx_test3_n1 的 l_blocks 为 1，d_blocks 为 15，但因为无需访问表，故 Cost =1，索引叶块物理读取 Cost 为 1。

如果 select 子句中选择的只有索引列的话，则不需要访问表数据块，实验如下：

```
test@TEST> select n1 from test2 where n1=100;
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=20 Bytes=60)
   1    0   INDEX (RANGE SCAN) OF 'IDX_TEST2_N1' (NON-UNIQUE) (Cost=1
         Card=20 Bytes=60)
```

Cost 计算大致如下：Cost =1，索引叶块物理读取 Cost 为 1。

26.3　初始化参数以及优化器模式对执行计划的影响

下面简要介绍初始化参数 db_file_multiblock_read_count 和 optimizer_index_cost_adj 如何影响 CBO 选择执行计划以及 CBO 在 FIRST_ROWS 优化器模式下的工作方式。

26.3.1　初始化参数 db_file_multiblock_read_count

由经验所得，在多块读取中，表 26-1 大致表示了 db_file_multiblock_read_count 与 Cost 的换算公式：

表 **26-1**　　**db_file_multiblock_read_count** 与 **Cost** 的换算公式

db_file_multiblock_read_count	调　整　值
4	4.175
8	6.589
16	10.398
32	16.409
64	25.895
128	40.865

本例中 db_file_multiblock_read_count 的参数值如下：

```
test@TEST> show parameter db_file_multiblock_read_count
NAME                                 TYPE       VALUE
------------------------------------ ---------- -------------
db_file_multiblock_read_count        integer    8
```

根据表的统计信息和换算因子大致可以估算出全表扫描的 Cost。

test1：158/6.589=23.979 约为 24。

test2：158/6.589=23.979 约为 24。

test3：20/6.589=3.035 约为 4。

以上值均大于使用索引的 Cost。

下面通过修改该参数来看看执行计划的变化：

```
test@TEST>
test@TEST> alter session set db_file_multiblock_read_count = 16
  2  /

会话已更改。
```

```
test@TEST> select *
  2  from test2
  3  where n1 = 100
  4  /

已选择 20 行。
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=16 Card=20 Bytes=206
         0)
   1   0   TABLE ACCESS (FULL) OF 'TEST2' (Cost=16 Card=20 Bytes=2060
         )
```

执行计划变成全表扫描。

Cost 计算大致如下：使用索引的 Cost 为 21，使用全表扫描的 Cost=158/10.398=15.195，取整后为 16，因而使用全表扫描。

```
test@TEST> select *
  2  from test3
  3  where n1 = 100
  4  /

已选择 20 行。

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=20 Bytes=60)
   1   0   INDEX (RANGE SCAN) OF 'IDX_TEST3_N1' (NON-UNIQUE) (Cost=1
         Card=20 Bytes=60)
```

执行计划保持不变，因为使用索引执行一次叶块读取的 Cost 为 1，全表扫描的 Cost 为 20/10.398=1.923，取整为 2。

```
test@TEST>
test@TEST> alter session set db_file_multiblock_read_count=32
  2  /

会话已更改。

test@TEST> select *
  2  from test2
  3  where n1 = 100
  4  /

已选择 20 行。

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=20 Bytes=206
         0)
   1   0   TABLE ACCESS (FULL) OF 'TEST2' (Cost=10 Card=20 Bytes=2060
         )
```

执行计划为全表扫描，Cost 计算大致如下：使用索引的 Cost 为 21，使用全表扫描的

Cost=158/16.409=9.628，取整后为 10，因而选择全表扫描。

```
test@TEST> select *
  2  from test3
  3  where n1 = 100
  4  /

已选择 20 行。
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=20 Bytes=60)
   1    0   INDEX (RANGE SCAN) OF 'IDX_TEST3_N1' (NON-UNIQUE) (Cost=1
         Card=20 Bytes=60)
```

执行计划保持不变，因为使用索引的 Cost 为 1，索引叶块的一次物理读取。

26.3.2　初始化参数 optimizer_index_cost_adj

参数 optimizer_index_cost_adj 是 1～100 之间的一个百分值，表示索引访问和全表扫描之间相关物理 I/O 请求 Cost 的一个比值。默认值 100 意味着索引访问与全表扫描是完全等价的。

基于以往的经验，最终 Cost=最初 Cost*optimizer_index_cost_adj/100。

```
test@TEST> show parameter db_file_multiblock_read_count
NAME                                 TYPE       VALUE
------------------------------------ ----------- --------
db_file_multiblock_read_count        integer    32
test@TEST> alter session set optimizer_index_cost_adj=50
  2  /

会话已更改。

test@TEST> select *
  2  from test2
  3  where n1 = 100
  4  /

已选择 20 行。
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=20 Bytes=206
         0)
   1    0   TABLE ACCESS (FULL) OF 'TEST2' (Cost=10 Card=20 Bytes=2060
         )
```

执行计划为全表扫描，Cost 计算大致如下：使用索引的 Cost=1+20*0.5=11，使用全表扫描的 Cost=158/16.409=9.628，取整后为 10，选择全表扫描。

```
test@TEST>
test@TEST> alter session set optimizer_index_cost_adj=25
  2  /
会话已更改。
test@TEST> select *
  2  from test2
  3  where n1 = 100
```

```
  4  /
已选择 20 行。

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=6 Card=20 Bytes=2060
         )
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST2' (Cost=6 Card=20 B
         ytes=2060)
   2    1     INDEX (RANGE SCAN) OF 'IDX_TEST2_N1' (NON-UNIQUE) (Cost=
         1 Card=20)
```

执行计划选择了使用索引，Cost 计算大致如下：使用索引的 Cost=1+20*0.25=6，索引叶物理读取 Cost 为 1，表数据块物理读取为 5；使用全表扫描的 Cost=158/16.409=9.628，取整后为 10，数据块物理读取 Cost 为 10，选择使用索引。

26.3.3 优化器模式 FIRST_ROWS 对执行计划的影响

FIRST_ROWS 优化模式适用于 OLTP 系统，因为 OLTP 用户关心快速看到某些行而很少关心看到整个查询的结果。

该优化模式混合使用成本和试探的方法搜寻快速提取最先一些行的最好计划。试探性的方法有时会引导 CBO 产生成本比没有试探性方式时大很多的计划。比如，一些试探性的方法有：

- 全表扫描在与其他表访问方式比较时被认为具有无限大的 Cost，因此在没有可用索引的情况下才会使用全表扫描。
- order by 语句会引起索引访问。
- OR 扩充无效，相当于指定了提示：NO_EXPAND。

注意：下面例子设置参数 db_file_multiblock_read_count 为 32。

```
test@TEST> alter session set db_file_multiblock_read_count=32
  2  /
会话已更改。
test@TEST> select *from test2 where n1 = 100
  2  /
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=20 Bytes=206
         0)
   1    0   TABLE ACCESS (FULL) OF 'TEST2' (Cost=10 Card=20 Bytes=2060
         )
 -- 加 hint,first_rows
test@TEST> select /*+first_rows*/ *from test2 where n1 = 100
  2  /
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=21 Card=20
          Bytes=2060)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST2' (Cost=21 Card=20
         Bytes=2060)
   2    1     INDEX (RANGE SCAN) OF 'IDX_TEST2_N1' (NON-UNIQUE) (Cost=
```

```
          1 Card=20)
```

执行计划中的表访问方式变为使用索引，虽然 Cost=21 要比全表扫描的 Cost=10 要高，但 FIRST_ROWS 优化模式使执行计划更倾向于选择使用索引。

26.4　小结

本章通过几个例子，初步探究了 CBO 成本计算的基本知识和初始化参数以及优化器模式对执行计划的影响。

参考信息

1．Oracle 在线文档：
http://tahiti.oracle.com
http://www.evdbt.com/
http://www.evdbt.com/SearchIntelligenceCBO.doc
http://www.evdbt.com/OOUG%20CBO.pps
2．Eygle，OPTIMIZER_INDEX_COST_ADJ 与成本计算
http://www.eygle.com/sql/OPTIMIZER_INDEX_COST_ADJ.htm
3．Biti_rainy，what is the clustering_factor ?
http://blog.itpub.net/post/330/2970

作者简介

何小栋，网名 husthxd，毕业于华中理工大学，一直从事政府行业软件的开发和实施，但时常“不务正业”地研究 Oracle 数据库。擅长后台开发、系统分析与设计，目前沉迷于 XP。希望能结交广大喜欢 Oracle 和 Java 的朋友，一起交流，共同进步。

E-mail：husthxd@itput.net
MSN：husthxd@hotmail.com
Blog：blog.itpub.net/husthxd

第27章 Bitmap索引

本章首先介绍了 Bitmap 索引的概念，然后通过一些例子来说明 Bitmap 索引的特点，接着描述了 Bitmap 索引的适用范围，最后简单介绍了一下 9i 中的新特性——Bitmap Join 索引。

27.1 Bitmap 索引的概念

Oracle 通过索引机制来加速数据的存取。Oracle 的 B 树索引（B-Tree 索引）以树型结构来存放键值（key value）。每个 B 树索引都有一个根节点（root），对索引进行访问时首先从根节点开始，根据和根节点上键值的比较结果，找到相应的分支节点（branch），再和分支节点的键值进行比较，最后一直找到叶子（leaf）节点。在叶子节点上，存放索引的键值和表中对应这个键值的记录的物理地址（rowid）。通过 rowid，就可以直接定位到表的记录。对于惟一索引，一个键值和一条记录的 rowid 相对应，对于非惟一索引，一个键值可能对应多条记录的 rowid。B 树索引对于具有惟一键值或者键值很少重复的情况，具有很高的存取效率。但是对于那些键值大量重复的列，B 树索引并不适用，因为即使找到对应的键值仍然要扫描大量的 rowid，这种情况下使用 B 树索引已经没有什么意义了。

Oracle 针对这种情况提供了 Bitmap 索引。Bitmap 索引适用于键值大量重复的列的查询。Bitmap 索引对索引列的每一个键值分别索引。对于一个键值，可能分成一到多个范围进行存储。每个键值的存储范围大致包括以下几个部分。首先是索引的键值，接着存放当前范围的起始 rowid 和终止 rowid，最后是这个键值在这个范围内的位置编码。将这个十六进制编码转化为二进制后，编码值是 1 的代表记录符合索引的键值，是 0 则表示不符合。由于保存了起始和终止的 rowid 以及在这个范围内的位置变化，因此通过转换，Bitmap 索引也可以对应到 rowid。所以，Bitmap 索引可以提供和 B-Tree 索引相同的功能。

键值的每个存储范围的大小和分段依据，主要与每次 DML 操作影响的记录数和这些键值相同的记录在数据库中的物理分布有关。不过二者比较起来，前者的影响要大得多。下面用 insert 操作来举个例子。如果插入一条记录（insert into values），那么 Oracle 会为这条记录中的键值建立一个范围，范围的大小是 8 条记录（Bitmap 索引中的最小范围，所有 Bitmap 索引的范围必然是 8 的倍数。其他 7 条记录为空，可以被后续的插入操作继续使用）。如果执行一条批量插入语句（insert into select），对于这条插入语句的中包含的每一个键值而言，都会建立一个范围。范围的大小为 CEIL（操

作记录数/8）*8。实际情况由于考虑利用原有范围内的空闲空间情况，以及考虑到键值的 rowid 分布等因素，要远比这里描述的复杂得多，而且对于 delete 和 update 操作而言，情况更为复杂。但是有一点是明确的，即 Oracle 会对批量插入（修改）的数据一起进行索引。如果一次性插入 1000 条键值相同的数据，则只会索引一次，并生成一个包括 1000 条记录的范围。如果一条一条地插入 1000 条数据，则会索引 1000 次，并生成 125 个包括 8 条记录的范围。无论是从效率的角度考虑还是从空间占用角度考虑，批量操作都是对包含 Bitmap 索引的表的首选。

这里只是简单介绍了一下 Bitmap 索引的概念，由于篇幅限制，没有对内部存储结构和内部机制进行深入讨论。如果对这方面内容感兴趣，可以参考 ITPUB 上的 Oracle 专题深入讨论板块中的文章“ Bitmap 的一点探究 ”（在本章后面的“ 参考信息 ”中给出了链接）。

27.2 建立测试例子

Oracle 的标准版不支持 Bitmap 索引。首先检查 Oracle 的版本和选项。

```
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 32-bit Windows: Version 9.2.0.4.0 - Production
NLSRTL Version 9.2.0.4.0 – Production

SQL> COL PARAMETER FORMAT A40
SQL> COL VALUE FORMAT A30
SQL> SELECT * FROM V$OPTION WHERE PARAMETER LIKE 'Bit%';
PARAMETER                              VALUE
---------------------------------------- --------------------
Bit-mapped indexes                       TRUE
```

检查和 Bitmap 索引有关的初始化函数：

```
SQL> SHOW PARAMETER BITMAP
NAME                                 TYPE       VALUE
------------------------------------ ----------- ---------
bitmap_merge_area_size               integer    1048576
create_bitmap_area_size              integer    8388608
SQL> SHOW PARAMETER SORT_AREA_SIZE
NAME                                 TYPE       VALUE
------------------------------------ ----------- ------------------
sort_area_size                       integer    524288
```

bitmap_merge_area_size 是 Bitmap 索引进行 merge 操作时使用的内存区域，create_bitmap_area_size 是创建 Bitmap 索引的内存区域。在 8i 中，可以根据 Bitmap 索引的大小和使用频繁程度来调整这两个初始化参数，来提高 Bitmap 索引的性能。在 9i 中，只需要设置初始化参数 pga_aggregate_target 的值，则 Oracle 会根据需要自动对内存的使用进行相应的调整。sort_area_size 是批量插入或批量更新数据时 Bitmap 索引使用的内存大小，应当调整 sort_area_size 的大小以容纳批量操作的数据。

确定 Oracle 版本支持 Bitmap 索引之后，下面来构造测试的例子。

```
SQL> CREATE TABLE TEST_BITMAP AS SELECT * FROM DBA_OBJECTS;
表已创建。

SQL> DESC TEST_BITMAP
 名称                                      是否为空? 类型
 ----------------------------------------- -------- ------------
 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
  2  COUNT(DISTINCT OWNER) OWNER,
  3     COUNT(DISTINCT OBJECT_NAME) OBJECT_NAME,
  4     COUNT(DISTINCT SUBOBJECT_NAME) SUBOBJECT_NAME,
  5     COUNT(DISTINCT OBJECT_ID) OBJECT_ID,
  6     COUNT(DISTINCT DATA_OBJECT_ID) DATA_OBJECT_ID
  7     FROM TEST_BITMAP;
    OWNER OBJECT_NAME SUBOBJECT_NAME  OBJECT_ID DATA_OBJECT_ID
---------- ----------- -------------- ---------- --------------
      31      19526           35     32206         2914

SQL> SELECT
  2     COUNT(DISTINCT OBJECT_TYPE) OBJECT_TYPE,
  3     COUNT(DISTINCT CREATED) CREATED,
  4     COUNT(DISTINCT LAST_DDL_TIME) LAST_DDL_TIME,
  5     COUNT(DISTINCT TIMESTAMP) TIMESTAMP,
  6     COUNT(DISTINCT STATUS) STATUS
  7     FROM TEST_BITMAP;
OBJECT_TYPE    CREATED LAST_DDL_TIME  TIMESTAMP    STATUS
----------- ---------- ------------- ---------- ----------
       34      2211         2761      2646         1

SQL> SELECT
  2     COUNT(DISTINCT TEMPORARY) TEMPORARY,
  3     COUNT(DISTINCT GENERATED) GENERATED,
  4     COUNT(DISTINCT SECONDARY) SECONDARY,
  5     COUNT(*) COUNT
  6  FROM TEST_BITMAP;
 TEMPORARY  GENERATED  SECONDARY     COUNT
---------- ---------- ---------- ----------
       2         2         1     32214
```

通过上面的查询可以发现，OBJECT_ID 和 OBJECT_NAME 的值比较接近表中的总记录数，这两列比较适合建立 B 树索引。

而 OWNER、SUBOBJECT_NAME、OBJECT_TYPE、TEMPORARY、GENERATED 等列，重复的值比较多，适合建立 Bitmap 索引。

下面再创建一个过程，将 dbms_space.unused_space 过程封装了一下，通过调用 dbms_output.put_line 打印结果。

```
create or replace procedure p_unused_space(p_object_name in varchar2,
   p_object_type in varchar2 default 'TABLE',
   p_owner in varchar2 default user,
   p_partition_name in varchar2 default '') is
   v_total_blocks number;
   v_total_bytes number;
   v_unused_blocks number;
   v_unused_bytes number;
   v_last_used_extent_file_id number;
   v_last_used_extent_block_id number;
   v_last_used_block number;
begin
   dbms_space.unused_space(upper(p_owner),  upper(p_object_name),  upper(p_object_type),
v_total_blocks,
      v_total_bytes, v_unused_blocks, v_unused_bytes, v_last_used_extent_file_id,
      v_last_used_extent_block_id, v_last_used_block, upper(p_partition_name));
   dbms_output.put_line('total_blocks is ' || v_total_blocks);
   dbms_output.put_line('total_bytes is ' || v_total_bytes);
   dbms_output.put_line('unused_blocks is ' || v_unused_blocks);
   dbms_output.put_line('unused_bytes is ' || v_unused_bytes);
   dbms_output.put_line('last_used_extent_file_id is ' || v_last_used_extent_file_id);
   dbms_output.put_line('last_used_extent_block_id is ' || v_last_used_extent_block_id);
   dbms_output.put_line('last_used_block is ' || v_last_used_block);
end;
/
```

然后，检查一下 PLAT_TABLE 表或同义词是否存在，如果不存在的话执行 $ORACLE_HOME/rdbms/admin/utlxplan.sql。

最后，建立 Bitmap 索引。其实建立 Bitmap 索引的语法十分简单，只不过是在 CREATE INDEX 之间增加了 Bitmap 关键字而已，其他语法和 B 树索引几乎完全一样。

```
SQL> CREATE BITMAP INDEX IND_B_OWNER ON TEST_BITMAP(OWNER);
索引已创建。
```

27.3 Bitmap 索引的特点

Bitmap 索引的特点主要体现在以下 7 个方面。

27.3.1 Bitmap 索引比 B 树索引要节省空间

一般来说，Bitmap 索引要比 B 树索引节省空间。因为 B 树索引要保存表中所有键值非空记录

的 rowid。而 Bitmap 索引采用分段记录的方法，它不会记录全部的 rowid，而是只记录一个范围的起始地址和终止地址。

Bitmap 存储空间大小由以下几点决定：

- 表中总的记录的大小

这一点勿庸置疑，在其他条件相同的情况下，肯定是表的记录越多，Bitmap 索引的存储空间越大。

- 索引列的键值多少

因为 Bitmap 索引对不同的键值分别存储。因此索引列的键值越多，所要占用的存储空间也就越多。

- 操作的类型

Bitmap 索引的范围多少与操作数量有关。同样插入 1000 条数据，采用批量方式和分 1000 次操作插入，产生的范围数相差极大。批量操作要比单条记录操作节省大量的空间。

- 相同键值记录的分布

如果相同键值的记录在物理分布上较为分散，那么在索引时可能会放到两个范围中。不过，与上面三点相比，这点对占用空间的影响要小得多。

在一般情况下，由于不需要存储大量 rowid，因此 Bitmap 索引一般比较节省空间。

```
SQL> SET SERVEROUTPUT ON

SQL> CREATE BITMAP INDEX IND_B_OBJECT_TYPE ON TEST_BITMAP (OBJECT_TYPE);
索引已创建。

SQL> EXEC P_UNUSED_SPACE('IND_B_OBJECT_TYPE', 'INDEX');
total_blocks is 64
total_bytes is 1048576
unused_blocks is 60
unused_bytes is 983040
last_used_extent_file_id is 9
last_used_extent_block_id is 1028
last_used_block is 4
PL/SQL 过程已成功完成。

SQL> DROP INDEX IND_B_OBJECT_TYPE;
索引已丢弃。

SQL> CREATE INDEX IND_OBJECT_TYPE ON TEST_BITMAP(OBJECT_TYPE);
索引已创建。

SQL> EXEC P_UNUSED_SPACE('IND_OBJECT_TYPE', 'INDEX')
total_blocks is 64
total_bytes is 1048576
unused_blocks is 16
unused_bytes is 262144
last_used_extent_file_id is 9
last_used_extent_block_id is 1028
last_used_block is 48
PL/SQL 过程已成功完成。
```

上面的例子中，在 OBJECT_TYPE 列，先后建立 Bitmap 索引和 B 树索引，比较二者所用空间。64 个 block 中 B 树索引使用了 48 个 block，而 Bitmap 索引只用了 4 个 block。

当然，Bitmap 索引节省空间也不是绝对的。如果把 Bitmap 索引建立在接近惟一的列上，那么它占用的空间可能会比 B 树索引还要大。例如：

```
SQL> CREATE BITMAP INDEX IND_B_OBJECT_ID ON TEST_BITMAP (OBJECT_ID);
索引已创建。

SQL> EXEC P_UNUSED_SPACE('IND_B_OBJECT_ID', 'INDEX')
total_blocks is 64
total_bytes is 1048576
unused_blocks is 5
unused_bytes is 81920
last_used_extent_file_id is 9
last_used_extent_block_id is 964
last_used_block is 59
PL/SQL 过程已成功完成。

SQL> DROP INDEX IND_B_OBJECT_ID;
索引已丢弃。

SQL> CREATE INDEX IND_OBJECT_ID ON TEST_BITMAP (OBJECT_ID);
索引已创建。

SQL> EXEC P_UNUSED_SPACE('IND_OBJECT_ID', 'INDEX')
total_blocks is 64
total_bytes is 1048576
unused_blocks is 25
unused_bytes is 409600
last_used_extent_file_id is 9
last_used_extent_block_id is 964
last_used_block is 39
PL/SQL 过程已成功完成。
```

27.3.2 Bitmap 索引建立的速度比较快

由于 B 树索引在建立时需要排序、定位等复杂的操作，而 Bitmap 索引并不需要排序，而且占用存储空间也较少，因此，建立 Bitmap 索引通常比建立 B 树索引更快。

一般来说，数据量越大，二者之间的速度差异越明显。

27.3.3 基于规则的优化器无法使用 Bitmap 索引

基于规则的优化器不会使用 Bitmap 索引。只有基于代价的优化器才会利用 Bitmap 索引。见下面例子：

```
SQL> SHOW PARAMETER OPTIMIZER_MODE
NAME                                 TYPE       VALUE
------------------------------------ ----------- -------
optimizer_mode                        string     CHOOSE
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OWNER = 'YANGTK';
  COUNT(*)
```

```
----------
       20

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
   1    0  SORT (AGGREGATE)
   2    1    TABLE ACCESS (FULL) OF 'TEST_BITMAP'

SQL> ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
会话已更改。

SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OWNER = 'YANGTK';
  COUNT(*)
----------
       20

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=2 Card=1 Bytes=17)
   1    0  SORT (AGGREGATE)
   2    1    BITMAP CONVERSION (COUNT)
   3    2      BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'

SQL> ALTER SESSION SET OPTIMIZER_MODE = RULE;
会话已更改。

SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OWNER = 'YANGTK';
  COUNT(*)
----------
       20

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=RULE
   1    0  SORT (AGGREGATE)
   2    1    TABLE ACCESS (FULL) OF 'TEST_BITMAP'

SQL> ALTER SESSION SET OPTIMIZER_MODE = CHOOSE;
会话已更改。

SQL> ANALYZE TABLE TEST_BITMAP COMPUTE STATISTICS;
表已分析。

SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OWNER = 'YANGTK';
  COUNT(*)
----------
       20

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=5)
```

```
   1   0  SORT (AGGREGATE)
   2   1    BITMAP CONVERSION (COUNT)
   3   2      BITMAP INDEX (FAST FULL SCAN) OF 'IND_B_OWNER'
```

可以看出，由于开始没有收集统计信息，则 CHOOSE 选择使用了基于规则的优化器，没有使用 Bitmap 索引，而是使用了全表扫描。将优化器改为基于代价的，则使用 Bitmap 索引。对于 Oracle 9i，默认采用 CHOOSE 模式，因此，一般要收集统计信息后，才会选择基于代价的优化器而使用 Bitmap 索引。

27.3.4 Bitmap 索引存储 NULL 值

和 B 树索引不一样，Bitmap 索引存储 NULL 值，因此对 NULL 进行判断的查询可以使用 Bitmap 索引。由于存储 NULL 值，所以在计算表中的记录总数时，会优先使用 Bitmap 索引。

```
SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OBJECT_ID IS NULL;
  COUNT(*)
----------
         8

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=23 Card=1 Bytes=4)
   1   0  SORT (AGGREGATE)
   2   1    TABLE ACCESS (FULL) OF 'TEST_BITMAP' (Cost=23 Card=8 Bytes=32)

SQL> CREATE BITMAP INDEX IND_B_OBJECT_TYPE ON TEST_BITMAP (OBJECT_TYPE);
索引已创建。

SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OBJECT_TYPE IS NULL;
  COUNT(*)
----------
         0

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=8)
   1   0  SORT (AGGREGATE)
   2   1    BITMAP CONVERSION (COUNT)
   3   2      BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OBJECT_TYPE'
```

由于 B 树索引不记录空值，因此索引列的 IS NULL 操作必须通过全表扫描。而对于 Bitmap 索引列 IS NULL 操作则可以访问索引完成。

```
SQL> SELECT COUNT(*) FROM TEST_BITMAP;
  COUNT(*)
----------
     32214

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1)
   1   0  SORT (AGGREGATE)
```

```
   2    1    BITMAP CONVERSION (COUNT)
   3    2      BITMAP INDEX (FAST FULL SCAN) OF 'IND_B_OWNER'

SQL> ALTER TABLE TEST_BITMAP ADD (ID NUMBER);
表已更改。

SQL> UPDATE TEST_BITMAP SET ID = ROWNUM;
已更新 32214 行。

SQL> COMMIT;
提交完成。

SQL> ALTER TABLE TEST_BITMAP ADD CONSTRAINT PK_TEST_BITMAP PRIMARY KEY (ID);
表已更改。

SQL> SELECT INDEX_NAME, INDEX_TYPE FROM USER_INDEXES WHERE TABLE_NAME = 'TEST_BITMAP';
INDEX_NAME                    INDEX_TYPE
------------------------------ --------------------------
IND_B_OBJECT_TYPE              BITMAP
IND_B_OWNER                   BITMAP
IND_OBJECT_ID                  NORMAL
PK_TEST_BITMAP                 NORMAL

SQL> SELECT COUNT(*) FROM TEST_BITMAP;
  COUNT(*)
----------
    32214

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     BITMAP CONVERSION (COUNT)
   3    2       BITMAP INDEX (FAST FULL SCAN) OF 'IND_B_OWNER'
```

即使表中包含了主键，在执行 SELECT COUNT(*)操作时，仍然会使用优先使用 Bitmap 索引。

27.3.5　通过 Bitmap 索引访问表记录

Bitmap 索引更适合计算记录总数，即 SELECT COUNT(*)操作，这是由于 Bitmap 索引的结构造成的。由于 Bitmap 索引中没有存储每个键值对应的 rowid，所以在访问表中的记录时，Oracle 必须使用内部函数，通过起始 rowid 和终止 rowid，以及索引键值在这个范围内位置的编码计算出这个记录的实际物理地址（rowid）。而 SELECT COUNT(*)操作，Oracle 通过累计编码就可以得到相应的值，省去了映射到实际 rowid 的操作的开销。

```
SQL> SELECT OWNER, COUNT(*) FROM TEST_BITMAP GROUP BY OWNER;
OWNER                         COUNT(*)
------------------------------ ----------
CTXSYS                            263
HR                                 34
HR1                                62
```

```
LBACSYS                          208
MDSYS                            247
ODM                              444
ODM_MTR                           12
OE                                91
OEM_YANGTINKUN                   670
OLAPSYS                          718
ORDPLUGINS                        29
ORDSYS                           970
OUTLN                              7
PM                                26
PUBLIC                         12602
QS                                48
QS_ADM                             7
QS_CBADM                          28
QS_CS                             28
QS_ES                             44
QS_OS                             44
QS_WS                             44
REPADMIN                           4
SH                               179
SYS                            13938
SYSTEM                           411
TEST                               2
WKSYS                            279
WMSYS                            131
XDB                              624
YANGTK                            20
已选择 31 行。

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=31 Bytes=155)
   1    0  SORT (GROUP BY NOSORT) (Cost=2 Card=31 Bytes=155)
   2    1    BITMAP CONVERSION (COUNT)
   3    2      BITMAP INDEX (FULL SCAN) OF 'IND_B_OWNER'

SQL> SELECT COUNT(*) FROM TEST_BITMAP WHERE OWNER = 'TEST';
  COUNT(*)
----------
         2

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=5)
   1    0  SORT (AGGREGATE)
   2    1    BITMAP CONVERSION (COUNT)
   3    2      BITMAP INDEX (FAST FULL SCAN) OF 'IND_B_OWNER'

SQL> COL OBJECT_NAME FORMAT A30
SQL> SELECT OWNER, OBJECT_NAME FROM TEST_BITMAP WHERE OWNER = 'TEST';
OWNER                      OBJECT_NAME
```

```
------------------------------ ------------------------------
TEST                           DEPT
TEST                           SYS_C003762

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=23 Card=1039 Bytes=29092)
   1    0   TABLE ACCESS (FULL) OF 'TEST_BITMAP' (Cost=23 Card=1039 Bytes=29092)
```

可以看出，执行 SELECT COUNT(*)操作，Oracle 选择了使用 Bitmap 索引，但是，查询详细记录信息时，虽然只有两条记录，Oracle 却选择了全表扫描，这是为什么呢，继续往下看。

```
SQL> ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
会话已更改。

SQL> SELECT OWNER, OBJECT_NAME FROM TEST_BITMAP WHERE OWNER = 'TEST';
OWNER                          OBJECT_NAME
------------------------------ ------------------------------
TEST                           DEPT
TEST                           SYS_C003762

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=31 Card=1039 Bytes=29092)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP' (Cost=31 Card=1039 Bytes=29092)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'

SQL> ALTER SESSION SET OPTIMIZER_MODE = ALL_ROWS;
会话已更改。

SQL> SELECT OWNER, OBJECT_NAME FROM TEST_BITMAP WHERE OWNER = 'TEST';
OWNER                          OBJECT_NAME
------------------------------ ------------------------------
TEST                           DEPT
TEST                           SYS_C003762

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=23 Card=1039 Bytes=29092)
   1    0   TABLE ACCESS (FULL) OF 'TEST_BITMAP' (Cost=23 Card=1039 Bytes=29092)
```

都是基于代价的优化模式，但是 FRIST_ROWS 和 ALL_ROWS 采用的执行计划并不一样。FIRST_ROWS 采用尽快返回部分结果的方式来计算代价，而 ALL_ROWS 采用计算返回总体结果代价最小的方法。通过索引定位数据，当返回的数据量很小时，效率很高。但是如果返回的结果集超过表大小的十分之一时，索引扫描的代价就有可能大于全表扫描的代价，而且随着返回记录的增多，索引扫描的代价增加得越多。这也就是为什么 FIRST_ROWS 和 ALL_ROWS 采用两种查询方案的原因。FIRST_ROWS 为了快速返回部分数据，则索引扫描是最迅速的。而 ALL_ROWS 需要考虑的是返回所有数据，而 Bitmap 索引键值重复的记录又比较多，因此 Oracle 在这里选择了全表扫描。虽然在这个例子中，只会返回两条记录，使用索引扫描要明显比采用全表扫描快，但是 Oracle 并不知道这些信息，因此它无法做出正确的判断。

```
SQL> ANALYZE TABLE TEST_BITMAP COMPUTE STATISTICS FOR ALL INDEXED COLUMNS;
表已分析。

SQL> SELECT OWNER, OBJECT_NAME FROM TEST_BITMAP WHERE OWNER = 'TEST';
OWNER                          OBJECT_NAME
------------------------------ ------------------------------
TEST                           DEPT
TEST                           SYS_C003762

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=2 Bytes=56)
   1    0  TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP' (Cost=2 Card=2 Bytes=56)
   2    1    BITMAP CONVERSION (TO ROWIDS)
   3    2      BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'

SQL> ALTER SESSION SET OPTIMIZER_MODE = CHOOSE;
会话已更改。

SQL> SELECT OWNER, OBJECT_NAME FROM TEST_BITMAP WHERE OWNER = 'TEST';
OWNER                          OBJECT_NAME
------------------------------ ------------------------------
TEST                           DEPT
TEST                           SYS_C003762

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=2 Bytes=56)
   1    0  TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP' (Cost=2 Card=2 Bytes=56)
   2    1    BITMAP CONVERSION (TO ROWIDS)
   3    2      BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'
```

通过收集 OWNER 列的统计信息，Oracle 了解到不同 OWNER 对应的记录数，因此做出了正确的判断。

在 WHERE 后面跟多个限定条件时，Bitmap 索引可以发挥出更大的作用。除了 Bitmap Join 索引外，Bitmap 索引基本上都是单列索引。当对多个 Bitmap 索引列进行限定时，Oracle 可以利用 BITMAP AND 和 BITMAP OR 操作，在读取数据前就把不需要的数据过滤掉。这种操作正是 Bitmap 索引的优势所在。

表 27-1 Object_Type

TABLE	INDEX	MATERIALIZED VIEW	OTHERS
1	0	0	0
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

表 27-2　　Owner

SYS	YANGTK	TEST	OTHERS
1	0	0	0
0	0	1	0
0	0	0	1
1	0	0	0
0	1	0	0
1	0	0	0
0	1	0	0

上面两张表格分别列出了 OBJECT_TYPE 和 OWNER 两列的 Bitmap 索引情况。由于键值较多，用 OTHERS 表示其他没有列出的情况之和。假设表中存在 7 行数据，根据上面 Bitmap 索引表格可以看出，第 1、2、7 行数据是表，3、6 行是索引，第 5 行是物化视图，第 4 行是其他类型。第 1、4、6 行是 SYS 用户数据，第 5、7 行是 YANGTK 用户数据，第 2 行是 TEST 用户数据，第 3 行是其他用户数据。

如果进行如下查询：

```
SQL> SELECT OWNER, OBJECT_NAME, OBJECT_TYPE FROM TEST_BITMAP
  2  WHERE (OWNER = 'TEST' AND OBJECT_TYPE = 'TABLE')
  3  OR OBJECT_TYPE = 'MATERIALIZED VIEW';
OWNER     OBJECT_NAME         OBJECT_TYPE
---------- -------------------- ------------------
SH        CAL_MONTH_SALES_MV   MATERIALIZED VIEW
SH        FWEEK_PSCAT_SALES_MV MATERIALIZED VIEW
TEST      DEPT                 TABLE

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=2 Bytes=72)
   1   0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP' (Cost=3 Card=2 Bytes=72)
   2   1     BITMAP CONVERSION (TO ROWIDS)
   3   2       BITMAP OR
   4   3         BITMAP AND
   5   4           BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'
   6   4           BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OBJECT_TYPE'
   7   3         BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OBJECT_TYPE'
```

则 Oracle 进行的 BITMAP AND 和 BITMAP OR 操作示意如下：

```
OWNER          OBJECT_TYPE        OBJECT_TYPE
TEST      TABLE                MATERIALIZED VIEW
0              1                    0                    0        0        0
1              1                    0                    1        0        1
0              0                    0                    0        0        0
0    AND       0        OR          0              =     0    OR  0    =   0
0              0                    1                    0        1        1
0              0                    0                    0        0        0
0              1                    0                    0        0        0
```

Oracle 通过 AND 和 OR 运算，得到结果在所有行中位置的编码，通过映射到 rowid 后，就可

以从表中读取相应的结果了。

27.3.6 Bitmap 索引对批量 DML 操作只需要索引一次

Bitmap 索引对批量操作进行了优化，它会在这一批操作后，统一对数据进行一次索引。对于 Bitmap 索引来说，索引一条记录和索引 1000 条记录所需要的时间相差不多，也就是说维护 Bitmap 索引所花费的时间主要取决与 DML 语句操作的次数，而与 DML 语句每次修改的记录数无关。对于批量插入和更新的系统来说，Bitmap 索引的效率要明显高于 B 树索引。

由于 Bitmap 的索引结构，插入、删除或修改一条记录都有可能导致原来索引的重构。因此 Bitmap 索引不适用于频繁变化的列，如果需要对 Bitmap 索引列进行修改，尽量采用批量的方式，批量方式无论从效率角度还是从占用空间角度都要大大地优于非批量方式。

27.3.7 Bitmap 索引的锁机制

Bitmap 索引的锁机制和 B 树索引不同。B 树索引中包含所有键值非空记录的 rowid，而 Bitmap 索引包含的是一个 rowid 范围和范围上面的编码。如果删除了一条记录，那么不仅仅锁住这一条记录，而是锁定 Bitmap 索引中与这条记录处于同一个索引范围中的所有记录。

```
SQL> SELECT OBJECT_ID FROM TEST_BITMAP WHERE OBJECT_TYPE = 'MATERIALIZED VIEW';
 OBJECT_ID
----------
    30160
    30163

SQL> DELETE TEST_BITMAP WHERE OBJECT_ID = 30160;
已删除 1 行。
```

在另一个 session 中执行下面语句时发现被锁住。

```
SQL> DELETE TEST_BITMAP WHERE OBJECT_ID = 30163;
```

由于 Bitmap 索引是在数据导入之后创建的，因此相同键值的记录应该处于一个范围之中。在一个 session 中删掉一条 OBJECT_TYPE 为物化视图的记录。在另一个 session 中去删除另外一条 OBJECT_TYPE 为物化视图的记录，结果发现 session 被锁住了。对于 delete 操作，被删记录的键值所处的范围被锁定；对于 insert 操作，增加记录的键值所处的范围被锁定；对于更新 Bitmap 索引列的 update 操作，新旧键值所处范围均被锁定。对于不包含非 Bitmap 索引的表，insert 操作的锁不会对其他 session 的 DML 操作造成影响，因为由于 insert 操作如果没有提交，其他 session 是访问不到的。但是如果建立了 Bitmap 索引，不仅 insert 操作可能锁定其他 session 的 update 和 delete 操作，甚至可能锁定其他 session 的 insert 操作。

由此可见，建立了 Bitmap 索引的表，锁的最小粒度变为的 Bitmap 索引的范围。这意味着对于多个用户并行访问时，表被锁定的几率大大增加。因此在对 Bitmap 索引操作时，应该采用批量修改、迅速提交的方式。

27.4 Bitmap 索引的适用范围

Bitmap 索引适合于包含键值数量有限，每个键值重复记录较多的列。Bitmap 索引对于采用批

量 DML 操作或无 DML 操作的列具有很高的效率。Bitmap 索引适合多个限定条件通过 AND 或 OR 连接。

根据上述特点，不难看出，Bitmap 索引更适用于数据仓库和决策支持系统，而对于 OLTP 系统，Bitmap 索引并不是首选。

数据仓库和决策支持系统绝大多数操作是复杂的查询操作，一般都会包含较多的查询条件。导入数据的操作也是采用批量导入的方式，几乎没有 update 和 delete 操作，这些正是使用 Bitmap 索引的优势所在。

反观 OLTP 系统，存在大量的并发事务，频繁插入、修改和删除记录。而这些操作都是 Bitmap 索引所应避免的操作。在 OLTP 系统中大量使用 Bitmap 索引可能导致记录被锁的机会大大增加，严重影响事务的并发性。

但是，并不是说 OLTP 系统不能使用 Bitmap 索引，对于那些数据很少发生变化的表，如果包含了大量重复键值，也可以建立 Bitmap 索引。根据 TOM 的观点，建立 Bitmap 索引的原则是根据实际情况去测试。

在建立 Bitmap 索引后要注意三点，第一点是批量操作，减少维护 Bitmap 索引的代价。第二点是在操作结束后马上提交，减少对表的锁定，同时应经常注意 v$lock 视图，检查包含 Bitmap 索引的表是否经常被锁。最后一点是注意收集统计信息，只有使优化器得到正确的信息，它才能做出正确的判断。

27.5　Bitmap 索引的使用限制

Bitmap 索引和普通 B 树索引的使用和维护上极为相似，只有以下几点额外限制：

- Bitmap 索引不能被基于规则的优化器所使用。这一点已经在上文中提到了。
- Bitmap 索引不能增加 UNIQUE 限制。

这一点其实也不算什么限制，因为 Bitmap 索引本身就不适用于惟一列或接近惟一的列。在这种列上建立 Bitmap 索引，不但效率远远比不上 B 树索引，还将丢掉 Bitmap 索引几乎所有的优点。

- 对于分区表，Bitmap 索引只能采用 Local 索引，不能使用 Global 索引。

27.6　Bitmap Join 索引简介

Bitmap Join 索引不是建立在一张表上，建立在两个或多个表的连接上。

这种索引可以在查询的时候消除对表的连接操作，而通过访问索引得到相应的结果。通过物化视图也可以实现这种功能，但是使用 Bitmap Join 索引比使用物化视图节省大量的空间。

下面通过一个例子来展开说明。

```
SQL> CREATE TABLE TEST_BITMAP_JOIN (ID NUMBER, F_ID NUMBER, COMMENTS VARCHAR2(200));
表已创建。

SQL> INSERT INTO TEST_BITMAP_JOIN SELECT ROWNUM, MOD(ID, 1000) + 1, NULL FROM TEST_BITMAP;
已创建 32214 行。

SQL> COMMIT;
```

```
提交完成。

SQL> ALTER TABLE TEST_BITMAP_JOIN ADD CONSTRAINT FK_TEST_BITMAP_JOIN_F_ID
  2  FOREIGN KEY (F_ID) REFERENCES TEST_BITMAP(ID);
表已更改。
```

建立 Bitmap Join 索引要求相连接的两张表存在主从关系。

由于输出结果较多，使用 SET AUTOTRACE TRACEONLY EXPLAIN 命令只查看查询的执行计划。

```
SQL> SET AUTOTRACE TRACEONLY EXPLAIN
SQL> SELECT A.* FROM TEST_BITMAP_JOIN A, TEST_BITMAP B WHERE A.F_ID = B.ID
  2  AND B.OWNER = 'YANGTK';

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=14 Card=902 Bytes=123574)
   1    0   HASH JOIN (Cost=14 Card=902 Bytes=123574)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP' (Cost=5 Card=28 Bytes=252)
   3    2       BITMAP CONVERSION (TO ROWIDS)
   4    3         BITMAP INDEX (SINGLE VALUE) OF 'IND_B_OWNER'
   5    1     TABLE ACCESS (FULL) OF 'TEST_BITMAP_JOIN' (Cost=7 Card=32223 Bytes=4124544)
```

如上面的执行计划所示，在没有建立 Bitmap 连接索引时，Oracle 必须把两个表 Join，才能得到查询结果。下面看看建立 Bitmap 连接索引后的情况。

```
SQL> CREATE BITMAP INDEX IND_B_TEST_BITMAP_OWNER
  2  ON TEST_BITMAP_JOIN(TEST_BITMAP.OWNER)
  3  FROM TEST_BITMAP_JOIN, TEST_BITMAP
  4  WHERE TEST_BITMAP_JOIN.F_ID = TEST_BITMAP.ID;
索引已创建。
```

建立 Bitmap 连接索引的语法和建立其他索引的有些不同。首先指明建立 Bitmap 索引，并指出是在一张表上建立另一张表中的列的索引，接着是指明表连接的语法。这里需要注意的是，如果在后面的查询语句中使用了匿名，那么在指定索引列时，也必须使用相同的匿名，使用表名会报错。例如：

```
SQL> CREATE BITMAP INDEX IND_B_TEST_BITMAP_OBJECT_TYPE
  2  ON TEST_BITMAP_JOIN(B.OBJECT_TYPE)
  3  FROM TEST_BITMAP_JOIN A, TEST_BITMAP B
  4  WHERE A.F_ID = B.ID;
索引已创建。

SQL> SELECT A.* FROM TEST_BITMAP_JOIN A, TEST_BITMAP B WHERE A.F_ID = B.ID
  2  AND B.OWNER = 'YANGTK';

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=9 Bytes=1110)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'TEST_BITMAP_JOIN' (Cost=3 Card=9 Bytes=1110)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP INDEX (SINGLE VALUE) OF 'IND_B_TEST_BITMAP_OWNER'
```

观察执行计划发现，通过访问 Bitmap Join 索引就完成了查询操作，避免了两张表的连接操作。

由于索引针对的表不同，这种 Bitmap Join 索引可以对已经索引过的列再次索引。像上面的例

子中，OWNER 和 OBJECT_TYPE 列在表 TEST_BITMAP 中均建立过单列索引，但仍然可以在 TEST_BITMAP_JOIN 表上建立 Bitmap Join 索引。

Bitmap Join 索引有 4 种模型，下面简要介绍一下。

F 代表事实表，D 代表维度表，pk 代表维度表的主键列，fk 代表事实表连接到维度表的外键列。事实上，表和维度表的定义，已经超出了本章的讨论范围，感兴趣的朋友可以参考 Orace Data Warehousing Guide。

第一种模型如图 27-1 所示。

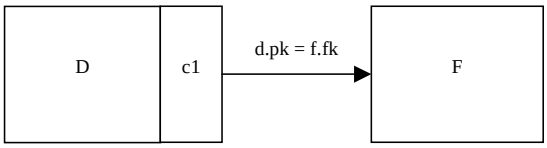

图 27-1　模型一

```
CREATE BITMAP INDEX bji ON f (d.c1) FROM f, d WHERE d.pk = f.fk;
```

例子中使用的就是这种模型。

第二种模型如图 27-2 所示。

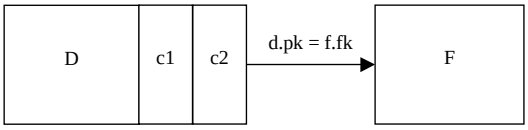

图 27-2　模型二

```
CREATE BITMAP INDEX bji ON f (d.c1, d.c2) FROM F, d WHERE d.pk = f.fk;
```

第三种模型如图 27-3 所示。

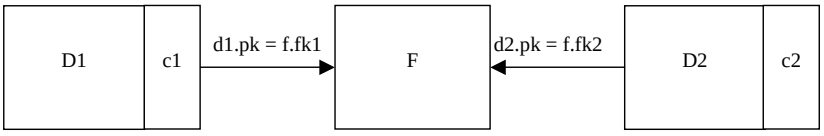

图 27-3　模型三

```
CREATE BITMAP INDEX bji ON f (d1.c1, d2.c2)
FROM f, d1, d2
WHERE d1.pk = f.fk1 AND d2.pk = f.fk2
```

第四种模型如图 27-4 所示。

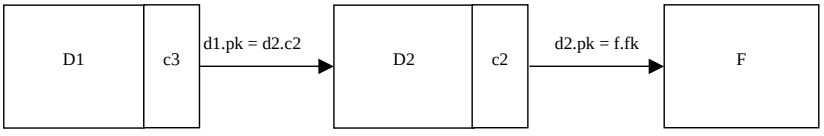

图 27-4　模型四

```
CREATE BITMAP INDEX bji ON f (d1.c3)
FROM f, d1, d2
WHERE d1.pk = d2.c2 AND d2.pk = f.fk;
```

Bitmap Join 索引的一些约束：

(1) 只有事实表支持并行 DML 操作，维度表的并行 DML 操作将导致索引不可用。

(2) 对于使用了 Bitmap Join 索引的表，不同的事务只能同时更新其中的一张表。

(3) 在表连接中，每个表只能出现一次。

(4) 不能在索引组织表和临时表上建立 Bitmap Join 索引。

(5) 被索引的列必须是维度表的列。

(6) 维度表参与连接的列必须是主键列或者具有惟一约束。

(7) 如果维度表使用复合主键，则主键中的每一列都必须在连接中出现。

参考信息

1．Oracle 9i Database Concepts
http://tahiti.oracle.com

2．Oracle 9i Data Warehousing Guide
http://tahiti.oracle.com

3．Oracle 9i Database Performance Tuning Guide and Reference
http://tahiti.oracle.com

4．http://www.itpub.net/114023.html

作者简介

杨廷琨，由于偶然的机会，接触了 Oracle，并对 Oracle 产生了较大的兴趣，自毕业以来一直从事 Oracle 开发和管理工作。2001 年 12 月加入 ITPUB，很喜欢这里的氛围，经常出入 Oracle 管理和 Oracle 开发版块，目前任 Oracle 数据库管理版版主。

任职于某电子商务公司。对 Oracle 数据库的安装、优化和 SQL 的调优积累了一定的经验。希望能和大家一起学习探讨 Oracle 技术。

E-mail：yangtingkun@itpub.net

28 第28章　翻页SQL优化实例

最近碰上一个需求需要把数据库从 MySQL 迁移到 Oracle，期间做了很多工作来优化 Oracle 平台的性能，不过这里面最大的性能调整还是来自 SQL。下面举一个翻页 SQL 调整的例子。此处的表名、索引、列名都不会采取真实名称。

28.1　系统环境

本实例的系统环境如下。

- Linux Version：2.4.20-8custom (gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)) #3 SMP
- MEM：2113466368
- SWAP：4194881536
- CPU：两个超线程的 Intel(R) Xeon(TM) CPU 2.40 GHz

28.2　优化效果

优化前语句在 MySQL 里面要查询 15 秒左右，转移到 Oracle 后进行在不调整索引和语句的情况下执行时间大概是 4～5 秒，调整后执行时间小于 0.5 秒。

翻页语句如下：

```
SELECT *
  FROM (SELECT t1.*, ROWNUM AS linenum
        FROM (SELECT   /*+ INDEX(A IND_OLD)*/
                     a.A
                FROM test a
               WHERE a.A = ' 170101 '
                 AND a.C = '0'
                 AND a.B > SYSDATE
                 AND (a.D >= 0)
            ORDER BY a.B) t1
       WHERE ROWNUM < 18681)
```

```
 WHERE linenum >= 18641;
```

被查询的表 test，其表结构如下：

```
SQL> desc test;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 A                                         NOT NULL   VARCHAR2(11)
 B                                         NOT NULL    DATE
 C                                                     CHAR(2)
 D                                         NOT NULL     NUMBER(1)
 C1                                           NOT NULL VARCHAR2(32)
 C2                                                    VARCHAR2(32)
 C3                                                    CLOB
 C4                                           NOT NULL DATE
 C5                                           NOT NULL DATE
 C6                                                    CLOB
 C7                                                    CLOB
 C8                                                    NUMBER
 C9                                                   NUMBER
 C10                                                   NUMBER
 C11                                                   CHAR(1)
 C12                                                  VARCHAR2(7)
 C13                                       NOT NULL  NUMBER
 C14                                                  VARCHAR2(30)
 C15                                                  VARCHAR2(20)
 C16                                                  VARCHAR2(40)
 C17                                                  VARCHAR2(6)
 C18                                                  CHAR(1)
 C19                                                  CLOB
 C20                                                  CHAR(1)
 C21                                                 NUMBER
 C22                                                  CHAR(1)
 C23                                                 NUMBER(11)
 C24                                                 CLOB
 C25                                       NOT NULL  NUMBER(1)
 C26                                       NOT NULL  NUMBER(1)
 C27                                      NOT NULL  NUMBER(1)
 C28                                       NOT NULL DATE
 C29                                                  VARCHAR2(10)
 C30                                       NOT NULL NUMBER(1)
 C31                                                  CHAR(1)
 C32                                        NOT NULL NUMBER(4)
 C33                                       NOT NULL NUMBER(1)
 C34                                                  VARCHAR2(16)
 C35                                                  NUMBER
 C46                                                  NUMBER
```

表记录数及大小：

```
SQL> select count(*) from test;

  COUNT(*)
----------
    537351
```

记录数为 537351。

```
SQL>
SELECT segment_name, BYTES, blocks
  FROM user_segments
 WHERE segment_name = 'TEST';

SEGMENT_NAME         BYTES         BLOCKS
-------------------- ----------    ----------
TEST                1059061760    129280
```

表大小为 1059061760 字节。

表上原有的索引如下：

```
CREATE INDEX ind_old ON
test(C,D,A,B)
TABLESPACE tbsindex COMPRESS 2;
```

Ind_old 是建立在 C、D、A、B 等 4 个列上的联合索引，压缩度为 2，存放表空间为 tbsindex。

```
SQL>
SELECT segment_name, BYTES, blocks
  FROM user_segments
 WHERE segment_name = 'IND_OLD';

SEGMENT_NAME         BYTES      BLOCKS
-------------------- ---------- ----------
IND_OLD              20971520   2560
```

Ind_old 索引的占用空间为 20971520 字节。

表和索引都已经分析过，下面来看一下 SQL 执行的开销：

```
SQL> set autotrace trace;
SQL>
 SELECT *
  FROM (SELECT t1.*, ROWNUM AS linenum
         FROM (SELECT   a.*
                  FROM test a
                 WHERE a.A LIKE '18%'
                   AND a.C = '0'
                   AND a.B > SYSDATE
                   AND (a.D >= 0)
              ORDER BY a.B) t1
        WHERE ROWNUM < 18681)
 WHERE linenum >= 18641;


40 rows selected.

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=19152 Card=18347 Byt
         es=190698718)

   1    0  VIEW (Cost=19152 Card=18347 Bytes=190698718)
   2    1    COUNT (STOPKEY)
   3    2      VIEW (Cost=19152 Card=18347 Bytes=190460207)
```

```
   4    3        TABLE ACCESS (BY INDEX ROWID) OF 'TEST'
          (Cost=19152 Card=18347 Bytes=20860539)

   5    4          INDEX (RANGE SCAN) OF 'IND_OLD' (NON-UNIQUE) (Cost
          =810 Card=186003)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
     19437  consistent gets
     18262  physical reads
         0  redo size
    114300  bytes sent via SQL*Net to client
     56356  bytes received via SQL*Net from client
       435  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        40  rows processed
```

来看看这条语句的执行计划和执行成本。首先看到最先被执行到的是步骤 5，这里是进行了 ind_old 的 range scan 操作，根据查询条件找出索引中的符合条件的 rowid，返回到 test 表中找到这些 rowid 对应的记录。接下来是一个 view 的操作，形成内部视图，然后通过一个 rownum stopkey 操作取出 rownum 小于 18681 的记录，再把这些结果集转化成一个内部视图，最后根据 linenum >= 18641 取到想要的 40 条记录。这条语句执行的时候产生了 18502 个 consistent gets 以及 17901 个 physical reads，这是比较大的一个消耗。再来看看可以从哪些地方入手改进这条 SQL 呢?主要从两个地方入手改进这条语句，首先是索引，其次是最里面的内部视图 SQL 的改进。

来看一下这个索引建的到底合不合理，先看下各个查寻列的 distinct 值：

```
SQL>  SELECT COUNT (DISTINCT B) FROM test;

COUNT(DISTINCT B)
-------------------
           338965
```

B 列有 338965 个非重复值。

```
SQL>  SELECT COUNT (DISTINCT A) FROM test;

COUNT(DISTINCT A)
-----------------------
                 1148
```

A 列有 1148 个非重复值。

```
SQL>  SELECT COUNT (DISTINCT C) FROM test;

COUNT(DISTINCT C)
--------------------
                 2
```

C 列有 2 个非重复值。

```
SQL>  SELECT COUNT (DISTINCT D) FROM test;

COUNT(DISTINCT D)
```

```
----------------------------
                         5
```

D 列有 5 个非重复值。

页索引里列平均存储长度如下：

```
SQL> select avg(vsize(B)) from test;

AVG(VSIZE(B))
---------------
              7
```

B 列的平均长度为 7 个字节，因为 date 类型的默认长度就是 7 字节。

```
SQL> select avg(vsize(C)) from test;

AVG(VSIZE(C))
-----------------
                2
```

C 列的平均长度为 2 字节。

```
SQL> select avg(vsize(A)) from test;

AVG(VSIZE(A))
-------------------
        5.52313106
```

A 列的平均长度为 5.52313106 字节。

```
SQL> select avg(vsize(D)) from test;

AVG(VSIZE(D))
-------------------------
             1.67639401
```

D 列的平均长度为 1.67639401 字节。

根据表 28-1 可以估算各种组合索引的大小，可以看到 C、D、A 都是相对较低集势的列（重复值较多），下面来大概计算一下各种页索引需要的空间：

表 **28-1**　　**A**、**B**、**C** 和 **D** 的比较

列　名	惟 一 值	列 长 度
B	338965	7
A	1148	5.5
C	2	2
D	5	1.7

```
index1: (B,C,A,D) compress 2
B:distinct number---338965
C: distinct number---2
index size=338965*2*(7+2)+ 537351*(1.7+5.5+6)=14603998

index2: (C,A,B,D) compress 2
C: distinct number---2
A: distinct number---1148
index size=2*1148*(2+5.5)+537351*(7+1.7+6)=7916279
```

```
index3: (C,D,A,B) compress 2
C: distinct number---2
D: distinct number—5
index size=2*5*(2+1.7)+537351*(7+5.5+6)=9941030
```

结果出来了，index2：(C,A,B,D)的索引最小。

注 意

如果知道索引结构的话，关于这个计算索引大小的公式就出来了，这只是一个估算的大小以做参考，索引的空间大小还可以通过 user_segments 等视图进行查询，compress 2 的时候页索引大概的 size=前导列 1 数*前导列 2 数*（前导列 1 位长+前导列 2 位长）+所有记录数*（第 3 列位长+第 4 列位长+6）+overhead，这里的 6 是 rowid 在索引里面的存储位长。

```
CREATE INDEX IDX_AUCTION_BROWSE ON
test(C,A,B,D)
TABLESPACE tbsindex COMPRESS 2;
```

IDX_AUCTION_BROWSE 是建立在 C、A、B、D 等 4 个列上的联合索引，压缩度为 2，存放表空间为 tbsindex。

再来看一下语句，调整第 2 个可以调整的地方——最里面的内部视图。

```
SELECT *
  FROM (SELECT t1.*, ROWNUM AS linenum
         FROM (SELECT   a.*
                 FROM test a
                WHERE a.A LIKE '18%'
                  AND a.C = '0'
                  AND a.B > SYSDATE
                  AND (a.D >= 0)
               ORDER BY a.B) t1
        WHERE ROWNUM < 18681)
 WHERE linenum >= 18641;
```

可以看出这个 SQL 语句有很大的优化余地，首先来看最里面的结果集：

```
SELECT   a.*
    FROM test a
   WHERE a.A LIKE '18%'
     AND a.C = '0'
     AND a.B > SYSDATE
     AND (a.D >= 0)
ORDER BY a.B
```

这里的话会走 index range scan，然后 table scan by rowid，这样的话，如果符合条件的记录数多的话就会相当耗资源，可以改写成：

```
SELECT   a.ROWID
    FROM test a
   WHERE a.A LIKE '18%'
     AND a.C = '0'
     AND a.B > SYSDATE
     AND (a.D >= 0)
ORDER BY a.B
```

这样的话因为只需要 index fast full scan 就可以完成了，因为 rowid 从索引中就可以得到，再改写一下得出以下语句：

```
SELECT *
  FROM test
 WHERE ROWID IN (
        SELECT rid
          FROM (SELECT t1.ROWID rid, ROWNUM AS linenum
                FROM (SELECT   a.ROWID
                         FROM test a
                        WHERE a.A LIKE '18%'
                          AND a.C = '0'
                          AND a.B > SYSDATE
                          AND (a.D >= 0)
                      ORDER BY a.B) t1
               WHERE ROWNUM < 18681)
         WHERE linenum >= 18641);
```

下面来测试一下这个索引的查询开销：

```
SELECT *
  FROM test
 WHERE ROWID IN (
        SELECT rid
          FROM (SELECT t1.ROWID rid, ROWNUM AS linenum
                FROM (SELECT   a.ROWID
                         FROM test a
                        WHERE a.A LIKE '18%'
                          AND a.C = '0'
                          AND a.B > SYSDATE
                          AND (a.D >= 0)
                      ORDER BY a.C, a.B) t1
               WHERE ROWNUM < 18681)
         WHERE linenum >= 18641)
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=18698 Card=18344 Byt
         es=21224008)

   1   0   NESTED LOOPS (Cost=18698 Card=18344 Bytes=21224008)
   2   1     VIEW (Cost=264 Card=18344 Bytes=366880)
   3   2       SORT (UNIQUE)
   4   3         COUNT (STOPKEY)
   5   4           VIEW (Cost=264 Card=18344 Bytes=128408)
   6   5             SORT (ORDER BY STOPKEY) (Cost=264 Card=18344 Byt
         es=440256)

   7   6               INDEX (FAST FULL SCAN) OF 'IDX_AUCTION_BROWSE'
          (NON-UNIQUE) (Cost=159 Card=18344 Bytes=440256)

   8   1     TABLE ACCESS (BY USER ROWID) OF 'TEST' (Cost
         =1 Card=1 Bytes=1137)

Statistics
----------------------------------------------------------
         0  recursive calls
```

```
          0  db block gets
       2080  consistent gets
       1516  physical reads
          0  redo size
     114840  bytes sent via SQL*Net to client
      56779  bytes received via SQL*Net from client
        438  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         40  rows processed
```

来看看这条语句的执行计划和执行成本，首先看到最先被执行到的是步骤 7，这里是进行了 IDX_AUCTION_BROWSE 的 fast full scan 操作，紧接着是一个 sort（order by stopkey），接下来是一个 view 的操作，形成内部视图，然后通过一个 rownum stopkey 操作取出 rownum 小于 18681 的 rowid，同时进行一个 sort（unique），取出 rownum，再把这些 rowid 和 rownum 转化成一个内部视图，最后根据 linenum >= 18641 取出 40 条结果集与 test 进行 nestloops 关联，最后得出想要的结果。

可以看到 consistent gets 从 19437 降到 2080，physical reads 从 18262 降到 1516，查询时间也从 4 秒左右下降到 0.5 秒，可以来说这次 SQL 调整取得了预期的效果。

继续修改一下该语句，以下语句排序顺序与上面有区别，取得的结果集也是不同的，把它列在这里主要是再说明一下在 sort 时还可以调整哪些地方。

```
SQL>
SELECT *
  FROM test
 WHERE ROWID IN (
        SELECT rid
          FROM (SELECT t1.ROWID rid, ROWNUM AS linenum
                 FROM (SELECT   a.ROWID
                         FROM test a
                        WHERE a.A LIKE '18%'
                          AND a.C = '0'
                          AND a.B > SYSDATE
                          AND a.D >= 0
                     ORDER BY a.C, a.A, a.B) t1
                WHERE ROWNUM < 18600)
         WHERE linenum >= 18560) ;

40 rows selected.

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=17912 Card=17604 Byt
         es=20367828)

   1   0  NESTED LOOPS (Cost=17912 Card=17604 Bytes=20367828)
   2   1    VIEW (Cost=221 Card=17604 Bytes=352080)
   3   2      SORT (UNIQUE)
   4   3        COUNT (STOPKEY)
   5   4          VIEW (Cost=221 Card=17604 Bytes=123228)
   6   5            INDEX (RANGE SCAN) OF 'IDX_AUCTION_BROWSE' (NON-
```

```
          UNIQUE) (Cost=221 Card=17604 Bytes=422496)

   7   1    TABLE ACCESS (BY USER ROWID) OF 'TEST' (Cost
          =1 Card=1 Bytes=1137)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
       550  consistent gets
        14  physical reads
         0  redo size
    117106  bytes sent via SQL*Net to client
     56497  bytes received via SQL*Net from client
       436  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
        40  rows processed
```

在 order by 里加上 A 列，消除了 sort（order by stopkey），因为索引内是根据 C、A、B 进行排序存放的，所以当做 index fast full scan 的时候就不需要再 sort 了，加了这一步骤可以把 consistent gets 从 2080 降到 550。SQL 调整是优化数据库性能最主要的手段之一，希望这篇文章能给大家一个启发，也希望大家更加重视 SQL 优化并从中得到更多的东西。

作者简介

汪海，2001 年毕业于杭州电子工业学院，毕业后在软件公司做数据库相关开发工作。现任职于国内某大型电子商务网站，专职 Oracle DBA。接触并掌握了多种数据库系统，现专注于 Oracle 的研究。对备份恢复、SQL 优化、Internal 都有相当兴趣。

第 29 章　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。

本章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。

29.1　系统环境

OS：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
```

29.2　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。

29.3 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.num
Experience 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.num
Experience 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 占用了大量的排序空间，确定是这些 SQL 引起了临时表空间的过量使用。这些 SQL 需要研究和优化。

29.4　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,
  <省略大量字段>
  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
```

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。

29.5　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,
    <省略大量字段>
  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间隔，本人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
```

可以注意到，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，性能有了大大的提高。

提　示

有效地降低逻辑读是 SQL 优化的基本原则之一。

29.6　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行进一步的优化。

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。

29.7 小结

Oracle 的物化试图功能强大，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，在我的站点上（www.eygle.com）上也有详细描述，文中不再赘述。

作者简介

盖国强，网名 eygle，ITPUB Oracle 管理版版主，ITPUB 论坛超级版主，曾任 ITPUB MS 版主。CSDN eMag Oracle 电子杂志主编。

曾任职于某国家大型企业，服务于烟草行业，开发过基于 Oracle 数据库的大型 ERP 系统，属国家信息产业部重点工程。同时负责 Oracle 数据库管理及优化，并为多家烟草企业提供 Oracle 数据库管理、优化及技术支持。

目前任职于北京某电信增值业务系统提供商企业，首席 DBA，负责数据库业务。管理全国 30 多个数据库系统。项目经验丰富，曾设计规划及支持中国联通增值业务等大型数据库系统。

实践经验丰富，长于数据库诊断、性能调整与 SQL 优化等。对于 Oracle 内部技术具有深入研究。

高级培训讲师，培训经验丰富，曾主讲 ITPUB DBA 培训及 ITPUB 高级性能调整等主要课程。《Oracle 数据库 DBA 专题技术精粹》一书的主编及主要作者。

可以在 http://www.eygle.com 上找到关于作者的更多信息。

第 30 章　如何给 Large Delete 操作提速近千倍

30.1　背景描述

首先介绍一下本章优化实例的背景。

30.1.1　任务描述

这个任务是需要从一系列大表中清理 3 个省的大批过时数据，具体的清理过程简单地说就是根据不同的 miscid 值创建不同的临时表，类似以下：

```
CREATE TABLE temp_mid
AS
SELECT mid FROM ssr WHERE SUBSTR(ssid,1,7) IN
 (SELECT prefixnum FROM prefix WHERE mcid='0012');
```

然后通过这个临时表连接另一个大表，做以下删除工作：

```
DELETE SSF
     WHERE mid IN (SELECT mid
                   FROM TEMP_MID_HUBEI);
```

上述任务根据不同的关键字，需要执行几十次，如果不加任何优化的话，每一次都需要执行几十个小时。由于需求、操作和优化思路大体相同，下面就以上面的例子详细说说实际应用中如何一步步优化并将操作提速到近千倍。

30.1.2　数量级统计和描述

首先统计这个操作涉及到的几张表：

```
SELECT COUNT(*) FROM PREFIX;

SELECT COUNT(*) SSR FROM SSR;

SELECT COUNT(*) SSF FROM SSF;
```

```
SELECT COUNT(*) AS SSF_0012 FROM SSF WHERE MID IN (SELECT MID FROM TEMP_MID_HUBEI);
```

上述脚本的执行过程如下（由于创建临时表 TEMP_MID_HUBEI 的过程比较简单，因此这里没有赘述，仅仅是从建立临时表后的删除操作开始分析的）：

```
SQL> @LUNAR.SQL

PREFIX
----------
    51854

ELAPSED: 00:00:00.14

SSF
-----------
   83446270

ELAPSED: 00:04:53.27

SSR
----------
  43466645

ELAPSED: 00:03:08.00

SSF_0012
----------------
        131098

ELAPSED: 00:00:57.02
```

这里需要做的是，从一个 8300 多万行的大表中，通过和一个 130 多万行的表进行连接并删除其中的大部分数据。整个操作的过程，要求所有的表都可以实时访问，并且除了手工建立的临时表（TEMP_MID_HUBEI）以外，其他的表都可以实时访问和修改。

30.2 背景知识——Bulk Binding

在下面的优化过程中，使用了批量绑定（Bulk Binding）的思想，因此首先对这一概念作些解释。

30.2.1 什么是 Bulk Binding

在 SQL 语句中（动态地）给 PL/SQL 变量赋值叫做绑定（Binding）。一次绑定一个完整的集合叫做批量绑定（Bulk Binding）。

从 Oracle 8i 开始，在 PL/SQL 可以使用两个新的数据操纵语言（DML）语句：BULK COLLECT 和 FORALL。这两个语句在 PL/SQL 内部按数组进行数据处理。

30.2.2　Bulk Binding 的优点是什么

批量绑定（Bulk Binds）通过最小化在 PL/SQL 和 SQL 引擎之间的上下文切换提高了性能，它以一个完整的集合（如 varray、nested tables、index-by table 或 host array）为单位（一批一批地）向前或者向后绑定变量。在 Oracle 8i 以前，每个 SQL 语句的执行需要在 PL/SQL 和 SQL 引擎之前切换上下文，使用绑定变量后，就只需要一次上下文切换。

其中，BULK COLLECT 提供对数据的高速检索，FORALL 可大大改进 INSERT、UPDATE 和 DELETE 操作的性能。

30.2.3　如何进行批量绑定（Bulk Binds）

绑定变量包扩下面两个部分：

- 输入集合：使用 FORALL 语句，一般用来改善 DML（INSERT、UPDATE 和 DELETE）操作的性能。
- 输出集合：使用 BULK COLLECT 子句，一般用来提高查询（SELECT）的性能。

1. 输入集合（FORALL）

输入集合是数据通过 PL/SQL 引擎到 SQL 引擎去执行 INSERT、UPDATE、DELETE 语句。输入集合使用 FORALL 语句，下面是 FORALL 的语法：

```
FORALL index IN lower_bound..upper_bound
   sql_statement;
```

2. FOR LOOP 语句和 FORALL 的比较

例 1：分别使用传统的 FOR…LOOP 操作和本章介绍的 FORALL 操作向 lunartest 表中加载 1000000 条记录，对比一下它们的执行效率。

测试过程如下。

首先创建一个用来记录操作时间的存储过程 get_time：

```
CREATE OR REPLACE PROCEDURE get_time (t OUT NUMBER)
IS
BEGIN
  SELECT TO_CHAR (SYSDATE, 'SSSSS')
    INTO t
    FROM DUAL;
END;
```

然后创建一个空表，分别使用 FOR…LOOP 和 FORALL…LOOP 插入数据，并记录和输出操作时间：

```
SQL> conn lunar/lunar
Connected.
SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE lunartest (pnum NUMBER(20), pname varchar2(50));

Table created.
```

```
Elapsed: 00:00:00.00
SQL> Create Or Replace PROCEDURE BulkTest IS
  2    TYPE NumTab IS TABLE OF NUMBER(20) INDEX BY BINARY_INTEGER;
  3    TYPE NameTab IS TABLE OF varchar2(50) INDEX BY BINARY_INTEGER;
  4    pnums NumTab;
  5    pnames NameTab;
  6    t1 CHAR(5);
  7    t2 CHAR(5);
  8    t3 CHAR(5);
  9  BEGIN
 10    FOR j IN 1..1000000 LOOP  -- load index-by tables
 11      pnums(j) := j;
 12      pnames(j) := 'Part No. ' || TO_CHAR(j);
 13    END LOOP;
 14
 15    get_time(t1);
 16
 17    FOR i IN 1..1000000 LOOP -- use FOR loop
 18      INSERT INTO lunartest VALUES (pnums(i), pnames(i));
 19    END LOOP;
 20
 21    get_time(t2);
 22
 23    FORALL i IN 1..1000000  --use FORALL statement
 24      INSERT INTO lunartest VALUES (pnums(i), pnames(i));
 25    get_time(t3);
 26
 27    DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
 28    DBMS_OUTPUT.PUT_LINE('---------------------');
 29    DBMS_OUTPUT.PUT_LINE('FOR loop: ' || TO_CHAR(t2 - t1));
 30    DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR(t3 - t2));
 31  END;
 32  /

Procedure created.

Elapsed: 00:00:00.00
SQL> exec BulkTest;
Execution Time (secs)
---------------------
FOR loop: 110
FORALL:  54

PL/SQL procedure successfully completed.

Elapsed: 00:03:24.07
SQL> select sum(bytes/1024/1024) MB from user_segments where segment_name='LUNARTEST';

       MB
----------
       57
```

```
Elapsed: 00:00:11.06
```

这里注意到使用 FOR...LOOP 语句插入 1000000 条记录，需要 110 秒；而使用 FORALL 语句只需要 54 秒。

3. 如何处理回滚

对于回滚的处理，FORALL 操作可以自动完成，也就是说，如果一个 FORALL 语句执行失败，那么 Oracle 会基于隐式的 SAVEPOINT 一次回滚 SQL 语句中先前执行的部分。

4. 输出集合

输出集合是数据作为一个通过 SQL 引擎到 PL/SQL 引擎的（SELECT 或者 FETCH）结果集。

输出集合通过在 SELECT INTO、FETCHINTO 和 RETURNING INTO 子句中加入 BULK COLLECT 子句实现，下面是 BULK COLLECT 子句的语法：

```
... BULK COLLECT INTO collection_name[, collection_name] ...
```

5. 在 SELECT INTO 中使用 BULK COLLECT

这里结合一个实例来理解一下在 SELECT INTO 语句中批量绑定的使用：

```
SQL> conn lunar/lunar
Connected.
SQL> SET SERVEROUTPUT ON
SQL> Create Or Replace Procedure lunartest2 Is
  2    TYPE NumTab IS TABLE OF emp.empno%TYPE;
  3    TYPE NameTab IS TABLE OF emp.ename%TYPE;
  4    enums NumTab; -- no need to initialize
  5    names NameTab;
  6  BEGIN
  7    SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;
  8    FOR i in enums.FIRST..enums.LAST LOOP
  9      DBMS_OUTPUT.PUT_LINE(enums(i) || ' ' || names(i));
 10    END LOOP;
 11  END;
 12  /

Procedure created.

Elapsed: 00:00:00.08
```

可以看到，使用 SELECT ...BULK COLLECT INTO 的方法和传统的 SELECT INTO 的语法基本上变化不大。现在看一下执行结果：

```
SQL>  select empno,ename from emp;

    EMPNO ENAME
---------- ----------
     7369 SMITH
     7499 ALLEN
     7521 WARD
     7566 JONES
     7654 MARTIN
```

```
        7698 BLAKE
        7782 CLARK
        7788 SCOTT
        7839 KING
        7844 TURNER
        7876 ADAMS
        7900 JAMES
        7902 FORD
        7934 MILLER

    14 rows selected.

    Elapsed: 00:00:00.00
    SQL> exec lunartest2;
    7369 SMITH
    7499 ALLEN
    7521 WARD
    7566 JONES
    7654 MARTIN
    7698 BLAKE
    7782 CLARK
    7788 SCOTT
    7839 KING
    7844 TURNER
    7876 ADAMS
    7900 JAMES
    7902 FORD
    7934 MILLER

    PL/SQL procedure successfully completed.

    Elapsed: 00:00:00.00
```

如果这里需要输出的记录不是 14 条，而是 140 万条甚至更多，那么 BULK COLLECT 就会发挥出强悍的优势，这里仅仅是在功能上得到了验证。

6. 在 FETCH INTO 中使用 BULK COLLECT

现在来看看在 FETCH INTO 语句中如何使用批量绑定：

```
SQL> conn lunar/lunar
Connected.
SQL> SET SERVEROUTPUT ON
SQL> Create Or Replace Procedure lunartest3 Is
  2    TYPE NameTab IS TABLE OF emp.ename%TYPE;
  3    TYPE SalTab IS TABLE OF emp.sal%TYPE;
  4    names NameTab;
  5    sals SalTab;
  6    CURSOR c1 IS SELECT ename, sal FROM emp;
  7  BEGIN
  8    OPEN c1;
  9    FETCH c1 BULK COLLECT INTO names, sals;
 10    FOR i IN names.FIRST..names.LAST LOOP
```

```
 11       DBMS_OUTPUT.PUT_LINE(names(i) || ' ' || sals(i));
 12    END LOOP;
 13    CLOSE c1;
 14  END;
 15  /

Procedure created.

Elapsed: 00:00:00.01
SQL> select ename,sal from emp;

ENAME           SAL
---------- ----------
SMITH           800
ALLEN          1600
WARD           1250
JONES          2975
MARTIN         1250
BLAKE          2850
CLARK          2450
SCOTT          3000
KING           5000
TURNER         1500
ADAMS          1100
JAMES           950
FORD           3000
MILLER         1300

14 rows selected.

Elapsed: 00:00:00.00
SQL> exec lunartest3;
SMITH 800
ALLEN 1600
WARD 1250
JONES 2975
MARTIN 1250
BLAKE 2850
CLARK 2450
SCOTT 3000
KING 5000
TURNER 1500
ADAMS 1100
JAMES 950
FORD 3000
MILLER 1300

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.00
```

这里有一点需要特别注意，bulk-tech 不能从游标（cursor）插入到一个记录的集合。

7. 使用新的游标（cursor）属性实现 BULK COLLECT

从 Oracle 9i 开始，Oracle 又提供了新的 Bulk Binds 游标属性，即%BULK_ROWCOUNT，其语法为：

```
IF SQL%BULK_ROWCOUNT(i) = ... THEN
……
ENDIF;
```

该游标属性的使用方法和含义与传统的 SQL% ROWCOUNT 基本相同。

下面结合一个实例，来介绍 FORALL 的强大作用和使用的注意事项。

30.3 优化过程详解

30.3.1 第一次优化——处理庞大的 IN - LIST 操作

回到最初的需求，首先看看这个删除工作的基本语句，请注意这个 IN 操作，由于这个 TEMP_MID_HUBEI 表有 130 多万行记录，这将是一个多么庞大的 IN-LIST。

```
DELETE from SSF
     WHERE mid IN (SELECT mid
                   FROM TEMP_MID_HUBEI);
```

下面，在测试环境下看看，系统机器空闲，资源极大丰富，数量比真实环境小很多的情况下，这条删除语句的执行效率：

```
SQL> DELETE from SSF WHERE mid IN (SELECT mid FROM TEMP_MID_HUBEI);

18 rows deleted.

Elapsed: 00:05:35.79
SQL>rollback;
```

可以看到原始的删除语句需要大约 5 分半钟。现在，使用 RBO（加上 RULE HINT），再观察一下语句的执行情况：

```
SQL> DELETE /*+ RULE */ from SSF WHERE mid IN (SELECT mid FROM TEMP_MID_HUBEI);

18 rows deleted.

Elapsed: 00:00:47.27
SQL>rollback;
```

可以看到，这条语句的执行时间缩短到只需要 47 秒钟，单条语句的执行效率提高了 7 倍以上，这也就是常说的，对付 IN-LIST 或者大量 OR 条件的一大有利武器。

好了，现在如果在生产库上直接运行修改后的 DELETE 语句来删除数据性能会提高多少呢？会是我们看到的 7 倍么？

删除操作所涉及到的所有表的操作写成一个脚本，首先使用少量行删除的操作做测试，得到如下的测试结果：

```
oracle@db02:/oracle/lunar > cat nohup.out

SQL*Plus: Release 9.2.0.4.0 - Production on Mon Mar 7 12:37:56 2005
```

```
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 and Real Application Clusters options
JServer Release 9.2.0.4.0 - Production


37 rows deleted.                henan_SUBSO

Elapsed: 00:06:39.81

Commit complete.

Elapsed: 00:00:00.01

4 rows deleted.                 hubei_SUB

Elapsed: 00:02:42.71

Commit complete.

Elapsed: 00:00:00.01

1 row deleted.                  hubei_SUBSC

Elapsed: 01:31:51.25

Commit complete.

Elapsed: 00:00:00.01

18727 rows deleted.             hubei_SUBNFO

Elapsed: 00:01:35.55

Commit complete.

Elapsed: 00:00:00.01

21 rows deleted.                fujian_SUINFO

Elapsed: 00:00:15.49

Commit complete.

Elapsed: 00:00:00.01

35423 rows deleted.             GINFO
```

```
    Elapsed: 00:02:19.80

    Commit complete.

    Elapsed: 00:00:00.04
    Disconnected from Oracle9i Enterprise Edition Release 9.2.0.4.0 - 64bit Production
    With the Partitioning and Real Application Clusters options
    JServer Release 9.2.0.4.0 - Production
    oracle@db02:/oracle/lunar >
```

可以看到上面可喜的结果，于是决定进行第二次优化——分段处理，也就是将所有超过 10000 条以上的删除操作拆分成小段操作（比如，以 10000 条记录为标准删除记录，然后提交）。

30.3.2 第二次优化——分段操作

这次优化的思想仅仅是通过 rownum 将完整的操作分成若干段，设定每次（每段）只操作指定数量的行，删除完成后立即提交。

该过程如下：

```
    CREATE OR REPLACE PROCEDURE del_hubei_ssf (
      p_count   IN   VARCHAR2           -- Commit after delete How many records
    )
    AS
      PRAGMA AUTONOMOUS_TRANSACTION;
      sql_stat   VARCHAR2 (1000) := '';
      n_delete   NUMBER        := 0;
    BEGIN
      /** 3. delete data from the hubei  SSF **/
      DBMS_OUTPUT.put_line ('3. Start delete from the hubei  SSF!!!');

      WHILE 1 = 1
      LOOP
        EXECUTE IMMEDIATE 'DELETE /*+ RULE */ from SSF WHERE mid IN (SELECT mid FROM temp_mid_hubei)
and rownum<=:rn'
                  USING p_count;

        IF SQL%NOTFOUND
        THEN
          EXIT;
        ELSE
          n_delete := n_delete + SQL%ROWCOUNT;
        END IF;

        COMMIT;
        DBMS_OUTPUT.put_line (sql_stat);
        DBMS_OUTPUT.put_line (TO_CHAR (n_delete) || ' records deleted ...');
      END LOOP;

      COMMIT;
      DBMS_OUTPUT.put_line ('Full Finished!!!');
      DBMS_OUTPUT.put_line (   'Totally '
```

```
                    || TO_CHAR (n_delete)
                    || ' records deleted from hubei_SSF  !!!'
                   );
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (SQLERRM);
END;
/
```

根据上面小表的测试结果，删除 10000 行的操作应该在几分钟之内完成，那么删除百万行的记录，应该在 20 个小时左右应该可以有结果了，于是决定再放一个专门利用上面的存储过程来进行大量删除的脚本，下班前放到后台跑，准备第二天来上班时间来拿结果。

次日午后，我检查 nohup.out，奇怪，居然还没有完成的信息，看来上面的问题有了答案，对于越大的表和越大的结果集来说，随着操作记录的成倍增加，操作时间将以一定的倍数增加，所以仅仅这样优化单个大表操作的语句是不能解决问题的。

于是有了第三个优化思路，拆分 DELETE 操作，将整个 DELETE 的操作拆分成原子级。

30.3.3　第三次优化——拆分 DELETE 操作

首先，尝试从临时表 temp_mid_hubei 中取出一行记录：

```
SQL> select mid ,rowid from  temp_mid_hubei where rownum=1;

MID          ROWID
-------------- ------------------
00046000019808 AAAngMAGkAAA5SkAAA

Elapsed: 00:00:00.01
```

然后，再根据条件在 SSF 表中删除相应的记录，看看需要多长时间：

```
SQL> DELETE from SSF WHERE mid='00046000019808';

1 rows deleted.

Elapsed: 00:00:00.01
```

请注意，这样的操作执行几乎瞬间就可以完成，于是开始拆分 DELETE 的思路就应运而生：

(1) 首先取出临时表中的一行（包括 ROWID）。

(2) 根据关键字在大表 SSF 中删除一行数据（使用惟一索引）。

(3) 根据最初记录的 ROWID 将这条记录从临时表中删除（以避免重复比较）。

修改了存储过程，如下：

```
CREATE OR REPLACE PROCEDURE DEL_HUBEI_SSF
AS
   v_i    NUMBER (10);
   v_mid   VARCHAR2 (30) := '';
   v_row   ROWID;
BEGIN
-- v_i:=10000;
```

```
    SELECT COUNT (*)
      INTO v_i
      FROM temp_mid_hubei_bak;

    WHILE v_i > 0
    LOOP
       SELECT mid, ROWID
         INTO v_mid, v_row
         FROM temp_mid_hubei_bak
        WHERE ROWNUM = 1;

       DELETE FROM SSF
            WHERE mid = v_mid;

       DELETE FROM temp_mid_hubei_bak
            WHERE ROWID = v_row;

       COMMIT;
       v_i := v_i - 1;
    END LOOP;

    DBMS_OUTPUT.put_line ('Full Finished!!!');
    DBMS_OUTPUT.put_line (   'Totally '
                          || TO_CHAR (v_i)
                          || ' records deleted from hubei_SSF !!!'
                         );
END;
```

当然，实际操作的时候，由于这次任务中有几类的操作都需要用到临时表 temp_mid_hubei，因此，在操作之前需要使用 CTAS（create table … as select … ）将该表做个备份。

经过测试，使用这种方法删除 10000 条数据，需要大概 1 分钟时间，于是准备启用这个方法来完成任务。

可是这样的操作也远远不能满足要求，因为删除 10 万行记录，并不意味着需要 10*1 分钟=10 分钟，而是需要大约 30 多分钟 !!

看来，上一个方法的问题（大量操作的时间会成倍增长于少量操作的时间）再一次体现出来，于是，想到了第四次优化——使用 FORALL 处理批量作业。

30.3.4 第四次优化——使用 FORALL 处理批量作业

上面已经介绍了批量作业的一些概念，这里不再赘述。

这次优化的思想主要是使用 FORALL 的思路来改写“ 第三次优化——拆分 DELETE 操作 ”，具体的修改如下：

```
SQL> l
  1  create or replace procedure del_hubei_SSR_forall
  2  as
  3       type ridArray is table of rowid index by binary_integer;
  4       type dtArray is table of varchar2(50) index by binary_integer;
  5
```

```
      6      v_rowid ridArray;
      7      v_mid_to_delete dtArray;
      8
      9  begin
     10      select mid,rowid bulk collect into v_mid_to_delete,v_rowid from temp_mid_hubei where
rownum<11;
     11
     12     forall i in 1 .. v_mid_to_delete.COUNT
     13         delete from SSR where mid = v_mid_to_delete(i);
     14
     15           DBMS_OUTPUT.PUT_LINE('Full Finished!!!');
     16            DBMS_OUTPUT.PUT_LINE('Totally '||to_char(v_mid_to_delete.COUNT)||' records
deleted from hubei_SSR   !!!');
     17
     18      forall i in 1 .. v_rowid.COUNT
     19         delete from temp_mid_hubei where rowid = v_rowid(i);
     20
     21           DBMS_OUTPUT.PUT_LINE('Full Finished!!!');
     22            DBMS_OUTPUT.PUT_LINE('Totally '||to_char(v_rowid.COUNT)||' records deleted
from temp_mid_hubei   !!!');
     23
     24* end;
    SQL> /

    Procedure created.

    Elapsed: 00:00:00.07
    SQL>
    SQL>  exec del_hubei_SSR_forall;
    Full Finished!!!
    Totally 10 records deleted from hubei_SSR   !!!
    Full Finished!!!
    Totally 10 records deleted from temp_mid_hubei   !!!

    PL/SQL procedure successfully completed.

    Elapsed: 00:00:00.16
```

这里应用了上面介绍过的 SELECT BULK COLLECT INTO 方法来进行批量绑定，根据以 10 条记录为一组的测试结果表明，速度比较理想，于是再测 1000 条记录的情况：

```
    SQL> exec del_hubei_SSR_forall;
    Full Finished!!!
    Totally 1000 records deleted from hubei_SSR   !!!
    Full Finished!!!
    Totally 100 records deleted from temp_mid_hubei   !!!

    PL/SQL procedure successfully completed.

    Elapsed: 00:00:00.66
```

测试结果还是比较稳定，于是再测 10000 和 100000 条记录的情况：

```
    SQL> exec del_hubei_SSR_forall;
    Full Finished!!!
```

```
Totally 10000 records deleted from hubei_SSR   !!!
Full Finished!!!
Totally 10000 records deleted from temp_mid_hubei   !!!

PL/SQL procedure successfully completed.

Elapsed: 00:00:04.07
SQL>
SQL> exec del_hubei_SSR_forall;
Full Finished!!!
Totally 100000 records deleted from hubei_SSR   !!!
Full Finished!!!
Totally 100000 records deleted from temp_mid_hubei   !!!

PL/SQL procedure successfully completed.

Elapsed: 00:03:51.29
```

这里注意到，完成删除 10000 记录需要 4 秒钟，而完成 100000 条记录的删除需要将近 4 分钟，也就是说，从删除 10000 条记录到删除 100000 条记录，所需要的操作时间已经出现了比较大的倍数关系。

下面再测试一下 50 万条记录的情况：

```
SQL> exec del_hubei_SSR_forall;
Full Finished!!!
Totally 509555 records deleted from hubei_SSR   !!!
Full Finished!!!
Totally 509555 records deleted from temp_mid_hubei   !!!

PL/SQL procedure successfully completed.

Elapsed: 00:39:04.41
```

这里，删除 50 万条记录已经需要将近 40 分钟了！

到这里，基本上可以得出结论，使用批量删除虽然可以解决一定的问题，但是如果需要操作的数组太大，那么执行结果会大打折扣，这个是不难理解的。

于是，第五次优化的思路就产生了：

(1) 分段处理。

(2) 拆分操作。

(3) 批量绑定。

30.3.5 第五次优化——使用 FORALL＋原子级操作

这一次的优化过程已经很清晰了，主要是综合了前面几次优化的思想，将它们的优点集中使用，即：

- 分段处理：将整个大的事务分割为以 10000 行为单位的小事务，其中，10000 这个基数来自于上面使用批量测试时的结果（从删除 10000 条记录到删除 100000 条记录，所需要的操作时间已经出现了比较大的倍数关系）。
- 拆分操作：避开庞大的 IN-LIST 条件，而是采用原子级的处理。

■　批量绑定：使用 FORALL 进行批量绑定，以数组方式处理，大大减少了 PL/SQL 引擎和 SQL 引擎的上下文切换，从而提高了处理速度。

现在来构造以 10000 为单位的 FORALL 处理过程：

```
create or replace procedure del_hubei_SSF_forall
as
    type ridArray is table of rowid index by binary_integer;
    type dtArray is table of varchar2(50) index by binary_integer;

    v_rowid ridArray;
    v_mid_to_delete dtArray;

begin
    select mid,rowid bulk collect into v_mid_to_delete,v_rowid from temp_mid_hubei_bak where
rownum<10001;

   forall i in 1 .. v_mid_to_delete.COUNT
       delete from SSF where mid = v_mid_to_delete(i);
--          DBMS_OUTPUT.PUT_LINE(to_char(v_mid_to_delete.COUNT)||' records deleted from
hubei_SSF  !!!');

    forall i in 1 .. v_rowid.COUNT
       delete from temp_mid_hubei_bak where rowid = v_rowid(i);
--               DBMS_OUTPUT.PUT_LINE(to_char(v_rowid.COUNT)||'  records  deleted  from
temp_mid_hubei_bak  !!!');

end;
/

……
```

然后构造始终按照 10000 条循环批量删除的过程：

```
create or replace procedure exec_forall
 (
  p_RowCount in number,          -- Total need to delete rows count
  p_ExeCount in number     -- Every times need to delete rows count
)
as
   n_RowCount number:=0;        -- Yet needed to delete rows count
   n_ExeTimes number:=0;    -- execute times(loop times)
   n_delete number:=0;                -- really delete rows count
begin
 n_RowCount := p_RowCount;

 while n_RowCount >0 loop
   EXECUTE IMMEDIATE 'begin del_hubei_SUBSREGINFO_forall; end;';
  commit;

  if n_RowCount>p_ExeCount then
              n_RowCount:= n_RowCount-p_ExeCount;
     n_ExeTimes := n_ExeTimes + 1;
  else
```

```
      n_ExeTimes := n_ExeTimes + 1;
                n_delete := n_RowCount;
                n_RowCount:= n_RowCount-p_ExeCount;
  end if;

   DBMS_OUTPUT.PUT_LINE('---------'||to_char(n_ExeTimes)||'-----------');
   n_delete := n_delete+p_ExeCount*(n_ExeTimes-1);
end loop;

DBMS_OUTPUT.PUT_LINE('Full Finished!!!');
DBMS_OUTPUT.PUT_LINE('Totally '||to_char(n_delete)||' records deleted.  !!!');

end;
/
```

好了，现在来测试一下这次优化的结果，看一下临时表的记录数目：

```
SQL> select count(*) temp_mid_hubei from temp_mid_hubei_bak;

TEMP_MID_HUBEI
--------------
      1444661

Elapsed: 00:00:02.41
```

再检查一下业务大表的记录数目：

```
SQL> SELECT COUNT(*) SSF FROM SSF;

SSF
-----------
  83446270

ELAPSED: 00:04:53.27
```

首先删除 500000 行记录，看看需要多少时间完成：

```
SQL> exec exec_forall(500010,10001);
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------1-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------2-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------3-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------4-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------5-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------6-----------
```

```
……（这里省略掉了一些雷同的输入）

---------50-----------
Full Finished!!!
Totally 500010 records deleted.  !!!

PL/SQL procedure successfully completed.

Elapsed: 00:02:57.78
```

这是个惊人的数字，2 分 57 秒 !! 于是再来删除 90 万试试看：

```
SQL> exec exec_forall(900010,10001);
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------1-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------2-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------3-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------4-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------5-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------6-----------

……

---------86-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------87-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------88-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------89-----------
10000 records deleted from hubei_SSF   !!!
10000 records deleted from temp_mid_hubei_bak   !!!
---------90-----------
Full Finished!!!
Totally 900010 records deleted.  !!!

PL/SQL procedure successfully completed.

Elapsed: 00:02:33.47
```

```
SQL>

SQL> select count(*) temp_mid_hubei from temp_mid_hubei_bak;

TEMP_MID_HUBEI
--------------
        34661

Elapsed: 00:00:00.46
```

删除 90 万的数据只需要 2 分 33 秒 !! 和删除 50 万数据量相比，使用的操作时间差不多。

到此，此次任务已经可以圆满完成了（将所有的操作使用这种优化思想写成脚本，很快就可以完成任务）。

30.4 小结

从最初几乎每次都因为报 ORA-01555 错误而无法完成的一个 DELETE 语句，到删除 100 万记录需要 20 多个小时，再发展到现在的删除 90 万记录只需要不到 3 分钟，速度提高了近千倍，这里主要的思想是：

- 避开繁琐的查询条件（即避开了庞大到上百万个元素的 IN-LIST）。
- 使用原子级的处理方式（使用两个原子级的处理，将原子级的处理保持在毫秒级）。
- 使用 Bulk Binds（FORALL）的特性，批量处理，大大降低了主机的资源压力（以 10000 为单位，批量处理）。

优化是一件很灵活的事情，没有千篇一律的规律，只有灵活地拆分、组合一些优化方法来达到目的，如果一定生搬硬套，那么很可能会把本来简单的事情搞复杂且无法处理了。这里就本章的优化思路和方法提几点注意事项：

(1) 对于有些条件负责的操作，需要做更多的逻辑拆分工作，以便拆分得更小。

(2) 对于 INSERT、UPDATE 的测试速度一般达不到本章中提高近千倍的速度。

(3) 对于 CTAS（create table as select）操作，不要使用这个方法来做，很多情况下，直接做（如果可以的话，最好加 NOLOGGING 选项）更快。

作者简介

张权（第一作者），中国科学院硕士研究生，现任职于某大型国家政府机关，一直从事公安警卫信息化研究，是信息化建设的骨干力量。有将近 7 年的 Oracle 和其他相关工具的开发和实战经验。

张大鹏（第二作者），网名 Lunar2000（Lunar），ITPUB 资深会员。现任职于某大型外资企业，服务于电信增值业务，从事专职 DBA 工作。主要负责 Oracle 数据库日常管理，包括备份和恢复，性能优化，故障诊断等。实践经验丰富，长于数据库故障诊断、性能优化和备份恢复。

第 31 章　Web 分页与优化技术

在现在 B/S 程序一统天下的时代，其实在大部分网站系统中，高速搜索与分页技术已经是搜索引擎的天下，如 Google、百度等，都是拥有着核心的搜索技术与大量的搜索服务器。包括本人现在所在的淘宝，商品搜索都是由搜索引擎服务器完成。但是，这不代表数据库直接的搜索与分页技术将退出历史舞台，其实，在很多地方，如对速度要求不高，数据量并不大的网页上，数据库的搜索与分页还是非常普遍的。包括 Oracle 比较有名的网站 ask tom 都还是采用的数据库搜索与分页技术完成的。

本章将讨论的就是直接从数据库中获取数据并分页，所以下文中的 Web 分页技术都是指从数据库中查询并分页显示的技术。

31.1　什么是 Web 分页

Web 分页技术就是利用特定的分页 SQL 语句从数据库中获得特定的数据，并展示在 Web 页面上，所有的分页查询都是由 SQL 语句完成，前台仅仅是一个展示的过程。

31.2　表数据普通查询分页

对于数据库中表的数据的 Web 显示，如果没有展示顺序的需要，而且因为满足条件的记录如此之多，就不得不对数据进行分页处理。常常用户并不是对所有数据都感兴趣的，或者大部分情况下，他们只看前几页。

通常有以下两种分页技术可供选择。

```
Select * from (
Select rownum rn,t.* from table t)
Where rn>&minnum and rn<=&maxnum
```

或者

```
Select * from (
Select rownum rn,t.* from table t rownum<=&maxnum)
Where rn>&minnum
```

看似相似的分页语句，在响应速度上其实有很大的差别。来看一个测试过程，首先创建一个测试表。

```
SQL>create table test as select * from dba_objects;
```

并反复地插入相同数据。

```
SQL>insert into test select * from test;
```

最后，查询该表，可以看到该表的记录数约为 80 万条。

```
SQL> select count(*) from test
  COUNT(*)
----------
   831104
```

现在分别采用两种分页方式，在第一种分页方式中：

```
SQL> select * from (
  2  select rownum rn,t.* from test t)
  3  where rn>0 and rn <=50;

已选择 50 行。
已用时间:  00: 00: 01.03

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=65 Bytes=12350)
   1   0  VIEW (Cost=10 Card=65 Bytes=12350)
   2   1    COUNT
   3   2      TABLE ACCESS (FULL) OF 'TEST' (Cost=10 Card=65 Bytes=5590)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
     10246  consistent gets
         0  physical reads
         0  redo size
         ……
```

可以看到，这种方式查询第一页的一致性读有 10246 个，结果满足了，但是效率是很差的，如果采用第二种方式：

```
SQL> select * from (
  2  select rownum rn,t.* from test t
  3  where rownum <=50)
  4  where rn>0;

已选择 50 行。
已用时间:  00: 00: 01.00

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=50 Bytes=9500)
   1   0  VIEW (Cost=10 Card=50 Bytes=9500)
   2   1    COUNT (STOPKEY)
   3   2      TABLE ACCESS (FULL) OF 'TEST' (Cost=10 Card=65 Bytes=5590)
```

```
Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
        82  consistent gets
         0  physical reads
         0  redo size
         ……
```

得到了同样的结果，一致性读只有 82 个，从以上的例子可以看到，通过把 rownum 引入到第二层，却得到了一个完全不一样的执行计划，注意在执行计划中的 stopkey，它是 8i 引入的新操作，这种操作专门为提取 Top n 的需求做了优化。

从上面的例子可以再想到，因为 stopkey 的功能影响到了分页的一致性读的多少，会不会越往后翻页速度就越慢呢？事实也的确如此，例如：

```
SQL> select * from (
  2  select rownum rn,t.* from test t
  3  where rownum <=10000)
  4  where rn>9950;

已选择 50 行。
已用时间:  00: 00: 01.01

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
      2616  consistent gets
         0  physical reads
         0  redo size
        ……
```

选择靠后一点的数据时，逻辑读开始变大，当选择到最后几页时，一致性读已经与上面的相似了。

```
SQL> select * from (
  2  select rownum rn,t.* from test t
  3  where rownum <=800000)
  4  where rn>799950;

已选择 50 行。
已用时间:  00: 00: 01.03

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
     10242  consistent gets
         0  physical reads
         0  redo size
         ……
```

不过，所幸的是，大部分的用户只看开始 5%的数据，而没有兴趣看最后面的数据，通过第

二种改良的分页技术，可以方便快速地显示前面的数据，而且不会让用户感觉到慢。

31.3 FIRST_ROWS 对分页的影响

FIRST_ROWS 设计的目的是为了更快地提供响应速度而设计的 CBO 提示，一般倾向在有索引的情况下走 nested_loop 连接方式，在分页技术中，合适地使用 FIRST_ROWS 可以部分地提高响应速度。

例如，创建一个测试表，并做关联查询分页：

```
create table page_test as select rownum id,t.* from test t;
```

这样创建出来的新表 page_test 的 id 是惟一的。

先分析该表：

```
SQL> analyze table page_test compute statistics for table for all columns;
Table analyzed
```

然后，分别用普通的分页与加提示的分页来进行测试：

```
SQL> select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4          page_test b,
  5          page_test c
  6     where a.id=b.id
  7     and b.id=c.id
  8     and rownum<=5
  9*    ) where rn>0

已用时间:  00: 00: 03.02

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=4215 Card=5 Bytes=395)
   1   0  VIEW (Cost=4215 Card=5 Bytes=395)
   2   1    COUNT (STOPKEY)
   3   2      HASH JOIN (Cost=4215 Card=831104 Bytes=23270912)
   4   3        HASH JOIN (Cost=2507 Card=831104 Bytes=6648832)
   5   4          TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=3324416)
   6   4          TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=3324416)
   7   3        TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=16622080)

Statistics
----------------------------------------------------------
        0  recursive calls
        0  db block gets
    21579  consistent gets
        0  physical reads
        0  redo size
        ……

SQL> select * from (
```

```
  2     select /*+ first_rows */
  3     rownum rn,a.object_name
  4     from page_test a,
  5         page_test b,
  6         page_test c
  7     where a.id=b.id
  8     and b.id=c.id
  9     and rownum<=5
 10    ) where rn>0;

已用时间:  00: 00: 04.03

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=1251281757 Card=5 Bytes=395)
   1    0  VIEW (Cost=1251281757 Card=5 Bytes=395)
   2    1    COUNT (STOPKEY)
   3    2      HASH JOIN (Cost=1251281757 Card=831104 Bytes=23270912)
   4    3        TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=3324416)
   5    3        MERGE JOIN (CARTESIAN) (Cost=873491355 Card=690733858816 Bytes=16577612611584)
   6    5          TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=16622080)
   7    5          BUFFER (SORT) (Cost=873490304 Card=831104 Bytes=3324416)
   8    7            TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=3324416)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
     21579  consistent gets
         0  physical reads
         0  redo size
         ……
```

从上面的例子可以看到，在关联的表没有索引的情况下，FIRST_ROWS 的效率是很差的，这里还是选择前面的几条记录，如果选择后面一些的记录，整个查询就会 hang 在那里而得不到结果。如果加上索引，情况将有所改变。

```
SQL> create index ind_page_test_id on page_test(id);
Index created
SQL> analyze index ind_page_test_id compute statistics;
Index analyzed

SQL>select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4         page_test b,
  5         page_test c
  6     where a.id=b.id
  7     and b.id=c.id
  8     and rownum<=5
  9*    ) where rn>0

已用时间:  00: 00: 02.09
```

```
    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2473 Card=5 Bytes=395)
       1   0  VIEW (Cost=2473 Card=5 Bytes=395)
       2   1    COUNT (STOPKEY)
       3   2      HASH JOIN (Cost=2473 Card=831104 Bytes=23270912)
       4   3        HASH JOIN (Cost=765 Card=831104 Bytes=6648832)
       5   4          INDEX (FAST FULL SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE) (Cost=180 Card=
831104 Bytes=3324416)
       6   4          INDEX (FAST FULL SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE) (Cost=180 Card=
831104 Bytes=3324416)
       7   3        TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=16622080)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
         3725  consistent gets
            0  physical reads
            0  redo size
            ……

    SQL> select * from (
      2     select /*+ first_rows */
      3     rownum rn,a.object_name
      4     from page_test a,
      5          page_test b,
      6          page_test c
      7     where a.id=b.id
      8     and b.id=c.id
      9     and rownum<=5
     10    ) where rn>0;

    已用时间:  00: 00: 01.01

    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=1666362 Card=5 Bytes=395)
       1   0  VIEW (Cost=1666362 Card=5 Bytes=395)
       2   1    COUNT (STOPKEY)
       3   2      NESTED LOOPS (Cost=1666362 Card=831104 Bytes=23270912)
       4   3        NESTED LOOPS (Cost=1666362 Card=831104 Bytes=19946496)
       5   4          INDEX (FULL SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE) (Cost=1852 Card=831104
Bytes=3324416)
       6   4          TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_TEST' (Cost=3 Card=1 Bytes=20)
       7   6            INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE) (Cost=2 Card=1)
       8   3        INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE) (Cost=2 Card=1 Bytes=4)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
```

```
          0  db block gets
         35  consistent gets
          0  physical reads
          0  redo size
          ……
```

从这里可以看到，使用 FIRST_ROWS 使查询的逻辑读降低到了 35 个，而且响应速度也变得更快，从执行计划也可以看到，这里用了我们倾向使用的 nested loops 的连接方式。再进一步观察其执行时间：

```
SQL>set autot off
SQL>select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4          page_test b,
  5          page_test c
  6     where a.id=b.id
  7     and b.id=c.id
  8     and rownum<=5
  9*    ) where rn>0

    ……
已用时间:  00: 00: 01.08

SQL> select * from (
  2     select /*+ first_rows */
  3     rownum rn,a.object_name
  4     from page_test a,
  5          page_test b,
  6          page_test c
  7     where a.id=b.id
  8     and b.id=c.id
  9     and rownum<=5
 10    ) where rn>0;

……

已用时间:  00: 00: 00.00
```

可以看到，实行时间几乎为 0。这个是一个比较好的结果，表示采用 FIRST_ROWS 提示起到了很好的效果，因为 FIRST_ROWS 在没有索引的表上关联是不合适的，而且根据 FIRST_ROWS 提示的功能是为了满足最快反应速度，也就是说对开始的页面返回速度很快，但是对后面的行返回速度很慢，下面就来测试一下。

```
SQL>select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4          page_test b,
  5          page_test c
  6     where a.id=b.id
  7     and b.id=c.id
  8     and rownum<=5005
  9*    ) where rn>5000
```

```
……
已用时间:  00: 00: 01.08

SQL> select * from (
  2     select /*+ first_rows */
  3     rownum rn,a.object_name
  4     from page_test a,
  5          page_test b,
  6          page_test c
  7     where a.id=b.id
  8     and b.id=c.id
  9     and rownum<=5005
 10    ) where rn>5000;

……
已用时间:  00: 00: 00.00

SQL> select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4          page_test b,
  5          page_test c
  6     where a.id=b.id
  7     and b.id=c.id
  8     and rownum<=50005
  9     ) where rn>50000;

……
已用时间:  00: 00: 01.09

SQL> select * from (
  2     select /*+ first_rows */
  3     rownum rn,a.object_name
  4     from page_test a,
  5          page_test b,
  6          page_test c
  7     where a.id=b.id
  8     and b.id=c.id
  9     and rownum<=50005
 10    ) where rn>50000;

……
已用时间:  00: 00: 00.07

SQL> select * from (
  2     select rownum rn,a.object_name
  3     from page_test a,
  4          page_test b,
  5          page_test c
  6     where a.id=b.id
  7     and b.id=c.id
```

```
  8     and rownum<=500005
  9     ) where rn>500000;

……
已用时间:  00: 00: 02.06

SQL> select * from (
  2     select /*+ first_rows */
  3     rownum rn,a.object_name
  4     from page_test a,
  5          page_test b,
  6          page_test c
  7     where a.id=b.id
  8     and b.id=c.id
  9     and rownum<=500005
 10    ) where rn>500000;

……
已用时间:  00: 00: 07.07
```

通过不同的测试，可以看到，在开始部分，增加提示的速度明显要比不增加提示要快，但是，随着往后翻页，速度会越来越慢。幸好的是，大部分的用户只对前面的数据感兴趣，所以增加前面部分的分页速度是很重要的。其实还可以通过程序转换的方式，即前面的分页查询使用 hint 提示，而后面部分则使用普通的分页查询。或者，对于很多条记录的分页查询，根本就没有最后一页这样显示最后部分的按钮，类似 Google，可以一次显示 10 个页面的链接。

31.4　带排序需求的分页

很多情况下，差不多是大部分情况下，是需要按照特定排序条件后再分页的，如论坛是按最后更新时间倒排序并分页显示的，在这样的情况下，就不能写成：

```
Select * from table where rownum <100 order by modify_time desc
```

因为这样的话，其实是先取前 100 条记录再排序，这并不符合要求，所以这样条件下的排序都只能写成：

```
Select * from (
Select rownum rn,t.* from (
Select * from table order by modify_time desc) t where rownum<=&maxnum)
Where rn>&minnum
```

或者，可以考虑用从 8i 开始引进的分析函数，如：

```
Select * from (
Select row_number() over(order by modify_time desc) rn,t.* from Select * from table)
Where rn>&minnum and rn<=&maxnum
```

从以上的语句可以看到，几乎是整个表或者是符合条件的表全排序后再取符合条件的记录，所以这里体现不到 rownum stopkey 的优势，可没有 FIRST_ROWS 的优势，反而通过分析函数能够进一步简化操作，使语句看起来没有那么复杂。

那么，使用 rownum 与分析函数的性能差别在哪里呢，或者有没有性能差别呢？先从不同的数据库版本来测试这个问题。以下是从 9204 得到的测试结果：

```
SQL>select * from (
  2       select rownum countnum,t.* from (
  3       select * from page_test where owner ='SYSTEM' order by object_id desc) t where
rownum<=50
  4*      ) where countnum>0

已选择 50 行。
已用时间:  00: 00: 01.04

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2580 Card=50 Bytes=10150)
   1   0  VIEW (Cost=2580 Card=50 Bytes=10150)
   2   1    COUNT (STOPKEY)
   3   2      VIEW (Cost=2580 Card=118729 Bytes=22558510)
   4   3        SORT (ORDER BY STOPKEY) (Cost=2580 Card=118729 Bytes=9617049)
   5   4          TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=118729 Bytes=9617049)

Statistics
----------------------------------------------------------
        0  recursive calls
        0  db block gets
    10787  consistent gets
        0  physical reads
        0  redo size
        ……

SQL> select * from (
  2       select row_number() over(order by object_id desc) countnum,t.*
  3       from page_test t where owner ='SYSTEM')
  4       where countnum>0 and countnum<=50;

已选择 50 行。
已用时间:  00: 00: 01.04

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2580 Card=118729 Bytes=24101987)
   1   0  VIEW (Cost=2580 Card=118729 Bytes=24101987)
   2   1    WINDOW (SORT PUSHED RANK) (Cost=2580 Card=118729 Bytes=9617049)
   3   2      TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=118729 Bytes=9617049)

Statistics
----------------------------------------------------------
        0  recursive calls
        0  db block gets
    10787  consistent gets
        0  physical reads
        0  redo size
        ……
```

从这里可以看到，rownum 与分析函数执行计划是全表扫描，时间与一致性读几乎没有差别，

这里先不考虑通过索引对以上两个语句进行优化的情况，而是看看大量反复执行的情况下，它们有什么差别。

先创建一个与统计信息有关的视图：

```
create or replace view stats as
select 'STAT...'||a.name name,b.value
from v$statname a,v$mystat b
where a.statistic#=b.statistic#
union all
select 'LATCH.'||name name,gets from v$latch
```

再创建一个用于存放临时数据的临时表：

```
create global temporary table RUN_STATS
 (
  RUNID VARCHAR2(20),
  NAME  VARCHAR2(80),
  VALUE NUMBER
)
on commit preserve rows;
```

最后创建一个包：

```
create or replace package runstats_pgk
as
   procedure rs_start;
   procedure rs_middle;
   procedure rs_stop(p_difference_threshold in number default 0);
end;

create or replace package body runstats_pgk
as
  g_start number;
  g_run1  number;
  g_run2  number;

 procedure rs_start
 is
 begin
    delete from run_stats;

    insert into run_stats
    select 'before',stats.* from stats;
    g_start := dbms_utility.get_time;
 end;

 procedure rs_middle
 is
 begin
   g_run1 :=(dbms_utility.get_time-g_start);

   insert into run_stats
     select 'after 1',stats.* from stats;
     g_start := dbms_utility.get_time;
 end;
```

```
procedure rs_stop(p_difference_threshold in number default 0)
is
begin
  g_run2 :=(dbms_utility.get_time-g_start);

  dbms_output.put_line
   ('run1 ran in '||g_run1||' hsecs');
  dbms_output.put_line
   ('run2 ran in '||g_run2||' hsecs');
  dbms_output.put_line
   ('run1 ran in '||round(g_run1/g_run2*100,2)||'% if the time');
  dbms_output.put_line(chr(9));

  insert into run_stats
   select 'after 2',stats.* from stats;

  dbms_output.put_line
   (rpad('Name',30)||lpad('Run1',10)||lpad('Run2',10)||lpad('Diff',10));

  for x in
   (select rpad(a.name,30)||to_char(b.value-a.value,'9,999,999')||
                  to_char(c.value-b.value,'9,999,999')||
                  to_char((c.value-b.value)-(b.value-a.value),'9,999,999') data
     from run_stats a,run_stats b,run_stats c
     where a.name=b.name
       and b.name=c.name
       and a.runid='before'
       and b.runid='after 1'
       and c.runid='after 2'
       and c.value-a.value>0
       and abs((c.value-b.value)-(b.value-a.value)) > p_difference_threshold
       order by abs((c.value-b.value)-(b.value-a.value))
    ) loop
      dbms_output.put_line(x.data);
 end loop;

 dbms_output.put_line(chr(9));
 dbms_output.put_line
  ('run1 latchs total versus runs -- defference and pct');
 dbms_output.put_line
  (lpad('Run1',10)||lpad('Run2',10)||lpad('Diff',10)||lpad('Pct',8));

 for x in
  (select to_char(run1,'9,999,999')||
         to_char(run2,'9,999,999')||
         to_char(diff,'9,999,999')||
         to_char(round(run1/run2*100,2),'9,999,999')||'%' data
    from
    (select sum(b.value-a.value) run1,sum(c.value-b.value) run2,
           sum((c.value-b.value)-(b.value-a.value)) diff
    from run_stats a,run_stats b,run_stats c
```

```
          where a.name=b.name
            and b.name=c.name
            and a.runid='before'
            and b.runid='after 1'
            and c.runid='after 2'
            and a.name like 'LATCH%'
            )
         ) loop
           dbms_output.put_line(x.data);
     end loop;
  end;
end;
```

有了该包后，就可以测试任何两个语句的执行差异了，为了反复测试 rownum 与分析函数，再创建一个测试用的存储过程：

```
create or replace procedure sp_test is
  num1 number;
  num2 number;
begin
  runstats_pgk.rs_start;
  for i in 1..100 loop
    select count(*) into num1 from
    (
       select * from (
       select rownum countnum,t.* from (
       select * from page_test where owner ='SYSTEM' order by object_id desc) t where rownum<=50
       ) where countnum>0
    );
  end loop;
  runstats_pgk.rs_middle;
  for i in 1..100 loop
    select count(*) into num2 from (
       select * from (
       select row_number() over(order by object_id desc) countnum,t.*
       from page_test t where owner ='SYSTEM')
       where countnum>0 and countnum<=50
    );
  end loop;
  runstats_pgk.rs_stop(100);
end sp_test;
```

执行这个过程：

```
SQL> set serveroutput on size 20000
SQL> exec sp_test;
run1 ran in 2982 hsecs
run2 ran in 2126 hsecs
run1 ran in 140.26% if the time

Name                            Run1     Run2     Diff
LATCH.enqueue hash chains          154       26     -128
LATCH.simulator hash latch      68,096   68,224      128
LATCH.checkpoint queue latch       961      740     -221
LATCH.library cache pin alloca     364        0     -364
```

```
LATCH.cache buffers chains    2,160,414 2,160,016     -398
LATCH.shared pool                564      109     -455
LATCH.library cache pin           742      222     -520
STAT...recursive cpu usage       2,981    2,129     -852
LATCH.library cache             1,149      222     -927


run1 latchs total versus runs -- defference and pct
     Run1     Run2     Diff    Pct
 2,234,287 2,230,906    -3,381      100%

PL/SQL procedure successfully completed.
```

从以上 100 次的执行结果可以看到，两者的资源消耗差不多，但是从时间上来说，使用分析函数的时间更短。也就是说，在内部有排序的情况下，并且在 OLTP 环境中，使用分析函数占用更大的优势。

当然以上是在 9i 而且是 9204 的环境下，Oracle 对分析函数做了较大的更改与优化，其实在 8i（分析函数刚出来的时候），分析函数用于分页效率是很差的，例如，以下是在版本 817 中的一个测试。

```
SQL>  select * from (
  2       select rownum countnum,t.* from (
  3       select * from page_test where owner ='SYSTEM' order by object_id desc) t where
rownum<=50
  4       ) where countnum>0;

已选择 50 行。
已用时间:  00: 00: 00.05

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=8167 Card=118730 Bytes=24102190)
   1   0  VIEW (Cost=8167 Card=118730 Bytes=24102190)
   2   1    COUNT (STOPKEY)
   3   2      VIEW (Cost=8167 Card=118730 Bytes=22558700)
   4   3        SORT (ORDER BY STOPKEY) (Cost=8167 Card=118730 Bytes=9617130)
   5   4          TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1623 Card=118730 Bytes=9617130)

Statistics
----------------------------------------------------------
        0  recursive calls
      138  db block gets
    10691  consistent gets
      269  physical reads
        0  redo size
        ……

SQL> select * from (
  2       select row_number() over(order by object_id desc) countnum,t.*
  3       from page_test t where owner ='SYSTEM')
  4       where countnum>0 and countnum<=50;
```

```
已选择 50 行。

已用时间:  00: 00: 14.05

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=8167 Card=118730 Bytes=24102190)
   1    0   VIEW (Cost=8167 Card=118730 Bytes=24102190)
   2    1     WINDOW (SORT)
   3    2       TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1623 Card=118730 Bytes=9617130)

Statistics
----------------------------------------------------------
     12965  recursive calls
      3505  db block gets
     15316  consistent gets
      4749  physical reads
    303772  redo size
       ……
```

以上的语句都是反复执行后的一个结果，避免了初次执行时带来的差异，可以发现，分析函数的效率比 rownum 差了很多，而且有大量的递归调用。也就证明了 8i 的分析函数内部其实还有很多其他的执行，并导致产生了大量的 redo。

从以上的分析可以看到，在 8i 的条件下，分析函数是不适合做分页的，在 9i 中对分析函数的专门优化后，分析函数已经有了比 rownum 更好的性能。

31.5　分页的速度优化

在以上的一些例子中，都是考虑如何分页以及怎么分页的问题，很多例子都是全表扫描，因为仅仅也只是测试性能的对比，对实际性能没有要求。但是实际上，大部分分页是有条件要求的，按照指定的条件筛选记录并分页，而且对分页的性能有非常强的要求，特别是访问比较频繁的 Web 站点。对于这样的语句，该如何进行优化呢？比如，还是在我们的测试表中，要选择出来“ SYSTEM ”用户下的所有表并进行分页，并按 object_id 排序。

当然，首先创建一个组合索引 owner+object_type。

```
SQL> create index ind_page_test_owner_type on page_test(owner,object_type,object_id);
Index created
SQL> analyze index ind_page_test_owner_type compute statistics;
Index analyzed

SQL> select rid from (
  2  select rownum rn,t.rid from (
  3  select rowid rid from page_test where owner='SYSTEM' and object_type='TABLE'
  4  order by object_id) t where rownum<=50)
  5* where rn>0

已选择 50 行。
```

```
    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=453 Card=50 Bytes=980)
       1   0  VIEW (Cost=453 Card=50 Bytes=980)
       2   1    COUNT (STOPKEY)
       3   2      VIEW (Cost=453 Card=4749 Bytes=33243)
       4   3        SORT (ORDER BY STOPKEY) (Cost=453 Card=4749 Bytes=104478)
       5    4             TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_TEST' (Cost=424 Card=4749
Bytes=104478)
       6   5           INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE'(NON-UNIQUE) (Cost=18
Card=4749)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
         2171  consistent gets
            0  physical reads
            0  redo size
            ……
```

可以看到这样的语句产生了 2171 个一致性读。如何优化与减少这个一致性读呢，其实可以考虑把 where 条件与排序语句放到一个索引中，利用这个索引获得 rowid，再根据 rowid 去获得分页数据。所以，再创建一个优化索引 owner+object_type+object_id。

```
    SQL> create index ind_page_test_owner_type_id on page_test(owner,object_type,object_id);
    Index created
    SQL> analyze index ind_page_test_owner_type_id compute statistics;
    Index analyzed
```

先看看用于查询分页所需要的 rowid 所产生的消耗：

```
    SQL>select rid from (
      2  select rownum rn,t.rid from (
      3  select rowid rid from page_test where owner='SYSTEM' and object_type='TABLE'
      4  order by object_id) t where rownum<=50)
      5* where rn>0

    已选择 50 行。

    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=47 Card=50 Bytes=980)
       1   0  VIEW (Cost=47 Card=50 Bytes=980)
       2   1    COUNT (STOPKEY)
       3   2      VIEW (Cost=47 Card=4524 Bytes=31668)
       4   3        SORT (ORDER BY STOPKEY) (Cost=47 Card=4524 Bytes=99528)
       5   4          INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID' (NON-UNIQUE) (Cost=19
Card=4524 Bytes=99528)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
```

```
          67  consistent gets
           0  physical reads
           0  redo size
          ……
```

可以发现，只消耗了 67 个逻辑读，可以说是比较少了，因为整个查询加排序都是落在索引上面的，其实，根据这获得的 rowid 再去获得该页的数据就变得相对简单了：

```
SQL> select * from  page_test t,
  2  (select rid from (
  3  select rownum rn,t.rid from (
  4  select rowid rid from page_test where owner='SYSTEM' and object_type='TABLE'
  5  order by object_id) t where rownum<=50)
  6  where rn>0) b
  7  where t.rowid=b.rid;

已选择 50 行。

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=96 Card=50 Bytes=5292)
   1   0  NESTED LOOPS (Cost=96 Card=50 Bytes=5292)
   2   1    VIEW (Cost=47 Card=50 Bytes=980)
   3   2      COUNT (STOPKEY)
   4   3        VIEW (Cost=47 Card=4524 Bytes=31668)
   5   4          SORT (ORDER BY STOPKEY) (Cost=47 Card=4524 Bytes=99528)
   6   5            INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID' (NON-UNIQUE) (Cost=19
Card=4524 Bytes=99528)
   7   1    TABLE ACCESS (BY USER ROWID) OF 'PAGE_TEST' (Cost=1 Card=1 Bytes=88)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        116  consistent gets
          0  physical reads
          0  redo size
          ……
```

整个语句也只消耗了 116 个一致性读的块，对比前面优化前的 2171 个已经是非常不错了。而且，如果在 9i 中，该语句也可以用分析函数代替，但是，在有些情况下，分析函数可能采用 hash_join 的连接方式，如：

```
SQL>select * from
  2  (select rid from (
  3  select row_number() over(order by object_id) rn,rowid rid from page_test
  4  where owner='SYSTEM' and object_type='TABLE') t
  5  where rn>0 and rn<=50) b,page_test t
  6* where t.rowid=b.rid

已选择 50 行。

Execution Plan
----------------------------------------------------------
```

```
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1188 Card=4524 Bytes=488592)
       1    0  HASH JOIN (Cost=1188 Card=4524 Bytes=488592)
       2    1    VIEW (Cost=47 Card=4524 Bytes=90480)
       3    2      WINDOW (BUFFER PUSHED RANK) (Cost=47 Card=4524 Bytes=99528)
       4    3        INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID'(NON-UNIQUE) (Cost=19
Card=4524 Bytes=99528)
       5    1    TABLE ACCESS (FULL) OF 'PAGE_TEST' (Cost=1051 Card=831104 Bytes=73137152)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
        10858  consistent gets
            0  physical reads
            0  redo size
            ……
```

对于这种方式，改为 nested_loop 连接即可：

```
    SQL> select /*+ ordered use_nl(b,t) */ * from
      2  (select rid from (
      3  select row_number() over(order by object_id) rn,rowid rid from page_test
      4  where owner='SYSTEM' and object_type='TABLE') t
      5  where rn>0 and rn<=50) b,page_test t,
      6  where t.rowid=b.rid;

    已选择 50 行。

    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=4571 Card=4524 Bytes=488592)
       1    0  NESTED LOOPS (Cost=4571 Card=4524 Bytes=488592)
       2    1    VIEW (Cost=47 Card=4524 Bytes=90480)
       3    2      WINDOW (BUFFER PUSHED RANK) (Cost=47 Card=4524 Bytes=99528)
       4    3        INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID'(NON-UNIQUE) (Cost=19
Card=4524 Bytes=99528)
       5    1    TABLE ACCESS (BY USER ROWID) OF 'PAGE_TEST' (Cost=1 Card=1 Bytes=88)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
          116  consistent gets
            0  physical reads
            0  redo size
            ……
```

以上的优化方案对单个大表的分页是颇有效的，在某些表要先做关联后分页的语句中，同样的优化也能明显提高查询的速度与效率。接下来看以下关联查询的分页语句。

```
    SQL>select * from (
      2       select rownum countnum,t.* from (
      3       select t1.* from page_test t1,page_test t2
      4       where t1.id=t2.id and
      5       t1.owner ='SYSTEM' order by t1.object_id desc) t where rownum<=50
```

```
  6*      ) where countnum>0

已选择 50 行。


Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
   1    0  VIEW
   2    1    COUNT (STOPKEY)
   3    2      VIEW
   4    3        SORT (ORDER BY STOPKEY)
   5    4          NESTED LOOPS
   6    5            TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_TEST'
   7   6              INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID' (NON-UNIQUE)
   8    5            INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE)

Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
       544  consistent gets
         0  physical reads
         0  redo size
         ……
```

以上语句最大的问题就是在于先关联，后分页的问题，其实，可以先在一个表中选择要分页的记录，然后关联另外一个表，这样的情况下，其实只需要关联当前页的内容，如当前 50 条记录，改写一下以上的语句。

```
SQL> select * from page_test t1,
  2  (select * from  page_test t,
  3     (select rid from (
  4     select rownum rn,t.rid from (
  5     select rowid rid from page_test where owner='SYSTEM' and object_type='TABLE'
  6     order by object_id) t where rownum<=50)
  7     where rn>0) b
  8     where t.rowid=b.rid) t2
  9* where t1.id=t2.id

已选择 50 行。


Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
   1    0  TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_TEST'
   2    1    NESTED LOOPS
   3    2      NESTED LOOPS
   4    3        VIEW
   5    4          COUNT (STOPKEY)
   6    5            VIEW
   7    6              SORT (ORDER BY STOPKEY)
```

```
   8    7              INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_OWNER_TYPE_ID' (NON-UNIQUE)
   9    3       TABLE ACCESS (BY USER ROWID) OF 'PAGE_TEST'
  10    2      INDEX (RANGE SCAN) OF 'IND_PAGE_TEST_ID' (NON-UNIQUE)


Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
       119  consistent gets
       ……
```

同样的结果，同样关联，但是逻辑读降低到只有 119 个，这个写法在两个大表的关联分页中非常有效而且明显，通常能把 hash_join 变成更快一些的 nest loop 关联，因为从一个表中筛选后并进行分页处理后的记录，只有当前页的数目，如 50 条，当然 50 条记录做 nested loop 的速度是可想而知的非常快了。当然，如果在特定的情况下，又出现了 hash join，也可以利用类似以前的方法，把小的已经分页处理过的表放前面，并使用/*+ ordered use_nl(t) */的提示。

31.6 分页中的注意事项

Oracle 的分页支持做的并不是很好的，所以在很多情况下，会有想不到的结果，以下就一些可能出现的问题用真实的案例说明一下。

31.6.1 真实案例——表中存在 union all 的视图时，可能选择错误的执行计划

假定 bwm_users 就是一个 union all 的视图。代码如下：

```
select *
from mv_bmw_users_db1
union all
select  *
from mv_bmw_users_db2
```

如果在该视图上执行如下操作，可以看到：

```
SQL> select * from
  2  (select rownum linenum,id,nick from
  3  (select id,nick from bmw_users  where nick ='test' order by id)
  4  where rownum < 50)
  5  where linenum >=1;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=20385 Card=50 Bytes=2401)
   1    0   VIEW (Cost=20385 Card=50 Bytes=2401)
   2    1     COUNT (STOPKEY)
   3    2       VIEW (Cost=20385 Card=1728633 Bytes=62230788)
   4    3         SORT (ORDER BY STOPKEY) (Cost=20385 Card=1728633 Bytes=62230788)
   5    4           VIEW OF 'BMW_USERS' (Cost=9278 Card=1728633 Bytes=62230788)
   6    5             UNION-ALL
```

```
       7    6              TABLE ACCESS (FULL) OF 'MV_BMW_USERS_DB1' (Cost=4639 Card=864090
Bytes=38884050)
       8    6              TABLE ACCESS (FULL) OF 'MV_BMW_USERS_DB2' (Cost=4639 Card=864543
Bytes=38904435)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
        97298  consistent gets
        20770  physical reads
            0  redo size
```

一个非常简单的查询，在 nick 上是有惟一索引的，而且表与索引都是分析过的，居然是全表扫描，耗费非常大的资源，这个时候，Oracle 已经不能正确地判断使用索引了，所以错误地使用了全表，从统计信息也可以看到，该查询产生了大量的一致性读与磁盘读。这个时候，就是强行指定 hint 也不能改变 Oracle 的执行计划，当然，这样是行不通的，必须找到一个行之有效的办法。

这样的问题怎么解决呢？有两个办法，一个是仍然使用 union all 语句在查询中，直接查询基表而不是视图。如以上语句可以改造为：

```
    SQL> select * from
      2  (select rownum linenum,id,nick from
      3  (select * from
      4  (select id,nick from MV_BMW_USERS_DB1 where nick ='test'
      5  union all
      6  select id,nick from MV_BMW_USERS_DB1 where nick ='test')
      7  order by id)
      8  where rownum < 50)
      9  where linenum >=1;

    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=17 Card=2 Bytes=98)
       1   0  VIEW (Cost=17 Card=2 Bytes=98)
       2   1    COUNT (STOPKEY)
       3   2      VIEW (Cost=17 Card=2 Bytes=72)
       4   3        SORT (ORDER BY STOPKEY) (Cost=17 Card=2 Bytes=72)
       5   4          VIEW (Cost=8 Card=2 Bytes=72)
       6   5            UNION-ALL
       7    6              TABLE ACCESS (BY INDEX ROWID) OF 'MV_BMW_USERS_DB1' (Cost=4 Card=1
Bytes=45)
       8   7               INDEX (RANGE SCAN) OF 'IND_MV_BMW_USERS_NICK1' (NON-UNIQUE) (Cost=3
Card=1)
       9    6              TABLE ACCESS (BY INDEX ROWID) OF 'MV_BMW_USERS_DB1' (Cost=4 Card=1
Bytes=45)
      10   9               INDEX (RANGE SCAN) OF 'IND_MV_BMW_USERS_NICK1' (NON-UNIQUE) (Cost=3
Card=1)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
```

```
          8  consistent gets
          0  physical reads
          0  redo size
```

语句基本上是一样的，只是这次查询了基表，而不是视图，执行计划马上发生了改变，这次能使用了索引，而且成本有了很大的减少，可以看到一致性读减少到只有 8 个块，而且磁盘读为 0。

或者采用第二种方法，分析函数的办法，把语句改为：

```
SQL>select * from
 1 (select row_number() over(order by id) rn,id,nick from bmw_users where nick ='test')
 2 where rn <50 and rn >=1;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=1 Bytes=50)
   1    0   VIEW (Cost=13 Card=1 Bytes=50)
   2    1     WINDOW (SORT PUSHED RANK) (Cost=13 Card=1 Bytes=45)
   3    2       VIEW OF 'BMW_USERS' (Cost=4 Card=1 Bytes=45)
   4    3         UNION-ALL (PARTITION)
   5    4           TABLE ACCESS (BY INDEX ROWID) OF 'MV_BMW_USERS_DB1' (Cost=4 Card=1 Bytes=45)
   6    5             INDEX (RANGE SCAN) OF 'IND_MV_BMW_USERS_NICK1' (NON-UNIQUE) (Cost=3
Card=1)
   7    4           TABLE ACCESS (BY INDEX ROWID) OF 'MV_BMW_USERS_DB2' (Cost=4 Card=1 Bytes=45)
   8    7             INDEX (RANGE SCAN) OF 'IND_MV_BMW_USERS_NICK2' (NON-UNIQUE) (Cost=3
Card=1)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
```

可以看到，同样的功能，分析函数的方法是最简单的，同样也能正确地使用索引，问题得以解决。

31.6.2 真实案例——rowid 分页中，执行计划的错误选择与处理

在 31.5 节中，介绍了可以根据选择出来的 rowid 去获得记录，以达到优化的目的，但是在特定情况下，也会出现执行计划不稳定的情况，这样的话，一般只能是通过 hint 来解决了。

```
SQL> set autot trace
SQL> select rid from
  2  (select a.rowid rid,row_number() over(order by a.topic_type DESC,a.topic_last_post_id
DESC) rn
  3  from  forum_topics a
  4  WHERE a.forum_id=40
  5  AND a.topic_type < 2
  6  AND a.topic_status <> 3
  7  ) WHERE rn < 2 and rn >= 1;
```

```
    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1678 Bytes=33560)
       1    0  VIEW (Cost=2 Card=1678 Bytes=33560)
       2    1    WINDOW (SORT PUSHED RANK) (Cost=2 Card=1678 Bytes=31882)
       3    2       INDEX (RANGE SCAN) OF 'IND_FORUM_TOP_FOR_TP_ST_ID' (NON-UNIQUE) (Cost=2
Card=1678 Bytes=31882)

    Statistics
    ----------------------------------------------------------
            3  recursive calls
            0  db block gets
           11  consistent gets
            0  physical reads
            0  redo size
            ……
```

以上语句，只返回一行记录，查询出来了一个 rowid，执行计划是索引扫描即可，可见，逻辑读是非常少的。但是，把这个语句作为一个子查询，情况马上就变了：

```
    SQL>                                                                          SELECT
t.topic_id,t.topic_type,t.topic_distillate,t.topic_vote,t.topic_status,t.topic_moved_id,
      2  TO_CHAR(t.topic_time,'YYYY-MM-DD HH24:MI:SS')  topic_time,
      3  t.topic_last_post_id,t.topic_views,t.topic_title, t.topic_replies,
      4  t.topic_poster FROM forum_topics t
      5  where rowid in
      6  (select rid from
      7  (select a.rowid rid,row_number() over(order by a.topic_type DESC,a.topic_last_post_id
DESC) rn
      8  from  forum_topics a
      9  WHERE a.forum_id=40
     10  AND a.topic_type < 2
     11  AND a.topic_status <> 3
     12* ) WHERE rn < 2 and rn >= 1)

    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=854 Card=1678 Bytes=194648)
       1    0  HASH JOIN (SEMI) (Cost=854 Card=1678 Bytes=194648)
       2    1    TABLE ACCESS (FULL) OF 'FORUM_TOPICS' (Cost=444 Card=221324 Bytes=24124316)
       3    1    VIEW OF 'VW_NSO_1' (Cost=2 Card=1678 Bytes=11746)
       4    3      VIEW (Cost=2 Card=1678 Bytes=33560)
       5    4        WINDOW (SORT PUSHED RANK) (Cost=2 Card=1678 Bytes=31882)
       6    5           INDEX (RANGE SCAN) OF 'IND_FORUM_TOP_FOR_TP_ST_ID'(NON-UNIQUE) (Cost=2
Card=1678 Bytes=31882)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
         4613  consistent gets
            0  physical reads
```

```
          0  redo size
          ……
```

这里居然走了全表扫描后与子查询的 hash join。子查询才返回一条 rowid，根据这个 rowid 的查询 Oracle 居然不走 rowid 扫描后的 nested loop 的连接。这个时候，就是采用/*+ ROWID(t) */或者/*+ use_nl(t) */的提示也不能改变 Oracle 的执行计划。但是，如果采用原始 rule 的执行计划，就会发现情况大有好转。

```
    SQL>                SELECT                /*+                rule                */
t.topic_id,t.topic_type,t.topic_distillate,t.topic_vote,t.topic_status,t.topic_moved_id,
      2   TO_CHAR(t.topic_time,'YYYY-MM-DD HH24:MI:SS')  topic_time,
      3   t.topic_last_post_id,t.topic_views,t.topic_title, t.topic_replies,
      4   t.topic_poster FROM forum_topics t
      5   where rowid in
      6   (select rid from
      7   (select a.rowid rid,row_number() over(order by a.topic_type DESC,a.topic_last_post_id
DESC) rn
      8   from  forum_topics a
      9   WHERE a.forum_id=40
     10   AND a.topic_type < 2
     11   AND a.topic_status <> 3
     12   ) WHERE rn < 2 and rn >= 1) ;


    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=HINT: RULE
       1   0   NESTED LOOPS
       2   1     VIEW OF 'VW_NSO_1'
       3   2       SORT (UNIQUE)
       4   3         VIEW
       5   4           WINDOW (SORT PUSHED RANK)
       6   5             INDEX (RANGE SCAN) OF 'IND_FORUM_TOP_FOR_TP_ST_ID' (NON-UNIQUE)
       7   1     TABLE ACCESS (BY USER ROWID) OF 'FORUM_TOPICS'

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
           11  consistent gets
            0  physical reads
            0  redo size
            ……
```

通过最后的反复测试，发现只有在以下的情况下，强行指定驱动顺序与连接方式，Oracle 才能最终选择正确的执行计划。

```
    SQL>           SELECT           /*+           ordered           use_nl(t)           */
t.topic_id,t.topic_type,t.topic_distillate,t.topic_vote,
      2      t.topic_status,t.topic_moved_id,TO_CHAR(t.topic_time,'YYYY-MM-DD    HH24:MI:SS')
topic_time,
      3  t.topic_last_post_id,t.topic_views,t.topic_title, t.topic_replies,
      4  t.topic_poster FROM (select rid from
      5  (select a.rowid rid,row_number()  over(order by a.topic_type DESC,a.topic_last_post_id
```

```
DESC) rn
      6  from  forum_topics a
      7  WHERE a.forum_id=40
      8  AND a.topic_type < 2
      9  AND a.topic_status <> 3
     10  ) WHERE rn < 2 and rn >= 1)  b,forum_topics t
     11* where t.rowid=b.rid


    Execution Plan
    ----------------------------------------------------------
       0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1680 Card=1678 Bytes=216462)
       1   0   NESTED LOOPS (Cost=1680 Card=1678 Bytes=216462)
       2   1     VIEW (Cost=2 Card=1678 Bytes=33560)
       3   2       WINDOW (SORT PUSHED RANK) (Cost=2 Card=1678 Bytes=31882)
       4   3         INDEX (RANGE SCAN) OF 'IND_FORUM_TOP_FOR_TP_ST_ID' (NON-UNIQUE) (Cost=2
Card=1678 Bytes=31882)
       5   1     TABLE ACCESS (BY USER ROWID) OF 'FORUM_TOPICS' (Cost=1 Card=1 Bytes=109)

    Statistics
    ----------------------------------------------------------
            0  recursive calls
            0  db block gets
           11  consistent gets
            0  physical reads
            0  redo size
            ……
```

31.6.3　真实案例——使用 rownum 得到意想不到的结果

执行以下类似的翻页语句：

```
    select countrownum, company_name, member_id  from
     (select rownum countrownum, t1.* from
     (select *  from nirvana.sfa_opportunity
    where  status = 'selected'  and sales_id ='xuyi'
    order by repeat_flag asc) t1
    where rownum < 21)
    where (countrownum >= 1)

    select countrownum, company_name, member_id  from
     (select rownum countrownum, t1.* from
     (select *  from nirvana.sfa_opportunity
    where  status = 'selected'  and sales_id ='xuyi'
    order by repeat_flag asc) t1
    where rownum < 41)
    where (countrownum >= 21)
```

此时，也就是相邻的两个页切换时，会发现前一个页面已经显示的数据在下一个页面继续显示，出现少许重复值，但是记录本身是不重复的，经过仔细检查，发现数据中 repeat_flag 基本都是重复的，这是不是导致排序后顺序不一样呢？结果测试后果然是这样，当 rownum 选择不同的

情况下，内层排序后顺序可能不一样，也就是说，估计是和 Oracle 排序的实现细节有关系，比如内存分配的不一样就可能导致相同字段值排序后记录顺序的不一样。为此决定在 order by repeat_flag 后面增加其他不同值的字段，在这里选择了惟一的 ID 号，从而解决了这个问题。

```
select countrownum, company_name, member_id   from
 (select rownum countrownum , t1.* from
 (select *  from nirvana.sfa_opportunity
where  status = 'selected'  and sales_id ='xuyi'
order by repeat_flag asc  ,ID
) t1 where rownum < 21)
where (countrownum >= 1)
```

或者在 9i 版本以上，使用分析函数，一样可以避免这样的问题出现。

31.7 小结

从以上的一些总结可以看到，从数据库直接做 Web 分页还是有很多解决方案与优化方法的，所以，在实际情况中，需要根据实际的情况采用最适合的分页方式，特别是 OLTP 系统，一个小的调整对整个系统来说，说不定就是一次巨大的改变。所以，在 Web 分页中，应该注意：

(1) 分页的需求是否有排序的需求。

(2) 用户是不是通常只对前面的数据感兴趣。

(3) 注意到 rownum 与分析函数在 8i 与 9i 中对分页的语句的巨大影响。

(4) 注意分页中经常可能碰到的问题，有基本的优化手段。

作者简介

陈吉平，ITPUB 的 ID 为 piner，现在任职于国内某大型电子商务网站，主要负责网站后台数据库维护，擅长备份与恢复，数据库高可用性与系统容灾，对数据库优化也有很深的研究。希望能广交 Oracle 朋友，共同进步。

32

第 32 章　Oracle 数据封锁机制研究

在众多基于 Oracle 数据库的应用中，有不少的性能问题是源于对数据封锁机制的错误使用，而当发生问题后，又由于相关人员缺乏经验，往往不能及时定位与解决问题。对于初学者来讲，完全理解 Oracle 的数据封锁机制并不容易，且容易混淆多种封锁类型。本章一方面以实例的形式介绍产生数据封锁的各类情况，另一方面从数据库理论的角度加以分析，以期使读者既知其然，更知其所以然，使读者能够更加深入地理解 Oracle 的数据封锁机制，而不是简单地记住某些结论。

由于本人的水平有限并且缺乏相关的内部资料，另外 Oracle 本身也是一个不断发展的软件，所以文中难免会有片面、过时甚至错误的内容，恳请大家能够及时指正。

32.1　数据库锁的基本概念

为了确保并发用户在存取同一数据库对象时的正确性（即无丢失修改、可重复读、不读“脏”数据），数据库中引入了锁机制。基本的锁类型有两种：排他锁（Exclusive Locks 记为 X 锁）和共享锁（Share Locks 记为 S 锁）。

排他锁：若事务 T 对数据 D 加 X 锁，则其他任何事务都不能再对 D 加任何类型的锁，直至 T 释放 D 上的 X 锁；一般要求在修改数据前要向该数据加排他锁，所以排他锁又称为写锁。

共享锁：若事务 T 对数据 D 加 S 锁，则其他事务只能对 D 加 S 锁，而不能加 X 锁，直至 T 释放 D 上的 S 锁；一般要求在读取数据前要向该数据加共享锁，所以共享锁又称为读锁。

32.2　Oracle 多粒度封锁机制介绍

根据保护对象的不同，Oracle 数据库锁可以分为以下几大类：

- DML lock（data locks，数据锁）：用于保护数据的完整性。
- DDL lock（dictionary locks，字典锁）：用于保护数据库对象的结构（例如表、视图、索引的结构定义）。
- internal locks 和 latches（内部锁与闩）：保护数据库内部结构。

本章主要讨论 DML 锁。DML 锁的目的在于保证在多个并发用户情况下的数据完整性。DML 锁防止并发的 DML 与 DDL 操作互相产生破坏性的影响。例如，DML 锁确保在同一时间，一行数据只被一个事务修改；另外，一个包含有未提交修改的表不能被其他用户删除。

从封锁粒度（封锁对象的大小）的角度看，Oracle DML 锁共有两个层次，即行级锁和表级锁。

32.2.1 Oracle 的 TX 锁（事务锁、行级锁）

许多对 Oracle 不太了解的技术人员可能会以为每一个 TX 锁代表一被封锁的数据行，其实不然。TX 的本义是 Transaction（事务），当一个事务第一次执行数据更改（insert、update、delete）或使用 select... for update 语句进行查询时，它即获得一个 TX（事务）锁，直至该事务结束（执行 COMMIT 或 ROLLBACK 操作）时，该锁才被释放。所以，一个 TX 锁，可以对应多个被该事务锁定的数据行。

在 Oracle 的每行数据上，都有一个标志位来表示该行数据是否被锁定。Oracle 不像其他一些 DBMS（数据库管理系统）那样，建立一个链表来维护每一行被加锁的数据，这样就大大减小了行级锁的维护开销，也在很大程度上避免了其他数据库系统使用行级封锁时经常发生的锁数量不够的情况，从而也没有必要在某些情况下将行级锁升级为表级锁（或“页”级锁）。数据行上的锁标志一旦被置位，就表明该行数据被加 X 锁，Oracle 在数据行上没有 S 锁。

Oracle 在每个块（block）的头部，都保留有一个叫做 ITL（Interested Transaction List，可译为“相关事务列表”）的结构，每当一个事务要修改（或 select for update）块中的行时，它都需要将自己的事务标识记录在 ITL 的一项（即 slot）中，而被其修改（或 select for update）的行的行首的锁标识字节（Lock Bytes），记录事务在 ITL 中所占用的项。如图 32-1 所示，使用 ITL 第 1 项的事务对第 8 行加了锁，使用 ITL 第 2 项的事务对第 1 行加了锁。从某种意义上说，未提交事务所占用的 ITL slot 与行首的锁标识字节一起构成了对数据行的隐式封锁。

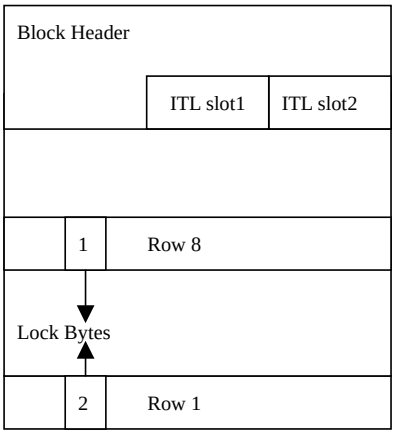

图 32-1 数据块中有关 Lock 的结构

当一个事务发现自己要修改的数据行已被其他未提交事务修改时，它便处于等待状态，但等待的不是行锁的释放，而是阻塞事务锁的释放。当阻塞事务提交或回滚后，其事务锁被释放，它所对应的隐式行锁也被释放，被阻塞的事务可以继续进行。虽然事务回滚到某个保留点（SAVEPOINT），能够释放某些隐式的行锁，但由于其事务锁没有被释放，所以被阻塞的事务仍将

继续等待，即使最初引起阻塞的隐式行锁已经被释放，情况也是一样，本章后面将有具体的例子予以说明。

数据块中 ITL 的项数取决于 INITRANS 与 MAXTRANS 这两个参数。INITRANS 是块被分配时初始建立的 ITL 项数，而 MAXTRANS 则是该块最多可以建立的 ITL 项数，也是该块允许并发修改的最大事务数。如果并发事务数超过 INITRANS，但仍小于 MAXTRANS，ITL 就会被分配新的项，当然分配成功与否还取决于该块是否还有足够的空闲空间（一般每一项占用 24 字节）。如果并发事务数已经达到 MAXTRANS 的限制，又有新的事务要修改块中的数据，则该事务将会被阻塞于一个活跃事务。

32.2.2　TM 锁（表级锁）

1. 意向锁的引出

表是由行组成的，当向某个表加锁时，一方面需要检查该锁的申请是否与原有的表级锁相容；另一方面，还要检查该锁是否与表中的每一行上的锁相容。比如一个事务要在一个表上加 S 锁，如果表中的一行已被另外的事务加了 X 锁，那么该锁的申请也应被阻塞。如果表中的数据很多，逐行检查锁标志的开销将很大，系统的性能将会受到影响。为了解决这个问题，可以在表级引入新的锁类型来表示其所属行的加锁情况，这就引出了“ 意向锁 ”的概念。

意向锁的含义是如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁；对任一结点加锁时，必须先对它的上层结点加意向锁。例如，对表中的任一行加锁时，必须先对它所在的表加意向锁，然后再对该行加锁。这样一来，事务对表加锁时，就不再需要检查表中每行记录的锁标志位了，系统效率得以大大提高。

2. 意向锁的类型

由两种基本的锁类型（S 锁、X 锁），可以自然地派生出两种意向锁：

- 意向共享锁（Intent Share Lock，简称 IS 锁）：如果要对一个数据库对象加 S 锁，首先要对其上级结点加 IS 锁，表示它的后裔结点拟（意向）加 S 锁。
- 意向排他锁（Intent Exclusive Lock，简称 IX 锁）：如果要对一个数据库对象加 X 锁，首先要对其上级结点加 IX 锁，表示它的后裔结点拟（意向）加 X 锁。

另外，基本的锁类型（S、X）与意向锁类型（IS、IX）之间还可以组合出新的锁类型，理论上可以组合出 4 种，即 S+IS、S+IX、X+IS 和 X+IX，但稍加分析不难看出，实际上只有 S+IX 有新的意义，其他三种组合都没有使锁的强度得到提高（即 S+IS=S、X+IS=X 和 X+IX=X，这里的“ = ”指锁的强度相同）。所谓锁的强度是指对其他锁的排斥程度。

这样又可以引入一种新的锁类型。

- 共享意向排他锁（Shared Intent Exclusive Lock，简称 SIX 锁）：如果对一个数据库对象加 SIX 锁，表示对它加 S 锁，再加 IX 锁，即 SIX=S+IX。例如，事务对某个表加 SIX 锁，则表示该事务要读整个表（所以要对该表加 S 锁），同时会更新个别行（所以要对该表加 IX 锁）。

这样数据库对象上所加的锁类型就可能有 5 种，即 S、X、IS、IX 和 SIX。

具有意向锁的多粒度封锁方法中任意事务 T 要对一个数据库对象加锁，必须先对它的上层结点加意向锁。申请封锁时应按自上而下的次序进行；释放封锁时则应按自下而上的次序进行；具

有意向锁的多粒度封锁方法提高了系统的并发度，减少了加锁和解锁的开销。

3. Oracle 的 TM 锁（表级锁）

Oracle 的 DML 锁（数据锁）正是采用了上面提到的多粒度封锁方法，其行级锁虽然只有一种（即 X 锁），但其 TM 锁（表级锁）类型共有 5 种，分别称为共享锁（S 锁）、排他锁（X 锁）、行级共享锁（RS 锁）、行级排他锁（RX 锁）和共享行级排他锁（SRX 锁），与上面提到的 S、X、IS、IX 和 SIX 相对应。需要注意的是，由于 Oracle 在行级只提供 X 锁，所以与 RS 锁（通过 select … for update 语句获得）对应的行级锁也是 X 锁（但是该行数据实际上还没有被修改），这与理论上的 IS 锁是有区别的。

表 32-1 为 Oracle 数据库 TM 锁的相容矩阵（Y=Yes 表示相容的请求；N=No 表示不相容的请求；-表示没有加锁请求）。

表 32-1 Oracle 数据库 TM 锁的相容矩阵

T1 \ T2	S	X	RS	RX	SRX	-
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
RS	Y	N	Y	Y	Y	Y
RX	N	N	Y	Y	N	Y
SRX	N	N	Y	N	N	Y
-	Y	Y	Y	Y	Y	Y

一方面，当 Oracle 执行 select…for update、insert、update、delete 等 DML 语句时，系统自动在所要操作的表上申请表级 RS 锁（select…for update）或 RX 锁（insert、update 和 delete），当表级锁获得后，系统再自动申请 TX 锁，并将实际锁定的数据行的锁标志位置位（指向该 TX 锁，更确切地说是指向该事务在该块中占用的 ITL Slot）；另一方面，程序或操作人员也可以通过 LOCK TABLE 语句来指定获得某种类型的 TM 锁。表 32-2 总结了 Oracle 中各 SQL 语句产生 TM 锁的情况。

表 32-2 Oracle 数据库 TM 锁小结

SQL 语句	表锁模式	允许的锁模式
Select * from table_name……	无	RS、RX、S、SRX、X
Insert into table_name……	RX	RS、RX
Update table_name……	RX	RS、RX
Delete from table_name……	RX	RS、RX
Select * from table_name for update	RS	RS、RX、S、SRX
lock table table_name in row share mode	RS	RS、RX、S、SRX
lock table table_name in row exclusive mode	RX	RS、RX
lock table table_name in share mode	S	RS、S
lock table table_name in share row exclusive mode	SRX	RS
lock table table_name in exclusive mode	X	无

可以看到，通常的 DML 操作（select…for update、insert、update、delete），在表级获得的

只是意向锁（RS 或 RX），其真正的封锁粒度还是在行级；另外，Oracle 数据库的一个显著特点是，在缺省情况下，单纯地读数据（select）并不加锁，Oracle 通过回滚段（Rollback Segment）来保证用户不读“脏”数据。这些都极大地提高了系统的并发程度。

由于意向锁及数据行上锁标志位的引入，极大地减小了 Oracle 维护行级锁的开销，这些技术的应用使 Oracle 能够高效地处理高度并发的事务请求。

32.3　Oracle 多粒度封锁机制的监控

这一节内容主要包括两部分：系统视图和监控脚本。

32.3.1　系统视图介绍

为了监控 Oracle 系统中锁的状况，就需要对几个系统视图有所了解。

1. v$lock 视图

v$lock 视图列出当前系统持有的或正在申请的所有锁的情况，其主要字段说明如表 32-3 所示：

表 32-3　　v$lock 视图主要字段说明

字段名称	类　型	说　　明
SID	NUMBER	会话（session）标识
TYPE	VARCHAR（2）	区分该锁保护对象的类型
ID1	NUMBER	锁标识 1
ID2	NUMBER	锁标识 2
LMODE	NUMBER	锁模式：0（None）、1（null）、2（row share）、3（row exclusive）、4 （share）、5（share row exclusive）和 6（exclusive）
REQUEST	NUMBER	申请的锁模式：具体值同上面的 LMODE
CTIME	NUMBER	已持有或等待锁的时间
BLOCK	NUMBER	是否阻塞其他锁申请

其中在 TYPE 字段的取值中，这里只关心 TM 和 TX 两种 DML 锁类型。

关于 ID1 和 ID2，TYPE 取值不同其含义也有所不同，具体如表 32-4 所示。

表 32-4　　v$lock 视图中 ID1 与 ID2 字段取值说明

TYPE	ID1	ID2
TM	被修改表的标识（object_id）	0
TX	以十进制数值表示该事务所占用的回滚段号与该事务在该回滚段的事务表（Transaction table）中所占用的槽号（slot number，可理解为记录号）。其组成形式为： 0xRRRRSSSS (RRRR = RBS number, SSSS = slot)	以十进制数值表示环绕（wrap）次数，即该槽（slot）被重用的次数

2. v$locked_object 视图

v$locked_object 视图列出了当前系统中哪些对象正被锁定，其主要字段说明如表 32-5 所示。

表 32-5 v$locked_object 视图字段说明

字段名称	类 型	说 明
XIDUSN	NUMBER	回滚段号
XIDSLOT	NUMBER	槽号
XIDSQN	NUMBER	序列号
OBJECT_ID	NUMBER	被锁对象标识
SESSION_ID	NUMBER	持有锁的会话（session）标识
ORACLE_USERNAME	VARCHAR2（30）	持有该锁的用户的 Oracle 用户名
OS_USER_NAME	VARCHAR2（15）	持有该锁的用户的操作系统用户名
PROCESS	VARCHAR2（9）	操作系统的进程号
LOCKED_MODE	NUMBER	锁模式，取值同表 32-3 中的 LMODE

3. v$session 视图

该视图列出了当前会话的信息。如果当前的某个会话因为某数据行被阻塞，该数据行的位置信息可以通过视图中的 ROW_WAIT_OBJ#（该行所在的数据对象标识）、ROW_WAIT_FILE#（该行所在的数据文件标识）、ROW_WAIT_BLOCK#（该行所在的数据块标识）、ROW_WAIT_ROW#（该行在数据块中的位置标识）等字段获得。另外，这几个字段只有在该会话等待其他事务结束时并且 ROW_WAIT_OBJ#的取值不为-1 时才有意义。

32.3.2 监控脚本

根据上述系统视图，可以编制脚本来监控数据库中锁的状况。

1. showlock.sql

第一个脚本 showlock.sql，该脚本通过连接 v$locked_object 与 all_objects 两视图，显示哪些对象被哪些会话锁住：

```
/* showlock.sql */
column o_name format a10
column lock_type format a20
column object_name format a15
select rpad(oracle_username,10) o_name,session_id sid,
   decode(locked_mode,0,'None',1,'Null',2,'Row share',
   3,'Row Exclusive',4,'Share',5,'Share Row Exclusive',6,'Exclusive') lock_type,
   object_name ,xidusn,xidslot,xidsqn
from v$locked_object,all_objects
where v$locked_object.object_id=all_objects.object_id;
```

2. showalllock.sql

第二个脚本 showalllock.sql，该脚本主要显示当前所有 TM 和 TX 锁的信息：

```
/* showalllock.sql */
select sid,type,id1,id2,
       decode(lmode,0,'None',1,'Null',2,'Row share',
       3,'Row Exclusive',4,'Share',5,'Share Row Exclusive',6,'Exclusive')
       lock_type,request,ctime,block
from v$lock
where TYPE IN('TX','TM');
```

3. showalllock.sql

第三个脚本 showlockorder.sql，在第二个脚本的基础上，将 type、id1、id2 等字段合并为一个 resource 字段，并按该字段与 ctime 时间字段排序（ctime 为逆序），这样有利于观察获得与申请锁的顺序关系。

```
/* showlockorder.sql */
break on resource
column sid format 99999
column resource format a15
column request format a15
select type||'-'||id1||'-'||id2 "resource",sid,
       decode(lmode,0,'None',1,'Null',2,'Row share',
       3,'Row Exclusive',4,'Share',5,'Share Row Exclusive',6,'Exclusive') lock_type,
       decode(request,0,'None',1,'Null',2,'Row share',
       3,'Row Exclusive',4,'Share',5,'Share Row Exclusive',6,'Exclusive') request,
       ctime,block
from v$lock
where TYPE IN('TX','TM')
order by "resource",ctime desc
/
clear breaks
```

32.4　Oracle 多粒度封锁机制示例

以下示例 32.4.1 节～32.4.4 节下的“1.在 Oracle 8i 上”运行在 Oracle 8.1.7 上，示例 32.4.4 节下的“2.在 Oracle 9i 上”～4.11 节运行在 Oracle 9.2.0 上，数据库版本不同，其输出结果也可能有所不同。首先建立 4 个会话，其中三个（以下用 SESS#1、SESS#2、SESS#3 表示）以 SCOTT 用户连入数据库，以操作 Oracle 提供的示例表（DEPT、EMP）；另一个（以下用 SESS#0 表示）以 SYS 用户连入数据库，用于监控。

32.4.1　操作同一行数据引发的锁阻塞

```
SESS#1:
SQL> select * from dept for update;
   DEPTNO DNAME         LOC
---------- -------------- -------------
       10 account       70
       20 research      8
       30 sales         8
```

```
       40 operations    8
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT        17 Row share      DEPT              8      2   5861
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   17 TX    524290      5861 Exclusive             0       761         0
   17 TM     32970         0 Row share             0       761         0
```

如第一个脚本 showlock 所示，执行完 select…for update 语句后，SESS#1（SID 为 17）在 DEPT 表上获得 Row Share 锁；如第二个脚本 showalllock 所示，SESS#1 获得的 TX 锁为 Exclusive，这些都验证了上面的理论分析。另外，可以将 TX 锁的 ID1 按如下方法进行分解：

```
SQL> select trunc(524290/65536) xidusn,mod(524290,65536) xidslot from dual;
XIDUSN XIDSLOT
------ -------
     8       2
```

分解结果与第一个脚本直接查出来的 XIDUSN 与 XIDSLOT 相同，而 TX 锁的 ID2（5861）与 XIDSQN 相同，可见当 LOCK TYPE 为 TX 时，ID1 实际上是该事务所占用的回滚段段号与事务表中的槽（SLOT）号的组合，ID2 即为该槽被重用的次数，而这三个值实际上可以惟一地标识一个事务，即 TRANSACTION ID，这三个值也可以从系统表 v$transaction 中查到。

另外，DEPT 表中有 4 条记录被锁定，但 TX 锁只有 1 个，这也与上面的理论分析一致。继续进行操作：

```
SESS#2:
SQL> update dept set loc=loc where deptno=20;
```

该更新语句被阻塞，此时再查看系统的锁情况：

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT        17 Row share      DEPT              8      2   5861
SCOTT        19 Row Exclusive   DEPT              0      0      0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   17 TX    524290      5861 Exclusive             0      3462         1
   17 TM     32970         0 Row share             0      3462         0
   19 TM     32970         0 Row Exclusive         0         7         0
   19 TX    524290      5861 None                  6         7         0
```

在 DEPT 表上除了 SESS#1（SID 为 17）持有 Row Share 锁外，又增加了 SESS#2（SID 为 19）持有的 Row Exclusive 锁，但还没有为 SESS#2 分配回滚段（XIDUSN、XIDSLOT、XIDSQN 的值均为 0）；而从第二个脚本看到，SESS#2 的 TX 锁的 LOCK_TYPE 为 None，其申请的锁类型（REQUEST）为 6（即 Exclusive），而其 ID1、ID2 的值与 SESS#1 所持有的 TX 锁的 ID1、ID2 相同，SESS#1 的 TX 锁的阻塞域（BLOCK）为 1，这就说明了由于 SESS#1 持有的 TX 锁，阻塞了 SESS#2 的更新操作（SESS#2 所更新的行与 SESS#1 所锁定的行相冲突）。还可以看出，SESS#2 先申请表级的 TM 锁，后申请行

（事务）级的 TX 锁，这也与前面的理论分析一致。

下面，将 SESS#1 的事务进行回滚，解除对 SESS#2 的阻塞，再对系统进行监控。

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT        19 Row Exclusive   DEPT              2     10   5803
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   19 TX    131082      5803 Exclusive             0       157        0
   19 TM     32970         0 Row Exclusive         0       333        0
```

可以看到，SESS#1 的事务所持有的锁已经释放，系统为 SESS#2 的事务分配了回滚段，而其 TX 锁也已经获得，并且 ID1、ID2 是其真正的 Transaction ID。再将会话 2 的事务进行回滚。

```
SESS#2:
SQL> rollback;
Rollback complete.
```

检查系统锁的情况：

```
SESS#0:
SQL> @showlock
no rows selected
SQL> @showalllock
no rows selected
```

可以看到，TM 与 TX 锁已全部被释放。

32.4.2　实体完整性引发的锁阻塞

DEPT（部门）表有如下字段 DEPTNO（部门编号）、DNAME（部门名称）、LOC（部门位置），其中 DEPTNO 列为主键。

```
SESS#1
SQL> INSERT INTO DEPT(DEPTNO) VALUES(50);
1 row created.
SESS#0
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         7 Row Exclusive   DEPT              6     88    29
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TX    393304        29 Exclusive             0         6         0
    7 TM      3574         0 Row Exclusive         0         6         0
```

向 DEPT 表中插入一条 DEPTNO 为 50 的记录后，SESS#1（SID 为 7）在 DEPT 表上获得 Row Exclusive 锁，并且由于进行了数据插入，该事务被分配了回滚段，获得 TX 锁。

```
SESS#2
INSERT INTO DEPT(DEPTNO) VALUES(50);
```

这时，SESS#2（SID 为 8）也向 DEPT 表中插入一条 DEPTNO 为 50 的记录，该语句被阻塞，检查锁情况：

```
SESS#0
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         8 Row Exclusive   DEPT              7     75    30
SCOTT         7 Row Exclusive   DEPT              6     88    29
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TX    393304        29 Exclusive              0        92         1
    7 TM      3574         0 Row Exclusive          0        92         0
    8 TX    458827        30 Exclusive              0        22         0
    8 TM      3574         0 Row Exclusive          0        22         0
    8 TX    393304        29 None                   4        22         0
```

SESS#2 在 DEPT 表上也获得了 Row Exclusive 锁，同样也获得了回滚段的分配，得到 TX 锁，但是由于其插入的记录与 SESS#1 插入的记录的 DEPTNO 均为 50，该语句成功与否取决于 SESS#1 的事务是提交还是回滚，所以 SESS#2 被阻塞，表现为 SESS#2 以 Share 方式（REQUEST=4）等待 SESS#1 所持有的 TX 锁的释放。

这时，如果 SESS#1 进行回滚：

```
SESS#1
SQL> ROLLBACK;
Rollback complete.
SESS#2
1 row created.
SESS#0
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME       XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         8 Row Exclusive   DEPT                7      75     30
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    8 TX    458827        30 Exclusive              0       136         0
    8 TM      3574         0 Row Exclusive          0       136         0
```

SESS#2 的阻塞将被解除，SESS#2 只持有原先已有的 TM 与 TX 锁，其等待的 TX 锁（由 SESS#1 持有）也消失了。

如果 SESS#1 提交而不是回滚，在 SESS#2 上将会出现如下提示：

```
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.PK_DEPT) violated 错误。
```

即发生主键冲突，SESS#1 与 SESS#2 的所有锁资源均被释放。

32.4.3 参照完整性引发的锁阻塞

EMP（员工）表有如下字段：EMPNO（员工编号），ENAME（员工姓名），DEPTNO（员工所

在部门编号），其中 DEPTNO 列为外键，其父表为 DEPT。

```
SESS#1
SQL> insert into dept(deptno) values(60);
1 row created.
SESS#0
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- -------------- -------------- ------ ------- ------
SCOTT         7 Row Exclusive   DEPT              2      6    33
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TX     131078         33 Exclusive              0        148         0
    7 TM       3574          0 Row Exclusive          0        148         0
```

SESS#1（SID 为 7）在 DEPT 表中先插入一条 DEPTNO 为 60 的记录，SESS#1 获得了 DEPT 表上的 Row Exclusive 锁及一个 TX 锁。

```
SESS#2
insert into emp(empno,deptno) values(2000,60);
```

被阻塞：

```
SESS#0
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- -------------- -------------- ------ ------- ------
SCOTT         7 Row Exclusive   DEPT              2      6    33
SCOTT         8 Row Exclusive   EMP               3     20    31
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TX     131078         33 Exclusive              0        228         1
    7 TM       3574          0 Row Exclusive          0        228         0
    8 TX     196628         31 Exclusive              0          9         0
    8 TM       3576          0 Row Exclusive          0          9         0
    8 TX     131078         33 None                   4          9         0
```

SESS#2（SID 为 8）向 EMP 表中出入一条新记录，该记录 DEPT 值为 60（即 SESS#1 刚插入，但还未提交的记录的 DEPTNO 值），SESS#2 获得了 EMP 表上的 Row Exclusive 锁，另外由于插入记录，还分配了回滚段及一个 TX 锁，但由于 SESS#2 的插入语句是否成功取决于 SESS#1 的事务是否进行提交，所以它被阻塞，表现为 SESS#2 以 Share（REQUEST=4）方式等待 SESS#1 释放其持有的 TX 锁。这时 SESS#1 如果提交，SESS#2 的插入也将执行成功，而如果 SESS#1 回滚，由于不符合参照完整性，SESS#2 将报错：

```
SESS#2
insert into emp(empno,deptno) values(2000,60)
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_DEPTNO) violated - parent key not
Found
```

SESS#2 持有的锁也被全部释放。

32.4.4 外键未加索引引发的锁阻塞

1. 在 Oracle 8i 上

EMP 表上的 DEPTNO 列为外键，但没有在该列上建索引。

```
SESS#1
SQL> delete emp where 0=1;
0 rows deleted.
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         7 Row Exclusive   EMP               0      0     0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TM      3576         0 Row Exclusive           0        10         0
```

首先 SESS#1（SID 为 7）做了一个删除操作，但由于条件（0=1）为永假，所以实际上并没有一行被删除，从监控脚本可以看出 SESS#1 在 EMP 表上获得 Row Exclusive 锁，但由于没有实际的行被删除，所以并没有 TX 锁，也没有为 SESS#1 分配回滚段。

```
SESS#2:
SQL> delete dept where 0=1;
```

该语句虽然也不会删除实际数据，但却被阻塞，查看系统的锁情况：

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         8 None           EMP               0      0     0
SCOTT         7 Row Exclusive   EMP               0      0     0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TM      3576         0 Row Exclusive           0        31         1
    8 TM      3576         0 None                    4        12         0
```

SESS#2 申请在 EMP 表上加 Share 锁（REQUEST=4），但该申请被 SESS#1 阻塞，因为 SESS#1 已经在 EMP 表上获得了 Row Exclusive 锁，与 Share 锁不相容。

下面对 SESS#1 进行回滚后，再进行监控。

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- --------------- --------------- ------ ------- ------
SCOTT         8 Share          EMP               0      0     0
SCOTT         8 Row Exclusive   DEPT              0      0     0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    8 TM      3574         0 Row Exclusive           0        16         0
    8 TM      3576         0 Share                   0        16         0
```

SESS#2 在 EMP 表上获得 Share 锁后，又在 DEPT 表上获得 Row Exclusive 锁，由于没有实际的行被修改，SESS#2 并没有获得 TX 锁。

在 Oracle 8 中，如果子表的外键上没有加索引，当在父表上删除记录时，会先在子表上申请获得 Share 锁，之后再在父表上申请 Row Exclusive 锁。由于表级 Share 锁的封锁粒度较大，所以容易引起阻塞，从而造成性能问题。

当在外键上建立索引后，在父表上删除数据将不再对子表上加 Share 锁，如下所示：

```
SESS#1:
SQL> create index i_emp_deptno on emp(deptno);
Index created.
SQL> delete dept where 0=1;
0 rows deleted.
SQL>
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME    XIDUSN XIDSLOT XIDSQN
---------- ----- -------------- -------------- ------ ------- ------
SCOTT         7 Row Exclusive   DEPT              0      0     0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
    7 TM      3574         0 Row Exclusive           0         9         0
```

可以看到，在 EMP 表 DEPTNO 列上建立索引后，在 DEPT 表上执行 delete 操作，不再要求在 EMP 表上加 Share 锁，只是在 DEPT 表上加 Row Exclusive 锁，封锁的粒度减小，引起阻塞的可能性也减小。

2. 在 Oracle 9i 上

在 Oracle 9i 中，如果子表外键所在的列没有加索引，那么在父表上对主键进行更新或删除操作时，仍然要对子表加 Share 锁，但是与 8i 不同的是，该锁在获得之后将会马上释放。如果有多个主键被更新或删除，那么对于每一行，都会有一个获得并释放锁的过程。如下面的例子所示：

```
SESS#1
SQL> delete emp where 0=1;
0 rows deleted.
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- -------------- -------------- ------- -------- -------
SCOTT        10 Row share      DEPT               0       0      0
SCOTT        10 Row Exclusive   EMP               0       0      0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   10 TM     30139         0 Row Exclusive           0       406         0
   10 TM     30137         0 Row share               0       406         0
```

与 8i 不同的是，SESS#1（SID 为 10）除了在 EMP 表上获得 Row Exclusive 锁外，还在其父表 DEPT 上申请获得 Row Share 锁。

```
SESS#2:
SQL> delete dept where 0=1;
```

该语句虽然也不会删除实际数据，但却被阻塞，查看系统的锁情况：

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- -------------- --------------- ------- -------- -------
SCOTT        10 Row share      DEPT               0       0      0
SCOTT        10 Row Exclusive   EMP               0       0      0
SCOTT        12 Row Exclusive   DEPT              0       0      0
SCOTT        12 None           EMP                0       0      0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   10 TM     30139          0 Row Exclusive           0       743         1
   10 TM     30137          0 Row share               0       743         0
   12 TM     30139          0 None                    4         9         0
   12 TM     30137          0 Row Exclusive           0         9         0
```

与 8i 不同的是，SESS#2（SID 为 12）先在 DEPT 表上申请获得 Row Exclusive 锁，再申请在 EMP 表上加 Share 锁（REQUEST=4），但该申请被 SESS#1（SID 为 10）阻塞，因为 SESS#1 已经在 EMP 表上获得了 Row Exclusive 锁，与 Share 锁不相容。

下面对 SESS#1 进行回滚后，再进行监控。

```
SESS#0:
SQL> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- -------------- --------------- ------- -------- -------
SCOTT        12 Row Exclusive   DEPT              0       0      0
SQL> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   12 TM     30137          0 Row Exclusive           0       507         0
```

这时，SESS#2（SID 为 12）只保留了在 DEPT 上的 Row Exclusive 锁，其申请的 Share 锁已经释放。对该锁的获得与释放是通过如下机制实现的：(1) 事务建立一个保留点（savepoint）；(2) 申请获得 Share 锁；(3) rollback to savepoint，释放 Share 锁。关于 rollback to savepoint，在下面的例子专门介绍。

32.4.5 部分回滚对锁的影响

如果一个事务进行部分回滚（rollback to savepoint），由于会撤消部分修改，Oracle 会根据情况释放或降低某些 TM 表级锁，而 TX 锁一般不会释放，而被 TX 锁阻塞的事务也不会继续执行（即使那些引发阻塞的行上的修改已经被回滚）。如下所示：

```
SESS#1> savepoint a;
Savepoint created.
SESS#1> select * from dept where deptno=10 for update;
   DEPTNO DNAME         LOC
---------- -------------- -------------
       10 ACCOUNTING     NEW YORK
SESS#0> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
```

```
---------- ----- -------------- -------------- ------- -------- -------
SCOTT        13 Row share      DEPT               8      18    436
SESS#0> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   13 TX    524306       436 Exclusive              0         7         0
   13 TM     30137         0 Row share              0         7         0
```

在 SESS#1（SID 为 13）的事务开始便设置一个保留点 a，然后执行一个 select…for update 语句，则该事务在 DEPT 表上获得 Row Share 锁，并获得一个 TX 锁。继续执行：

```
SESS#1>  savepoint b;
Savepoint created.
SESS#1> update emp set job=job where empno=7369;
1 row updated.
SESS#0> @showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- -------------- -------------- ------- -------- -------
SCOTT        13 Row share      DEPT               8      18    436
SCOTT        13 Row Exclusive   EMP               8      18    436
SESS#0> @showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   13 TX    524306       436 Exclusive              0        83         0
   13 TM     30139         0 Row Exclusive           0         9         0
   13 TM     30137         0 Row share               0        83         0
```

SESS#1 在设置保留点 b 后又在 EMP 表上做一个更新操作，其在 EMP 表上获得 Row Exclusive 锁。

```
SESS#2>update dept set loc=loc where deptno=10;
SESS#3>lock table emp in share mode;
```

上面两个操作均被阻塞。

```
SESS#0>@showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- -------------- -------------- ------- -------- -------
SCOTT        13 Row share      DEPT               8      18    436
SCOTT        13 Row Exclusive   EMP               8      18    436
SCOTT        10 Row Exclusive   DEPT              0       0      0
SCOTT        23 None           EMP                0       0      0
SESS#0>@showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   10 TM     30137         0 Row Exclusive           0       205         0
   10 TX    524306       436 None                   6       205         0
   13 TX    524306       436 Exclusive              0       408         1
   13 TM     30139         0 Row Exclusive           0       334         1
   13 TM     30137         0 Row share               0       408         0
   23 TM     30139         0 None                    4       191         0
```

在 SESS#2（SID 为 10）在 DEPT 表获得 Row Exclusive 锁，但其修改被 SESS#1（SID 为 13）的 TX 锁所阻塞；SESS#3（SID 为 23）要在 EMP 表申请 Share 锁，但其申请被 SESS#1 在其上的 Row Exclusive 锁所阻塞。

```
SESS#1>rollback to b;
```

```
Rollback complete.
SESS#0>@showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- --------------- --------------- ------- -------- -------
SCOTT        13 Row share      DEPT                8      18    436
SCOTT        10 Row Exclusive   DEPT               0       0      0
SCOTT        23 Share          EMP                 0       0      0
SESS#0>@showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   10 TM     30137         0 Row Exclusive           0      1052         0
   10 TX    524306       436 None                    6      1052         0
   13 TX    524306       436 Exclusive               0      1255         1
   13 TM     30137         0 Row share               0      1255         0
   23 TM     30139         0 Share                   0       123         0
```

在 SESS#1（SID 为 13）在回滚到保留点 b 后，释放掉在 EMP 表上的 Row Exclusive 锁，从而使 SESS#3 获得在 EMP 上的 Share 锁。

```
SESS#1>rollback to a;
Rollback complete.
SESS#0>@showlock
O_NAME      SID LOCK_TYPE      OBJECT_NAME     XIDUSN  XIDSLOT  XIDSQN
---------- ----- --------------- --------------- ------- -------- -------
SCOTT        10 Row Exclusive   DEPT               0       0      0
SCOTT        23 Share          EMP                 0       0      0
SESS#0>@showalllock
  SID TY       ID1       ID2 LOCK_TYPE         REQUEST     CTIME     BLOCK
----- -- ---------- ---------- --------------- ---------- ---------- ----------
   10 TM     30137         0 Row Exclusive           0      1396         0
   10 TX    524306       436 None                    6      1396         0
   13 TX    524306       436 Exclusive               0      1599         1
   23 TM     30139         0 Share                   0       467         0
```

在 SESS#1（SID 为 13）继续回滚到保留点 a（在保留点 a 之前该事务并没有任何 DML 操作）后，其释放了在 DEPT 上的 Row Share 锁，但仍然保留 TX 锁，而 SESS#2（SID 为 10）仍然继续被阻塞，而这时如果其他 session 对 DEPTNO 为 10 的记录进行修改将不再会被阻塞，如下所示：

```
SESS#3>update dept set loc=loc where deptno=10;
1 row updated.
```

可见，TX 锁只有等到事务完全结束时才被释放，尽管由于 rollback to savepoint 会使 TX 锁所对应的修改行有所变化，而被其阻塞的其他事务不会因为这种变化而得以继续执行，直至事务完全结束。另外这也再一次证实，TX 锁的本意还是事务锁，而非“行”锁（虽然有时候这两者可以混用），虽然最初造成 TX 锁阻塞的原因可能是对于某行的修改冲突，但如果事务不结束，即使冲突事实上已经不存在，但被阻塞的事务仍将等待 TX 锁的释放。

32.4.6 锁的排队机制

许多 Oracle 锁又被称为“enqueue locks”。从英文一般的构词规则来看，enqueue 应该是动词（翻译成“排队”或“加入队列”），但在这里，经常被用做形容词（可以翻译成“队列式的”）、甚至是名词（可以翻译成“队列”）。计算机软件中的“锁”往往是用来解决稀缺资源的争用问题，

而此种问题更完善的解决还会用到排队机制（如同人类社会解决稀缺资源争用问题一样，比如众多顾客到银行办理各种业务），所以“锁”（lock）与“队列”（enqueue）这两个词在这一领域的某些场合可以互换。下面还是看一个具体的例子：

```
SESS#1>update dept set loc=loc where deptno=10;
SESS#2> lock table dept in share mode;
```

该事务被阻塞，采用第三个监控脚本。

```
SESS#0>@showlockorder
resource          SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0         10 Row Exclusive      None                227     1
                  12 None              Share                181     0
TX-196618-524      10 Exclusive         None                227     0
```

在 SESS#1（SID 为 10）在 DEPT 表上获得 Row Exclusive 锁，而 SESS#2（SID 为 12）在 DEPT 表上申请 Share 锁，被 SESS#1 阻塞。

```
SESS#3>update dept set loc=loc where deptno=20;
```

该事务也被阻塞。

```
SESS#0>@showlockorder
resource          SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0         10 Row Exclusive      None                357     1
                  12 None              Share                311     0
                  13 None              Row Exclusive          3     0
TX-196618-524      10 Exclusive         None                357     0
```

SESS#3（SID 为 13）申请 DEPT 表的 Row Exclusive 锁，虽然与 SESS#1 所持有的 Row Exclusive 锁相容，但由于 SESS#2（SID 为 12）在之前已被阻塞，SESS#3 也被阻塞。

```
SESS#1>rollback;
SESS#0>@showlockorder
resource          SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0         13 None              Row Exclusive        635     0
                  12 Share             None                  56     1
```

SESS#1 回滚后，SESS#2（SID 为 12）获得 Share，SESS#3（SID 为 13）申请的 Row Exclusive 锁被其阻塞。

由此例可以看出，一旦某一事务阻塞于某一资源（如上例的 SESS#2），其后向该资源新申请锁的事务都将被阻塞，而不论新申请的封锁类型是否相容于已获得的封锁类型。

表面上看，这样的处理方法好像不是很合理，但深入分析一下，这样做是有道理的，它主要是要避免数据库理论中所谓的“活锁”问题。所谓“活锁”，是指：如果事务 T_1 封锁了数据 D，事务 T_2 又请求封锁 D，于是 T_2 等待。T_3 也请求封锁 D，当 T_1 释放了 D 上的封锁之后系统首先批准了 T_3 的请求，T_2 仍然等待。然后 T_4 又请求封锁 D，当 T_3 释放了 D 上的封锁后系统又批准了 T_4 的请求……则 T_2 可能永远等待，这就是所谓的“活锁”。避免“活锁”的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，系统按请求的先后顺序对事务排队，数据对象上的锁一旦释放就批准申请队列中第 1 个事务获得锁。

可见，Oracle 也采用了上述原则，否则的话，在有大量并发事务的情况下，高强度的锁（如本例 SESS#2 申请表级的 Share 锁）申请有可能长时间无法得到满足。

32.4.7 ITL Slot 不足引发的锁阻塞

在 32.2.1 节已经说明，块首的 ITL（相关事务列表）记录了当前修改该块的事务，而 ITL 列表的最大长度（或称为项数，即 slot），由定义数据对象时指定的 MAXTRANS 参数决定，当并发修改的事务超过该参数的限制，则会出现事务的阻塞。

```
SESS#1>CREATE TABLE SCOTT.try_itl
  2  (
  3    c1 NUMBER
  4  )
  5  INITRANS   1
  6  MAXTRANS   2;
Table created.
SESS#1>insert into try_itl values(1);
1 row created.
SESS#1>insert into try_itl values(2);
1 row created.
SESS#1>insert into try_itl values(3);
1 row created.
SESS#1>commit;
SESS#1>select c1,dbms_rowid.rowid_relative_fno(rowid) as in_file,
  2  dbms_rowid.rowid_block_number(rowid) as in_block,
  3  dbms_rowid.rowid_row_number(rowid) as row_num
  4  from try_itl;
       C1    IN_FILE   IN_BLOCK    ROW_NUM
---------- ---------- ---------- ----------
        1         1     50794         0
        2         1     50794         1
        3         1     50794         2
```

首先建立一个实验用表 try_itl，其 MAXTRANS 为 2，即每块的最大并发事务数为 2，再插入 3 条数据，并验证这 3 条数据确实位于同一个数据块中。

```
SESS#1>delete try_itl where c1=1;
1 row deleted.
SESS#2>delete try_itl where c1=2;
1 row deleted.
SESS#3>delete try_itl where c1=3;
```

该事务被阻塞。用第三个监控脚本进行监控：

```
SESS#0>@showlockorder
resource          SID LOCK_TYPE          REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30917-0         10 Row Exclusive      None                 257     0
                   12 Row Exclusive      None                 254     0
                   13 Row Exclusive      None                 251     0
TX-524310-565      12 Exclusive          None                 254     1
                   13 None               Share                245     0
TX-655390-596      10 Exclusive          None                 257     0
```

三个事务均获得了 TM 锁，但 SESS#3（SID 为 13）由于 ITL 项不足，被阻塞，从监控的结果看，其以申请 Share 锁的方式被 SESS#2（SID 为 12）的 TX 锁阻塞，所以之后即使 SESS#1（SID

为 10）的事务结束，释放一个 ITL 项对 SESS#3 也不会起作用，它只会等待 SESS#2 的事务结束，如下所示，SESS#1 回滚之后，SESS#3 仍被阻塞。

```
SESS#1>rollback;
Rollback complete.
SESS#0>@showlockorder
resource          SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30917-0         12 Row Exclusive      None               1032     0
                   13 Row Exclusive      None               1029     0
TX-524310-565      12 Exclusive          None               1032     1
                   13 None               Share              1023     0
```

32.4.8　Bitmap 索引引发的锁阻塞

位图（Bitmap）索引比较适合于那些取值较为单一（low-cardinality，低基数）的列，如“性别”、“职位”等。每个位图索引被分为多个位图段（Bitmap Segment），而对于位图索引的更新，其封锁粒度在位图段一级，而一个位图段会对应多个数据行，这样对于频繁更新的表，位图索引就会加大发生锁阻塞的可能性，从而降低系统效率。

```
SESS#1>CREATE BITMAP INDEX SCOTT.idx_emp_job  ON SCOTT.EMP(JOB) TABLESPACE INDX;
SESS#1>update emp set job='ANALYST' where empno=7369;
1 row updated.
SESS#2>update emp set job='CLERK' WHERE EMPNO=7499;
```

该事务被阻塞。用第三个监控脚本进行监控：

```
SESS#0>@showlockorder
resource          SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30139-0         10 Row Exclusive      None                156     0
                   12 Row Exclusive      None                 74     0
TX-262180-614      10 Exclusive          None                156     1
                   12 None               Share                74     0
TX-589833-613      12 Exclusive          None                 74     0
```

SESS#1（SID 为 10）与 SESS#2（SID 为 12）的事务都获得了各自的 TM 与 TX 锁，但由于位图索引段冲突，SESS#2 被阻塞，表现为 SESS#2（SID 为 12）以 Share 方式等待 SESS#1（SID 为 10）释放其持有的 TX 锁。SESS#1 回滚后，SESS#2 也执行成功。

位图索引的更新代价比较高，且易于发生锁阻塞，所以其比较适用于静态表，如决策支持系统，而不适用于 OLTP 系统。

32.4.9　死锁分析

在互联网的相关技术论坛甚至有些公开发表的文章中，经常可以看到有些朋友将上述列举的“锁阻塞”（或称“锁等待”）的情况称为“死锁”，这是不正确的。

所谓“死锁”，是指两个或两个以上的用户互相等待对方持有的锁的情况。Oracle 能够自动检测死锁的情况，并通过回滚其中一个事务的操作来解决死锁问题。

```
SESS#1>select * from dept where deptno=10 for update;
   DEPTNO DNAME        LOC
```

```
---------- -------------- -------------
      10 ACCOUNTING    NEW YORK
SESS#2>select * from dept where deptno=20 for update;
   DEPTNO DNAME        LOC
---------- -------------- -------------
      20 RESEARCH      DALLAS
SESS#1>update dept set loc=loc where deptno=20;
```

该事务被阻塞。

```
SESS#2>update dept set loc=loc where deptno=10;
```

该事务也被阻塞，快速执行监控脚本：

```
SESS#0>@showlockorder
resource         SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0         10 Row Exclusive       None                32     0
                   12 Row Exclusive       None                 0     0
TX-262189-628      10 Exclusive           None                54     1
                   12 None                Exclusive            0     0
TX-458799-615      12 Exclusive           None                47     1
                   10 None                Exclusive           32     0
```

这时，发现 SESS#1（SID 为 10）所持有的 TX 锁阻塞了 SESS#2（SID 为 12），而 SESS#2 所持有的 TX 锁也阻塞了 SESS#1，即发生了死锁。

这种状况不会持续很久，过一会儿发现 SESS#1 出现错误提示：

```
SESS#1>update dept set loc=loc where deptno=20;
update dept set loc=loc where deptno=20
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
```

监控此时锁的情况：

```
SESS#0>@showlockorder
resource         SID LOCK_TYPE           REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0         12 Row Exclusive       None               610     0
                   10 Row share           None               608     0
TX-262189-628      10 Exclusive           None               664     1
                   12 None                Exclusive          610     0
TX-458799-615      12 Exclusive           None               657     0
```

可以发现，SESS#1（SID 为 10）已不再等待 SESS#2（SID 为 12）了，Oracle 自动回滚了 SESS#1 引发死锁的操作，但并没有回滚整个事务，所以 SESS#2 还处于阻塞状态。要想使 SESS#2 继续执行，必须将 SESS#1 的事务提交或回滚。

由于 Oracle 主要的封锁粒度在行级，所以发生死锁的情况并不多见，一旦发生死锁，Oracle 会将相关信息记录在 alert 文件及 trace 文件中。

在应用程序开发中，下述做法有利于减少死锁的发生：(1) 不同的程序员应尽量按照一定的顺序对不同的表依次加锁；(2) 首先申请高强度的锁，再申请低强度的锁。

32.4.10 表级锁的使能

在本章的开始已经提到，DML 锁的目的在于保证在多个并发用户情况下的数据完整性。DML

锁防止并发的 DML 与 DDL 操作互相产生破坏性的影响。DML 锁又分为 TM（表级）锁与 TX（事务、行）锁。

实际上，可以通过设置 DML_LOCKS 初始化参数为 0 取消数据库获得 TM 表级锁的能力，这时候数据库的数据封锁粒度实际只有一级，即行级。数据修改仍能进行，但是会话只能获得 TX 锁，而不能获得 TM 锁，由于没有 TM 锁的保护，任何的 DDL 语句均被禁止（为了执行 DDL 而检查表中各行是否被加锁的做法对系统的影响恐怕是不可取的）。在 OPS（RAC）环境下，获得 TM 实例锁的开销远大于单实例环境下获得 TM 锁的开销，所以在此种情况下将 DML_LOCKS 设置为 0，将使系统性能得到提升。

当然，将 DML_LOCKS 设置为 0 将会给数据库的维护工作带来很多麻烦，用户会因为要重建一个索引而不得不修改参数并且关闭/启动数据库两次。所以，用户更愿意对某些表进行单独的设置，使其失去获得 TM 锁的能力，Oracle 提供了这样的功能，如下所示。

```
SESS#1>select table_name,table_lock from user_tables where table_name='DEPT';
TABLE_NAME                 TABLE_LO
------------------------------ --------
DEPT                       ENABLED
SESS#1>alter table dept disable table lock;
Table altered.
SESS#1>select table_name,table_lock from user_tables where table_name='DEPT';
TABLE_NAME                 TABLE_LO
------------------------------ --------
DEPT                       DISABLED
```

可以通过 alter table *table_name* disable table lock 语句禁止某个表获得 TM 锁的能力，其状态可以通过 user_tables 表中的 table_lock 字段查知。从某种意义上讲，这也可以理解为在该表上加了一个特殊的锁（用于阻止其他 TM 锁与 DDL 锁），并且该锁不会因为事务的结束而释放。

```
SESS#1>lock table dept in share mode;
lock table dept in share mode
         *
ERROR at line 1:
ORA-00069: cannot acquire lock -- table locks disabled for DEPT
SESS#1>drop index idx_dept_loc;
drop index idx_dept_loc
         *
ERROR at line 1:
ORA-00069: cannot acquire lock -- table locks disabled for DEPT
```

这时候，任何的表级封锁及 DDL 语句都被禁止（除了使其恢复 TM 锁的 DDL）。

```
SESS#1>update dept set loc=loc where deptno=10;
1 row updated.
SESS#0>@showlockorder
resource         SID LOCK_TYPE         REQUEST            CTIME  BLOCK
---------------- ------ ------------------ --------------- ---------- ------
TX-393224-674      10 Exclusive          None                  3     0
```

DML 修改仍可以进行，但 session 只获得 TX 锁，不再获得 TM 锁。可以通过下面的 DDL 使 DEPT 表恢复获得 TM 锁的能力。

```
SESS#1>alter table dept ENABLE table lock;
Table altered.
```

32.4.11 row_locking 参数

Oracle 初始化参数 row_locking 的缺省值为 ALWAYS，该设置使 Oracle 在执行 DML 语句时，尽量使用低级别与低强度的封锁，从而增大系统的并发性。该参数的另一个取值为 intent，它将使 Oracle 在执行 DML（除 select for update 外）语句时，使用更高强度的封锁。参看下面的例子。

```
SESS#0>alter system set row_locking=intent scope=spfile;
System altered.
```

修改 row_locking 初始化参数取值为 intent，关闭并重启数据库，各 session 进行重新连接。

```
SESS#1>select * from dept where deptno=10 for update;
   DEPTNO DNAME        LOC
---------- -------------- -------------
       10 ACCOUNTING    NEW YORK
SESS#0>@showlockorder
resource         SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0        10 Row share         None                 5     0
TX-327706-694      10 Exclusive         None                 5     0
```

对于 select for update 语句，其加锁的情况并没有变化。

```
SESS#1>update dept set loc=loc where deptno=10;
1 row updated.
SESS#0>@showlockorder
resource         SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0        10 Share Row Exclusive None                6     0
TX-327706-694      10 Exclusive         None               256     0
```

而对于其他 DML 语句，其所加的 TM 锁类型由 Row Exclusive 变为 Share Row Exclusive，Row Exclusive 锁实际上是一个单纯的意向锁，其真正的封锁粒度还是在行级，而 Share Row Exclusive=Share+Row Exclusive，在原有锁强度的基础上加了一个表级的共享锁，封锁的真正粒度到了表级。而如果此时另一个 session 对该表做修改，将被阻塞。

```
SESS#2>update dept set loc=loc where deptno=0;
SESS#0>@showlockorder
resource         SID LOCK_TYPE         REQUEST            CTIME  BLOCK
--------------- ------ ------------------ --------------- ---------- ------
TM-30137-0        10 Share Row Exclusive  None               556     1
                  12 None                 Share Row Exclu      6     0
                                          sive
TX-327706-694      10 Exclusive           None               806     0
```

SESS#2（SID 为 12）对 DEPT 表申请的 Share Row Exclusive 锁被 SESS#1（SID 为 10）所阻塞，在此种情况下，对同一个表已不可能同时有两个事务进行修改，系统并发性大为降低。

32.5 Oracle 多粒度封锁机制总结

Oracle 通过具有意向锁的多粒度封锁机制进行并发控制，保证数据的一致性。其 DML 锁（数据锁）分为两个层次（粒度）：即表级和行级。通常的 DML 操作在表级获得的只是意向锁（RS 或

RX)，其真正的封锁粒度还是在行级；另外，在 Oracle 数据库中，单纯地读数据（select）并不加锁，这些都极大地提高了系统的并发程度。

在支持高并发度的同时，Oracle 利用意向锁及数据行上加锁标志位等设计技巧，减小了 Oracle 维护行级锁的开销，使其在数据库并发控制方面有着显著的优势。

参考信息

1．萨师煊、王珊. 数据库系统概论（第三版）[M]. 北京:高等教育出版社,2000

2．Oracle. Oracle 8i Concepts[M]. Oracle Corporation,1999

3．Oracle. Oracle 9i Database Concepts[M]. Oracle Corporation,2002

4．Steve Adams.Oracle8i Internal Services for waits,Latches,Locks,and Memory[M]. O'REILLY,1999

5．Thomas Kyte.export one—on—one Oracle[M].北京:清华大学出版社,2002

6．http://metalink.oracle.com/

作者简介

邢海捷，毕业于北京邮电大学，主要从事基于大型关系数据库的应用系统的研发，曾主持、参与多个电信计费帐务系统的开发、实施工作，并作为技术方面的主要负责人，组织实施了原中国电信本地电信业务计费帐务系统、中国联通 CDMA 专业计费子系统的统一检测工作。近两年主要从事多媒体消息业务中心（MMSC）的研发。本人长期使用 Oracle 数据库，对某些方面有较为深入的研究，希望能够与大家共同提高。

E—mail：jeffli73@sina.com

后　记

掩卷成书，刚好是我来到北京两周年的日子。不是刻意选择，碰巧就是这一天。想起两年前的这个日子，离开生活工作了多年的城市来到完全陌生的北京，开始了自己新的旅程。上午办完辞职手续，下午买好机票，晚上就到达北京，一切出乎意料的顺利。

然后在北京开始工作，再然后和北京共历“非典”，我想那段日子对于很多朋友必定都是终生难忘。后来 coolyl 和 biti_rainy 先后来到北京，三个人住在一起，我们家的实力空前壮大。用 4 个字来形容那段日子就是：黄金时代。美好的时光总是让人怀念，虽然短暂。

初到北京的时候，仅有的朋友都是通过 ITPUB 结识的，在最初以及随后的日子里，这些朋友给予我的帮助以及关心都是我难以忘怀的。所以常常心存感激，网络之于我们，并非虚幻。通过网络，结识了很多的朋友，很多人在生活里变得不可缺少；很多人虽然素未谋面，但是却惺惺相惜。技术、生活都可以通过网络进行交流，在网络世界里，天涯变为咫尺。

2004 年 6 月，我开始制作自己的个人网站（www.eygle.com），12 月的时候，又搭建了个人的 Blog 站点（blog.eygle.com），成为了一个 Blogger，通过自己的网站随意记下自己的一些文章和心情点滴，所以主页上体现的主题是 Friends Life & Oracle。有朋友，认真生活，热爱 Oracle，就是这么简单地生活。

而算起来，到 ITPUB 已经 5 个年头了，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。

在 ITPUB 上，很多朋友一起成长，大家从素昧平生到相识相知，这条路漫长却充满欣喜。现在生活中的朋友，基本上都是来自 ITPUB，经常有人问我，你们怎么能变得那么熟悉？也许没有别的原因，只因那条路大家一起走过，又碰巧彼此喜欢而已。总有那么多人会在你的一生中都难以忘怀。

我想在 ITPUB 里的每一个人都会有自己的一个圈子，自己熟识的人，自己从这里发现的好友知音。熟悉或者不熟悉的朋友，我想在这里你们都不可缺少，你们有你们的朋友，而我(们)或许老了点，但是你总能发现，属于你的 ITPUB，属于你的朋友。

相信这个圈子里的很多人都是和 ITPUB 一起成长起来的，我是如此，很多人也应该如此。ITPUB 之于我们，也许已经不再仅仅是一个论坛，ITPUB 是一种文化、一种凝聚力、一种生活，而且我们也相信，ITPUB 会越来越壮大、越成熟，并且能够提供越来越多、越来越好的服务给大家。

2004 年有了 ITPUB 第一本书《Oracle 数据库 DBA 专题技术精粹》的出版，那是 ITPUB 首次的出版尝试，得到了大家的支持以及鼓励，现在我们把精心编辑的第二本书奉献给大家，对于这本书我们投入了更多的精力以及心血，目的只有一个，那就是，不辜负大家对我们的支持及期待。

最后，我要感谢你们，熟悉的或者不熟悉的读者朋友们，你们使得这本书有了它存在的价值，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
2005 年 4 月 1 日 于北京

ITPUB 的第二本数据库技术图书即将出版了。2004 年 3 月份，ITPUB 的第一本数据库技术图书《Oracle 数据库 DBA 专题技术精粹》出版后，我们就马上开始筹集稿件策划本书的出版，前后经历了一年时间的精心准备，现在终于和大家见面了。

在此，感谢 4 位主编盖国强（eygle）、冯春培（biti_rainy）、叶梁（coolyl）和冯大辉（Fenng）在收集、整理和审订稿件过程中所付出的辛勤劳动，感谢 ITPUB 上众多朋友的鼓励和提供素质优良的稿件，感谢人民邮电出版社所给予的大力支持。

在当今这个信息爆炸的年代，一个人再以传统的方式去深入学习 IT 知识几乎是不可能了。作为一个成熟的商业数据库产品，Oracle 有完善的文档和支持体系，但由于文档的规模庞大，几乎没有用户能够有时间和精力进行全面的阅读和学习。而在一个生产系统中，用户所能遇到的问题总是有限的，所以现实工作环境能给技术人员带来的经验也终归是有限的。

网络交流是解决此问题的好方法，作为国内最大的 IT 技术交流平台之一，ITPUB 在三年以来一直主张和鼓励 IT 行业的朋友在网上多进行技术互动，交流实际经验和案例，交流对技术的理解和心得，交流相关的技术资源，我们深信，通过这样的交流，有利于促进每一位参与者的技术提高，从而在某些方面为推进我国 IT 技术的发展做出一些

贡献。我们这本关于 Oracle 数据库优化的图书，也正是基于这么一个背景而诞生的。

数据库的优化，有一定的工具和规则，但却没有固定的模式与步骤（如果有的话，那早就被融入到产品中），很大程度上取决于实施者的个人经验与对系统的理解程度，本书收集了几十个与优化有关的案例，并介绍了一些相关的理论背景，相信可以对从事 Oracle 数据库管理与调优工作的朋友在增长经验、开拓视野方面有所裨益。

ITPUB 社区成立于 2001 年（关于 ITPUB 的历史可参看《Oracle 数据库 DBA 专题技术精粹》一书的后记或访问 ITPUB 网站上 tigerfish 发布的相关帖子：http://www.itpub.net/61197.html），多年来 ITPUB 一直在为 IT 界的朋友服务，也得到广大 IT 同行的支持，注册会员到现在已经超过 30 万，累计页面点击超过 1.7 亿次，成为热点技术网站之一。

ITPUB 是一个由它的成员所组成的有机整体，如果只有服务器、程序和数据，那么 ITPUB 充其量也就是一个交流平台而已，正是众多本身就具有极强能力和扶危助困精神的版主和会员，以自己的生命力赋予给 ITPUB 旺盛的生命力，使 ITPUB 除了成为一个技术论坛以外，还成为一个活生生的网络社会，这个社会充分体现了善良、仁爱、诚信、互助的特质及积极向上的精神，培养出众多的技术高手，并且有人因此而完全改写了自己的人生道路。所有这一切，形成一种被称为“ ITPUB 模式 ”的概念，我个人觉得，无论从社会学或心理学的角度去看，ITPUB 作为一种独特的网络社会形态，可以成为相关学者研究的很好对象。

ITPUB 将来会向深度与广度两个方向发展，在服务上，现在已经开设有论坛和 BLOG，在年内还会添加 WIKI（维基）服务，以及根据各个论坛讨论板块内容的不同分别开设相关的外围主题门户网站，形成以论坛——BLOG——WIKI 为核心，内容门户为辅助的框架结构。另外在知识发布方面，会展开一些开源开发、图书写作和翻译计划以及 IT 技术视频辅导等项目，并且会开发 ITPUB 的肥客户端，使会员能够更加方便快捷地运用 ITPUB 平台。

在未来一年里，我们也会有更多的类似于本书的作品陆续奉献给大家，以感激所有朋友对 ITPUB 的长期支持。

tigerfish

2005 年 04 月 于广州