目录

序

“Think”（思考）。1914 年，Thomas J. Watson 先生加入后来成为 IBM 的公司时，带来了这样一个简简单单的座右铭。后来，这成为每一位 IBM 员工的训词，不论他们身居何职，只要需要做出决策，并利用自己的才智完成所承担的工作，就要把“Think“谨记于心。一时间，”Think“成为一个象征、一个标志，屡屡出现在出版物上，人们把它写在日历上提醒自己，而且不仅在 IBM 内部，就连其他一些公司的 IT 和企业管理者的办公室墙上也悬挂着这个牌匾，甚至《纽约客》杂志的漫画里都有它的身影。”Think“在 1914 年是一个很好的观念，即使在今天也同样有着重要的意义。

“Think different“（不同凡想）是 20 世纪 90 年代苹果公司在其旷日持久的宣传活动中提出的一个口号，想借此重振公司的品牌，更重要的是，想改变人们对技术在日常生活中作用的看法。苹果公司的口号不是”think differently“（换角度思考，暗含如何去思考），而是把”different“用作动词”think“的宾语，暗含该思考些什么（与”think big“句式相同）。这个宣传活动强调的是创造性和有创造性的人，暗示苹果电脑在支持重新和艺术成就方面与之不同。

我在 1981 年加入 Oracle 公司（那时还叫 Relational Software 公司）时，包含了关系模型的数据库系统还是一种新兴技术。开发人员、程序员和队伍逐渐壮大的数据库管理员都在学习采用规范化方法的数据库设计原则。在此之后出现了非过程性的 SQL 语言。尽管人们对它很陌生，但无不为其强大的能力所折服，因为利用 SQL 语 言能有效地管理数据，而以前同样的工作需要进行非常辛苦地编程才能完成。那时要思考的东西很多，现在也依然如此。这些新技术不仅要求人们学习新的概念和方 法，还要以新的思想来思考。不论是过去还是现在，做到了这一点的人最终都大获成功，他们能最大限度地利用数据库技术，为企业遇到的问题建立有效的创新性解 决方案。

想一想 SQL 数据库语言吧，历史上是 Oracle 首次推出了商业化的 SQL 实现。有了 SQL，应用设计人员可以利用一种非过程性语言（或称“描述性语言“）管理行集（即记录集），而不必使用传统的过程性语言编写循环（一次只能处理一条记录）。刚开始接触 SQL 时，我发现自己必须”转 45° “考 虑问题，以确定如何使用诸如联结和子查询之类的集合处理操作来得到我想要的结果。那时，集合处理对大多数人来说还是全新的概念，不仅如此，这也是非过程性 语言的概念，也就是说，你只需指定想要的结果，而无需指出如何得到这些结果。这种新技术确实要求我”换角度思考“，当然也使我有机会”不同凡想“。

集合处理比一次处理一条记录要高效得多，所以如果应用程序能以这种方式充分利用 SQL，就能比那些没有使用集合处理的应用程序表现得更出色。不过，遗憾的是，应用程序的性能往往都不尽如人意。实际上，大多数情况下，最能直接影响整体性能的是应用程序设计，而不是 Oracle 参数设置或其他配置选项。所以，应用程序开发人员不仅要学习数据库特性和编程接口的详细内容，还要掌握新的思路，并在应用程序中适当地使用这些特性和接口。

在 Oracle 社区中，关于如何对系统调优以得到最佳的性能（或者如何最佳地使用各种 Oracle 特性）有许多“常识“。这种原本明智的”常识“有时却演变成为一种”传说“甚至”

神话“，这是因为开发人员和数据库管理员可能不加如何批判地采纳这些思想，或者不做如何思考就盲目扩展它们。

举一个例子，比如说这样一个观点：“如果一个东西很好，那么更多——更多些——会更好。“这种想法很普通，但一般并不成立。以 Oracle 的数组接口为例，它允许开发人员只用一个系统调用就能插入或获取多行记录。显然，能减少应用程序和数据库之间传递的网络消息数当然很好。但是再想想看，到达某个”临界“点后，情况可能会改变。一次获取 100 行比一次获取 1 行要好得多，但是如果一次获取 1000 行而不是 100 行，这对于提供整体性能通常意义并不大，特别是考虑到内存需求时更是如此。

再来看另一个不加判断就采纳的例子，一般主张关注系统设计或配置中有问题的地方，而不是最有可能改善性能的方面（或者是能提高可靠性、可用性或增强安全性的 方面）。请考虑一个系统调优的“常识“：要尽可能提高缓冲区的命中率。对于某些应用，要尽量保证所需数据在内存中，这会最大限度地提高性能。不过，对于大 多数应用，最好把注意力放在它的性能瓶颈上（我们称之为”等待状态“），而不要过分强调某些系统级指标。消除应用程序设计中那些导致延迟的因素，就能得到 最佳的性能。

我发现，将一个问题分解为多个小部分再逐个加以解决，是一种很好的应用程序设计方法。采用这种方式，往往能极好地、创造性地使用 SQL 解决应用需求。通常，只需一条 SQL 语句就能完成许多工作，而你原来可能认为这些工作需要编写复杂的过程性程序才能实现。如果能充分利用 SQL 的强大能力一次处理一个行集（可能并行处理），这不仅说明你是一个高效率的游泳池开发人员，也说明应用程序能以更快的速度运行！

一些最佳实践取决于（或部分取决于）事实的真实性，有时，随着事实的改变，这些最佳实践可能不再适用。请考虑一句古老的格言：“要得到最好的性能，应当把索 引和数据放在单独的表空间中。“我经常发现许多数据库管理员都恪守着这个观点，根本不考虑如今磁盘的速度和容量已经大为改观，也不考虑给定工作负载的特殊 要求。要评价这个”规则“是否合适，应该考虑这样一个事实：Oracle数据库会在内存中缓存最近经常使用的数据库块（通常这些块属于某个索引）。还有一点需要考虑：对于给定的请求，Oracle数据库会顺序使用索引和数据块，而不是同时访问。这说明，所有并发用户实际上应该都会执行涉及索引好数据的I/O操作，而且每块磁盘上都会程序I/O操作。可能你会出于管理方面的原因（或者根据你的个人喜好）将索引和数据块分置于不同的表空间中，但就不能说这样做是为了提高性能。（Thomas Kyte在Ask Tom网站 http://asktom.oracle.com 上对这个主题做了深入的分析，有关文章可以在”index data table space“中查到。）从中我们可以得到一个教训，要根据事实做出决定，而且事实必须是当前的、完备的。

不论我们的计算机速度变得多快，数据库变得多复杂，也不管编程工具的能力如何，人类的智慧和一套正确的“思考原则“仍是无可替代的。所以，对于应用中使用的技术，尽管学习其细节很重要，但更重要的是，应该知道如何考虑适当地使用这些技术。

Thomas Kyte 是我认识的最聪明的人之一，他在 Oracle 数据库、SQL、性能调优和应用设计方面具有渊博的学识。我敢肯定，Thomas 绝对是“Think“和”Think different“这两个口号不折不扣的追随者。有位中国的智者说过”授人以鱼，为一饭之惠；授人以渔，则终身受用“，显然 Thomas 对此深以为然。Thomas 很乐于把自己的 Oracle 知识与大家共享，但他并不是只是罗列问题的答案，而是尽力帮助大家学会如何思考和推理。

在Thomas的网站（http://asktom.oracle.com）上、发言稿中以及书中，他其实不断鼓励人们在使用Oracle数据库设计数据库应用时要“换角度思考“。他从不墨守成规，而坚持通过实例，用事实证明。Thomas采用一种注重实效的简单方法来解决问题，按照他的建议和方法，你将成为更高效的开发人员，能开发出更好、更快的应用。

Thomas 的这本书不仅介绍 Oracle 的诸多特性，教你使用这些特性，还反映了以下简单的观点：

- 不要相信神话，要自己思考。
- 不要墨守成规，所有人都知道的事情其实很可能是错的！
- 不要相信传言，要自己测试，根据经过证明的示例做出决定。
- 将问题分解为更简单的小问题，再把每一步的答案组合为一个优秀、高效的解决方案。
- 如果数据库能更好、更快地完成工作，就不要事必躬亲地自己编写程序来完成。
- 理解理想和现实之间的差距。
- 对于公司制定的未加证实的技术标准，要敢于提出质疑。
- 要针对当前需求从大局考虑怎样做最好。
- 要花时间充分地思考。

Thomas 建议，不要只是把 Oracle 当作一个黑盒。你不只是在 Oracle 中放入和取出数据。他会帮助你理解 Oracle 是如何工作的，如何充分利用它强大的能力。通过学习如何深思熟虑地、创造性地应用 Oracle 技术，你会更快、更好地解决大多数应用设计问题。

通过阅读这本书，你会了解到 Oracle 数据库技术的许多新动态，还会掌握应用设计的一些重要概念。如果你确实领会了这些思想，相信你肯定也会对所面对的难题“换角度思考”。

IBM 的 Watson 曾经说过：“自始以来，每一个进步都源自于思考。仅仅因为‘没有思考’，就造成全世界白白浪费了无数资金。”Thomas 和我都赞同这种说法。学完这本书后，利用你掌握的知识和技术，希望你能为这个世界（至少能为你的企业）节省无数资金，把工作干得更出色。

过去我一直在开发 Oracle 软件，并与其他 Oracle 开发人员一同工作，帮助他们构建可靠、健壮的应用程序。在这个过程中积累了一些经验，这是这些经验赋予我灵感，才有了本书中的内容。这本书实际上反映了我每天做了些什么，汇集了我所看到的人们每天遇到的问题。

本书涵盖了我认为最重要的一些内容，即 Oracle 数据库及其体系结构。我也可以写一本书名类似的其他方面的书，向你解释如何用一种特定的语言和体系结构开发应用程序。例如，我可以告诉你如何使用 JavaServer Pages（JSP）与 Enterprise JavaBeans（EJB）通信，EJB 再如何使用 JDBC 与 Oracle 通信。不过，归根到底，你最后还是要了解 Oracle 数据库及其体系结构（本书介绍的内容），才能成功地构建这样一个应用程序。要想成功地使用 Oracle 进行开发，我认为有些内容你必须了解，而不论你是一位使用 ODBC 的 Visual Basic 程序员、使用 EJB 和 JDBC 的 Java 程序员，还是使用 DBI Perl 的 Perl 程序员，这本书都会介绍这些通用的知识。本书并不推崇哪一种特定的应用体系结构，在此没有比较三层机构和客户/服务器结构孰优孰劣。我们只是讨论了数据库能做什么，另外关于数据库如何工作，我们还会指出你必须了解哪些内容。由于数据库是所有应用体系结构的核心，所以这本书使用面很广。

在编写本书时，我对 Expert One-on-One Oracle 一书中关于体系结构的章节做了全面修订和更新，并补充了大量新的内容。Expert One-on-One Oracle 一书所基于的版本是 Oracle 8.1.7，在此之后又推出了 3 个版本——两个 Oracle9i 版本和 Oracle 数据库 10g Release 1，这也是写这本书时的 Oracle 发行版本。因此，有许多新的功能和新的特性需要介绍。

如果针对 9i 和 10g 更新 Expert One-on-One Oracle，那么需要补充的内容太多了，那本书原本篇幅较多，再加太多内容就会很难处理。出于这个考虑，我们决定分两本书来介绍。这是其中的第一本，第二本书暂定名为 Expert Oracle Programming。

顾名思义，本书的重点是数据库体系结构，并强调数据库本身如何工作。我会深入地分析 Oracle 数据库体系结构，包括文件、内存结构以及构成 Oracle 数据库（database）和实例（instance）的底层进程。然后讨论一些重要的数据库主题，如锁定、并发控制、事务、redo 和 undo，还会解释为什么了解这些内容很重要。最后，我们再来分析数据库这的物理结构，如表、索引和数据类型，并介绍哪些技术能最优地使用这些物理结构。

本书内容

如果开发的选择余地很大，则会带来一些问题，其中一个问题是有时很难确定哪种选择是满足特定需求的最佳选择。每个人都希望灵活性尽可能大（有尽可能多的选择），同时他们又希望能简单明了，换句话说，希望尽量容易。Oracle 为开发人员提供的选择几乎无穷无尽。没有人会说：“这在 Oracle 中做不到”，而只会说“在 Oracle 中你想用多少种不同的方法来实现？”希望这本书能帮助你做出正确的选择。

如果你不只是想知道做何选择，还想了解有关 Oracle 特性和功能的一些原则和实现细节，这本书就很适合你。例如，Oracle 有一个很棒的特性，称为并行执行（parallel execution）。Oracle 文档会告诉你如何使用这个特性，并说明它到底能做什么。不过，Oracle 文档没有告诉你应该在什么时候用这个特性，更重要的是没有指出什么时候不该使用这个特性。另外，

文档一般没有提供特性的实现细节，如果你不清楚，可能会因此而困扰（我指的不是 bug，而是说你可能很想知道这个特性如何工作，以及为此是怎样具体设计的，但从文档中找不到答案 。

在本书中，我不仅会尽力阐明各个特性如何工作，还会指出什么情况下要考虑使用某个特性或实现，并解释为什么。我认为，理解“怎么做”固然很重要，但理解“什么时候做”和“为什么这样做”（以及“什么时候不做”和“为什么不做”）也同样重要！

读者对象

这本书面向那些使用 Oracle 作为数据库后端开发应用程序的人员。专业 Oracle 开发人员如果想了解如何在数据库中完成某些工作，同样可以参考本书。本书相当实用，所以 DBA 也会对书中的许多内容感兴趣。书中大部分例子都使用 SQL*Plus 来展示关键特性，所以如果你想通过本书来了解如何开发一个很酷的 GUI，可能不能如愿。不过，从这本书中，你将知道 Oracle 数据库如何工作，它的关键特性能做什么，以及什么时候应该（和不应该）使用这些特性。

如果你想事半功倍地使用 Oracle，如果你想了解使用现有特性的新方法，如果你想知道这些特性在真实世界中如何应用（不只是展示如何使用特性，而是首先分析为什么要用这个特性），就请阅读这本书。作为技术经理，如果你手下的开发人员在开发 Oracle 项目，你可能也会对这本书感兴趣。从某种程度上讲，技术经理也要懂数据库，而且要知道这对于成功至关重要。如果技术经理想安排员工进行适当的技术培训，或者想确保员工了解他们应该掌握的技术，就可以利用这本书来“充电”。

要想更好地学习本书的内容，要求读者：

- 了解 SQL。不要求你能编写很棒的 SQL 代码，但是如果用过 SQL，对 SQL 有实战经验，这会很有帮助。
- 掌握 PL/SQL。这不是一个必要的前提，但是有助于你“领会”书中的例子。例如，本书不会教你这样编写一个 FOR 循环，或者如何声明一个记录类型，这些内容可以参考 Oracle 文档和许多相关的图书。不过，这并不是说你从本书中学不到 PL/SQL 的知识。不是这样的。通过阅读本书，你会 PL/SQL 的许多特性相当熟悉，而且会学到一些新方法，还会注意到你以前以为不存在的一些包和特性。
- 接触过某种第三代语言（third-generation language, 3GL），如 C 或 Java。我相信，如果你能阅读 3GL 语言编写的代码，或者编写过这种代码，肯定能顺利地阅读和理解本书中的例子。
- 熟悉 Oracle Concepts 手册。

最后再说两句，由于 Oracle 文档实在太庞大了，这让很多人有些畏惧。如果你刚开始读 Oracle Concepts 手册，或者还没有看过，那我可以告诉你，这个手册绝对是一个很好的起点。它大约有 700 页，涉及了你需要知道的许多重要的 Oracle 概念。其中不会涵盖每一个技术细节（Oracle 文档提供了技术细节，不过它有 10,000～20,000 页之多），但你能从中学到所有重要的概念。

这个手册涉及以下主题（这里所列的并不完整）：

- 数据库中的结构，数据如何组织和存储；
- 分布式处理；
- Oracle 的内存体系结构；
- Oracle 的进程体系结构；
- 你要使用的模式对象（表、索引、聚簇等）；
- 内置数据类型和用户定义的数据类型；
- SQL 存储过程；
- 事务如何工作；
- 优化器；
- 数据完整性；
- 并发控制。

我自己也会时不时地温习这些内容。这些都是基础，如果不了解这些知识，你创建的 Oracle 应用程序就很容易失败。建议你通读 Oracle Concepts 手册来了解这些主题。

本书组织结构

为了帮助你更好地使用这本书，大部分章节都组织为 4 个部分。这个划分并不严格，不过有助于你快速地找到感兴趣的方面，从中获得所需的更多信息。本书有 15 章，每一章都像一本“迷你书”，可以单独成册。有时我会引用其他章中的例子或特性，不过你完全可以从书中任选一章，不参考其他章也能顺利阅读。例如，要理解或使用第 14 章关于并行机制的知识，就不必阅读介绍数据库表的第 10 章。

许多章的格式和风格基本上都一样：

- 首先是特性和功能的介绍。
- 说明为什么可能想使用（或者不想使用）这个特性或功能。我会概要地指出哪些情况想要考虑使用这个特性，而哪些情况下这个特性不适用。
- 如何使用这个特性。这里提供的信息不是完全照搬 SQL 参考资料中的内容，而是会以一种循序渐进的方式组织。我会清楚地指出哪些是你需要的，哪些是你必须做的，另外哪些环节需要仔细检查。这一部分包括以下内容：

  - 如何实现这个特性；
  - 许许多多的例子；
  - 如何调试这个特性；
  - 使用这个特性的忠告；
  - 如何（主动地）处理错误。

- 对上述内容的小结。

书中有相对多的例子和大量的代码，这些都可以从http://www.apress.com的Source Code区下载。下面将详细介绍每一章的内容。

第 1 章：开发成功的 Oracle 应用

从 这一章开始，我将介绍数据库编程的基本方法。所有数据库创建得并不一样，要想按时、成功地开发数据库驱动的应用，你必须了解你的数据库能做什么，它是怎么 做的。如果不清楚你的数据库能做什么，就很可能不断地遭遇“闭门造车”的窘境，徒劳地从头开发数据库本已提供的功能；如果不清楚你的数据库是怎么工作的， 很可能开发出性能很差的应用，达不到预期的要求。

这 一章先根据经验分析了一些应用，这些应用都因为缺乏对数据库的基本理解而导致项目失败。这一章就采用这种“拿例子说话“的方式，讨论了开发人员必须了解数 据库的哪些基本特性和功能。关键是，不要把数据库当成一个黑盒，不要认为它能自己努力得出答案并自行负责可扩展性和性能。

第 2 章：体系结构概述

这一章介绍 Oracle 体系结构的基础知识。首先给出两个术语——“实例”（instance）和“数据库”（database）的明确定义，Oracle 领域中的许多人都对这两个词存在误解。我们还会简要介绍系统全局区（System Global Area， SGA）和 Oracle 实例底层的进程，并分析“连接 Oracle”这样一个简单的动作是如何实现的。

第 3 章：文件

这一章将深入介绍构成 Oracle 数据库和实例的 8 类文件。从简单的参数文件到数据文件和重做日志文件（redo log file）都会涵盖。我们将说明这些文件是什么，为什么有这些文件，以及如何使用它们。

第 4 章：内存结构

这一章讨论 Oracle 如何使用内存，包括各个进程中的内存（PGA 内存，PGA 即进程全局区）和共享内存（SGA）。我们会分析手动和自动 PGA 内存管理之间的区别，并介绍 Oracle 10g 中的 SGA 内存管理，还会说明各种方法适用于什么情况。读完这一章之后，你会对 Oracle 如何使用和管理内存有深入的了解。

第 5 章：Oracle 进程

这一章概述了各种 Oracle 进程（服务器进程和后台进程），另外还相当深入地讨论了通过共享服务器进程或专用服务器进程连接数据库有何区别。启动 Oracle 实例时会看到一些后台进程，这一章将逐一介绍其中一些重要的后台进程（如 LGWR、DBWR、PMON 和 SMON），并分别讨论这些进程的功能。

第 6 章：锁

不同的数据库有不同的行事方法（SQL Server 里能做的在 Oracle 中不一定能做）。应当了解 Oracle 如何实现锁定和并发控制，这对于应用的成功至关重要。这一章将讨论 Oracle 解决这些问题的基本方法，可以应用哪些类型的锁[DML、DDL 和闩（latch）]，还会指出如果锁定实现不当会出现哪些问题（死锁、阻塞和锁升级）。

第 7 章：并发与多版本

这一章介绍我最喜欢的 Oracle 特性——多版本（multi-versioning），并讨论它并发控制和应用设计有什么影响。在这里能清楚地看到，所有数据库创建得都不一样，具体的实现会对应用的设计产生影响。我们先回顾 ANSI SQL 标准定义的各个事务隔离级别，并介绍它们在 Oracle 中的具体实现（还会介绍其他数据库中的实现）。基于多版本特性，Oracle 能够在数据库中提供非阻塞读（non-blocking read）。本章接下来会分析多版本特性对我们有什么影响。

第 8 章：事务

事务是所有数据库的一个基本特性，这也是数据库区别于文件系统的一个方面。不过，事务常常遭到误解，很多开发人员甚至不知道他们有时没有使用事务。这一章讨论 Oracle 中应当如何使用事务，还列出了使用数据库进行开发时可能出现的一些“坏习惯”。特别地，我们将讨论原子性的含义，并说明原子性对 Oracle 中的语句有何影响。这一章还会讨论事务控制语句（COMMIT、SAVEPOINT 和 ROLLBACK）、完整性约束和分布式事务（两段提交或 2PC），最后介绍自治事务。

第 9 章：redo 与 undo

可能有人说，开发人员不用像 DBA 那样深入了解 redo（重做信息）和 undo（撤销信息）的细节，但是开发人员确实要清楚 redo 和 undo 在数据库中所起的重要作用。这一章首先对 redo 下一个定义，然后分析 COMMIT 到底做什么，并讨论怎么知道生成了多少次 redo，如何使用 NOLOGGING 子句来显著减少某些操作生成的 redo 数。我们还研究了 redo 生成与块清除（block cleanout）和（log contention）等问题的关系。

这一章的 undo 一节中讨论了撤销数据的作用，并介绍哪些操作会生成最多/最少的 undo。最后分析“讨厌”的 ORA-01555：snapshot too old（ORA-01555：快照太旧）错误，解释导致这个错误的可能原因，并说明如何避免。

第 10 章：数据库表

Oracle 现在支持多种表类型。这一章将分别介绍每一种类型，包括堆组织表（heap organized，也就是默认的“普通”表）、索引组织表（index organized）、索引聚簇表（index clustered）、散列聚簇表（hash clustered）、嵌套表（nested）、临时表（temporary）和对象表（object），并讨论什么时候使用这些类型的表、如何使用以及为什么使用。大多数情况下，堆组织表就足够了，不过这一章还将帮助你认识到在哪些情况下使用其他类型的表更合适。

第 11 章：索引

索引是应用设计的一个重要方面。要想正确地实现索引，要求深入地了解数据，清楚数据如何发布，并且知道要如何使用数据。人们经常把索引当作“马后炮”，直到应用开发的后期才增加，这就会导致应用的性能低下。

这一章将详细分析各种类型的索引，包括 B*Tree 索引、位图索引（bitmap index）、基于函数索引（function-based index）和应用域索引（application domain index），并讨论各种索引应该在哪些场合使用，以及哪些场合不适用。我会在“有关索引的常见问题和神话”一

节回答常常被问到的一些问题，如“索引能在视图上使用吗？”和“为什么没有使用我的索引？”。

第 12 章：数据类型

有许多数据类型（datatype）可供选择。这一章会逐一分析 22 种内置数据类型，解释这些类型是如何实现的，并说明如何以及何时使用这些数据类型。首先对国家语言支持（National Language Support，NLS）做一个简要的概述；要想充分理解 Oracle 中简单的串类型，必须先掌握这个基础知识。接下来再讨论广泛使用的 NUMBER 类型，并介绍 Oracle 10g 对于在数据库中存储数值又提供了哪些新的选项。我们主要从历史角度介绍 LONG 和 LONG RAW 类型，目的是讨论如何处理应用中遗留的 LONG 列，并将其移植为 LOB 类型。然后会深入介绍分析存储日期和时间的各种数据类型，讨论如何处理这些数据类型来得到我们想要的结果。这里还会谈到时区支持的有关细节。

接下来讨论 LOB 数据类型。我们会说明 LOB 类型的存储方式，并指出各种设置（如 IN ROW、CHUNK、RETENTION、CACHE 等）对我们有什么意义。处理 LOB 时，重要的是要了解默认情况下它们如何实现和存储，在对 LOB 的获取和存储进行调优时这一点尤其重要。本章的最后介绍 ROWID 和 UROWID 类型。这些是 Oracle 专用的特殊类型，用于表示行地址。我们会介绍什么时候可以将它们用作表中的列数据类型（这种情况几乎从来不会出现）。

第 13 章：分区

分区（partitioning）的目的是为了便于管理非常大的表和索引，即实现一种“分而治之”的逻辑，实际上就是把一个表或索引分解为多个较小的、更可管理的部分。在这方面，DBA 和开发人员必须协作，使应用能有最大的可用性和最高的性能。这一章介绍了表分区和索引分区。我们会谈到使用局部索引（在数据仓库中很常用）和全局索引（常见于 OLTP 系统）的分区。

第 14 章：并行执行

这一章介绍了 Oracle 中并行执行（parallel execution）的概念，并说明了如何使用并行执行。首先指出并行处理在什么情况下有用，以及哪些情况下不应考虑使用它。有了一定的认识后，再来讨论并行查询的机制，大多数人提到并行执行都会想到这个特性。接下来讨论并行 DML（parallel DML， PDML），利用 PDML，可以使用并行执行完成修改。我们会介绍 PDML 在物理上如何实现，并说明为什么这个实现会对 PDML 带来一系列限制。

然后再来看并行 DDL。在我看来，这才是并行执行真正的闪光之处。通常，DBA 会利用一些小的维护窗口来完成大量的操作。利用并行 DDL，DBA 就能充分利用可用的机器资源，在很短的时间内完成很大、很复杂的操作（它只需原先串行执行所需时间的很小一部分）。

这一章的最后将讨论过程并行机制（procedural parallelism），采用这种方法可以并行地执行应用程序代码。这里将介绍两个技术。首先是并行管线函数（parallel pipelined function），即 Oracle 能动态地并行执行存储函数。第二个技术是 DIY 并行机制（DIY parallelism），利用这个技术可以把应用设计为并发地运行。

第 15 章：数据加载和卸载

这一章第一部分重点介绍 SQL*Loader（SQLLDR），并说明可以采用哪些方法使用这个工具来加载和修改数据库中的数据。我们会讨论以下问题：加载定界数据，更新现有的行和插入新行，卸载数据，以及从存储过程调用 SQLLDR。重申一遍，SQLLDR 是一个完备而重要的工具，但它的实际使用也带来很多问题。这一章第二部分主要讨论外部表，这是另外一种数据批量加载和卸载的高效方法。

源代码和有关更新

使用这本书中的例子时，你可能想亲手键入所有代码。很多读者都喜欢这样做，因为这是熟悉编码技术的一种好方法。

无论你是否想自己键入代码，都能从Apress网站（http://www.apress.com）的Source Code区下载本书的所有源代码。即使确实想自己键入代码，下载源代码也很有必要，你可以使用下载的源代码文件检查正确的结果是什么。如果你认为自己的录入可能有误，就可以先从这一步开始。倘若不想自己键入代码，那么除了从Apress网站下载源代码外别无选择！不论采用哪种方式，代码文件都能帮助你完成更新和调试。

勘误表

Apress极力确保文字或代码不会出错。不过，出错也是人之常情，所以只要发现并修改了错误，我们就会及时告诉你。Apress所有书籍的勘误表都可以在http://www.apress.com 上找到。如果你发现一个还没有报告的错误，请通知我们。

Apress 网站还提供了其他的信息和支持，包括所有 Apress 书籍的代码、样章、新书预告以及相关主题的文章等。

配置环境

这里我会介绍如何建立一个执行本书实例的环境，具体包括以下主题：

- 如何正确地建立 SCOTT/TIGER 演示模式；
- 需要建立和运行的环境；
- 如何配置 SQL*Plus 工具 AUTOTRACE；
- 如何安装 Statspack；
- 如何安装和运行 runstats 以及本书中用到的其他定制实用程序；
- 本书所用的编码约定。

所有代码（只要不是Oracle自动生成的脚本）都能从Apress网站（http://www.apress.com）的Source Code区下载。

建立 SCOTT/TIGER 模式

你的数据库里可能已经有 SCOTT/TIGER 模式（schema）了。经典安装通常就包括这

个模式，不过并不要求数据库一定得有这个组件。可以把 SCOTT 示例模式安装到任何数据库账户下，使用 SCOTT 账户并没有特殊的意义。如果乐意，你还可以把 EMP/DEPT 表直接安装在你自己的数据库账户下。

这本书里许多例子都用到了 SCOTT 模式中的表。如果你想顺利地使用这些例子，也需要有这些表。如果你使用的是一个共享数据库，最好复制这些表，并把你自己的副本安装在 SCOTT 以外的某个账户下，以避免使用同一数据的其他用户可能带来的副作用。

要创建 SCOTT 演示表，步骤很简单：

（1）cd [ORACLE_HOME]/sqlplus/demo。

（2）以任意用户身份连接后运行 demobld.sql。

注意 对于 Oracle 10g 及以后版本，必须安装配套光盘中的演示子目录。后面我会列出 demobld.sql 中必要的部分。

demobld.sql 会创建 5 个表并填入数据。执行结束后，它会自动退出 SQL*Plus，所以运行完这个脚本后 SQL*Plus 窗口将消失，对此不要感到奇怪，这是正常的。

这些标准演示表上没有定义任何引用完整性，但后面有些例子要求表必须有引用完整性，所以运行完 demobld.sql 后，建议你再执行以下脚本：

```
alter table emp add constraint emp_pk primary key(empno);

alter table dept add constraint dept_pk primary key(deptno);

alter table emp add constraint emp_fk_dept

foreign key(deptno) references dept;

alter table emp add constraint emp_fk_emp foreign key(mgr) references emp;
```

这就完成了演示模式的安装。如果以后你整理系统时删除这个模式，只需执行 [ORACLE_HOME]/sqlplus/demo/demodrop.sql。这个脚本会删除 5 个演示表，并退出 SQL*Plus。

如果你无法访问 demobld.sql，也不用着急，以下脚本就足以运行本书中的示例：

```
CREATE TABLE EMP

(EMPNO NUMBER(4) NOT NULL,

ENAME VARCHAR2(10),

JOB VARCHAR2(9),

MGR NUMBER(4),
```

```
HIREDATE DATE,

SAL NUMBER(7, 2),

COMM NUMBER(7, 2),

DEPTNO NUMBER(2) );

INSERT INTO EMP VALUES (7369, 'SMITH', 'CLERK', 7902, TO_DATE('17-DEC-1980', 'DD-MON-YYYY'), 800, NULL, 20); INSERT INTO EMP VALUES (7499, 'ALLEN', 'SALESMAN', 7698, TO_DATE('20-FEB-1981', 'DD-MON-YYYY'), 1600, 300, 30); INSERT INTO EMP VALUES (7521, 'WARD', 'SALESMAN', 7698, TO_DATE('22-FEB-1981', 'DD-MON-YYYY'), 1250, 500, 30); INSERT INTO EMP VALUES (7566, 'JONES', 'MANAGER', 7839, TO_DATE('2-APR-1981', 'DD-MON-YYYY'), 2975, NULL, 20); INSERT INTO EMP VALUES (7654, 'MARTIN', 'SALESMAN', 7698, TO_DATE('28-SEP-1981', 'DD-MON-YYYY'), 1250, 1400, 30); INSERT INTO EMP VALUES (7698, 'BLAKE', 'MANAGER', 7839, TO_DATE('1-MAY-1981', 'DD-MON-YYYY'), 2850, NULL, 30); INSERT INTO EMP VALUES (7782, 'CLARK', 'MANAGER', 7839, TO_DATE('9-JUN-1981', 'DD-MON-YYYY'), 2450, NULL, 10); INSERT INTO EMP VALUES (7788, 'SCOTT', 'ANALYST', 7566, TO_DATE('09-DEC-1982', 'DD-MON-YYYY'), 3000, NULL, 20); INSERT INTO EMP VALUES (7839, 'KING', 'PRESIDENT', NULL, TO_DATE('17-NOV-1981', 'DD-MON-YYYY'), 5000, NULL, 10); INSERT INTO EMP VALUES (7844, 'TURNER', 'SALESMAN', 7698,

TO_DATE('8-SEP-1981', 'DD-MON-YYYY'), 1500, 0, 30); INSERT INTO EMP VALUES (7876, 'ADAMS', 'CLERK', 7788, TO_DATE('12-JAN-1983', 'DD-MON-YYYY'), 1100, NULL, 20); INSERT INTO EMP VALUES (7900, 'JAMES', 'CLERK', 7698, TO_DATE('3-DEC-1981', 'DD-MON-YYYY'), 950, NULL, 30); INSERT INTO EMP VALUES (7902, 'FORD', 'ANALYST', 7566, TO_DATE('3-DEC-1981', 'DD-MON-YYYY'), 3000, NULL, 20); INSERT INTO EMP VALUES (7934, 'MILLER', 'CLERK', 7782, TO_DATE('23-JAN-1982', 'DD-MON-YYYY'), 1300, NULL, 10);

CREATE TABLE DEPT

(DEPTNO NUMBER(2),

DNAME VARCHAR2(14),

LOC VARCHAR2(13) );

INSERT INTO DEPT VALUES (10, 'ACCOUNTING', 'NEW YORK'); INSERT INTO DEPT VALUES (20, 'RESEARCH', 'DALLAS'); INSERT INTO DEPT VALUES (30, 'SALES', 'CHICAGO'); INSERT INTO DEPT VALUES (40, 'OPERATIONS', 'BOSTON');
```

环境

本书中大多数例子都完全能在 SQL*Plus 环境中运行。除了 SQL*Plus 之外，不需要再安装和配置其他工具。不过，对于使用 SQL*Plus 我有一个建议。本书中几乎所有示例都以某种方式使用了 DBMS_OUTPUT。要让 DBMS_OUTPUT 正常工作，必须执行以下 SQL*Plus 命令：

```
SQL> set serveroutput on
```

如果你像我一样，每次都键入这个命令很快就会厌烦的。幸运的是，SQL*Plus 允许建立一个 login.sql 文件，每次启动 SQL*Plus 时都会执行这个脚本。另外，还允许设置一个环境变量 SQLPATH，这样不论这个 login.sql 脚本具体在哪个目录中，SQL*Plus 都能找到它。

对于本书中的所有示例，我使用的 login.sql 如下：

```
define _editor=vi

set serveroutput on size 1000000

set trimspool on set long 5000

set linesize 100

set pagesize 9999

column plan_plus_exp format a80

column global_name new_value gname

set termout off

define gname=idle

column global_name new_value gname

select lower(user) || '@' || substr( global_name, 1,

   decode( dot, 0, length(global_name), dot-1) ) global_name

from (select global_name, instr(global_name,'.') dot from global_name );

set sqlprompt '&gname> '

set termout on
```

下面对这个脚本做些说明：

- DEFINE_EDITOR=VI：设置 SQL*Plus 使用的默认编辑器。可以把默认编辑器设置为你最喜欢的文本编辑器（而不是字处理器），如记事本（Notepad）或 emacs。
- SET SERVEROUTPUT ON SIZE 1000000：这会默认地打开 DBMS_OUTPUT（这样就不必每次再键入这个命令了）。另外也将默认缓冲区大小设置得尽可能大。
- SET TRIMSPOOL ON：假脱机输出文本时，会去除文本行两端的空格，而且行宽不定。如果设置为 OFF（默认设置），假脱机输出的文本行宽度则等于所设置的 LINESIZE。
- SET LONG 5000：设置选择 LONG 和 CLOB 列时显示的默认字节数。
- SET LINESIZE 100：设置 SQL*Plus 显示的文本行宽为 100 个字符。
- SET PAGESIZE 9999：PAGESIZE 可以控制 SQL*Plus 多久打印一次标题，这里将 PAGESIZE 设置为一个很大的数（所以每页只有一组标题）。
- □ COLUMN PLAN_PLUS_EXP FORMAT A80：设置由 AUTOTRACE 得到的解释计划输出（explain plan output）的默认宽度。A80 通常足以放下整个计划。

login.sql 中下面这部分用于建立 SQL*Plus 提示符：

```
define gname=idle

column global_name new_value gname

select lower(user) || '@' || substr( global_name, 1,

   decode( dot, 0, length(global_name), dot-1) ) global_name

from (select global_name, instr(global_name,'.') dot from global_name );

set sqlprompt '&gname> '

set termout on
```

COLUMN GLOBAL_NAME NEW_VALUE GNAME 指令告诉 SQL*Plus 取得 GLOBAL_NAME 列中的最后一个值，并将这个值赋给替换变量 GNAME。接下来，我从数据库中选出 GLOBAL_NAME，并与我的登录用户名连接。这样得到的 SQL*Plus 提示符为：

```
ops$tkyte@ora10g>
```

这样一来，我就能知道我是谁，还有我在哪儿。

设置 SQL*Plus 的 AUTOTRACE

AUTOTRACE 是 SQL*Plus 中一个工具，可以显示所执行查询的解释计划（explain plan）以及所用的资源。这本书中大量使用了 AUTOTRACE 工具。

配置 AUTOTRACE 的方法不止一种，以下是我采用的方法：

（1）cd [ORACLE_HOME]/rdbms/admin；

（2）作为 SYSTEM 登录 SQL*Plus；

（3）运行@utlxplan；

（4）运行 CREATE PUBLIC SYNONYM PLAN_TABLE FOR PLAN_TABLE；

（5）运行 GRANT ALL ON PLAN_TABLE TO PUBLIC。

如果愿意，可以把 GRANT TO PUBLIC 中的 PUBLIC 替换为某个用户。通过将 PLAN_TABLE 置为 public，任何人都可以使用 SQL*Plus 进行跟踪（在我看来这并不是件坏事）。这么一来，就不需要每个用户都安装自己的计划表。还有一种做法是，在想要使用 AUTOTRACE 的每个模式中分别运行@utlxplan。

下一步是创建并授予 PLUSTRACE 角色：

（1）cd [ORACLE_HOME]/sqlplus/admin；

（2）作为 SYS 或 SYSDBA 登录 SQL*Plus；

（3）运行@plustrce；

（4）运行 GRANT PLUSTRACE TO PUBLIC。

重申一遍，如果愿意，可以把 GRANT 命令中 PUBLIC 替换为每个用户。

关于 AUTOTRACE

你会自动得到一个 AUTOTRACE 报告，其中可能列出 SQL 优化器所用的执行路径，以及语句的执行统计信息。成功执行 SQL DML（即 SELECT、DELETE、UPDATE、MERGE 和 INSERT）语句后就会生成这个报告。它对于监视并调优这些语句的性能很有帮助。

控制报告

通过设置 AUTOTRACE 系统变量可以控制这个报告：

- SET AUTOTRACE OFF：不生成 AUTOTRACE 报告，这是默认设置。
- SET AUTOTRACE ON EXPLAIN：AUTOTRACE 报告只显示优化器执行路径。
- SET AUTOTRACE ON STATISTICS：AUTOTRACE 报告只显示 SQL 语句的执行统计信息。
- SET AUTOTRACE ON：AUTOTRACE 报告既包括优化器执行路径，又包括 SQL 语句的执行统计信息。
- SET AUTOTRACE TRACEONLY：这与 SET AUTOTRACE ON 类似，但是不显示用户的查询输出（如果有的话）。

配置 Statspack

只有作为 SYSDBA 连接时才能安装 Statspack。所以，要想安装 Statspack，必须你完成 CONNECT /AS SYSDBA 操作。在许多安装中，必须由 DBA 或管理员来完成这个任务。

只要能（作为 SYSDBA）连接，安装 Statspack 就是小菜一碟了，只需运行@spcreate.sql。这个脚本可以在[ORACLE_HOME]\rdbms\admin 中找到，作为 SYSDBA 连接时，应该像下面这样通过 SQL*Plus 执行这个脚本：

```
[tkyte@desktop admin]$ sqlplus / as sysdba

SQL*Plus: Release 10.1.0.4.0 - Production on Sat Jul 23 16:26:17 2005

Copyright (c) 1982, 2005, Oracle. All rights reserved.

Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.4.0 - Production

With the Partitioning, OLAP and Data Mining options

sys@ORA10G> @spcreate

... Installing Required Packages

... <output omitted for brevity> ...
```

运行 spcreate.sql 脚本之前，你要了解 3 个信息：

- 将创建的 PERFSTAT 模式使用什么密码？
- PERFSTAT 使用的默认表空间是什么？
- PERFSTAT 使用的临时表空间是什么？

执行脚本时会提示你输入这些信息。如果输入有误，或者不小心取消了安装，在下一次尝试安装 Statspack 之前应该先用 spdrop.sql 删除用户（PERFSTAT）和已经安装的视图。安装 Statspack 会创建一个名为 spcpkg.lis 的文件。如果出现错误就应该检查这个文件。不过，只要提供了合法的表空间名（而且尚没有 PERFSTAT 用户），Statspack 包应该能顺利地安装。

定制脚本

在这一节中，我会介绍本书所用脚本的相关需求。另外还会分析脚本底层的代码。

runstats

runstats 是我开发的一个工具，能对做同一件事的两个不同方法进行比较，得出孰优孰劣的结果。你只需提供两个不同的方法，余下的事情都由 runstats 负责。runstats 只是测量 3 个要素：

- 墙上时钟（wall clock）或耗用时间（elapsed time）：知道墙上时钟或耗用时间很有用，不过这不是最重要的信息。

- 系统统计结果：会并排显示每个方法做某件事（如执行一个解析调用）的次数，并展示二者之差。
- 闩定（latching）：这是这个报告的关键输出。

你在本书中会了解到，闩（latch）是一种轻量级的锁。锁（lock）是一种串行化设备，而串行化设备不支持并发。如果应用不支持并发，可扩缩性就比较差，只能支持较少的用户，而且需要更多的资源。构建应用时，我们往往希望应用能很好地扩缩，也就是说，为 1 位用户服务与为 1,000 或 10,000 位用户服务应该是一样的。应用中使用的闩越少，性能就越好。如果一种方法从墙上时钟来看运行时间较长，但是只使用了另一种方法 10%的闩，我可能会选择前者。因为我知道，与使用更多闩的方法相比，使用较少闩的方法能更好地扩缩。

runstats 最后独立使用，也就是说，最好在一个单用户数据库上运行。我们会测量各个方法的统计结果和闩定（锁定）活动。runstats 在运行过程中，不希望其他任务对系统的负载或闩产生影响。只需一个小的测试数据库就能很好地完成这些测试。例如，我就经常使用我的台式机或手提电脑进行测试。

注意 我相信所有开发人员都应该有一个自行控制的试验台（test bed）数据库，可以在它上面尝试自己的想法，而不必要求 DBA 参与。假定数据库的个人开发版本称“如果数据库用来开发和测试，但不进行部署，那你可以放心使用”。只要有此许可，开发人员都应该在自己的台式机上建立一个数据库。这样你就不会漏掉任何细节！另外，我在会议和讲座上做过一些非正式的调查，发现几乎每个 DBA 都是从开发人员做起的。通过控制自己的数据库，开发人员能积累丰富的经验并得到最好的培训，能够了解数据库到底是怎样工作的，从长远来看这绝对是一笔不小的财富。

要使用 runstats，需要能访问几个 V$视图，并创建一个表来存储统计结果，还要创建 runstats 包。为此，需要访问 3 个 V$表（就是那些神奇的动态性能表）：V$STATNAME、V$MYSTAT 和 V$LATCH。以下是我使用的视图：

```
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, gets

from v$latch;
```

如果你能得到 V$STATNAME、V$MYSTAT、V$LATCH 和 V$TIMER 的直接授权，就能直接对这些表执行 SELECT 操作（相应地可以自行创建视图）；否则，可以由其他人对这些表执行 SELECT 操作为你创建视图，并授予你在这个视图上执行 SELECT 的权限。

一旦建立视图，接下来只需要一个小表来收集统计结果：

```
create global temporary table run_stats

( runid varchar2(15),

  name varchar2(80),

  value int )

on commit preserve rows;
```

最后，需要创建 runstats 包。其中包含 3 个简单的 API 调用：

- runstats 测试开始时调用 RS_STAT（runstats 开始）。
- 正如你想象的，RS_MIDDLE 会在测试之间调用。
- 完成时调用 RS_STOP，打印报告。

创建 runstats 包的规范如下：

```
ops$tkyte@ORA920> create or replace package runstats_pkg

  2 as

  3 procedure rs_start;

  4 procedure rs_middle;

  5 procedure rs_stop( p_difference_threshold in number default 0 );

  6 end;

  7 /

Package created.
```

参数 P_DIFFERENCE_THRESHOLD 用于控制最后打印的数据量。runstats 会收集并得到每次运行的统计结果和闩信息，然后打印一个报告，说明每次测试（每个方法）使用了多少资源，以及不同测试（不同方法）的结果之差。可以使用这个输入参数来控制只查看差值大于这个数的统计结果和闩信息。由于这个参数默认为 0，所以默认情况下可以看到所有输出。

下面我们逐一分析包体中的过程。包前面是一些全局变量，这些全局变量用于记录每次运行的耗用时间：

```
ops$tkyte@ORA920> create or replace package body runstats_pkg

  2 as

  3
```

```
4 g_start number;

5 g_run1 number;

6 g_run2 number;

7
```

下面是 RS_START 例程。这个例程只是清空保存统计结果的表，并填入“上一次”（before）得到的统计结果和闩信息。然后获得当前定时器值，这是一种时钟，可用于计算耗用时间（单位百分之一秒）：

```
8 procedure rs_start

9 is

10      begin

11              delete from run_stats;

12

13              insert into run_stats

14              select 'before', stats.* from stats;

15

16              g_start := dbms_utility.get_time;

17      end;

18
```

接下来是 RS_MIDDLE 例程。这个例程只是把第一次测试运行的耗用时间记录在 G_RUN1 中。然后插入当前的一组统计结果和闩信息。如果把这些值与先前在 RS_START 中保存的值相减，就会发现第一个方法使用了多少闩，以及使用了多少游标（一种统计结果），等等。

最后，记录下一次运行的开始时间：

```
19      procedure rs_middle

20      is

21      begin

22              g_run1 := (dbms_utility.get_time-g_start);
```

```
23
24              insert into run_stats
25              select 'after 1', stats.* from stats;
26              g_start := dbms_utility.get_time;
27
28      end;
29
30      procedure rs_stop(p_difference_threshold in number default 0)
31      is
32      begin
33              g_run2 := (dbms_utility.get_time-g_start);
34
35              dbms_output.put_line
36                      ( 'Run1 ran in ' || g_run1 || ' hsecs' );
37              dbms_output.put_line
38                      ( 'Run2 ran in ' || g_run2 || ' hsecs' );
39              dbms_output.put_line
40                      ( 'run 1 ran in ' || round(g_run1/g_run2*100,2) ||
41                      '% of the time' );
42              dbms_output.put_line( chr(9) );
43
44              insert into run_stats
45                      select 'after 2', stats.* from stats;
```

```
46
47              dbms_output.put_line
48                      ( rpad( 'Name', 30 ) || lpad( 'Run1', 10 ) ||
49                      lpad( 'Run2', 10 ) || lpad( 'Diff', 10 ) );
50
51              for x in
52                      ( select rpad( a.name, 30 ) ||
53                              to_char( b.value-a.value, '9,999,999' ) ||
54                              to_char( c.value-b.value, '9,999,999' ) ||
55                              to_char( ( (c.value-b.value)-(b.value-a.value)), '9,999,999' )
data
56                      from run_stats a, run_stats b, run_stats c
57                      where a.name = b.name
58                              and b.name = c.name
59                              and a.runid = 'before'
60                              and b.runid = 'after 1'
61                              and c.runid = 'after 2'
62                              and (c.value-a.value) > 0
63                              and abs( (c.value-b.value) - (b.value-a.value) )
64                                      > p_difference_threshold
65                      order by abs( (c.value-b.value)-(b.value-a.value))
66              ) loop
67              dbms_output.put_line( x.data );
68              end loop;
```

```
69
70              dbms_output.put_line( chr(9) );
71              dbms_output.put_line
72                      ( 'Run1 latches total versus runs -- difference and pct' );
73              dbms_output.put_line
74                      ( lpad( 'Run1', 10 ) || lpad( 'Run2', 10 ) ||
75                      lpad( 'Diff', 10 ) || lpad( 'Pct', 8 ) );
76
77              for x in
78                      ( select to_char( run1, '9,999,999' ) ||
79                              to_char( run2, '9,999,999' ) ||
80                              to_char( diff, '9,999,999' ) ||
81                              to_char( round( run1/run2*100,2 ), '999.99' ) || '%' data
82                       from ( select sum(b.value-a.value) run1, sum(c.value-b.value)
run2,
83                                      sum( (c.value-b.value)-(b.value-a.value)) diff
84                              from run_stats a, run_stats b, run_stats c
85                              where a.name = b.name
86                                      and b.name = c.name
87                                      and a.runid = 'before'
88                                      and b.runid = 'after 1'
89                                      and c.runid = 'after 2'
90                                      and a.name like 'LATCH%'
91                              )
```

```
 92                         ) loop
 93                                 dbms_output.put_line( x.data );
 94                         end loop;
 95                 end;
 96
 97         end;
 98 /

Package body created.
```

下面可以使用 runstats 了。我们将通过例子来说明如何使用 runstats 对批量插入（INSERT）和逐行处理进行比较，看看哪种方法效率更高。首先建立两个表，要在其中插入 1,000,000 行记录：

```
ops$tkyte@ORA10GR1> create table t1
  2 as
  3 select * from big_table.big_table
  4 where 1=0;

Table created.



ops$tkyte@ORA10GR1> create table t2
  2 as
  3 select * from big_table.big_table
  4 where 1=0;

Table created.
```

接下来使用第一种方法插入记录，也就是使用单独一条 SQL 语句完成批量插入。首先调用 RUNSTATS_PKG.RS_START：

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_start;

PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10GR1> insert into t1 select * from big_table.big_table;

1000000 rows created.



ops$tkyte@ORA10GR1> commit;

Commit complete.
```

下面准备执行第二种方法，即逐行地插入数据：

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_middle;

PL/SQL procedure successfully completed.



ops$tkyte@ORA10GR1> begin

  2 for x in ( select * from big_table.big_table )

  3 loop

  4 insert into t2 values X;

  5 end loop;

  6 commit;

  7 end;

  8 /

PL/SQL procedure successfully completed.
```

最后生成报告：

```
Name                                  Run1        Run2        Dif
f

STAT...recursive
calls                                 8,089       1,015,451   1,007,362

STAT...db                                                             block
changes                               109,355     2,085,099   1,975,744
```

```
LATCH.library
cache                          9,914          2,006,563      1,996,649

LATCH.library                                                        cache
pin                            5,609          2,003,762      1,998,153

LATCH.cache                                                          buffers
chains                       575,819          5,565,489      4,989,670

STAT...undo                                     change                vector
size                       3,884,940         67,978,932     64,093,992

STAT...redo
size                     118,854,004        378,741,168    259,887,164



Run1 latches total versus runs -- difference and pct

Run1                                  Run2            Diff                Pc
t

825,530                               11,018,773      10,193,243          7.49%



PL/SQL procedure successfully completed.
```

mystat

mystat.sql 和相应的 mystat2.sql 用于展示完成某操作之前和之后的某些 Oracle“统计结果”的变化情况。mystat.sql 只是获得统计结果的开始值：

```
set echo off

set verify off

column value new_val V

define S="&1"



set autotrace off

select a.name, b.value
```

```
from v$statname a, v$mystat b

where a.statistic# = b.statistic#

and lower(a.name) like '%' || lower('&S')||'%'

/

set echo on
```

mystat2.sql 用于报告统计结果的变化情况（差值）：

```
set echo off

set verify off

select a.name, b.value V, to_char(b.value-&V,'999,999,999,999') diff

from v$statname a, v$mystat b

where a.statistic# = b.statistic#

and lower(a.name) like '%' || lower('&S')||'%'

/

set echo on
```

例如，要查看某个 UPDATE 生成的 redo 数，可以使用以下命令：

```
big_table@ORA10G> @mystat "redo size"

big_table@ORA10G> set echo off



NAME                                                  VALUE

----------------------------                          ----------

redo size                                             496



big_table@ORA10G> update big_table set owner = lower(owner)

2 where rownum <= 1000;
```

```
1000 rows updated.

big_table@ORA10G> @mystat2
big_table@ORA10G> set echo off

NAME                                 V                    DIFF
------------------------------       ----------           ----------------
redo size                            89592                89,096
```

由此可见，1,000 行记录的 UPDATE 会生成 89.096 字节的 redo。

SHOW_SPACE

SHOW_SPACE 例程用于打印数据库段空间利用率信息。其接口如下：

```
ops$tkyte@ORA10G> desc show_space
PROCEDURE show_space
Argument Name                  Type                    In/Out   Default?
------------------------------ ----------------------  ------   --------
P_SEGNAME                      VARCHAR2                IN
P_OWNER                        VARCHAR2                IN       DEFAULT
P_TYPE                         VARCHAR2                IN       DEFAULT
P_PARTITION                    VARCHAR2                IN       DEFAULT
```

参数如下：

- P_SEGNAME：段名（例如，表或索引名）。
- P_OWNER：默认为当前用户，不过也可以使用这个例程查看另外某个模式。
- P_TYPE：默认为 TABLE，这个参数表示查看哪种类型的对象（段）。例如，SELECT DISTINCT SEGMENT_TYPE FROM DBA_SEGMENTS 会列出合法的段类型。

- P_PARTITION：显示分区对象的空间时所用的分区名。SHOW_SPACE 一次只显示一个分区的空间利用率。

这个例程的输出如下，这里段位于一个自动段空间管理（Automatic Segment Space Management， ASSM）表空间中：

```
big_table@ORA10G> exec show_space('BIG_TABLE');

Unformatted Blocks .....                0

FS1 Blocks (0-25) .....                 0

FS2 Blocks (25-50) .....                0

FS3 Blocks (50-75) .....                0

FS4 Blocks (75-100).....                0

Full Blocks .....                       14,469

Total Blocks.....                       15,360

Total Bytes.....                        125,829,120

Total MBytes.....                       120

Unused Blocks.....                      728

Unused Bytes.....                       5,963,776

Last Used Ext FileId.....               4

Last Used Ext BlockId.....              43,145

Last Used Block.....                    296


PL/SQL procedure successfully completed.
```

报告的各项结果说明如下：

- Unformatted Blocks：为表分配的位于高水位线（high-water mark, HWM）之下但未用的块数。把未格式化和未用的块加在一起，就是已为表分配但从未用于保存 ASSM 对象数据的总块数。
- FS1 Blocks-FS4 Blocks：包含数据的格式化块。项名后的数字区间表示各块的“空闲度”。例如，（0-25）是指空闲度为 0~25%的块数。
- Full Blocks：已满的块数，不能再对这些执行插入。

- Total Blocks、Total bytes、Total Mbytes：为所查看的段分配的总空间量，单位分别是数据库块、字节和兆字节。
- Unused Blocks、Unused Bytes：表示未用空间所占的比例（未用空间量）。这些块已经分配给所查看的段，但目前在段的 HWM 之上。
- Last Used Ext FileId：最后使用的文件的文件 ID，该文件包含最后一个含数据的区段（extent）。
- Last Used Ext BlockId：最后一个区段开始处的块 ID；这是最后使用的文件中的块 ID。
- Last Used Block：最后一个区段中最后一个块的偏移量。

如果对象在用户空间管理的表空间中，使用 SHOW_SPACE 查看时，输出如下：

```
big_table@ORA10G> exec show_space( 'BIG_TABLE' )

Free Blocks.....                                1
Total Blocks.....                         147,456
Total Bytes.....                    1,207,959,552
Total MBytes.....                           1,152
Unused Blocks.....                          1,616
Unused Bytes.....                      13,238,272
Last Used Ext FileId.....                       7
Last Used Ext BlockId.....                139,273
Last Used Block.....                        6,576

PL/SQL procedure successfully completed.
```

这里惟一的区别是报告中最前面的 Free Blocks 项。这是段的第一个 freelist（自由列表）组中的块数。我的脚本只测试了第一个 freelist 组。如果你想测试多个 freelist 组，还需要修改这个脚本。

我为以下代码加了注释。这个实用程序直接调用了数据库的 DBMS_SPACE API。

```
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 )
-- this procedure uses authid current user so it can query DBA_*
-- views using privileges from a ROLE, and so it can be installed
-- once per database, instead of once per user who wanted to use it
 authid current_user
as
        l_free_blks number;
        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;
        l_segment_space_mgmt varchar2(255);
        l_unformatted_blocks number;
        l_unformatted_bytes number;
        l_fs1_blocks number; l_fs1_bytes number;
        l_fs2_blocks number; l_fs2_bytes number;
        l_fs3_blocks number; l_fs3_bytes number;
        l_fs4_blocks number; l_fs4_bytes number;
        l_full_blocks number; l_full_bytes number;
        -- inline procedure to print out numbers nicely formatted
```

```
        -- with a simple label
        procedure p( p_label in varchar2, p_num in number )
        is
        begin
                dbms_output.put_line( rpad(p_label,40,'.') ||
                to_char(p_num,'999,999,999,999') );
        end;
begin
        -- this query is executed dynamically in order to allow this procedure
        -- to be created by a user who has access to DBA_SEGMENTS/TABLESPACES
        -- via a role as is customary.
        -- NOTE: at runtime, the invoker MUST have access to these two
        -- views!
        -- this query determines if the object is an ASSM object or not
        begin
                execute immediate
                        'select ts.segment_space_management
                        from dba_segments seg, dba_tablespaces ts
                        where seg.segment_name = :p_segname
                                and (:p_partition is null or
                                        seg.partition_name = :p_partition)
                                and seg.owner = :p_owner
                                and seg.tablespace_name = ts.tablespace_name'
```

```
                        into l_segment_space_mgmt
                        using p_segname, p_partition, p_partition, p_owner;
                exception
                        when too_many_rows then
                        dbms_output.put_line
                        ( 'This must be a partitioned table, use p_partition => ');
                        return;
                end;
                -- if the object is in an ASSM tablespace, we must use this API
                -- call to get space information, otherwise we use the FREE_BLOCKS
                -- API for the user-managed segments
                if l_segment_space_mgmt = 'AUTO'
                then
                        dbms_space.space_usage
                        ( p_owner, p_segname, p_type, l_unformatted_blocks,
                        l_unformatted_bytes, l_fs1_blocks, l_fs1_bytes,
                        l_fs2_blocks, l_fs2_bytes, l_fs3_blocks, l_fs3_bytes,
                                l_fs4_blocks, l_fs4_bytes, l_full_blocks, l_full_bytes,
p_partition);
                        p( 'Unformatted Blocks ', l_unformatted_blocks );
                        p( 'FS1 Blocks (0-25) ', l_fs1_blocks );
                        p( 'FS2 Blocks (25-50) ', l_fs2_blocks );
                        p( 'FS3 Blocks (50-75) ', l_fs3_blocks );
                        p( 'FS4 Blocks (75-100)', l_fs4_blocks );
```

```
                p( 'Full Blocks ', l_full_blocks );
        else
                dbms_space.free_blocks(
                        segment_owner => p_owner,
                        segment_name => p_segname,
                        segment_type => p_type,
                        freelist_group_id => 0,
                        free_blks => l_free_blks);
                p( 'Free Blocks', l_free_blks );
        end if;
        -- and then the unused space API call to get the rest of the
        -- information
        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( 'Total MBytes', trunc(l_total_bytes/1024/1024) );

        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;

/
```

BIG_TABLE

在全书的例子中，我使用了一个名为 BIG_TABLE 的表。根据所用的系统，这个表的记录数在 1 条和 400 万条之间，而且大小也不定，为 200～800MB。不过，不论这样，表结构都是一样的。

为了创建 BIG_TABLE，我编写了一个可以完成以下功能的脚本：

- 根据 ALL_OBJECTS 创建一个空表。这个字典视图用于填充 BIG_TABLE。
- 置这个表为 NOLOGGING。这是可选的，我之所以这样做是为了提高性能。对测试表使用 NOLOGGING 模式是安全的；由于生产系统中不会使用这样一个测试表，所以不会启用诸如 Oracle Data Guard 之类的特性。
- 用 ALL_OBJECTS 的内容填充表，然后迭代地插入其自身中，每次迭代会使表大小几乎加倍。
- 对这个表创建一个主键约束。
- 收集统计结果。
- 显示表中的行数。

要建立 BIG_TABLE 表，可以在 SQL*Plus 提示窗口运行以下脚本，传入希望在表中插入多少行记录。如果达到这个行数，脚本则停止执行。

```
create table big_table
```

```
as
select rownum id, a.*
        from all_objects a
where 1=0
/
alter table big_table nologging;
declare
        l_cnt number;
        l_rows number := &1;
begin
        insert /*+ append */
                into big_table
        select rownum, a.*
                from all_objects a;
        l_cnt := sql%rowcount;
        commit;
        while (l_cnt < l_rows)
        loop
                insert /*+ APPEND */ into big_table
                select rownum+1_cnt,
                        OWNER, OBJECT_NAME, SUBOBJECT_NAME,
                        OBJECT_ID, DATA_OBJECT_ID,
                        OBJECT_TYPE, CREATED, LAST_DDL_TIME,
```

```
                        TIMESTAMP, STATUS, TEMPORARY,

                        GENERATED, SECONDARY

                from big_table

                where rownum <= l_rows-l_cnt;

                l_cnt := l_cnt + sql%rowcount;

                commit;

        end loop;

end;

/



alter table big_table add constraint

big_table_pk primary key(id)

/

begin

        dbms_stats.gather_table_stats

        ( ownname => user,

        tabname => 'BIG_TABLE',

        method_opt => 'for all indexed columns',

        cascade => TRUE );

end;

/

select count(*) from big_table;
```

这里会收集与主键相关的表和索引的基准统计结果。另外，我还收集了加索引的列的统计直方图（这是我的一贯做法）。也可以收集其他列的直方图，但对 big_table 表来说没有必要这样做。

代码约定

需要指出这本书里使用的一个编码约定，也就是 PL/SQL 代码中的变量如何命名。例如，考虑如下包体：

```
create or replace package body my_pkg
as
        g_variable varchar2(25);
        procedure p( p_variable in varchar2 )
        is
                l_variable varchar2(25);
        begin
                null;
        end;
end;
```

这里有 3 个变量，一个全局包变量 G_VARIABLE，一个过程形参 P_VARIABLE，还有一个局部变量 L_VARIABLE。我是根据变量的作用域来命名的。所有全局变量都以 G_开头，参数用 P_开头，局部变量用 L_开头。这样做的这样原因是为了区别 PL/SQL 变量和数据库表中的列。例如，如果有以下过程：

```
create procedure p( ENAME in varchar2 )
as
begin
        for x in ( select * from emp where ename = ENAME ) loop
                Dbms_output.put_line( x.empno );
        end loop;
end;
```

EMP 表中 ENAME 非空的所有行都会打印出来。SQL 看到 ename=ENAME 时，它会把 ENAME 列与自身比较（这是自然的）。为了避免这种错误，可以使用 ename=P.ENAME，也就是说，用过程名来限定对 PL/SQL 变量的引用，但很容易忘记加过程名进行限定；一旦忘了，就会导致错误。

我总是根据作用域来对变量命名。这样一来，可以很容易地把参数与局部变量和全局变量区分开，而且还可以消除列名和变量名的任何歧义。

第 1 章 开发成功的 Oracle 应用程序

我花了大量时间使用 Oracle 数据库软件，更确切地讲，一直在与和使用 Oracle 数据库软件的人打交道。在过去的 18 年间，我参与过许多项目，有的相当成功，有点却彻底失败，如果把这些经验用几句话来概括，可以总结如下：

- 基于数据库（或依赖于数据库）构建的应用是否成功，这取决于如何使用数据库。另外，从我的经验看，所有应用的构建都围绕着数据库。如果一个应用未在任何地方持久地存储数据，很难想象这个应用真的有用。
- 应用总是在“来来去去”，而数据不同，它们会永远存在。从长远来讲，我们的目标并不是构建应用，而应该是如何使用这些应用底层的数据。

- 开发小组的核心必须有一些精通数据库的开发人员，他们要负责确保数据库逻辑是可靠的，系统能够顺利构建。如果已成事实（应用已经部署）之后再去调优，这通常表明，在开发期间你没有认真考虑这些问题。

这些话看上去再显然不过了。然而，我发现太多的人都把数据库当成是一个黑盒（black box），好像不需要对它有输入了解。他们可能有一个 SQL 生成器，认为有了这个工具，就不需要再费功夫去学 SQL 语言。也可能认为使用数据库就像使用平面文件一样，只需要根据索引读数据就行。不管他们怎么想，有一点可以告诉你，如果按这种思路来考虑，往往会被误导：不了解数据库，你将寸步难行。这一章将讨论为什么需要了解数据库，具体地讲，就是为什么需要理解以下内容：

- 数据库的体系结构，数据库如何工作，以及有怎样的表现。
- 并发控制是什么，并发控制对你意味着什么。
- 性能、可扩缩性和安全性都是开发时就应该考虑的需求，必须适当地做出设计，不要指望能碰巧满足这些需求。
- 数据库的特性如何实现。某个特定数据库特性的实际实现方式可能与你想象的不一样。你必须根据数据库实际上如何工作（而不是认为它应该如何工作）来进行设计。
- 数据库已经提供了哪些特性，为什么使用数据库已提供的特性要优于自行构建自己的特性。
- 为什么粗略地了解 SQL 还不够，还需要更深入地学习 SQL。
- DBA 和开发人员都在为同一个目标努力，他们不是敌对的两个阵营，不是想在每个回合中比试谁更聪明。

初看上去，好像要学习的东西还不少，不过可以做个对照，请考虑这样一个问题：如果你在一个全新的操作系统（operating system，OS）上开发一个高度可扩缩的企业应用，首先要做什么？希望你的答案是：“了解这个新操作系统如何工作，应用在它上面怎样运行，等等”。如果不是这样，你的开发努力将会付诸东流。

例如，可以考虑一下 Windows 和 Linux，它们都是操作系统。能为开发人员提供大致相同的一组服务，如文件管理、内存管理、进程管理、安全性等。不过，这两个操作系统的体系结构却大相径庭。因此，如果你一直是 Windows 程序员，现在给你一个任务，让你在 Linux 平台上开发新应用，那么很多东西都得从头学起。内存管理的处理就完全不同。建立服务器进程的方式也有很大差异。在 Window 下，你会开发一个进程，一个可执行程序，但有许多线程。在 Linux 下则不同，不会开发单个独立的可执行程序；相反，会有多个进程协作。总之，你在 Windows 环境下学到的许多知识到了 Linux 上并不适用（公平地讲，反之亦然）。要想在新平台上也取得成功，你必须把原来的一些习惯丢掉。

在不同操作系统上运行的应用存在上述问题，基于不同数据库运行的应用也存在同样的问题：你要懂得，数据库对于成功至关重要。如果不了解你的数据库做什么，或者不清楚它怎么做，那你的应用很可能会失败。如果你认为应用在 SQL Server 上能很好地运行，那它在 Oracle 上也肯定能很好地工作，你的应用往往会失败。另外，公平地讲，反过来也一样：一个 Oracle 应用可能开发得很好，可扩缩性很好，但是如果不对体系结构做重大改变，它在 SQL Server 上不一定能正常运行。Windows 和 Linux 都是操作系统，但是二者截然不同，同样地，Oracle 和 SQL Server（甚至可以是任何其他数据库）尽管都是数据库，但二者

也完全不同。

1.1 我的方法

在阅读下面的内容之前，我 觉得有必要先解释一下我的开发方法。针对问题，我喜欢采用一种以数据库为中心的方法。如果能在数据库中完成，我肯定就会让数据库来做，而不是自行实现。对 此有几个原因。首先，也是最重要的一点，我知道如果让数据库来实现功能，应用就可以部署在任何环境中。据我所知，没有哪个流行的服务器操作系统不支持Oracle；从 Windows 到一系列 UNIX/Linux 系统，再到 OS/390 大型机系统，都支持 Oracle 软件和诸多选项。我经常在我的笔记本电脑上构建和测试解决方案，其中在 Linux 或 Windows XP 上（或者使用 VMware 来模拟这些环境）运行 Oracle9i/Oracle 10g。这样一来，我就能把这些解决方案部署在运行相同数据库软件但有不同操作系统的多种服务器上。我发现，如果某个特性不是在数据库中实现，要想在希望的任何环境中部署这个特性将极其困难。Java 语言之所以让人趋之若鹜，一个主要原因就是 Java 程序总在同样的虚拟环境［即 Java 虚拟机（Java Virtual Machine，JVM）］中编译，这使得这些程序的可移植性很好。有意思的是，也正是这个特性让我对数据库着迷不已。数据库就是我的“虚拟机”，它也是我的“虚拟操作系统”。

前面已经提到，我采用的方法是尽可能在数据库中实现功能。如果数据库环境无法满足我的需求，我也会在数据库之外使用 Java 或 C 来实现。采用这种方式，操作系统的复杂细节对我来说几乎是隐藏的。我要了解我的“虚拟机”如何工作（也就是 Oracle 如何工作，有时可能还需要用到 JVM），毕竟，起码要了解自己使用的工具才行。不过，至于在一个给定的操作系统上怎么才能最好地工作，这些都由“虚拟机”来负责。

所以，只需知道这个“虚拟操作系统”的细节，我构建的应用就能在多种操作系统上很好地工作和扩缩。我并不是暗示你可以完全忽略底层的操作系统。不过，作为一个构建数据库应用的软件开发人员，还是尽量避开它比较好，你不必处理操作系统的诸多细微之处。应该让你的 DBA（负责运行 Oracle 软件）来考虑如何适应操作系统（如果他或她做不到，你就该换个新的 DBA 了！）。如果你在开发客户/服务器软件，而且大量代码都是在数据库和虚拟机（VM；JVM 可能是最流行的虚拟机了）之外实现，那你还得再次考虑你的操作系统。

对于开发数据库软件，我有一套很简单的哲学，这是我多年以来一直信守的思想：

- 如果可能，尽量利用一条 SQL 语句完成工作。
- 如果无法用一条 SQL 语句完成，就通过 PL/SQL 实现（不过，尽可能少用 PL/SQL！）。
- 如果在 PL/SQL 中也无法做到（因为它缺少一些特性，如列出目录中的文件），可以试试使用 Java 存储过程来实现。不过，有了 Oracle9i 及以上版本后，如今需要这样做的可能性极小。
- 如果用 Java 还办不到，那就在 C 外部过程中实现。如果速度要求很高，或者要使用采用 C 编写的一个第三方 API，就常常使用这种做法。
- 如果在 C 外部例程中还无法实现，你就该好好想想有没有必要做这个工作了。

在这本书中，你会看到我是怎样将上述思想付诸实现的。我会尽可能使用 SQL，充分利用它强大的新功能，如使用分析函数来解决相当复杂的问题，而不是求助于过程性代码。如果需要，我会使用 PL/SQL 和 PL/SQL 中的对象类型来完成 SQL 本身办不到的事情。PL/SQL 发展至今已经有很长时间了，它得到了长达 18 年的调整和优化。实际上，Oracle 10g

编译器本身就首次重写为一个优化编译器。你会发现，没有哪种语言能像 PL/SQL 这样与 SQL 如此紧密地耦合，也没有哪种语言得到如此优化，可以与 SQL 更好地交互。在 PL/SQL 中使用 SQL 是一件相当自然的事情，而在几乎所有其他语言（从 Visual Basic 到 Java）中，使用 SQL 感觉上都很麻烦。对于这些语言来说，使用 SQL 绝对没有“自然”的感觉；它不是这些语言本身的扩缩。如果 PL/SQL 还无法做到（这在 Oracle 9i 或 10g 中可能相当少见），我们会使用 Java。有时，如果 C 是惟一的选择，或者需要 C 才能提供的高速度，我们也会用 C 来完成工作，不过这往往是最后一道防线。随着本地 Java 编译（native Java compilation）的闪亮登场（可以把 Java 字节码转换为具体平台上特定于操作系统的对象码），你会发现，在许多情况下，Java 与 C 的运行速度相差无几。所以，需要用到 C 的情况越来越少。

1.2 黑盒方法

根据我个人的第一手经验（这表示，在学习软件开发时我自己也曾犯过错误），我对基于数据库的软件开发为什么如此频繁地遭遇失败有一些看法。先来澄清一下，这 里提到的这些项目可能一般不算失败，但是启用和部署所需的时间比原计划多出许多，原因是需要大幅重写，重新建立体系结构，或者需要充分调优。我个人把这些 延迟的项目称为“失败”，因为它们原本可以按时完成（甚至可以更快完成）。

数据库项目失败的最常见的一个原因是对数据库的实际认识不足，缺乏对所用基本工具的了解。黑盒方法是指有意让开发人员对数据库退避三舍，甚至鼓励开发人员根本不要学习数据库！在很多情况下，开发人员没有充分利用数据库。这种方法的出现，原因可以归结为 FUD［恐惧（fear）、不确定（uncertainty）和怀疑（doubt）］。一般都认为数据库“很难”，SQL、事务和数据完整性都“很难”。所以“解决方法”就是：不要卷入难题中，要知难而退。他们把数据库当成一个黑盒，利用一些软件工具来生成所有代码。他们试图利用重重保护与数据库完全隔离，以避免接触这么“难”的数据库。

我一直很难理解这种数据库开发方法，原因有二。一个原因是对我来说，学习 Java 和 C 比学习数据库基本概念要难多了。现在我对 Java 和 C 已经很精通，但是在能熟练使用 Java 和 C 之前我经受了许多磨炼，而掌握数据库则没有这么费劲。对于数据库，你要知道它是怎么工作的，但无需了解每一个细节。用 C 或 Java 编程时，则确实需要掌握每一个细枝末节，而这些语言实在是很“庞大”。

让我无法理解这种方法的另一个原因是，构建数据库应用时，最重要的软件就是数据库。成功的开发小组都会认识到这一点，而且每个开发人员都要了解数据库，并把重点放在数据库上。但我接触到的许多项目中，情况却几乎恰恰相反。例如，下面就是一种典型的情况：

- 在构建前端所用的 GUI 工具或语言（如 Java）方面，开发人员得到了充分的培训。在很多情况下，他们会有数周甚至数月的培训。
- 开发人员没有进行过 Oracle 培训，也没有任何 Oracle 经验。大多数人都没有数据库经验，所以并未理解如何使用核心的数据库构造（如各种可用的索引和表结构 。
- 开发人员力图谨守“数据库独立性”这一原则，但是出于许多原因，他们可能做不到。最明显的一个原因是：他们对于数据库没有足够的了解，也不清楚这些数据库可能有什么区别。这样一个开发小组无法知道要避开数据库的哪些特性才能保证数据库独立性。

- 开发人员遇到大量性能问题、数据完整性问题、挂起问题等（但这些应用的界面往往很漂亮）。

因为出现了无法避免的性能问题，他们把我找来，要求帮助解决这些难题。我最早就是从构建数据库独立的应用做起的（从某种程度上讲，在 ODBC 问世之前，我就已经编写了自己的 ODBC 驱动程序），我知道哪些地方可能会犯错误，因为我以前就曾犯过这些错误。我总会查看是否存在下面这些问题：存在效率低下的 SQL；有大量过程性代码，但这些工作原本用一条 SQL 语句就足够了；为了保持数据库独立性，没有用到新特性（1995 年以后的新特性都不敢用），等等。

我还记得这样一个特例，有个项目找我来帮忙。当时要用到一个新命令，但我记不清那个新命令的语法。所以我让他们给我拿一本 SQL Reference 手册，谁知他们给了我一本 Oracle 6.0 文档。那个项目开发用的是 7.3 版本，要知道，6.0 版本和 7.3 版本之间整整相差 5 年！7.3 才是所有开发人员使用的版本，但似乎谁都不关心这一点。不用说他们需要了解的跟踪和调优工具在 6.0 版本中甚至不存在。更不用说在这 5 年间又增加了诸如触发器、存储过程和数百个其他特性，这些都是编写 6.0 版文档（也就是他们现在参考的文档）时根本没有的特性。由此很容易看出他们为什么需要帮助，但解决起来就是另一码事了。

注意 甚至时至今日，已经到了 2005 年，我还是经常发现有些数据库应用开发人员根本不花些时间来看文档。我的网站（http://asktom.oracle.com）上经常会有：“……的语法是什么”这样的问题，并解释说“我们拿不到文档，所以请告诉我们”。对于许多这样的问题，我拒绝直接做出回答，而是会把在线文档的地址告诉他们。无论你身处何地，都能免费得到这些文档。在过去 10 年中，“我们没有文档”或“我们无法访问资源”之类的借口已经站不住脚了。如今已经有了诸如http://otn.oracle.com（Oracle技术网络）和http://groups. google.com (Google Groups Usenet论坛）等网站，它们都提供了丰富的资源，如果你手边还没有一套完整的文档，那就太说不过去了！

构建数据库应用的开发人员要避开数据库的主张实在让我震惊，不过这种做法还顽固不化地存在着。许多人还认为开发人员没办法花那么多时间来进行数据库培训，而且他们根本不需要了解数据库。为什么？我不止一次地听到这样的说法：“Oracle 是世界上最可扩缩的数据库，所以我们不用了解它，它自然会按部就班地把事情做好的。”Oracle 是世界上最可扩缩的数据库，这一点没错。不过，用 Oracle 不仅能写出好的、可扩缩的代码，也同样能很容易地写出不好的、不可扩缩的代码（这可能更容易）。把这句话里的“Oracle”替换为其他任何一种技术的名字，这句话仍然正确。事实是：编写表现不佳的应用往往比编写表现优秀的应用更容易。如果你不清楚自己在做什么，可能会发现你打算用世界上最可扩缩的数据库建立一个单用户系统！

数据库是一个工具；不论是什么工具，如果使用不当都会带来灾难。举个例子，你想用胡桃钳弄碎胡桃，会不会把胡桃钳当锤子一样用呢？当然这也是可以的，不过这 样用胡桃钳很不合适，而且后果可能很严重，没准会重重地伤到你的手指。如果还是对你的数据库一无所知，你也会有类似的结局。

例如，最近我参与了一个项目。开发人员正饱受性能问题之苦，看上去他们的系统中许多事务在串行进行。他们的做法不是大家并发地工作，而是每个人都要排一个长长的队，苦苦等着前面的人完成后才能继续。应用架构师向我展示了系统的体系结构，这是经典的三层方法。他们想让 Web 浏览器与一个运行 JSP（JavaServer Pages）的中间层应用服务器通信。JSP 再使用另一个 EJB（Enterprise JavaBeans）层，在这一层执行所有 SQL。EJB 中的 SQL 由某个第三方工具生成，这是采用一种数据库独立的方式完成的。

现在看来，对这个系统很难做任何诊断，因为没有可测量或可跟踪的代码。测量代码（instrumenting code）堪称一门艺术，可以把开发的每行代码变成调试代码，这样就能跟踪应用的执行，遇到性能、容量甚至逻辑问题时就能跟踪到问题出在哪里。在这里，我们只能肯定地说问题出在“浏览器和数据库之间的某个地方”。换句话说，整个系统都是怀疑对象。对此有好消息也有坏消息。一方面，Oracle 数据库完全可测量；另一方面，应用必须能够在适当的位置打开和关闭测量，遗憾的是，这个应用不具备这种能力。

所以，我们面对的困难是，要在没有太多细节的情况下诊断出导致性能问题的原因，我们只能依靠从数据库本身收集的信息。一般地，要分析应用的性能问题，采用应用级跟踪更合适。不过，幸运的是，这里的解决方案很简单。通过查看一些 Oracle V$表（V$ 表是 Oracle 提供其测量结果或统计信息的一种方法），可以看出，竞争主要都围绕着一个表，这是一种排队表。结论是根据 V$LOCK 视图和 V$SQL 做出的，V$LOCK 视图可以显示阻塞的会话，V$SQL 会显示这些阻塞会话试图执行的 SQL。应用想在这个表中放记录，而另外一组进程要从表中取出记录并进行处理。通过更深入地“挖掘”，我们发现这个表的 PROCESSED_FLAG 列上有一个位图索引。

注意 第 12 章会详细介绍位图索引，并讨论为什么位图索引只适用于低基数值，但是对频繁更新的列不适用。

原因在于，PROCESSED_FLAG 列只有两个值：Y 和 N。对于插入到表中的记录，该列值为 N（表示未处理）。其他进程读取和处理这个记录时，就会把该列值从 N 更新为 Y。这些进程要很快地找出 PROCESSED_FLAG 列值为 N 的记录，所以开发人员知道，应该对这个列建立索引。他们在别处了解到，位图索引适用于低基数（low-cardinality）列，所谓低基数列就是指这个列只有很少的可取值，所以看上去位图索引是一个很自然的选择。

不过，所有问题的根由正是这个位图索引。采用位图索引，一个键指向多行，可能数以百计甚至更多。如果更新一个位图索引键，那么这个键指向的数百条记录会与你实际更新的那一行一同被有效地锁定。

所以，如果有人插入一条新记录（PROCESSED_FLAG 列值为 N），就会锁定位图索引中的 N 键，而这会有效地同时锁定另外数百条 PROCESSED_FLAG 列值为 N 的记录（以下记作 N 记录）。此时，想要读这个表并处理记录的进程就无法将 N 记录修改为 Y 记录（已处理的记录）。原因是，要想把这个列从 N 更新为 Y，需要锁定同一个位图索引键。实际上，想在这个表中插入新记录的其他会话也会阻塞，因为它们同样想对这个位图索引键锁定。简单地讲，开发人员实现了这样一组结构，它一次最多只允许一个人插入或更新！

可以用一个简单的例子说明这种情况。在此，我使用两个会话来展示阻塞很容易发生：

```
ops$tkyte@ORA10G> create table t ( processed_flag varchar2(1) );

Table created.

ops$tkyte@ORA10G> create bitmap index t_idx on t(processed_flag);

Index created.

ops$tkyte@ORA10G> insert into t values ( 'N' );

1 row created.
```

现在，如果我在另一个 SQL*Plus 会话中执行以下命令：

```
ops$tkyte@ORA10G> insert into t values ( 'N' );
```

这条语句就会“挂起”，直到在第一个阻塞会话中发出 COMMIT 为止。

这里的问题就是缺乏足够的了解造成的；由于不了解数据库特性（位图索引），不清楚它做些什么以及怎么做，就导致这个数据库从一开始可扩缩性就很差。一旦找出了问题，修正起来就很容易了。处理标志列上确实要有一个索引，但不能是位图索引。这里需要一个传统的 B*Tree 索引。说服开发人员接受这个方案很是费了一番功夫，因为这个列只有两个不同的可取值，却需要使用一个传统的索引，对此没有人表示赞同。不过，通过仿真（我很热衷于仿真、测试和试验），我们证明了这确实是正确的选择。对这个列加索引有两种方法：

- 在处理标志列上创建一个索引。
- 只在处理标志为 N 时在处理标志列上创建一个索引，也就是说，只对感兴趣的值加索引。通常，如果处理标志为 Y，我们可能不想使用索引，因为表中大多数记录处理标志的值都可能是 Y。注意这里的用辞，我没有说“我们绝对不想使用索引”，如果出于某种原因需要频繁地统计已处理记录的数目，对已处理记录加索引可能也很有用。

最后，我们只在处理标志为 N 的记录上创建了一个非常小的索引，由此可以快速地访问感兴趣的记录。

到此就结束了吗？没有，绝对没有结束。开发人员的解决方案还是不太理想。由于他们对所用工具缺乏足够的了解，我们只是修正了由此导致的主要问题，而且经过大量研究后才发现系统不能很好地测量。我们还没有解决以下问题：

- 构建应用时根本没有考虑过可扩缩性。可扩缩性必须在设计中加以考虑。
- 应用本身无法调优，甚至无法修改。经验证明，80%～90%的调优都是在应用级完成的，而不是在数据库级。
- 应用完成的功能（排队表）实际上在数据库中已经提供了，而且数据库是以一种高度并发和可扩缩的方式提供的。我指的就是数据库已有的高级排队（Advanced Queuing，AQ）软件，开发人员没有直接利用这个功能，而是在埋头重新实现。
- 开发人员不清楚 bean 在数据库中做了什么，也不知道出了问题要到哪里去查。

这个项目的问题大致如此，所以我们需要解决以下方面的问题：

- 如何对 SQL 调优而不修改 SQL。这看起来很神奇，不过在 Oracle 10g 中确实可以办得到，从很大程度上讲堪称首创。
- 如何测量性能。
- 如何查看哪里出现了瓶颈。
- 如何建立索引，对什么建立索引。
- 如此等等。

一周结束后，原本对数据库敬而远之的开发人员惊讶地发现，数据库居然能提供如此之多的功能，而且了解这些信息是如此容易。最重要的是，这使得应用的性能发生了翻天覆地的变化。最终他们的项目还是成功了，只是比预期的要晚几个星期。

这个例子不是批评诸如 EJB 和容器托管持久存储之类的工具或技术。我们要批评的是故意不去了解数据库，不去学习数据库如何工作以及怎样使用数据库这种做法。这个案例中使用的技术本来能很好地工作，但要求开发人员对数据库本身有一些深入的了解。

关键是：数据库通常是应用的基石。如果它不能很好地工作，那其他的都没有什么意义了。如果你手上有一个黑盒，它不能正常工作，你能做些什么呢？可能只能做一 件事，那就是盯着这个黑盒子发愣，搞不明白为什么它不能正常工作。你没办法修理它，也没办法进行调整。你根本不知道它是怎样工作的，最后的决定只能是保持 现状。我提倡的方法则是：了解你的数据库，掌握它是怎么工作的，弄清楚它能为你做什么，并且最大限度地加以利用。

1.3 开发数据库应用的正确（和不正确）方法

到 目前为止，我一直是通过闲聊来强调理解数据库的重要性。本章后面则会靠实验说话，明确地讨论为什么了解数据库及其工作原理对于成功的实现大有帮助（而无需 把应用写两次!）。有些问题修改起来很简单，只要你知道怎么发现这些问题即可。还有一些问题则不同，必须大动干戈地重写应用方能更正。这本书的目标之一就 是首先帮助避免问题的发生。

注意 在下面的几节中，我会谈到一些核心的 Oracle 特性，但并不深入地讨论这些特性到底是什么，也不会全面地介绍使用这些特性的全部细节。如果想了解更多的信息，建议你接着阅读本书后面的章节，或者参考相关的 Oracle 文档。

1.3.1 了解 Oracle 体系结构

最近，我参与了一个客户的项目，他运行着一个大型的生产应用。这个应用已经从 SQL Server“移植到”Oracle。之所以把“移植”一词用引号括起来，原因是我看到的大多数移植都只是“怎么能只对 SQL Server 代码做最少的改动，就让它们在 Oracle 上编译和执行”。要把应用从一个数据库真正移植到另一个数据库，这绝对是一项大工程。必须仔细检查算法，看看算法在目标数据库上能否正确地工作；诸如并发控制和锁定机制等特性在不同的数据库中实现不同，这会直接影响应用在不同数据库上如何工作。另外还要深入地分析算法，看看在目标数据库中实现这些算法是否合 理。坦率地讲，我看到的大多数应用通常都是根据“最小移植”得来的，因为这些应用最需要帮助。当然，反过来也一样：把一个 Oracle 应用简单地“移植”到 SQL Server，而且尽可能地避免改动，这也会得到一个很成问题、表现极差的应用。

无论如何，这种“移植”的目标都是扩缩应用，要支持更大的用户群。不过，“移植”应用的开发人员一方面想达到这个目的，另一方面又想尽量少出力。所以，这些开发人员往往保持客户和数据库层的体系结构基本不变，简单地将数据从 SQL Server 移到 Oracle，而且尽可能不改动代码。如果决定将原来 SQL Server 上的应用设计原封不动地用在 Oracle 上，就会导致严重的后果。这种决定最糟糕的两个后果是：

- ☐ Oracle 中采用与 SQL Server 同样的数据库连接体系结构。
- ☐ 开发人员在 SQL 中使用直接量（而不是绑定变量）。

这两个结果导致系统无法支持所需的用户负载（数据库服务器的可用内存会耗尽），即使用户能登录和使用应用，应用的性能也极差。

1. **在 Oracle 中使用一个连接**

目前 SQL Server 中有一种很普遍的做法，就是对想要执行的每条并发语句都打开一个数据库连接。如果想执行 5 个查询，可能就会在 SQL Server 中看到 5 个连接。SQL Server 就是这样设计的，就好像 Windows 是针对多线程而不是多进程设计的一样。在 Oracle 中，不论你想执行 5 个查询还是 500 个查询，都希望最多打开一个连接。Oracle 就是本着这样的理念设计的。所以，SQL Server 中常用的做法在 Oracle 中却不提倡；你可能并不想维护多个数据库连接。

不过，他们确实这么做了。一个简单的 Web 应用对于每个网页可能打开 5 个、10 个、15 个甚至更多连接，这意味着，相对于服务器原本能支持的并发用户数，现在服务器只能支持其 1/5、1/10、1/15 甚至更少的并发用户数。另外，开发人员只是在 Windows 平台本身上运行数据库，这是一个平常的 Windows XP 服务器，而没有使用 Datacenter 版本的 Windows。这说明，Windows 单进程体系结构会限制 Oracle 数据库服务器总共只有大约 1.75 GB 的 RAM。由于每个 Oracle 连接要同时处理多条语句，所以 Oracle 连接通常比 SQL Server 连接占用更多的 RAM（不过 Oracle 连接比 SQL Server 连接能干多了）。开发人员能否很好地扩缩应用，很大程度上受这个硬件的限制。尽管服务器上有 8 GB 的 RAM，但是真正能用的只有 2 GB 左右。

注意 Windows 环境中还能通过其他办法得到更多的 RAM，如利用/AWE 开关选项，但是只有诸如 Windows Server Datacenter Edition 等版本的操作系统才支持这个选项，而在这个项目中并没有使用这种版本。

针对这个问题，可能的解决方案有 3 种，无论哪一种解决方案都需要做大量工作（另外要记住，这可是在原先以为“移植”已经结束的情况下补充的工作！）。具体如下：

- 重建应用，充分考虑到这样一个事实：应用是在 Oracle 上运行，而不是在另外某个数据库上；另外生成一个页面只使用一个连接，而不是 5～15 个连接。这是从根本上解决这个问题的惟一方法。
- 升级操作系统（这可不是小事情），使用 Windows Datacenter 版本中更大的内存模型（这本身就非区区小事，而且还会带来相当复杂的数据库安装，需要一些间接的数据缓冲区和其他非标准的设置）。
- 把数据库从 Windows 系列操作系统移植到另外某个使用多进程的操作系统，以便数据库使用所安装的全部 RAM（重申一遍，这个任务也不简单）。

可以看到，以上都不是轻轻松松就能办到的。不论哪种方法，你都不会毫无芥蒂地一口应允“好的，我们下午就来办”。每种方案都相当复杂，所要解决的问题原本在 数据库“移植”阶段修正才最为容易，那是你查看和修改代码的第一个机会。另外，如果交付生产系统之前先对“可扩缩性”做一个简单的测试，就能在最终用户遭 遇苦痛之前及时地捕捉到这些问题。

2. 使用绑定变量

如果我要写一本书谈谈如何构建不可扩缩的 Oracle 应用，肯定会把“不要使用绑定变量”作为第一章和最后一章的标题重点强调。这是导致性能问题的一个主要原因，也是阻碍可扩缩性的一个重要因素。Oracle 将已解析、已编译的 SQL 连同其他内容存储在共享池（shared pool）中，这是系统全局区（System Global Area ，SGA）中一个非常重要的共享内存结构。第 4 章将详细讨论共享池。这个结构能完成“平滑”操作，但有一个前提，要求开发人员在大多数情况下都会使用绑定变量。如果你确实想让 Oracle 缓慢地运行，甚至几近停顿，只要根本不使用绑定变量就可以办到。

绑定变量（bind variable）是查询中的一个占位符。例如，要获取员工 123 的相应记录，可以使用以下查询：

```
select * from emp where empno = 123;
```

或者，也可以将绑定变量:empno 设置为 123，并执行以下查询：

```
select * from emp where empno = :empno;
```

在典型的系统中，你可能只查询一次员工 123，然后不再查询这个员工。之后，你可能会查询员工 456，然后是员工 789，如此等等。如果在查询中使用直接量（常量），那么每个查询都将是一个全新的查询，在数据库看来以前从未见过，必须对查询进行解析、限定（命名解析）、安全性检查、优化等。简单地讲，就是你执行的每条不同的语句都要在执行时进行编译。

第二个查询使用了一个绑定变量:empno，变量值在查询执行时提供。这个查询只编译一次，随后会把查询计划存储在一个共享池（库缓存）中，以便以后获取和重用这个查询计划。以上两个查询在性能和可扩缩性方面有很大差别，甚至可以说有天壤之别。

从前面的描述应该能清楚地看到，与重用已解析的查询计划（称为软解析，soft parse）相比，解析包含有硬编码变量的语句（称为硬解析，hard parse）需要的时间更长，而且要消耗更多的资源。硬解析会减少系统能支持的用户数，但程度如何不太明显。这部分取决于多耗费了多少资源，但更重要的因素是库缓存所用的闩定（latching）机制。硬解析一个查询时，数据库会更长时间地占用一种低级串行化设备，这称为闩（latch），有关的详细内容请参见第 6 章。这些闩能保护 Oracle 共享内存中的数据结构不会同时被两个进程修改（否则，Oracle 最 后会得到遭到破坏的数据结构），而且如果有人正在修改数据结构，则不允许另外的人再来读取。对这些数据结构加闩的时间越长、越频繁，排队等待闩的进程就越 多，等待队列也越长。你可能开始独占珍贵的资源。有时你的计算机显然利用不足，但是数据库中的所有应用都运行得非常慢。造成这种现象的原因可能是有人占据 着某种串行化设备，而其他等待串行化设备的人开始排队，因此你无法全速运行。数据库中只要有一个应用表现不佳，就会严重地影响所有其他应用的性能。如果只 有一个小应用没有使用绑定变量，那么即使其他应用原本设计得很好，能适当地将已解析的 SQL 放在共享池中以备重用，但因为这个小应用的存在，过一段时间就会从共享池中删除已存储的 SQL。这就使得这些设计得当的应用也必须再次硬解析 SQL。真是一粒老鼠屎就能毁了一锅汤。

如果使用绑定变量，无论是谁，只要提交引用同一对象的同一个查询，都会使用共享池中已编译的查询计划。这样你的子例程只编译一次就可以反复使用。这样做效率很高，这也正是数据库期望你采用的做法。你使用的资源会更少（软解析耗费的资源相当少），不仅如此，占用闩的时间也更短，而且不再那么频繁地需要闩。这些 都会改善应用的性能和可扩缩性。

要想知道使用绑定变量在性能方面会带来多大的差别，只需要运行一个非常小的测试来看看。在这个测试中，将在一个表中插入一些记录行。我使用如下所示的一个简单的表：

```
ops$tkyte@ORA9IR2> drop table t;

Table dropped.

ops$tkyte@ORA9IR2> create table t ( x int );
```

```
Table created.
```

下面再创建两个非常简单的存储过程。它们都向这个表中插入数字 1 到 10 000；不过，第一个过程使用了一条带绑定变量的 SQL 语句：

```
ops$tkyte@ORA9IR2> create or replace procedure proc1
  2          as
  3          begin
  4                  for i in 1 .. 10000
  5                  loop
  6                          execute immediate
  7                    'insert into t values ( :x )' using i;
  8                  end loop;
  9          end;
10 /

Procedure created.
```

第二个过程则分别为要插入的每一行构造一条独特的 SQL 语句：

```
ops$tkyte@ORA9IR2> create or replace procedure proc2
  2 as
  3 begin
  4          for i in 1 .. 10000
  5          loop
  6                  execute immediate
  7                'insert into t values ( '||i||')';
  8          end loop;
  9 end;
  10 /
```

```
Procedure created.
```

现在看来，二者之间惟一的差别，是一个过程使用了绑定变量，而另一个没有使用。它们都使用了动态 SQL（所谓动态 SQL 是指直到运行时才确定的 SQL），而且过程中的逻辑也是相同的。不同之处只在于是否使用了绑定变量。

下面用我开发的一个简单工具 runstats 对这两个方法详细地进行比较：

注意 关于安装 runstats 和其他工具的有关细节，请参见本书开头的“配置环境”一节。

```
ops$tkyte@ORA9IR2> exec runstats_pkg.rs_start

PL/SQL procedure successfully completed.



ops$tkyte@ORA9IR2> exec proc1

PL/SQL procedure successfully completed.



ops$tkyte@ORA9IR2> exec runstats_pkg.rs_middle

PL/SQL procedure successfully completed.



ops$tkyte@ORA9IR2> exec proc2

PL/SQL procedure successfully completed.



ops$tkyte@ORA9IR2> exec runstats_pkg.rs_stop(1000)

Run1 ran in 159 hsecs

Run2 ran in 516 hsecs

run 1 ran in 30.81% of the time
```

结果清楚地显示出，从墙上时钟来看，proc2（没有使用绑定变量）插入 10 000 行记录的时间比 proc1（使用了绑定变量）要多出很多。实际上，proc2 需要的时间是 proc1 的 3 倍多，这说明，在这种情况下，对于每个“无绑定变量”的 INSERT，执行语句所需时间中有 2/3 仅用于解析语句！

注意 如果愿意，也可以不用 runstats，而是在 SQL*Plus 中执行命令 SET TIMING ON，然后运行 proc1 和 proc2，这样也能进行比较。

不过，对于 proc2，还有更糟糕的呢！runstats 工具生成了一个报告，显示出这两种方法在闩利用率方面的差别，另外还提供了诸如解析次数之类的统计结果。这里我要求 runstats 打印出差距在 1 000 以上的比较结果（这正是 rs_stop 调用中 1000 的含义）。查看这个信息时，可以看到各方法使用的资源存在显著的差别：

```
Name                                  Run1        Run2        Diff

STAT...parse count (hard)                4      10,003       9,999

LATCH.library cache pin             80,222     110,221      29,999

LATCH.library cache pin alloca      40,161      80,153      39,992

LATCH.row cache enqueue latch           78      40,082      40,004

LATCH.row cache objects 98          40,102      40,004

LATCH.child cursor hash table 35    80,023      79,988

LATCH.shared pool                   50,455     162,577     112,122

LATCH.library cache                110,524     250,510     139,986

Run1 latches total versus runs -- difference and pct

Run1            Run2            Diff            Pct

407,973         889,287         481,314         45.88%

PL/SQL procedure successfully completed.
```

注意 你自己测试时可能会得到稍微不同的值。如果你得到的数值和上面的一样，特别是如果闩数都与我的测试结果完全相同，那倒是很奇怪。不过，假设你也像我一样，也是在 Linux 平台上使用 Oracle9i Release 2，应该能看到类似的结果。不论哪个版本，可以想见，硬解析处理每个插入所用的闩数总是要高于软解析（对于软解析，更确切的说法应该是，只解析一次插入，然后反复执行）。还在同一台机器上，但是如果使用 Oracle 10g Release 1 执行前面的测试，会得到以下结果：与未使用绑定变量的方法相比，绑定变量方法执行的耗用时间是前者的 1/10，而所用的闩总数是前者的 17%。这有两个原因，首先，10g 是一个新的版本，有一些内部算法有所调整；另一个原因是在 10g 中，PL/SQL 采用了一种改进的方法来处理动态 SQL。

可以看到，如果使用了绑定变量（后面称为绑定变量方法），则只有 4 次硬解析；没有使用绑定变量时（后面称为无绑定变量方法），却有不下 10 000 次的硬解析（每次插入都会带来一次硬解析）。还可以看到，无绑定变量方法所用的闩数是绑定变量方法的两倍之多。这是因为，要想修改这个共享结构，Oracle 必须当心，一次只能让一个进程处理（如果两个进程或线程试图同时更新同一个内存中的数据结构，将非常糟糕，可能会导致大量破坏）。因此，Oracle 采用了一种闩定（latching）机制来完成串行化访问，闩（latch） 是一种轻量

级锁定设备。不要被“轻量级”这个词蒙住了，作为一种串行化设备，闩一次只允许一个进程短期地访问数据结构。闩往往被硬解析实现滥用，而遗憾的 是，这正是闩最常见的用法之一。共享池的闩和库缓存的闩就是不折不扣的闩；它们成为人们频繁争抢的目标。这说明，想要同时硬解析语句的用户越多，性能问题 就会变得越来越严重。人们执行的解析越多，对共享池的闩竞争就越厉害，队列会排得越长，等待的时间也越久。

注意 如果机器的处理器不止一个，在 9i 和以上版本中，共享池还可以划分为多个子池，每个子池都由其自己的闩保护。这样即使应用没有使用绑定变量，也可以提高可扩缩性，但是这并没有从根本上克服闩定问题。

执行无绑定变量的 SQL 语句，很像是在每个方法调用前都要编译子例程。假设把 Java 源代码交付给客户，在调用类中的方法之前，客户必须调用 Java 编译器，编译这个类，再运行方法，然后丢掉字节码。下一次想要执行同样的方法时，他们还要把这个过程再来一遍：先编译，再运行，然后丢掉字节码。你肯定不希望在应用中这样做。数据库里也应该一样，绝对不要这样做。

对于这个特定的项目，可以把现有的代码改写为使用绑定变量，这是最好的做法。改写后的代码与原先比起来，速度上有呈数量级的增长，而且系统能支持的并发用户 数也增加了几倍。不过，在时间和精力投入方面却要付出很大的代价。并不是说使用绑定变量有多难，也不是说使用绑定变量容易出错，而只是因为开发人员最初没 有使用绑定变量的意识，所以必须回过头去，几乎把所有代码都检查和修改一遍。如果他们从第一天起就很清楚在应用中使用绑定变量至关重要，就不用费这么大的 功夫了。

1.3.2 理解并发控制

并发控制在不同的数据库中各不相同。正是因为并发控制，才使得数据库不同于文件系统，也使得不同的数据库彼此有所区别。你的数据库应用要在并发访问条件下正 常地工作，这一点很重要，但这也是人们时常疏于测试的一个方面。有些技术在一切都顺序执行的情况下可能工作得很好，但是如果任务要同时进行，这些技术的表 现可能就差强人意了。如果对你的特定数据库如何实现并发控制了解不够，就会遭遇以下结果：

- ☐ 破坏数据的完整性。
- ☐ 随着用户数的增多，应用的运行速度减慢。
- ☐ 不能很好地扩缩应用来支持大量用户。

注意我没有说“你可能……”或者“有……的风险”，而是直截了当地说：如果没有适当的并发控制，甚至如果未能充分了解并发控制，你肯定会遇到这些情况。如果没有正确的并发控制，就会破坏数据库的完整性，因为有些技术单独能工作，但是在多用户情况下就不能像你预期的那样正常工作了。你的应用会比预想的运行得 慢，因为它总在等待资源。另外因为存在锁定和竞争问题，你将不能很好地扩缩应用。随着访问资源的等待队列变得越来越长，等待的时间也会越来越久。

这里可以打个比方，考虑一下发生在收费站的阻塞情况。如果所有汽车都以顺序的、可预料的方式到来，一辆接着一辆，就不会出现阻塞。但是，如果多辆车同时到 达，就要排队。另外，等待时间并不是按照到达收费站的车辆数量线性增长。达到某个临界点后，一方面要花费大量额外的时间来“管理”排队等待的人，另一方面 还要为他们提供服务（在数据库中这称为上下文切换）。

并发问题最难跟踪，就像调试多线程程序一样。在调试工具控制的人工环境下，程序

可能工作得很好，但到实际中却可怕地崩溃了。例如，在竞争条件下，你会发现两个线程力图同时修改同一个数据结构。这种 bug 跟踪起来非常困难，也很难修正。如果你只是独立地测试你的应用，然后部署，并交给数十个并发用户使用，就很有可能痛苦地遭遇原先未能检测到的并发问题。

在下面两节中，我会通过两个小例子来谈谈对并发控制缺乏了解可能会破坏你的数据，或者会影响应用的性能和可扩缩性。

1. 实现锁定

数据库使用锁（lock） 来保证任何给定时刻最多只有一个事务在修改给定的一段数据。实质上讲，正是锁机制才使并发控制成为可能。例如，如果没有某种锁定模型来阻止对同一行的并发 更新，数据库就不可能提供多用户访问。不过，如果滥用或者使用不当，锁反倒会阻碍并发性。如果你或数据库本身不必要地对数据锁定，能并发地完成操作的人数 就会减少。因此，要理解什么是锁定，你的数据库中锁定是怎样工作的，这对于开发可扩缩的、正确的应用至关重要。

还 有一点很重要，你要知道每个数据库会以不同的方式实现锁定。有些数据库可能有页级锁，另外一些则有行级锁；有些实现会把行级锁升级为页级锁，另外一些则不 然；有些使用读锁，另外一些不使用；有些通过锁定实现串行化事务，另外一些则通过数据的读一致视图来实现（没有锁）。如果你不清楚这些微小的差别，它们就 会逐步膨胀为严重的性能问题，甚至演变成致命的 bug。

以下是对 Oracle 锁定策略的总结：

- Oracle 只在修改时才对数据加行级锁。正常情况下不会升级到块级锁或表级锁（不过两段提交期间的一段很短的时间内除外，这是一个不常见的操作）。
- 如果只是读数据，Oracle 绝不会对数据锁定。不会因为简单的读操作在数据行上锁定。
- 写入 器（writer）不会阻塞读取器（reader）。换种说法：读（read）不会被写（write）阻塞。这一点几乎与其他所有数据库都不一样。在其他数据库中，读往往会被写阻塞。尽管听上去这个特性似乎很不错（一般情况下确实如此），但是，如果你没有充分理解这个思想，而且想通过应用逻辑对应用施加完整性约束，就极有可能做得不对。第 7 章介绍并发控制时还会更详细地讨论这个内容。
- 写入器想写某行数据，但另一个写入器已经锁定了这行数据，此时该写入器才会被阻塞。读取器绝对不会阻塞写入器。

开发应用时必须考虑到这些因素，而且还要认识到这个策略是 Oracle 所独有的，每个数据库实现锁定的方法都存在细微的差别。即使你在应用中使用最通用的 SQL，由于各数据库开发商采用的锁定和并发控制模型不同，你的应用也可能有不同的表现。倘若开发人员不了解自己的数据库如何处理并发性，肯定会遇到数据完整性问题。（开发人员从另外某种数据库转向 Oracle，或者从 Oracle 转向其他数据库时，如果没有考虑在应用中采用不同的并发机制，这种情况就尤为常见。）

2. 防止丢失更新

Oracle 的无阻塞方法有一个副作用，如果确实想保证一次最多只有一个用户访问一行数据，开发人员就得自己做些工作。

考 虑下面这个例子。一位开发人员向我展示了他刚开发的一个资源调度程序（可以用来调度会议室、投影仪等资源），这个程序正在部署当中。应用中实现了这样一个 业务规则：在给定的任何时间段都不能将一种资源分配给多个人。也就是说，应用中包含了实现这个业务规则的代码，它会明确地检查此前这个时间片没有分配给其 他用户（至少，这个开发人员认为是这样）。这段代码先查询 SCHEDULES 表，如果不存在与该时间片重叠的记录行（该时间片尚未分配），则插入新行。所以，开发人员主要考虑两个表：

```
create table resources ( resource_name varchar2(25) primary key, ... );

create table schedules

( resource_name references resources,

   start_time date not null,

   end_time date not null,

   check (start_time < end_time ),

   primary key(resource_name,start_time)

);
```

在分配资源（如预订房间）之前，应用将查询：

```
select count(*)

        from schedules

where resource_name = :room_name

        and (start_time <= :new_end_time)

        and (end_time >= :new_start_time)
```

看上去很简单，也很安全（在开发人员看来）：如果得到的计数为 0，这个房间就是你的了。如果返回的数非 0，那在此期间你就不能预订这个房间。了解他的逻辑后，我建立了一个非常简单的测试，来展示这个应用运行时可能出现的一个错误，这个错误极难跟踪，而且事后也很难诊断。有人甚至以为这必定是一个数据库 bug。

我所做的其实很简单，就是让另外一个人使用这个开发人员旁边的一台机器，两个人都浏览同一个屏幕，然后一起数到 3 时，两人都单击 Go 按钮，尽量同时预订同一个房间，一个人想预订下午 3:00 到下午 4:00 这个时段，另一个人要预订下午 3:30 到下午 4:00 这个时段。结果两个人都预订成功。这个逻辑独立执行时原本能很好地工作，但到多用户环境中就不行了。为什么会出现这个问题？部分原因就在于 Oracle 的非阻塞读。这两个会话都不会阻塞对方，它们只是运行查询，然后完成调度房间的逻辑。两个会话都通过运行查询来查找是否已经有预订，尽管另一个会话可能已经开始修改 SCHEDULES 表，但查询看不到这些修改（所做的修改在提交之前对其他会话来说是不可见的，而等到提交时已为时过晚）。由于这两个会话并没有试图修改 SCHEDULES 表中的同一行，所以它们不会相互阻塞。由

此说来，这个应用不能像预期的那样保证前面提到的业务规则。

开发人员需要一种方法使得这个业务规则在多用户环境下也能得到保证，也就是要确保一次只有一个人预订一种给定的资源。在这种情况下，解决方案就是加入他自己的一些串行化机制。他的做法是，在对 SCHEDULES 表进行修改之前，先对 RESOURCES 表中的父行锁定。这样一来，SCHEDULES 表中针对给定 RESOURCE_NAME 值的所有修改都必须依次按顺序进行，一次只能执行一个修改。也就是说，要预订资源 X 一段时间，就要锁定 RESOURCES 表中对应 X 的那一行，然后修改 SCHEDULES 表。所以，除了前面的 count(*) 外，开发人员首先需要完成以下查询：

```
select * from resources where resource_name = :room_name FOR UPDATE;
```

这里，他在调度资源之前先锁定了资源（这里指房间），换句话说，就是在 SCHEDULES 表中查询该资源的预订情况之前先锁定资源。通过锁定所要调度的资源，开发人员可以确保别人不会同时修改对这个资源的调度。其他人都必须等待，直到他提交了事务为止，此时就能看到他所做的调度。这样就杜绝了调度重叠的可能性。

开发人员必须了解到，在多用户环境中，他们必须不时地采用多线程编程中使用的一些技术。在这里，FOR UPDATE 子句的作用就像是一个信号量（semaphore），只允许串行访问 RESOURCES 表中特定的行，这样就能确保不会出现两个人同时调度的情况。我建议把这个逻辑实现为一个事务 API，也就是说，把所有逻辑都打包进一个存储过程中，只允许应用通过这个 API 修改数据。代码如下：

```
create or replace procedure schedule_resource

( p_resource_name in varchar2,

p_start_time in date,

p_end_time in date

)

as

        l_resource_name resources.resource_name%type;

        l_cnt number;

begin
```

首先在 RESOURCES 表中锁定我们想调度的那个资源的相应行。如果别人已经锁定了这一行，我们就会阻塞并等待 ：

```
        select resource_name into l_resource_name

        from resources

        where resource_name = p_resource_name
```

```
        FOR UPDATE;
```

既然我们已经有了锁，那么只有我们能在这个 SCHEDULES 表中插入对应此资源名的调度，所以如下查看这个表是安全的：

```
        select count(*)
                into l_cnt
        from schedules
        where resource_name = p_resource_name
                and (start_time <= p_end_time)
                and (end_time >= p_start_time);
        if ( l_cnt <> 0 )
        then
                raise_application_error
                        (-20001, 'Room is already booked!' );
        end if;
```

如果能运行到这里而没有发生错误，就可以安全地在 SCHEDULES 表中插入预订资源的相应记录行，而不用担心出现重叠：

```
        insert into schedules
                ( resource_name, start_time, end_time )
        values
                ( p_resource_name, p_start_time, p_end_time );
end schedule_resources;
```

这个解决方案仍是高度并发的，因为可能有数以千计要预订的资源。这里的做法是，确保任何时刻只能有一个人修改资源。这是一种很少见的情况，在此要对并不会真 正更新的数据手动锁定。我们要知道哪些情况下需要这样做，还要知道哪些情况下不需要这样做（稍后会给出这样一个例子），这同样很重要。另外，如果别人只是 读取数据，就不会锁定资源不让他们读（但在其他数据库中可能不是这样），所以这种解决方案可以很好地扩缩。

如果你想把应用从一个数据库移植到另一个数据库，这一节讨论的问题就有很大的影响（本章后面还会再谈这个内容），而且可能会一再地把人“绊倒”。例如，如果 你有使用另外某些数据库的经验，其中写入器会阻塞读取器，而且读取器也会阻塞写入器，你可能就

会有成见，过分依赖这一点来避免数据完整性问题。一种解决办 法是干脆不要并发，在许多非 Oracle 数据库中就是这样做的。但在 Oracle 中则是并发规则至上，因此，你必须知道可能会发生不同的情况（或者遭受不同的后果）。

注意 第 7 章还会再次谈到这个例子。以上代码有一个前提，即假设事务隔离级别是 READ COMMITTED。如果事务隔离级别是 SERIALIZABLE，这个逻辑将无法正常工作。倘若现在就详细介绍这两种模式的区别，这一章就会变得过于复杂，所以这个内容以后再讨论。

99%的情况下，锁定是完全透明的，无需你来操心。但还有另外的 1%，你必须清楚哪些情况下需要自己考虑锁定。对于这个问题，并没有非黑即白的直接结论，无法简单地罗列出“如果你要这样做，就应该这样做”之类的条条框框。关键是要了解应用在多用户环境中有何表现，另外在你的数据库中表现如何。

第 7 章会更深入地讨论这个内容，你会进一步了解本节介绍的这种完整性约束，有些情况下，一个表中的多行必须遵循某个规则，或者两个表或多个表之间必须保证某个规 则（如引用完整性约束），一定要特别注意这些情况，而且这些情况也最有可能需要采用手动锁定或者另外某种技术来确保多用户环境下的完整性。

1.3.3 多版本

这个主题与并发控制的关系非常紧密，因为这正是 Oracle 并发控制机制的基础，Oracle 采用了一种多版本、读一致（read-consistent）的并发模型。再次说明，我们将在第 7 章更详细地介绍有关的技术。不过，实质上讲，Oracle 利用这种机制提供了以下特性：

- 读一致查询：对于一个时间点（point in time），查询会产生一致的结果。
- 非阻塞查询：查询不会被写入器阻塞，但在其他数据库中可能不是这样。

Oracle 数据库中有两个非常重要的概念。多版本（multi-versioning）一词实质上指 Oracle 能够从数据库同时物化多个版本的数据。如果你理解了多版本如何工作，就会知道能从数据库得到什么。在进一步深入讨论 Oracle 如何实现多版本之前，下面用我认为最简单的一个方法来演示 Oracle 中的多版本：

```
ops$tkyte@ORA10G> create table t
  2 as
  3 select *
  4 from all_users;
Table created.


ops$tkyte@ORA10G> variable x refcursor
ops$tkyte@ORA10G> begin
```

```
  2 open :x for select * from t;

  3 end;

  4 /

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> delete from t;

28 rows deleted.



ops$tkyte@ORA10G> commit;

Commit complete.



ops$tkyte@ORA10G> print x

USERNAME                       USER_ID       CREATED

------------------------------ ----------    ---------

BIG_TABLE                      411           14-NOV-04

OPS$TKYTE                      410           14-NOV-04

DIY                            69            26-SEP-04

...

OUTLN                          11            21-JAN-04

SYSTEM                         5             21-JAN-04

SYS                            0             21-JAN-04

28 rows selected.
```

在前面的例子中，我创建了一个测试表 T，并把 ALL_USERS 表的一些数据加载到这个表中。然后在这个表上打开一个游标。在此没有从该游标获取数据，只是打开游标而已。

注意　要记住，Oracle 并不“回答”这个查询。打开游标时，Oracle 不复制任何数据，你

可以想想看，即使一个表有十亿条记录，是不是也能很快就打开游标？没错，游标会立即打开，它会边行进边回答查询。换句话说，只是在你获取数据时它才从表中读数据。

在同一个会话中（或者也可以在另一个会话中；这同样能很好地工作），再从该表删除所有数据。甚至用 COMMIT 提交了删除所做的工作。记录行都没有了，但是真的没有了吗？实际上，还是可以通过游标获取到数据。OPEN 命令返回的结果集在打开的那一刻（时间点）就已经确定。打开时，我们根本没有碰过表中的任何数据块，但答案已经是铁板钉钉的了。获取数据之前，我们无法知道答案会是什么；不过，从游标角度看，结果则是固定不变的。打开游标时，并非 Oracle 将所有数据复制到另外某个位置；实际上是 DELETE 命令为我们把数据保留下来，把它放在一个称为 undo 段（undo segment）的数据区，这个数据区也称为回滚段（rollback segment）。

读一致性（read-consistency）和多版本就是这么回事。如果你不了解 Oracle 的多版本机制是怎样工作的，不清楚这意味着什么，你就不可能充分利用 Oracle，也不可能在 Oracle 上开发出正确的应用（也就是说，能确保数据完整性的应用）。

1. 多版本和闪回

过去，Oracle 总是基于查询的某个时间点来做决定（从这个时间点开始查询是一致的）。也就是说，Oracle 会保证打开的结果集肯定是以下两个时间点之一的当前结果集：

- 游标打开时的时间点。这是 READ COMMITTED 隔离模式的默认行为，该模式是默认的事务模式（第 7 章将介绍 READ COMMITTED、READ ONLY 和 SERIALIZABLE 事务级别之间的差别）。
- 查询所属事务开始的时间点。这是 READ ONLY 和 SERIALIZABLE 隔离级别中的默认行为。

不过，从 Oracle9i 开始，情况要灵活得多。实际上，我们可以指示 Oracle 提供任何指定时间的查询结果（对于回放的时间长度有一些合理的限制；当然，这要由你的 DBA 来控制），这里使用了一种称为闪回查询（flashback query）的特性。

请考虑以下例子。首先得到一个 SCN，这是指系统修改号（System Change Number）或系统提交号（System Commit Number）；这两个术语可互换使用。SCN 是 Oracle 的内部时钟：每次发生提交时，这个时钟就会向上滴答（递增）。实际上也可以使用日期或时间戳，不过这里 SCN 很容易得到，而且相当准确：

```
scot@ORA10G> variable SCN number

scott@ORA10G> exec :scn := dbms_flashback.get_system_change_number

PL/SQL procedure successfully completed.

scott@ORA10G> print scn

SCN

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

```
33295399
```

现在可以让 Oracle 提供 SCN 值所表示时间点的数据。以后再查询 Oracle 时，就能看看这一时刻表中的内容。首先来看 EMP 表中现在有什么：

```
scott@ORA10G> select count(*) from emp;

COUNT(*)

----------

14
```

下面把这些信息都删除，并验证数据是否确实“没有了”：

```
scott@ORA10G> delete from emp;

14 rows deleted.

scott@ORA10G> select count(*) from emp;

COUNT(*)

----------

0
```

此外，使用闪回查询（即 AS OF SCN 或 AS OF TIMESTAMP 子句），可以让 Oracle 告诉我们 SCN 值为 33295399 的时间点上表中有什么：

```
scott@ORA10G> select count(*) from emp AS OF SCN :scn;

COUNT(*)

----------

14
```

不仅如此，这个功能还能跨事务边界。我们甚至可以在同一个查询中得到同一个对象在“两个时间点”上的结果！因此可以做些有意思的事情：

```
scott@ORA10G> commit;

Commit complete.



scott@ORA10G> select *

  2 from (select count(*) from emp),
```

```
  3 (select count(*) from emp as of scn :scn)

  4 /

COUNT(*)         COUNT(*)

----------       ----------

0                14
```

如果你使用的是 Oracle 10g 及以上版本，就有一个“闪回”（flashback）命令，它使用了这种底层多版本技术，可以把对象返回到以前某个时间点的状态。在这个例子中，可以将 EMP 表放回到删除信息前的那个时间点：

```
scott@ORA10G> flashback table emp to scn :scn;

Flashback complete.



scott@ORA10G> select *

  2 from (select count(*) from emp),

  3 (select count(*) from emp as of scn :scn)

  4 /

COUNT(*) COUNT(*)

----------       ----------

14               14
```

注意 如果你得到一个错误“ORA-08189: cannot flashback the table because row movement is not enabled using the FLASHBACK command”（ORA-08189：无法闪回表，因为不支持使用 FLASHBACK 命令完成行移动），就必须先执行一个命令：ALTER TABLE EMP ENABLE ROW MOVEMENT。这个命令的作用是，允许 Oracle 修改分配给行的 rowid。在 Oracle 中，插入一行时就会为它分配一个 rowid，而且这一行永远拥有这个 rowid。闪回表处理会对 EMP 完成 DELETE，并且重新插入行，这样就会为这些行分配一个新的 rowid。要支持闪回就必须允许 Oracle 执行这个操作。

2. 读一致性和非阻塞读

下面来看多版本、读一致查询以及非阻塞读的含义。如果你不熟悉多版本，下面的代码看起来可能有些奇怪。为简单起见，这里假设我们读取的表在每个数据库块（数据库中最小的存储单元）中只存放一行，而且这个例子要全面扫描这个表。

我们查询的表是一个简单的 ACCOUNTS 表。其中包含了一家银行的账户余额。其结

构很简单：

```
create table accounts

( account_number number primary key,

 account_balance number

);
```

在实际中，ACCOUNTS 表中可能有上百万行记录，但是为了力求简单，这里只考虑一个仅有 4 行的表（第 7 章还会更详细地分析这个例子），如表 1-1 所示。

表 1-1 ACCOUNTS 表的内容

行	账 号	账户余额
1	123	$500.00
2	234	$250.00
3	345	$400.00
4	456	$100.00

我们可能想运行一个日报表，了解银行里有多少钱。这是一个非常简单的查询：

```
select sum(account_balance) from accounts;
```

当然，这个例子的答案很明显：$1 250.00。不过，如果我们现在读了第 1 行，准备读第 2 行和第 3 行时，一台自动柜员机（ATM）针对这个表发生了一个事务，将$400.00 从账户 123 转到了账户 456，又会怎么样呢？查询会计算出第 4 行的余额为$500.00，最后就得到了$1 650.00，是这样吗？当然，应该避免这种情况，因为这是不对的，任何时刻账户余额列中的实际总额都不是这个数。读一致性就是 Oracle 为避免发生这种情况所采用的办法，你要了解，与几乎所有的其他数据库相比，Oracle 采用的方法有什么不同。

在几乎所有的其他数据库中，如果想得到“一致”和“正确”的查询答案，就必须在计算总额时锁定整个表，或者在读取记录行时对其锁定。这样一来，获取结果时就 可以防止别人再做修改。如果提前锁定表，就会得到查询开始时数据库中的结果。如果在读取数据时锁定（这通常称为共享读锁（shared read lock），可以防止更新，但不妨碍读取器访问数据），就会得到查询结束时数据库中的结果。这两种方法都会大大影响并发性。由于存在表锁，查询期间会阻止对整个表进行更新（对于一个仅有 4 行的表，这可能只是很短的一段时间，但是对于有上百万行记录的表，可能就是几分钟之多）。“边读边锁定”的办法也有问题，不允许对已经读取和已经处理过的数据再做更新，实际上这会导致查询与其他更新之间产生死锁。

我曾经说过，如果你没有理解多版本的概念，就无法充分利用 Oracle。下面告诉你一个原因。Oracle 会利用多版本来得到结果，也就是查询开始时那个时间点的结果，然后完成查询，而不做任何锁定（转账事务更新第 1 行和第 4 行时，这些行会对其他写入器锁定，但不会对读取器锁定，如这里的 SELECT SUM...查询）。实际上，Oracle 根本没有“共享读”

锁（这是其他数据库中一种常用的锁），因为这里不需要。对于可能妨碍并发性的一切因素，只要能去掉的，Oracle 都已经去掉了。

我见过这样一些实际案例，开发人员没有很好地理解 Oracle 的多版本功能，他编写的查询报告将整个系统紧紧地锁起来。之所以会这样，主要是因为开发人员想从查询得到读一致的（即正确的）结果。这个开发人员以前用过其他一些数据库，在这些数据库中，要做到这一点都需要对表锁定，或者使用一个 SELECT ... WITH HOLDLOCK（这是 SQL Server 中的一种锁定机制，可以边读取边以共享模式对行锁定）。所以开发人员想在运行报告前先对表锁定，或者使用 SELECT ... FOR UPDATE（这是 Oracle 中与 holdlock 最接近的命令）。这就导致系统实质上会停止处理事务，而这完全没有必要。

那么，如果 Oracle 读取时不对任何数据锁定，那它又怎么能得到正确、一致的答案（$1 250.00）呢？换句话说，如何保证得到正确的答案同时又不降低并发性？秘密就在于 Oracle 使用的事务机制。只要你修改数据，Oracle 就会创建撤销（undo）条目。这些 undo 条目写至 undo 段（撤销段，undo segment）。如果事务失败，需要撤销，Oracle 就会从这个回滚段读取“之前”的映像，并恢复数据。除了使用回滚段数据撤销事务外，Oracle 还会用它撤销读取块时对块所做的修改，使之恢复到查询开始前的时间点。这样就能摆脱锁来得到一致、正确的答案，而无需你自己对任何数据锁定。

所以，对我们这个例子来说，Oracle 得到的答案如表 1-2 所示。

表 1-2　实际的多版本例子

时　间	查　询	转账事务
T1	读第 1 行；到目前为止 sum = $500	
T2		更新第 1 行；对第 1 行加一个排他锁（也称独占锁，exclusive lock)，阻止其他更新第 1 行。现在有$100
T3	读第 2 行；到目前为止 sum = $750	
T4	读第 3 行；到目前为止 sum = $1 150	
T5		更新第 4 行；对第 4 行加一个排他锁，阻止其他更新（但不阻止读操作）。第 4 行现在有$500
T6	读第 4 行，发现第 4 行已修改。这会将块回滚到 T1 时刻的状态。查询从这个块读到值$100	
T7	得到答案$1 250	

在 T6 时，Oracle 有效地“摆脱”了事务加在第 4 行上的锁。非阻塞读是这样实现的：

Oracle 只看数据是否改变，它并不关心数据当前是否锁定（锁定意味着数据已经改变）。Oracle 只是从回滚段中取回原来的值，并继续处理下一个数据块。

下一个例子也能很好地展示多版本。在数据库中，可以得到同一个信息处于不同时间点的多个版本。Oracle 能充分使用不同时间点的数据快照来提供读一致查询和非阻塞查询。

数据的读一致视图总是在 SQL 语句级执行。SQL 语句的结果对于查询开始的时间点来说是一致的。正是因为这一点，所以下面的语句可以插入可预知的数据集：

```
Begin
    or x in (select * from t)
    loop
        insert into t values (x.username, x.user_id, x.created);
    end loop;
end;
```

SELECT * FROM T 的结果在查询开始执行时就已经确定了。这个 SELECT 并不看 INSERT 生成的任何新数据。倘若真的能看到新插入的数据，这条语句就会陷入一个无限循环。如果 INSERT 在 T 中生成了更多的记录行，而 SELECT 也随之能“看到”这些新插入的行，前面的代码就会建立数目未知的记录行。如果表 T 刚开始有 10 行，等结束时 T 中可能就会有 20、21、23 或无限行记录。这完全不可预测。Oracle 为所有语句都提供了这种读一致性，所以如下的 INSERT 也是可预知的：

```
insert into t select * from t;
```

这个 INSERT 语句得到了 T 的一个读一致视图。它看不到自己刚刚插入的行，而只是插入 INSERT 操作刚开始时表中已有的记录行。许多数据库甚至不允许前面的这种递归语句，因为它们不知道到底可能插入多少行。

所以，如果你用惯了其他数据库，只熟悉这些数据库中处理查询一致性和并发性的方法，或者你根本没有接触过这些概念（也就是说，你根本没有使用数据库的经验），现在应该知道，理解 Oracle 的做法对你来说有何等重要的意义。要想最大限度地发挥 Oracle 的潜能，以及为了实现正确的代码，你必须了解 Oracle 中的这些问题是怎么解决的（而不是其他数据库中是如何实现的）。

1.3.4 数据库独立性?

至此，你可能想到这一节要讲什么了。我提到了其他的数据库，也谈到各个数据库中会以不同的方式实现特性。除了一些只读应用外，我的观点是：要构建一个完全数据库独立的应用，而且是高度可扩缩的应用，是极其困难的。实际上，这几乎不可能，除非你真正了解每个数据库具体如何工作。另外，如果你清楚每个数据库工作 的具体细节，就会知道，数据库独立性可能并不是你真正想要的（这个说法有点绕！）。

例如，再来看最早提到的资源调度例子（增加 FOR UPDATE 子句之前）。假设在另一个数据库上开发这个应用，这个数据库有着与 Oracle 完全不同的锁定/并发模型。我想说的

是，如果把应用从一个数据库移植到另一个数据库，就必须验证它在完全不同的环境下还能正常地工作，而且为此我们要做大幅修改！

假设把这个资源调度应用部署在这样一个数据库上，它采用了阻塞读机制（读会被写阻塞）。现在业务规则通过一个数据库触发器实现（在 INSERT 之后，但在事务提交之前，我们要验证表中对应特定时间片的记录只有一行，也就是刚插入的记录）。在阻塞读系统中，由于有这种新插入的数据，所以表的插入要串行完成。第一个人插入他（她）的请求，要在星期五的下午 2:00 到下午 3:00 预订“房间 A”， 然后运行一个查询查看有没有重叠的预订。下一个人想插入一个重叠的请求，查找重叠情况时，这个请求会被阻塞（它发现有新插入的数据，但要等待直到这些数据 确实可以读取）。在这个采用阻塞读机制的数据库中，我们的应用显然可以正常工作（不过如果两个人都插入自己的行，然后试图读对方的数据，就有可能得到一个 死锁，这个概念将在第 6 章讨论），但不能并发工作，因为我们是一个接一个地检查是否存在重叠的资源分配。

如果把这个应用移植到 Oracle，并简单地认为它也能同样地工作，结果可能让人震惊。由于 Oracle 会在行级锁定，并提供了非阻塞读，所以看上去一切都乱七八糟。如前所示，必须使用 FOR UPDATE 子句来完成串行访问。如果没有这个子句，两个用户就可能同时调度同一个资源。如果不了解所用数据库在多用户环境中如何工作，就会导致这样的直接后果。

将应用从数据库 A 移植到数据库 B 时，我时常遇到这种问题：应用在数据库 A 上原本无懈可击，到了数据库 B 上却不能工作，或者表现得很离奇。看到这种情况，我们的第一个想法往往是，数据库 B 是一个“不好的”数据库。而真正的原因其实是数据库 B 的工作方式完全不同。没有哪个数据库是错的或“不好的”，它们只是有所不同而已。应当了解并理解它们如何工作，这对于处理这些问题有很大的帮助。将应用从 Oracle 移植到 SQL Server 时，也会暴露 SQL Server 的阻塞读和死锁问题，换句话说，不论从哪个方向移植都可能存在问题。

例如，有人请我帮忙将一些 Transact-SQL（T-SQL，SQL Server 的存储过程语言）转换为 PL/SQL。做这个转换的开发人员一直在抱怨 Oracle 中 SQL 查询返回的结果是“错的”。查询如下所示：

```
declare
        l_some_variable varchar2(25);
begin
        if ( some_condition )
        then
                l_some_variable := f( ... );
        end if;
        for C in ( select * from T where x = l_some_variable )
```

```
        loop

                ...
```

这个查询的目标是：在 T 表中，如果不满足某个条件，则找出 x 为 NULL 的所有行；如果满足某个条件，就找出 x 等于某个特定值的所有行。

开发人员抱怨说，在 Oracle 中，如果 L_SOME_VARIABLE 未设置为一个特定的值（仍为 NULL），这个查询居然不返回任何数据。但是在 Sybase 或 SQL Server 中不是这样的，查询会找到将 x 设置为 NULL 值的所有行。从 Sybase 或 SQL Server 到 Oracle 的转换中，几乎都能发现这个问题。SQL 采用一种三值逻辑来操作，Oracle 则是按 ANSI SQL 的要求来实现 NULL 值。基于这些规则的要求，x 与 NULL 的比较结果既不为 true 也不为 false，也就是说，实际上，它是未知的（unknown）。从以下代码可以看出我的意思：

```
ops$tkyte@ORA10G> select * from dual where null=null;

no rows selected



ops$tkyte@ORA10G> select * from dual where null <> null;

no rows selected



ops$tkyte@ORA10G> select * from dual where null is null;

D

-

X
```

第一次看到这些结果可能会被搞糊涂。这说明，在 Oracle 中，NULL 与 NULL 既不相等，也不完全不相等。默认情况下，SQL Server 则不是这样处理；在 SQL Server 和 Sybase 中，NULL 就等于 NULL。不能说 Oracle 的 SQL 处理是错的，也不能说 Sybase 或 SQL Server 的处理不对，它们只是方式不同罢了。实际上，所有这些数据库都符合 ANSI，但是它们的具体做法还是有差异。有许多二义性、向后兼容性等问题需要解决。例如， SQL Server 也支持 ANSI 方法的 NULL 比较，但这不是默认的方式（如果改成 ANSI 方法的 NULL 比较，基于 SQL Server 构建的数千个遗留应用就会出问题）。

在这种情况下，一种解决方案是编写以下查询：

```
select *

from t

where ( x = l_some_variable OR (x is null and l_some_variable is NULL ))
```

不过，这又会带来另一个问题。在 SQL Server 中，这个查询会使用 x 上的索引。Oracle 中却不会这样，因为 B*树索引不会对一个完全为 NULL 的项加索引（索引技术将在第 12 章介绍）。因此，如果需要查找 NULL 值，B*树索引就没有什么用处。

这里，为了尽量减少对代码的影响，我们的做法是赋给 x 某个值，不过这个值并没有实际意义。在此，根据定义可知，x 的正常值是正数，所以可以选择 －1。这样一来，查询就变成：

```
select * from t where nvl(x,-1) = nvl(l_some_variable,-1)
```

由此创建一个基于函数的索引：

```
create index t_idx on t( nvl(x,-1) );
```

只需做最少的修改，就能在 Oracle 中得到与 SQL Server 同样的结果。从这个例子可以总结出以下几个要点：

- 数据库是不同的。在一个数据库上取得的经验也许可以部分应用于另一个数据库，但是你必须有心理准备，二者之间可能存在一些基本差别，可能还有一些细微的差别。
- 细微的差别（如对 NULL 的处理）与基本差别（如并发控制机制）可能有同样显著的影响。
- 应当了解数据库，知道它是如何工作的，它的特性如何实现，这是解决这些问题的惟一途径。

常有开发人员问我如何在数据库中做某件特定的事情（通常这样的问题一天不止一个），例如“如何在一个存储过程中创建临时表？”对于这些问题，我并不直接回答，而是反过来问他们“你为什么想那么做？”给我的回答常常是：“我们在 SQL Server 中就是用存储过程创建临时表，所以在 Oracle 中也要这么做。”这不出我所料，所以我的回答很简单：“你根本不是想在 Oracle 中用存储过程创建临时表，你只是以为自己想那么做。”实际上，在 Oracle 中这样做是很不好的。在 Oracle 中，如果在存储过程中创建表，你会发现存在以下问题：

- DDL 操作会阻碍可扩缩性。
- DDL 操作的速度往往不快。
- DDL 操作会提交事务。
- 必须在所有存储过程中使用动态 SQL 而不是静态 SQL 来访问这个表。
- PL/SQL 的动态 SQL 没有静态 SQL 速度快，或者说没有静态 SQL 优化。

关键是，即使真的需要在 Oracle 中创建临时表，你也不愿意像在 SQL Server 中那样在过程中创建临时表。你希望在 Oracle 中能以最佳方式工作。反过来也一样，在 Oracle 中，你会为所有用户创建一个表来共享临时数据；但是从 Oracle 移植到 SQL Server 时，可能不希望这样做，这会影响 SQL Server 的可扩缩性和并发性。所有数据库创建得都不一样，它们存在很大的差异。

1. 标准的影响

如果所有数据库都符合 SQL99，那它们肯定一样。至少我们经常做这个假设。在这一节中，我将揭开它的神秘面纱。

SQL99 是数据库的一个 ANSI/ISO 标准。这个标准的前身是 SQL92 ANSI/ISO 标准，而 SQL92 之前还有一个 SQL89 ANSI/ISO 标准。它定义了一种语言（SQL）以及数据库的行为（事务、隔离级别等）。你知道许多商业数据库至少在某种程度上是符合 SQL99 的吗？不过，这对于查询和应用的可移植性没有多大的意义，这一点你也清楚吗？

SQL92 标准有 4 个层次：

- 入门级（Entry level）。这是大多数开发商符合的级别。这一级只是对前一个标准 SQL89 稍做修改。所有数据库开发商都不会有更高的级别，实际上，美国国家标准和技术协会 NIST（National Institute of Standards and Technology，这是一家专门检验 SQL 合规性的机构）除了验证入门级外，甚至不做其他的验证。Oracle 7.0 于 1993 年通过了 NIST 的 SQL92 入门级合规性验证，那时我也是小组中的一个成员。如果一个数据库符合入门级，它的特性集则是 Oracle 7.0 的一个功能子集。
- 过渡级。这一级在特性集方面大致介于入门级和中间级之间。
- 中间级。这一级增加了许多特性，包括（以下所列并不完整）：
  - 动态 SQL
  - 级联 DELETE 以保证引用完整性
  - DATE 和 TIME 数据类型
  - 域
  - 变长字符串
  - CASE 表达式
  - 数据类型之间的 CAST 函数
- 完备级。增加了以下特性（同样，这个列表也不完整）：
  - 连接管理
  - BIT 串数据类型
  - 可延迟的完整性约束
  - FROM 子句中的导出表
  - CHECK 子句中的子查询
  - 临时表

入门级标准不包括诸如外联结（outer join）、新的内联结（inner join）语法等特性。过渡级则指定了外联结语法和内联结语法。中间级增加了更多的特性，当然，完备级就是 SQL92

全部。有关 SQL92 的大多数书都没有区别这些级别，这就会带来混淆。这些书只是说明了一个完整实现 SQL92 的理论数据库会是什么样子。所以无论你拿起哪一本书，都无法将书中所学直接应用到任何 SQL92 数据库上。关键是，SQL92 最多只达到入门级，如果你使用了中间级或更高级里的特性，就存在无法“移植”应用的风险。

SQL99 只定义了两级一致性：核心（core）一致性和增强（enhanced）一致性。SQL99 力图远远超越传统的“SQL”，并引入了一些对象—关系构造（数组、集合等）。它包括 SQL MM（多媒体，multimedia）类型、对象—关系类型等。还没有哪个开发商的数据库经认证符合 SQL99 核心级或增强级，实际上，据我所知，甚至没有哪个开发商声称他们的产品完全达到了某级一致性。

对于不同的数据库来说，SQL 语法可能存在差异，实现有所不同，同一个查询在不同数据库中的性能也不一样，不仅如此，还存在并发控制、隔离级别、查询一致性等问题。我们将在第 7 章详细讨论这些问题，并介绍不同数据库的差异对你会有什么影响。

SQL92/SQL99 试 图对事务应如何工作以及隔离级别如何实现给出一个明确的定义，但最终，不同的数据库还是有不同的结果。这都是具体实现所致。在一个数据库中，某个应用可能 会死锁并完全阻塞。但在另一个数据库中，同样是这个应用，这些问题却有可能不会发生，应用能平稳地运行。在一个数据库中，你可能利用了阻塞（物理串行 化），但在另一个数据库上部署时，由于这个数据库不会阻塞，你就会得到错误的答案。要将一个应用部署在另一个数据库上，需要花费大量的精力，付出艰辛的劳 动，即使你 100%地遵循标准也不例外。

关 键是，不要害怕使用开发商特有的特性，毕竟，你为这些特性花了钱。每个数据库都有自己的一套“技巧”，在每个数据库中总能找到一种完成操作的好办法。要使 用最适合当前数据库的做法，移植到其他数据库时再重新实现。要使用合适的编程技术，从而与这些修改隔离，我把这称为防御式编程（defensive programming）。

2. 防御式编程

我推崇采用防御式编程技术来构建真正可移植的数据库应用，实际上，编写操作系统可移植的应用时也采用了这种技术。防御式编程的目标是充分利用可用的工具，但是确保能够根据具体情况逐一修改实现。

可以对照来看，Oracle 是一个可移植的应用。它能在许多操作系统上运行。不过，在 Windows 上，它就以 Windows 方式运行，使用线程和其他 Windows 特有的工具。在 UNIX 上，Oracle 则作为一个多进程服务器运行，使用进程来完成 Windows 上线程完成的工作，也就是采用 UNIX 的方式运行。两个平台都提供了“核心 Oracle”功能，但是在底层却以完全不同的方式来实现。如果你的数据库应用要在多个数据库上运行，道理也是一样的。

例如，许多数据库应用都有一个功能，即为每一行生成一个惟一的键。插入行时，系统应自动生成一个键。为此，Oracle 实现了一个名为 SEQUENCE 的数据库对象。Informix 有一个 SERIAL 数据类型。Sybase 和 SQL Server 有一个 IDENTITY 类型。每个数据库都有一个解决办法。不过，不论从做法上讲，还是从输出来看，各个数据库的方法都有所不同。所以，有见识的开发人员有两条路可走：

- 开发一个完全独立于数据库的方法来生成惟一的键。
- 在各个数据库中实现键时，提供不同的实现，并使用不同的技术。

从理论上讲，第一种方法的好处是从一个数据库转向另一个数据库时无需执行任何修

改。我把它称为“理论上” 的好处，这是因为这种实现实在太庞大了，所以这种方案根本不可行。要开发一个完全独立于数据库的进程，你必须创建如下所示的一个表：

```
ops$tkyte@ORA10G> create table id_table

  2 ( id_name varchar2(30) primary key,

  3 id_value number );

Table created.



ops$tkyte@ORA10G> insert into id_table values ( 'MY_KEY', 0 );

1 row created.



ops$tkyte@ORA10G> commit;

Commit complete.
```

然后，为了得到一个新的键，必须执行以下代码：

```
ops$tkyte@ORA10G> update id_table

  2 set id_value = id_value+1

  3 where id_name = 'MY_KEY';

1 row updated.



ops$tkyte@ORA10G> select id_value

  2 from id_table

  3 where id_name = 'MY_KEY';

ID_VALUE

----------

1
```

看上去很简单，但是有以下结果（注意结果不止一项）：

- 一次只能有一个用户处理事务行。需要更新这一行来递增计数器，这会导致程序必须串行完成这个操作。在最好的情况下，一次只有一个人生成一个新的键值。
- 在 Oracle 中（其他数据库中的行为可能有所不同），倘若隔离级别为 SERIALIZABLE，除第一个用户外，试图并发完成此操作的其他用户都会接到这样一个错误：“ORA-08177: can't serialize access for this transaction”（ORA-08177：无法串行访问这个事务）。

例如，使用一个可串行化的事务（在 J2EE 环境中比较常见，其中许多工具都自动将 SERIALIZABLE 用作默认的隔离模式，但开发人员通常并不知道），你会观察到以下行为。注意 SQL 提示符（使用 SET SQLPROMPT SQL*Plus 命令）包含了活动会话的有关信息：

```
OPS$TKYTE session(261,2586)> set transaction isolation level serializable;

Transaction set.



OPS$TKYTE session(261,2586)> update id_table

  2 set id_value = id_value+1

  3 where id_name = 'MY_KEY';

1 row updated.



OPS$TKYTE session(261,2586)> select id_value

  2 from id_table

  3 where id_name = 'MY_KEY';



ID_VALUE

----------

1
```

下面，再到另一个 SQL*Plus 会话完成同样的操作，并发地请求惟一的 ID：

```
OPS$TKYTE session(271,1231)> set transaction isolation level serializable;

Transaction set.
```

```
OPS$TKYTE session(271,1231)> update id_table
  2 set id_value = id_value+1
  3 where id_name = 'MY_KEY';
```

此时它会阻塞，因为一次只有一个事务可以更新这一行。这展示了第一种可能的结果，即这个会话会阻塞，并等待该行提交。但是由于我们使用的是 Oracle，而且隔离级别是 SERIALIZABLE，提交第一个会话的事务时会观察到以下行为：

```
OPS$TKYTE session(261,2586)> commit;
Commit complete.
```

第二个会话会立即显示以下错误：

```
OPS$TKYTE session(271,1231)> update id_table
  2 set id_value = id_value+1
  3 where id_name = 'MY_KEY';
update id_table
*
ERROR at line 1:
ORA-08177: can't serialize access for this transaction
```

所 以，尽管这个逻辑原本想做到独立于数据库，但它根本不是数据库独立的。取决于隔离级别，这个逻辑甚至在单个数据库中都无法可靠地完成，更不用说跨数据库 了！有时我们会阻塞并等待，但有时却会得到一条错误消息。说得简单些，无论是哪种情况（等待很长时间，或者等待很长时间后得到一个错误），都至少会让最终 用户不高兴。

实际上，我们的事务比上面所列的要大得多，所以问题也更为复杂。实际的事务中包含多条语句，上例中的 UPDATE 和 SELECT 只 是其中的两条而已。我们还要用刚生成的这个键向表中插入行，并完成这个事务所需的其他工作。这种串行化对于应用的扩缩是一个很大的制约因素。如果把这个技 术用在处理订单的网站上，而且使用这种方式来生成订单号，可以想想看可能带来的后果。这样一来，多用户并发性就会成为泡影，我们不得不按顺序做所有事情。

对于这个问题，正确的解决方法是针对各个数据库使用最合适的代码。在 Oracle 中，代码应该如下（假设表 T 需要所生成的主键）：

```
create table t ( pk number primary key, ... );
```

```
create sequence t_seq;

create trigger t_trigger before insert on t for each row

begin

        select t_seq.nextval into :new.pk from dual;

end;
```

其效果是为所插入的每一行自动地（而且透明地）指定一个惟一键。还有一种性能更优的方法：

```
Insert into t ( pk, ... ) values ( t_seq.NEXTVAL, ... );
```

也就是说，完全没有触发器的开销（这是我的首选方法）。

在第一个例子中，我们特意使用了各个数据库的特性来生成一个非阻塞、高度并发的惟一键，而且未对应用代码带来任何真正的改动，因为在这个例子中所有逻辑都包含在 DDL 中。

提示 在其他数据库中也可以使用其内置的特性或者生成惟一的数来达到同样的效果。CREATE TABLE 语法可能不同，但是最终结果是一样的。

理解了每个数据库会以不同的方式实现特性，再来看一个支持可移植性的防御式编程的例子，这就是必要时将数据库访问分层。例如，假设你在使用 JDBC 进行编程，如果你用的都是直接的 SQL（SELECT、INSERT、UPDATE 和 DELETE），可能不需要抽象层。你完全可以在应用程序中直接编写 SQL，前提是只能用各个数据库都支持的构造，而且经验证，这些构造在不同数据库上会以同样的方式工作（还记得关于 NULL=NULL 的讨论吧！）。另一种方法的可移植性更好，而且可以提供更好的性能，就是使用存储过程来返回结果集。你会发现，每个开发商的数据库都可以从存储过程返回结果集，但是返回的方式不同。针对不同的数据库，要编写的具体源代码会有所不同。

这 里有两个选择，一种做法是不使用存储过程返回结果集，另一种做法是针对不同的数据库实现不同的代码。我就坚持第二种做法，即针对不同的开发商编写不同的代 码，而且大量使用存储过程。初看上去，另换一个数据库实现时这好像会增加开发时间。不过你会发现，在多个数据库上实现时，采用这种方法实际上容易得多。你 不用寻找适用于所有数据库的最佳 SQL（也许在某些数据库上表现好一些，但在另外一些数据库上可能并不理想），而只需实现最适合该数据库的 SQL。这些工作可以在应用之外完成，这样对应用调优时就有了更大的灵活性。你可以在数据库自身中修正一个表现很差的查询，并立即部署所做的改动，而无需修改应用。另外，采用这种方法，还可以充分利用开发商提供的 SQL 扩缩。例如，Oracle 在其 SQL 中提供了 CONNECT BY 操作，能支持层次查询。这个独有的特性对于处理递归查询很有意义。在 Oracle 中，你可以自由地使用这个 SQL 扩缩，因为它在应用“之外”（也就是说，隐藏在数据库中）。在其他数据库中，则可能需要使用一个临时表，并通过存储过程中的过程性代码才能得到同样的结果。既然你花钱购买了这些特性，自然可以充分地加以使用。

应用要在哪个数据库上部署，就针对这个数据库开发一个专用的代码层，这种技术与实现多平台代码所用的开发技术是一样的。例如，Oracle 公司在开发 Oracle 数据库时就使用了这些技术。这一层代码量很大（但相对于数据库的全部代码来讲，还只是很少的一部分），

称为操作系统相关（operating system-dependent，OSD）代码，是专门针对各个平台实现的。使用这层抽象，Oracle 就能利用许多本地 OS 特性来提高性能和支持集成，而无需重写数据库本身的很大一部分代码。Oracle 能作为一个多线程应用在 Windows 上运行，也能作为一个多进程应用在 UNIX 上运行，这就反映出 Oracle 利用了这种 OSD 代码。它将进程间通信的机制抽象到这样一个代码层上，可以根据不同的操作系统重新实现，所以允许有完全不同的实现，它们的表现与直接（专门）为各平台编写的应用相差无几。

采用这个方法还有一个原因，要想找到一个样样精通的开发人员，要求他熟知 Oracle、SQL Server 和 DB2 之间的细微差别（这里只讨论这 3 个数据库）几乎是不可能的，更别说找到这样一个开发小组了。我在过去 11 年间一直在用 Oracle（大体如此，但不排除其他软件）。每一天使用 Oracle，都会让我学到一些新的东西。但我还是不敢说同时精通这 3 种数据库，知道它们之间的差别，并且清楚这些差别会对要构建的“泛型代码”层有什么影响。我觉得自己无法准确或高效地实现这样一个“泛型代码”层。再说了，我们指的是一般的开发人员，有多少开发人员能真正理解或充分使用了手上的数据库呢？更别说掌握这 3 种数据库了！要寻找这样一个“全才”，他能开发安全、可扩缩而且独立于数据库的程序，就像是大海捞针一样。而希望由这样的人员组建一支开发队伍更是绝无可能。反过来，如果去找一个 Oracle 专家、一个 DB2 专家和一个 SQL Server 专家，告诉他们“我们需要事务完成 X、Y 和 Z”，这倒是很容易。只需告诉他们“这是你的输入，这些是我们需要的输出，这是业务过程要做的事情”，根据这些来生成满足要求的事务性 API（存储过程）就很简单了。针对特定的数据库，按照数据库特有的一组功能，可以采用最适于该数据库的方式来实现。开发人员可以自由地使用底层数据库平台的强大能力（也可能底层数据库缺乏某种能力，而需要另辟蹊径）。

3. 特性和功能

你不必努力争取数据库独立性，这还有一个很自然的理由：你应当准确地知道特定数据库必须提供什么，并充分加以利用。这一节不会列出 Oracle 10g 提供的所有特性，光是这些特性本身就需要一本很厚的书才能讲完。Oracle 9i Release 1、9i Release 2 和 10g Release 1 本身的新特性在 Oracle 文档中已做介绍。Oracle 为此提供了大约 10 000 页的文档，涵盖了每一个有意义的特性和功能。你起码要对数据库提供的特性和功能有一个大致的了解，这一节只是讨论大致了解有什么好处。

前面提到过，我总在 http://asktom.oracle.com 上回答有关 Oracle 的问题。我说过，我的答案中 80%都只是给出相关文档的 URL（这是指我公开提出的那些问题，其中许多答案都只是指向文档，另外还会有几个问题我没有公开提出，因为这些问题的答案几乎都是“读读这本书”）。人们问我怎么在数据库中编写一些复杂的功能（或者在数据库之外编写），我就会告诉他们在文档的哪个地方可以了解到 Oracle 已经实现了这个功能，并且还说明了应该如何使用这个功能。我时常会遇到一些有关复制的问题。可能有这样一个问题：“我想在每个地方都留有数据的一个副本。我希望这是一个只读的副本，而且每天只在半夜更新一次。我该怎么编写代码来做到呢？”答案很简单，只是一个 CREATE MATERIALIZED VIEW 命令而已。这是数据库中的一个内置功能。实际上，实现复制还有许多方法，从只读的物化视图到可更新的物化视图，再到对等复制以及基于流的复制，等等。

你当然可以编写你自己的复制，这么做可能很有意思，但是从最后看来，自己编写可能不是最明智的做法。数据库做了很多工作。一般来说，数据库会比我们自己做得更好。例如，Oracle 中复制是用 C 编写的，充分考虑到了国际化。不仅速度快、相当容易，而且很健壮。它允许跨版本和跨平台，并且提供了强大的技术支持，所以倘若你遇到问题，Oracle

Support 会 很乐意提供帮助。如果你要升级，也会同步地提供复制支持，可能还会增加一些新的特性。下面考虑一下如果由你自己来开发会怎么样。你必须为每一个版本都提供 支持。老版本和新版本之间的互操作性谁来负责？这个任务会落在你的头上。如果出了“问题”，你没有办法寻求支持，至少在得到一个足够小的测试用例（但足以 展示你的主要问题）之前，没有人来帮助你。当新版本的 Oracle 推出时，也要由你自己将你的复制代码移植到这个新版本。

如果没有充分地了解数据库已经提供了哪些功能，从长远看，其坏影响还会几次三番地出现。我曾经与一些有多年数据库应用开发经验的人共事，不过他们原先是在其他数据库上开发应用。这一次他们在 Oracle 上构建了一个分析软件（趋势分析、报告和可视化软件），要用于分析临床医学数据（与保健相关）。这些开发人员不知道 SQL 的一些语法特性，如内联视图、分析功能和标量子查询。他们遇到的一个主要问题是需要分析一个父表及两个子表的数据。相应的实体—关系图（entity-relationship diagram，ERD）如图 1-1 所示。

图 1-1 简单的 ERD

他们想生成父记录的报告，并提供子表中相应子记录的聚集统计。他们原来使用的数据库不支持子查询分解（WITH 子 句），也不支持内联视图（所谓内联视图，就是 “查询一个查询”，而不是查询一个表）。由于不知道有这些特性，开发人员们在中间层编写了他们自己的一个数据库。他们的做法是先查询父表，对应返回的每一 行，再对各个子表分别运行聚集查询。这样做的后果是：对于最终用户想要运行的每一个查询，他们都要运行数千个查询才能得到所需的结果。或者，他们的另一种 做法是在中间层获取完整的聚集子表，再放入内存中的散列表，并完成一个散列联结（hash join）。

简而言之，他们重新开发了一个数据库，自行完成了与嵌套循环联结或散列联结相当的功能，而没有充分利用临时表空间、复杂的查询优化器等所提供的好处。这些开发人员把大量时间都花费在这个软件的开发、设计、调优和改进上，而这个软件只是要做数据库已经做了的事情，要知道他们原本已经花钱买了这些功能！与此同 时，最终用户还在要求增加新特性，但是一直没有如愿，因为开发人员总忙于开发报告“引擎”，没有更多的时间来考虑这些新特性，实际上这个报告引擎就是一个伪装的数据库引擎。

我告诉他们，完全可以联结两个聚集来比较用不同方法以不同详细程度存储的数据（见代码清单 1-1～代码清单 1-3）。

代码清单 1-1 内联视图：对“查询”的查询

```
select p.id, c1_sum1, c2_sum2

from p,

    (select id, sum(q1) c1_sum1

    from c1
```

```
        group by id) c1,
        (select id, sum(q2) c2_sum2
        from c2
        group by id) c2
where p.id = c1.id
        and p.id = c2.id
/
```

代码清单 1-2 标量子查询：每行运行另一个查询

```
select p.id,
        (select sum(q1) from c1 where c1.id = p.id) c1_sum1,
        (select sum(q2) from c2 where c2.id = p.id) c2_sum2
from p
where p.name = '1234'
/
```

代码清单 1-3 WITH 子查询分解

```
with c1_vw as
        (select id, sum(q1) c1_sum1
        from c1
        group by id),
c2_vw as
        (select id, sum(q2) c2_sum2
        from c2
        group by id),
c1_c2 as
```

```
        (select c1.id, c1.c1_sum1, c2.c2_sum2

        from c1_vw c1, c2_vw c2

        where c1.id = c2.id )

select p.id, c1_sum1, c2_sum2

from p, c1_c2

where p.id = c1_c2.id

/
```

更何况他们还可以使用 LAG、LEAD、ROW_NUMBER 之类的分析函数、分级函数等。我们没有再花时间去考虑如何对他们的中间层数据库引擎进行调优，而是把余下的时间都用来学习 SQL Reference Guide，我们把它投影在屏幕上，另外还打开一个 SQL*Plus 实际演示到底如何工作。最终目标不是对中间层调优，而是尽快地把中间层去掉。

我曾经见过许多人在 Oracle 数据库中建立后台进程从管道（一种数据库 IPC 机制）读消息。这些后台进程执行管道消息中包含的 SQL， 并提交工作。这样做是为了在事务中执行审计，即使更大的事务（父事务）回滚了，这个事务（子事务）也不会回滚。通常，如果使用触发器之类的工具来审计对某 数据的访问，但是后来有一条语句失败，那么所有工作都会回滚。所以，通过向另一个进程发送消息，就可以有一个单独的事务来完成审计工作并提交。即使父事务 回滚，审计记录仍然保留。在 Oracle8i 以前的版本中，这是实现此功能的一个合适的方法（可能也是惟一的方法）。我告诉他们，数据库还有一个称为自治事务（autonomous transaction） 的特性，他们听后很是郁闷。自治事务的实现只需一行代码，就完全可以做到他们一直在做的事情。好的一面是，这说明他们可以丢掉原来的大量代码，不用再维护 了。另外，系统总的来讲运行得更快，而且更容易理解。不过，他们还在为“重新发明”浪费了那么多时间而懊恼不已。特别是那个写后台进程的开发人员更是沮 丧，因为他写了一大堆的代码。

还是我反复重申的那句话：针对某个问题，开发人员力图提供复杂的大型解决方案，但数据库本身早已解决了这个问题。在这个方面，我自己也有些心虚。我还记得，有一天我的 Oracle 销售顾问走进我的办公室（那时我还只是一个客户），看见我被成堆的 Oracle 文档包围着。我抬起头，问他“这是真的吗？”接下来的几天我一直在深入研究这些文档。此前我落入一个陷阱，自以为“完全了解数据库”，因为我用过 SQL/DS、DB2、Ingress、Sybase、Informix、SQLBase、Oracle，还有其他一些数据库。我没有花时间去了解每个数据库提供了什么，而只是把从其他数据库学到的经验简单地应用到当时正在使用的数据库上（移植到 Sybase/SQL Server 时对我的触动最大，它与其他数据库的工作根本不一样）。等到我真正发现 Oracle（以及其他数据库）能做什么之后，我才开始充分利用它，不仅能更快地开发，而且写的代码更少。我认识到这一点的时候是 1993 年。请仔细想想你能用手头的软件做些什么，不过与我相比，你已经晚了十多年了。

除非你花些时间来了解已经有些什么，否则你肯定会在某个时候犯同样的错误。在这本书中，我们会深入地分析数据库提供的一些功能。我选择的是人们经常使用的特性和功能，或者是本应更多地使用但事实上没有得到充分利用的功能。不过，这里涵盖的内容只是冰山

一角。Oracle 的知识太多了，单用一本书来讲清楚是做不到的。

重申一遍：每天我都会学到 Oracle 的一些新知识。这需要“与时俱进”，时刻跟踪最新动态。我自己就常常阅读文档（不错，我还在看文档）。即使如此，每天还是会有人指出一些我不知道的知识。

4. 简单地解决问题

通常解决问题的途径有两种：容易的方法和困难的方法。我总是看到人们在选择后者。这并不一定是故意的，更多的情况下，这么做只是出于无知。他们没想到数据库 能“做那个工作”。而我则相反，我总是希望数据库什么都能做，只有当我发现它确实做不了某件事时才会选择困难的办法（自己来编写）。

例如，人们经常问我，“怎么确保最终用户在数据库中只有一个会话？”（其实类似这样的例子还有很多，我只是随便选了一个）。可能许多应用都有这个需求，但是 我参与的应用都没有这样做，我不知道有什么必要以这种方式限制用户。不过，如果确实想这样做，人们往往选择困难的方法来实现。例如，他们可能建立一个由操作系统运行的批作业，这个批作业将查看 V$SESSION 表；如果用户有多个会话，就坚决地关闭这 些会话。还有一种办法，他们可能会创建自己的表，用户登录时由应用在这个表中插入一行，用户注销时删除相应行。这种实现无疑会带来许多问题，于是咨询台的 铃声大作，因为应用“崩溃”时不会将该行删除。为了解决这个问题，我见过许多“有创意的”方法，不过哪一个也没有下面这种方法简单：

```
ops$tkyte@ORA10G> create profile one_session limit sessions_per_user 1;

Profile created.



ops$tkyte@ORA10G> alter user scott profile one_session;

User altered.



ops$tkyte@ORA10G> alter system set resource_limit=true;

System altered.



ops$tkyte@ORA10G> connect scott/tiger

Connected.



scott@ORA10G> host sqlplus scott/tiger
```

```
SQL*Plus: Release 10.1.0.2.0 - Production on Sun Nov 28 12:49:49 2004

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

ERROR:

ORA-02391: exceeded simultaneous SESSIONS_PER_USER limit



Enter user-name:
```

仅此而已。现在有 ONE_SESSION 配置文件的所有用户都只能登录一次。每次我提出这个解决方案时，人们总是拍着自己的脑门，不无惊羡地说：“我不知道居然还能这么做！”正所谓磨刀不误砍柴工，花些时间好好熟悉一下你所用的工具，了解它能做些什么，在开发时这会为你节省大量的时间和精力。

还 是这句“力求简单”，它也同样适用于更宽泛的体系结构层。我总是鼓励人们在采用非常复杂的实现之前先要再三思量。系统中不固定的部分越多，出问题的地方就 越多。在一个相当复杂的体系结构中，要想准确地跟踪到错误出在哪里不是一件容易的事。实现一个有“无数”层的应用可能看起来很“酷”，但是既然用一个简单 的存储过程就能更好、更快地完成任务，而且只利用更少的资源，实现为多层的做法就不是正确的选择。

我 见过许多项目的应用开发持续数月之久，好像没有尽头。开发人员都在使用最新、最好的技术和语言，但是开发速度还是不快。应用本身的规模并不大，也许这正是 问题所在。如果你在建一个狗窝（这是一个很小的木工活），就不会用到重型机器。你只需要几样小工具就行了，大玩艺是用不上的。另一方面，如果你在建一套公 寓楼，就要下大功夫，可能要用到大型机器。与建狗窝相比，解决这个问题所用的工具完全不同。应用开发也是如此。没有一种“万全的体系结构”，没有一种“完 美的语言”，也没有一个“无懈可击的方法”。

例如，我就使用了 HTML DB 来建我的网站。这是一个很小的应用，只有一个（或两个）开发人员参与。它有大约 20 个界面。这个实现使用 PL/SQL 和 HTML DB 是合适的，这里不需要用 Java 编写大量的代码，不需要建立 EJB，等等。这是一个简单的问题，所以应该用简单的方式解决。确实有一些大型应用很复杂、规模很大（如今这些应用大多会直接购买，如人力资源 HR 系统、ERP 系统等），但是小应用更多。我们要选用适当的方法和工具来完成任务。

不论什么时候，我总是提倡用最简单的体系结构来解决问题，而不要采用复杂的体系结构。这样做可能有显著的回报。每种技术都有自己合适的位置。不要把每个问题都当成钉子，高举铁锤随处便砸，我们的工具箱里并非只有铁锤。

5. 开放性

我经常看到，人们选择艰难的道路还有一个原因。这还是与那种观点有关，我们总认为要不遗余力地追求开放性和数据库独立性。开发人员希望避免使用封闭的专有数据库特性，即使像存储过程或序列这样简单的特性也不敢用，因为使用这些专有特性会把他们锁定到某个数据库系统。这么说吧，我的看法是只要你开发一个涉及读/写的应用，就已经在某种程度上被锁定了。一旦开始运行查询和修改，你就会发现数据库间存在着一些微小的差别

（有时还可能存在显著差异）。例如，在一个数据库中，你可能发现 SELECT COUNT(*) FROM T 查询与两行记录的更新发生了死锁。在 Oracle 中，却发现 SELECT COUNT(*)绝 对不会阻塞写入器。你可能见过这样的情况，一个数据库看上去能保证某种业务规则，这是由于该数据库锁定模型的副作用造成的，但另一个数据库则不能保证这个 业务规则。给定完全相同的事务，在不同数据库中却有可能报告全然不同的答案，原因就在于数据库的实现存在一些基本的差别。你会发现，要想把一个应用轻轻松 松地从一个数据库移植到另一个数据库，这种应用少之又少。不同数据库中对于如何解释 SQL（例如，NULL=NULL 这个例子）以及如何处理 SQL 往往有不同的做法。

在我最近参与的一个项目中，开发人员在使用 Visual Basic、ActiveX 控件、IIS 服务器和 Oracle 构建一个基于 Web 的产品。他们不无担心地告诉我，由于业务逻辑是用 PL/SQL 编写的，这个产品已经依赖于数据库了。他们问我：“怎么修正这个问题？”

先不谈这个问题，退一步说，针对他们所选的技术，我实在看不出依赖于数据库有什么“不好”：

- 开发人员选择的语言已经把他们与一个开发商提供的一个操作系统锁定（要想独立于操作系统，其实他们更应选择 Java）。
- 他们选择的组件技术已经把他们与一个操作系统和一个开发商锁定（选择 J2EE 更合适）。
- 他们选择的 Web 服务器已经将他们与一个开发商和一个平台锁定（为什么不用 Apache 呢？）。

所选择的每一项技术都已经把他们锁定到一个非常特定的配置，实际上，就操作系统而言，惟一能让他们有所选择的技术就是数据库。

暂且不管这些（选择这些技术可能有他们自己的原因），这些开发人员还刻意不去用体系结构中一个重要部件的功能，而美其名曰是为了开放性。在我看来，既然精心 地选择了技术，就应该最大限度地加以利用。购买这些技术你已经花了不少钱，难道你想白白地花冤枉钱吗？我认为，他们一直想尽力发挥其他技术的潜能，那么为 什么要把数据库另眼相看呢？再者，数据库对于他们的成功至关重要，单凭这一点也说明，不充分利用数据库是说不过去的。

如果从开放性的角度来考虑，可以稍稍换个思路。你把所有数据都放在数据库中。数据库是一个很开放的数据池。它支持通过大量开放的系统协议和访问机制来访问数据。这听起来好像很不错，简直就是世界上最开放的事物。

不过接下来，你把所有应用逻辑还有（更重要的）安全都放在数据库之外。可能放在访问数据的 bean 中；也可能放在访问数据的 JSP 中；或者置于在 Microsoft 事务服务器（Microsoft Transaction Server，MTS）管理之下运行的 Visual Basic 代码中。最终结果就是，你的数据库被封闭起来，这么一来，数据库已经被你弄得“不开放”了。人们无法再采用现有技术使用这些数据；他们必须使用你的访问方法（或者干脆绕过你的安全防护）。尽管现在看上去还不错，但是你要记住，今天响当当的技术（比如说，EJB）也会成为昨日黄花，到了明天可能就是一个让人厌倦的技术了。在关系领域中（以及大多数对象实现中），过去 25 年来只有数据库自己傲然屹立。数据前台技术几乎每年一变，如果应用把安全放在内部实现，而不是在数据库中实现，随着前台技术的变革，这些应用就会成为前进道路上的绊脚石。

Oracle 数据库提供了一个称为细粒度访问控制（fine-grained access control，FGAC）的特性。简而言之，这种技术允许开发人员把过程嵌入数据库中，向数据库提交查询时可以修改查询。这种查询修改可用于限制客户只能接收或修改某些行。过程在运行查询时能查看是谁在运行查询，他们从哪个终端运行查询，等等，然后能适当地约束对数据的访问。利用FGAC，可以保证以下安全性，例如：

- 某类用户在正常工作时间之外执行的查询将返回 0 条记录。
- 如果终端在一个安全范围内，可以向其返回所有数据，但是远程客户终端只能得到不敏感的信息。

实质上讲，FGAC 允许我们把访问控制放在数据库中，与数据“如影随形”。不论用户从 bean、JSP、使用 ODBC 的 Visual Basic 应用，还是通过 SQL*Plus 访问数据，都会执行同样的安全协议。这样你就能很好地应对即将到来的下一种新技术。

现在我再来问你，你想让所有数据访问都通过调用 Visual Basic 代码和 ActiveX 控件来完成（如果愿意，也可以把 Visual Basic 换成 Java，把 ActiveX 换成 EJB，我并不是推崇哪一种技术，这里只是泛指这种实现）；还是希望能从任何地方访问数据（只要能与数据库通信，而不论协议是 SSL、HTTP、Oracle Net，还是其他协议，也不论使用的是 ODBC、JDBC、OCI，还是其他 API，这两种实现中哪一种更“开放”？ 我还没见过哪个报告工具能“查询”Visual Basic 代码，但是能查询 SQL 的工具却有不少。

人们总是不遗余力地去争取数据库独立性和完全的开放性，但我认为这是一个错误的决定。不管你使用的是什么数据库，都应该充分地加以利用，把它的每一个功能都“挤出来”。不论怎样，等到调优阶段你也会这样做的（不过，往往在部署之后才会调优）。如果通过充分利用软件的功能，会让你的应用快上 5 倍，你会惊讶地发现，居然这么快就把数据库独立性需求抛在脑后了。

1.3.5 “怎么能让应用运行得更快?”

总是有人问我这个问题：“怎么能让应用运行得更快?”所有人都希望有一个“fast = true”开关，认为“数据库调优”就意味着让你调整数据库。实际上，根据我的经验，80%以上（甚至经常是 100%）的性能问题都出现在设计和实现级，而不是数据库级。通过修改应用，我常常能让性能呈数量级地增长。但是，如果只是在数据库级做修改，就不太可能得到这么大幅度的提高。在对数据库上运行的应用进行调优之前，先不要对数据库进行调优。

随着时间的推移，数据库级也有了一些开关，有助于减轻编程错误带来的影响。例如，Oracle 8.1.6 增加了一个新参数 CURSOR_SHARING=FORCE。如果你愿意，这个特性会实现一个自动绑定器（auto-binder）。如果有一个查询编写为 SELECT * FROM EMP WHERE EMPNO = 1234，自动绑定器会悄无声息地把它改写成 SELECT * FROM EMP WHERE EMPNO = :x。这确实能动态地大大减少硬解析数，并减少前面讨论的库闩等待时间——但是（凡事总有个“但是”），它可能有一些副作用。游标共享的一个常见副作用如下所示：

```
ops$tkyte@ORA10G> select /* TAG */ substr( username, 1, 1 )
  2 from all_users au1
  3 where rownum = 1;
```

```
S

-

B

ops$tkyte@ORA10G> alter session set cursor_sharing=force;

Session altered.



ops$tkyte@ORA10G> select /* TAG */ substr( username, 1, 1 )

  2 from all_users au2

  3 where rownum = 1;

SUBSTR(USERNAME,1,1)



------------------------------

B
```

这里到底发生了什么？为什么到第二个查询时 SQL*Plus 报告的列突然变得这么大？要知道，这还是同一个查询呀！如果查看一下游标共享设置为我们做了些什么，原因就会很清楚了（还会明白其他一些问题）：

```
ops$tkyte@ORA10G> select sql_text from v$sql

  2 where sql_text like 'select /* TAG */ %';

SQL_TEXT

--------------------------------------------------------------------------------

select /* TAG */ substr( username, 1, 1 ) from all_users au1 where rownum = 1

select /* TAG */ substr( username, :"SYS_B_0", :"SYS_B_1" ) from all_users au2 where
rownum = :"SYS_B_2"
```

游标共享会删除查询中的信息。它找到每一个直接量（literal），包括内置求子串函数（substr）的参数，直接量就是我们使用的常量。它把这些直接量从查询中删除，并代之以绑定变量。SQL 引擎再也不知道这个列是长度为 1 的子串，它的长度是不确定的。另外，

可以看到 where rownum = 1 现在也已经绑定。看上去似乎不错；不过，优化器把一个重要的信息也一并删除了。它不知道“这个查询将获取一行”；现在只认为“这个查询将返回前 N 行，而 N 可能是任何值”。实际上，如果加上 SQL_TRACE=TRUE 后再运行这些查询，你会发现每个查询使用的查询计划都不同，它们完成的工作量也大相径庭。考虑以下查询：

```
select /* TAG */ substr( username, 1, 1 )

from all_users au1

where rownum = 1

call     count       cpu    elapsed       disk      query    current        r
ows

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

Parse        1      0.00       0.00          0          0          0           0

Execute      1      0.00       0.00          0          0          0           0

Fetch        2      0.00       0.00          0         77          0           1

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

total        4      0.00       0.00          0         77          0           1


Misses in library cache during parse: 0

Optimizer mode: ALL_ROWS

Parsing user id: 412


Rows     Row Source Operation

-------  ---------------------------------------------------

1  COUNT STOPKEY (cr=77 pr=0 pw=0 time=5767 us)

1  HASH JOIN (cr=77 pr=0 pw=0 time=5756 us)

1028  HASH JOIN (cr=70 pr=0 pw=0 time=8692 us)
```

```
9                TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=335 us)

1028          TABLE ACCESS FULL USER$ (cr=55 pr=0 pw=0 time=2140 us)

4                TABLE ACCESS FULL TS$ (cr=7 pr=0 pw=0 time=56 us)

********************************************************************************
******

select /* TAG */ substr( username, :"SYS_B_0", :"SYS_B_1" )

from all_users au2

where rownum = :"SYS_B_2"


call          count                cpu          elapsed     disk             query          current          r
ows

-------       ------    --------     ----------     ----------     ----------       ----------       ----------

Parse        1          0.00        0.00          0              0                0                0

Execute 1          0.00        0.00          0              0                0                0

Fetch        2          0.00        0.00          0              85               0                1

-------       ------    --------     ----------     ----------     ----------       ----------       ----------

total          4          0.00        0.00          0              85               0                1


Misses in library cache during parse: 0

Optimizer mode: ALL_ROWS

Parsing user id: 412

Rows          Row Source Operation

-------          ---------------------------------------------------

1                COUNT (cr=85 pr=0 pw=0 time=3309 us)
```

```
1          FILTER (cr=85 pr=0 pw=0 time=3301 us)

1028       HASH JOIN (cr=85 pr=0 pw=0 time=5343 us)

1028       HASH JOIN (cr=70 pr=0 pw=0 time=7398 us)

9          TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=148 us)

1028       TABLE ACCESS FULL USER$ (cr=55 pr=0 pw=0 time=1079 us)

9          TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=90 us)
```

查询计划有一些微小的差别（有时甚至完全不同）；另外它们的工作量也有很大差异。所以，打开游标共享确实需要特别谨慎（而且需要进行充分测试）。游标共享可能会改变应用的行为（例如，列宽发生变化），而且由于它删除了 SQL 中的所有直接量，甚至包括那些绝对不会变化的直接量，所以可能会对查询计划带来负面影响。

另外，与解析和优化大量各不相同的查询相比，尽管使用 CURSOR_SHARING = FORCE 会 让运行速度更快，但同时我也发现，倘若开发人员确实在查询中使用了绑定变量，查询的速度就比使用游标共享要快。这不是因为游标共享代码的效率不高，而是因 为程序本身的效率低下。在许多情况下，如果应用没有使用绑定变量，也不会高效地解析和重用游标。因为应用认为每个查询都是惟一的（并把查询分别建立为不同 的语句），所以绝对不会多次使用一个游标。事实上，如果程序员刚开始就使用了绑定变量，他（或她）就能只解析一次查询，然后多次重用它。正是这种解析开销 降低了总体性能。

实质上讲，一定要记住重要的一点，只打开 CURSOR_SHARING = FORCE 并不一定能解决你的问题。而且游标共享还可能带来新的问题：在有些情况下 CURSOR_SHARING 是一个非常有用的工具，但它不是银弹。开发得很好的应用从不需要游标共享。从长远来看，要尽可能地使用绑定变量，而在需要时才使用常量，这才是正确的做法。

注意 世上没有银弹——要记住，根本没有。如果有的话，自然就会默认地采用那种做法，这样也就无所谓银弹了。

就算是确实能在数据库级放几个开关（这种开关真的很少），但是有些问题与并发控制和执行不佳的查询（可能是因为查询写得不好，也可能是因为数据的结构性差）有关，这些问题用开关是解决不了的。这些情况往往需要重写（而且时常需要重建）。移动数据文件、修改多块读计数（multiblock read count）和其他数据库级开关对应用的总体性能通常影响很小。你想让用户接受你的应用，可能需要让性能提升 2 倍、3 倍、……、n 倍才行。你的应用是不是只慢了 10%，这种情况多不多？如果只是慢 10%，没有人会有太多抱怨。但是如果慢了 5 倍，就会让人很不高兴。再说一遍，如果只是移动数据文件，性能不会提升 5 倍。要想达到这个目的，只能通过调整应用才能办到，可能要让它大幅减少 I/O 操作。

在 整个开发阶段，你都要把性能作为一个目标精心地设计，合理地构建，并且不断地测试。绝对不能把它当作马后炮，事后才想起来。我真是很奇怪，为什么那么多人 根本不对应用调优，就草率地把应用交付到客户手里，匆匆上马，并运行起来。我见过一些应用除了主键索引外，居然没有其他的任何索引。从来没有对查询执行过 调优，也没有执行过压力测试。应用的用户数很少，从未让更多的用户试用过。这些应用总是把调优当成产品安装的一部分。对我来说，这种做法绝对不可接受。最 终用户应该第一天就拿到一个响应迅速、充分优化的系统。肯定还有许多“产品问题”需要处理，但不能让用户从一开始就领教糟糕

的性能。对用户来说，一个新应 用里有几个 bug 尚能容忍，但你别指望他们能耐心地在屏幕前等待漫长的时间。

1.3.6 DBA 与开发人员的关系

有一点很肯定，要建立最成功的信息系统，前提是 DBA 与应用开发人员之间要有一种“共生关系”。在这一节里，我想从开发人员的角度谈谈开发人员与 DBA 之间的分工（假设所有正式开发都有 DBA 小组的参与）。

作为一名开发人员，你不必知道如何安装和配置软件。这应该是 DBA 或者系统管理员（system administrator，SA）的任务。安装 Oracle Net、配置监听器、配置共享服务器、建立连接池、安装数据库、创建数据库等，这些事情我都会交给 DBA/SA 来做。

一般来讲，开发人员不必知道如何对操作系统调优。我个人通常会让系统的 SA 负责这个任务。作为数据库应用的软件开发人员，应该能熟练地使用你选择的操作系统，但是不要求你能对它调优。

DBA 最重大的职责是数据库恢复。注意，我说的可不是“备份”，而是“恢复”。而且，我认为这也是 DBA 惟一重要的职责。DBA 要知道回滚（rollback）和重做（redo）怎么工作，不错，这也是开发人员要了解的。DBA 还要知道如何完成表空间时间点恢复，这一点开发人员不必介入。如果你能有所了解，也许以后会用得上，但是作为开发人员目前不必亲力而为。

在数据库实例级调优，并得出最优的 PGA_AGGREGATE_TARGET 是什么，这一般是 DBA 的任务（数据库往往能帮助他们得出正确的答案）。也有一些例外情况，有时开发人员可能需要修改会话的某个设置，但是如果在数据库级修改设置，就要由 DBA 来负责。一般数据库并不是只支持一位开发人员的应用，而是运行着多个应用，因此只有支持所有应用的 DBA 才能做出正确的决定。

分配空间和管理文件也是 DBA 的工作。开发人员可以对分配的空间做出估计（他们觉得需要多少空间），但是余下的都要由 DBA/SA 决定。

实质上讲，开发人员不必知道如何运行数据库，他们只需要知道如何在数据库中运行。开发人员和 DBA 要协同解决问题，但各有分工。假设你是一位开发人员，如果你的查询用的资源太多，DBA 就会来找你；如果你不知道怎么让系统跑得更快，可以去找 DBA（如果应用已经得到充分调优，此时就可以完成实例级调优）。

这些任务因环境而异，不过我还是认为存在着分工。好的开发人员往往是很糟糕的 DBA， 反之亦然。在我看来，他们的能力不同、思路不同，而且个性也不同。很自然地，人们都爱做自己最喜欢的工作，而且能越做越好，形成良性循环。如果一个人比较 喜欢某项工作，他会做得更好，但是这并不是说其他工作就一定做得很糟。就我而言，我觉得我更应算是一位开发人员，但兼有 DBA 的许多观点。我不仅喜欢开发，也很喜欢“服务器方面”的工作（这大大提高了我的应用调优水平，而且总会有很多收获）。

1.4 小结

这一章好像一直在东拉西扯地闲聊，我是想用这种方式让你认识到为什么需要了解数据库。这里提到的例子并不是个别现象，这些情况每天都在出现。我注意到，诸如此类的问题总在连续不断地发生。

下面把要点再重述一遍。如果你要用 Oracle 开发，应该做到：

- 需要理解 Oracle 体系结构。不要求你精通到能自行重写服务器的程度，不过确实需要有足够的了解，知道使用某个特定特性的含义。
- 需要理解锁定和并发控制特性，而且知道每个数据库都以不同的方式实现这些特性。如果不清楚这一点，你的数据库就可能给出“错误”的答案，而且应用会遭遇严重的竞争问题，以至于性能低下。
- 不要把数据库当作黑盒，也就是说，不要以为无需了解数据库。在大多数应用中，数据库都是最为重要的部分。如果忽略它，后果是致命的。
- 用尽可能简单的方法解决问题，要尽量使用 Oracle 提供的内置功能。这可是你花大价钱买来的。
- 软件项目、编程语言以及框架总是如走马灯似地在变。作为开发人员，我们希望几周（可能几个月）内就把系统建立并运行起来，然后再去解决下一个问题。如果总是从头开始重新“创造”，就永远也追不上开发的脚步。你肯定不会用 Java 建立你自己的散列表，因为 Java 已经提供了一个散列表，同样，你也应该使用手头可用的数据库功能。当然，为此第一步是要了解有哪些数据库功能可用。我曾经见过不止一个开发小组遇到麻烦，不光技术上有困难，人员也很紧张，而造成这种结果的原因只是不清楚 Oracle 已经免费提供了哪些功能。
- 还 是上面这一条（软件项目和编程语言总是像走马灯似的），但数据是永远存在的。我们构建了使用数据的应用，从长远看，这些数据会由多个应用使用。所以重点不 是应用，而是数据。应该采用允许使用和重用数据的技术和实现。如果把数据库当成一个桶，所有数据访问都必须通过你的应用，这就错了。这样一来，你将无法自 主地查询应用，也无法在老应用之上构建新应用。但是，如果充分地使用数据库，你就会发现，无论是增加新应用、新报告，还是其他任何功能，都会容易得多

牢记以上这几点，再接着看下面的内容。

第 2 章体系结构概述

Oracle 被设计为一个相当可移植的数据库；在当前所有平台上都能运行，从 Windows 到 UNIX 再到大型机都支持 Oracle。出于这个原因，在不同的操作系统上，Oracle 的物理体系结构也有所不同。例如，在 UNIX 操作系统上可以看到，Oracle 实现为多个不同的操作系统进程，实际上每个主要功能分别由一个进程负责。这种实现对于 UNIX 来说是正确的，因为 UNIX 就是以多进程为基础。不过，如果放到 Windows 上就不合适了，这种体系结构将不能很好地工作（速度会很慢，而且不可扩缩）。在 Windows 平台上，Oracle 实现为一个多线程的进程。如果是一个运行 OS/390 和 z/OS 的 IBM 大型机系统，针对这种操作系统的 Oracle 体系结构则充分利用了多个 OS/390 地址空间，它们都作为一个 Oracle 实例进行操作。一个数据库实例可以配置多达 255 个地址空间。另外，Oracle 还能与 OS/390 工作负载管理器（Workload Manager，WLM）协作，建立特定 Oracle 工作负载相互之间的相对执行优先级，还能建立相对于 OS/390 系统中所有其他工作的执行优先级。尽管不同平台上实现 Oracle 所用的物理机制存在变化，但 Oracle 体系结构还是很有一般性，所以你能很好地了解 Oracle 在所有平台上如何工作。

这一章会从全局角度概要介绍这个体系结构。我们会分析 Oracle 服务器，并给出“数据库”和“实例”等术语的定义（这些术语通常很容易混淆）。这里还会介绍“连接”到 Oracle 时会发生什么，另外将从高层分析服务器如何管理内存。在后续 3 章中，我们还会详细介绍 Oracle 体系结构中的 3 大部分：

- 第 3 章将介绍文件，其中涵盖构成数据库的 5 大类文件：参数文件、数据文件、临时文件、控制文件和重做日志文件。我们还会介绍另外几类文件，包括跟踪文件、警告文件、转储文件（DMP）、数据泵文件（data pump）和简单的平面文件。这一章将谈到 Oracle 10g 新增的一个文件区，称为闪回恢复区（Flashback Recovery Area），另外我们还会讨论自动存储管理（Automatic Storage Management，ASM）对文件存储的影响。

- 第 4 章介绍 Oracle 的一些内存结构，分别称为系统全局区（System Global Area，SGA）、进程全局区（Process Global Area，PGA）和用户全局区（User Global Area，UGA）。我们会分析这些结构之间的关系，并讨论共享池（shared pool）、大池（big pool）、Java 池（Java pool）以及 SGA 中的其他一些组件。

- 第 5 章介绍 Oracle 的物理进程或线程。我们会讨论数据库上运行的 3 类不同的进程：服务器进程（server process）、后台进程（background process）和从属进程（slave process）。

先介绍哪一部分实在很难定夺。由于进程使用了 SGA，所以如果在进程之前先介绍 SGA 可能不太合适。另一方面，讨论进程及其工作时，又会引用 SGA。另外两部分的关系也很紧密：文件由进程处理，如果不先了解进程做什么，将很难把文件搞清楚。

正因如此，我会在这一章定义一些术语，对 Oracle 是什么提供一个一般性的概述（也许你会把它画出来）。有了这些准备，你就能深入探访各个部分的具体细节了。

2.1 定义数据库和实例

在 Oracle 领域中有两个词很容易混淆，这就是“实例”（instance）和“数据库”（database）。作为 Oracle 术语，这两个词的定义如下：

- 数据库（database）：物理操作系统文件或磁盘（disk）的集合。使用 Oracle 10g 的自动存储管理（Automatic Storage Management，ASM）或 RAW 分区时，数据库可能不作为操作系统中单独的文件，但定义仍然不变。

- 实例（instance）：一组 Oracle 后台进程/线程以及一个共享内存区，这些内存由同一个计算机上运行的线程/进程所共享。这里可以维护易失的、非持久性内容（有些可以刷新输出到磁盘）。就算没有磁盘存储，数据库实例也能存在。也许实例不能算是世界上最有用的事物，不过你完全可以把它想成是最有用的事物，这有助于对实例和数据库划清界线。

这两个词有时可互换使用，不过二者的概念完全不同。实例和数据库之间的关系是：数据库可以由多个实例装载和打开，而实例可以在任何时间点装载和打开一个数据库。实际上，准确地讲，实例在其整个生存期中最多能装载和打开一个数据库！稍后就会介绍这样的一个例子。

是不是更糊涂了？我们还会做进一步的解释，应该能帮助你搞清楚这些概念。实例就是一组操作系统进程（或者是一个多线程的进程）以及一些内存。这些进程可以操 作数据库；而数据库只是一个文件集合（包括数据文件、临时文件、重做日志文件和控制文件）。在任何时刻，一个实例只能有一组相关的文件（与一个数据库关 联）。大多数情况下，反过来也成立：一个数据库上只有一个实例对其进行操作。不过，Oracle 的真正应用集群（Real Application Clusters，RAC）是一个例外，这是 Oracle 提供的一个选项，允许在集群环境中的多台计算机上操作，这样就可以有多台实例同时装载并打开一个数据库（位于一组共享物理磁盘上）。由此，我们可以同时从多台不同的计算机访问这个数据库。Oracle RAC 能支持高度可用的系统，可用于构建可扩缩性极好的解决方案。

下面来看一个简单的例子。假设我们刚安装了 Oracle 10g 10.1.0.3。我们执行一个纯软件安装，不包括初始的“启动”数据库，除了软件以外什么都没有。

通过 pwd 命令可以知道当前的工作目录（这个例子使用一个 Linux 平台的计算机）。我们的当前目录是 dbs（如果在 Windows 平台上，则是 database 目录）。执行 ls－l 命令显示出这个目录为“空”。其中没有 init.ora 文件，也没有任何存储参数文件（stored parameter file，SPFILE）；存储参数文件将在第 3 章详细讨论。

```
[ora10g@localhost dbs]$ pwd

/home/ora10g/dbs

[ora10g@localhost dbs]$ ls -l

total 0
```

使用 ps（进程状态）命令，可以看到用户 ora10g 运行的所有进程，这里假设 ora10g 是 Oracle 软件的所有者。此时还没有任何 Oracle 数据库进程。

```
[ora10g@localhost dbs]$ ps -aef | grep ora10g

ora10g 4173 4151 0 13:33 pts/0 00:00:00 -su

ora10g 4365 4173 0 14:09 pts/0 00:00:00 ps -aef

ora10g 4366 4173 0 14:09 pts/0 00:00:00 grep ora10g
```

然后使用 ipcs 命令，这个 UNIX 命令可用于显示进程间的通信设备，如共享内存、信号量等。目前系统中没有使用任何通信设备。

```
[ora10g@localhost dbs]$ ipcs -a

------ Shared Memory Segments --------

key          shmid      owner      perms      bytes      nattch     status



------ Semaphore Arrays --------

key          semid      owner      perms      nsems



------ Message Queues --------

key          msqid      owner      perms      used-bytes           messages
```

然后启动 SQL*Plus（Oracle 的命令行界面），并作为 SYSDBA 连接（SYSDBA 账户可以在数据库中做任何事情）。连接成功后，SQL*Plus 报告称我们连上了一个空闲的实例：

```
[ora10g@localhost dbs]$ sqlplus "/ as sysdba"

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Dec 19 14:09:44 2004

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

Connected to an idle instance.

SQL>
```

我们的“实例”现在只包括一个 Oracle 服务器进程，见以下输出中粗体显示的部分。此时还没有分配共享内存，也没有其他进程。

```
SQL> !ps -aef | grep ora10g

ora10g    4173 4151 0 13:33 pts/0    00:00:00 -su

ora10g    4368 4173 0 14:09 pts/0    00:00:00 sqlplus as sysdba

ora10g    4370 1 0 14:09 ?           00:00:00 oracleora10g (...)

ora10g    4380 4368 0 14:14 pts/0    00:00:00 /bin/bash -c ps -aef | grep ora10g

ora10g    4381 4380 0 14:14 pts/0    00:00:00 ps -aef

ora10g    4382 4380 0 14:14 pts/0    00:00:00 grep ora10g


SQL> !ipcs -a


------ Shared Memory Segments --------

key        shmid      owner      perms      bytes      nattch     status


------ Semaphore Arrays --------

key        semid      owner      perms      nsems


------ Message Queues --------
```

```
key        msqid      owner      perms      used-bytes        messages


SQL>
```

现在来启动实例：

```
SQL> startup

ORA-01078: failure in processing system parameters

LRM-00109: could not open parameter file '/home/ora10g/dbs/initora10g.ora'

SQL>
```

这里提示的文件就是启动实例时必须要有的一个文件，我们需要有一个参数文件（一种简单的平面文件，后面还会详细说明），或者要有一个存储参数文件。现在就来 创建参数文件，并放入启动数据库实例所需的最少信息（通常还会指定更多的参数，如数据库块大小、控制文件位置，等等）。

```
$ cat initora10g.ora

db_name = ora10g
```

然后再回到 SQL*Plus：

```
SQL> startup nomount

ORACLE instance started.
```

这里对 startup 命令加了 nomount 选项，因为我们现在还不想真正“装载”数据库（要了解启动和关闭的所有选项，请参见 SQL*Plus 文档）。

注意 在 Windows 上运行 startup 命令之前，还需要使用 oradim.exe 实用程序执行一条服务创建语句。

现在就有了所谓的“实例”。运行数据库所需的后台进程都有了，如进程监视器（process monitor，PMON）、日志写入器（log writer，LGWR）等，这些进程将在第 5 章详细介绍。

```
Total System Global Area      113246208 bytes

Fixed Size                       777952 bytes

Variable Size                  61874464 bytes

Database Buffers               50331648 bytes

Redo Buffers                     262144 bytes
```

```
SQL> !ps -aef | grep ora10g

ora10g     4173 4151 0 13:33     pts/0 00:00:00 -su

ora10g     4368 4173 0 14:09     pts/0 00:00:00 sqlplus as sysdba

ora10g     4404 1          0 14:18     ?          00:00:00 ora_pmon_ora10g

ora10g     4406 1          0 14:18     ?          00:00:00 ora_mman_ora10g

ora10g     4408 1          0 14:18     ?          00:00:00 ora_dbw0_ora10g

ora10g     4410 1          0 14:18     ?          00:00:00 ora_lgwr_ora10g

ora10g     4412 1          0 14:18     ?          00:00:00 ora_ckpt_ora10g

ora10g     4414 1          0 14:18     ?          00:00:00 ora_smon_ora10g

ora10g     4416 1          0 14:18     ?          00:00:00 ora_reco_ora10g

ora10g     4418 1          0 14:18     ?          00:00:00 oracleora10g (...)

ora10g     4419 4368 0 14:18     pts/0 00:00:00 /bin/bash -c ps -aef | grep ora10g

ora10g     4420 4419 0 14:18     pts/0 00:00:00 ps -aef

ora10g     4421 4419 0 14:18     pts/0 00:00:00 grep ora10g
```

再使用 ipcs 命令，它会首次报告指出使用了共享内存和信号量，这是 UNIX 上的两个重要的进程间通信设备：

```
SQL> !ipcs –a



------ Shared Memory Segments --------

key                shmid        owner           perms        bytes              nattch     status

0x99875060 458760       ora10g          660             115343360       8



------ Semaphore Arrays --------

key                semid        owner           perms        nsems
```

```
0xf182650c    884736     ora10g          660             34



------ Message Queues --------

key             msqid      owner       perms       used-bytes              messages

SQL>
```

注意，我们还没有“数据库”呢！此时，只有数据库之名（在所创建的参数文件中），而没有数据库之实。如果试图“装载”这个数据库，就会失败，因为数据库根本就不存在。下面就来创建数据库。有人说创建一个 Oracle 数据库步骤很繁琐，真是这样吗？我们来看看：

```
SQL> create database;

Database created.
```

这里创建数据库就是这么简单。但在实际中，也许要使用一个稍有些复杂的 CREATE DATABASE 命令，因为可能需要告诉 Oracle 把日志文件、数据文件、控制文件等放在哪里。不过，我们现在已经有了一个完全可操作的数据库了。可能还需要运行 $ORACLE_HOME/rdbms/admin/ catalog.sql 脚本和其他编录脚本（catalog script）来建立我们每天使用的数据字典（这个数据库中还没有我们使用的某些视图，如 ALL_OBJECTS），但不管怎么说，数据库已经有了。可以简单地查询一些 Oracle V$视图（具体就是 V$DATAFILE、V$LOGFILE 和 V$CONTROLFILE），列出构成这个数据库的文件：

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



NAME

--------------------------------------------------------------------------------

/home/ora10g/dbs/dbs1ora10g.dbf

/home/ora10g/dbs/dbx1ora10g.dbf



SQL> select member from v$logfile;



MEMBER
```

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

/home/ora10g/dbs/log1ora10g.dbf

/home/ora10g/dbs/log2ora10g.dbf



SQL> select name from v$controlfile;



NAME

--------------------------------------------------------------------------------

/home/ora10g/dbs/cntrlora10g.dbf



SQL>
```

Oracle 使用默认设置，把所有内容都放在一起，并把数据库创建为一组持久的文件。如果关闭这个数据库，再试图打开，就会发现数据库无法打开：

```
SQL> alter database close;

Database altered.

SQL> alter database open;

alter database open

*

ERROR at line 1:

ORA-16196: database has been previously opened and closed
```

一个实例在其生存期中最多只能装载和打开一个数据库。要想再打开这个（或其他）数据库，必须先丢弃这个实例，并创建一个新的实例。

重申一遍：

- 实例是一组后台进程和共享内存。
- 数据库是磁盘上存储的数据集合。
- 实例“一生”只能装载并打开一个数据库。

- 数据库可以由一个或多个实例（使用 RAC）装载和打开。

前面提到过，大多数情况下，实例和数据库之间存在一种一对一的关系。可能正因如此，才导致人们很容易将二者混淆。从大多数人的经验看来，数据库就是实例，实例就是数据库。

不过，在许多测试环境中，情况并非如此。在我的磁盘上，可以有 5 个不同的数据库。测试主机上任意时间点只会运行一个 Oracle 实例，但是它访问的数据库每天都可能不同（甚至每小时都不同），这取决于我的需求。只需有不同的配置文件，我就能装载并打开其中任意一个数据库。在这种情况下，任何时刻我都只有一个“实例”，但有多个数据库，在任意时间点上只能访问其中的一个数据库。

所以，你现在应该知道，如果有人谈到实例，他指的就是 Oracle 的进程和内存。提到数据库时，则是说保存数据的物理文件。可以从多个实例访问一个数据库，但是一个实例一次只能访问一个数据库。

2.2 SGA 和后台进程

你可能已经想到了 Oracle 实例和数据库的抽象图是个什么样子（见图 2-1）。

图 2-1 以最简单的形式展示了 Oracle 实例和数据库。Oracle 有一个很大的内存块，称为系统全局区（SGA），在这里它会做以下工作：

- 维护所有进程需要访问的多种内部数据结构；
- 缓存磁盘上的数据，另外重做数据写至磁盘之前先在这里缓存；
- 保存已解析的 SQL 计划；
- 等等。

图 2-1 Oracle 实例和数据库

Oracle 有一组“附加到”SGA 的进程，附加机制因操作系统而异。在 UNIX 环境中，

这些进程会物理地附加到一个很大的共享内存段，这是操作系统中分配的一个内存块，可以由多个进程并发地访问（通常要使用 shmget()和 shmat()）。

在 Windows 中，这些进程只是使用 C 调用（malloc()）来分配内存，因为它们实际上是一个大进程中的线程，所以会共享相同的虚拟内存空间。Oracle 还有一组供数据库进程/线程读写的文件（只允许 Oracle 进程读写这些文件）。这些文件保存了所有的表数据、索引、临时空间、重做日志等。

如果在一个 UNIX 系统上启动 Oracle，并执行 ps 命令，会看到运行着许多物理进程，还会显示出这些进程的名字。在前面的例子中，我们已经观察到了 pmon、smon 以及其他一些进程。我会在第 5 章逐一介绍这些进程，现在只要知道它们通称为 Oracle 后台进程（background process）就足够了。这些后台进程是构成实例的持久性进程，从启动实例开始，这些进程会一直运行，直至实例关闭。

有一点要注意，这些都是进程，而不是单个的程序。UNIX 上只有一个 Oracle 二进制可执行文件；根据启动时所提供的选项，这个可执行文件会有多种不同的“个性”。执行 ora_pmon_ora10g 进程要运行这个二进制可执行文件，执行 ora_ckpt_ora10g 进程时运行的可执行文件仍是它。二进制可执行文件只有一个，就是 oracle，只是会以不同的名字执行多次。

在 Windows 上，使用 pslist 工具（http://www.sysinternals.com/ntw2k/freeware/ pslist.shtml）只会看到一个进程 oracle.exe。同样，Windows 上也只有一个二进制可执行文件（oracle.exe）。在这个进程中，可以看到表示 Oracle 后台进程的多个线程。

使用 pslist（或另外的某个工具），可以看到以下线程：

```
C:\Documents and Settings\tkyte>pslist oracle



PsList 1.26 - Process Information Lister

Copyright (C) 1999-2004 Mark Russinovich

Sysinternals - www.sysinternals.com



Process information for ORACLE-N15577HE:



Name Pid      Pri Thd     Hnd       Priv        CPU        Time       Elapsed Time

oracle          1664        8          19          284          354684                   0:0
0:05.          687     0:02:42.218
```

从中可以看出，这个 Oracle 进程里有 19 个线程（以上所示的 Thd 列）。这些线程就对应于 UNIX 上的进程（pmon、arch、lgwr 等 Oracle 进程）。还可以用 pslist 查看各线程的更多详细信息：

```
C:\Documents and Settings\tkyte>pslist -d oracle

PsList 1.26 - Process Information Lister

Copyright (C) 1999-2004 Mark Russinovich

Sysinternals - www.sysinternals.com

Thread detail for ORACLE-N15577HE:

oracle 1664:


Tid       Pri Cswtch       State          User Time     KernelTime    Elapsed Time

1724        9       148   Wait:Executive 0:00:00.000   0:00:00.218   0:02:46.625

 756        9       236   Wait:UserReq   0:00:00.000   0:00:00.046   0:02:45.984

1880        8         2   Wait:UserReq   0:00:00.000   0:00:00.000   0:02:45.953

1488        8       403   Wait:UserReq   0:00:00.000   0:00:00.109   0:02:10.593

1512        8       149   Wait:UserReq   0:00:00.000   0:00:00.046   0:02:09.171

1264        8       254   Wait:UserReq   0:00:00.000   0:00:00.062   0:02:09.140

 960        9       425   Wait:UserReq   0:00:00.000   0:00:00.125   0:02:09.078

2008        9       341   Wait:UserReq   0:00:00.000   0:00:00.093   0:02:09.062

1504        8      1176   Wait:UserReq   0:00:00.046   0:00:00.218   0:02:09.015

1464        8        97   Wait:UserReq   0:00:00.000   0:00:00.031   0:02:09.000

1420        8       171   Wait:UserReq   0:00:00.015   0:00:00.093   0:02:08.984

1588        8       131   Wait:UserReq   0:00:00.000   0:00:00.046   0:02:08.890

1600        8        61   Wait:UserReq   0:00:00.000   0:00:00.046   0:02:08.796

1608        9         5   Wait:Queue     0:00:00.000   0:00:00.000   0:02:01.953

2080        8        84   Wait:UserReq   0:00:00.015   0:00:00.046   0:01:33.468
```

2088	8	127	Wait:UserReq	0:00:00.000	0:00:00.046	0:01:15.968
2092	8	110	Wait:UserReq	0:00:00.000	0:00:00.015	0:01:14.687
2144	8	115	Wait:UserReq	0:00:00.015	0:00:00.171	0:01:12.421
2148	9	803	Wait:UserReq	0:00:00.093	0:00:00.859	0:01:09.718

不同于 UNIX，这里看不到线程的“名字”（UNIX 上则会显示 ora_pmon_ora10g 等进程名），不过，我们可以看到线程 ID（Tid），优先级（Pri）以及有关的其他操作系统审计信息。

2.3 连接 Oracle

这一节将介绍 Oracle 服务器处理请求的两种最常见的方式，并分析它们的基本原理，这两种方式分别是专用服务器（dedicated server）连接和共享服务器（shared server）连接。要想登录数据库并在数据库中真正做事情，必须先连接，我们会说明建立连接时客户端和服务器端会发生什么。最后会简要地介绍如何建立 TCP/IP 连接（TCP/IP 是网络上连接 Oracle 所用的主要网络协议），并说明对于专用服务器连接和共享服务器连接，服务器上的监听器（listener）进程会以不同的方式工作，这些监听器进程负责建立与服务器的物理连接。

2.3.1 专用服务器

从图 2-1 和 pslist 输出可以看出启动 Oracle 之后是什么样子。如果现在使用一个专用服务器登录数据库，则会创建一个新的进程，提供专门的服务：

```
C:\Documents and Settings\tkyte>sqlplus tkyte/tkyte

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Dec 19 15:41:53 2004

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

Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options



tkyte@ORA10G> host pslist oracle

PsList 1.26 - Process Information Lister

Copyright (C) 1999-2004 Mark Russinovich

Sysinternals - www.sysinternals.com
```

```
Process information for ORACLE-N15577HE:


Name      Pid      Pri Thd      Hnd      Priv          CPU Time       Elapsed Time

oracle    1664 8   20   297     356020   0:00:05.906   0:03:21.546

tkyte@ORA10G>
```

现在可以看到，线程有 20 个而不是 19 个，多加的这个线程就是我们的专用服务器进程（稍后会介绍专用服务器进程的更多内容）。注销时，这个额外的线程也没有了。在 UNIX 上，可以看到正在运行的 Oracle 进程列表上会多加一个进程，这就是我们的专用服务器。

再回过来看前面的那个图。现在如果按最常用的配置连接 Oracle，则如图 2-2 所示。

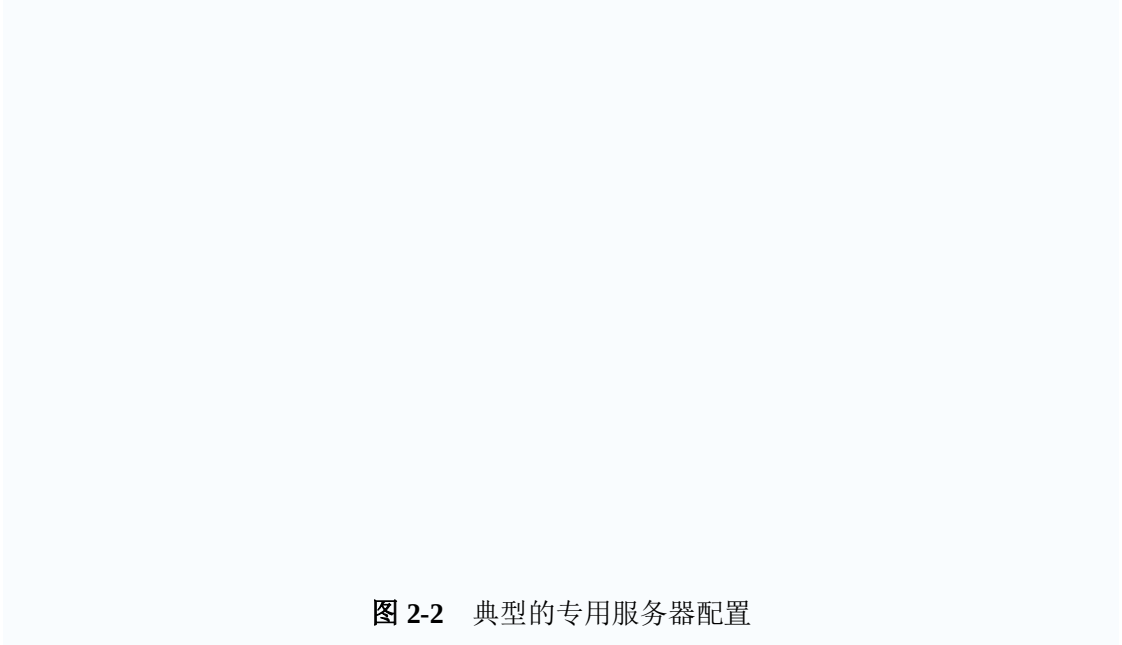

图 2-2 典型的专用服务器配置

如前所述，在我登录时，Oracle 总 会为我创建一个新的进程。这通常称为专用服务器配置，因为这个服务器进程会在我的会话生存期中专门为我服务。对于每个会话，都会出现一个新的专用服务器， 会话与专用服务器之间存在一对一的映射。按照定义，这个专用服务器不是实例的一部分。我的客户进程（也就是想要连接数据库的程序）会通过某种网络通道（如 TCP/IP socket）与这个专用服务器直接通信，并由这个服务器进程接收和执行我的 SQL。如果必要，它会读取数据文件，并在数据库的缓存中查找我要的数据。也许它会完成我的更新语句，也可能会运行我的 PL/SQL 代码。这个服务器进程的主要目标就是对我提交的 SQL 调用做出响应。

2.3.2 共享服务器

Oracle 还可以接受另一种方式的连接，这称为共享服务器（shared server），正式的说法是多线程服务器（Multi-Threaded Server）或 MTS。如果采用这种方式，就不会对每条用户连接创建另外的线程或新的 UNIX 进程。在共享服务器中，Oracle 使用一个“共享进程”池

为大量用户提供服务。共享服务器实际上就是一种连接池机制。利用共享服务器，我们不必为 10,000 个数据库会话创建 10,000 个专用服务器（这样进程或线程就太多了），而只需建立很少的一部分进程/线程，顾名思义，这些进程/线程将由所有会话共享。这样 Oracle 就能让更多的用户与数据库连接，否则很难连接更多用户。如果让我的机器管理 10,000 个进程，这个负载肯定会把它压垮，但是管理 100 个或者 1,000 个进程还是可以的。采用共享服务器模式，共享进程通常与数据库一同启动，使用 ps 命令可以看到这个进程。

共 享服务器连接和专用服务器连接之间有一个重大区别，与数据库连接的客户进程不会与共享服务器直接通信，但专用服务器则不然，客户进程会与专用服务器直接通 信。之所以不能与共享服务器直接对话，原因就在于这个服务器进程是共享的。为了共享这些进程，还需要另外一种机制，通过这种机制才能与服务器进程“对 话”。为此，Oracle 使用了一个或一组称为调度器（dispatcher，也称分派器）的进程。客户进程通过网络与一个调度器进程通信。这个调度器进程将客户的请求放入 SGA 中的请求队列（这也是 SGA 的用途之一）。第一个空闲的共享服务器会得到这个请求，并进行处理（例如，请求可能是 UPDATE T SET X = X+5 WHERE Y = 2）。完成这个命令后，共享服务器会把响应放在原调度器（即接收请求的调度器）的响应队列中。调度器进程一直在监听这个队列，发现有结果后，就会把结果传给客户。从概念上讲，共享服务器请求的流程如图 2-3 所示。

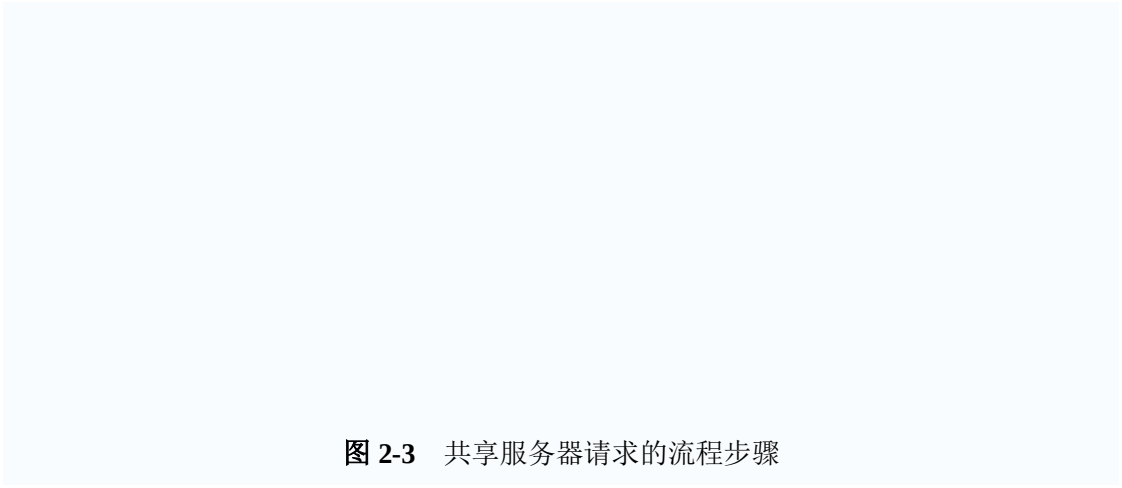

图 2-3 共享服务器请求的流程步骤

如图 2-3 所示，客户连接向调度器发送一个请求。调度器首先将这个请求放在 SGA 中的请求队列中①。第一个可用的共享服务器从请求队列中取出这个请求②并处理。共享服务器的处理结束后，再把响应（返回码、数据等）放到响应队列中③，接下来调度器拿到这个响应④，传回给客户。

在开发人员看来，共享服务器连接和专用服务器连接之间并没有什么区别。

既然已经了解了专用服务器连接和共享服务器连接是什么，你可能会有一些疑问：

- 首先怎么才能连接呢？
- 谁来启动这个专用服务器？
- 怎么与调度器联系？

这些问题的答案取决于你的特定平台，不过下一节会概括介绍一般的过程。

2.3.3 TCP/IP 连接的基本原理

这里将分析网络上最常见的一种情形：在 TCP/IP 连接上建立一个基于网络的连接请求。在这种情况下，客户在一台机器上，而服务器驻留在另一台机器上，这两台机器通过一个 TCP/IP 网络连接。客户率先行动，使用 Oracle 客户软件（Oracle 提供的一组应用程序接口，或 API）建立一个请求，力图连接数据库。例如，客户可以发出以下命令：

```
[tkyte@localhost tkyte]$ sqlplus scott/tiger@ora10g.localdomain



SQL*Plus: Release 10.1.0.3.0 - Production on Sun Dec 19 16:16:41 2004

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

Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options



scott@ORA10G>
```

这里，客户是程序 SQL*Plus，scott/tiger 为用户名/密码，ora10g.localdomain 是一个 TNS 服务名。TNS 代表透明网络底层（Transparent Network Substrate），这是 Oracle 客户中处理远程连接的“基础”软件，有了它才有可能建立对等通信。TNS 连接串告诉 Oracle 软件如何与远程数据库连接。一般地，你的机器上运行的客户软件会读取一个 tnsnames.ora 文件。这是一个纯文本的配置文件，通常放在 [ORACLE_HOME]\network\admin 目录下（[ORACLE_HOME] 表示 Oracle 安装目录的完整路径）。如果有以下配置：

```
ORA10G.LOCALDOMAIN =

    (DESCRIPTION =

        (ADDRESS_LIST =

                (ADDRESS = (PROTOCOL = TCP)(HOST = localhost.localdomain)(PORT = 1521))

        )

    (CONNECT_DATA =

        (SERVICE_NAME = ora10g)
```

)

根据这个配置信息，Oracle 客户软件可以把我们使用的 TNS 连接串 ora10g.localdomain 映 射到某些有用的信息，也就是主机名、该主机上“监听器”进程接受（监听）连接的端口、该主机上所连接数据库的服务名，等等。服务名表示具有公共属性、服务 级阈值和优先级的应用组。提供服务的实例数量对应用是透明的，每个数据库实例可以向监听器注册，表示要提供多个服务。所以，服务就映射到物理的数据库实 例，并允许 DBA 为之关联阈值和优先级。

这个串（ora10g.localdomain）还可以用其他方式来解析。例如，可以使用 Oracle Internet 目录（Oracle Internet Directory，OID），这是一个分布式轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）服务器，其作用就相当于解析主机名的 DNS。不过，tnsnames.ora 文件通常只适用于大多数小到中型安装，在这些情况下，这个配置文件的副本不算太多，尚可管理。

既然客户软件知道要连接到哪里，它会与主机名为 localhost.localdomain 的服务器在端口 1521 上打开一条 TCP/IP socket 连接。如果服务器 DBA 安装并配置了 Oracle Net，并且有一个监听器在端口 1521 上监听连接请求，就会收到这个连接。在网络环境中，我们会在服务器上运行一个称为 TNS 监听器的进程。就是这个监听器进程能让我们与数据库物理连接。当它收到入站连接请求时，它会使用自己的配置文件检查这个请求，可能会拒绝请求（例如，因为没有这样的数据库，或者可能我们的 IP 地址受到限制，不允许连接这个主机），也可能会接受请求，并真正建立连接。

如果建立一条专用服务器连接，监听器进程就会为我们创建一个专用服务器。在 UNIX 上，这是通过 fork()和 exec()系统调用做到的（在 UNIX 中，要在初始化之后创建新进程，惟一的办法就是通过 fork()）。这个新的专用服务器进程继承了监听器建立的连接，现在就与数据库物理地连接上了。在 Windows 上，监听器进程请求数据库进程为连接创建一个新线程。一旦创建了这个线程，客户就会“重定向”到该线程，相应地就能建立物理连接。图 2-4 显示了 UNIX 上的监听器进程和专用服务器连接。

图 2-4 监听器进程和专用服务器连接

另 一方面，如果我们发出共享服务器连接请求，监听器的表现则会有所不同。监听器进程知道实例中运行了哪些调度器。接收到连接请求后，监听器会从可用的调度器 池中选择一个调度器进程。监听器会向客户返回连接信息，其中说明了客户如何与调度器进程连接；如果可能的话，还可以把连接“转发”给调度器进程（这依赖于 不同的操作系统和数据库版本，不过实际效果是一样的）。监听器发回连接信息后，它的工作就结束了，因为监听器

一直在特定主机的特定端口上运行（主机名和端 口号大家都知道），而调度器会在服务器上随意指派的端口上接受连接。监听器要知道调度器指定的这些随机端口号，并为我们选择一个调度器。客户再与监听器断 开连接，并与调度器直接连接。现在就与数据库有了一个物理连接。这个过程如图 2-5 所示。

图 2-5 监听器进程和共享服务器连接

2.4 小结

以上就是 Oracle 体系结构的概述。在这一章中，我们给出了“实例”和“数据库”的定义，并且了解了如何通过专用服务器连接或共享服务器连接来连接数据库。图 2-6 对本章的所有内容做了一个总结，展示了使用共享服务器连接的客户和使用专用服务器连接的客户之间的交互方式。由此还显示出，一个 Oracle 实例可以同时使用这两类连接（实际上，即使配置为使用共享服务器连接，Oracle 数据库也总是支持专用服务器连接）。

图 2-6 连接概述

有了以上的介绍，下面就能更深入地了解服务器底层的进程，这些进程做什么，以及进程间如何交互。你也可以看看 SGA 的内部包含什么，它有什么用途。下一章先来介绍 Oracle

管理数据所用的不同文件类型，并讨论各种文件类型的作用。

第 3 章 文件

这一章中，我们将分析构成数据库和实例的 8 种文件类型。与实例相关的文件只有：

- 参数文件（parameter file）：这些文件告诉 Oracle 实例在哪里可以找到控制文件，并且指定某些初始化参数，这些参数定义了某种内存结构有多大等设置。我们还会介绍存储数据库参数文件的两种选择。
- 跟踪文件（trace file）：这通常是一个服务器进程对某种异常错误条件做出响应时创建的诊断文件。
- 警告文件（alert file）：与跟踪文件类似，但是包含“期望”事件的有关信息，并且通过一个集中式文件（其中包括多个数据库事件）警告 DBA。
- 构成数据库的文件包括：
- 数据文件（data file）：这些文件是数据库的主要文件；其中包括数据表、索引和所有其他的段。
- 临时文件（temp file）：这些文件用于完成基于磁盘的排序和临时存储。
- 控制文件（control file）：这些文件能告诉你数据文件、临时文件和重做日志文件在哪里，还会指出与文件状态有关的其他元数据。

- 重做日志文件（redo log file）：这些就是事务日志。
- 密码文件（password file）：这些文件用于对通过网络完成管理活动的用户进行认证。我们不打算详细讨论这些文件。

从 Oracle 10g 开始，又增加了两种新的可选文件类型，可以帮助 Oracle 实现更快的备份和更快的恢复操作。这两类新文件是：

- 修改跟踪文件（change tracking file）：这个文件有利于对 Oracle 数据建立真正的增量备份。修改跟踪文件不一定非得放在闪回恢复区（Flash Recovery Area），不过它只与数据库备份和恢复有关，所以我们将在介绍闪回恢复区时再讨论这个文件。
- 闪回日志文件（flashback log file）：这些文件存储数据库块的“前映像”，以便完成新增加的 FLASHBACK DATABASE 命令。

我们还会讨论通常与数据库有关的其他类型的文件，如：

- 转储文件（dump file ，DMP file）：这些文件由 Export（导出）数据库实用程序生成，并由 Import（导入）数据库实用程序使用。
- 数据泵文件（Data Pump file）：这些文件由 Oracle 10g 新增的数据泵导出（Data Pump Export）进程生成，并由数据泵导入（Data Pump Import）进程使用。外部表也可以创建和使用这种文件格式。
- 平面文件（flat file）：这些无格式文件可以在文本编辑器中查看。通常会使用这些文件向数据库中加载数据。

以 上文件中，最重要的是数据文件和重做日志文件，因为其中包含了你辛辛苦苦才积累起来的数据。只要有这两个文件，就算是其他文件都没有了，我也能得到我的数 据。如果把重做日志文件弄丢失了，可能会丢失一些数据。如果把数据文件和所有备份都丢失了，那么这些数据就永远也找不回来了。

下面将分别介绍上述各类文件，并分析这些文件中会有哪些内容。

3.1 参数文件

与 Oracle 数据库有关的参数文件有很多，从客户工作站上的 tnsnames.ora 文件（用于“查找”网络上的一个服务器）到服务器上的 listener.ora 文件（用于启动网络监听器），还有 sqlnet.ora、cman.ora 和 ldap.ora 等文件。不过，最重要的参数文件是数据库的参数文件，如果没有这个参数文件，甚至无法启动数据库。其他文件也很重要；它们涉及网络通信以及与数据库连接的各个方面。不过，这些参数文件超出了我们的范围，这里不做讨论。要了解如何配置和建立这些参数文件，建议你参考 Net Services Administrator’s Guide。不过，作为开发人员，这些文件应该已经为你设置好了，不需要你来设置。

数据库的参数文件通常称为初始文件（init file），或 init.ora 文件。这是因为历史上它的默认名就是 init<ORACLE_SID>.ora。之所以称之为“历史上”的默认名，原因是从 Oracle9i Release 1 以来，对于存储数据库的参数设置，引入了一个有很大改进的新方法：服务器参数文件（server parameter file），或简称为 SPFILE。这个文件的默认名为 spfile<ORACLE_SID>.ora。接下来分别介绍这两种参数文件。

注意 如果你还不熟悉术语 SID 或 ORACLE_SID，下面给出一个完整的定义。SID 是站点

标识符（site identifie）。在 UNIX 中，SID 和 ORACLE_HOME（Oracle 软件的安装目录）一同进行散列运算，创建一个惟一的键名从而附加到 SGA。如果 ORACLE_SID 或 ORACLE_HOME 设置不当，就会得到 ORACLE NOT AVAILABLE （ORACLE 不可用）错误，因为无法附加到这个惟一键所标识的共享内存段。在 Windows 上，使用共享内存的方式与 UNIX 中有所不同，不过，SID 还是很重要。同一个 ORACLE_HOME 上可以有多个数据库，所以需要有办法惟一地标识各个数据库及相应的配置文件。

如果没有参数文件，就无法启动一个 Oracle 数据库。所以参数文件相当重要，到了 Oracle9i Release 2（9.2 及以上版本），备份和恢复工具——恢复管理器（Recovery Manager，RMAN）认识到了这个文件的重要性，允许把服务器参数文件包括在备份集中（而不是遗留的 init.ora 参数文件类型）。不过，由于 init.ora 参数文件只是一个纯文本文件，可以用任何文本编辑器创建，所以这个文件不需要你花大力气去“保卫”。只要知道文件中的内容，完全可以重新创建（例如，如果你能访问数据库的警告日志，就可以从中获得参数文件的信息）。

下面依次介绍这两类参数文件（init.ora 和 SPFILE），不过，在此之前，先来看看数据库参数文件是什么样子。

3.1.1 什么是参数？

简单地说，可以把数据库参数想成是一个“键”/“值”对。在上一章你已经看到过一个很重要的参数，即 DB_NAME。这个 DB_NAME 参数简单地存储为 db_name = ora10g。这里的“键”是 DB_NAME，“值 ”是 ora10g，这就是我们的键/值对。要得到一个实例参数的当前值，可以查询 V$视图 V$PARAMETER。另外，还可以在 SQL*Plus 中使用 SHOW PARAMETER 命令来查看，如：

```
sys@ORA10G> select value

  2 from v$parameter

  3 where name = 'pga_aggregate_target';

VALUE

------------------------------------------------------------------------------

1073741824



sys@ORA10G> show parameter pga_agg

NAME                                 TYPE             VALUE

------------------------------------ ---------- ------------------------------

pga_aggregate_target                 big integer      1G
```

无论采用哪种方法，输出的信息基本上都一样，不过从 V$PARAMETER 能得到更多信息（这个例子中只选择了一列，实际上还可以选择更多的列）。但是，我还是比较倾向于使用 SHOW PARAMETER，因为这个命令使用更简单，而且它会自动完成“通配”。注意我只键入了 pga_agg；SHOW PARAMETER 会自动在前面和后面添加%。

注意 所有 V$视图和所有字典视图在 Oracle Database Reference 手册中都有充分的说明。要想了解给定视图里有什么，这个手册可以作为一个权威资源。

对于 Oracle 9.0.1、9.2.0 和 10.1.0 版本，如果对可以设置的有记录的（documented）参数做一个统计，可能会分别得到参数个数为 251、258 和 255（我相信，在不同的操作系统上可能还会增加另外的参数）。换句话说，参数个数（和参数名）因版本而异。大多数参数（如 DB_BLOCK_SIZE）留存已久（它们不会因版本变化而消失），不过，随着时间的推移，其他的很多参数会随着实现的改变而过时。

例如，在 Oracle 9.0.1 中有一个 DISTRIBUTED_TRANSACTIONS 参数，这个参数可以设置为某个正整数，它能控制数据库可以执行的并发分布式事务的个数。以前的版本中都有这个参数，但是在 9.0.1 以后，这个参数就被去掉了。实际上，如果在以后的版本中还想使用这个参数，将产生一个错误：

```
ops$tkyte@ORA10G> alter system set distributed_transactions = 10;

alter system set distributed_transactions = 10

*

ERROR at line 1:

ORA-25138: DISTRIBUTED_TRANSACTIONS initialization parameter has been made

obsolete
```

如果你想查看这些参数，了解有哪些参数，以及各个参数能做什么，请参考 Oracle Database Reference 手册。这个手册的第 1 章就详细地分析了每一个有记录的参数。需要指出，一般来讲，这些参数的默认值对于大多数系统都已经足够了（如果某些参数是从其他参数得到默认设置，则完全可以使用所得到的值）。一般而言，要为各个数据库分别设置不同的参数值，如 CONTROL_FILES 参数（指定系统上控制文件的位置）、DB_BLOCK_SIZE（数据库块大小）以及与内存相关的各个参数。

注意，在上一段中我用了“有记录的”（documented）一词。还有一些无记录的（undocumented）参数。如果参数名用下划线（_）开头，就说明这个参数在文档中未做说明，即所谓的“无记录”。关于这些参数有很多推测。因为文档中没有这些参数，有些人以为它们肯定是“神奇的”，许多人都认为大家都知道这些参数，它们是 Oracle“内 部人士”用的。不过在我看来，实际上恰恰相反。这些参数并不是大家都知道的，而且也很少用到。其中大多数参数实际上令人厌烦，因为它们表示的只是过时的功 能以及为保证向后兼容性而设置的标志。还有一些参数有助于数据的恢复，而不是数据库本身的恢复；例如，有些无记录的参数允许数据库在某些极端环境中启动， 但是时间不长，只足以把数据取出来。取出数据后还是得重新构建。

除非 Oracle Support 明确要求，否则没有理由在你的配置中使用这种无记录的参数。其中很多参数都有副作用，而且可能是破坏性的。在我的开发数据库中，即使有无记录的参数，也只会设置一个这样的参数：

```
_TRACE_FILES_PUBLIC
```

有了这个参数，所有人都可以读取跟踪文件，而不仅限于 DBA 小组。在我的开发数据库上，我希望开发人员经常使用 SQL_TRACE、TIMED_STATISTICS 和 TKPROF 实用程序（真的，我强烈建议使用它们）；所以他们必须能读取跟踪文件。不过，由于 Oracle 9.0.1 及以上版本增加了外部表，可以看到，即便是要允许别人访问跟踪文件，也不再需要使用这个参数了。

我的生产数据库则没有设置任何无记录的参数。实际上，前面提到的看似“安全”的无记录参数可能会在实际系统中产生不好的副作用。想想看跟踪文件中的敏感信息，如 SQL 甚至数据值（见后面的“跟踪文件”一节），问问自己，“我真的想让所有最终用户读取这个数据吗？”大多数情况下答案都是否定的。

警告 只有在 Oracle Support 要求的情况下才使用无记录的参数。使用这些参数可能对数据库有害，而且这些参数在不同版本中的实现可能有变化（而且将会改变）。

可以用两种方式来设置各个参数值：只设置当前实例的参数值，或者永久性地设置。你要确保参数文件包含你期望的值。使用遗留的 init.ora 参数文件时，这是一个手动过程。如果使用 init.ora 文件，要永久地修改一个参数值（即使服务器重启这个新设置也有效），就必须手动地编辑和修改 init.ora 参数文件。如果是服务器参数文件，则只需一条命令就能轻松完成，这多少有些全自动的味道。

3.1.2 遗留的 init.ora 参数文件

遗留的 Oracle init.ora 文件从结构来讲是一个相当简单的文件。这是一系列可变的键/值对。以下是一个 init.ora 文件示例：

```
db_name = "ora9ir2"

db_block_size = 8192

control_files = ("C:\oradata\control01.ctl", "C:\oradata\control02.ctl")
```

实际上，这与你在实际生活中可能遇到的最基本的 init.ora 文件很接近。如果块大小正是平台上的默认块大小（默认块大小随平台不同会有所不同），可以不要 db_block_size 参数。使用这个参数文件只是要得到数据库名和控制文件的位置。控制文件告诉 Oracle 其他的各个文件的位置，所以它们对于启动实例的“自启”过程（也称自举）非常重要。

既然已经知道了这些遗留的数据库参数文件是什么，也知道了在哪里能更详细地了解可设置的有效参数，最后还需要知道这些参数文件在磁盘上的什么位置。这个文件的命名约定默认为：

```
init$ORACLE_SID.ora      (Unix environment variable)

init%ORACLE_SID%.ora          (Windows environment variable)
```

而且，默认地把它放在以下目录中：

```
$ORACLE_HOME/dbs (Unix)

%ORACLE_HOME%\DATABASE (Windows)
```

有意思的是，许多情况下，你会发现这个参数文件中只有一行内容：

```
IFILE='C:\oracle\admin\ora10g\pfile\init.ora'
```

IFILE 指令与 C 中的#include 很类似。它会在当前文件中包含指定文件的内容。前面的指令就会包含一个非默认位置上的 init.ora 文件。

需要注意，参数文件不必放在特定的位置上。启动一个实例时，可以在启动命令上使用 pfile=filename 选项。如果你想在数据库上尝试不同的 init.ora 参数，来看看不同设置带来的影响，这就非常有用。

遗留的参数文件可以利用任何纯文本编辑器来维护。例如，在 UNIX/Linux 上，我会用 vi；在很多版本的 Windows 操作系统上，我会使用记事本；在大型机上，可能会使用 Xedit。重要的是，你要全盘负责这个文件的编辑和维护。Oracle 数据库本身没有命令可以用来维护 init.ora 文件中包含的值。例如，如果使用 init.ora 参数文件，发出 ALTER SYSTEM 命令来改变 SGA 组件的大小时，这并不会作为一个永久修改反映到 init.ora 文件中。如果希望这个修改是永久的，换句话说，如果希望这成为以后数据库重启时的默认值，你就要负责确保可能用于启动数据库的所有 init.ora 参数文件都得到手动地更新。

最后要注意，有意思的是，遗留的参数文件不一定位于数据库服务器上。之所以会引入存储参数（稍后将介绍），原因之一就是为了补救这种情况。试图启动数据库的客户机上必须有遗留的参数文件，这说明，如果你运行一台 UNIX 服务器，但是通过网络使用一台 Windows 台式机上安装的 SQL*Plus 来管理，这台计算机上就需要有数据库参数文件。

我还记得，发现参数文件没有存放在服务器上时我是多么的沮丧！那是几年前的事了，当时推出了一个全新的工具，名叫 SQL*DBA。利用这个工具，可以完成远程操作（具体地讲，可以完成远程管理操作）。从我的服务器（那时运行的是 SunOS），我能远程地连接一个大型机数据库服务器。而且我还能发出“关机”命令。不过，此时我意识到遇到了麻烦，启动实例时，SQL*DBA 会“抱怨”无法找到参数文件。我发现这些参数文件（init.ora 纯文本文件）放在客户机上，而不是在服务器上。SQL*DBA 则是在启动大型机数据库的本地系统上查找参数。我不仅没有这样一个文件，也不知道要在这个文件中放什么内容才能让系统再次启动！我不知道 db_name 或控制文件位置（光是得到这些大型机文件的正确的命名约定都很困难），而且我无法访问大型机系统本身的日志。从那以后，我再也没有犯过同样的错误；这个教训实在太惨痛了。

当 DBA 认识到 init.ora 参数文件必须放在启动数据库的客户机上时，这会导致这些参数文件大面积“繁殖”。每个 DBA 都想从自己的桌面运行管理工具，所以每个 DBA 都需要在自己的台式机上留有参数文件的一个副本。Oracle 企业管理器（Oracle Enterprise Manager，OEM）之类的工具还会再增加一个参数文件，这会使情况更加混乱。这些工具试图将所有数据库的管理都集中到一台机器上，有时称之为“管理服务器”（management server）。这台机器会运行一个软件，所有 DBA 均使用这个软件来启动、关闭、备份和管理数据库。听上去是一个很不错的解决方案：把所有参数文件都集中在一个位置上，并使用 GUI 工具来完成所有操作。但事实是，完成某个管理任务时，有时直接从数据库服务器主机上的 SQL*Plus

发出启动命令会方便得多，这样又会有多个参数文件：一个参数文件在管理服务器上，另一个参数文件在数据库服务器上。而且这些参数文件彼此不同步，人们可能会奇怪，为什么他们上个月做的参数修改“不见了”，不过这些修改有可能会随机地再次出现。

所以引入了服务器参数文件（server parameter file，SPFILE），如今这可以作为得到数据库参数设置的惟一“信息来源”。

3.1.3 服务器参数文件

在访问和维护实例参数设置方面，SPFILE 是 Oracle 做出的一个重要改变。有了 SPFILE，可以消除传统参数文件存在的两个严重问题：

- 可以杜绝参数文件的繁殖。SPFILE 总是存储在数据库服务器上；必须存在于服务器主机本身，不能放在客户机上。对参数设置来说，这样就可以只有一个“信息来源”。
- 无需在数据库之外使用文本编辑器手动地维护参数文件（实际上，更确切的说法是不能手动地维护）。利用 ALTER SYSTEM 命令，完全可以直接将值写入 SPFILE。管理员不必再手动地查找和维护所有参数文件。

这个文件的命名约定默认为：

```
spfile$ORACLE_SID.ora (Unix environment variable)

spfile%ORACLE_SID%.ora (Windows environment variable)
```

我强烈建议使用默认位置；否则会影响 SPFILE 的简单性。如果 SPFILE 在其默认位置，几乎一切都会为你做好。如果将 SPFILE 移到一个非默认的位置，你就必须告诉 Oracle 到哪里去找 SPFILE，这又会导致遗留参数文件的一大堆问题卷土重来！

1. 转换为 SPFILE

假设有一个数据库，它使用了前面所述的遗留参数文件。转换为 SPFILE 非常简单；这里使用了 CREATE SPFILE 命令。

注意 还可以使用一个“逆”命令从 SPFILE 创建参数文件（parameter file，PFILE），稍后我们会解释为什么希望这样做。

所以，假设使用一个 init.ora 参数文件，而且这个 init.ora 参数文件确实在服务器的默认位置上，那么只需发出 CREATE SPFILE 命令，并重启服务器实例就行了：

```
sys@ORA10G> show parameter spfile;



NAME                                 TYPE        VALUE

------------------------------------ ----------- ------------------------------

spfile                               string
```

```
sys@ORA10G> create spfile from pfile;

File created.



sys@ORA10G> startup force;

ORACLE instance started.



Total System Global Area 603979776 bytes

Fixed Size 780300 bytes

Variable Size 166729716 bytes

Database Buffers 436207616 bytes

Redo Buffers 262144 bytes

Database mounted.

Database opened.

sys@ORA10G> show parameter spfile;



NAME                                 TYPE        VALUE

------------------------------------ ----------- ------------------------------

spfile                               string      /home/ora10g/dbs/spfileora10g.ora
```

这里使用 SHOW PARAMETER 命令显示出原先没有使用 SPFILE，但是创建 SPFILE 并重启实例后，确实使用了这个 SPFILE，而且它采用了默认名。

注意 在集群环境中，通过使用 Oracle RAC，所有实例共享同一个 SPFILE，所以要以一种受控的方式完成这个转换过程（从 PFILE 转换为 SPFILE）。这个 SPFILE 可以包含所有参数设置，甚至各个实例特有的设置都可以放在这一个 SPFILE 中 ，但是必须把所有必须的参数文件合并为一个有以下格式的 PFILE。

在集群环境中，为了从使用各个 PFILE 转换为所有实例都共享一个公共的 SPFILE，需要把各个 PFILE 合并为如下一个文件：

```
*.cluster_database_instances=2

*.cluster_database=TRUE

*.cluster_interconnects='10.10.10.0'

*.compatible='10.1.0.2.0'

*.control_files='/ocfs/O10G/control01.ctl','/ocfs/O10G/control02.ctl'

*.db_name='O10G'

...

*.processes=150

*.undo_management='AUTO'

O10G1.instance_number=1

O10G2.instance_number=2

O10G1.local_listener='LISTENER_O10G1'

O10G2.local_listener='LISTENER_O10G2'

O10G1.remote_listener='LISTENER_O10G2'

O10G2.remote_listener='LISTENER_O10G1'

O10G1.thread=1

O10G2.thread=2

O10G1.undo_tablespace='UNDOTBS1'

O10G2.undo_tablespace='UNDOTBS2'
```

也就是说，集群中所有实例共享的参数设置都以*.开头。单个实例特有的参数设置（如INSTANCE_NUMBER 和所用的重做 THREAD）都以实例名（Oracle SID）为前缀。在前面的例子中，

- PFILE 对应包含两个节点的集群，其中的实例分别为 O10G1 和 O10G2。
- *.db_name = 'O10G'这个赋值指示，使用这个 SPFILE 的所有实例会装载一个名为 O10G 的数据库。
- O10G1.undo_tablespace='UNDOTBS1'指示，名为 O10G1 的实例会使用这个特定

的撤销（undo）表空间，等等。

2. 设置 SPFILE 中的参数值

一旦根据 SPFILE 启动并运行数据库，下一个问题就是如何设置和修改其中的值。要记住，SPFILE 是二进制文件，它们可不能用文本编辑器来编辑。这个问题的答案就是使用 ALTER SYSTEM 命令，语法如下（< > 中的部分是可选的，其中的管道符号（|）表示“取候选列表中的一个选项”）：

```
Alter system set parameter=value <comment='text'> <deferred>

                    <scope=memory|spfile|both> <sid='sid|*'>
```

默认情况下，ALTER SYSTEM SET 命令会更新当前运行的实例，并且会为你修改 SPFILE，这就大大简化了管理；原先使用 init.ora 参数文件时，通过 ALTER SYSTEM 命令设置参数后，如果忘记更新 init.ora 参数文件，或者把 init.ora 参数文件丢失了，就会产生问题，使用 SPFILE 则会消除这些问题。

记住这一点，下面来详细分析这个命令中的各个元素：

- parameter=value 这个赋值提供了参数名以及参数的新值。例如，pga_aggregate_target = 1024m 会把 PGA_AGGREGATE_TARGET 参数值设置为 1,024 MB（1 GB）。
- comment='text'是一个与此参数设置相关的可选注释。这个注释会出现在 V$PARAMETER 视图的 UPDATE_COMMENT 字段中。如果使用了相应选项允许同时保存对 SPFILE 的修改，注释会写入 SPFILE，而且即便服务器重启也依然保留，所以将来重启数据库时会看到这个注释。
- deferred 指定系统修改是否只对以后的会话生效（对当前建立的会话无效，包括执行此修改的会话）。默认情况下，ALTER SYSTEM 命令会立即生效，但是有些参数不能“立即”修改，只能为新建立的会话修改这些参数。可以使用以下查询来看看哪些参数要求必须使用 deferred：

```
ops$tkyte@ORA10G> select name

  2 from v$parameter

  3 where ISSYS_MODIFIABLE = 'DEFERRED';



NAME

------------------------------

backup_tape_io_slaves

audit_file_dest
```

```
object_cache_optimal_size

object_cache_max_size_percent

sort_area_size

sort_area_retained_size

olap_page_pool_size



7 rows selected.
```

上面的代码表明，SORT_AREA_SIZE 可以在系统级修改，但是必须以延迟方式修改。以下代码显示了有 deferred 选项和没有 deferred 选项时修改这个参数的值会有什么结果：

```
ops$tkyte@ORA10G> alter system set sort_area_size = 65536;

alter system set sort_area_size = 65536

*

ERROR at line 1:

ORA-02096: specified initialization parameter is not modifiable with this

option



ops$tkyte@ORA10G> alter system set sort_area_size = 65536 deferred;

System altered.
```

- SCOPE=MEMORY|SPFILE|BOTH 指示了这个参数设置的“作用域”。设置参数值时作用域有以下选择：
  - SCOPE=MEMORY 只在实例中修改；数据库重启后将不再保存。下一次重启数据库时，设置还是修改前的样子。
  - SCOPE=SPFILE 只修改 SPFILE 中的值。数据库重启并再次处理 SPFILE 之前，这个修改不会生效。有些参数只能使用这个选项来修改，例如，processes 参数就必须使用 SCOPE=SPFILE，因为我们无法修改活动实例的 processes 值。
  - SCOPE=BOTH 是指，内存和 SPFILE 中都会完成参数修改。这个修改将反映在当前实例中，下一次重启时，这个修改也会生效。这是使用 SPFILE 时默认的作用域值。如果使用 init.ora 参数文件，默认值则为 SCOPE=MEMORY，

这也是此时惟一合法的值。

- sid='sid|*'主要用于集群环境；默认值为 sid='*'。这样可以为集群中任何给定的实例惟一地指定参数设置。除非你使用 Oracle RAC，否则一般不需要指定 sid=设置。

这个命令的典型用法很简单：

```
ops$tkyte@ORA10G> alter system set pga_aggregate_target=1024m;

System altered.
```

或者，更好的做法是，还可以指定 COMMENT=赋值来说明何时以及为什么执行某个修改：

```
ops$tkyte@ORA10G> alter system set pga_aggregate_target=1024m

  2 comment = 'changed 01-jan-2005 as per recommendation of George';



System altered.



ops$tkyte@ORA10G> select value, update_comment

  2 from v$parameter

  3 where name = 'pga_aggregate_target';



VALUE

--------------------------------------------------------------------------------

UPDATE_COMMENT

--------------------------------------------------------------------------------

1073741824

changed 01-jan-2005 as per recommendation of George
```

3. 取消 SPFILE 中的值设置

下一个问题又来了，“好吧，这样就设置了一个值，但是现在我们又想‘取消这个设置’，换句话说，我们根本不希望 SPFILE 有这个参数设置，想把它删掉。但是既然不能使用文本编辑器来编辑这个文件，那我们该怎么办呢？”同样地，这也要通过 ALTER SYSTEM 命令

来完成，但是要使用 RESET 子句：

```
Alter system reset parameter <scope=memory|spfile|both> sid='sid|*'
```

在这里，SCOPE/SID 设置的含义与前面一样，但是 SID=部分不再是可选的。Oracle SQL Reference 手册在介绍这个命令时有点误导，因为从手册来看，好像这只对 RAC（集群）数据库有效。实际上，手册中有下面的说明：

alter_system_reset_clause（ALTER SYSTEM 命令的 RESET 子句）用于“真正应用集群”（RAC）环境。

接下来，它又说：

在非 RAC 环境中，可以为这个子句指定 SID='*'。

这就有点让人糊涂了。不过，要从 SPFILE“删除”参数设置，也就是仍然采用参数原来的默认值，就要使用这个命令。所以，举例来说，如果我们想删除 SORT_AREA_SIZE，以允许使用此前指定的默认值，可以这样做：

```
sys@ORA10G> alter system reset sort_area_size scope=spfile sid='*';

System altered.
```

这样会从 SPFILE 中删除 SORT_AREA_SIZE，通过执行以下命令可以验证这一点：

```
sys@ORA10G> create pfile='/tmp/pfile.tst' from spfile;

File created.
```

然后可以查看/tmp/pfile.tst 的内容，这个文件将在数据库服务器上生成。可以看到，参数文件中不再有 SORT_AREA_SIZE 参数了。

4. 从 SPFILE 创建 PFILE

上一节用到的 CREATE PFILE...FROM SPFILE 命令刚好与 CREATE SPFILE 相反。这个命令根据二进制 SPFILE 创建一个纯文本文件，生成的这个文件可以在任何文本编辑器中编辑，并且以后可以用来启动数据库。正常情况下，使用这个命令至少有两个原因：

- 创建一个“一次性的”参数文件，用于启动数据库来完成维护，其中有一些特殊的设置。所以，可以执行 CREATE PFILE...FROM SPFILE 命令，并编辑得到的文本 PFILE，修改所需的设置。然后启动数据库，使用 PFILE=<FILENAME>选项指定要使用这个 PFILE 而不是 SPFILE。完成后，可以正常地启动，数据库又会使用 SPFILE。
- 维护修改历史，在注释中记录修改。过去，许多 DBA 会在参数文件中加大量的注释，来记录修改历史。例如，如果一年内把缓冲区缓存大小修改过 20 次，在 db_cache_size init.ora 参数设置前就会有 20 条注释，这些注释会指出修改的日期，以及修改的原因。SPFILE 不支持这样做，但是如果习惯了以下用法，也可以达到同样的效果：

```
sys@ORA10G> create pfile='init_01_jan_2005_ora10g.ora' from spfile;
```

```
File created.



sys@ORA10G> !ls -l $ORACLE_HOME/dbs/init_*

-rw-rw-r-- 1 ora10g ora10g 871 Jan 1 17:04 init_01_jan_2005_ora10g.ora

sys@ORA10G> alter system set pga_aggregate_target=1024m

2 comment = 'changed 01-jan-2005 as per recommendation of George';
```

通过这种方式，修改历史就会在一系列参数文件中长久保存。

5. 修正被破坏的 SPFILE

关于 SPFILE 还有最后一个问题，“SPFILE 是二进制文件，如果 SPFILE 被破坏了，数据库无法启动，那该怎么办？还是 init.ora 文件更好一些，至少它是文本文件，我们可以直接编辑和修正”。嗯，这么说吧，SPFILE 不会像数据文件、重做日志文件、控制文件等那样被破坏，但是，倘若真的发生了这种情况，还是有几种选择的。

首先，SPFILE 中的二进制数据量很小。如果在 UNIX 平台上，只需一个简单的 strings 命令就能提取出所有设置：

```
[tkyte@localhost dbs]$ strings spfile$ORACLE_SID.ora

*.compatible='10.1.0.2.0'

*.control_files='/home/ora10g/oradata/ora10g/control01.ctl','/home/ora10g/oradata/or

a10g/control02.ctl','/home/ora10g/oradata/ora10g/control03.ctl'

...

*.user_dump_dest='/home/ora10g/admin/ora10g/udump'
```

在 Windows 上，则要用 write.exe（WordPad，即写字板）打开这个文件。WordPad 会显示出文件中的所有文本，只需将其剪切并粘贴到 init<ORACLE_SID>.ora 中，就能创建启动实例的 PFILE。

万一 SPFILE 丢失了（不论是什么原因，反正我没有见过 SPFILE 消失的情况），还可以从数据库的警告日志恢复参数文件的信息（稍后将介绍警告日志的更多内容）。每次启动数据库时，警告日志都会包含如下一部分内容：

```
System parameters with non-default values:

processes = 150

timed_statistics = TRUE
```

```
shared_pool_size = 67108864

large_pool_size = 8388608

java_pool_size = 33554432

control_files = C:\oracle\oradata\ora9ir2w\CONTROL01.CTL,

C:\oracle\oradata\ora9ir2w\CONTROL02.CTL,

C:\oracle\oradata\ora9ir2w\CONTROL03.CTL

....

pga_aggregate_target = 25165824

aq_tm_processes = 1

PMON started with pid=2

DBW0 started with pid=3
```

通过这一部分内容，可以很容易地创建一个 PFILE，再用 CREATE SPFILE 命令将其转换为一个新的 SPFILE。

3.1.4 参数文件小结

在这一节中，我介绍了管理 Oracle 初始化参数和参数文件的所有基础知识。我们了解了如何设置参数、查看参数值，以及如何让这些设置在数据库重启时依然保留。我们分析了两类数据库参数文件：传统的 PFILE（简单的文本文件）和 SPFILE（服务器参数文件）。对于所有现有的数据库，都推荐使用 SPFILE，因为这更易于管理，而且也更为简洁。由于数据库的参数只有一个“信息来源”，而且可以使用 ALTER SYSTEM 命令持久地保存参数值，这使得 SPFILE 相当引人注目。自从有了 SPFILE，我就一直在使用 SPFILE，而且从来没有想过再回头去使用 PFILE。

3.2 跟踪文件

跟踪文件（Trace file）能提供调试信息。服务器遇到问题时，它会生成一个包含大量诊断信息的跟踪文件。如果开发人员设置了 SQL_TRACE=TRUE，服务器就会生成一个包含性能相关信息的跟踪文件。我们之所以可以使用这些跟踪文件，是因为 Oracle 是一个允许充分测量的软件。我所说的“可测量”（instrumented）是指，编写数据库内核的程序员在内核中放入了调试代码，而且调试代码相当多。这些调试代码仍然被程序员有意留在内核中。

我见过许多开发人员都认为调试代码会带来开销，认为系统投入生产阶段之前必须把这些调试代码去掉，希望从代码中“挤出”点滴的性能。当然，随后他们可能又会发现代码中有一个“bug”，或者“运行得没有应有的那么快”（最终用户也把这称为“bug”。对于最终用户来说，性能差就是 bug！）。此时，他们多么希望调试代码还在原处未被删掉（或者，

如果原来没有加过调试代码，他们可能很后悔当初为什么没有添加）。特别是，他们无法再向生产系统中添加调试代码，在生产环境中，新代码必须先经过测试，这可不是说添加就添加那么轻松。

Oracle 数据库（以及应用服务器和 Oracle 应用）都是可以充分测量的。数据库中这种测量性反映在以下几方面：

- V$ 视图：大多数 V$ 视图都包含“调试”信息。V$WAITSTAT、V$SESSION_EVENT 还有其他许多 V$视图之所以存在，就是为了让我们知道内核内部到底发生了什么。
- 审计命令：利用这个命令，你能指定数据库要记录哪些事件以便日后分析。
- 资源管理器（DBMS_RESOURCE_MANAGER）：这个特性允许你对数据库中的资源（CPU、I/O 等）实现微管理。正是因为数据库能访问描述资源使用情况的所有运行时统计信息，所以才可能有资源管理器。
- Oracle“事件”：基于 Oracle 事件，能让 Oracle 生成所需的跟踪或诊断信息。
- DBMS_TRACE：这是 PL/SQL 引擎中的一个工具，它会全面地记录存储过程的调用树、所产生的异常，以及遇到的错误。
- 数据库事件触发器：这些触发器（如 ON SERVERERROR）允许你监控和记录你觉得“意外”或非正常的情况。例如，可以记录发生“临时空间用尽”错误时正在运行的 SQL。
- SQL_TRACE：这个 SQL 跟踪工具还可以采用一种扩展方式使用，即通过 10046 Oracle 事件。

还不止这些。在应用设计和开发中，测量至关重要，每一版 Oracle 数据库的测量性都越来越好。实际上，Oracle9i Release 2 和 Oracle 10g Release 1 这两个版本之间增加的测量代码量就相当显著。Oracle 10g 将内核中的代码测量发展到一个全新的层次。

在这一节中，我们将重点讨论各种跟踪文件中的信息。这里会分析有哪些跟踪文件，这些跟踪文件存放在哪里，以及对这些跟踪文件能做些什么。

通常有两类跟踪文件，对这两类跟踪文件的处理完全不同：

- 你想要的跟踪文件：例如，启用 SQL_TRACE=TRUE 选项的结果，其中包含有关会话的诊断信息，有助于你调整应用，优化应用的性能，并诊断出遭遇的瓶颈。
- 你不想要的跟踪文件，但是由于出现了以下错误，服务器会自动生成这些跟踪文件。这些错误包括 ORA-00600“Internal Error”（内部错误）、ORA-03113“End of file on communication channel”（通信通道上文件结束）或 ORA-07445“Exception Encountered”（遇到异常）。这些跟踪文件包含一些诊断信息，它们主要对 Oracle Support 的分析人员有用，但对我们来说，除了能看出应用中哪里出现了内部错误之外，用处不大。

3.2.1 请求的跟踪文件

你想要的跟踪文件通常都是因为设置了 SQL_TRACE=TRUE 生成的结果，或者是通过

10046 事件使用扩展的跟踪工具生成的，如下所示：

```
ops$tkyte@ORA10G> alter session set events

  2 '10046 trace name context forever, level 12';

Session altered.
```

1. 文件位置

不论是使用 SQL_TRACE 还是扩展的跟踪工具，Oracle 都会在数据库服务器主机的以下两个位置生成一个跟踪文件：

- 如果使用专用服务器连接，会在 USER_DUMP_DEST 参数指定的目录中生成跟踪文件。
- 如果使用共享服务器连接，则在 BACKGROUND_DUMP_DEST 参数指定的目录中生成跟踪文件。

要想知道跟踪文件放在哪里，可以从 SQL*Plus 执行 SHOW PARAMETER DUMP_DEST 命令来查看，也可以直接查询 V$PARAMETER 视图：

```
ops$tkyte@ORA10G> select name, value

  2 from v$parameter

  3 where name like '%dump_dest%'

  4 /



NAME                           VALUE

------------------------------ ------------------------------

background_dump_dest           /home/ora10g/admin/ora10g/bdump

user_dump_dest                 /home/ora10g/admin/ora10g/udump

core_dump_dest                 /home/ora10g/admin/ora10g/cdump
```

这里显示了 3 个转储（跟踪）目标。后台转储（background dump）目标由所有“服务器”进程使用（第 5 章会全面介绍 Oracle 后台进程及其作用）。

如果使用 Oracle 的共享服务器连接，就会使用一个后台进程；因此，跟踪文件的位置由 BACKGROUND_DUMP_DEST 确定。如果使用的是专用服务器连接，则会使用一个用户或前台进程与 Oracle 交互；所以跟踪文件会放在 USER_DUMP_DEST 参数指定的目录中。如果出现严重的 Oracle 内部错误（如 UNIX 上的“segmentation fault”错误），或者如果 Oracle Support 要求你生成一个跟踪文件来得到额外的调试信息，CORE_DUMP_DEST 参数则定义

了此时这个“内核”文件应该放在哪里。一般而言，我们只对后台和用户转储目标感兴趣。需要说明，除非特别指出，这本书里都使用专用服务器连接。

如果你无法访问 V$PARAMETER 视图，那么可以使用 DBMS_UTILITY 来访问大多数（但不是全部）参数的值。从下面的例子可以看出，要看到这个信息（还不止这些），只需要 CREATE SESSION 权限：

```
ops$tkyte@ORA10G> create user least_privs identified by least_privs;

User created.



ops$tkyte@ORA10G> grant create session to least_privs;

Grant succeeded.



ops$tkyte@ORA10G> connect least_privs/least_privs

Connected.

least_privs@ORA10G> declare

  2          l_string varchar2(255);

  3          l_dummy number;

  4 begin

  5          l_dummy := dbms_utility.get_parameter_value

  6          ( 'background_dump_dest', l_dummy, l_string );

  7          dbms_output.put_line( 'background: ' || l_string );

  8          l_dummy := dbms_utility.get_parameter_value

  9          ( 'user_dump_dest', l_dummy, l_string );

 10         dbms_output.put_line( 'user: ' || l_string );

 11 end;

 12 /
```

```
background: /home/ora10g/admin/ora10g/bdump

user: /home/ora10g/admin/ora10g/udump



PL/SQL procedure successfully completed.
```

2. 命名约定

Oracle 中跟踪文件的命名约定总在变化，不过，如果把你的系统上的跟踪文件名作为示例，应该能很容易地看出这些命名有一个模板。例如，在我的各台服务器上，跟踪文件名如表 3-1 所示。

表 3-1 跟踪文件名示例

跟踪文件名	平 台	数据库版本
ora10g_ora_24574.trc	Linux	10g Release 1
ora9ir2_ora_24628.trc	Linux	9i Release 2
ora_10583.trc	Linux	9i Release 1
ora9ir2w_ora_688.trc	Windows	9i Release 2
ora10g_ora_1256.trc	Windows	10g Release 1

在我的服务器上，跟踪文件名可以分为以下几部分：

- 文件名的第一部分是 ORACLE_SID（但 Oracle9i Release 1 例外，在这一版本中，Oracle 决定去掉这一部分）。
- 文件名的下一部分只有一个 ora。
- 跟踪文件名中的数字是专用服务器的进程 ID，可以从 V$PROCESS 视图得到。
- 因此，在实际中（假设使用专用服务器模式），需要访问 4 个视图：
- V$PARAMETER：找到 USER_DUMP_DEST 指定的跟踪文件位置。
- V$PROCESS：查找进程 ID。
- V$SESSION：正确地标识其他视图中的会话信息。
- V$INSTANCE：得到 ORACLE_SID。

前面提到过，可以使用 DBMS_UTILITY 来找到位置，而且通常你“知道”ORACLE_SID，所以从理论上讲只需要访问 V$SESSION 和 V$PROCESS，但是，为了便于使用，这 4 个视图你可能都想访问。

以下查询可以生成跟踪文件名：

```
ops$tkyte@ORA10G> alter session set sql_trace=true;

Session altered.

ops$tkyte@ORA10G> select c.value || '/' || d.instance_name ||
  2 '_ora_' || a.spid || '.trc' trace
  3 from v$process a, v$session b, v$parameter c, v$instance d
  4 where a.addr = b.paddr
  5 and b.audsid = userenv('sessionid')
  6 and c.name = 'user_dump_dest'
  7 /
TRACE
------------------------------------------------------------------------------
/home/ora10g/admin/ora10g/udump/ora10g_ora_24667.trc


ops$tkyte@ORA10G>
```

显然，在 Windows 平台上要把 / 换成 \。如果使用 9i Release 1，只需发出以下查询，不用在跟踪文件名中增加实例名：

```
select c.value || 'ora_' || a.spid || '.trc'
```

3. 对跟踪文件加标记

有一种办法可以对跟踪文件“加标记”，这样即使你无权访问 V$PROCESS 和 V$SESSION，也能找到跟踪文件。假设你能读取 USER_DUMP_DEST 目录，就可以使用会话参数 TRACEFILE_IDENTIFIER。采用这种方法，可以为跟踪文件名增加一个可以惟一标识的串，例如：

```
ops$tkyte@ORA10G> alter session set tracefile_identifier = 'Look_For_Me';

Session altered.


ops$tkyte@ORA10G> alter session set sql_trace=true;
```

```
Session altered.



ops$tkyte@ORA10G> !ls /home/ora10g/admin/ora10g/udump/*Look_For_Me*

/home/ora10g/admin/ora10g/udump/ora10g_ora_24676_Look_For_Me.trc



ops$tkyte@ORA10G>
```

可以看到，跟踪文件还是采用标准的<ORACLE_SID>_ora_<PROCESS_ID>格式命名，但是这里还有我们为它指定的一个惟一的串，这样就能很容易地找到“我们的”跟踪文件名。

3.2.2 针对内部错误生成的跟踪文件

这一节最后我再来谈谈另一类跟踪文件，这些跟踪文件不是我们想要的，只是由于ORA-00600 或另外某个内部错误而自动生成。对这些跟踪文件我们能做些什么吗？

答案很简单，一般来讲，这些跟踪文件不是给你我用的。它们只对 Oracle Support 有用。不过，我们向 Oracle Support 提交 iTAR 时，这些跟踪文件会很有用。有一点很重要：如果得到内部错误，修改这个错误的惟一办法就是提交一个 iTAR。如果你只是将错误忽略，除非出现意外，否则它们不会自行修正。

例如，在 Oracle 10g Release 1 中，如果创建下表，并运行以下查询，就会得到一个内部错误（也可能不会得到错误，因为这个错误已经作为一个 bug 提交，并在后来的补丁版本中得到修正）：

```
ops$tkyte@ORA10G> create table t ( x int primary key );

Table created.



ops$tkyte@ORA10G> insert into t values ( 1 );

1 row created.



ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T' );

PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10G> select count(x) over ()

  2 from t;

from t

*

ERROR at line 2:

ORA-00600: internal error code, arguments: [12410], [], [], [], [], [], [], []
```

如果你是一名 DBA，会发现用户转储目标中突然冒出这个跟踪文件。 或者，如果你是一名开发人员，你的应用将产生一个 ORA-00600 错误，你肯定想知道到底发生了什么。跟踪文件中信息很多（实际上，另外还有 35,000 行），但一般来讲，这些信息对你我来说都没有用。我们只是想压缩这个跟踪文件，并将其上传来完成 iTAR 处理。

不过，确实有些信息能帮助你跟踪到“谁”造成了错误，错误是“什么”，以及错误在“哪里”，另外，利用 http://metalink.oracle.com，你还能发现这些问题是不是别人已经遇到过（许多次），以及为什么会出现这些错误。快速检查一下跟踪文件的最前面，你会得到一些有用的信息，如：

```
Dump file c:\oracle\admin\ora10g\udump\ora10g_ora_1256.trc

Sun Jan 02 14:21:29 2005

ORACLE V10.1.0.3.0 - Production vsnsta=0

vsnsql=13 vsnxtr=3

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options

Windows XP Version V5.1 Service Pack 2

CPU :                    1 - type 586

Process Affinity:        0x00000000

Memory (A/P) :           PH:11M/255M, PG:295M/1002M, VA:1605M/2047M

Instance name:           ora10g

Redo thread mounted by this instance:     1

Oracle process number: 21
```

```
Windows thread id: 1256, image:      ORACLE.EXE (SHAD)
```

你在 http://metalink.oracle.com 上提交 iTAR 时，数据库信息当然很重要，不仅如此，如果在 http://metalink.oracle.com 上查看是否以前已经提出过这个问题，这些数据库信息也很有用。另外，可以看出错误出现在哪个 Oracle 实例上。并发地运行多个实例是很常见的，所以把问题隔离到一个实例上会很有用。

```
*** 2005-01-02 14:21:29.062

*** ACTION NAME:() 2005-01-02 14:21:28.999

*** MODULE NAME:(SQL*Plus) 2005-01-02 14:21:28.999

*** SERVICE NAME:(SYS$USERS) 2005-01-02 14:21:28.999
```

跟踪文件中的这一部分是 Oracle 10g 新增的，Oracle9i 里没有。它显示了 V$SESSION 的 ACTION 和 MODULE 列中的会话信息。这里可以看到，是一个 SQL*Plus 会话导致了错误（开发人员应该设置 ACTION 和 MODULE 信息；有些环境已经为你做了这项工作，如 Oracle Forms 和 HTML DB）。

另外还可以得到 SERVICE NAME。这就是连接数据库所用的服务名（这里就是 SYS$USERS），由此看出没有通过 TNS 服务来连接。如果使用 user/pass@ora10g.localdomain 登录，可以看到：

```
*** SERVICE NAME:(ora10g) 2005-01-02 15:15:59.041
```

其中 ora10g 是服务名（而不是 TNS 连接串；这是所连接 TNS 监听器中注册的最终服务）。这对于跟踪哪个进程/模块受此错误影响很有用。

最后，在查看具体的错误之前，可以看到会话 ID 和相关的日期/时间等进一步的标识信息（所有版本都提供了这些信息）：

```
*** SESSION ID:(146.2) 2005-01-02 14:21:28.999
```

现在可以深入到内部看看错误本身了：

```
ksedmp: internal or fatal error

ORA-00600: internal error code, arguments: [12410], [], [], [], [], [], [], []

Current SQL statement for this session:

select count(x) over ()

        from t

----- Call Stack Trace -----

_ksedmp+524
```

```
_ksfdmp.160+14

_kgeriv+139

_kgesiv+78

_ksesic0+59

_qerixAllocate+4155

_qknRwsAllocateTree+281

_qknRwsAllocateTree+252

_qknRwsAllocateTree+252

_qknRwsAllocateTree+252

_qknDoRwsAllocate+9

...
```

这里也有一些重要的信息。首先，可以看到产生内部错误时正在执行的 SQL 语句，这有助于跟踪哪个（哪些）应用会受到影响。同时，由于这里能看到 SQL，所以可以研究采用哪些“迂回路线”，用不同的方法编写 SQL，看看能不能很快绕过问题解决 bug。另外，也可以把出问题的 SQL 剪切并粘贴到 SQL*Plus 中，看看能不能为 Oracle Support 提供一个可再生的测试用例（当然，这些是最棒的测试用例）。

另一个重要信息是错误码（通常是 600、3113 或 7445）以及与错误码相关的其他参数。使用这些信息，再加上一些栈跟踪信息（显示按顺序调用的一组 Oracle 内部子例程），可能会发现这个 bug 已经报告过（还能找到解决方法、补丁等）。例如，使用以下查询串：

```
ora-00600 12410 ksesic0 qerixAllocate qknRwsAllocateTree
```

利用 MetaLink 的高级搜索（全文搜索 bug 数据库），很快就能发现 bug 3800614，“ORA-600 [12410] ON SIMPLE QUERY WITH ANALYTIC FUNCTION”。如果访问 http://metalink.oracle.com，并使用这个文本进行搜索，可以找到这个 bug，了解到下一版中已经修正了这个 bug，并注意到已经有相应的补丁，所有这些信息我们都能得到。很多次我都发现，所遇到的错误以前已经出现过，而且事实上已经有了修正和解决的办法。

3.2.3 跟踪文件小结

现在你知道有两种一般的跟踪文件，它们分别放在什么位置，以及如何找到这些跟踪文件。希望你使用跟踪文件主要是为了调整和改善应用的性能，而不只是提交 iTAR。最后再说一句，Oracle Support 确实会利用文档中没有记录的一些“事件”，如果数据库遭遇错误，可以利用这些事件得到大量的诊断信息。例如，如果得到一个 ORA-01555，但你自认为不该有这个错误，此时 Oracle Support 就会教你设置这种诊断事件，每次遇到错误时都会创建一个跟踪文件，由此可以帮助你准确地跟踪到为什么会产生错误。

3.3 警告文件

警告文件（也称为警告日志（alert log）） 就是数据库的日记。这是一个简单的文本文件，从数据库“出生”（创建）那一天起就会编写该文件，直到数据库“完结”（被你删除）为止。在这个文件中，可以 看到数据库的“编年史”，包括日志开关；可能出现的内部错误；表空间何时创建、离线以及恢复为在线，等等。这是一个查看数据库历史的极其有用的文件。我喜 欢让警告文件尽量地增长，直到非常大了才会对其“编卷”（归档）。在我看来，这个文件里的信息越多越好。

我不会介绍警告日志里的每一处细节，这个范围实在太大了。不过，建议你把自己的警告日志拿来看看，你会发现其中的大量信息。在这一节中，我们会介绍一个特定的例子，并通过这个例子来说明如何挖掘警告日志中的信息，并建立一个正常运行报告。

最近我利用http://asktom.oracle.com网站的警告日志文件生成了我的数据库的一个正常运行报告。我不想全面盘查文件，然后手动地得到报告（警告文件中有关闭和启动时间），而是决定充分利用数据库和SQL来自动完成，所以我开发了一种技术，从而由警告日志直接创建动态的正常运行报告。

通过 EXTERNAL TABLE（第 10 章将更详细地介绍）可以查询警告日志，并了解其中有什么。我发现，每次启动数据库时警告日志里都会产生几天记录：

```
Thu May 6 14:24:42 2004

Starting ORACLE instance (normal)
```

这是一个时间戳记录（定宽格式），还有一条消息：Starting ORACLE instance。我还注意到，在这些记录前面，可能有一个 ALTER DATABASE CLOSE 消 息（正常关闭期间）或者一个关闭异常中止的消息，也可能“什么也没有”，没有消息，这意味着系统崩溃了。但是只要有消息，就肯定有相关的每个时间戳。所 以，只要系统没有“崩溃”，就会在警告日志里记录一些有意义的时间戳（如果系统崩溃了，也会记录崩溃前不久的一个时间戳，因为警告日志写得相当频繁）。

我注意到，如果做到以下几条，就能很容易地生成一个正常运行报告：

- 收集所有 Starting ORACLE instance %之类的记录。
- 收集所有与日期格式匹配的记录（实际上就是日期）。
- 将每个 Starting ORACLE instance 记录与前面的两个记录想关联（前面的两个记录应该是日期）。

以下代码创建了一个外部表，以便查询警告日志（注意：要把/background/dump/dest/换成你自己的后台转储目标目录，并在 CREATE TABLE 语句中使用你自己的警告日志名 。

```
ops$tkyte@ORA10G> create or replace directory data_dir as '/background/dump/dest/'

  2 /

Directory created.
```

```
ops$tkyte@ORA10G> CREATE TABLE alert_log
  2          (
  3                  text_line varchar2(255)
  4          )
  5          ORGANIZATION EXTERNAL
  6          (
  7                  TYPE ORACLE_LOADER
  8                  DEFAULT DIRECTORY data_dir
  9                  ACCESS PARAMETERS
 10                 (
 11                         records delimited by newline
 12                         fields
 13                         REJECT ROWS WITH ALL NULL FIELDS
 14                 )
 15                 LOCATION
 16                 (
 17                         'alert_AskUs.log'
 18                 )
 19         )
 20         REJECT LIMIT unlimited
 21 /
Table created.
```

这样任何时间都能查询这些信息了：

```
ops$tkyte@ORA10G> select to_char(last_time,'dd-mon-yyyy hh24:mi') shutdown,
  2                 to_char(start_time,'dd-mon-yyyy hh24:mi') startup,
  3                 round((start_time-last_time)*24*60,2) mins_down,
  4                 round((last_time-lag(start_time) over (order by r)),2) days_up,
  5                 case when (lead(r) over (order by r) is null )
  6                         then round((sysdate-start_time),2)
  7                 end days_still_up
  8           from (
  9 select r,
 10                 to_date(last_time, 'Dy Mon DD HH24:MI:SS YYYY') last_time,
 11                 to_date(start_time,'Dy Mon DD HH24:MI:SS YYYY') start_time
 12 from (
 13 select r,
 14                 text_line,
 15                 lag(text_line,1) over (order by r) start_time,
 16                 lag(text_line,2) over (order by r) last_time
 17 from (
 18 select rownum r, text_line
 19 from alert_log
 20 where text_line like '___ ___ __ __:__:__ 20__'
 21        or text_line like 'Starting ORACLE instance %'
 22        )
 23        )
```

```
  24 where text_line like 'Starting ORACLE instance %'

  25       )

  26 /

SHUTDOWN                       STARTUP                              MINS_DOWN
DAYS_UP     DAYS_STILL_UP

-------------------------      -------------------------            ------------------
--------------     ----------------------

                              06-may-2004 14:00

              06-may-2004              14:24                        06-may-2004
14:24              .25            .02

              10-may-2004              17:18                        10-may-2004
17:19              .93            4.12

               26-jun-2004             13:10                        26-jun-2004
13:10              .65            46.83

              07-sep-2004              20:13                        07-sep-2004
20:20             7.27            73.29              116.83
```

这里不会详细讨论 SQL 查询的细节，不过要知道，第 18～21 行的最内层查询用于收集“Starting”消息和日期行记录（记住，使用 LIKE 子句时，“_”只匹配一个字符，不多不少只能是一个字符）。通过使用 ROWNUM 还对行”编号“。然后，下一层查询（外层查询）使用内置的 LAG()分析函数，对每一行记录，分别返回到其前一行及前两行，然后将数据综合起来，这样查询的第 3 行就同时包含了第 1 行、第 2 行和第 3 行的数据。相应地，第 4 行就有了第 2 行、第 3 行和第 4 行的数据，依此类推。最后只留下形如 Starting ORACLE instance %的行，而且现在每一行都有了前面的两个相关的时间戳。从这个查询结果计算停机数据就很容易了：只需将两个日期相减。计算正常运行时间也不难（你已经了解了 LAG()函数的作用）：只需回到前一行，得到其启动时间，再将当前行的停机时间减去前一行的启动时间，就可以得到正常运行时间。

我的 Oracle 10g 数据库是 5 月 6 日启动的，其间关闭过 4 次（到编写这本书时，它已经连续运行了 116.83 天）。平均正常运行时间越来越好（要知道，利用强大的 SQL，我们还能很轻松地计算出平均运行时间的改进程度）。

如果你对此很感兴趣，想再看一个从警告日志中挖掘有用信息的例子，可以访问http://asktom.oracle.com/~tkyte/alert_arch.html。这个网页演示了如果计算将给定在线重做日志文件归档所用的平均时间。一旦了解了警告日志中有些什么，你自己来生成这些查询就很容易了。

3.4 数据文件

数据文件和重做日志文件是数据库中最重要的文件。你的数据最终就是要存储在数据文件中。每个数据库都至少有一个相关的数据文件，通常还不止一个。最简单的“测试”数据库只有一个数据文件。实际上，在第 2 章中我们已经见过一个例子，其中用最简单的 CREATE DATABASE 命令根据默认设置创建了一个数据库，这个数据库中有两个数据文件，其中一个对应 SYSTEM 表空间（真正的 Oracle 数据字典），另一个对应 SYSAUX 表空间（在 10g 及以上版本中，非字典对象都存储在这个表空间中）。不过，所有实际的数据库都至少有 3 个数据文件；一个存储 SYSTEM 数据，一个存储 SYSAUX 数据，还有一个存储 USER 数据。

简要回顾文件系统类型之后，我们将讨论如何组织这些文件，以及文件中如何组织数据。要了解这些内容，需要知道什么是表空间（tablespace）、什么是段（segment）、什么是区段（extent），以及什么是块（block）。这些都是 Oracle 在数据库中存储对象所用的分配单位，稍后将详细介绍。

3.4.1 简要回顾文件系统机制

在 Oracle 中，可以用 4 种文件系统机制存储你的数据。这里强调了“你的数据”，是指你的数据字典、redo 记录、undo 记录、表、索引、LOB 等，也就是你自己每天关心的数据。简单地讲，这包括：

- “Cooked”操作系统（OS）文件系统：这些文件就像字处理文档一样放在文件系统中。在 Windows 资源管理器中可以看到这些文件，在 UNIX 上，可以通过 ls 命令看到这些文件。可以使用简单的 OS 工具（如 Windows 上的 xcopy 或 UNIX 上的 cp）来移动文件。从历史上看，Cooked OS 文件一直是 Oracle 中存储数据的“最流行”的方法，不过我个人认为，随着 ASM（稍后再详细说明）的引入，这种情况会有所改观。Cooked 文件系统（“加工”文件系统或“熟”文件系统）通常也会缓存，这说明在你读写磁盘时，OS 会为你缓存信息。

- 原始分区（raw partitions，也称裸分区）：这不是文件，而是原始磁盘。不能用 ls 来查看；不能在 Windows 资源管理器中查看其内容。它们就是磁盘上的一些大扇区，上面没有任何文件系统。对 Oracle 来说，整个原始分区就是一个大文件。这与 cooked 文件系统不同，cooked 文件系统上可能有几十个甚至数百个数据库数据文件。目前，只有极少数 Oracle 安装使用原始分区，因为原始分区的管理开销很大。原始分区不是缓冲设备，所完成的所有 I/O 都是直接 I/O，对数据没有任何 OS 缓冲（不过，对于数据库来说，这通常是一个优点）。

- 自动存储管理（Automatic Storage Management，ASM）：这是 Oracle 10g Release 1 的一个新特性（标准版和企业版都提供了这个特性）。ASM 是专门为数据库设计的文件系统。可以简单地把它看作一个数据库文件系统。在这个文件系统上，不是把购物清单存储在文本文件中；这里只能存储与数据库相关的信息：你的表、索引、备份、控制文件、参数文件、重做日志、归档文件等。不过，即使是 ASM，也同样存在着相应的数据文件；从概念上讲，数据库仍存储在文件中，不过现在的文件系统是 ASM。ASM 设计成可以在单机环境或者集群环境中工作。

- 集群文件系统：这个文件系统专用于 RAC（集群）环境，看上去有些像由集群

环境中多个节点（计算机）共享的 cooked 文件系统。传统的 cooked 文件系统只能由集群环境中的一台计算机使用。所以，尽管可以在集群中的多个节点之间使用 NFS 装载或 Samba 共享一个 cooked 文件系统（Samba 与 NFS 类似，可以在 Windows/UNIX 环境之间共享磁盘），但这会导致一损俱损。如果安装有文件系统并提供共享的节点失败，这个文件系统都将不可用。Oracle 集群文件系统（Oracle Cluster File System，OCFS）是 Oracle 在这个领域推出的一个新的文件系统，目前只能在 Windows 和 Linux 上使用。其他第三方开发商也提供了一些经认证的集群文件系统，也可以用于 Oracle。集群文件系统让 cooked 文件系统的优点延伸到了集群环境中。

有意思的是，数据库可能包含来自上述所有文件系统中的文件，你不必只选其中的一个。在你的数据库中，可能部分数据存储在一个传统的 cooked 文件系统中，有些在原始分区上，有一些在 ASM 中，还有一些在集群文件系统中。这样就能很容易地切换技术，或者只是涉及一个新的文件系统，而不必把整个数据库都搬到这个文件系统中。现在，因为完整地讨论文件系统及其详细的属性超出了本书的范围，所以我们还是回过头来深入探讨 Oracle 文件类型。不论文件是存储在 cooked 文件系统、原始分区、ASM 中，还是存储在集群文件系统中，以下概念都适用。

3.4.2 Oracle 数据库中的存储层次体系

数据库由一个或多个表空间构成。表空间（tablespace）是 Oracle 中的一个逻辑存储容器，位于存储层次体系的顶层，包括一个或多个数据文件。这些文件可能是文件系统中的 cooked 文件、原始分区、ASM 管理的数据库文件，或者是集群文件系统上的文件。表空间包含段，请看下面的介绍。

1. 段

现在开始分析存储层次体系，首先讨论段，这是表空间中主要的组织结构。段（segment）就是占用存储空间的数据库对象，如表、索引、回滚段等。创建表时，会创建一个表段。创建分区表时，则每个分区会创建一个段。创建索引时，就会创建一个索引段，依此类推。占用存储空间的每一个对象最后都会存储在一个段中，此外还有回滚段（rollback segment）、临时段（temporary segment）、聚簇段（cluster segment）、索引段（index segment）等。

注意 上面有这样一句话：“占用存储空间的每一个对象最后都会存储在一个段中”，这可能会把你搞糊涂。你会发现许多 CREATE 语句能创建多段的对象。之所以会产生困惑，原因是一条 CREATE 语句最后创建的对象可能包含 0 个、1 个或多个段！例如，CREATE TABLE T (x int primary key, y clob)就会创建 4 个段：一个是 TABLE T 的段，还有一个段对应索引（这个索引是为支持主键而创建的），另外还有两个 CLOB 段（一个 CLOB 段是 LOB 索引，另一个段是 LOB 数据本身）。与之不同，CREATE TABLE T (x int, y date) cluster MY_CLUSTER 则不会创建任何段。第 10 章还会更深入地讨论这个概念。

2. 区段

段本身又由一个或多个区段组成。区段（extent）是文件中一个逻辑上连续分配的空间（一般来讲，文件本身在磁盘上并不是连续的；否则，根本就不需要消除磁盘碎片的工具了！）。另外，利用诸如独立磁盘冗余阵列（Redundant Array of Independent Disks，RAID） 之类的磁盘技术，你可能会发现，一个文件不仅在一个磁盘上不连续，还有可能跨多个物理磁

盘。每个段都至少有一个区段，有些对象可能还需要至少两个区段（回 滚段就至少需要两个区段）。如果一个对象超出了其初始区段，就会请求再为它分配另一个区段。第二个区段不一定就在磁盘上第一个区段旁边，甚至有可能不在第 一个区段所在的文件中分配。第二个区段可能与第一个区段相距甚远，但是区段内的空间总是文件中的一个逻辑连续空间。区段的大小可能不同，可以是一个 Oracle 数据块，也可以大到 2 GB。

3. 块

区段又进一步由块组成。块（block）是 Oracle 中最小的空间分配单位。数据行、索引条目或临时排序结果就存储在块中。通常 Oracle 从磁盘读写的就是块。Oracle 中块的常见大小有 4 种：2 KB、4 KB、8 KB 或 16 KB（尽管在某些情况下 32 KB 也是允许的；但是操作系统可能对最大大小有限制）。

注意 有一点可能很多人都不知道：数据库的默认块大小不必是 2 的幂。2 的幂只是一个常用的惯例。实际上，你完全可以创建块大小为 5 KB、7 KB 或 n KB 的数据库，这里 n 介于 2～32 KB 之间。不过，我还是建议你在实际中不要考虑这样做，块大小还是用 2 KB、4 KB、8 KB 或 16 KB 比较好。

段、区段和数据块之间的关系如图 3-1 所示。

一个段由一个或多个区段组成，区段则由连续分配的一些块组成。从 Oracle9i Release 1 起，数据库中最多可以有 6 种不同的块大小（block size）。

图 3-1 段、区段和数据块

注意 之所以引入这个特性，即一个数据库中允许有多种块大小，目的是为了可以在更多的情况下使用可传输的表空间。如果能传输表空间，DBA 就能从一个数据库移动或复制格式化的数据文件，把它放在另一个数据库中，例如，可以从一个联机事务处理（Online Transaction Processing，OLTP）数据库中把所有表和索引复制到一个数据仓库（Data Warehouse，DW）中。不过，在许多情况下，OLTP 数据库使用的块大小可能很小，如 2 KB 或 4 KB，而 DW 使用的块大小可能很大（8 KB 或 16 KB）。如果一个数据库中不支持多种块大小，就无法传输这些信息。有多种块大小的表空间主要用于传输表空间，一般没有其他用途。

数据库还有一个默认的块大小，即执行 CREATE DATABASE 命令时初始化文件中指定的大小。SYSTEM 表空间总是使用这个默认块大小，不过你完全可以按非默认块大小（2 KB、4 KB、8 KB 或 16 KB）创建其他表空间，如果操作系统允许，还可以使用 32 KB 的块大小。

当且仅当创建数据库时指定了一个非标准的块大小（不是 2 的幂）时，才会有 6 种不同的块大小。因此，在实际中，数据库最多有 5 种不同的块大小：默认大小和另外 4 种非默认的块大小。

在所有给定的表空间内部，块大小都是一致的，这说明，一个表空间中的所有块大小都相同。对于一个多段对象，如一个包含 LOB 列的表，可能每个段在不同的表空间中，而这些表空间分别有不同的块大小，但是任何给定段（包含在表空间中）都由相同大小的块组成。无论大小如何，所有块格式都一样，如图 3-2 所示。

图 3-2 块结构

块首部（block header） 包含块类型的有关信息（表块、索引块等）、块上发生的活动事务和过去事务的相关信息（仅事务管理的块有此信息，例如临时排序块就没有事务信息），以及块在 磁盘上的地址（位置）。块中接下来两部分是表目录和行目录，最常见的数据库块中（即堆组织表的数据块）都有这两部分。第 10 章将更详细地介绍数据库表类型，不过，现在知道大多数表都是这种类型就足够了。如果有表目录（table directory），则其中会包含把行存储在这个块上的表的有关信息（可能一个块上存储了多个表的数据）。行目录（row directory）包含块中行的描述信息。这是一个指针数组，指向块中数据部分中的行。块中的这 3 部分统称为块开销（block overhead），这部分空间并不用于存放数据，而是由 Oracle 用来管理块本身。块中余下的两部分就很清楚了：块上可能有一个空闲空间（free space），通常还会有一个目前已经存放数据的已用空间（used space）。

从以上介绍可以知道，段由区段组成，区段由块组成，对段有了大致的了解后，下面再来更深入地分析表空间，然后说明文件在这个存储层次体系中的位置。

4. 表空间

前 面已经提到，表空间是一个容器，其中包含有段。每个段都只属于一个表空间。一个表空间中可能有多个段。一个给定段的所有区段都在与段相关联的表空间中。段 绝对不会跨越表空间边界。表空间本身可以有一个或多个相关的数据文件。表空间中给定段的一个区段完全包含在一个数据文件中。不过，段可以有来自多个不同数 据文件的区段。表空间如图 3-3 所示。

图 3-3 这个表空间包含两个数据文件、3 个段和 4 个区段

图 3-3 显示了一个名为 USER_DATA 的表空间。其中包括两个数据文件：user_data01 和 user_data02。并分配了 3 个段：T1、T2 和 I1（可能是两个表和一个索引）。这个表空间中分配了 4 个区段，每个区段表示为逻辑上连续分配的一组数据库块。段 T1 包括两个区段，分别在不同的文件中。段 T2 和 I1 都各有一个区段。如果这个表空间需要更多的空间，可以调整已经分配给表空间的数据文件的大小，或者可以再增加第三个数据文件。

表空间是 Oracle 中的逻辑存储容器。作为开发人员，我们会在表空间中创建段，而绝对不会深入到原始的“文件级”。我们可不希望在一个特定的文件中分配区段（当然这也是可以的，但我们一般都不会这么做）。相反，我们会在表空间中创建对象，余下的工作都由 Oracle 负责。如果将来某个时刻 DBA 决定在磁盘上移动数据文件，从而使 I/O 分布得更均匀，这对我们来说没有任何关系，它根本不会影响我们的处理。

5. 存储层次体系小结

总结一下，Oracle 中的存储层次体系如下：

(1) 数据库由一个或多个表空间组成。

(2) 表空间由一个或多个数据文件组成。这些文件可以是文件系统中的 cooked 文件、原始分区、ASM 管理的数据库文件，或集群文件系统上的文件。表空间包含段。

(3) 段（TABLE、INDEX 等）由一个或多个区段组成。段在表空间中，但是可以包含这个表空间中多个数据文件中的数据。

(4) 区段是磁盘上一组逻辑连续的块。区段只在一个表空间中，而且总是在该表空间内的一个文件中。

(5) 块是数据库中最小的分配单位，也是数据库使用的最小 I/O 单位。

3.4.3 字典管理和本地管理的表空间

在继续讨论之前，我们再来看看关于表空间的一个问题：在表空间中如何管理区段。在 Oracle 8.1.5 之前，表空间中管理区段的分配只有一种方法：字典管理的表空间（dictionary-managed tablespace）。也就是说，表空间中的空间在数据字典表中管理，这与管理账户数据（利用 DEBIT 和 CREDIT 表）的方法是一样的。借方有已经分配给对象的所有区段。贷方是所有可用的自由区段。如果一个对象需要另一个区段，就会向系统“申请”。然后 Oracle 访问其数据字典表，运行一些查询，查找到空间（也许找不到），然后更新一个表中的一行（或者从表中将这一行删除），再向另一个表插入一行。Oracle 管理空间与你编写应用可谓异曲同工：同样是要修改数据以及移动数据。

为了得到额外的空间而在后台代表你执行的 SQL 称为递归 SQL（recursive SQL）。你的 SQL INSERT 语句会导致执行其他递归 SQL 来得到更多空间。如果频繁地执行这种递归 SQL，开销可能相当大。对数据字典的这种更新必须是串行的；它们不能同时进行，所以要尽量避免。

在 Oracle 的早期版本中，可以看到，这种空间管理问题（递归 SQL 开销）在“临时表空间”中最常见（这还不是“真正的”临时表空间，真正的临时表空间是通过 CREATE TEMPORARY TABLESPACE 命令创建的）。空间会频繁地分配（从字典表删除，而插入到另一个表）和撤销（把刚移动的行再移回原来的位置）。这些操作必须串行执行，这就大大削弱了并发性，而增加了等待时间。在 7.3 版本中，Oracle 引入了一个真正的临时表空间（true temporary tablespace）概念，这是一个新的表空间类型，专门用于存储临时数据，从而帮助缓解这个问题。在引入这个特殊的表空间类型之前，临时数据与永久数据在同样的表空间中管理，处理方式也与永久数据一样。

而临时表空间则不同，你不能在其中创建自己的永久对象。实际上根本的区别只有这一条；空间还是在数据字典表中管理。不过，一旦在临时表空间中分配了一个区段，系统就会一直持有（也就是说，不会把空间交回）。下一次有人出于某种目的在临时表空间中请求空间时，Oracle 会 在其内部的已分配区段列表中查找已经分配的区段。如果找到，就会直接重用，否则还是用老办法来分配一个区段。采用这种方式，一旦数据库启动，并运行一段时 间，临时段看上去就好像满了，但是实际上只是“已分配”。里面都是空闲区段，它们的管理完全不同。当有人需要临时空间时，Oracle 会在内存中的数据结构里查找空间，而不是执行代价昂贵的递归 SQL。

在 Oracle 8.1.5 及以后版本中，Oracle 在减少这种空间管理开销方面又前进了一步。它引入了一个本地管理表空间（locally-managed tablespace ）概念，而不是字典管理表空间。与 Oracle 7.3 中对临时表空间的管理一样，本地空间管理采用了同样的办法来管理所有表空间：这样就无需使用数据字典来管理表空间中的空间。对于本地管理表空间，会使用每个数据文件中存储的一个位图来管理区段。现在要得到一个区段，系统所做的只是在位图中将某一位设置为 1。要释放空间，系统再把这一位设置为 0。 与使用字典管理的表空间相比，这样分配和释放空间就相当快。为了处理跨所有表空间的空间请求，我们不再需要在数据库级串行完成这些耗时的操作，相反，只需 在表空间级串行执行一个速度相当快的操作。本地管理的表空间还有另外一些很好的特点，如可以保证区段的大小统一，不过这一点 DBA 更关心。

再往后，则只应使用本地管理的表空间作为存储管理方法。实际上，在 Oracle9i 及以上版本中，如果使用数据库配置助手（database configuration assistant，DBCA）创建一个数据库，它就会创建一个 SYSTEM 作为本地管理的表空间，如果 SYSTEM 是本地管理的，那么该数据库中所有其他表空间也会是本地管理的，而且遗留的字典管理方法将无法工作。如果数据库中的 SYSTEM 是本地管理的表空间，并不是说这样的数据库中不支持字典管理的表空间，而是说其中根本无法创建字典管理的表空间：

```
ops$tkyte@ORA10G> create tablespace dmt
  2 datafile '/tmp/dmt.dbf' size 2m
  3 extent management dictionary;
```

```
create tablespace dmt

*

ERROR at line 1:

ORA-12913: Cannot create dictionary managed tablespace

ops$tkyte@ORA10G> !oerr ora 12913

12913, 00000, "Cannot create dictionary managed tablespace"

// *Cause: Attempt to create dictionary managed tablespace in database

// which has system tablespace as locally managed

// *Action: Create a locally managed tablespace.
```

这是一个正面的副作用，因为这样可以杜绝你使用遗留的存储机制，要知道它的效率相对较低，而且很可能导致碎片。本地管理的表空间除了在空间分配和撤销方面效率更高以外，还可以避免出现表空间碎片，这正是以本地管理表空间的方式分配和撤销空间的一个副作用。有关内容将在第 10 章更深入地讨论。

3.5 临时文件

Oracle 中的临时数据文件（Temporary data files）即临时文件（temp files）是一种特殊类型的数据文件。Oracle 使用临时文件来存储大规模排序操作和散列操作的中间结果，如果 RAM 中没有足够的空间，还会用临时文件存储全局临时表数据，或结果集数据。永久数据对象（如表或索引）不会存储在临时文件中，但是临时表及其索引的内容要存储在临时文件中。所以，你不可能在临时文件中创建表，但是使用临时表时完全可以在其中存储数据。

Oracle 以一种特殊的方式处理临时文件。一般而言，你对对象所做的每一个修改都会存储在重做日志中；这些事务日志会在以后某个时间重放以“重做事务”，例如，失败后进行恢复时就可能需要“重做事务”。临时文件不包括在这个重放过程内。对临时文件并不生成 redo 日志，不过可以生成 undo 日志。由于 UNDO 总是受 redo 的“保护”，因此，这就会生成使用临时表的 redo 日志，有关详细内容见第 9 章。为全局临时表生成 undo 日志的目的是为了回滚在会话中所做的一些工作，这可能是因为处理数据时遇到一个错误，也可能因为某个一般性的事务失败。DBA 不需要备份临时数据文件，实际上，备份临时数据文件只会浪费时间，因为你无法恢复临时数据文件。

建议将数据库配置为使用本地管理的临时表空间。作为 DBA，要确保使用 CREATE TEMPORARY TABLESPACE 命令。你肯定不想把一个永久表空间改成临时表空间，因为这样得不到临时文件的任何好处。

关于真正的临时文件，有一个细节需要注意，如果操作系统允许创建临时文件，则会稀疏（sparse）地创建，也就是说，在需要之前它们不会真正占用磁盘存储空间。通过下面这个例子能很容易看出这一点（这里的平台是 Red Hat Linux）：

```
ops$tkyte@ORA10G> !df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/hda2      74807888  41999488  29008368  60% /
/dev/hda1        102454     14931     82233  16% /boot
none            1030804         0   1030804   0% /dev/shm

ops$tkyte@ORA10G> create temporary tablespace temp_huge
  2 tempfile '/d01/temp/temp_huge' size 2048m
  3 /
Tablespace created.

ops$tkyte@ORA10G> !df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/hda2      74807888  41999616  29008240  60% /
/dev/hda1        102454     14931     82233  16% /boot
none            1030804         0   1030804   0% /dev/shm
```

注意　df 是显示“磁盘空闲空间”的 Unix 命令。这个命令显示出，向数据库中添加一个 2 GB 的临时文件之前，包含/d01/temp 的文件系统中有 29 008 368 KB 的空闲空间。添加了这个文件之后，文件系统中有 29 008 240 KB 的空闲空间。

显然，这个文件只占了 128 KB 的存储空间，但是，如果用 ls 将其列出，可以得到：

```
ops$tkyte@ORA10G> !ls -l /d01/temp/temp_huge
-rw-rw---- 1 ora10g ora10g 2147491840 Jan 2 16:34 /d01/temp/temp_huge
```

看上去是一个正常的 2 GB 文件，但它实际上只用了 128 KB 的存储空间。之所以要指出这一点，原因是我们实际上可能创建了数百个 2 GB 的临时文件，尽管空闲的磁盘空间只有大约 29 GB。 听起来不错，空闲空间那么多！问题是，真正开始使用这些临时文件时，它们就会膨胀，很快我们就会得到“没有更多空间”的错误。由于空间会按操作系统的需要来分配或者物理地分配文件，所以我们肯定会用光空间（特别是这样一种情况，我们创建了临时文件后，有人又用其他内容把文件系统填满了，此时临时文件实际上 根本没有可用的

空间）。

这个问题的解决因操作系统而异。在 Linux 上，可以使用 dd 在文件中填入数据，这样，操作系统就会物理地为文件分配磁盘空间，或者使用 cp 创建一个非稀疏的文件，例如：

```
ops$tkyte@ORA10G> !cp --sparse=never /d01/temp/temp_huge /d01/temp/temp_huge2


ops$tkyte@ORA10G> !df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/hda2      74807888  44099336  26908520  63% /
/dev/hda1        102454     14931     82233  16% /boot
none            1030804         0   1030804   0% /dev/shm


ops$tkyte@ORA10G> drop tablespace temp_huge;
Tablespace dropped.


ops$tkyte@ORA10G> create temporary tablespace temp_huge
  2 tempfile '/d01/temp/temp_huge2' reuse;
Tablespace created.


ops$tkyte@ORA10G> !df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/hda2      74807888  44099396  26908460  63% /
/dev/hda1        102454     14931     82233  16% /boot
none            1030804         0   1030804   0% /dev/shm
```

将稀疏的 2 GB 文件复制到/d01/temp/temp_huge2 中，并使用 REUSE 选项利用该临时文件创建临时表空间，这样就能肯定这个临时文件已经分配了所有文件系统空间，而且数据库确实有了 2 GB 的临时空间可以使用。

注意 根据我的经验，Windows NTFS 不支持稀疏文件，以上讨论只适用于 UNIX/Linux 平台。好的一面是，如果必须在 UNIX/Linux 上创建一个 15 GB 的临时表空间，而且支持临时文件，你会发现创建过程相当快（几乎立即完成），但是要保证确实有 15 GB 的空闲空间，而且一定要记住保留这些空间。

3.6 控制文件

控制文件（control file）是一个相当小的文件（最多能增长到 64 MB 左右），其中包含 Oracle 需要的其他文件的一个目录。参数文件告知实例控制文件的位置，控制文件则告知实例数据库和在线重做日志文件的位置。

控制文件还告知了 Oracle 其他一些事情，如已发生检查点的有关信息、数据库名（必须与 DB_NAME 参数匹配）、创建数据库的时间戳、归档重做日志的历史（有时这会让控制文件变大）、RMAN 信息等。

控制文件应该通过硬件（RAID）多路保存，如果不支持镜像，则要通过 Oracle 多路保存。应该有不止一个副本，而且它们应该保存在不同的磁盘上，以防止万一出现磁盘故障而丢失控制文件。丢失控制文件并不是致命的，但会使恢复变得困难得多。

开发人员实际上可能不会接触到控制文件。对于 DBA 来说，控制文件是数据库中一个非常重要的部分，但是对于软件开发人员，它们并不是太重要。

3.7 重做日志文件

重做日志文件（redo log file）对于 Oracle 数据库至关重要。它们是数据库的事务日志。通常只用于恢复，不过也可以用于以下工作：

- ☐ 系统崩溃后的实例恢复
- ☐ 通过备份恢复数据文件之后恢复介质
- ☐ 备用（standby）数据库处理
- ☐ 输入到流中，这是一个重做日志挖掘过程，用于实现信息共享（这也是一种奇特的复制）

重做日志文件的主要目的是，万一实例或介质失败，重做日志文件就能派上用场，或者可以作为一种维护备用数据库（standby database）的方法来完成故障恢复。如果数据库所在主机掉电，导致实例失败，Oracle 会使用在线重做日志将系统恢复到掉电前的那个时刻。如果包含数据文件的磁盘驱动器出现了永久性故障，Oracle 会使用归档重做日志以及在线重做日志，将磁盘驱动器的备份恢复到适当的时间点。另外，如果你“无意地”删除了一个表，或者删掉了一些重要的信息，而且提交了操作，则可以恢复一个备份，并让 Oracle 使用这些在线和归档重做日志文件将其恢复到意外发生前的那个时刻。

你在 Oracle 中完成的每个操作几乎都会生成一定的 redo 信息，并写入在线重做日志文件。向表中插入一行时，插入的最终结果会写入重做日志。删除一行时，则会在重做日志中写入你删除了这一行这一事实。删除一个表时，删除的效果会写入重做日志。从表中删除的数据不会写入；不过，Oracle 删除表时执行的递归 SQL 确实会生成 redo。例如，Oracle 从 SYS.OBJ$表（和其他内部字典对象）中删除一行时，这就会生成 redo，另外如果支持不同模式的补充日志（supplemental logging ），还会把具体的 DROP TABLE 语句写入重做日志

流。

有些操作可能会以尽量少生成 redo 的模式完成。例如，可以使用 NOLOGGING 属性创建一个索引。这说明，最初创建索引数据的操作不会记入日志，但是 Oracle 完成的所有递归 SQL 会写入日志。例如，创建索引后，将向 SYS.OBJ$表中插入一行表示索引存在，这个插入会记入日志，以后使用 SQL 插入、更新和删除等操作完成的修改也会记入日志。但是，最初向磁盘写索引结构的操作不会记入日志。

前面我提到了两种类型的重做日志文件：在线（online）和归档（archived）。下面几节将详细介绍这两类重做日志文件。在第 9 章中，我们还会结合回滚段来讨论 redo，看看它们对开发人员有什么影响。现在，我们只关注这些重做日志文件是什么，它们有什么用途。

3.7.1 在线重做日志

每个 Oracle 数据库都至少有两个在线重做日志文件组。每个重做日志组都包含一个或多个重做日志成员（redo 按成员组来管理）。这些组的单个重做日志文件成员之间实际上形成彼此真正的镜像。这些在线重做日志文件的大小是固定的，并以循环方式使用。Oracle 先写日志文件组 1，当到达这组文件的最后时，会切换至日志文件组 2，从头到尾重写这些文件的内容。日志文件组 2 填满时，再切换回到日志文件组 1（假设只有两个重做日志文件组；如果有 3 个重做日志文件组，当然会继续写第 3 个组）。如图 3-4 所示。

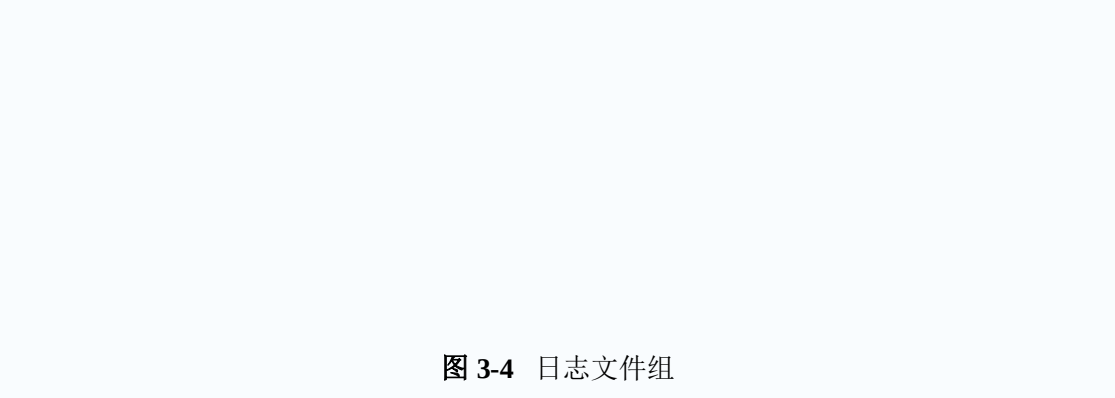

图 3-4 日志文件组

从一个日志文件组切换到另一个日志文件组的动作称为日志切换（log switch）。重要的是注意到，如果数据库配置得不好，日志切换可能会导致临时性“暂停”。由于重做日志的目的是在失败时恢复事务，所以我们自己必须保证一点：在重用重做日志之前，失败时应该不需要重做日志文件的内容。如果 Oracle 不能肯定这一点，也就是说，它不清楚是否真的不需要日志文件的内容，就会暂时挂起数据库中的操作，确保将缓存中的数据（即 redo“保护”的数据）安全地写入磁盘本身（建立检查点）。一旦 Oracle 能肯定这一点，再恢复处理，并重用重做文件。

我们刚刚提到一个重要的数据库概念：检查点（checkpointing）。要理解在线重做日志如何使用，就需要了解检查点，知道数据库缓冲区缓存如何工作，还要知道一个称为数据块写入器（data block writer，DBWn）的进程会做什么。数据库缓冲区缓存和 DBWn 将在后面详细讨论，但是我们先提前说两句，不过点到为止。

数据库缓冲区缓存（database buffer cache）就是临时存储数据库块的地方。这是 Oracle SGA 中的一个结构。读取块时，会存储在这个缓存中，这样以后就不必再物理地重新读取

它们。缓冲区缓存首先是一个性能调优设备，其目的只是让非常慢的物理 I/O 过程看上去快一些。修改块（更新块上的一行）时，这些修改会在内存中完成，写至缓冲区缓存中的块。另外，会把重做这些修改所需的足够信息保存在重做日志缓冲区（redo log buffer）中，这是另一个 SGA 数据结构。提交（COMMIT）修改时，会使这些修改成为永久的。Oracle 并不是访问 SGA 中修改的所有块，并把它们写到磁盘上。相反，它只是把重做日志缓冲区的内容写到在线重做日志中。只要修改的块还在缓冲区缓存中，而不在磁盘上，数据库失败时我们就会需要该在线重做日志的内容。如果提交过后，突然掉电，数据库缓冲区缓存就会彻底清空。

如果发生这种情况，则只有重做日志文件中有修改记录。重启数据库时，Oracle 实际上会重放我们的事务，再用同样的方式修改块，并提交。所以，只要修改的块被缓存而未写入磁盘，就不能重用重做日志文件。

在这里 DBWn 就能起作用了。这是 Oracle 的一个后台进程，负责在缓冲区缓存填满时请求空间，更重要的是，它会建立检查点。建立检查点就是把脏块（已修改的块）从缓冲区缓存写至磁盘。Oracle 会在后台为我们做这个工作。有很多情况都会导致建立检查点，最常见的事件就是重做日志切换。

在填满日志文件 1 并切换到日志文件 2 时，Oracle 就会启动一个检查点。此时，DBWn 开始将日志文件组 1 所保护的所有脏块写至磁盘。在 DBWn 把该日志文件保护的所有块刷新输出之前，Oracle 不能重用这个日志文件。如果 DBWn 在完成其检查点之前就想使用日志文件，就会在数据库的 ALERT 日志中得到以下消息：

```
...

Thread 1 cannot allocate new log, sequence 66

Checkpoint not complete

Current log# 2 seq# 65 mem# 0: C:\ORACLE\ORADATA\ORA10G\REDO02.LOG

...
```

所以，出现这个消息时，数据库中的处理会挂起，因为 DBWn 正忙于完成它的检查点。此时，Oracle 会尽可能地把所有处理能力都交给 DBWn，希望它能更快地完成。

如果数据库实例得到了妥善地调优，是不会看到这个消息的。如果你确实看到了这个消息，就应该知道肯定让最终用户陷入了不必要的等待，而这是可以避免的。我们的目标是分配足够的在线重做日志文件（这是对 DBA 而言，对开发人员则不一定），这样就不会在检查点完成之前试图重用日志。如果经常看到这个消息，这说明 DBA 未能为应用分配足够多的在线重做日志文件，或者要对 DBWn 进行调优才能更高效地工作。

不同的应用会生成不同数量的重做日志。很自然地，决策支持系统（Decision Support System，DSS，仅查询）或数据仓库（DW）系统生成的在线重做日志总是比 OLTP（事务处理）系统生成的在线重做日志少得多。如果一个系统在数据库中对二进制大对象（Binary Large Object，BLOB）完成了大量图像处理，相对于简单的订单输入系统来说，则会生成更多的 redo。有 100 位用户的订单输入系统与有 1,000 位用户的系统相比，生成的 redo 可能只是后者的十分之一。至于重做日志多大才合适，这没有“正确”的答案，不过你肯定希望

重做日志足够大，能适应你的工作负载。

在设置在线重做日志的大小和数目时，还有一些问题需要考虑。其中很多问题都超出了这本书的范围，不过在此把它们都列出来，以便你有一个大致的认识：

- 高峰负载（peak workload）：你可能希望系统不必等待对未完成的消息建立检查点，不要在高峰处理期间遭遇瓶颈。你不能针对“平均”的小时吞吐量来确定重做日志的大小，而要针对高峰处理来确定。如果每天生成 24 GB 的日志，但是其中 10 GB 的日志都是在 9:00 am 到 11:00 am 这一时段生成的，就要把重做日志的大小调整到足以放下那两小时高峰期间生成的日志。如果只是针对每小时 1 GB 来确定日志大小可能是不够的。

- 大量用户修改相同的块：如果大量用户都要修改相同的块，你可能希望重做日志文件很大。因为每个人都在修改同样的块，最好尽可能多地更新之后才将其写出到磁盘。每个日志切换都会导致一个检查点，所以你可能不希望频繁地切换日志。不过，这样一来又会影响恢复时间。

- 平 均恢复时间：如果必须确保恢复尽可能快地完成，即便是大量用户要修改相同的块，也可能倾向于使用较小的重做日志文件。如果只是处理一两个小的重做日志文 件，而不是一个巨大的日志文件，则所需的恢复时间会比较短。由于重做日志文件小，往往会过多地建立检查点，时间长了，整个系统会越来越慢（本不该如此 ， 但是恢复所花的时间确实会更短。要减少恢复时间，除了使用小的重做日志文件外，还可以使用其他的数据库参数。

3.7.2 归档重做日志

Oracle 数据库可以采用两种模式运行：ARCHIVELOG 模式和 NOARCHIVELOG 模式。这两种模式的区别只有一点，即 Oracle 重用重做日志文件时会发生什么情况。“会保留 redo 的一个副本吗？还是 Oracle 会将其重写，而永远失去原来的日志？”这是一个很重要的问题，下面就来回答。除非你保留了这个文件，否则无法从备份将数据恢复到当前的时间点。

假设你每周的星期六做一次备份。现在是星期五下午，已经生成了这一周的数百个重做日志，突然你的磁盘出问题了。如果没有以 ARCHIVELOG 模式运行，那么现在的选择只有：

- 删除与失败磁盘相关的表空间。只要一个表空间有该磁盘上的文件，就要删除这个表空间（包括表空间的内容）。如果影响到 SYSTEM 表空间（Oracle 的数据字典），就不能用这个办法。

- 恢复上周六的数据，这一周的工作就白做了。

不论是哪种选择都不太好。这两种做法都意味着你会丢失数据。不过另一方面，如果之前以 ARCHIVELOG 模式运行，那么只需再找一个磁盘就行了。你要根据上周六的备份将受影响的文件恢复到这个磁盘上。最后，再对这些文件应用归档重做日志和（最终的）在线重做日志，实际上是以一种快进的方式重放整个星期的事务。这样一来，什么也不会丢失。数据会恢复到发生失败的那个时间点。

人们经常告诉我，他们的生产系统不需要 ARCHIVELOG 模式。在我的印象里，这样说的人没有一个说对的。我认为，如果系统不以 ARCHIVELOG 模式运行，那它根本就不能

算是生产系统。未以 ARCHIVELOG 模式运行的数据库总有一天会丢失数据。这是在所难免的；如果你的数据库不以 ARCHIVELOG 模式运行，你肯定会丢失数据。

“我们在使用 RAID-5，所以可以得到完全的保护”，这是一种很常见的托辞。我曾见过，由于制造方面的错误，RAID 中的所有磁盘都“冻结”了，而且几乎是同时发生的。我也见过，有时硬件控制器会对数据文件带来破坏，所以他们只是在用 RAID 设备安全地保护已经被破坏的数据。另外，对于避免操作员错误（这也是丢失数据的一个最常见的原因），RAID 也无能为力。

“在出现硬件或操作员错误之前，而且归档尚未受到影响，如果此时建立了备份，就能很好地恢复”。关键是，既然系统上的数据是有价值的，有什么理由不采用 ARCHIVELOG 模式呢？性能不能作为理由；适当配置的归档只会增加极少的开销甚至根本不增加开销。由于这一点，再加上另外一条：如果一个系统会“丢失数据”，那它再快也是没有用的，所以退一万步说，即使归档会增加 100%的开销，我们也不得不做。如果可以把一个特性删除而没有任何重大损失，这个特性就叫做开销（overhead）；开销就像是蛋糕上的糖霜，可以不要而不会影响蛋糕的美味。但归档不同，利用归档可以保住你的数据，确保数据不会丢失，这不是开销，而且正是 DBA 的主要任务!

只有测试或开发系统才应当采用 NOARCHIVELOG 模式执行。不要受人蛊惑在非 ARCHIVELOG 模式下运行。你花了很长时间开发你的应用，肯定希望人们相信你。如果把他们的数据丢失了，也会让他们对你的系统失去信心。

注意 有些情况下，大型的 DW（数据仓库）以 NOARCHIVELOG 模式运行也是合适的，因为它可能适当地使用了 READ ONLY（只读）表空间，而且会通过重新加载数据来完全重建受失败影响的所有 READ WRITE（读写）表空间。

3.8 密码文件

密码文件（password file）是一个可选的文件，允许远程 SYSDBA 或管理员访问数据库。

启动 Oracle 时，还没有数据库可以用来验证密码。在“本地”系统上启动 Oracle 时（也就是说，不在网络上，而是从数据库实例所在的机器启动），Oracle 会利用操作系统来执行这种认证。

安装 Oracle 时，会要求完成安装的人指定管理员“组”。在 UNIX/Linux 上，这个组一般默认为 DBA，在 Windows 上则默认为 OSDBA。不过，也可以是平台上任何合法的组名。这个组很“特殊”，因为这个组中的任何用户都可以作为 SYSDBA 连接 Oracle ，而无需指定用户名或密码。例如，在安装 Oracle 10g Release 1 时，我指定了一个 ora10g 组。ora10g 组中的任何用户都无需用户名/密码就能连接：

```
[ora10g@localhost ora10g]$ sqlplus / as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:13:04 2005

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

Connected to:
```

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

With the Partitioning, OLAP and Data Mining options



SQL> show user

USER is "SYS"
```

这是可以的——我就成功地连接了 Oracle，现在我能启动这个数据库，将其关闭，或者完成我想做的任何管理工作。不过，假设我想从另外一台机器通过网络完成这些操作，会怎么样呢？在这种情况下，我试图使用@tns-connect-string 来连接。不过这会失败：

```
[ora10g@localhost admin]$ sqlplus /@ora10g_admin.localdomain as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:14:20 2005

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

ERROR:

ORA-01031: insufficient privileges



Enter user-name:
```

在网络上，对于 SYSDBA 的操作系统认证不再奏效，即使把很不安全的 REMOTE_OS_AUTHENT 参数设置为 TRUE 也 不例外。所以，操作系统认证不可行。如前所述，如果你想启动一个实例进行装载，并打开一个数据库，根据定义，在连接的另一端实际上“还没有数据库”，也无 法从中查找认证的详细信息。这就是一个鸡生蛋还是蛋生鸡的问题。因此密码文件“应运而生”。密码文件保存了一个用户名和密码列表，这些用户名和密码分别对 应于可以通过网络远程认证为 SYSDBA 的用户。Oracle 必须使用这个文件来认证用户，而不是数据库中存储的正常密码列表。

下面校正这种情况。首先，我们要本地启动数据库，以便设置 REMOTE_LOGIN_PASSWORDFILE。其默认值为 NONE，这意味着没有密码文件；不存在“远程 SYSDBA 登录”。这个参数还有另外两个设置：SHARED（多个数据库可以使用同样的密码文件）和 EXCLUSIVE（只有一个数据库使用一个给定的密码文件）。这里设置为 EXCLUSIVE，因为我们只想对一个数据库使用这个密码文件（这也是一般用法）：

```
SQL> alter system set remote_login_passwordfile=exclusive scope=spfile;

System altered.
```

实例启动和运行时，这个设置不能动态改变，所以要想让它生效必须重启实例。下一步是使用命令行工具（UNIX 和 Windows 平台上）orapwd 创建和填写这个初始的密码文件：

```
[ora10g@localhost dbs]$ orapwd

Usage: orapwd file=<fname> password=<password> entries=<users> force=<y/n>
```

在此：

file——密码文件名（必要）。

password——SYS 的密码（必要）。

entries——DBA 和操作员的最大数目（可选）。

force——是否重写现有的文件（可选）。

等号（=）两边没有空格。

我们使用的命令为：

```
$ orapwd file=orapw$ORACLE_SID password=bar entries=20
```

对我来说，这样会创建一个名为 orapwora10g 的密码文件（我的 ORACLE_SID 是 ora10g）。

这是大多数 UNIX 平台上密码文件的命名约定（有关各平台上密码文件的命名，详细内容请参见你的安装/操作系统管理员指南），这个文件位于$ORACLE_HOME/dbs 目录中。在 Windows 上，文件名为 PW%ORACLE_SID%.ora，在%ORACLE_HOME%\database 目录中。

目前该文件中只有一个用户，也就是用户 SYS，尽管数据库上还有其他 SYSDBA 账户，但它们还不在密码文件中。不过，基于以上设置，我们可以第一次作为 SYSDBA 通过网络连接 Oracle：

```
[ora10g@localhost dbs]$ sqlplus sys/bar@ora10g_admin.localdomain as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:49:15 2005

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

Connected to an idle instance.

SQL>
```

我们通过了认证，所以登录成功，现在可以使用 SYSDBA 账户成功地启动、关闭和远程管理这个数据库了。下面，再看另一个用户 OPS$TKYTE，它已经是一个 SYSDBA 账户（已经授予 SYSDBA），但是还不能远程连接：

```
[ora10g@localhost dbs]$ sqlplus 'ops$tkyte/foo' as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:51:07 2005

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

```
Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options

SQL> show user

USER is "SYS"

SQL> exit

[ora10g@localhost dbs]$ sqlplus 'ops$tkyte/foo@ora10g_admin.localdomain' as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:52:57 2005

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

ERROR:

ORA-01031: insufficient privileges

Enter user-name:
```

原因是，OPS$TKYTE 还不在密码文件中。要把 OPS$TKYTE 放到密码文件中，需要重新对该账户授予 SYSDBA：

```
SQL> grant sysdba to ops$tkyte;

Grant succeeded.



Disconnected from Oracle Database 10g

Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options

[ora10g@localhost dbs]$ sqlplus 'ops$tkyte/foo@ora10g_admin.localdomain' as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:57:04 2005

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

```
Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

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

这样会在密码文件中创建一个条目，Oracle 现在会保持密码“同步”。如果 OPS$TKYTE 修改了他的密码，原来的密码将无法完成远程 SYSDBA 连接，新密码才能启动 SYSDBA 连接：

```
SQL> alter user ops$tkyte identified by bar;

User altered.

[ora10g@localhost dbs]$ sqlplus 'ops$tkyte/foo@ora10g_admin.localdomain' as sysdba

SQL*Plus: Release 10.1.0.3.0 - Production on Sun Jan 2 20:58:36 2005

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

ERROR:

ORA-01017: invalid username/password; logon denied

Enter user-name: ops$tkyte/bar@ora10g_admin.localdomain as sysdba

Connected to:

Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options

SQL> show user

USER is "SYS"

SQL>
```

对于其他不在密码文件中的 SYSDBA 用户，再重复同样的过程。

3.9 修改跟踪文件

修改跟踪文件（change tracking file）是一个可选的文件，这是 Oracle 10g 企业版中新增的。这个文件惟一的目的是跟踪自上一个增量备份以来哪些块已经修改。采用这种方式，恢复管理器（Recovery Manager，RMAN）工具就能只备份确实有变化的数据库块，而不必读取整个数据库。

在 Oracle 10g 之前的版本中，要完成增量备份，必须读取整个数据库文件，查找自上一次增量备份以来修改的块。所以，如果有一个 1 TB 的数据库，只在其中增加了 500 MB 的新数据（例如，数据仓库负载），增量备份就必须读取 1 TB 的数据，在其中找出要备份的 500 MB 新信息。所以，尽管增量备份存储的数据确实少得多，但它还是要读取整个数据库。

在 Oracle 10g 企业版中，就不是这样了。Oracle 运行时，如果块被修改，Oracle 可能会维护一个文件，告诉 RMAN 哪些块已经修改。创建这个修改跟踪文件的过程相当简单，只需通过 ALTER DATABASE 命令就可以完成：

```
ops$tkyte@ORA10GR1> alter database enable block change tracking

  2 using file

  3 '/home/ora10gr1/product/10.1.0/oradata/ora10gr1/ORA10GR1/changed_blocks.bct';

Database altered.
```

警告 我在这本书里再三强调一点：要记住，不要轻易执行设置参数、修改数据库和产生重大改变的命令， 在你的“实际”系统上用这些命令之前一定要先进行测试。实际上，前面这个命令会导致数据库做更多工作。它会消耗资源。

要关闭和删除块修改跟踪文件，还要再用一次 ALTER DATABASE 命令：

```
ops$tkyte@ORA10GR1> alter database disable block change tracking;

Database altered.



ops$tkyte@ORA10GR1> !ls -l /home/ora10gr1/.../changed_blocks.bct

ls: /home/ora10gr1/.../changed_blocks.bct: No such file or directory
```

注意，这个命令实际上会清除块修改跟踪文件。它不只是禁用这个特性，而是连文件也一并删除了。可以采用 ARCHIVELOG 或 NOARCHIVELOG 模式再次启用这个新的块修改跟踪特性。不过，要记住，NOARCHIVELOG 模式的数据库中并不保留每天生成的重做日志，所以一旦介质（磁盘/设备）出现故障，所有修改都将无法恢复！NOARCHIVELOG 模式的数据库总有一天会丢失数据。我们将在第 9 章更详细地讨论这两种数据库模式。

3.10 闪回日志文件

闪回日志文件（flashback log file）简称为闪回日志（flashback log），这是 Oracle 10g 中为支持 FLASHBACK DATABASE 命令而引入的，也是 Oracle 10g 企业版的一个新特性。闪回日志包含已修改数据库块的“前映像”，可用于将数据库返回（恢复）到该时间点之前的状态。

3.10.1 闪回数据库

引入 FLASHBACK DATABASE 命令是为了加快原本很慢的时间点数据库恢复（point in

time database recovery）过程。闪回可以取代完整的数据库恢复和使用归档日志完成的前滚，主要目的是加快从“意外状态”中恢复。例如，下面来看这样一种情况，如果 DBA“意外地”删除了模式（schema），该如何恢复？他在本来要在测试环境中删除的数据库中删除了正确的模式。DBA 立即意识到做错了，并且随即关闭了数据库。现在该怎么办？

在引入闪回数据库功能之前，可能只能这样做：

(1) DBA 要关闭数据库。

(2) DBA（一般）要从磁带机恢复上一个完整的数据库备份。这通常是一个很长的过程。

(3) DBA 要恢复所生成的全部归档重做日志，因为系统上没有备份。

(4) DBA 要前滚数据库，并在出错的 DROP USER 命令之前的时间点停止。

(5) 要以 RESETLOGS 选项打开数据库。

这 个过程很麻烦，步骤很多，通常要花费很长的时间（当然，这个期间任何人都无法访问数据库）。导致这种时间点恢复的原因有很多：如升级脚本错误，升级失败， 有权限的某个人无意地发出了某个命令而导致时间点恢复（无意的错误，这可能是最常见的原因），或者是某个进程对一个大型数据库带来了数据完整性问题（同 样，这可能也是意外；也许是进程运行了两次而不是一次，也可能是因为存在 bug）。不论是什么原因，最终的结果都是很长时间的宕机。

Oracle 10g 企业版的恢复步骤如下，这里假设已经配置了闪回数据库功能：

(1) DBA 关闭数据库。

(2) DBA 启动并装载数据库，可以使用 SCN、Oracle 时钟或时间戳（墙上时钟时间）发出闪回数据库命令，时间可以精确到一两秒钟。

(3) DBA 以 RESETLOGS 选项打开数据库。

要使用这个特性，数据库必须采用 ARCHIVELOG 模式，而且必须配置为支持 FLASHBACK DATABASE 命令。我的意思是，在你使用这个功能之前，必须先行配置。等到真正发生了破坏，再想启用这个功能就为时已晚了；使用时必须早做打算。

3.10.2 闪回恢复区

闪回恢复区（Flash Recovery Area）也是 Oracle 10g 中的一个新概念。这么多年来（不止 25 年 ，Oracle 中数据库备份的基本概念第一次有了变化。过去，数据库中备份和恢复的设计都围绕着一种顺序介质（如磁带设备）的概念。也就是说，总是认为随机存取设备（磁盘设备）太过昂贵，只是用来完成备份有些浪费，而应该使用相对廉价但存储量大的磁带设备。

不过，现如今完全可以用很少的价钱买到容量达 TB 的磁盘。实际上，到 2007 年，HP 还打算推出磁盘容器达 TB 级的台式机。我还记得我的个人计算机上的第一块硬盘：在当时它的容量可是大得惊人：40 MB。实际上，我不得不把它分为两个逻辑盘，因为我所用的操作系统（当时是 MS-DOS）无法识别超过 32 MB 的硬盘。在过去的 20 年间，情况已经发生了翻天覆地的变化。

Oracle 10g 中的闪回恢复区（Flash Recovery Area）是一个新位置，Oracle 会在这里管理与数据库备份和恢复相关的多个文件。在这个区（area）中（这里“区”表示用于此目的的一个预留的磁盘区；例如一个目录），其中可以找到：

- 磁盘上数据文件的副本。
- 数据库的增量备份。
- 重做日志（归档重做日志）。
- 控制文件和控制文件的备份。
- 闪回日志。

Oracle 利 用这个新的闪回恢复区来管理这些文件，这样服务器就能知道磁盘上有什么，以及磁盘上没有什么（可能在别处的磁带上）。使用这些信息，数据库可以对被破坏的 数据文件完成磁盘到磁盘的恢复操作，或者对数据库完成闪回（这是一种“倒带”操作），从而撤销一个不该发生的操作。例如，可以使用闪回数据库命令，将数据 库放回到 5 分钟之前的状态（而不需要完整的数据库恢复和时间点恢复）。这样你就能“找回”无意删除的用户账户。

闪回恢复区更应算是一个“逻辑”概念。这是为本章讨论的各种文件类型所预留的一个区。使用闪回恢复区是可选的，没有必要非得使用，不过，如果你想使用诸如闪回数据库之类的高级特性，就必须用闪回恢复区存储信息。

3.11 DMP 文件（EXP/IMP 文件）

导出工具（Export）和导入工具（Import）是年头已久的 Oracle 数据抽取和加载工具，很多个版本中都有这些工具。导出工具的任务是创建一个平台独立的 DMP 文件（转储文件），其中包含所有必要的元数据（CREATE 和 ALTER 语句形式），可能还有数据本身，可以用于重新创建表、模式甚至整个数据库。导入工具的惟一作用就是读取这些 DMP 文件，执行其 DDL 语句，并加载它找到的所有数据。

DMP 文件设计为向后兼容，这说明新版本可以读取老版本的 DMP，并成功地处理。我听说有人导出过一个 Oracle 5 的数据库，并将其成功地导入到 Oracle 10g 中（只是一个测试！）。所以导入工具可以读取老版本的 DMP 文件，并处理其中的数据。不过，大多数情况下反过来不成立：Oracle9i Release 1 的导入工具进程不能（也不会）成功地读取 Oracle9i Release 2 或 Oracle 10g Release 1 创建的 DMP。例如，我曾经从 Oracle 10g Release 1 和 Oracle9i Release 2 导出过一个简单的表。我试图在 Oracle9i Release 1 中使用这些 DMP 文件时，很快发现 Oracle9i Release 1 导入工具甚至不打算处理 Oracle 10g Release 1 的 DMP 文件：

```
[tkyte@localhost tkyte]$ imp userid=/ full=y file=10g.dmp

Import: Release 9.0.1.0.0 - Production on Sun Jan 2 21:08:56 2005

(c) Copyright 2001 Oracle Corporation. All rights reserved.

Connected to: Oracle9i Enterprise Edition Release 9.0.1.0.0 - Production
```

```
With the Partitioning option

JServer Release 9.0.1.0.0 - Production

IMP-00010: not a valid export file, header failed verification

IMP-00000: Import terminated unsuccessfully
```

处理 Oracle9i Release 2 文件时，情况也好不到哪儿去：

```
[tkyte@localhost tkyte]$ imp userid=/ full=y file=9ir2.dmp

Import: Release 9.0.1.0.0 - Production on Sun Jan 2 21:08:42 2005

(c) Copyright 2001 Oracle Corporation. All rights reserved.

Connected to: Oracle9i Enterprise Edition Release 9.0.1.0.0 - Production

With the Partitioning option

JServer Release 9.0.1.0.0 – Production



Export file created by EXPORT:V09.02.00 via conventional path

import done in WE8ISO8859P1 character set and AL16UTF16 NCHAR character set

. importing OPS$TKYTE's objects into OPS$TKYTE

IMP-00017: following statement failed with ORACLE error 922:

"CREATE TABLE "T" ("X" NUMBER(*,0)) PCTFREE 10 PCTUSED 40 INITRANS 1
MAXTRA"

"NS 255 STORAGE(INITIAL 65536 FREELISTS 1 FREELIST GROUPS 1)
TABLESPACE "USE"

"RS" LOGGING NOCOMPRESS"

IMP-00003: ORACLE error 922 encountered

ORA-00922: missing or invalid option

Import terminated successfully with warnings.
```

9i Release 1 试图读取文件，但它无法处理其中包含的 DDL。Oracle9i Release 2 中增加了一个新特性，称为表压缩（table compression）。因此，这个版本的导出工具开始对每条

CREATE TABLE 语句增加一个 NOCOMPRESS 或 COMPRESS 关键字。Oracle9i Release 2 的 DDL 在 Oracle9i Release 1 中无法执行。

不过，如果对 Oracle9i Release 2 或 Oracle 10g Release 1 使用 Oracle9i Release 1 导出工具，总会得到一个有效的 DMP 文件，并可以成功地导入到 Oracle9i Release 1 中。所以，对于 DMP 文件的规则是：创建 DMP 文件的 Export 版本必须小于或等于使用该 DMP 文件的 Import 的版本。要将数据导入 Oracle9i Release 1 中，必须使用 Oracle9i Release 1 的导出工具（或者也可以使用一个 8i 的 Export 进程；创建 DMP 文件的 Export 版本必须小于或等于 Oracle9i Release 1）。

这些 DMP 文件是平台独立的，所以可以安全地用任何平台的导出工具创建 DMP 文件，然后转换到另一个平台，再导入这个 DMP 文件（只要 Oracle 版本允许）。不过，对于 Windows 和文件的 FTP 传输有一点警告，Windows 会默认地把 DMP 文件当成是一个“文本”文件，并把换行符（UNIX 上为行末标记）转换为回车/换行对，这就会完全破坏 DMP 文件。在 Windows 中通过 FTP 传输 DMP 文件时，要确保所执行的是二进制传输。如果导入不成功，请检查源文件大小和目标文件大小是否一样。这种问题常常导致令人痛苦的异常中止，而不得不重传文件，这种情况发生过多少次我简直都记不清了。

DMP 文件是二进制文件，这说明你不能编辑这些文件来进行修改。可以从中抽取大量信息（CREATE DDL），但是不能在文本编辑器（或者实际上任何类型的编辑器）中编辑它们。在第一版的 Expert One-on-One Oracle 中（你手上的是第二版，本书配套光盘提供了第一版的电子文档），我花了大量篇幅讨论导入和导出工具，并介绍了如何使用 DMP 文件。随着这些工具越来越失宠，取而代之的是更为灵活的数据泵工具，所以要想全面地了解如何管理导入和导出工具、如何从中抽取数据以及如何使用这些工具，请参考第一版的电子文档。

3.12 数据泵文件

Oracle 10g 中至少有两个工具使用数据泵（data pump）文件格式。外部表（external table）可以加载和卸载数据泵格式的数据，新的导入/导出工具 IMPDP 和 EXPDP 使用这种文件格式的方式与 IMP 和 EXP 使用 DMP 文件格式的方式完全一样。

注意 数据泵格式只在 Oracle 10g Release 1 及以后版本中可用，Oracle9i release 中没有也不能使用它。

前面提到过对 DMP 文件的警告，这些警告同样适用于数据泵文件。它们都是跨平台（可移植）的二进制文件，包含有元数据（并非存储为 CREATE/ALTER 语句，而是作为 XML 存储），可能还包含数据。数据泵文件使用 XML 作为元数据表示结构，这一点对你和我这些最终用户来说非常重要。IMPDP 和 EXPDP 有一些复杂的过滤和转换功能，这些在老版本的 IMP/EXP 工具中是没有的。从某种程度上讲，这就归功于使用了 XML，另外还因为 CREATE TABLE 语句并非存储为 CREATE TABLE，而是存储为一个有标记的文档。这样就能很容易地实现一些请求，如“请把表空间 FOO 的所有引用替换为表空间 BAR”。DMP 中元数据存储为 CREATE/ALTER 语句，导入工具在执行 SQL 语句之前实际上必须解析每一条 SQL 语句，才能完成这个工作（做得并不漂亮）。与之不同，IMPDP 只需应用一个简单的 XML 转换就能达到同样的目的，FOO（指一个 TABLESPACE）会转换为 <TABLESPACE>FOO</TABLESPACE>标记或另外某种表示。

由于使用了 XML，这使得 EXPDP 和 IMPDP 工具的功能相对于原来的 EXP 和 IMP 工具来说有了大幅的提升。在第 15 章，我们将更深入地介绍这些工具。不过，在此之前，先

来看看如何使用数据泵格式快速地从数据库 A 抽取数据，并移至数据库 B。这里我们将使用一个“反过来的外部表”。

外部表（external table）最早在 Oracle9i Release 1 中引入，利用外部表，我们能像读取数据库表一样读取平面文件（无格式的文本文件），完全可以用 SQL 来处理外部表。外部表是只读的，设计为从外部向 Oracle 提供数据。Oracle 10g Release 1 及以上版本中的外部表还可以走另外一条路：用于以数据泵格式从数据库获取数据，以便将数据移至另一台机器（另一个平台）。要完成这个练习，首先需要一个 DIRECTORY 对象，告诉 Oracle 卸载的位置：

```
ops$tkyte@ORA10G> create or replace directory tmp as '/tmp'

  2 /

Directory created.
```

接下来，从 ALL_OBJECTS 视图卸载数据。数据可以来自任意查询，涉及我们想要的所有表或 SQL 构造：

```
ops$tkyte@ORA10G> create table all_objects_unload

    2          organization external

    3          ( type oracle_datapump

    4           default directory TMP

    5           location( 'allobjects.dat' )

    6          )

    7          as

    8          select * from all_objects

    9          /

Table created.
```

从字面上可以很清楚地看出其含义：在/tmp 中有一个名为 allobjects.dat 的文件，其中包含查询 select * from all_objects 的内容。可以看一下这个信息：

```
ops$tkyte@ORA10G> !head /tmp/allobjects.dat

.........Linuxi386/Linux-2.0.34-8.1.0WE8ISO8859P1.........

<?xml version="1.0"?>

       <ROWSET>
```

```
<ROW>
    <STRMTABLE_T>
    <VERS_MAJOR>1</VERS_MAJOR>
    <VERS_MINOR>0 </VERS_MINOR>
    <VERS_DPAPI>3</VERS_DPAPI>
    <ENDIANNESS>0</ENDIANNESS>
    <CHARSET>WE8ISO8859P1</CHARSET>
```

这只是文件的开头，即最前面的部分；二进制数据表示为……（如果查看这个数据时你的终端发出“嘟嘟”声，不要奇怪）。下面使用二进制 FTP 传输（DMP 文件的警告同样适用！），将这个 allobject.dat 文件移至一个 Windows XP 服务器，并创建一个目录对象与之对应：

```
tkyte@ORA10G> create or replace directory TMP as 'c:\temp\'
  2 /
Directory created.
```

然后创建一个表指向这个外部表：

```
tkyte@ORA10G> create table t
  2          ( OWNER VARCHAR2(30),
  3           OBJECT_NAME VARCHAR2(30),
  4           SUBOBJECT_NAME VARCHAR2(30),
  5           OBJECT_ID NUMBER,
  6           DATA_OBJECT_ID NUMBER,
  7           OBJECT_TYPE VARCHAR2(19),
  8           CREATED DATE,
  9           LAST_DDL_TIME DATE,
 10           TIMESTAMP VARCHAR2(19),
```

```
  11         STATUS VARCHAR2(7),
  12         TEMPORARY VARCHAR2(1),
  13         GENERATED VARCHAR2(1),
  14         SECONDARY VARCHAR2(1)
  15 )
  16 organization external
  17         ( type oracle_datapump
  18         default directory TMP
  19         location( 'allobjects.dat' )
  20 )
  21 /
Table created.
```

现在就能查询从另一个数据库卸载的数据了：

```
tkyte@ORA10G> select count(*) from t;
COUNT(*)
----------
48018
```

这就是数据泵文件格式的强大之处：如果需要，它能立即通过一个“隐秘的网”将数据从一个系统传输到另一个系统。想想看，有了数据泵，下一次测试时，周末你就能把一部分数据带回家去工作了。

有一点不太明显：这两个数据库的字符集不同。如果你注意以上输出的开头部分，可以发现 Linux 数据库 WE8ISO8859P1 的字符集已经编码写入到文件中。我的 Windows 服务器则有：

```
tkyte@ORA10G> select *
  2 from nls_database_parameters
  3 where parameter = 'NLS_CHARACTERSET';
```

```
PARAMETER                      VALUE

------------------------------ ----------------

NLS_CHARACTERSET               WE8MSWIN1252
```

归功于数据泵文件格式，Oracle 现在能识别不同的字符集，并能加以处理。字符集转换会根据需要动态地完成，使得各个数据库表示中的数据“正确”。

我们还是会在第 15 章再详细讨论数据泵文件格式，不过通过这一节的介绍，你应该对数据泵文件格式是什么以及这个文件中可能包含什么有一定的认识了。

3.13 平面文件

自从有了电子数据处理，就有了平面文件（flat file）。我们每天都会看到平面文件。前面讨论的警告日志就是一个平面文件。

我在 Web 上看到有关“平面文件”的以下定义，觉得这些定义实在太绕了：

平面文件是去除了所有特定应用（程序）格式的电子记录，从而使数据元素可以迁移到其他的应用上进行处理。这种去除电子数据格式的模式可以避免因为硬件和专有软件的过时而导致数据丢失。

平面文件是一种计算机文件，所有信息都在一个信号字符串中。

实际上，平面文件只是这样一个文件，其中每一“行”都是一个“记录”，而且每行都有一些定界的文本，通常用逗号或管道符号（竖线）分隔。通过使用遗留的数据加载工具 SQLLDR 或外部表，Oracle 可以很容易地读取平面文件，实际上，我会在第 15 章详细讨论这个内容（还会在第 10 章谈到外部表）。不过，Oracle 生成平面文件可就不那么容易了，不管由于什么原因，确实没有一个简单的命令行工具能把信息导出到一个平面文件中。诸如 HTML DB 和企业管理器之类的工具有助于完成这个过程，但是并没有一个官方的命令行工具可以轻松地在脚本中用来完成这个操作。

正是出于这个原因，所以我决定在这一章对平面文件说两句，我提议能有一些生成简单平面文件的工具。多年来，为此我开发过 3 种方法，每种方法都各有特点。第一种方法是使用 PL/SQL 和 UTL_FILE（利用动态 SQL）来完成任务。如果数据量不大（几百或几千行），这个工具则有足够的灵活性，速度也不错。不过，它必须在数据库服务器主机上创建文件，但有时我们并不想在数据库服务器上创建文件。因此，我又开发了一个 SQL*Plus 实用程序，可以在运行 SQL*Plus 的机器上创建平面文件。由于 SQL*Plus 可以连接网络上任何位置的 Oracle 服务器，所以能从网络上的任何数据库把任何数据卸载到一个平面文件中。最后，如果速度要求很高，那么非 C 莫属。为此，我还开发了一个 Pro*C 命令行卸载工具来生成平面文件。这些工具都可以从 http://asktom.oracle.com/~tkyte/flat/index.html 免费得到，另外我还会在这里提供为了把数据卸载到平面文件而开发的新工具。

3.14 小结

在这一章中，我们分析了 Oracle 数据库使用的各种重要的文件类型，从底层的参数文件（如果没有参数文件，甚至无法启动数据库）到所有重要的重做日志和数据文件。我们分析了 Oracle 的存储结构，从表空间到段，再到区段，最后是数据库块（这是最小的存储单

位）。我们还简要介绍了检查点在数据库中如何工作，并提前了解了 Oracle 的一些物理进程或线程的工作。

第 4 章 内存结构

这一章将讨论 Oracle 的 3 个主要的内存结构：

- 系统全局区（System Global Area，SGA）：这是一个很大的共享内存段，几乎所有 Oracle 进程都要访问这个区中的某一点。
- 进程全局区（Process Global Area，PGA）：这是一个进程或线程专用的内存，其他进程/线程不能访问。
- 用户全局区（User Global Area，UGA）：这个内存区与特定的会话相关联。它可能在 SGA 中分配，也可能在 PGA 中分配，这取决于是用共享服务器还是用专用服务器来连接数据库。如果使用共享服务器，UGA 就在 SGA 中分配；如果使用专用服务器，UGA 就会在 PGA（即进程内存区）中。

注意 在 Oracle 的较早版本中，共享服务器称为多线程服务器（Multi-Threaded Server）或 MTS。这本书中我们会一直用“共享服务器”的说法。

下面首先讨论 PGA 和 UGA，然后再来介绍 SGA，SGA 确实是一个很庞大的结构。

4.1 进程全局区和用户全局区

进程全局区（PGA）是特定于进程的一段内存。换句话说，这是一个操作系统进程或线程专用的内存，不允许系统中的其他进程或线程访问。PGA 一般通过 C 语言的运行时调用 malloc()或 memmap()来分配，而且可以在运行时动态扩大（甚至可以收缩）。PGA 绝对不会在 Oracle 的 SGA 中分配，而总是由进程或线程在本地分配。

实际上，对你来说，用户全局区（UGA）就是你的会话的状态。你的会话总能访问这部分内存。UGA 的位置完全取决于你如何连接 Oracle。如果通过一个共享服务器连接，UGA 肯定存储在每个共享服务器进程都能访问的一个内存结构中，也就是 SGA 中。如果是这样，你的会话可以使用任何共享服务器，因为任何一个共享服务器都能读写你的会话的数据。另一方面，如果使用一个专用服务器连接，则不再需要大家都能访问你的会话状态，UGA 几乎成了 PGA 的同义词；实际上，UGA 就包含在专用服务器的 PGA 中。查看系统统计信息时可以看到，采用专用服务器模式时，总是会报告 UGA 在 PGA 中（PGA 大于或等于所用的 UGA 内存；而且 PGA 内存的大小会包括 UGA 的大小）。

所以，PGA 包含进程内存，还可能包含 UGA。PGA 内存中的其他区通常用于完成内存中的排序、位图合并以及散列。可以肯定地说，除了 UGA 内存，这些区在 PGA 中的比重最大。

从 Oracle9i Release 1 起，有两种办法来管理 PGA 中的这些非 UGA 内存：

- 手动 PGA 内存管理，采用这种方法时，你要告诉 Oracle：如果一个特定进程中需要排序或散列，允许使用多少内存来完成这些排序或散列。
- 自动 PGA 内存管理，这要求你告诉 Oracle：在系统范围内可以使用多少内存。

分配和使用内存的方式因情况不同而有很大的差异，因此，我们将分别进行讨论。需要说明，在 Oracle9i 中，如果采用共享服务器连接，就只能使用手动 PGA 内存管理。这个限制到 Oracle 10g Release 1（及以上版本）中就没有了。在 Oracle 10g Release 1 中，对于共享服务器连接，既可以使用手动 PGA 内存管理，也可以使用自动 PGA 内存管理。

PGA 内存管理受数据库初始化参数 WORKAREA_SIZE_POLICY 的控制，而且可以在会话级修改。在 Oracle9i Release 2 及以上版本中，这个初始化参数默认为 AUTO，表示自动 PGA 内存管理。而在 Oracle9i Release 1 中，这个参数的默认设置为 MANUAL。

下面几节将分别讨论这两种方法。

4.1.1 手动 PGA 内存管理

如果采用手动 PGA 内存管理，有些参数对 PGA 大小的影响最大，这是指 PGA 中除了会话为 PL/SQL 表和其他变量分配的内存以外的部分，这些参数如下：

- SORT_AREA_SIZE：在信息换出到磁盘之前，用于对信息排序的 RAM 总量。
- SORT_AREA_RETAINED_SIZE：排序完成后用于保存已排序数据的内存总量。也就是说，如果 SORT_AREA_SIZE 是 512 KB，SORT_AREA_RETAINED_SIZE 是 256 KB，那么服务器进程最初处理查询时会用 512 KB 的内存对数据排序。等到排序完成时，排序区会“收缩”为 256 KB，这 256 KB 内存中放不下的已排序数据会写出到临时表空间中。
- HASH_AREA_SIZE：服务器进程在内存中存储散列表所用的内存量。散列联结时会使用这些散列表结构，通常把一个大集合与另一个集合联结时就会用到这些结构。两个集合中较小的一个会散列到内存中，散列区中放不下的部分都会通过联结键存储在临时表空间中。

Oracle 将数据写至磁盘（或换出到磁盘）之前，数据排序或散列所用的空间量就由这些参数控制，这些参数还控制着排序完成后会保留多少内存段。SORT_AREA_SIZE～SORT_AREA_ RETAINED_SIZE 这部分内存一般从 PGA 分配，SORT_AREA_RETAINED_SIZE 这部分内存会在 UGA 中分配。通过查询一些特殊的 Oracle V$视图，可以看到 PGA 和 UGA 内存的当前使用情况，并监视其大小的变化，这些特殊的 V$视图也称为动态性能视图（dynamic performance view）。

例如，下面来运行一个小测试，这里会在一个会话中对大量数据排序，在第二个会话中，我们将监视第一个会话中 UGA/PGA 内存的使用。为了以一种可预测的方式完成这个工作，我们建立了 ALL_OBJECTS 表的一个副本，其中有大约 45 000 行，而且没有任何索引（这样就能知道肯定会发生排序）：

```
ops$tkyte@ORA10G> create table t as select * from all_objects;

Table created.



ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T' );
```

```
PL/SQL procedure successfully completed.
```

为了消除最初硬解析查询所带来的副作用，我们将运行以下脚本，不过现在先不管它的输出。后面还会在一个新会话中再次运行这个脚本，查看受控环境中对内存使用的影响。我们会依次使用大小为 64 KB、1 MB 和 1 GB 的排序区：

```
create table t as select * from all_objects;

exec dbms_stats.gather_table_stats( user, 'T' );

alter session set workarea_size_policy=manual;

alter session set sort_area_size = 65536;

set termout off

select * from t order by 1, 2, 3, 4;

set termout on

alter session set sort_area_size=1048576;

set termout off

select * from t order by 1, 2, 3, 4;

set termout on

alter session set sort_area_size=1073741820;

set termout off

select * from t order by 1, 2, 3, 4;

set termout on
```

注意 数据库中处理 SQL 时，首先必须“解析”SQL 语句，有两种类型的解析。第一种是硬解析（hard parse），数据库实例第一次解析查询时完成的就是硬解析，其中包括查询计划的生成和优化。第二种解析是软解析（soft parse），在此会跳过硬解析的许多步骤。由于对前面的查询完成了硬解析，后面再查询时就可以避免硬解析，所以在下面的操作中，我们不必测量与硬解析相关的开销（因为后面都只是软解析）。

现在，建议你注销刚才的 SQL*Plus 会话，紧接着再登录，这样能得到一个一致的环境；也就是说，相对于刚才的环境来讲，还没有做任何其他的工作。

为了确保使用手动内存管理，我们要专门设置，并指定一个很小的排序区大小（64 KB）。另外，还要标识会话 ID（SID），以便监视该会话的内存使用情况。

```
ops$tkyte@ORA10G> alter session set workarea_size_policy=manual;

Session altered.

ops$tkyte@ORA10G> select sid from v$mystat where rownum = 1;

SID

----------

151
```

下面需要在第二个单独的会话中测量第一个会话（SID 151）使用的内存。如果使用同一个会话测量自身的内存使用情况，在查询排序所用的内存时，这个查询本身可能会影响我们查看的结果。为了在第二个会话中测量内存，要使用我为此开发的一个 SQL*Plus 小脚本。实际上这是一对脚本，其中一个脚本名为 reset_stat.sql，用于重置一个小表，并将一个 SQL*Plus 变量设置为 SID，这个脚本如下：

```
drop table sess_stats;



create table sess_stats



( name varchar2(64), value number, diff number );

variable sid number

exec :sid := &1
```

注意 使用这个脚本（或任何脚本）之前，先要确保你了解脚本会做什么。这个脚本会删除一个 SESS_STATS 表，然后重新创建。如果你的模式中已经有这样一个表，你可能得换个名字！

另一个脚本是 watch_stat.sql，对于这个案例研究，脚本中使用了 MERGE SQL 语句，这样就能首先插入（INSERT）一个会话的统计值，以后再回过来对其进行更新，而无需单独的 INSERT/UPDATE 脚本：

```
merge into sess_stats

using

(

select a.name, b.value
```

```
from v$statname a, v$sesstat b
where a.statistic# = b.statistic#
        and b.sid = :sid
       and (a.name like '%ga %'
       or a.name like '%direct temp%')
) curr_stats
on (sess_stats.name = curr_stats.name)
when matched then
        update set diff = curr_stats.value - sess_stats.value,
                        value = curr_stats.value
when not matched then
        insert ( name, value, diff )
        values
        ( curr_stats.name, curr_stats.value, null )
/
select *
from sess_stats
order by name;
```

这里我强调了“对于这个案例研究”，因为上面粗体显示的行（我们感兴趣的统计名）在不同的示例中会有所不同。在这个例子中，我们感兴趣的是名字里包括 ga 的统计结果（pga 和 uga），或者名字里有 direct temp 的统计结果（在 Oracle 10g 中，这会显示对临时空间的直接读写，也就是读写临时空间所执行的 I/O 次数）。

注意 在 Oracle9i 中，对临时空间的直接 I/O 不是这样表示的。我们要使用一个 WHERE 子句，其中应包括 and (a.name like '%ga %'or a.name like '%physical % direct%')。

从 SQL*Plus 命令行运行这个 watch_stat.sql 脚本时，可以看到会话的 PGA 和 UGA 内存统计信息列表，而且列出了对临时空间执行的 I/O。在对会话 151（也就是使用手动 PGA 内存管理的会话）做任何工作之前，下面使用以上脚本来看看这个会话当前使用了多少内存，以及对临时空间执行了多少次 I/O：

```
ops$tkyte@ORA10G> @watch_stat

6 rows merged.



NAME                                          VALUE          DIFF

--------------------------------------------  ---------      ----------

physical reads direct temporary tablespace            0

physical writes direct temporary tablespace           0

session pga memory                               498252

session pga memory max                           498252

session uga memory                               152176

session uga memory max                           152176
```

可以看出，开始查询之前，UGA 中大约有 149 KB（152 176/1 024）的数据，PGA 中大约有 487 KB 的数据。第一个问题是：“在 PGA 和 UGA 之间使用了多少内存？”也就是说，用了 149 KB + 487 KB 的内存吗？还是另外的某个数？这是一个很棘手的问题，除非你了解所监视的会话（SID 为 151）通过专用服务器还是共享服务器连接数据库，否则这个问题无法回答，而且就算你知道使用的是专用服务器连接还是共享服务器连接，可能也很难得出答案。如果采用专用服务器模式，UGA 完全包含在 PGA 中，在这种情况下，进程或线程就使用 487 KB 的内存。如果使用共享服务器，UGA 将从 SGA 中分配，PGA 则在共享服务器中。所以，在共享服务器模式下，从前面的查询得到最后一行时[1]，共享服务器进程可能会由其他人使用。这个 PGA 不再是“我们的” 了，所以，从技术上讲，我们使用了 149 KB 的内存（除非正在运行查询，此时还使用了 PGA 和 UGA 之间的 487 KB 内存）。下面在会话 151 中运行第一个大查询，这个会话采用专用服务器模式，并使用手动 PGA 内存管理。需要说明，这里还是使用前面的脚本，SQL 文本完全一样，因此可以避免硬解析：

注 意 由于我们还没有设置 SORT_AREA_RETAINED_SIZE，所以报告的 SORT_AREA_RETAINED_SIZE 值将是 0，但是排序区实际保留的大小等于 SORT_AREA_SIZE。

```
ops$tkyte@ORA10G> alter session set sort_area_size = 65536;

Session altered.



ops$tkyte@ORA10G> set termout off;
```

```
query was executed here

ops$tkyte@ORA10G> set termout on;
```

现在，如果在第二个会话中再次运行脚本，会得到下面的结果。注意，这一次 session xxx memory 和 session xxx memory max 值并不一样。session xxx memory 值表示我们现在使用了多少内存。session xxx memory max 值表示会话处理查询时某个时刻所使用内存的峰值。

```
ops$tkyte@ORA10G> @watch_stat

6 rows merged.



NAME                                          VALUE          DIFF

----------------------------------------      ---------      ----------

physical reads direct temporary tablespace     2906           2906

physical writes direct temporary tablespace    2906           2906

session pga memory                            498252                0

session pga memory max                        563788           65536

session uga memory                            152176                0

session uga memory max                        217640           65464

6 rows selected.
```

可以看到，使用的内存增加了，这里对数据做了某种排序。在处理查询期间，UGA 临时从 149 KB 增加到 213 KB(增加了 64 KB)，然后再收缩回原来的大小。这是因为，为了完成查询和排序，Oracle 为会话分配了一个排序区。另外，PGA 内存从 487 KB 增加到 551 KB，增加了 64 KB。另外可以看到，我们对临时空间执行了 2 906 次读和写。

如以上结果所示，完成查询并得到结果集之前，UGA 内存又退回到原来的大小（从 UGA 释放了排序区），PGA 也会有某种程度的收缩（注意，在 Oracle8i 和以前的版本中，可能根本看不到 PGA 收缩；这是 Oracle9i 及以上版本中新增的特性）。

下面再来完成这个操作，不过这一次 SORT_AREA_SIZE 增加到 1 MB。注销所监视的会话，再登录，然后使用 reset_stat.sql 脚本从头开始。因为开始的统计结果都是一样的，所以这里不再显示，我只给出最后的结果：

```
ops$tkyte@ORA10G> alter session set sort_area_size=1048576;

Session altered.
```

```
ops$tkyte@ORA10G> set termout off;

query was executed here

ops$tkyte@ORA10G> set termout on
```

下面再在另一个会话中测量内存的使用情况：

```
ops$tkyte@ORA10G> @watch_stat

6 rows merged.

NAME                                          VALUE          DIFF

--------------------------------------------  ---------      ----------

physical reads direct temporary tablespace        684             684

physical writes direct temporary tablespace       684             684

session pga memory                             498252               0

session pga memory max                        2398796         1900544

session uga memory                             152176               0

session uga memory max                        1265064         1112888

6 rows selected.
```

可以看到，这一次处理查询期间，PGA 大幅增长。它临时增加了大约 1 728 KB，但是进行数据排序所必须执行的物理 I/O 次数则显著下降（使用更多的内存，就会减少与磁盘的交换）。而且，我们还可以避免一种多趟排序（multipass sort），如果有太多很小的排序数据集合要合并（或归并），Oracle 最后就要多次将数据写至临时空间，这种情况下就会发生多趟排序。现在，再来看一个极端情况：

```
ops$tkyte@ORA10G> alter session set sort_area_size=1073741820;

Session altered.

ops$tkyte@ORA10G> set termout off;
```

```
query was executed here

ops$tkyte@ORA10G> set termout on
```

从另一个会话进行测量，可以看到迄今为止所使用的内存：

```
ops$tkyte@ORA10G> @watch_stat

6 rows merged.



NAME                                     VALUE          DIFF

---------------------------------------- ---------- ----------

physical reads direct temporary tablespace          0          0

physical writes direct temporary tablespace          0          0

session pga memory                              498252          0

session pga memory max                         7445068    6946816

session uga memory                              152176          0

session uga memory max                         7091360    6939184

6 rows selected.
```

可以观察到，尽管允许 SORT_AREA_SIZE 有 1 GB，但实际上只用了大约 6.6 MB。这说明 SORT_AREA_SIZE 设置只是一个上界，而不是默认的分配大小。还要注意，这里也做了排序，但是这一次完全在内存中进行；而没有使用磁盘上的临时空间，从物理 I/O 次数为 0 可以看出这一点。

如果在不同版本的 Oracle 上运行这个测试，甚至在不同操作系统上运行这个测试，都可能会看到不同的行为，相信你得到的数值肯定和我得到的结果稍有差异。但是一般的行为应该是一样的。换句话说，增加允许的排序区大小并完成大规模排序时，会话使用的内存量会增加。你可能注意到 PGA 内存上上下下地变化，或者可能一段时间总保持不变（前面已经介绍过这种情况）。例如，如果你在 Oracle8i 上执行前面的测试，肯定会注意到 PGA 内存大小根本没有收缩（也就是说，无论什么情况，SESSION PGA MEMORY 都等于 SESSION PGA MEMORY MAX）。这是可以想见的，因为在 8i 中，PGA 作为堆来管理，并通过 malloc() 分配内存来创建。在 9i 和 10g 中，则使用了新的方法，会根据需要使用操作系统特有的内存分配调用来分配和释放工作区。

在使用*_AREA_SIZE 参数时，需要记住以下重要的几点：

- 这些参数控制着 SORT、HASH 和/或 BITMAP MERGE 操作所用的最大内存量。

- 一个查询可能有多个操作，这些操作可能都要使用这个内存，这样会创建多个排序/散列区。要记住，可以同时打开多个游标，每个游标都有自己的SORT_AREA_RETAINED 需求。所以，如果把排序区大小设置为 10 MB，在会话中实际上可以使用 10 MB、100 MB、1 000 MB 或更多 RAM。这些设置并非对会话的限制；它们只是对一个操作的限制。你的会话中，一个查询可以有多个排序，或者多个查询需要一个排序。

- 这些内存区都是根据需要来分配的。如果像我们一样，将排序区大小设置为 1 GB，这并不是说你要分配 1 GB 的 RAM，而只是说，你允许 Oracle 进程为一个排序/散列操作最多分配 1 GB 的内存。

4.1.2 自动 PGA 内存管理

从 Oracle9i Release 1 起，又引入了一种新的方法来管理 PGA 内存，即自动 PGA 内存管理。这种方法中不再使用 SORT_AREA_SIZE、BITMAP_MERGE_AREA_SIZE 和 HASH_AREA_SIZE 这些参数。引入自动 PGA 内存管理是为了解决以下问题：

- 易用性：很多人并不清楚如何设置适当的*_AREA_SIZE 参数。另外这些参数具体如何工作，内存究竟如何分配，这些问题都很让人困惑。

- 手动分配是一种“以一概全”的方法：一般地，随着在一个数据库上运行类似应用的用户数的增加，排序/散列所用的内存量也会线性增长。如果有 10 个并发用户，排序区大小为 1 MB，这就会使用 10 MB 的内存，100 个并发用户可能使用 100 MB，1 000 个并发用户则可能使用 1 000 MB，依此类推。除非 DBA 一直坐在控制台前不断地调整排序/散列区大小设置，否则每个人每天可能都会使用同样的设置值。考虑一下前面的例子，你自己可以清楚地看到， 随着允许使用的 RAM 量的增加，对临时空间执行的物理 I/O 在减少。如果你自己运行这个例子，肯定会注意到，随着排序可用 RAM 的增加，响应时间会减少。手动分配会把排序所用的内存量固定为某个常量值，而不论实际有多少内存。利用自动内存管理的话，只有当内存真正可用时我们才会使用；自动内存管理会根据工作负载动态地调整实际使用的内存量。

- 内存控制：根据上一条，手动分配很难保证 Oracle 实例“合法”地使用内存，甚至不可能保证。你不能控制实例要用的内存量，因为你根本无从控制会发生多少并发的排序/散列。很有可能要使用的实际内存（真正的物理空闲内存）过多，而机器上并没有这么多可用内存。

下面来看自动 PGA 内存管理。首先简单地建立 SGA 并确定其大小。SGA 是一段大小固定的内存，所以你可以准确地看到它有多大，这将是 SGA 的总大小（除非你改变了这个大小）。得到了 SGA 的大小后，再告诉 Oracle：“你就要在这么多内存中分配所有工件区，所谓工作区（work area）只是排序区和散列区的另一种通用说法。”现在，理论上讲，如果一台机器有 2 GB 的物理内存，可以分配 768 MB 内存给 SGA，768 MB 内存分配给 PGA，余下的 512 MB 内存留给操作系统和其他进程。我提到了“理论上”，这是因为实际情况不会毫厘不差，但是会很接近。为什么会这样呢？在做进一步的解释之前，先来看一下如何建立和打开自动 PGA 内存管理。

建立自动 PGA 内存管理时，需要为两个实例初始化参数确定适当的值，这两个参数是：

- WORKAREA_SIZE_POLICY：这个参数可以设置为 MANUAL 或 AUTO，如果是 MANUAL，会使用排序区和散列区大小参数来控制分配的内存量；如果是 AUTO，分配的内存量会根据数据库中的当前工作负载而变化。默认值是 AUTO，这也是推荐的设置。

- PGA_AGGREGATE_TARGET：这个参数会控制实例为完成数据排序/散列的所有工作区（即排序区和散列区）总共应分配多少内存。在不同的版本中，这个参数的默认值有所不同，可以用多种工具来设置，如 DBCA。一般来讲，如果使用自动 PGA 内存管理，就应该显式地设置这个参数。

所以，假设 WORKAREA_SIZE_POLICY 设置为 AUTO，PGA_AGGREGATE_TARGET 有一个非 0 值，就会使用这种新引入的自动 PGA 内存管理。可以在会话中通过 ALTER SESSION 命令“打开”自动 PGA 内存管理，也可以在系统级通过 ALTER SYSTEM 命令[2]打开。

注意 要记住前面的警告，在 Oracle9i 中，共享服务器连接不会使用自动 PGA 内存管理；而是使用 SORT_AREA_SIZE 和 HASH_AREA_SIZE 参数来确定为各个操作分配多少 RAM。在 Oracle 10g 及以上版本中，无论哪种连接（专用服务器连接或共享服务器连接）都能使用自动 PGA 内存管理。对于 Oracle9i，使用共享服务器连接时，要适当地设置 SORT_AREA_SIZE 和 HASH_AREA_SIZE 参数，这很重要。

所以，自动 PGA 内存管理的总目标就是尽可能充分地使用 RAM，而且不会超出可用的 RAM。倘若采用手动内存管理，这个目标几乎无法实现。如果将 SORT_AREA_SIZE 设置为 10 MB，一个用户完成一个排序操作时，该用户会用掉排序工作区的 10 MB 内存。如果 100 个用户执行同样的操作，就会用掉 1 000 MB 的内存。如果你只有 500 MB 的空闲内存，那么无论对于单独的 1 个用户还是对于 100 个用户，这种设置都是不合适的。对于单独一个完成排序的用户来说，他本来可以使用更多的内存（而不只是 10 MB），而对于 100 个想同时完成排序的用户来说，应该少用一些内存才行（应该少于 10 MB）。自动 PGA 内存管理就是要解决这种问题。如果工作负载小，随着系统上负载的增加，会最大限度地使用内存；随着更多的用户完成排序或散列操作，分配给他们的内存量会减少，这样就能达到我们的目标，一方面使用所有可用的 RAM，另一方面不要超额使用物理上不存在的内存。

1. 确定如何分配内存

有几个问题经常被问到：“内存是怎么分配的？”以及“我的会话使用了多少 RAM？”这些问题都很难回答，原因只有一个，文档中没有说明采用自动模式时分配内存的算法，而且在不同版本中这个算法还可能（而且将会）改变。只要技术以 A 开头（表示自动，automatic），你就会丧失一定的控制权，而由底层算法确定做什么以及如何进行控制。

我们可以根据 MetaLink 147806.1 中的一些信息来做一些观察：

- PGA_AGGREGATE_TARGET 是一个上限目标，而不是启动数据库时预分配的内存大小。可以把 PGA_AGGREGATE_TARGET 设置为一个超大的值（远远大于服务器上实际可用的物理内存量），你会看到，并不会因此分配很大的内存。

- 串行（非并行查询）会话会使用 PGA_AGGREGATE_TARGET 中的很少一部分，大约 5%或者更少。所以，如果把 PGA_AGGREGATE_TARGET 设置为 100 MB，可能每个工作区（例如，排序或散列工作区）只会使用大约不到 5 MB。你的会话中可能为多个查询分配有多个工作区，或者一个查询中就有多个排序/散列操作，

但是不论怎样，每个工作区只会用 PGA_AGGREGATE_TARGET 中不到 5%的内存。

- 随着服务器上工作负载的增加（可能有更多的并发查询和更多的并发用户），分配给各个工作区的 PGA 内存量会减少。数据库会努力保证所有 PGA 分配的总和不超过 PGA_AGGREGATE_TARGET 设置的阈值。这就像有一位 DBA 整天坐在控制台前，不断地根据数据库中完成的工作量来设置 SORT_AREA_SIZE 和 HASH_AREA_SIZE 参数。稍后会通过一个测试来观察这种行为。

- 一个并行查询最多可以使用 PGA_AGGREGATE_TARGET 的 30%，每个并行进程会在这 30%中得到自己的那一份。也就是说，每个并行进程能使用的内存量大约是 0.3*PGA_ AGGREGATE_TARGET / (并行进程数)。

那么，怎么观察分配给会话的不同工作区的大小呢？在介绍手动内存管理那一节中，我们介绍过一种技术，现在可以采用同样的技术来观察会话所用的内存以及对临时空间执行的 I/O。以下测试在 Red Hat Advanced Server 3.0 Linux 主机上完成，使用了 Oracle 10.1.0.3 和专用服务器连接。这是一个双 CPU 的 Dell PowerEdge，支持超线程（hyperthreading），所以就好像有 4 个 CPU 一样。这里又使用了 reset_stat.sql，并对前面的 watch_stat.sql 稍做修改，不仅要得到一个会话的会话统计信息，还将得到实例总的统计信息。这个稍做修改的 watch_stat.sql 脚本通过 MERGE 语句来得到这些信息：

```
merge into sess_stats

using

(

select a.name, b.value

from v$statname a, v$sesstat b

where a.statistic# = b.statistic#

        and b.sid = &1

        and (a.name like '%ga %'

        or a.name like '%direct temp%')

union all

select 'total: ' || a.name, sum(b.value)

from v$statname a, v$sesstat b, v$session c

where a.statistic# = b.statistic#
```

```
        and (a.name like '%ga %'
        or a.name like '%direct temp%')
        and b.sid = c.sid
        and c.username is not null
group by 'total: ' || a.name
) curr_stats
on (sess_stats.name = curr_stats.name)
when matched then
        update set diff = curr_stats.value - sess_stats.value,
        value = curr_stats.value
when not matched then
        insert ( name, value, diff )
        values
        ( curr_stats.name, curr_stats.value, null )
/
```

除了单个会话的统计信息外，我只增加了 UNION ALL 部分，通过将所有会话的统计结果累加，从而得到总的 PGA/UGA 使用情况和排序写次数。然后在这个会话中运行以下 SQL*Plus 脚本。在此之前已经创建了 BIG_TABLE 表，而且填入了 50 000 行记录。我删除了这个表的主键，这样余下的只是表本身（从而确保肯定会执行一个排序过程）：

```
set autotrace traceonly statistics;
select * from big_table order by 1, 2, 3, 4;
set autotrace off
```

注意 BIG_TABLE 表创建为 ALL_OBJECTS 的一个副本，并增加了主键，行数由你来定。big_table.sql 脚本在本书开头的“配置环境”一节中介绍过。

下面，对一个数据库运行这个小查询脚本，该数据库的 PGA_AGGREGATE_TARGET 设置为 256 MB，这说明我希望 Oracle 使用最多约 256 MB 的 PGA 内存来完成排序。我还建立了另一个脚本，可以在其他会话中运行，它会在机器上生成很大的排序负载。这个脚本有一个循环，并使用一个内置的包（DBMS_ALERT）查看是否继续处理。如果是，则再一次

运行这个大查询，对整个 BIG_TABLE 表排序。仿真结束后，再由一个会话通知所有排序进程（也就是负载生成器）“停止”并退出。执行排序的脚本如下：

```
declare
        l_msg long;
        l_status number;
begin
        dbms_alert.register( 'WAITING' );
        for i in 1 .. 999999 loop
                dbms_application_info.set_client_info( i );
                dbms_alert.waitone( 'WAITING', l_msg, l_status, 0 );
                exit when l_status = 0;
                for x in ( select * from big_table order by 1, 2, 3, 4 )
                loop
                        null;
                end loop;
        end loop;
end;
/
exit
```

以下脚本会让这些进程停止运行：

```
begin
        dbms_alert.signal( 'WAITING', '' );
        commit;
end;
```

为了观察对所测量的会话分配的 RAM 量有什么不同，首先独立地运行 SELECT 查询，

这样就只有一个会话。我得到了与前面相同的 6 个统计结果，并把这些统计结果连同活动会话数[3]保存在另一个表中。然后，再向系统增加 25 个会话（也就是说，在 25 个新会话中运行以上带循环的基准测试脚本）。接下来等待很短时间（1 分钟），让系统能针对这个新负载做出调整。然后我创建了一个新会话，用 reset_stat.sql 得到其统计结果，再运行执行排序的查询，接下来运行 watch_stat.sql 得到排序前后的统计结果之差。然后重复地做了这个工作，直至并发用户数达到 500。

需要说明，这里实际上在要求数据库实例做它根本做不了的事情。前面已经提到，第一次运行 watch_stat.sql 时，每个 Oracle 连接在完成排序之前会使用大约 0.5 MB 的 RAM。如果有 500 个并发用户全部登录，单单是他们登录所用的内存就已经非常接近所设置的 PGA_AGGREGATE_TARGET（PGA_AGGREGATE_TARGET 设置为 256 MB），更不用说具体做工作了！由此再一次表明，PGA_AGGREGATE_TARGET 只是一个目标，而不是明确地指定要分配多少空间。出于很多原因，实际分配的空间还可能超过这个值。

表 4-1 总结了每次增加大约 25 个用户时得到的统计结果。

表 4-1 随着活动会话数的增加，PGA 内存分配行为的变化（PGA_AGGREGATE_TARGET 设置为 256 MB）

活动会话	一个会话使用的 PGA	系统使用的 PGA	一个会话的临时写	一个会话的临时读
1	7.5	2	0	0
27	7.5	189	0	0
51	4.0	330	728	728
76	4.0	341	728	728
101	3.2	266	728	728
126	1.5	214	728	728
151	1.7	226	728	728
177	1.4	213	728	728
201	1.3	218	728	728

8				
226	1.3	211	728	72
8				
251	1.3	237	728	72
8				
276	1.3	251	728	72
8				
301	1.3	281	728	72
8				
326	1.3	302	728	72
8				
351	1.3	324	728	72
8				
376	1.3	350	728	72
8				
402	1.3	367	728	72
8				
426	1.3	392	728	72
8				
452	1.3	417	728	72
8				
476	1.3	439	728	72
8				
501	1.3	467	728	72
8				

注意 你可能会奇怪，有 1 个活动用户时，为什么系统使用的 RAM 只报告为 2 MB。这与我的测量方法有关。这个仿真会对所测会话中的统计结果建立快照。接下来，我会在所测的这个会话中运行上述大查询，然后再次记录该会话统计结果的快照。最后再来测量系统使用了多少 PGA。在我测量所用的 PGA 时，这个会话已经结束，并交回了它用于排序的一些 PGA。所以，这里系统使用的 PGA 只是对测量时系统所用 PGA 内存的准确度量。

可以看到，如果活动会话不多，排序则完全在内存中执行。活动会话数在 1～50 之间

时，我就可以完全在内存中执行排序。不过，等到有 50 个用户登录并执行排序时，数据库就会开始控制一次能使用的内存量。所用的 PGA 量要退回到可接受的限值（256 MB）以内，在此之前需要几分钟的时间来调整，不过最后总是会落回到阈值范围内。分配给会话的 PGA 内存量从 7.5 MB 降到 4 MB，随后又降到 3.2 MB，最后降至 1.7～1.3 MB 之间（要记住，PGA 中有一部分不用于排序，也不用于其他操作，光是登录这个动作就要创建 0.5 MB 的 PGA）。系统使用的总 PGA 量仍保持在可以接受的限值内，直至用户数达到 300～351 之间。从这里开始，系统使用的 PGA 开始有规律地超过 PGA_AGGREGATE_TARGET，并延续至测试结束。在这个例子中，我交给数据库实例一个不可能完成的任务，光是支持 350 个用户（大多数都执行一个 PL/SQL，再加上他们都要请求排序），我设定的这个目标（256 MB 的 RAM）就无法胜任。每个会话只能使用尽可能少的内存，但另一方面又必须分配所需的足够内存，所以这根本就办不到。等我完成这个测试时，500 个活动会话已经使用了总共 467 MB 的 PGA 内存，大大超出了我所设定的目标（256 MB），但对每个会话来说使用的内存已经够少的了。

不过，再想想在手动内存管理情况下表 4-1 会是什么样子。假设 SORT_AREA_SIZE 设置为 5 MB。计算很简单：每个会话都能在 RAM 中执行排序（如果实际 RAM 用完了，还可以使用虚拟内存），这样每个会话会使用 6～7 MB 的 RAM（与前面只有一个用户而且不在磁盘上排序时使用的内存量相当）。再运行前面的测试，将 SORT_AREA_SIZE 设置为 5 MB，从 1 个用户开始，每次增加 25 个用户，得到的结果保持一致，如表 4-2 所示。

表 4-2 随着活动会话数的增加，PGA 内存分配行为的变化（手动内存管理，SORT_AREA_SIZE 设置为 5 MB）

活动会话	一个会话使用的 PGA	系统使用的 PGA	一个会话的临时写	一个会话的临时读
1	6.4	5	728	728
26	6.4	137	728	728
51	6.4	283	728	728
76	6.4	391	728	728
102	6.4	574	728	728
126	6.4	674	728	728
151	6.4	758	728	728

176	6.4	987	728	728
202	6.4	995	728	728
226	6.4	1227	728	728
251	6.4	1383	728	728
277	6.4	1475	728	728
302	6.4	1548	728	728

如果我能完成这个测试（这个服务器上有 2 GB 的实际内存，我的 SGA 是 600 MB；等到用户数达到 325 时，机器换页和交换开始过于频繁，而无法继续工作），有 500 个并发用户时，我就要分配大约 2 750 MB 的 RAM！所以，在这个系统上 DBA 可能不会将 SORT_AREA_SIZE 设置为 5 MB，而是设置为 0.5 MB，力图使高峰期的最大 PGA 使用量在可以忍受的范围内。现在如果并发用户数为 500，就要分配大约 500 MB 的 PGA，这可能与采用自动内存管理时所观察到的结果类似，但是即使用户不太多，还是会写临时空间，而不是在内存中执行排序。实际上，如果 SORT_AREA_SIZE 设置为 0.5 MB，再运行以上测试，会观察到表 4-3 所示的数据。

表 4-3 随着活动会话数的增加，PGA 内存分配行为的变化（手动内存管理，SORT_AREA_SIZE 设置为 0.5 MB）

活动会话	一个会话使用的 PGA	系统使用的 PGA	一个会话的临时写	一个会话的临时读
1	1.2	1	728	728
26	1.2	29	728	728
51	1.2	57	728	728
76	1.2	84	728	728
101	1.2	112	728	728

8

126	1.2	140	728	72
8				
151	1.2	167	728	72
8				
176	1.2	194	728	72
8				
201	1.2	222	728	72
8				
226	1.2	250	728	72
8				

工作负载随着时间的推移而增加或减少时，这种内存的使用完全可以预计，但是并不理想。自动 PGA 内存管理正是为此设计的。在有足够的内存时，自动内存管理会让少量的用户尽可能多地使用 RAM，而过一段时间负载增加时，可以减少分配，再过一段时间，随着负载的减少，为每个操作分配的 RAM 量又能增加。

2. 使用 PGA_AGGREGATE_TARGET 控制内存分配

之前我曾说过，“理论上”可以使用 PGA_AGGREGATE_TARGET 来控制实例使用的 PGA 内存的总量。不过，从上一个例子中可以看到，这并不是一个硬性限制。实例会尽力保持在 PGA_AGGREGATE_TARGET 限制以内，但是如果实在无法保证，它也不会停止处理；只是要求超过这个阈值。

这个限制只是一个“理论上”的限制，对此还有一个原因：尽管工作区在 PGA 内存中所占的比重很大，但 PGA 内存中并非只有工作区。PGA 内存分配涉及很多方面，其中只有工作区在数据库实例的控制之下。如果创建并执行一个 PL/SQL 代码块将数据填入一个很大的数组，这里采用专用服务器模式，因此 UGA 在 PGA 中，倘若是这样，Oracle 只能任由你这样做，而无法干涉。

考虑下面这个小例子。我们将创建一个包，其中可以保存服务器中的一些持久（全局）数据：

```
ops$tkyte@ORA10G> create or replace package demo_pkg
  2 as
  3 type array is table of char(2000) index by binary_integer;
  4 g_data array;
  5 end;
```

```
  6 /

Package created.
```

下面，测量这个会话当前使用的 PGA/UGA 内存量（这个例子使用了专用服务器，所以 UGA 在 PGA 内存中，是 PGA 的一个子集）：

```
ops$tkyte@ORA10G> select a.name, to_char(b.value, '999,999,999') value
  2 from v$statname a, v$mystat b
  3 where a.statistic# = b.statistic#
  4 and a.name like '%ga memory%';

NAME                           VALUE
------------------------------ ------------
session uga memory             1,212,872
session uga memory max         1,212,872
session pga memory             1,677,900
session pga memory max         1,677,900
```

所以，最初会话中使用了大约 1.5 MB 的 PGA 内存（因为还要编译 PL/SQL 包，运行这个查询，等等）。现在，再对 BIG_TABLE 运行这个查询，这里 PGA_AGGREGATE_TARGET 同样是 256 MB（这一回是在一个空闲的实例上执行查询；现在我们是惟一需要内存的会话）：

```
ops$tkyte@ORA10GR1> set autotrace traceonly statistics;
ops$tkyte@ORA10GR1> select * from big_table order by 1,2,3,4;
50000 rows selected.


Statistics
----------------------------------------------------------
          0  recursive calls
```

```
          0  db block gets
        721  consistent gets
          0  physical reads
          0  redo size
2644246  bytes sent via SQL*Net to client
      37171  bytes received via SQL*Net from client
       3335  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      50000  rows processed
ops$tkyte@ORA10GR1> set autotrace off
```

可以看到，排序完全在内存中完成，实际上，如果看一下这个会话的 PGA/UGA 使用情况，就能看出我们用了多少 PGA/UGA 内存：

```
ops$tkyte@ORA10GR1> select a.name, to_char(b.value, '999,999,999') value
  2 from v$statname a, v$mystat b
  3 where a.statistic# = b.statistic#
  4          and a.name like '%ga memory%';
NAME                                VALUE
------------------------------  ------------
session uga memory                1,212,872
session uga memory max            7,418,680
session pga memory                1,612,364
session pga memory max 7,838,284
```

还是前面观察到的 7.5 MB 的 RAM。现在，再填入包中的 CHAR 数组（CHAR 数据类型用空格填充，这样每个数组元素的长度都正好是 2 000 个字符）：

```
ops$tkyte@ORA10G> begin
  2 for i in 1 .. 100000
  3 loop
  4          demo_pkg.g_data(i) := 'x';
  5 end loop;
  6 end;
  7 /
PL/SQL procedure successfully completed.
```

在此之后，测量会话当前使用的 PGA，可以看到下面的结果：

```
ops$tkyte@ORA10GR1> select a.name, to_char(b.value, '999,999,999') value
    2 from v$statname a, v$mystat b
    3 where a.statistic# = b.statistic#
    4          and a.name like '%ga memory%';

NAME                                VALUE
------------------------------      ------------
session uga memory 312,952,440
session uga memory max     312,952,440
session pga memory 313,694,796
session pga memory max     313,694,796
```

现在，数据库本身无法控制 PGA 中分配的这些内存。我们已经超过了 PGA_AGGREGATE_TARGET，但数据库对此无计可施，如果它能干预的话，肯定会拒绝我们的请求，不过只有当操作系统报告称再没有更多可用内存时我们的请求才会失败。如果想试试看，你可 以在数组中分配更多的空间，再向其中放入更多的数据，等到操作系统报告再无内存可用时，数据库就会“忍无可忍”地拒绝内存分配请求。

不过，数据库很清楚我们做了什么。尽管有些内存无法控制，但它不会忽略这部分内存；而是会识别已经使用的内存，并相应地减少为工作区分配的内存大小。所以，如果再运

行同样的排序查询，可以看到，这一次会在磁盘上排序，如果要在内存中排序，这需要 7 MB 左右的 RAM，但是数据库没有提供这么多 RAM，原因是分配的内存已经超过了 PGA_AGGREGATE_TARGET：

```
ops$tkyte@ORA10GR1> set autotrace traceonly statistics;

ops$tkyte@ORA10GR1> select * from big_table order by 1,2,3,4;

50000 rows selected.



Statistics

----------------------------------------------------------

          6  recursive calls

          2  db block gets

        721  consistent gets

        728  physical reads

          0  redo size

    2644246  bytes sent via SQL*Net to client

      37171  bytes received via SQL*Net from client

       3335  SQL*Net roundtrips to/from client

          0  sorts (memory)

          1  sorts (disk)

      50000  rows processed

ops$tkyte@ORA10GR1> set autotrace off
```

因此，由于一些 PGA 内存不在 Oracle 的控制之下，所以如果在 PL/SQL 代码中分配了大量很大的数据结构，就很容易超出 PGA_AGGREGATE_TARGET。在此并不是建议你绝对不要这样做，我只是想说明 PGA_AGGREGATE_TARGET 不能算是一个硬性限制，而更应该算是一个请求。

4.1.3 手动和自动内存管理的选择

那么，你要用哪种方法呢？手动还是自动？默认情况下，我倾向于自动 PGA 内存管理。

警告 在这本书里我一而再、再而三地提醒你：不要对生产系统（实际系统）做任何修改，除非先测试修改有没有副作用。例如，先别阅读这一章，检查你的系统，看看是不是在使用手动内存管理，然后再打开自动内存管理。查询计划可能改变，而且也许还会影响性能。可能会发生以下 3 种情况之一：

- 运行得完全一样。
- 比以前运行得好。
- 比以前运行得差。

在做出修改之前一定要谨慎，应当先对要做的修改进行测试。

最让 DBA 头疼的一件事可能就是设置各个参数，特别是像 SORT|HASH_AREA_SIZE 之类的参数。系统运行时这些参数的值可能设置得相当小，而且实在是太小了，以至于性能受到了负面影响，这种情况我已经屡见不鲜。造成这种情况的原因可能是默认值本身就非常小：排序区的默认大小为 64 KB，散列区的默认大小也只是 128 KB。这些值应该多大才合适呢？对此人们总是很困惑。不仅如此，在一天中的不同时段，你可能还想使用不同的值。早上 8:00 只有两个用户，此时登录的每个用户使用 50 MB 大小的排序区可能很合适。不过，到中午 12:00，已经有 500 个用户，再让每个用户使用 50 MB 的排序区就不合适了。针对这种情况，WORKAREA_SIZE_POLICY = AUTO 和相应的 PGA_AGGREGATE_TARGET 就能派上用场了。要设置 PGA_AGGREGATE_TARGET，也就是你希望 Oracle 能自由使用多大的内存来完成排序和散列，从概念上讲这比得出最佳的 SORT|HASH_AREA_SIZE 要容易得多，特别是，SORT|HASH_AREA_SIZE 之类的参数并没有一个最佳的值；“最佳值”会随工作负载而变化。

从历史上看，DBA 都是通过设置 SGA（缓冲区缓存、日志缓冲区、共享池、大池和 Java 池）的大小来配置 Oracle 使用的内存量。机器上余下的内存则由 PGA 区中的专用或共享服务器使用。对于会使用（或不会使用）其中的多少内存，DBA 无从控制。不错，DBA 确实能设置 SORT_AREA_SIZE，但是如果有 10 个并发的排序，Oracle 就会使用 10 * SORT_AREA_SIZE 字节的 RAM。如果有 100 个并发的排序，Oracle 将使用 100 * SORT_AREA_SIZE 字节；倘若是 1 000 个并发的排序，则会使用 1 000 *SORT_AREA_SIZE；依此类推。不仅如此，再加上 PGA 中还有其他内容，你根本不能很好地控制系统上最多能使用的 PGA 内存量。

你所希望的可能是：随着系统上内存需求的增加和减少，会使用不同大小的内存。用户越多，每个用户使用的 RAM 就越少。用户越少，每个用户能使用的 RAM 则越多。设置 WORKAREA_SIZE_POLICY = AUTO 就是要达到这个目的。现在 DBA 只指定一个大小值，即 PGA_AGGREGATE_TARGET，也就是数据库应当努力使用的最大 PGA 内存量。Oracle 会根据情况将这个内存适当地分配给活动会话。另外，在 Oracle9i Release 2 及以上版本中，甚至还有一个 PGA 顾问（PGA advisory），这是 Statspack 的一部分，可以通过一个 V$动态性能视图得到，也可以在企业管理器（EM）中看到，PGA 顾问与缓冲区缓存顾问很相似。它会一直告诉你，为了尽量减少对临时表空间执行的物理 I/O，系统最优的 PGA_AGGREGATE_TARGET 是什么。可以使用这个信息动态修改 PGA 大小（如果你有足够多的 RAM），或者确定是否需要服务器上的更多 RAM 来得到最优性能。

不过，有没有可能不想使用自动 PGA 内存管理的情况呢？当然有，好在不想用的情况只是例外，而不是一般现象。自动内存管理力图对多个用户做到“公平”。由于预见到可能

有另外的用户加入系统，因此自动内存管理会限制分配的内存量只是PGA_AGGREGATE_TARGET 的一部分。但是假如你不想要公平，确实知道应该得到所有可用的内存（而不是其中的一部分），这该怎么办？倘若如此，就应该使用 ALTER SESSION 命令在你的会话中禁用自动内存管理（而不影响其他会话），并且根据需要，手动地设置你的 SORT|HASH_AREA_SIZE。例如，凌晨 2:00 要做一个大型的批处理，它要完成大规模的散列联结、建立索引等工作，对于这样一个批处理作业，你要怎么做？它应该可以使用机器上的所有资源[5]。在内存使用方面，它不想“公平”，而是全部都想要，因为它知道，现在数据库中除了它以外再没有别的任务了。当然这个批处理作业可以发出 ALTER SESSION 命令，充分使用所有可用的资源。

所以，简单地讲，对于成天在数据库上运行的应用，我倾向于对最终用户会话使用自动 PGA 内存管理。手动内存管理则适用于大型批处理作业（它们在特殊的时段运行，此时它们是数据库中惟一的活动）。

4.1.4 PGA 和 UGA 小结

以上讨论了两种内存结构：PGA 和 UGA。你现在应该了解到，PGA 是进程专用的内存区。这是 Oracle 专用或共享服务器需要的一组独立于会话的变量。PGA 是一个内存“堆”，其中还可以分配其他结构。UGA 也是一个内存堆，其中定义不同会话特有的结构。如果使用专用服务器来连接 Oracle，UGA 会从 PGA 分配，如果使用共享服务器连接，UGA 则从 SGA 分配。这说明，使用共享服务器时，必须适当地设置 SGA 中大池（large pool）的大小，以便有足够的空间来适应可能并发地连接数据库的每一个用户。所以，如果数据库支持共享服务器连接，与有类似配置但只使用专用服务器模式的数据库相比，前者的 SGA 通常比后者大得多。下面将更详细地讨论 SGA。

4.2 系统全局区

每个 Oracle 实例都有一个很大的内存结构，称为系统全局区（System Global Area，SGA）。这是一个庞大的共享内存结构，每个 Oracle 进程都会访问其中的某一点。SGA 的大小不一，在小的测试系统上只有几 MB，在中到大型系统上可能有几百 MB，对于非常大的系统，甚至多达几 GB。

在 UNIX 操作系统上，SGA 是一个物理实体，从操作系统命令行上能“看到”它。它物理地实现为一个共享内存段，进程可以附加到这段独立的内存上。系统上也可以只有 SGA 而没有任何 Oracle 进程；只有内存而已。不过，需要说明，如果有一个 SGA 而没有任何 Oracle 进程，这就说明数据库以某种方式崩溃了。这是一种很罕见的情况，但是确实有可能发生。以下是 Red Hat Linux 上 SGA 的“样子”：

```
[tkyte@localhost tkyte]$ ipcs -m | grep ora

0x99875060 2031619 ora10g 660 538968064 15

0x0d998a20 1966088 ora9ir2 660 117440512 45

0x6b390abc 1998857 ora9ir1 660 130560000 50
```

这里表示了 3 个 SGA：一个属于操作系统用户 ora10g，另一个属于操作系统用户 ora9ir2，第 3 个属于操作系统用户 ora9ir1，大小分别约 512 MB、112 MB 和 124 MB。

在 Windows 上，则无法像 UNIX/Linux 上那样把 SGA 看作一个实体。由于在 Windows 平台上， Oracle 会作为有一个地址空间的单个进程来执行，所以 SGA 将作为专用（私有）内存分配给 oracle.exe 进程。如果使用 Windows Task Manager（任务管理器）或其他性能工具，则可以看到 oracle.exe 总共分配了多少空间，但是 SGA 和其他已分配的内存无法看到。

在 Oracle 自身内，则完全可以看到 SGA，而不论平台是什么。为此，只需使用另一个神奇的 V$视图，名为 V$SGASTAT。它可能如下所示（注意，这个代码并非来自前面的系统；而是来自一个已经适当地配置了相应特性的系统，从而可以查看所有可用的池）：

```
ops$tkyte@ORA10G> compute sum of bytes on pool

ops$tkyte@ORA10G> break on pool skip 1

ops$tkyte@ORA10G> select pool, name, bytes

  2 from v$sgastat

  3 order by pool, name;


POOL                 NAME                           BYTES

------------         ------------------------------ ----------

java pool            free memory                  16777216

************                                        ----------

sum                                               16777216

large pool           PX msg pool                     64000

                     free memory                  16713216

************                                        ----------

sum                                               16777216

shared pool          ASH buffers                  2097152

                     FileOpenBlock                 746704

                     KGLS heap                     777516

                     KQR L SO                       29696
```

```
                         KQR M PO                  599576
                         KQR M SO                   42496
...
                         sql area                 2664728
                         table definiti               280
                         trigger defini              1792
                         trigger inform              1944
                         trigger source               640
                         type object de            183804
**********                                         ----------
sum                                               352321536
streams pool             free memory              33554432
**********                                         ----------
sum                                                33554432
                         buffer_cache            1157627904
                         fixed_sga                  779316
                         log_buffer                 262144
**********                                         ----------
sum                                               1158669364
43 rows selected.
```

SGA 分为不同的池（pool）：

- Java 池（Java pool）：Java 池是为数据库中运行的 JVM 分配的一段固定大小的内存。在 Oracle10g 中，Java 池可以在数据库启动并运行时在线调整大小。
- 大池（Large pool）：共享服务器连接使用大池作为会话内存，并行执行特性使用大池作为消息缓冲区，另外 RMAN 备份可能使用大池作为磁盘 I/O 缓冲区。在 Oracle 10g 和 9i Release 2 中，大池都可以在线调整大小。

- 共享池（Shared pool）：共享池包含共享游标（cursor）、存储过程、状态对象、字典缓存和诸如此类的大量其他数据。在 Oracle 10g 和 9i 中，共享池都可以在线调整大小。
- 流池（Stream pool）：这是 Oracle 流（Stream）专用的一个内存池，Oracle 流是数据库中的一个数据共享工具。这个工具是 Oracle 10g 中新增的，可以在线调整大小。如果未配置流池，但是使用了流功能，Oracle 会使用共享池中至多 10%的空间作为流内存。
- “空”池（“Null”pool）：这个池其实没有名字。这是块缓冲区（缓存的数据库块）、重做日志缓冲区和“固定 SGA”区专用的内存。

典型的 SGA 可能如图 4-1 所示。

图 4-1 典型的 SGA

对 SGA 整体大小影响最大的参数如下：

- JAVA_POOL_SIZE：控制 Java 池的大小。
- SHARED_POOL_SIZE：在某种程度上控制共享池的大小。
- LARGE_POOL_SIZE：控制大池的大小。
- DB_*_CACHE_SIZE：共有 8 个 CACHE_SIZE 参数，控制各个可用的缓冲区缓存的大小。
- LOG_BUFFER：在某种程度上控制重做缓冲区的大小。
- SGA_TARGET：Oracle 10g 及以上版本中用于自动 SGA 内存管理。
- SGA_MAX_SIZE：用于控制数据库启动并运行时 SGA 可以达到的最大大小。

在 Oracle9i 中，各个 SGA 组件必须由 DBA 手动地设置大小，但是从 Oracle 10g 开始，又有了一个新的选择：自动 SGA 内存管理。如果采用自动 SGA 内存管理，数据库实例会根据工作负载条件在运行时分配和撤销（释放）各个 SGA 组件。在 Oracle 10g 中使用自动 SGA 内存管理时，只需把 SGA_TARGET 参数设置为所需的 SGA 大小，其他与 SGA 相关的参数都不用管。只要设置了 SGA_TARGET 参数，数据库实例就会接管工作，根据需要为各个池分配内存，随着时间的推移，甚至还会从一个池取出内存交给另一个池。

不论是使用自动内存管理还是手动内存管理，都会发现各个池的内存以一种称为颗粒

（granule，也称区组）的单位来分配。一个颗粒是大小为 4 MB、8 MB 或 16 MB 的内存区。颗粒是最小的分配单位，所以如果想要一个 5 MB 的 Java 池，而且颗粒大小为 4 MB，Oracle 实际上会为这个 Java 池分配 8 MB（在 4 的倍数中，8 是大于或等于 5 的最小的数）。颗粒的大小由 SGA 的大小确定（听上去好像又转回来了，因为 SGA 的大小取决于颗粒的大小）。通过查询 V$SGA_DYNAMIC_ COMPONENTS，可以查看各个池所用的颗粒大小。实际上，还可以使用这个视图来查看 SGA 的总大小如何影响颗粒的大小：

```
sys@ORA10G> show parameter sga_target

NAME                                 TYPE                         VALUE
------------------------------------ ----------- ------------------------------
sga_target                           big integer                  576M

sys@ORA10G> select component, granule_size from v$sga_dynamic_components;

COMPONENT                                GRANULE_SIZE
------------------------                              ------------
shared pool                                           4194304
large pool                                            4194304
java pool                                             4194304
streams pool                                          4194304
DEFAULT buffer cache                                  4194304
KEEP buffer cache                                     4194304
RECYCLE buffer cache                                  4194304
DEFAULT 2K buffer cache                               4194304
DEFAULT 4K buffer cache                               4194304
DEFAULT 8K buffer cache                               4194304
DEFAULT 16K buffer cache                              4194304
```

```
DEFAULT 32K buffer cache                4194304

OSM Buffer Cache                        4194304

13 rows selected.
```

在这个例子中，我使用了自动 SGA 内存管理，并通过一个参数（SGA_TARGET）来控制 SGA 的大小。SGA 小于 1 GB 时，颗粒为 4 MB。当 SGA 大小增加到超过阈值 1 GB 时（对于不同的操作系统，甚至对于不同的版本，这个阈值可能稍有变化），可以看到颗粒大小有所增加：

```
sys@ORA10G> alter system set sga_target = 1512m scope=spfile;

System altered.



sys@ORA10G> startup force

ORACLE instance started.



Total System Global Area 1593835520 bytes

Fixed Size                  779316 bytes

Variable Size             401611724 bytes

Database Buffers         1191182336 bytes

Redo Buffers                262144 bytes

Database mounted.

Database opened.

sys@ORA10G> select component, granule_size from v$sga_dynamic_components;



COMPONENT            GRANULE_SIZE

-------------------------       ------------

shared pool                  16777216
```

```
large pool                      16777216
java pool                       16777216
streams pool                    16777216
DEFAULT buffer cache            16777216
KEEP buffer cache               16777216
RECYCLE buffer cache            16777216
DEFAULT 2K buffer cache         16777216
DEFAULT 4K buffer cache         16777216
DEFAULT 8K buffer cache         16777216
DEFAULT 16K buffer cache 16777216
DEFAULT 32K buffer cache 16777216
OSM Buffer Cache                16777216
13 rows selected.
```

可以看到，SGA 为 1.5 GB 时，会以 16 MB 的颗粒为池分配空间，所以池大小都将是 16 MB 的某个倍数。

记住这一点，下面逐一分析各个主要的 SGA 组件。

4.2.1 固定 SGA

固定 SGA（fixed SGA）是 SGA 的一个组件，其大小因平台和版本而异。安装时，固定 SGA 会“编译到”Oracle 二进制可执行文件本身当中（所以它的名字里有“固定”一词）。在固定 SGA 中，有一组指向 SGA 中其他组件的变量，还有一些变量中包含了各个参数的值。我们无法控制固定 SGA 的大小，不过固定 SGA 通常都很小。可以把这个区想成是 SGA 中的“自启”区，Oracle 在内部要使用这个区来找到 SGA 的其他区。

4.2.2 重做缓冲区

如果数据需要写到在线重做日志中，则在写至磁盘之前要在重做缓冲区（redo buffer）中临时缓存这些数据。由于内存到内存的传输比内存到磁盘的传输快得多，因此使用重做日志缓冲区可以加快数据库的操作。数据在重做缓冲区里停留的时间不会太长。实际上，LGWR 会在以下某个情况发生时启动对这个区的刷新输出（flush）：

- ☐ 每 3 秒一次
- ☐ 无论何时有人提交请求

- 要求 LGWR 切换日志文件
- 重做缓冲区 1/3 满，或者包含了 1 MB 的缓存重做日志数据

由于这些原因，如果重做缓冲区的大小超过几 MB，通常对系统就没有什么意义了，实际上，能从这么大的重做缓冲区得到好处的系统极为少见。如果是一个有大量并发事务的大型系统，也许大的重做日志缓冲区会对它有利，因为 LGWR（这个进程负责将重做日志缓冲区刷新输出到磁盘）写日志缓冲区的一部分时，其他会话可能会在缓冲区中填入新的数据。一般而言，如果一个事务长时间运行，就会生成大量重做日志，倘若采用更大的日志缓冲区而不是正常的日志缓冲区，对这种事务最有好处。因为在 LGWR 忙于将部分重做日志写出到磁盘时，重做日志缓冲区还会继续填入日志。事务越大、越长，大日志缓冲区的好处就越显著。

重做缓冲区的默认大小由 LOG_BUFFER 参数控制，取值为 512 KB 和(128 * CPU 个数)KB 中的较大者。这个区的最小大小取决于操作系统。如果想知道到底是多少，只需将 LOG_BUFFER 设置为 1 字节，再重启数据库。例如，在我的 Red Hat Linux 实例上，可以看到以下输出：

```
sys@ORA10G> alter system set log_buffer=1 scope=spfile;

System altered.



sys@ORA10G> startup force

ORACLE instance started.



Total System Global Area     1593835520 bytes

Fixed Size                        779316 bytes

Variable Size                  401611724 bytes

Database Buffers              1191182336 bytes

Redo Buffers                      262144 bytes

Database mounted.

Database opened.

sys@ORA10G> show parameter log_buffer

NAME                                 TYPE                    VALUE
```

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

log_buffer                       integer                 262144
```

在这个系统上，可能的最小日志缓冲区就是 256 KB（而不论我设置的是多少）。

4.2.3 块缓冲区缓存

到目前为止，我们已经介绍了 SGA 中一些相对较小的组件。下面再来看看可能比较大的组件。Oracle 将数据库块写至磁盘之前，另外从磁盘读取数据库块之后，就会把这些数据库块存储在块缓冲区缓存（block buffer cache）中。对我们来说，这是 SGA 中一个很重要的区。如果太小，我们的查询就会永远也运行不完。如果太大，又会让其他进程饥饿（例如，没有为专用服务器留下足够的空间来创建其 PGA，甚至无法启动）。

在 Oracle 的较早版本中，只有一个块缓冲区缓存，所有段的所有块都放在这个区中。从 Oracle 8.0 开始，可以把 SGA 中各个段的已缓存块放在 3 个位置上：

- 默认池（default pool）：所有段块一般都在这个池中缓存。这就是原先的缓冲区池（原来也只有一个缓冲区池）。
- 保持池（keep pool）：按惯例，访问相当频繁的段会放在这个候选的缓冲区池中，如果把这些段放在默认缓冲区池中，尽管会频繁访问，但仍有可能因为其他段需要空间而老化（aging）。
- 回收池（recycle pool）：按惯例，访问很随机的大段可以放在这个候选的缓冲区池中，这些块会导致过量的缓冲区刷新输出，而且不会带来任何好处，因为等你想要再用这个块时，它可能已经老化退出了缓存。要把这些段与默认池和保持池中的段分开，这样就不会导致默认池和保持池中的块老化而退出缓存。

需要注意，在保持池和回收池的描述中，我用了一个说法“按惯例”。因为你完全有可能不按上面描述的方式使用保持池或回收池，这是无法保证的。实际上，这 3 个池会以大体相同的方式管理块；将块老化或缓存的算法并没有根本的差异。这样做的目标是让 DBA 能把段聚集到“热”区（hot）、“温”区（warm）和“不适合缓存”区（do not care to cache）。理论上讲，默认池中的对象应该足够热（也就是说，用得足够多），可以保证一直呆在缓存中。缓存会把它们一直留在内存中，因为它们是非常热门的块。可能还有 一些段相当热门，但是并不太热；这些块就作为温块。这些段的块可以从缓存刷新输出，为不常用的一些块（“不适合缓存”块）腾出空间。为了保持这些温段的块 得到缓存，可以采取下面的某种做法：

- 将这些段分配到保持池，力图让温块在缓冲区缓存中停留得更久。
- 将“不适合缓存”段分配到回收池，让回收池相当小，以便块能快速地进入缓存和离开缓存（减少管理的开销）。

这样会增加 DBA 所要执行的管理工作，因为要考虑 3 个 缓存，要确定它们的大小，还要为这些缓存分配对象。还要记住，这些池之间没有共享，所以，如果保持池有大量未用的空间，即使默认池或回收池空间不够用了， 保持池也不会把未用空间交出来。总之，这些池一般被视为一种非常精细的低级调优设备，只有所有其他调优手段大多用过之后才应考虑使用（如果可以重写查询， 将 I/O 减少为原来的 1/10，而不是建立多个缓冲区池，我肯定会选择前者！）。

从 Oracle9i 开始，除了默认池、保持池和回收池外，DBA 还要考虑第 4 种可选的缓存：db_Nk_caches。增加这些缓存是为了支持数据库中多种不同的块大小。在 Oracle9i 之前，数据库中只有一种块大小（一般是 2 KB、4 KB、8 KB、16 KB 或 32 KB）。从 Oracle9i 开始，数据库可以有一个默认的块大小，也就是默认池、保持池或回收池中存储的块的大小，还可以有最多 4 种非默认的块大小，请见第 3 章的解释。

与原来默认池中的块一样，这些缓冲区缓存中的块会以同样的方式管理，没有针对不同的池采用任何特殊的算法。下面来看在这些池中如何管理块。

1. 在缓冲区缓存中管理块

为简单起见，这里假设只有一个默认池。由于其他池都以同样的方式管理，所以我们只需要讨论其中一个池。

缓冲区缓存中的块实质上在一个位置上管理，但有两个不同的列表指向这些块：

- 脏（dirty）块列表，其中的块需要由数据库块写入器（DBWn；稍后将介绍这个进程）写入磁盘。
- 非脏（nondirty）块列表。

在 Oracle 8.0 及以前版本中，非脏块列表就是最近最少使用（Least Recently Used，LRU）列表。块按使用的顺序列出。在 Oracle8i 及以后版本中，算法稍有修改。不再按物理顺序来维护块列表，Oracle 采用了一种接触计数（touch count，也称使用计数）算法，如果命中缓存中的一个块，则会增加与之相关联的计数器。不是说每次命中这个块都会增加计数，而是大约每 3 秒一次（如果你连续命中的话）。有一组相当神奇的 X$表，利用其中的某个表就可以看出这个算法是怎样工作的。在 Oracle 的文档中完全没有提到 X$表，但是有关的信息还是时不时地会漏出来一些。

X$BH 表显示了块缓冲区缓存中块的有关信息（文档中有记录的 V$BH 视图也能提供块的有关信息，不过 X$BH 表提供的信息更多）。在这个表中可以看到，我们命中块时，接触计数会增加。可以对这个表运行以下查询，得到 5 个“当前最热的块”，并把这个信息与 DBA_OBJECTS 视图联结，得出这些块属于哪些段。这个查询按 TCH（接触计数）列对 X$BH 中的行排序，并保留前 5 行。然后按 X$BH.OBJ 等于 DBA_OBJECTS.DATA_OBJECT_ID 为条件将 X$BH 信息与 DBA_OBJECTS 联结：

```
sys@ORA10G> select tch, file#, dbablk,
  2         case when obj = 4294967295
  3                 then 'rbs/compat segment'
  4         else (select max( '('||object_type||') ' ||
  5                         owner || '.' || object_name ) ||
  6                         decode( count(*), 1, '', ' maybe!' )
```

```
  7                 from dba_objects
  8                 where data_object_id = X.OBJ )
  9          end what
 10        from (
 11                 select tch, file#, dbablk, obj
 12                 from x$bh
 13                 where state <> 0
 14                 order by tch desc
 15                 ) x
 16        where rownum <= 5
 17 /

    TCH       FILE#       DBABLK         WHAT
---------- ---------- ---------- ---------------------------------------
  51099          1          1434        (TABLE) SYS.JOB$
  49780          1          1433        (TABLE) SYS.JOB$
  48526          1          1450        (INDEX) SYS.I_JOB_NEXT
  11632          2            57        rbs/compat segment
  10241          1          1442        (INDEX) SYS.I_JOB_JOB
```

注意 (2^32 – 1) 或 4 294 967 295 是一个神奇的数，常用来指示“特殊”的块。如果想了解块关联的信息，可以使用查询 select * from dba_extents where file_id = FILE# and block_id <= <DBABLK and block_id+blocks-1 >= DBABLK。

你可能会问，'maybe!'是什么意思，前面的标量子查询中为什么使用 MAX()。这是因为，DATA_OBJECT_ID 不是 DBA_OBJECTS 视图中的“主键”，通过下面的例子可以说明这一点：

```
sys@ORA10G> select data_object_id, count(*)
  2          from dba_objects
```

```
  3          where data_object_id is not null
  4          group by data_object_id
  5          having count(*) > 1;


DATA_OBJECT_ID        COUNT(*)
------------------------      --------------
                     2                17
                     6                 3
                     8                 3
                    10                 3
                    29                 3
                   161                 3
                   200                 3
                   210                 2
                   294                 7
                   559                 2

10 rows selected.
```

这是因为存在聚簇（cluster），有关内容见第 10 章的讨论，其中可能包含多个表。因此，从 X$BH 联结 DBA_OBJECTS 来打印一个段名时，从技术上讲，我们必须列出聚簇中所有对象的所有名字，因为数据库块并不一直属于一个表。

甚至对于重复查询的块，我们也可以观察 Oracle 如何递增这个块的接触计数。在这个例子中我们使用了一个神奇的表 DUAL，可以知道这是一个只有一行一列的表。我们想得出这一行的块信息。内置的 DBMS_ROWID 包就很适合得到这个信息。另外，由于我们要从 DUAL 查询 ROWID，所以 Oracle 会从缓冲区缓存读取真正的 DUAL 表，而不是 Oracle 10g 增加的“虚拟”DUAL 表。

注意 在 Oracle 10g 以前，查询 DUAL 就会导致对数据字典中存储的一个实际 DUAL 表进行全表扫描。如果打开 autotrace（自动跟踪），并查询 SELECT DUMMY FROM DUAL，不论是哪个 Oracle 版本，你都会观察到存在一些 I/O（一致获取，consistent

get)。在 9i 及以前的版本中，如果在 PL/SQL 中查询 SELECT SYSDATE FROM DUAL 或 variable := SYSDATE，也会看到出现实际的 I/O。不过，在 Oracle 10g 中，会把 SELECT SYSDATE 识别为不需要真正查询 DUAL 表（因为你没有从该表中请求列或 rowid），而会使用另一种方法，就像是调用一个函数。因此，不会对 DUAL 做全表扫描，而只是向应用返回 SYSDATE。对于大量使用 DUAL 的系统来说，仅仅这样一个很小的改动，就能大大减少所要执行的一致获取操作的次数。

所以，每次运行以下查询时，都会命中真正的 DUAL 表：

```
sys@ORA9IR2> select tch, file#, dbablk, DUMMY
  2          from x$bh, (select dummy from dual)
  3          where obj = (select data_object_id
  4          from dba_objects
  5          where object_name = 'DUAL'
  6                  and data_object_id is not null)
  7 /

       TCH                 FILE#         DBABLK          D
  ----------          ----------     ----------          -
         1                    1           1617          X
         0                    1           1618          X

sys@ORA9IR2> exec dbms_lock.sleep(3.2);

PL/SQL procedure successfully completed.

sys@ORA9IR2> /
       TCH                 FILE#         DBABLK          D
  ----------          ----------     ----------          -
         2                    1           1617          X
         0                    1           1618          X
```

```
sys@ORA9IR2> exec dbms_lock.sleep(3.2);

PL/SQL procedure successfully completed.

sys@ORA9IR2> /

       TCH                 FILE#         DBABLK          D

----------              ----------      ----------        -

         3                      1            1617        X

         0                      1            1618        X

sys@ORA9IR2> exec dbms_lock.sleep(3.2);

PL/SQL procedure successfully completed.

sys@ORA9IR2> /

       TCH                 FILE#         DBABLK          D

----------              ----------      ----------        -

         4                      1            1617        X

         0                      1            1618        X
```

在不同的 Oracle 版本上，输出可能不同，你可能还会看到返回不止两行。也许你观察到 TCH 并没有每次都递增。在一个多用户系统上，结果可能更难预料。Oracle 试图每 3 秒将 TCH 递增一次（还有一个 TIM 列，它会显示对 TCH 列最后一次更新的时间），但是这个数是否 100%正确并不重要，只要接近就行。另外，Oracle 会有意地“冷却”块，过一段时间会让 TCH 计数递减。所以，如果你在自己的系统上运行这个查询，可能会看到完全不同的结果。

因此，在 Oracle8i 及 以上版本中，块缓冲区不再像以前那样移到块列表的最前面；而是原地留在块列表中，只是递增它的接触计数。不过，过一段时间后，块会很自然地在列表中“移 动”。这里把“移动”一词用引号括起来，这是因为块并不是物理地移动；只是因为维护了多个指向块的列表，所以块会在这些列表间“移动”。例如，已修改的块 由脏列表指示（要由 DBWn 写至磁盘）。过一段时间要重用块时，如果缓冲区缓存满了，就要将接触计数较小的某个块释放，将其“放回到”新数据块列表的接近于中间的位置。

管理这些列表的整个算法相当复杂，而且随着 Oracle 版本的变化也在变化，并不断改进。作为开发人员，我们并不需要关心所有细节，只要知道频繁使用的块会被缓存，不常使用的块不会缓存太久，这就够了。

2. 多个块大小

从 Oracle9i 开始，同一个数据库中可以有多个不同的数据库块大小。此前，一个数据库中的所有块大小都相同，要想使用一个不同的块大小，必须重新建立整个数据库。现在就不同了，你可以有一个“默认的”块大小（最初创建数据库时使用的块大小；即 SYSTEM 和所有 TEMPORARY 表空间的块大小），以及最多 4 个其他的块大小。每个不同的块大小都必须有其自己的缓冲区缓存。默认池、保持池和回收池只缓存具有默认大小的块。为了在数据库中使用非默认的块大小，需要配置一个缓冲区池来保存这些块。

在这个例子中，我的默认块大小是 8 KB。我想创建一个块大小为 16 KB 的表空间：

```
ops$tkyte@ORA10G> create tablespace ts_16k

  2 datafile size 5m

  3 blocksize 16k;

create tablespace ts_16k

*

ERROR at line 1:

ORA-29339: tablespace blocksize 16384 does not match configured blocksizes

ops$tkyte@ORA10G> show parameter 16k

NAME                                 TYPE        VALUE

------------------------------------ ----------- ------------------------------

db_16k_cache_size                    big integer 0
```

现在，由于我还没有配置一个 16 KB 的缓存，所以无法创建这样一个表空间。要解决这个问题，可以在以下方法中选择一种。我可以设置 DB_16K_CACHE_SIZE 参数，并重启数据库。也可以缩小另外的某个 SGA 组件，从而在现有的 SGA 中腾出空间来建立一个 16 KB 的缓存。或者，如果 SGA_MAX_SIZE 参数大于当前的 SGA 大小，我还可以直接分配一个 16 KB 的缓存。

注意 从 Oracle9i 开始，即使数据库已经启动并且正在运行，你也能重新设置各个 SGA 组件的大小。如果你想拥有这个能力，能够“扩大”SGA 的大小（超过初始分配的大小），就必须把 SGA_MAX_SIZE 参数设置为大于已分配 SGA 的某个值。例如，如果启动之后，你的 SGA 大小为 128 MB，你想再为缓冲区缓存增加另外的 64 MB，就必须把 SGA_MAX_SIZE 设置为 192 MB 或更大，以便扩展。

在这个例子中，我采用收缩的办法，即缩小我的 DB_CACHE_SIZE，因为目前这个参数设置得太大了：

```
ops$tkyte@ORA10G> show parameter db_cache_size
```

```
NAME                                 TYPE         VALUE
------------------------------------ -----------  ------------------------------
db_cache_size                        big integer  1G
ops$tkyte@ORA10G> alter system set db_cache_size = 768m;
System altered.
ops$tkyte@ORA10G> alter system set db_16k_cache_size = 256m;
System altered.
ops$tkyte@ORA10G> create tablespace ts_16k
  2 datafile size 5m
  3 blocksize 16k;
Tablespace created.
```

这样一来，我就建立了另外一个缓冲区缓存，要用来缓存 16 KB 大小的块。默认池（由 db_cache_size 参数控制）大小为 768 MB，16 KB 缓存（由 db_16k_cache_size 参数控制）大小为 256 MB。这两个缓存是互斥的，如果一个“填满”了，也无法使用另一个缓存中的空间。这样 DBA 就 能很精细地控制内存的使用，但是这也是有代价的。代价之一就是复杂性和管理。使用多个块大小的目的并不是为了性能或作为一个调优特性，而是为了支持可传输 的表空间，也就是可以把格式化的数据文件从一个数据库传输或附加到另一个数据库。比如说，通过实现多个块大小，可以取得一个使用 8 KB 块大小的事务系统中的数据文件，并将此信息传输到使用 16 KB 或 32 KB 块大小的数据仓库。

不过，对于测试来说，有多个块大小很有好处。如果你想看看你的数据库如何处理另一个块大小，例如，如果使用 4 KB 的块而不是 8 KB 的块，一个表会占用多大的空间。现在由于可以支持多个块大小，你就能很轻松地进行测试，而不必创建一个全新的数据库实例。

还可以把多个块大小用作一种精细调优工具，对一组特定的段进行调优，也就是为这些段分配各自的私有缓冲区池。或者，在一个具有事务用户的混合系统中，事务用户可能使用一组数据，而报告/仓库用户查询另外一组单独的数据。如果块比较小，这对事务数据很有好处，因为这样会减少块上的竞争（每个块上的数据/行越少，就意味着同时访问同一个块的人越少），另外还可以更好地利用缓冲区缓存（用户只向缓存读入他们感兴趣的数据，可能只有一行或者很少的几行）。报告/仓库数据（可能以事务数据为基础）则不同，块更大一些会更好，其部分原因在于这样块开销会较少（所占的总存储空间较小），而且逻辑 I/O 处理的数据更多。由于报告/仓库数据不存在事务数据那样的更新竞争问题，所以如果每个块上有更多的行，这并不是问题，反而是一个优点。另外，事务用户实际上会得到自己的缓冲区缓存；他们并不担心报告查询会过分占用缓存。

但是一般来讲，默认池、保持池和回收池对于块缓冲区缓存精细调优来说应该已经足

够了，多个块大小主要用于从一个数据库向另一个数据库传输数据，可能在混合的报告/事务系统中也会用到这种机制。

4.2.4 共享池

共享池是 SGA 中最重要的内存段之一，特别是对于性能和可扩缩性来说。共享池如果太小，会严重影响性能，甚至导致系统看上去好像中止了一样。如果共享池太大，也会有同样的效果。共享池使用不当会导致灾难性的后果。

那么，到底什么是共享池？共享池就是 Oracle 缓存一些“程序”数据的地方。在解析一个查询时，解析得到的表示（representation）就缓存在那里。在完成解析整个查询的任务之前， Oracle 会搜索共享池，看看这个工作是否已经完成。你运行的 PL/SQL 代码就在共享池中缓存，所以下一次运行时，Oracle 不会再次从磁盘重新读取。PL/SQL 代码不仅在这里缓存，还会在这里共享。如果有 1 000 个会话都在执行同样的代码，那么只会加载这个代码的一个副本，并由所有会话共享。Oracle 把系统参数存储在共享池中。数据字典缓存（关于数据库对象的已缓存信息）也存储在这里。简单地讲，就像是厨房的水池一样，什么东西都往共享池里放。

共享池的特点是有大量小的内存块（chunk），一般为 4 KB 或更小。要记住，4 KB 并不是一个硬性限制，可能有的内存分配会超过这个大小，但是一般来讲，我们的目标是使用小块的内存来避免碎片问题。如果分配的内存块大小显著不同（有的很小，有的却相当大），就可能出现碎片问题。共享池中的内存根据 LRU（最近最少使用）的原则来管理。在这方面，它类似于缓冲区缓存，如果你不用某个对象，它就会丢掉。为此提供了一个包，名叫 DBMS_SHARED_POOL，这个包可用于改变这种行为，强制性地“钉住”共享池中的对象。可以使用这个过程在数据库启动时加载频繁使用的过程和包，并使它们不至于老化。不过，通常如果过一段时间共享池中的一段内存没有得到重用，它就会老化。甚至 PL/SQL 代码（可能相当大）也以一种分页机制来管理，这样当你执行一个非常大的包中的代码时，只有所需的代码会加载到共享池的小块中。如果你很长时间都没有用它，而且共享池已经填满，需要为其他对象留出空间，它就会老化。

如果你真的想破坏 Oracle 的共享池，最容易的办法是不使用绑定变量。在第 1 章已经看到，如果不使用绑定变量，可能会让系统陷于瘫痪，这有两个原因：

- 系统要花大量 CPU 时间解析查询。
- 系统使用大量资源来管理共享池中的对象，因为从来不重用查询。

如果提交到 Oracle 的 每个查询都是具有硬编码值的惟一查询，则共享池的概念就一点用都没有。设计共享池是为了反复使用查询计划。如果每个查询都是全新的，都是以前从来没有见过 的查询，那么缓存只会增加开销。共享池反而会损害性能。为了解决这个问题，很多人都会用一种看似合理的常用技术，也就是向共享池增加更多的空间，但是这种 做法一般只会使问题变得比以前更糟糕。由于共享池不可避免地会再次填满，比起原来较小的共享池来说，开销甚至更大，原因很简单，与管理一个较小的满共享池 相比，管理一个大的满共享池需要做更多的工作。

对于这个问题，真正的解决方案只有一个，这就是使用共享 SQL，也就是重用查询。在前面（第 1 章），我们简要介绍了参数 CURSOR_SHARING，在这方面，游标共享可以作

为一种短期的解决方案。不过，真正要解决这个问题，首当其冲地还是要使用可重用的 SQL。即使在最大的系统上，我发现一般也最多有 10 000～20 000 条不同的 SQL 语句。大多数系统只执行数百个不同的查询。

下面是一个真实的示例，从这个例子可以看出，如果共享池使用不当后果会有多严重。我曾参与过这样一个系统，它的标准操作过程是每天晚上关闭数据库，清空 SGA，再重启。之所以这样做只是因为，系统白天运行时有问题，会完全占用 CPU，所以如果数据库运行的时间超过一天，性能就开始严重下降。他们原来在一个 1.1 GB 的 SGA 中使用了 1 GB 的共享池。确实如此：0.1 GB 由块缓冲区缓存和其他元素专用，另外 1 GB 则 完全用于缓存不同的查询，但这些查询从来都不会再次执行。必须冷启动的原因是，如果让系统运行一天以上的时间，就会用光共享池中的空闲内存。此时，结构老 化的开销就太大了（特别是对于一个如此大的结构），系统会为此疲于奔命，而性能也会大幅恶化（不过原来的性能也好不到哪里去，因为他们管理的是一个 1 GB 的共享池）。另外，使用这个系统的人一直想向机器增加越来越多的 CPU，因为硬解析 SQL 太耗费 CPU。通过对应用进行修正，让它使用绑定变量，不仅消除了物理的硬件需求（现在的 CPU 能力比他们实际需要的已经高出几倍），而且对各个池的内存分配也反过来了。现在不是使用 1 GB 的共享池，而只为共享池分配了不到 100 MB 的空间，即使是经过数周连续地运行，也不会用光共享池的内存。

关于共享池和参数 SHARED_POOL_SIZE 还有一点要说明。在 Oracle9i 及以前的版本中，查询的结果与 SHARED_POOL_SIZE 参数之间没有直接的关系，查询结果是：

```
ops$tkyte@ORA9IR2> select sum(bytes) from v$sgastat where pool = 'shared pool';

SUM(BYTES)

----------

100663296
```

SHARED_POOL_SIZE 参数是：

```
ops$tkyte@ORA9IR2> show parameter shared_pool_size

NAME                                 TYPE        VALUE

------------------------------------ ----------- ------------------------------

shared_pool_size                     big integer 83886080
```

如果实在要谈谈它们的关系，只能说 SUM(BYTES) FROM V$SGASTAT 总是大于 SHARED_ POOL_SIZE。共享池还保存了另外的许多结构，它们不在相应参数的作用域内。SHARED_POOL_SIZE 通常占了共享池（SUM(BYTES)报告的结果）中最大的一部分，但这不是共享池中惟一的一部分。例如，参数 CONTROL_FILES 就为共享池“混合”部分做出了贡献，每个文件有 264 字节。遗憾的是，V$SGASTAT 中的“共享池”与参数 SHARED_POOL_SIZE 的命名让人很容易混淆，这个参数对共享池大小贡献最大，但是它并不是惟一有贡献的参数。

不过，在 Oracle 10g 及以上版本中，应该能看到二者之间存在一对一的对应关系，假

设你使用手动的 SGA 内存管理（也就是说，自己设置 SHARED_POOL_SIZE 参数）：

```
ops$tkyte@ORA10G> select sum(bytes)/1024/1024 mbytes
  2 from v$sgastat where pool = 'shared pool';

MBYTES
----------
128

ops$tkyte@ORA10G> show parameter shared_pool_size;

NAME                                 TYPE            VALUE
------------------------------------ ----------- ------------------------------
shared_pool_size                     big integer     128M
```

如果你从 Oracle9i 或之前的版本转向 10g，这是一个相当重要的改变。在 Oracle10g 中，SHARED_POOL_SIZE 参数控制了共享池的大小，而在 Oracle9i 及之前的版本中，它只是共享池中贡献最大的部分。你可能想查看 9i（或之前版本）中实际的共享池大小（根据 V$SGASTAT），并使用这个数字来设置 Oracle 10g（及以上版本）中的 SHARED_POOL_SIZE 参数。用于增加共享池大小的多种其他组件现在期望由你来分配内存。

4.2.5 大池

大池（large pool）并不是因为它是一个“大”结构才这样取名（不过，它可能确实很大）。之所以称之为大池，是因为它用于大块内存的分配，共享池不会处理这么大的内存块。

在 Oracle 8.0 引入大池之前，所有内存分配都在共享池中进行。如果你使用的特性要利用“大块的”内存分配（如共享服务器 UGA 内存分配），倘若都在共享池中分配就不太好。另外，与共享池管理内存的方式相比，处理（需要大量内存分配）会以不同的方式使用内存，所以这个问题变得更加复杂。共享池根据 LRU 来管理内存，这对于缓存和重用数据很合适。不过，大块内存分配则是得到一块内存后加以使用，然后就到此为止，没有必要缓存这个内存。

Oracle 需要的应该是像为块缓冲区缓存实现的回收和保持缓冲区池之类的组件。这正是现在的大池和共享池。大池就是一个回收型的内存空间，共享池则更像是保持缓冲区池。如果对象可能会被频繁地使用，就将其缓存起来。

大池中分配的内存在堆上管理，这与 C 语言通过 malloc()和 free()管理内存很相似。一旦“释放”了一块内存，它就能由其他进程使用。在共享池中，实际上没有释放内存块的概念。你只是分配内存，然后使用，再停止使用而已。过一段时间，如果需要重用那个内存，Oracle 会让你的内存块老化。如果只使用共享池，问题在于：一种大小不一定全局适用。

大池专门用于以下情况：

- 共享服务器连接，用于在 SGA 中分配 UGA 区
- 语句的并行执行，允许分配进程间的消息缓冲区，这些缓冲区用于协调并行查询服务器。
- 备份，在某些情况下用于 RMAN 磁盘 I/O 缓冲区。

可以看到，这些内存分配都不应该在 LRU 缓冲区池中管理，因为 LRU 缓冲区池的目标是管理小块的内存。例如，对于共享服务器连接内存，一旦会话注销，这个内存就不会再重用，所以应该立即返回到池中。另外，共享服务器 UGA 内存分配往往“很大”。如果查看前面使用 SORT_AREA_RETAINED_SIZE 或 PGA_AGGREGATE_TARGET 的例子，可以看到，UGA 可能扩张得很大，成为绝对大于 4 KB 的块。把 MTS 内存放在共享池中，这会导致把它分片成很小的内存，不仅如此，你还会发现从不重用的大段内存会导致可能重用的内存老化。这就要求数据库以后多做更多的工作来重建内存结构。

对于并行查询消息缓冲区也是如此，因为它们不能根据 LRU 原则来管理。并行查询消息缓冲区可以分配，但是在使用完之前不能释放。一旦发送了缓冲区中的消息，就不再需要这个缓冲区，应该立即释放。对于备份缓冲区更是如此，备份缓冲区很大，而且一旦 Oracle 用完了这些缓冲区，它们就应该“消失”。

使用共享服务器连接时，并不是一定得使用大池，但是强烈建议你使用大池。如果没有大池，而且使用了一个共享服务器连接，就会像 Oracle 7.3 及以前版本中一样从共享池分配空间。过一段时间后，这会导致性能恶化，一定要避免这种情况。如果 DBWR_IO_SLAVES 或者 PARALLEL_MAX_SERVERS 参数设置为某个正值，大池会默认为某个大小。如果你使用了一个用到大池的特性，建议你手动设置大池的大小。默认机制一般并不适合你的具体情况。

4.2.6 Java 池

Java 池（Java pool）是 Oracle 8.1.5 版本中增加的，目的是支持在数据库中运行 Java。如果用 Java 编写一个存储过程，Oracle 会在处理代码时使用 Java 池的内存。参数 JAVA_POOL_SIZE 用于确定为会话特有的所有 Java 代码和数据分配多大的 Java 池内存量。

Java 池有多种用法，这取决于 Oracle 服务器运行的模式。如果采用专用服务器模式，Java 池包括每个 Java 类的共享部分，由每个会话使用。这些实质上是只读部分（执行向量、方法等），每个类的共享部分大约 4～8 KB。

因此，采用专用服务器模式时（应用使用纯 Java 存储过程时往往就会出现这种情况），Java 池所需的总内存相当少，可以根据要用的 Java 类的个数来确定。应该知道，如果采用专用服务器模式，每个会话的状态不会存储在 SGA 中，因为这个信息要存储在 UGA 中，你应该记得，使用专用服务器模式时，UGA 包括在 PGA 中。

使用共享服务器连接来连接 Oracle 时，Java 池包括以下部分：

- 每个 Java 类的共享部分。
- UGA 中用于各会话状态的部分，这是从 SGA 中的 JAVA_POOL 分配的。UGA 中余下的部分会正常地在共享池中分配，或者如果配置了大池，就会在大池中分配。

由于 Oracle9i 及以前版本中 Java 池的总大小是固定的，应用开发人员需要估计应用的

总需求，再把估计的需求量乘以所需支持的并发会话数，所得到的结果能指示出 Java 池的总大小。每个 Java UGA 会根据需要扩大或收缩，但是要记住，池的大小必须合适，所有 UGA 加在一起必须能同时放在里面。在 Oracle10g 及以上版本中，这个参数可以修改，Java 池可以随着时间的推移而扩大和收缩，而无需重启数据库。

4.2.7 流池

流池（stream pool）是一个新的 SGA 结构，从 Oracle 10g 开始才增加的。流（Stream）本身就是一个新的数据库特性，Oracle9i Release 2 及以上版本中才有。它设计为一个数据库共享/复制工具，这是 Oracle 在数据复制方面发展的方向。

注意 上面提到了流“是 Oracle 在数据复制方面发展的方向”，这句话不能解释为高级复制（Advanced Replication，这是 Oracle 现有的复制特性）会很快过时。相反，将来几个版本中还会支持高级复制。要了解流本身的更多内容，请参考 Streams Concepts Guide （在 http://otn.oracle.com 的 Documentation 部分）。

流池（或者如果没有配置流池，则是共享池中至多 10%的空间）会用于缓存流进程在数据库间移动/复制数据时使用的队列消息。这里并不是使用持久的基于磁盘的队列（这些队列有一些附加的开销），流使用的是内存中的队列。如果这些队列满了，最终还是会写出到磁盘。如果使用内存队列的 Oracle 实例由于某种原因失败了，比如说因为实例错误（软件瘫痪）、掉电或其他原因，就会从重做日志重建这些内存中的队列。

因此，流池只对使用了流数据库特性的系统是重要的。在这些环境中，必须设置流池，以避免因为这个特性从共享池“窃取”10%的空间。

4.2.8 自动 SGA 内存管理

与管理 PGA 内存有两种方法一样，从 Oracle 10g 开始，管理 SGA 内存也有两种方法：手动管理和自动管理。手动管理需要设置所有必要的池和缓存参数，自动管理则只需设置少数几个内存参数和一个 SGA_TARGET 参数。通过设置 SGA_TARGET 参数，实例就能设置各个 SGA 组件的大小以及调整它们的大小。

注意 在 Oracle9i 及以前版本中，只能用手动 SGA 内存管理，不存在参数 SGA_TARGET，而且参数 SGA_MAX_SIZE 只是一个上限，而不是动态目标。

在 Oracle 10g 中，与内存相关的参数可以归为两类：

- 自动调优的 SGA 参数：目前这些参数包括 DB_CACHE_SIZE、SHARED_POOL_SIZE、LARGE_POOL_SIZE 和 JAVA_POOL_SIZE。
- 手动 SGA 参数：这些参数包括 LOG_BUFFER、STREAMS_POOL、DB_NK_CACHE_SIZE、DB_KEEP_CACHE_SIZE 和 DB_RECYCLE_CACHE_SIZE。

在 Oracle 10g 中，任何时候你都能查询 V$SGAINFO，来查看 SGA 的哪些组件的大小可以调整。

注意 要使用自动 SGA 内存管理，参数 STATISTICS_LEVEL 必须设置为 TYPICAL 或 ALL。如果不支持统计集合，数据库就没有必要的历史信息来确定大小。

采用自动 SGA 内存管理时，确定自动调整组件大小的主要参数是 SGA_TARGET，这个参数可以在数据库启动并运行时动态调整，最大可以达到 SGA_MAX_SIZE 参数设置的值（默认等于 SGA_TARGET，所以如果想增加 SGA_TARGET，就必须在启动数据库实例之前先把 SGA_MAX_SIZE 设置得大一些）。数据库会使用 SGA_TARGET 值，再减去其他手动设置组件的大小（如 DB_KEEP_CACHE_SIZE、DB_RECYCLE_CACHE_SIZE 等），并使用计算得到的内存量来设置默认缓冲区池、共享池、大池和 Java 池的大小。在运行时，实例会根据需要动态地对这 4 个内存区分配和撤销内存。如果共享池内存用光了，实例不会向用户返回一个 ORA-04031“Unable to allocate N bytes of shared memory”（无法分配 N 字节的共享内存）错误，而是会把缓冲区缓存缩小几 MB（一个颗粒的大小），再相应地增加共享池的大小。

随着时间的推移，当实例的内存需求越来越确定时，各个 SGA 组件的大小也越来越固定。即便数据库关闭后又启动，数据库还能记得组件的大小，因此不必每次都从头再来确定实例的正确大小。这是通过 4 个带双下划线的参数做到的：__DB_CACHE_SIZE、__JAVA_POOL_SIZE、__LARGE_POOL_SIZE 和__SHARED_POOL_SIZE。如果正常或立即关闭数据库，则数据库会把这些值记录到存储参数文件（SPFILE）中，并在启动时再使用这些值来设置各个区的默认大小。

另外，如果知道 4 个区中某个区的最小值，那么除了设置 SGA_TARGET 外，还可以设置这个参数。实例会使用你的设置作为下界（即这个区可能的最小大小）。

4.3 小结

这一章介绍了 Oracle 内存结构。首先从进程和会话级开始，我们分析了 PGA 和 UGA 以及它们的关系。还了解到连接 Oracle 的模式可以指示内存组织的方式。相对于共享服务器连接来说，专用服务器连接表示服务器进程中会使用更多的内存，但是使用共享服务器连接的话，则说明需要的 SGA 大得多。接下来，我们讨论了 SGA 本身的主要结构，揭示了共享池和大池之间的区别，并说明为什么希望有一个大池来“节省”我们的共享池。我们还介绍了 Java 池，以及在各种情况下如何使用 Java 池。此外还分析了块缓冲区缓存，以及如何将块缓冲区缓存划分为更小、更“专业”的池。

下面可以转向 Oracle 实例的余下一部分，即构成 Oracle 实例的物理进程。

第 5 章 Oracle 进程

现在要谈到 Oracle 体系结构的最后一部分了。我们已经研究了数据库以及构成数据库的物理文件集。讨论 Oracle 使用的内存时，我们分析了实例的前半部分。Oracle 体系结构中还有一个问题没有讲到，这就是构成实例另一半的进程（process）集。

Oracle 中的各个进程要完成某个特定的任务或一组任务，每个进程都会分配内部内存（PGA 内存）来完成它的任务。Oracle 实例主要有 3 类进程：

- 服务器进程（server process）：这些进程根据客户的请求来完成工作。我们已经对专用服务器和共享服务器有了一定的了解。它们就是服务器进程。
- 后台进程（background process）：这些进程随数据库而启动，用于完成各种维护任务，如将块写至磁盘、维护在线重做日志、清理异常中止的进程等。
- 从属进程（slave process）：这些进程类似于后台进程，不过它们要代表后台进程或服务器进程完成一些额外的工作。

其中一些进程（如数据库块写入器（DBWn）和日志写入器（LGWR））在前面已经提到过，不过现在我们要更详细地介绍这些进程的功能，说明这些进程会做什么，并解释为什么。

注意 这一章谈到“进程”时，实际上要把它理解为两层含义，在某些操作系统（如 Windows）上，Oracle 使用线程实现，所以在这种操作系统上，就要把我们所说的“进程”理解为“线程”的同义词。在这一章中，“进程”一词既表示进程，也涵盖线程。如果你使用的是一个多进程的 Oracle 实现，比如说 UNIX 上的 Oracle 实现，“进程”就很贴切。如果你使用的是单进程的 Oracle 实现，如 Windows 上的 Oracle 实现，“进程”实际是指“Oracle 进程中的线程”。所以，举例来说，当我谈到 DBWn 进程时，在 Windows 上就对应为 Oracle 进程中的 DBWn 线程 。

5.1 服务器进程

服务器进程就是代表客户会话完成工作的进程。应用向数据库发送的 SQL 语句最后就要由这些进程接收并执行。

在第 2 章中，我们简要介绍了两种 Oracle 连接，包括：

- 专用服务器（dedicated server）连接，采用专用服务器连接时，会在服务器上得到针对这个连接的一个专用进程。数据库连接与服务器上的一个进程或线程之间存在一对一的映射。
- 共享服务器（shared server）连接，采用共享服务器连接时，多个会话可以共享一个服务器进程池，其中的进程由 Oracle 实例生成和管理。你所连接的是一个数据库调度器（dispatcher），而不是特意为连接创建的一个专用服务器进程。

注意 有一点很重要，要知道 Oracle 术语中连接和会话之间的区别。连接（connection）就是客户进程与 Oracle 实例之间的一条物理路径（例如，客户与实例之间的一个网络连接）。会话（session）则不同，这是数据库中的一个逻辑实体，客户进程可以在会话上执行 SQL 等。多个独立的会话可以与一个连接相关联，这些会话甚至可以独立于连接存在。稍后将进一步讨论这个问题。

专用服务器进程和共享服务器进程的任务是一样的：要处理你提交的所有 SQL。当你向数据库提交一个 SELECT * FROM EMP 查询时，会有一个 Oracle 专用/共享服务器进程解析这个查询，并把它放在共享池中（或者最好能发现这个查询已经在共享池中）。这个进程要提出查询计划，如果必要，还要执行这个查询计划，可能在缓冲区缓存中找到必要的数据，或者将数据从磁盘读入缓冲区缓存中。

这些服务器进程是干重活的进程。在很多情况下，你都会发现这些进程占用的系统 CPU 时间最多，因为正是这些进程来执行排序、汇总、联结等等工作，几乎所有工作都是这些进程做的。

5.1.1 专用服务器连接

在专用服务器模式下，客户连接和服务器进程（或者有可能是线程）之间会有一个一对一的映射。如果一台 UNIX 主机上有 100 条专用服务器连接，就会有相应的 100 个进程在执行。可以用图来说明，如图 5-1 所示。

客户应用中链接着 Oracle 库，这些库提供了与数据库通信所需的 API。这些 API 知道如何向数据库提交查询，并处理返回的游标。它们知道如何把你的请求打包为网络调用，专用服务器则知道如何将这些网络调用解开。这部分软件称为 Oracle Net，不过在以前的版本中可能称之为 SQL*Net 或 Net8。这是一个网络软件/协议，Oracle 利用这个软件来支持客户/服务器处理（即使在一个 n 层体系结构中也会“潜伏”着客户/服务器程序）。不过，即使

从技术上讲没有涉及 Oracle Net，Oracle 也采用了同样的体系结构。也就是说，即使客户和服务器在同一台机器上，也会采用这种两进程（也称为两任务）体系结构。这个体系结构有两个好处：

图 5-1 典型的专用服务器连接

- 远程执行（remote execution）：客户应用可能在另一台机器上执行（而不是数据库所在的机器），这是很自然的。
- 地址空间隔离（address space isolation）：服务器进程可以读写 SGA。如果客户进程和服务器进程物理地链接在一起，客户进程中一个错误的指针就能轻松地破坏 SGA 中的数据结构。

我们在第 2 章中了解了这些专用服务器如何“产生”，也就是如何由 Oracle 监听器进程创建它们。这里不再介绍这个过程；不过，我们会简要地说明如果不涉及监听器会是什么情况。这种机制与使用监听器的机制基本上是一样的，但不是由监听器通过 fork()/exec()调用（UNIX）或进程间通信 IPC 调用（Windows）来创建专用服务器，而是由客户进程自己来创建。

注意 有多种 fork()和 exec()调用，如 vfork()、execve()等。Oracle 所用的调用可能根据操作系统和实现的不同而有所不同，但是最后的结果是一样的。fork()创建一个新进程，这是父进程的一个克隆，而且在 UNIX 上这也是创建新进程的惟一途径。exec()在内存中现有的程序映像上加载一个新的程序映像，这就启动了一个新程序。所以 SQL*Plus 可以先“fork”（即复制自身），然后“exec”Oracle 二进制可执行程序，用这个新程序覆盖它自己的副本。

在 UNIX 上，可以在同一台机器上运行客户和服务器，就能很清楚地看出这种父/子进程的创建：

```
ops$tkyte@ORA10G> select a.spid dedicated_server,
  2                 b.process clientpid
```

```
  3          from v$process a, v$session b
  4          where a.addr = b.paddr
  5                  and b.sid = (select sid from v$mystat where rownum=1)
  6 /

DEDICATED_SE                    CLIENTPID
  ----------------------                  ----------------
                  5114                            5112


ops$tkyte@ORA10G> !/bin/ps -p 5114 5112
   PID TTY     STAT     TIME COMMAND
5112 pts/1               R        0:00        sqlplus
5114          ?                        S                        0:00                        oracleora10g
(DESCRIPTION=(LOCAL=YES)..(PROTOCOL=beq)))
```

在此，我使用了一个查询来发现与专用服务器相关联的进程 ID（PID），从 V$PROCESS 得到的 SPID 是执行该查询时所用进程的操作系统 PID。

5.1.2 共享服务器连接

下面更详细地介绍共享服务器进程。共享服务器连接强制要求必须使用 Oracle Net，即使客户和服务器都在同一台机器上也不例外。如果不使用 Oracle TNS 监听器，就无法使用共享服务器。如前所述，客户应用会连接到 Oracle TNS 监听器，并重定向或转交给一个调度器。调度器充当客户应用和共享服务器进程之间的“导管”。图 5-2 显示了与数据库建立共享服务器连接时的体系结构。

图 5-2 典型的共享服务器连接

在此可以看到，客户应用（其中链接了 Oracle 库）会与一个调度器进程物理连接。对于给定的实例，可以配置多个调度器，但是对应数百个（甚至数千个）用户只有一个调度器的情况并不鲜见。调度器只负责从客户应用接收入站请求，并把它们放入 SGA 中的一个请求队列。第一个可用的共享服务器进程（与专用服务器进程实质上一样）从队列中选择请求，并附加相关会话的 UGA（图 5-2 中标有“S” 的方框）。共享服务器处理这个请求，把得到的输出放在响应队列中。调度器一直监视着响应队列来得到结果，并把结果传回给客户应用。就客户而言，它分不清到 底是通过一条专用服务器连接还是通过一条共享服务器连接进行连接，看上去二者都一样，只是在数据库级二者的区别才会明显。

5.1.3 连接与会话

连接并不是会话的同义词，发现这一点时很多人都很诧异。在大多数人眼里，它们都是一样的，但事实上并不一定如此。在一条连接上可以建立 0 个、一个或多个会话。各个会话是单独而且独立的，即使它们共享同一条数据库物理连接也是如此。一个会话中的提交不会影响该连接上的任何其他会话。实际上，一条连接上的各个会话可以使用不同的用户身份！

在 Oracle 中，连接只是客户进程和数据库实例之间的一条特殊线路，最常见的就是网络连接。这条连接可能连接到一个专用服务器进程，也可能连接到调度器。如前所述，连接上可以有 0 个或多个会话，这说明可以有连接而无相应的会话。另外，一个会话可以有连接也可以没有连接。使用高级 Oracle Net 特性（如连接池）时，客户可以删除一条物理连接，而会话依然保留（但是会话会空闲）。客户在这个会话上执行某个操作时，它会重新建立物理连接。下面更详细地定义这些术语：

- 连接（connection）：连接是从客户到 Oracle 实例的一条物理路径。连接可以在网络上建立，或者通过 IPC 机制建立。通常会在客户进程与一个专用服务器或一个调度器之间建立连接。不过，如果使用 Oracle 的连接管理器（Connection Manager ，CMAN），还可以在客户和 CMAN 之间以及 CMAN 和数据库之间建立连接。CMAN 的介绍超出了本书的范围，不过 Oracle Net Services Administrator’s

Guide（可以从 http://otn.oracle.com 免费得到）对 CMAN 有详细的说明。

- 会话（session）：会话是实例中存在的一个逻辑实体。这就是你的会话状态（session state），也就是表示特定会话的一组内存中的数据结构。提到“数据库连接”时，大多数人首先想到的就是“会话”。你要在服务器中的会话上执行 SQL、提交事务和运行存储过程。

可以使用 SQL*Plus 来看一看实际的连接和会话是什么样子，从中还可以了解到，实际上一条连接有多个会话的情况相当常见。这里使用了 AUTOTRACE 命令，并发现有两个会话。我们在一条连接上使用一个进程创建了两个会话。以下是其中的第一个会话：

```
ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status
  2 from v$session
  3 where username = USER
  4 /

USERNAME SID     SERIAL#     SERVER         PADDR         STATUS
---------          ----       --------         ---------          --------          --------
OPS$TKYTE                  153                  3196                  DEDICATED
AE4CF614       ACTIVE
```

这说明现在有一个会话：这是一个与单一专用服务器连接的会话。以上 PADDR 列是这个专用服务器进程的地址。下面，只需打开 AUTOTRACE 来查看 SQL*Plus 中所执行语句的统计结果：

```
ops$tkyte@ORA10G> set autotrace on statistics
ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status
  2 from v$session
  3 where username = USER
  4 /


USERNAME SID     SERIAL#     SERVER         PADDR         STATUS
---------          ----       --------         ---------          --------          --------
OPS$TKYTE                  151                  1511                  DEDICATED
AE4CF614       INACTIVE
```

```
OPS$TKYTE                153               3196                DEDICATED
AE4CF614       ACTIVE


Statistics

----------------------------------------------------------

          0 recursive calls

          0 db block gets

          0 consistent gets

          0 physical reads

          0 redo size

        756 bytes sent via SQL*Net to client

        508 bytes received via SQL*Net from client

          2 SQL*Net roundtrips to/from client

          0 sorts (memory)

          0 sorts (disk)

          2 rows processed

ops$tkyte@ORA10G> set autotrace off
```

这样一来，我们就有了两个会话，但是这两个会话都使用同一个专用服务器进程，从它们都有同样的 PADDR 值就能看出这一点。从操作系统也可以得到确认，因为没有创建新的进程，对这两个会话只使用了一个进程（一条连接）。需要注意，其中一个会话（原来的会话）是 ACTIVE（活动的）。这是有道理的：它正在运行查询来显示这个信息，所以它当然是活动的。但是那个 INACTIVE（不活动的）会话呢？那个会话要做什么？这就是 AUTOTRACE 会话，它的任务是“监视”我们的实际会话，并报告它做了什么。

在 SQL*Plus 中启用（打开）AUTOTRACE 时，如果我们执行 DML 操作（INSERT、UPDATE、DELETE、SELECT 和 MERGE），SQL*Plus 会完成以下动作：

(1) 如果还不存在辅助会话[1]，它会使用当前连接创建一个新会话。

(2) 要求这个新会话查询 V$SESSTAT 视图来记住实际会话（即运行 DML 的会话）的初始统计值。这与第 4 章中 watch_stat.sql 脚本完成的功能非常相似。

(3) 在原会话中运行 DML 操作。

(4) DML 语句执行结束后，SQL*Plus 会请求另外那个会话（即“监视”会话）再次查询 V$SESSTAT，并生成前面所示的报告，显示出原会话（执行 DML 的会话）的统计结果之差。

如果关闭 AUTOTRACE，SQL*Plus 会终止这个额外的会话，在 V$SESSION 中将无法看到这个会话。你可能会问：“SQL*Plus 为什么要这样做，为什么要另建一个额外的会话？”答案很简单。我们在第 4 章中曾经使用第二个 SQL*Plus 会话来监视内存和临时空间的使用情况，原因是：如果使用同一个会话来监视内存使用，那执行监视本身也要使用内存。SQL*Plus 之所以会另外创建一个会话来执行监视，原因也是一样的。如果在同一个会话中观察统计结果，就会对统计结果造成影响（导致对统计结果的修改）。倘若 SQL*Plus 使用一个会话来报告所执行的 I/O 次数，网络上传输了多少字节，以及执行了多少次排序，那么查看这些详细信息的查询本身也会影响统计结果。这些查询可能自己也要排序、执行 I/O 以及在网络上传输数据等（一般来说都会如此!）。因此，我们需要使用另一个会话来正确地测量。

到目前为止，我们已经看到一条连接可以有一个或两个会话。现在，我们想使用 SQL*Plus 来查看一条没有任何会话的连接。这很容易。在上例所用的同一个 SQL*Plus 窗口中，只需键入一个“很容易误解”的命令即 DISCONNECT：

```
ops$tkyte@ORA10G> disconnect

Disconnected from Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 –

Production

With the Partitioning, OLAP and Data Mining options

ops$tkyte@ORA10G>
```

从技术上讲，这个命令应该叫 DESTROY_ALL_SESSIONS 更合适，而不是 DISCONNECT，因为我们并没有真正物理地断开连接。

注意 在 SQL*Plus 中要真正地断开连接，应该执行“exit”命令，因为你必须退出才能完全撤销连接。

不过，我们已经关闭了所有会话。如果使用另一个用户账户[2]打开另一个会话，并查询（当然要用你原来的账户名代替 OPS$TKYTE）。

```
sys@ORA10G> select * from v$session where username = 'OPS$TKYTE';

no rows selected
```

可以看到，这个账户名下没有会话，但是仍有一个进程，相应地有一条物理连接（使用前面的 ADDR 值）[3]：

```
sys@ORA10G> select username, program

  2 from v$process

  3 where addr = hextoraw('AE4CF614');
```

```
USERNAME     PROGRAM

---------------        ----------------------------------------------

        tkyte          oracle@localhost.localdomain (TNS V1-V3)
```

所以，这就有了一条没有相关会话的“连接”。可以使用 SQL*Plus 的 CONNECT 命令（这个命令的名字也起得不恰当），在这个现有的进程中创建一个新会话（CONNECT 命令叫 CREATE_SESSION 更合适）：

```
ops$tkyte@ORA10G> connect /

Connected.



ops$tkyte@ORA10G> select username, sid, serial#, server, paddr, status

  2 from v$session

  3 where username = USER

  4 /



USERNAME SID      SERIAL#      SERVER          PADDR          STATUS

---------            ----       --------          ---------            --------          --------

OPS$TKYTE            150            233                DEDICATED AE4CF614        ACTIVE
```

可以注意到，PADDR 还是一样的，所以我们还是在使用同一条物理连接，但是（可能）有一个不同的 SID。我说“可能有”，是因为也许还会分配同样的 SID，这取决于在我们注销时是否有别人登录，以及我们原来的 SID 是否可用。

到此为止，这些测试都是用一条专用服务器连接执行的，所以 PADDR 正是专用服务器进程的进程地址。如果使用共享服务器会怎么样呢？

注意　要想通过共享服务器连接，你的数据库实例必须先做必要的设置才能启动。有关如何配置共享服务器，这超出了本书的范围，不过这个主题在 Oracle Net Services Administrator’s Guide 中有详细说明。

那好，下面使用共享服务器登录，并在这个会话中查询：

```
ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
```

```
  2 a.paddr, a.status, b.program
  3 from v$session a left join v$process b
  4 on (a.paddr = b.addr)
  5 where a.username = 'OPS$TKYTE'
  6 /

USERNAME SID     SERIAL# SERVER PADDR         STATUS     PROGRAM
--------- --- ------- ------- -------- ------ ----------
-------
OPS$TKYTE 150 261 SHARED AE4CF118 ACTIVE
                oracle@localhost(S000)
```

这个共享服务器连接与一个进程关联，利用 PADDR 可以联结到 V$PROCESS 来得出进程名。在这个例子中，可以看到这确实是一个共享服务器，由文本 S000 标识。

不过，如果使用另一个 SQL*Plus 窗口来查询这些信息，而保持我们的共享服务器会话空闲，就会看到下面的信息：

```
sys@ORA10G> select a.username, a.sid, a.serial#, a.server,
  2         a.paddr, a.status, b.program
  3 from v$session a left join v$process b
  4         on (a.paddr = b.addr)
  5 where a.username = 'OPS$TKYTE'
  6 /

USERNAME SID     SERIAL# SERVER PADDR         STATUS      PROGRAM
--------- --- ------- ------ -------- -------- ----------
---------
OPS$TKYTE 150 261 NONE AE4CEC1C INACTIVE
                oracle@localhost(D000)
```

可以注意到，PADDR 不一样了，与连接关联的进程名也改变了。空闲的共享服务器连

接现在与一个调度器 D000 关联。这样一来，我们又有了一种方法来观察指向一个进程的多个会话。可以有数百个（甚至数千个）会话指向一个调度器。

共享服务器连接有一个很有意思的性质：我们使用的共享服务器进程可能随调用而改变。如果只有我一个人在使用这个系统（因为我在执行这些测试），作为 OPS$TKYTE 反复运行这个查询，会反复生成同样的 PADDR：AE4CF118。不过，如果打开多条共享服务器连接，并开始在其他会话中使用这个共享服务器，可能就会注意到使用的共享服务器有变化。

考 虑下面的例子。这里要查询我当前的会话信息，显示所用的共享服务器。然后在另一个共享服务器会话中完成一个运行时间很长的操作（也就是说，我要独占这个共 享服务器）。再次询问数据库我所用的共享服务器时，很有可能看到一个不同的共享服务器（如果原来的共享服务器已经开始为另一个会话提供服务）。在下面的例 子中，粗体显示的代码表示通过共享服务器连接的第二个 SQL*Plus 会话：

```
ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
  2                a.paddr, a.status, b.program
  3       from v$session a left join v$process b
  4                on (a.paddr = b.addr)
  5       where a.username = 'OPS$TKYTE'
  6 /

USERNAME SID      SERIAL# SERVER PADDR          STATUS       PROGRAM
---------          ---       -------          -------          --------          ------          ----------
--------
OPS$TKYTE          150          261          SHARED       AE4CF118       ACTIVE
                  oracle@localhost(S000)

sys@ORA10G> connect system/manager@shared_server.us.oracle.com
Connected.
system@ORA10G> exec dbms_lock.sleep(20)

ops$tkyte@ORA10G> select a.username, a.sid, a.serial#, a.server,
  2                a.paddr, a.status, b.program
  3       from v$session a left join v$process b
```

```
  4                 on (a.paddr = b.addr)
  5         where a.username = 'OPS$TKYTE'
  6 /

USERNAME SID      SERIAL# SERVER PADDR          STATUS       PROGRAM
-----------------
-----     -------          ------------     --------------     ------------      -------------------------------
OPS$TKYTE            150            261            SHARED       AE4CF614         ACTIVE
                    oracle@localhost(S001)
```

注意第一次是怎么查询的，我使用了 S000 作为共享服务器。然后在另一个会话中执行了一个长时间运行的语句，它会独占这个共享服务器，而此时所独占的这个共享服务器恰好是 S000。要执行这个长时间运行的操作，这个工作会交给第一个空闲的共享服务器来完成，由于此时没有人要求使用 S000 共享服务器，所以 DBMS_LOCK 命令就选择了 S000。现在，再次在第一个 SQL*Plus 会话中查询时，将分配另一个共享服务器进程，因为 S000 共享服务器正在忙。

有一点很有意思，解析查询（尚未返回任何行）可能由共享服务器 S000 处理，第一行的获取可能由 S001 处理，第二行的获取可能由 S002 处理，而游标的关闭可能由 S003 负责。也就是说，一条语句可能由多个共享服务器一点一点地处理。

总之，从这一节可以了解到，一条连接（从客户到数据库实例的一条物理路径）上可以建立 0 个、1 个或多个会话。我们看到了这样一个用例，其中使用了 SQL*Plus 的 AUTOTRACE 工具。还有许多其他的工具也利用了这一点。例如，Oracle Forms 就使用一条连接上的多个会话来实现其调度功能。Oracle 的 n 层代理认证特性可用于提供从浏览器到数据库的端到端用户鉴别，这个特性也大量使用了有多个会话的单连接概念，但是每个会话中可能会使用一个不同的用户账户。我们已经看到，随着时间的推移，会话可能会使用多个进程，特别是在共享服务器环境中这种情况更常见。另外，如果使用 Oracle Net 的连接池，会话可能根本不与任何进程关联；连接空闲一段时间后，客户会将其删除，然后再根据检测活动（是否需要连接）透明地重建连接。

简单地说，连接和会话之间有一种多对多的关系。不过，最常见的是专用服务器与单一会话之间的一对一关系，这也是大多数人每天所看到的情况。

5.1.4 专用服务器与共享服务器

在继续介绍其他进程之前，下面先来讨论为什么会有两种连接模式，以及各个模式在哪些情况下更适用。

1. 什么时候使用专用服务器

前面提到过，在专用服务器模式中，客户连接与服务器进程之间存在一种一对一的映射。对于目前所有基于 SQL 的应用来说，这是应用连接 Oracle 数据库的最常用的方法。设置专用服务器最简单，而且采用这种方法建立连接也最容易，基本上不需要什么配置。

因为存在一对一的映射，所以不必担心长时间运行的事务会阻塞其他事务。其他事务

只通过其自己的专用进程来处理。因此，在非 OLTP 环境中，也就是可能有长时间运行事务的情况下，应该只考虑使用这种模式。专用服务器是 Oracle 的推荐配置，它能很好地扩缩。只要服务器有足够的硬件（CPU 和 RAM）来应对系统所需的专用服务器进程个数，专用服务器甚至可以用于数千条并发连接。

某些操作必须在专用服务器模式下执行，如数据库启动和关闭，所以每个数据库中可能同时有专用服务器和共享服务器，也可能只设置一个专用服务器。

2. 什么时候使用共享服务器

共享服务器的设置和配置尽管并不困难，但是比设置专用服务器要多一步。不过，二者之间的主要区别还不在于其设置；而是操作的模式有所不同。对于专用服务器，客户连接和服务器进程之间存在一对一的映射。对于共享服务器，则有一种多对一的关系：多个客户对应一个共享服务器。

顾名思义，共享服务器是一种共享资源，而专用服务器不是。使用共享资源时，必须当心，不要太长时间独占这个资源。如前所示，在会话中使用一个简单的 DBMS_LOCK.SLEEP(20)就会独占共享服务器进程 20 秒的时间。如果独占了共享服务器资源，会导致系统看上去好像挂起了一样。

图 5-2 中有两个共享服务器。如果我有 3 个客户，这些客户都试图同时运行一个 45 秒左右的进程，那么其中两个会在 45 秒内得到响应，第 3 个进程则会在 90 秒内才得到响应。这就是共享服务器的首要原则：要确保事务的持续时间尽量短。事务可以频繁地执行，但必须在短时间内执行完（这正是 OLTP 系统的特点）。如果事务持续时间很长，看上去整个系统都会慢下来，因为共享资源被少数进程独占着。在极端情况下，如果所有共享服务器都很忙，除了少数几个独占着共享服务器的幸运者以外，对其他用户来说，系统就好像挂起了。

使用共享服务器时，你可能还会观察到另一个有意思的现象，这就是人工死锁（artificial deadlock）。对于共享服务器，多个服务器进程被多个用户所“共享”，用户数量可能相当大。考虑下面这种情况，你有 5 个共享服务器，并建立了 100 个用户会话。现在，一个时间点上最多可以有 5 个用户会话是活动的。假设其中一个用户会话更新了某一行，但没有提交。正当这个用户呆坐在那里对是否修改还有些迟疑时，可能又有另外 5 个用户会话力图锁住这一行。当然，这 5 个会话会被阻塞，只能耐心地等待这一行可用。现在，原来的用户会话（它持有这一行的锁）试图提交事务，相应地释放行上的锁。这个用户会话发现所有共享服务器都已经被那 5 个 等待的会话所垄断。这就出现了一个人工死锁的情况：锁的持有者永远也拿不到共享服务器来完成提交，除非某个等待的会话放弃其共享服务器。但是，除非等待的会话所等待的是一个有超时时间的锁，否则它们绝对不会放弃其共享服务器（当然，你也可以让管理员通过一个专用服务器“杀死”（撤销）等待的会话来摆脱这种 困境）。

因此，由于这些原因，共享服务器只适用于 OLTP 系统，这种系统的特点是事务短而且频繁。在一个 OLTP 系统中，事务以毫秒为单位执行，任何事务的执行都会在 1 秒以内的片刻时间内完成。共享服务器对数据仓库很不适用，因为在数据仓库中，可能会执行耗时 1 分钟、2 分钟、5 分钟甚至更长时间的查询。如果采用共享服务器模式，其后果则是致命的。如果你的系统中 90%都是 OLTP，只有 10%“不那么 OLTP”，那么可以在同一个实例上适当地混合使用专用服务器和共享服务器。采用这种方式，可以大大减少机器上针对 OLTP 用户的服务器进程个数，并使得“不那么 OLTP”的用户不会独占共享服务器。另外，DBA 可以使用内置的资源管理器（Resource Manager）进一步控制资源利用率。

当然，使用共享服务器还有一个很重要的原因，这就是有时你别无选择。许多高级连

接特性都要求使用共享服务器。如果你想使用 Oracle Net 连接池，就必须使用共享服务器。如果你想在数据库之间使用数据库链接集合（database link concentration），也必须对这些连接使用共享服务器。

注意 如果你的应用中已经使用了一个连接池特性（例如，你在使用 J2EE 连 接池），而且适当地确定了连接池的大小，再使用共享服务器只会成为性能“杀手”，导致性能下降。你已经确定了连接池的大小，以适应任何时间点可能的并发连 接数，所以你希望这些连接都是直接的专用服务器连接。否则，在你应用的连接池特性中，只是“嵌套”了另一个连接池特性。

3. 共享服务器的潜在好处

前面说明了要当心哪些事务类型能使用共享服务器，那么，如果记住了这些原则，共享服务器能带来什么好处呢？共享服务器主要为我们做 3 件事：减少操作系统进程/线程数，刻意地限制并发度，以及减少系统所需的内存。在后面几节中将更详细地讨论这几点。

□ 减少操作系统进程/线程数

在一个有数千个用户的系统上，如果操作系统力图管理数千个进程，可能很快就会疲于奔命。在一个典型的系统中，尽管有数千个用户，但任何时间点上只有其中很少一部分用户同时是活动的。例如，我最近参与开发了一个有 5 000 个并发用户的系统。在任何时间点上，最多只有 50 个用户是活动的。这个系统利用 50 个共享服务器进程就能有效地工作，这使得操作系统必须管理的进程数下降了两个数量级（100 倍）。从很大程度上讲，这样一来，操作系统可以避免上下文切换。

□ 刻意地限制并发度

由于我曾经参加过大量的基准测试，所以在这方面我应该很有发言权。运行基准测试时，人们经常要求支持尽可能多的用户，直至系统崩溃。这些基准测试的输出之一往往是一张图表，显示出并发用户数和事务数之间的关系（见图 5-3）。

图 5-3 并发用户数与每秒事务数

最 初，增加并发用户数时，事务数会增加。不过到达某一点时，即使再增加额外的用

户，也不会增加每秒完成的事务数，图中的曲线开始下降。吞吐量有一个峰值，从 这个峰值开始，响应时间开始增加（你每秒完成的事务数还是一样，但是最终用户观察到响应时间变慢）。随着用户数继续增加，会发现吞吐量实际上开始下降。这 个下降点之前的并发用户数就是系统上允许的最大并发度。从这一点开始，系统开始出现拥塞，为了完成工作，用户将开始排队。就像收费站出现的阻塞一样，系统 将无法支撑。此时不仅响应时间开始大幅增加，系统的吞吐量也可能开始下跌，另外上下文切换要占用资源，而且在这么多消费者之间共享资源就存在着开销，这本 身也会占用额外的资源。如果把最大并发度限制为这个下降点之前的某一点，就可以保持最大吞吐量，并尽可能减少大多数用户的响应时间。利用共享服务器，我们 就能把系统上的最大并发度限制为这个合适的数。

可以把这个过程比作一扇简单的门。门有宽度，人也有宽度，这就限制了最大吞吐量，即每分钟最多可以有多少人通过这扇门。如果“负载”低，就没有什么问题；不过，随着越来越多的人到来，就会要求某些人等待（CPU 时 间片）。如果很多人都想通过这扇门，反而会“退步”，假如有太多的人在你身后排队，吞吐量就开始下降。每个人通过时都会有延迟。使用队列意味着吞吐量增 加，有些人会很快地通过这扇门，就好像没有队列一样，而另外一些人（排在队尾的人）却要忍受最大的延迟，焦燥地认为“这是一个不好的做法”。但实际上，在 测量每个人（包括最后一个人）通过门的速度时，排队模型（共享服务器）确实比“各行其是”（free-for-all）的方法要好。如果是各行其是，即使大家都很文明，也不乏有这样的情形：商店大降价时，开门后，每个人都争先恐后地涌入大门，结果却是大家都挤不动。

□ **减少系统所需的内存**

这是使用共享服务器最主要的原因之一：它能减少所需的内存量。共享服务器确实能减少内存需求，但是没有你想象中的那么显著，特别是假设采用了第 4 章介绍的自动 PGA 内存管理，那么共享服务器在减少内存需求方面更没有太大意义。自动 PGA 内存管理是指，为进程分配工作区后，使用完毕就释放，而且所分配工作区的大小会根据并发工作负载而变化。所以，这个理由在较早版本的 Oracle 中还算合理，但对当前来讲意义不大。另外要记住，使用共享服务器时，UGA 在 SGA 中分配。这说明，转变为共享服务器时，必须能准确地确定需要多少 UGA 内存，并适当地在 SGA 中分配（通过 LARGE_POOL_SIZE 参数）。所以，共享服务器配置中对 SGA 的需求通常很大。这个内存一般要预分配，从而只能由数据库实例使用。

注意 对于大小可以调整的 SGA，随着时间的推移确实可以扩大或收缩这个内存，但是在大多数情况下，它会由数据库实例所“拥有”，其他进程不能使用。

专用服务器与此正好相反，任何人都可以使用未分配给 SGA 的任何内存。那么，既然 UGA 在 SGA 中分配而使得 SGA 相当大，又怎么能节省内存呢？ 之所以能节省内存，这是因为共享服务器会分配更少的 PGA。每个专用/共享服务器都有一个 PGA，即进程信息。PGA 是排序区、散列区以及其他与进程相关的结构。采用共享服务器，就可以从系统去除这部分内存的需求。如果从使用 5 000 个专用服务器发展到使用 100 个共享服务器，那么通过使用共享服务器，累计起来就能节省 4 900 个 PGA 的内存（但不包括其 UGA）。

5.1.5 专用/共享服务器小结

除非你的系统负载过重，或者需要为一个特定的特性使用共享服务器，否则专用服务器可能最合适。专用服务器设置起来很简单（实际上，根本没有设置！），而且调优也更容易。

注意 对于共享服务器连接，会话的跟踪信息（SQL_TRACE=TRUE 时的输出）可能分布

在多个独立的跟踪文件中，重建会话可能更困难。

如 果用户群很大，而且你知道要部署共享服务器，强烈建议你先开发并测试这个共享服务器。如果只是在一个专用服务器下开发，而且从来没有对共享服务器进行测 试，出现失败的可能性会更大。要对系统进行压力测试，建立基准测试，并确保应用使用共享服务器时也能很好地工作。也就是说，要确保它不会独占共享服务器太 长时间。如果在开发时发现它会长时间地独占共享服务器，与等到部署时才发现相比，修正会容易得多。你可以使用诸如高级排队（Advanced Queuing，AQ）之类的特性使长时间运行的进程变得很短，但是必须在应用中做相应的设计。这种工作最好在开发时完成。另外，共享服务器连接和专用服务器连接可用的特性集之间还有一些历史上的差别。例如，我们已经讨论过 Oracle 9i 中没有自动 PGA 内存管理，但是过去有的一些基本特性（如两个表之间的散列联结）在共享服务器连接中反倒没有了。

5.2 后台进程

Oracle 实例包括两部分：SGA 和 一组后台进程。后台进程执行保证数据库运行所需的实际维护任务。例如，有一个进程为我们维护块缓冲区缓存，根据需要将块写出到数据文件。另一个进程负责当 在线重做日志文件写满时将它复制到一个归档目标。另外还有一个进程负责在异常中止进程后完成清理，等等。每个进程都专注于自己的任务，但是会与所有其他进 程协同工作。例如，负责写日志文件的进程填满一个日志后转向下一个日志时，它会通知负责对填满的日志文件进行归档的进程，告诉它有活干了。

可以使用一个 V$视图查看所有可能的 Oracle 后台进程，确定你的系统中正在使用哪些后台进程：

```
ops$tkyte@ORA9IR2> select paddr, name, description
  2 from v$bgprocess
  3 order by paddr desc
  4 /

PADDR       NAME DESCRIPTION
--------    ----    ----------------------------------------------------------
5F162548 ARC1    Archival Process 1
5F162198 ARC0    Archival Process 0
5F161A38 CJQ0    Job Queue Coordinator
5F161688 RECO distributed recovery
```

```
5F1612D8              SMON       System Monitor Process

5F160F28 CKPT     checkpoint

5F160B78 LGWR Redo etc.

5F1607C8              DBW0       db writer process 0

5F160418 PMON process cleanup

00              DIAG      diagnosibility process

00              FMON File Mapping Monitor Process

00              LMON global enqueue service monitor

00              LMD0     global enqueue service daemon 0

...

00              LMS7     global cache service process 7

00              LMS8     global cache service process 8

00              LMS9     global cache service process 9

69 rows selected.
```

这个视图中 PADDR 不是 00 的行都是系统上配置和运行的进程（线程）。

有两类后台进程：有一个中心（focused）任务的进程（如前所述）以及完成各种其他任务的进程（即工具进程）。例如，内部作业队列（job queue）有一个工具后台进程，可以通过 DBMS_JOB 包使用它。这个进程会监视作业队列，并运行其中的作业。在很多方面，这就像一个专用服务器进程，但是没有客户连接。下面会分析各种后台进程，先来看有中心任务的进程，然后再介绍工具进程。

5.2.1 中心后台进程

图 5-4 展示了有一个中心（focused）用途的 Oracle 后台进程。

图 5-4 中心后台进程

启动实例时也许不会看到所有这些进程，但是其中一些主要的进程肯定存在。如果在 ARCHIVELOG 模式下，你可能只会看到 ARCn（归档进程），并启用自动归档。如果运行了 Oracle RAC，这种 Oracle 配置允许一个集群中不同机器上的多个实例装载并打开相同的物理数据库，就只会看到 LMD0、LCKn、LMON 和 LMSn（稍后会更详细地介绍这些进程）。

注意 为简洁起见，图 5-4 中没有画出共享服务器调度器（Dnnn）和共享服务器（Snnn）进程。

因此，图 5-4 大致展示了启动 Oracle 实例并装载和打开一个数据库时可能看到哪些进程。例如，在我的 Linux 系统上，启动实例后，有以下进程：

```
$ ps -aef | grep 'ora_.*_ora10g$'

ora10g 5892    1    0    16:17    ?        00:00:00        ora_pmon_ora10g

ora10g 5894    1    0    16:17    ?        00:00:00        ora_mman_ora10g

ora10g 5896    1    0    16:17    ?        00:00:00        ora_dbw0_ora10g

ora10g 5898    1    0    16:17    ?        00:00:00        ora_lgwr_ora10g

ora10g 5900    1    0    16:17    ?        00:00:00        ora_ckpt_ora10g

ora10g 5902    1    0    16:17    ?        00:00:00        ora_smon_ora10g

ora10g 5904    1    0    16:17    ?        00:00:00        ora_reco_ora10g

ora10g 5906    1    0    16:17    ?        00:00:00        ora_cjq0_ora10g
```

```
ora10g 5908  1  0  16:17  ?  00:00:00  ora_d000_ora10g
ora10g 5910  1  0  16:17  ?  00:00:00  ora_s000_ora10g
ora10g 5916  1  0  16:17  ?  00:00:00  ora_arc0_ora10g
ora10g 5918  1  0  16:17  ?  00:00:00  ora_arc1_ora10g
ora10g 5920  1  0  16:17  ?  00:00:00  ora_qmnc_ora10g
ora10g 5922  1  0  16:17  ?  00:00:00  ora_mmon_ora10g
ora10g 5924  1  0  16:17  ?  00:00:00  ora_mmnl_ora10g
ora10g 5939  1  0  16:28  ?  00:00:00  ora_q000_ora10g
```

这些进程所用的命名约定很有意思。进程名都以 ora_开头。后面是 4 个字符，表示进程的具体名字，再后面是_ora10g。因为我的 ORACLE_SID（站点标识符）是 ora10g。在 UNIX 上，可以很容易地标识出 Oracle 后台进程，并将其与一个特定的实例关联（在 Windows 上则没有这么容易，因为在 Windows 上这些后台进程实际上只是一个更大进程中的线程）。最有意思的是（但从前面的代码可能不太容易看出来），这些进程实际上都是同一个二进制可执行程序，对于每个“程序”，并没有一个单独的可执行文件。你可以尽可能地查找一下，但是不论在磁盘的哪个位置上肯定都找不到一个 arc0 二进制可执行程序，同样也找不到 LGWR 或 DBW0。这些进程实际上都是 oracle（也就是所运行的二进制可执行程序的名字）。它们只是在启动时对自己建立别名，以便更容易地标识各个进程。这样就能在 UNIX 平台上高效地共享大量对象代码。Windows 上就没有什么特别的了，因为它们只是进程中的线程，因此，当然只是一个大的二进制文件。

下面来介绍各个进程完成的功能，先从主要的 Oracle 后台进程开始。

1. PMON：进程监视器（Process Monitor）

这个进程负责在出现异常中止的连接之后完成清理。例如，如果你的专用服务器“失败”或者出于某种原因被撤销，就要由 PMON 进程负责修正（恢复或撤销工作），并释放你的资源。PMON 会回滚未提交的工作，并释放为失败进程分配的 SGA 资源。

除了出现异常连接后完成清理外，PMON 还负责监视其他的 Oracle 后台进程，并在必要时（如果可能的话）重启这些后台进程。如果一个共享服务器或调度器失败（崩溃），PMON 会介入，并重启另一个共享服务器或调度器（对失败进程完成清理之后）。PMON 会查看所有 Oracle 进程，可能重启这些进程，也可能适当地终止实例。例如，如果数据库日志写入器进程（LGWR）失败，就最好让实例失败。这是一个严重的错误，最安全的做法就是立即终止实例，并根据正常的恢复来修正数据（注意，这是一种很罕见的情况，要立即报告给 Oracle Support）。

PMON 还会为实例做另一件事，这就是向 Oracle TNS 监听器注册这个实例。实例启动时，PMON 进程会询问公认的端口地址（除非直接指定），来查看是否启动并运行了一个监听器。Oracle 使用的公认/默认端口是 1521。如果此时监听器在另外某个端口上启动会怎么样呢？在这种情况下，原理是一样的，只不过需要设置 LOCAL_LISTENER 参数来显式地指定监听器地址。如果数据库实例启动时有监听器在运行，PMON 会与这个监听器通信，

并向它传递相关的参数，如服务名和实例的负载度量等。如果监听器未启动，PMON 则会定期地试图与之联系来注册实例。

2. SMON：系统监视器（System Monitor）

SMON 进程要完成所有“系统级”任务。PMON 感兴趣的是单个的进程，而 SMON 与之不同，它以系统级为出发点，这是一种数据库“垃圾收集器”。SMON 所做的工作包括：

- 清理临时空间：原先清理临时空间这样的杂事都要由我们来完成，随着引入了“真正” 的临时表空间，这个负担已经减轻，但并不是说完全不需要清理临时空间。例如，建立一个索引时，创建时为索引分配的区段标记为 TEMPORARY。如果出于某种原因 CREATE INDEX 会话中止了，SMON 就要负责清理。其他操作创建的临时区段也要由 SMON 负责清理。
- 合并空闲空间：如果你在使用字典管理的表空间，SMON 要负责取得表空间中相互连续的空闲区段，并把它们合并为一个更大的空闲区段。只有当字典管理的表空间有一个默认的存储子句，而且 pctincrease 设置为一个非 0 值时，才会出现空闲空间的合并。
- 针对原来不可用的文件恢复活动的事务：这类似于数据库启动时 SMON 的作用。在实例/崩溃恢复时由于某个文件（或某些文件）不可用，可能会跳过一些失败的事务（即无法恢复），这些失败事务将由 SMON 来恢复。例如，磁盘上的文件可能不可用或者未装载，当文件确实可用时，SMON 就会由此恢复事务。
- 执行 RAC 中失败节点的实例恢复：在一个 Oracle RAC 配置中，集群中的一个数据库实例失败时（例如，实例所在的主机失败），集群中的另外某个节点会打开该失败实例的重做日志文件，并为该失败实例完成所有数据的恢复。
- 清理 OBJ$：OBJ$是一个低级数据字典表，其中几乎对每个对象（表、索引、触发器、视图等）都包含一个条目。很多情况下，有些条目表示的可能是已经删除的对象，或者表示“not there”（不在那里）对象（“not there”对象是 Oracle 依赖机制中使用的一种对象）。要由 SMON 进程来删除这些不再需要的行。
- 收缩回滚段：如果有设置，SMON 会自动将回滚段收缩为所设置的最佳大小。
- “离线”回滚段：DBA 有可能让一个有活动事务的回滚段离线（offline），或置为不可用。也有可能活动事务会使用离线的回滚段。在这种情况下，回滚段并没有真正离线；它只是标记为“将要离线”。在后台，SMON 会定期尝试真正将其置为离线，直至成功为止。

从以上介绍你应该对 SMON 做些什么有所认识了。除此之外，它还会做许多其他的事情 如将 DBA_TAB_MONITORING 视图中的监视统计信息刷新输出，将 SMON_SCN_TIME 表中的 SCN-时间戳映射信息刷新输出，等等。随着时间的推移，SMON 进程可能会累积地占用大量 CPU 时间，这应该是正常的。SMON 会定期地醒来（或者被其他后台进程唤醒），来执行这些维护工作。

3. RECO：分布式数据库恢复（Distributed Database Recovery）

RECO 有一个很中心的任务：由于两段提交（two-phase commit，2PC）期间的崩溃或连接丢失等原因，有些事务可能会保持准备状态，这个进程就是要恢复这些事务。2PC 是一种分布式协议，允许影响多个不同数据库的修改实现原子提交。它力图在提交之前尽可能地关闭分布式失败窗口[4]。如果在 N 个数据库之间采用 2PC，其中一个数据库（通常是客户

最初登录的那个数据库，但也不一定）将成为协调器（coordinator）。这个站点会询问其他 N □ 1 个站点是否准备提交。实际上，这个站点会转向另外这 N □ 1 个站点，问它们是否准备好提交。这 N □ 1 个站点都会返回其“准备就绪状态”，报告为 YES 或 NO[5]。如果任何一个站点投票（报告）NO，整个事务都要回滚。如果所有站点都投票 YES，站点协调器就会广播一条消息，使这 N □ 1 个站点真正完成提交（提交得到持久地存储）。

如果某个站点投票 YES，称其准备好要提交，但是在此之后，并且在得到协调器的指令真正提交之前，网络失败了，或者出现了另外某个错误，事务就会成为一个可疑的分布式事务（in-doubt distributed transaction）。2PC 力图限制出现这种情况的时间窗口，但是无法根除这种情况。如果正好在那时（这个时间窗口内）出现一个失败，处理事务的工作就要由 RECO 负责。RECO 会试图联系事务的协调器来发现协调的结果。在此之前，事务会保持未提交状态。当再次到达事务协调器时，RECO 可能会提交事务，也可能将事务回滚。

需要说明，如果失败（outrage）持续很长一段时间，而且你有一些很重要的事务，可以自行手动地提交或回滚。有时你可能想要这样做，因为可疑的分布式事务可能导致写入器阻塞读取器（Oracle 中只有此时会发生“写阻塞读”的情况）。你的 DBA 可以通知另一个数据库的 DBA，要求他查询那些可疑事务的状态。然后你的 DBA 再提交或回滚，而不再由 RECO 完成这个任务。

4. CKPT：检查点进程（Checkpoint Process）

检查点进程并不像它的名字所暗示的那样真的建立检查点（检查点在第 3 章介绍重做日志一节中已经讨论过），建立检查点主要是 DBWn 的任务。CKPT 只是更新数据文件的文件首部，以辅助真正建立检查点的进程（DBWn）。以前 CKPT 是一个可选的进程，但从 8.0 版本开始，这个进程总会启动，所以，如果你在 UNIX 上执行一个 ps 命令，就肯定能看到这个进程。原先，用检查点信息更新数据文件首部的工作是 LGWR 的任务；不过，一段时间后，随着数据库大小的增加以及文件个数的增加，对 LGWR 来说，这个额外的工作负担太重了。如果 LGWR 必须更新数十个、数百个甚至数千个文件，等待提交事务的会话就可能必须等待太长的时间。有了 CKPT，这个任务就不用 LGWR 操心了。

5. DBWn：数据库块写入器（Database Block Writer）

数据库块写入器（DBWn）是负责将脏块写入磁盘的后台进程。DBWn 会写出缓冲区缓存中的脏块，通常是为了在缓存中腾出更多的空间（释放缓冲区来读入其他数据），或者是为了推进检查点（将在线重做日志文件中的位置前移，如果出现失败，Oracle 会从这个位置开始读取来恢复实例）。如第 3 章所述，Oracle 切换日志文件时就会标记（建立）一个检查点。Oracle 需要推进检查点，推进检查点后，就不再需要它刚填满的在线重做日志文件了。如果需要重用这个重做日志文件，而此时它还依赖于原来的重做日志文件，我们就会得到一个“检查点未完成”消息，而必须等待。

注意 推进日志文件只是导致检查点活动的途径之一。有一些增量检查点由诸如 FAST_START_ MTTR_TARGET 之类的参数以及导致脏块刷新输出到磁盘的其他触发器控制。

可以看到，DBWn 的性能可能很重要。如果它写出块的速度不够快，不能很快地释放缓冲区（可以重用来缓存其他块），就会看到 Free Buffer Waits 和 Write Complete Waits 的等待数和等待时间开始增长。

可以配置多个 DBWn；实际上，可以配置多达 20 个 DBWn（DBW0…DBW9、DBWa…DBWj）。大多数系统都只有一个数据库块写入器，但是更大的多 CPU 系统可以利用多个块

写入器。通常为了保持 SGA 中的一个大缓冲区缓存“干净”，将脏的（修改过的）块刷新输出到磁盘，就可以利用多个 DBWn 来分布工作负载。

最好的情况下，DBWn 使用异步 I/O 将块写至磁盘。采用异步 I/O，DBWn 会收集一批要写的块，并把它们交给操作系统。DBWn 并不等待操作系统真正将块写出；而是立即返回，并收集下一批要写的块。当操作系统完成写操作时，它会异步地通知 DBWn 写操作已经完成。这样，与所有工作都串行进行相比，DBWn 可以更快地工作。在后面的“从属进程”一节中，我们还将介绍如何使用 I/O 从属进程在不支持异步 I/O 的平台或配置上模拟异步 I/O。

关于 DBWn，还有最后一点需要说明。根据定义，块写入器进程会把块写出到所有磁盘，即分散在各个磁盘上；也就是说，DBWn 会做大量的分散写（scattered write）。执行一个更新时，你会修改多处存储的索引块，还可能修改随机地分布在磁盘上的数据块。另一方面，LGWR 则是向重做日志完成大量的顺序写（sequential write）。这是一个很重要的区别，Oracle 之所以不仅有一个重做日志和 LGWR 进程，还有 DBWn 进程，其原因就在于此。分散写比顺序写慢多了。通过在 SGA 中缓存脏块，并由 LGWR 进程完成大规模顺序写（可能重建这些脏缓冲区），这样可以提升性能。DBWn 在后台完成它的任务（很慢），而 LGWR 在用户等待时完成自己的任务（这个任务比较快），这样我们就能得到更好的整体性能。尽管从技术上讲这样会使 Oracle 执行更多不必要的 I/O（写日志以及写数据文件），但整体性能还是会提高。从理论上讲，如果提交期间 Oracle 已经将已修改的块物理地写出到磁盘，就可以跳过写在线重做日志文件。但在实际中并不是这样[6]：LGWR 还是会把每个事务的重做信息写至在线重做日志，DBWn 则在后台将数据库块刷新输出到磁盘。

6. LGWR：日志写入器（Log Writer）

LGWR 进程负责将 SGA 中重做日志缓冲区的内容刷新输出到磁盘。如果满足以下某个条件，就会做这个工作：

- ☐ 每 3 秒会刷新输出一次
- ☐ 任何事务发出一个提交时
- ☐ 重做日志缓冲区 1/3 满，或者已经包含 1 MB 的缓冲数据

由于这些原因，分配超大的（数百 MB）重做日志缓冲区并不实际，Oracle 根本不可能完全使用这个缓冲区。日志会通过顺序写来写至磁盘，而不像 DBWn 那样必须执行分散 I/O。与向文件的各个部分执行多个分散写相比，像这样大批的写会高效得多。这也是使用 LGWR 和重做日志的主要原因。通过使用顺序 I/O，只写出有变化的字节，这会提高效率；尽管可能带来额外的 I/O，但相对来讲所提高的效率更为突出。提交时，Oracle 可以直接将数据库块写至磁盘，但是这需要对已满的块执行大量分散 I/O，而让 LGWR 顺序地写出所做的修改要比这快得多。

7. ARCn：归档进程（Archive Process）

ARCn 进程的任务是：当 LGWR 将在线重做日志文件填满时，就将其复制到另一个位置。此后这些归档的重做日志文件可以用于完成介质恢复。在线重做日志用于在出现电源故障（实例终止）时“修正”数据文件，而归档重做日志则不同，它是在出现硬盘故障时用于“修正”数据文件。如果丢失了包含数据文件/d01/oradata/ora10g/system.dbf 的磁盘，可以去找上一周的备份，恢复旧的文件副本，并要求在数据库上应用自这次备份之后生成的所有归档和在线重做日志。这样就能使这个数据文件“赶上”数据库中的其他数据文件，所以我们

可以继续处理而不会丢失数据。

ARCn 通 常将在线重做日志文件复制到至少两个位置（冗余正是不丢失数据的关键所在!）。这些位置可能是本地机器上的磁盘，或者更确切地讲，至少有一个位置在另一台 机器上，以应付灾难性的失败。在许多情况下，归档重做日志文件会由另外某个进程复制到一个第三辅存设备上，如磁带。也可以将这些归档重做日志文件发送到另 一台机器上，应用于“备用数据库”（standby database），这是 Oracle 提供的一个故障转移选项。稍后将讨论其中涉及的进程。

8. 其他中心进程

取决于所用的 Oracle 特性，可能还会看到其他一些中心进程。这里只是简单地列出这些进程，并提供其功能的简要描述。前面介绍的进程都是必不可少的，如果运行一个 Oracle 实例，就肯定会有这些进程。以下进程则是可选的，只有在利用了某个特定的特性时才会出现。下面的进程是使用 ASM 的数据库实例所特有的，见第 3 章的讨论：

- 自动存储管理后台（Automatic Storage Management Background，ASMB）进程：ASMB 进程在使用了 ASM 的数据库实例中运行。它负责与管理存储的 ASM 实例通信、向 ASM 实例提供更新的统计信息，并向 ASM 实例提供一个“心跳”，让 ASM 实例知道它还活着，而且仍在运行。

- 重新平衡(Rebalance，RBAL)进程：RBAL 进程也在使用了 ASM 的数据库实例中运行。向 ASM 磁盘组增加或去除磁盘时，RBAL 进程负责处理重新平衡请求（即重新分布负载的请求）。

- 以下进程出现在 Oracle RAC 实例中。RAC 是一种 Oracle 配置，即集群中的多个实例可以装载和打开一个数据库，其中每个实例在一个单独的节点上运行（通常节点是一个单独的物理计算机）。这样，你就能有多个实例访问（以一种全读写方式）同样的一组数据库文件。RAC 的主要目标有两个：

- 高度可用性：利用 Oracle RAC，如果集群中的一个节点/计算机由于软件、硬件或人为错误而失败，其他节点可以继续工作，还可以通过其他节点访问数据库。你也许会丧失一些计算能力，但是不会因此而无法访问数据库。

- 可扩缩性：无需购买更大的机器来处理越来越大的工作负载（这称为垂直扩缩），RAC 允许以另一种方式增加资源，即在集群中增加更多的机器（称为水平扩缩）。举例来说，不必把你的 4 CPU 机器扩缩为有 8 个或 16 个 CPU，通过利用 RAC，你可以选择增加另外一个相对廉价的 4 CPU 机器（或多台这样的机器）。

以下进程是 RAC 环境所特有的，如果不是 RAC 环境，则看不到这些进程。

- 锁监视器（Lock monitor，LMON）进程：LMON 监视集群中的所有实例，检测是否有实例失败。这有利于恢复失败实例持有的全局锁。它还负责在实例离开或加入集群时重新配置锁和其他资源（实例失败时会离开集群，恢复为在线时又会加入集群，或者可能有新实例实时地增加到集群中）。

- 锁管理器守护（Lock manager daemon，LMD）进程：LMD 进程为全局缓存服务（保持块缓冲区在实例间一致）处理锁管理器服务请求。它主要作为代理（broker）向一个队列发出资源请求，这个队列由 LMSn 进程处理。LMD 会处理全局死锁的检测/解析，并监视全局环境中的锁超时。

- 锁管理器服务器（Lock manager server，LMSn）进程：前面已经提到，在一个

RAC 环境中，各个 Oracle 实例在集群中的不同机器上运行，它们都以一种读写方式访问同样的一组数据库文件。为了达到这个目的，SGA 块缓冲区缓存相互之间必须保持一致。这也是 LMSn 进程的主要目标之一。在以前版本的 Oracle 并行服务器（Oracle Parallel Server，OPS）中，这是通过 ping 实现的。也就是说，如果集群中的一个节点需要块的一个读一致视图，而这个块以一种独占模式被另一个节点锁定，数据的交换就要通过磁盘刷新输出来完成（块被 ping）。如果本来只是要读取数据，这个操作（ping）的代价就太昂贵了。现在则不同，利用 LMSn，可以在集群的高速连接上通过非常快速的缓存到缓存交换来完成数据交换。每个实例可以有多达 10 个 LMSn 进程。

- 锁（Lock，LCK0)进程：这个进程的功能与前面所述的 LMD 进程非常相似，但是它处理所有全局资源的请求，而不只是数据库块缓冲区的请求。
- 可诊断性守护（Diagnosability daemon，DIAG）进程：DIAG 只能用于 RAC 环境中。它负责监视实例的总体“健康情况”，并捕获处理实例失败时所需的信息。

5.2.2 工具后台进程

这些后台进程全都是可选的，可以根据你的需要来选用。它们提供了一些工具，不过这些工具并不是每天运行数据库所必需的，除非你自己要使用（如作业队列），或者你要利用使用了这些工具的特性（如新增的 Oracle 10g 诊断功能）。

在 UNIX 中，这些进程可以像其他后台进程一样可见，如果你执行 ps 命令，就能看到这些进程。在介绍中心后台进程那一节的开始，我列出了 ps 命令的执行结果（这里列出其中一部分），可以看到，我有以下进程：

- 配置了作业队列。CJQ0 进程是作业队列协调器（job queue coordinator）。
- 配置了 Oracle AQ，从 Q000（AQ 队列进程，AQ queue process）和 QMNC（AQ 监视器进程，AQ monitor process）可以看出。
- 启用了自动设置 SGA 大小，由内存管理器（memory manager ，MMAN）进程可以看出。
- 启用了 Oracle 10g 可管理性/诊断特性，由可管理性监视器（manageability monitor，MMON）和可管理性监视器灯（manageability monitor light，MMNL）进程可以看出。

```
ora10g 5894 1 0 16:17 ?     00:00:00     ora_mman_ora10g

ora10g 5906 1 0 16:17 ?     00:00:00     ora_cjq0_ora10g

ora10g 5920 1 0 16:17 ?     00:00:00     ora_qmnc_ora10g

ora10g 5922 1 0 16:17 ?     00:00:00     ora_mmon_ora10g

ora10g 5924 1 0 16:17 ?     00:00:00     ora_mmnl_ora10g

ora10g 5939 1 0 16:28 ?     00:00:00     ora_q000_ora10g
```

下面来看看根据所用的特性可能会看到哪些进程。

1. CJQ0 和 Jnnn 进程：作业队列

在第一个 7.0 版本中，Oracle 通过一种称为快照（snapshot）的数据库对象来提供复制特性。作业队列就是刷新快照（或将快照置为当前快照）时使用的内部机制。

作业队列进程监视一个作业表，这个作业表告诉它何时需要刷新系统中的各个快照。在 Oracle 7.1 中，Oracle 公司通过一个名为 DBMS_JOB 的数据库包来提供这个功能。所以，原先 7.0 中与快照相关的进程到了 7.1 及以后版本中变成了“作业队列”。后来，控制作业队列行为的参数（检查的频度，以及应该有多少个队列进程）的名字也发生了变化，从 SNAPSHOT_REFRESH_INTERVAL 和 SNAPSHOT_REFRESH_PROCESSES 变成了 JOB_QUEUE_INTERVAL 和 JOB_QUEUE_PROCESSES。在当前的版本中，只有 JOB_QUEUE_PROCESSES 参数的设置是用户可调的。

最多可以有 1 000 个作业队列进程。名字分别是 J000，J001，…，J999。这些进程在复制中大量使用，并作为物化视图刷新进程的一部分。基于流的复制（Oracle9i Release 2 中新增的特性）使用 AQ 来 完成复制，因此不使用作业队列进程。开发人员还经常使用作业队列来调度一次性（后台）作业或反复出现的作业，例如，在后台发送一封电子邮件，或者在后台完 成一个长时间运行的批处理。通过在后台做这些工作，就能达到这样一种效果：尽管一个任务耗时很长，但在性急的最终用户看来所花费的时间并不多（他会认为任 务运行得快多了，但事实上可能并非如此）。这与 Oracle 用 LGWR 和 DBWn 进程所做的工作类似，他们在后台做大量工作，所以你不必实时地等待它们完成所有任务。

Jnnn 进程与共享服务器很相似，但是也有专用服务器中的某些方面。它们处理完一个作业之后再处理下一个作业，从这个意义上讲是共享的，但是它们管理内存的方式更像是一个专用服务器（其 UGA 内存在 PGA 中，而不是在 SGA 中）。每个作业队列进程一次只运行一个作业，一个接一个地运行，直至完成。正因为如此，如果我们想同时运行多个作业，就需要多个进程。这里不存在多线程或作业的抢占。一旦运行一个作业，就会一直运行到完成（或失败）。

你会注意到，经过一段时间，Jnnn 进程会不断地来来去去，也就是说，如果配置了最多 1 000 个 Jnnn 进程，并不会看到真的有 1 000 个进程随数据库启动。相反，开始时只会启动一个进程，即作业队列协调器（CJQ0），它在作业队列表中看到需要运行的作业时，会启动 Jnnn 进程。如果 Jnnn 进程完成其工作，并发现没有要处理的新作业，此时 Jnnn 进程就会退出，也就是说，会消失。因此，如果将大多数作业都调度为在凌晨 2:00 运行（没有人在场），你可能永远也看不到这些 Jnnn 进程。

2. QMNC 和 Qnnn：高级队列

QMNC 进程对于 AQ 表来说就相当于 CJQ0 进程之于作业表。QMNC 进程会监视高级队列，并警告从队列中删除等待消息的“出队进程”（dequeuer）：已经有一个消息变为可用。QMNC 和 Qnnn 还要负责队列传播（propagation），也就是说，能够将在一个数据库中入队（增加）的消息移到另一个数据库的队列中，从而实现出队（dequeueing）。

Qnnn 进程对于 QMNC 进程就相当于 Jnnn 进程与 CJQ0 进程的关系。QMNC 进程要通知 Qnnn 进程需要完成什么工作，Qnnn 进程则会处理这些工作。

QMNC 和 Qnnn 进程是可选的后台进程。参数 AQ_TM_PROCESSES 可以指定最多创建 10 个这样的进程（分别名为 Q000，…，Q009），以及一个 QMNC 进程。如果 AQ_TM_PROCESSES 设置为 0，就没有 QMNC 或 Qnnn 进程。不同于作业队列所用的 Jnnn 进程，Qnnn 进程是持久的。如果将 AQ_TM_PROCESSES 设置为 10，数据库启动时可以看

到 10 个 Qnnn 进程和一个 QMNC 进程，而且在实例的整个生存期中这些进程都存在。

3. EMNn：事件监视器进程（Event Monitor Process）

EMNn 进程是 AQ 体系结构的一部分，用于通知对某些消息感兴趣的队列订购者。通知会异步地完成。可以用一些 Oracle 调用接口（Oracle Call Interface，OCI）函数来注册消息通知的回调。回调是 OCI 程序中的一个函数，只要队列中有了订购者感兴趣的消息，就会自动地调用这个函数。EMNn 后台进程用于通知订购者，第一次向实例发出通知时会自动启动 EMNn 进程。然后应用可以发出一个显式的 message_receive(dequeue)来获取消息。

4. MMAN：内存管理器（Memory Manager）

这个进程是 Oracle 10g 中新增的，自动设置 SGA 大小特性会使用这个进程。MMAN 进程用于协调共享内存中各组件（默认缓冲区池、共享池、Java 池和大池）的大小设置和大小调整。

5. MMON、MMNL 和 Mnnn：可管理性监视器（Manageability Monitor）

这些进程用于填充自动工作负载存储库（Automatic Workload Repository，AWR），这是 Oracle 10g 中新增的一个特性。MMNL 进程会根据调度从 SGA 将统计结果刷新输出至数据库表。MMON 进程用于“自动检测”数据库性能问题，并实现新增的自调整特性。Mnnn 进程类似于作业队列的 Jnnn 或 Qnnn 进程；MMON 进程会请求这些从属进程代表它完成工作。Mnnn 进程本质上是临时性的，它们将根据需要来来去去。

6. CTWR：修改跟踪进程（Change Tracking Process）

这是 Oracle 10g 数据库中新增的一个可选进程。CTWR 进程负责维护新的修改跟踪文件，有关内容见第 3 章的介绍。

7. RVWR：恢复写入器（Recovery Writer）

这个进程也是 Oracle 10g 数据库中新增的一个可选进程，负责维护闪回恢复区中块的“前”映像（见第 3 章的介绍），要与 FLASHBACK DATABASE 命令一起使用。

8. 其他工具后台进程

这就是完整的列表吗？不，还有另外一些工具进程没有列出。例如，Oracle Data Guard 有一组与之相关的进程，有利于将重做信息从一个数据库移送到另一个数据库，并应用这些重做信息（详细内容请见 Oracle 的 Data Guard Concepts and Administration Guide）。还有一些进程与 Oracle 10g 新增的数据泵工具有关，在某些数据泵操作中会看到这些进程。另外还有一些流申请和捕获进程。不过，以上所列已经基本涵盖了你可能遇到的大多数常用的后台进程。

5.3 从属进程

下面来看最后一类 Oracle 进程：从属进程（slave process）。Oracle 中有两类从属进程：I/O 从属进程和并行查询从属进程。

5.3.1 I/O 从属进程

I/O 从属进程用于为不支持异步 I/O 的系统或设备模拟异步 I/O。例如，磁带设备（相当慢）就不支持异步 I/O。通过使用 I/O 从属进程，可以让磁带机模仿通常只为磁盘驱动器提供的功能。就好像支持真正的异步 I/O 一样，写设备的进程（调用者）会收集大量数据，

并交由写入器写出。数据成功地写出时，写入器（此时写入器是 I/O 从属进程，而不是操作系统）会通知原来的调用者，调用者则会从要写的数据列表中删除这批数据。采用这种方式，可以得到更高的吞吐量，这是因为会由 I/O 从属进程来等待慢速的设备，而原来的调用进程得以脱身，可以做其他重要的工作来收集下一次要写的数据。

I/O 从属进程在 Oracle 中有两个用途。DBWn 和 LGWR 可以利用 I/O 从属进程来模拟异步 I/O，另外 RMAN 写磁带时也可能利用 I/O 从属进程。

有两个参数控制着 I/O 从属进程的使用：

- BACKUP_TAPE_IO_SLAVES：这个参数指定 RMAN 是否使用 I/O 从属进程将数据备份、复制或恢复到磁带上。由于这个参数是围绕着磁带设备设计的，而且磁带设备一次只能由一个进程访问，所以这个参数是一个布尔值，而不是所用从属进程的个数（这可能出乎你的意料）。RMAN 会为所用的物理设备启动多个必要的从属进程。BACKUP_TAPE_IO_SLAVES = TRUE 时，则使用一个 I/O 从属进程从磁带设备读写。如果这个参数为 FALSE（默认值），就不会使用 I/O 从属进程完成备份。相反，完成备份的专用服务器进程会直接访问磁带设备。

- DBWR_IO_SLAVES：这个参数指定了 DBW0 进程所用 I/O 从属进程的个数。DBW0 进程及其从属进程总是将缓冲区缓存中的脏块写至磁盘。这个值默认为 0，表示不使用 I/O 从属进程。注意，如果将这个参数设置为一个非 0 的值，LGWR 和 ARCH 也会使用其自己的 I/O 从属进程，LGWR 和 ARCH 最多允许 4 个 I/O 从属进程。

DBWR I/O 从属进程的名字是 I1nn，LGWR I/O 从属进程的名字是 I2nn，这里 nn 是一个数。

5.3.2 并行查询从属进程

Oracle 7.1.6 引入了并行查询功能。这个功能是指：对于 SELECT、CREATE TABLE、CREATE INDEX、UPDATE 等 SQL 语句，创建一个执行计划，其中包含可以同时完成的多个（子）执行计划。将每个执行计划的输出合并在一起构成一个更大的结果。其目标是仅用少量的时间来完成操作，这只是串行完成同一操作所需时间的一小部分。例如，假设有一个相当大的表，分布在 10 个不同的文件上。你配置有 16 个 CPU，并且需要在这个表上执行一个即席查询。另一种方法是：可以将这个查询计划分解为 32 个小部分，并充分地利用机器；而不是只使用一个进程串行地读取和处理所有数据。相比之下，前一种做法要好得多。

使用并行查询时，会看到名为 Pnnn 的进程，这些就是并行查询从属进程。处理一条并行语句时，服务器进程则称为并行查询协调器（parallel query coordinator）。操作系统上服务器进程的名字并不会改变，但是阅读有关并行查询的文档时，如果提到了协调器进程，你应该知道这就是原来的服务器进程。

5.4 小结

我们已经介绍了 Oracle 使用的文件，涵盖了从低级但很重要的参数文件到数据文件、重做日志文件等等。此外深入分析了 Oracle 使用的内存结构，包括服务器进程中的内存（PGA）和 SGA，还了解了不同的服务器配置（如共享服务器模式和专用服务器模式的连接）对于系统如何使用内存有着怎样显著的影响。最后，我们介绍了进程（或线程，这取决于操作系统），Oracle 正是通过这些进程来完成功能。下面可以具体看看 Oracle 另外一些特

性的实现，如锁定、并发控制和事务。

第 6 章 锁

开发多用户、数据库驱动的应用时，最大的难点之一是：一方面要力争取得最大限度的并发访问，与此同时还要确保每个用户能以一致的方式读取和修改数据。为此就有了锁定（locking）机制，这也是所有数据库都具有的一个关键特性，Oracle 在这方面更是技高一筹。不过，Oracle 的这些特性的实现是 Oracle 所特有的，就像 SQL Server 的实现只是 SQL Server 特有的一样，应用执行数据处理时，要正确地使用这些机制，而这一点要由你（应用的开发人员）来保证。如果你做不到，你的应用可能会表现得出人意料，而且不可避免地会危及数据的完整性（见第 1 章的说明）。

在这一章中，我们将详细介绍 Oracle 如何对数据（例如，表中的行）和共享数据结构（如 SGA 中的内存结构）锁定。这里会分析 Oracle 以怎样的粒度锁定数据，并指出这对你来说意味着什么。在适当的时候，我会把 Oracle 的 锁定机制与其他流行的锁实现（即其他数据库中的锁定机制）进行对照比较，主要是为了消除关于行级锁的一个“神话”：人们认为行级锁总会增加开销；而实际 上，在不同的实现中情况有所不同，只有当实现本身会增加开销时，行级锁才会增加开销。在下一章中，我们还会继续讨论这个内容，进一步研究 Oracle 的多版本技术，并说明锁定策略与多版本技术有什么关系。

6.1 什么是锁？

锁（lock）机制用于管理对共享资源的并发访问。注意，我说的是“共享资源”而不是“数据库行”。Oracle 会在行级对表数据锁定，这固然不错，不过 Oracle 也 会在其他多个级别上使用锁，从而对多种不同的资源提供并发访问。例如，执行一个存储过程时，过程本身会以某种模式锁定，以允许其他用户执行这个过程，但是 不允许另外的用户以任何方式修改这个过程。数据库中使用锁是为了支持对共享资源进行并发访问，与此同时还能提供数据完整性和一致性。

在单用户数据库中，并不需要锁。根据定义，只有一个用户修改信息。不过，如果有多个用户访问和修改数据或数据结构，就要有一种机制来防止对同一份信息的并发修改，这一点至关重要。这正是锁定所要做的全部工作。

需要了解的重要一点是：有多少种数据库，其中就可能有多少种实现锁定的方法。你可能对某个特定的关系数据库管理系统（relational database management system，RDBMS）的锁定模型有一定的经验，但凭此并不意味着你通晓锁定的一切。例如，在我“投身”Oracle

之前，曾经使用过许多其他的数据库，如 Sybase、Microsoft SQL Server 和 Informix。这 3 个数据库都为并发控制提供了锁定机制，但是每个数据库中实现锁定的方式都大相径庭。为了说明这一点，下面简要地概括一下我的“行进路线”，告诉你我是怎样从 SQL Server 开发人员发展为 Informix 用户，最后又怎样成为 Oracle 开发人员。那是好多年前的事情了，可能有些 SQL Server 支持者会说：“但是我们现在也有行级锁了！”没错， SQL Server 现在确实可以使用行级锁，但是其实现方式与 Oracle 中的实现方式完全不同。它们就像是苹果和桔子，是截然不同的两个物体，这正是关键所在。

作为 SQL Server 程序员，我很少考虑多个用户并发地向表中插入数据的可能性。在 SQL Server 数据库中，这种情况极少发生。那时，SQL Server 只提供页级锁，对于非聚簇表，由于所有数据都会插入到表的最后一页，所以两个用户并发插入的情况根本不可能发生。

注意 从某种程度上讲，SQL Server 聚簇表（有一个聚簇索引的表）与 Oracle 聚簇有点相似，但二者存在很大的差别。SQL Server 以前只支持页（块）级锁，如果所插入的每一行都会插入到表的“末尾”，那么这个数据库中绝对不会有并发插入和并发事务。利用 SQL Server 中的聚簇索引，就能按聚簇键的顺序在整个表中插入行（而不是只在表的末尾插入），这就能改善 SQL Server 数据库的并发性。

并发更新也存在同样的问题（因为 UPDATE 实际上就是 DELETE 再加上一个 INSERT）。可能正是由于这个原因，默认情况下，SQL Server 每执行一条语句后就会立即提交或回滚。这样做的目的是为了得到更大的并发性，但是会破坏事务完整性。

因此，大多数情况下，如果采用页级锁，多个用户就不能同时修改同一个表。另外，如果正在修改一个表，这会有效地阻塞对这个表的多个查询。如果我想查询一个表，而且所需的页被一个更新锁住了，那我就必须等待（等待，再等待……）。这种锁定机制太糟糕了，要想支持耗时超过 1 秒 的事务，结果可能是致命的；倘若真的这样做了，整个数据库看上去可能就像是“冻住”了一样。我从这里学到了很多坏习惯。我认识到：事务很“不好”，应该尽 快地提交，而且永远不要持有数据的锁。并发性要以一致性为代价。要么保证正确，要么保证速度，在我看来，鱼和熊掌不可兼得。

等我转而使用 Informix 之后，情况好了一些，但也不是太好。只要创建表时记得启用行级锁，就能允许两个人同时向这个表中插入数据。遗憾的是，这种并发性的代价很高。Informix 实 现中的行级锁开销很大，不论从时间上讲还是从内存上讲都是如此。它要花时间获得和“不要”（或释放）这些行级锁，而且每个锁都要占用实际内存。另外，在启 动数据库之前，必须计算系统可用的锁的总数。如果超过这个数，那你可就要倒霉了。由于这些原因，大多数表都采用页级锁创建，而且与 SQL Server 一样，Informix 中的行级锁和页级锁都会阻塞查询。所以，我再一次发现需要尽快地提交。在 SQL Server 中学到的坏习惯在此得到了巩固，而且，我还学会了一条：要把锁当成一种很稀有的资源，一种可望而难求的事物。我了解到，应该手动地将行级锁升级为表级锁，从而尽量避免需要太多的锁而导致系统崩溃，我就曾经因此多次使系统崩溃。

等开始使用 Oracle 时，我没有费心去读手册，看看这个特定的数据库中锁定是怎么工作的。毕竟，我用数据库已经不是一年半载了，而且也算得上是这个领域的专家（除了 Sybase、SQL Server 和 Informix，我还用过 Ingres、DB2、Gupta SQLBase 和许多其他的数据库）。我落入了过于自信的陷阱，自以为知道事情应该怎么做，所以想当然地认为事情肯定就会这样做。这一次我可大错特错了。

直到一次基准测试时，我才认识到犯了多大的错误。在这些数据库的早期阶段（大约

1992/1993 年），开发商常常对确实很大的数据库产品进行“基准测试”，想看看哪一个数据库能最快、最容易地完成工作，而且能提供最丰富的特性。

这个基准测试在 Informix、Sybase、SQL Server 和 Oracle 之间进行。最先测试的是 Oracle。他们的技术人员来到现场，读过基准测试规范后，他们开始进行设置。我首先注意到的是，Oracle 技术人员只使用一个数据库表来记录他们的计时信息，尽管我们要建立数十条连接来执行测试，而且每条连接都需要频繁地向这个日志表中插入和更新数据。不仅如此，他们还打算在基准测试期间读这个日志表！出于好心，我把一位 Oracle 技 术人员叫到一边，问他这样做是不是疯了，为什么还要故意往系统里引入竞争呢？基准测试进程难道不是对这个表串行地执行操作吗？别人正在对表做大量的修改， 而此时他们还要读这个表，基准测试不会因此阻塞吗？为什么他们想引入所有这些额外的锁，要知道这些锁都需要他们来管理呀！我有一大堆“为什么你会那么 想？”之类的问题。那时，我认为 Oracle 的技术人员有点傻。也就是说，直到我摆脱 SQL Server 或 Informix 的阴影，显示了让两个人同时插入一个表会有什么结果时；或者有人试图查询一个表，而其他人正在向这个表中插入行，此时会有什么结果（查询将每秒返回 0 行），我的观念才有所转变。Oracle 的做法与几乎所有其他数据库的做法有显著的差别，简直是天壤之别。

不用说，无论是 Informix 还是 SQL Server 技术人员，都对这种数据库日志表方法不太热心。他们更倾向于把计时信息记录到操作系统上的平面文件中。Oracle 人员对于如何胜出 SQL Server 和 Informix 很有自己的一套：他们只是问测试人员：“如果数据已经锁定，你当前的数据库每秒返回多少行？”并以此为出发点“展开攻势”。

从 这个故事得到的教训是两方面的。首先，所有数据库本质上都不同。其次，为一个新的数据库平台设计应用时，对于数据库如何工作不能做任何假设。学习每一个新 数据库时，应该假设自己从未使用过数据库。在一个数据库中能做的事情在另一个数据库中可能没有必要做，或者根本不能做。

在 Oracle 中，你会了解到：

- 事务是每个数据库的核心，它们是“好东西”。
- 应该延迟到适当的时刻才提交。不要太快提交，以避免对系统带来压力。这是因为，如果事务很长或很大，一般不会对系统有压力。相应的原则是：在必要时才提交，但是此前不要提交。事务的大小只应该根据业务逻辑来定。
- 只要需要，就应该尽可能长时间地保持对数据所加的锁。这些锁是你能利用的工具，而不是让你退避三舍的东西。锁不是稀有资源。恰恰相反，只要需要，你就应该长期地保持数据上的锁。锁可能并不稀少，而且它们可以防止其他会话修改信息。
- 在 Oracle 中，行级锁没有相关的开销，根本没有。不论你是有 1 个行锁，还是 1 000 000 个行锁，专用于锁定这个信息的“资源”数都是一样的。当然，与修改 1 行相比，修改 1 000 000 行要做的工作肯定多得多，但是对 1 000 000 行锁定所需的资源数与对 1 行锁定所需的资源数完全相同，这是一个固定的常量。
- 不要以为锁升级“对系统更好”（例如，使用表锁而不是行锁）。在 Oracle 中，锁升级（lock escalate）对系统没有任何好处，不会节省任何资源。也许有时会使用表锁，如批处理中，此时你很清楚会更新整个表，而且不希望其他会话锁定表中的行。但是使用表锁绝对不是为了避免分配行锁，想以此来方便系统。

- 可以同时得到并发性和一致性。每次你都能快速而准确地得到数据。数据读取器不会被数据写入器阻塞。数据写入器也不会被数据读取器阻塞。这是 Oracle 与大多数其他关系数据库之间的根本区别之一。

接下来在这一章和下一章的介绍中，我还会强调这几点。

6.2 锁定问题

讨论 Oracle 使用的各种类型的锁之前，先了解一些锁定问题会很有好处，其中很多问题都是因为应用设计不当，没有正确地使用（或者根本没有使用）数据库锁定机制产生的。

6.2.1 丢失更新

丢失更新（lost update）是一个经典的数据库问题。实际上，所有多用户计算机环境都存在这个问题。简单地说，出现下面的情况时（按以下所列的顺序），就会发生丢失更新：

(1) 会话 Session1 中的一个事务获取（查询）一行数据，放入本地内存，并显示给一个最终用户 User1。

(2) 会话 Session2 中的另一个事务也获取这一行，但是将数据显示给另一个最终用户 User2。

(3) User1 使用应用修改了这一行，让应用更新数据库并提交。会话 Session1 的事务现在已经执行。

(4) User2 也修改这一行，让应用更新数据库并提交。会话 Session2 的事务现在已经执行。

这个过程称为“丢失更新”，因为第(3)步所做的所有修改都会丢失。例如，请考虑一个员工更新屏幕，这里允许用户修改地址、工作电话号码等信息。应用本身非常简单：只有一个很小的搜索屏幕要生成一个员工列表，然后可以搜索各位员工的详细信息。这应该只是小菜一碟。所以，编写应用程序时没有考虑锁定，只是简单的 SELECT 和 UPDATE 命令。

然后最终用户（User1）转向详细信息屏幕，在屏幕上修改一个地址，单击 Save（保存）按钮，得到提示信息称更新成功。还不错，但是等到 User1 第二天要发出一个税表时，再来检查记录，会发现所列的还是原先的地址。到底出了什么问题？很遗憾，发生这种情况太容易了。在这种情况下，User1 查询记录后，紧接着另一位最终用户（User2）也查询了同一条记录；也就是说，在 User1 读取数据之后，但在她修改数据之前，User2 也读取了这个数据。然后，在 User2 查询数据之后，User1 执行了更新，接到成功信息，甚至还可能再次查询看看是否已经修改。不过，接下来 User2 更新了工作电话号码字段，并单击 Save（保存）按钮，完全不知道他已经用旧数据重写（覆盖）了 User1 对地址字段的修改！之所以会造成这种情况，这是因为应用开发人员编写的程序是这样的：更新一个特定的字段时，该记录的所有字段都会“刷新”（只是因为更新所有列更容易，这样就不用先得出哪些列已经修改，并且只更新那些修改过的列）。

可以注意到，要想发生这种情况，User1 和 User2 甚至不用同时处理记录。他们只要在大致同一时间处理这个记录就会造成丢失更新。

我发现，如果 GUI 程序员在数据库方面的培训很少（或者没有），编写数据库应用程序时就时常会冒出这个数据库问题。这些程序员了解了如何使用 SELECT、INSERT、UPDATE

和 DELETE 等语句后，就着手开始编写应用程序。如果开发出来的应用程序有上述表现，就会让用户完全失去对它的信心，特别是这种现象只是随机地、零星地出现，而且在受控环境中完全不可再生（这就导致开发人员误以为是用户的错误）。

许多工具可以保护你避免这种情况，如 Oracle Forms 和 HTML DB，这些工具能确保：从查询记录的那个时刻开始，这个记录没有改变，而且对它执行任何修改时都会将其锁定，但是其他程序做不到这一点（如手写的 Visual Basic 或 Java 程序）。为了保护你不丢失更新，这些工具在后台做了哪些工作呢？或者说开发人员必须自己做哪些工作呢？实际上就是要使用某种锁定策略，共有两种锁定策略：悲观锁定或乐观锁定。

6.2.2 悲观锁定

用户在屏幕上修改值之前，这个锁定方法就要起作用。例如，用户一旦有意对他选择的某个特定行（屏幕上可见）执行更新，如单击屏幕上的一个按钮，就会放上一个行锁。

悲观锁定（pessimistic locking）仅用于有状态（stateful）或有连接（connected）环境，也就是说，你的应用与数据库有一条连续的连接，而且至少在事务生存期中只有你一个人使用这条连接。这是 20 世纪 90 年代中期客户/服务器应用中的一种流行做法。每个应用都得到数据库的一条直接连接，这条连接只能由该应用实例使用。这种采用有状态方式的连接方法已经不太常见了（不过并没有完全消失），特别是随着 20 世纪 90 年代中后期应用服务器的出现，有状态连接更是少见。

假设你在使用一条有状态连接，应用可以查询数据而不做任何锁定：

```
scott@ORA10G> select empno, ename, sal from emp where deptno = 10;

     EMPNO     ENAME     SAL

     ----------     ----------     ----------

     7782         CLARK     2450

     7839         KING         5000

     7934         MILLER     1300
```

最后，用户选择他想更新的一行。在这个例子中，假设用户选择更新 MILLER 行。在这个时间点上（即用户还没有在屏幕上做任何修改，但是行已经从数据库中读出一段时间了），应用会绑定用户选择的值，从而查询数据库，并确保数据尚未修改。在 SQL*Plus 中，为了模拟应用可能执行的绑定调用，可以发出以下命令：

```
scott@ORA10G> variable empno number

scott@ORA10G> variable ename varchar2(20)

scott@ORA10G> variable sal number

scott@ORA10G> exec :empno := 7934; :ename := 'MILLER'; :sal := 1300;
```

```
PL/SQL procedure successfully completed.
```

下面，除了简单地查询值并验证数据尚未修改外，我们要使用 FOR UPDATE NOWAIT 锁定这一行。应用要执行以下查询：

```
scott@ORA10G> select empno, ename, sal
  2 from emp
  3 where empno = :empno
  4         and ename = :ename
  5         and sal = :sal
  6 for update nowait
  7 /
EMPNO    ENAME    SAL
---------    ----------    ----------
7934        MILLER    1300
```

根据屏幕上输入的数据，应用将提供绑定变量的值（在这里就是 7934、MILLER 和 1300），然后重新从数据库查询这一行，这一次会锁定这一行，不允许其他会话更新；因此，这种方法称为悲观锁定（pessimistic locking）。在试图更新之前我们就把行锁住了，因为我们很悲观，对于这一行能不能保持未改变很是怀疑。

所有表都应该有一个主键（前面的 SELECT 最多会获取一个记录，因为它包括主键 EMPNO），而且主键应该是不可变的（不应更新主键），从这句话可以得出三个结论：

- 如果底层数据没有改变，就会再次得到 MILLER 行，而且这一行会被锁定，不允许其他会话更新（但是允许其他会话读）。
- 如果另一个用户正在更新这一行，我们就会得到一个 ORA-00054：resource busy（ORA-00054：资源忙）错误。相应地，必须等待更新这一行的用户执行工作。
- 在选择数据和指定有意更新之间，如果有人已经修改了这一行，我们就会得到 0 行。这说明，屏幕上的数据是过时的。为了避免前面所述的丢失更新情况，应用需要重新查询（requery），并在允许在最终用户修改之前锁定数据。有了悲观锁定，User2 试图更新电话号码字段时，应用现在会识别出地址字段已经修改，所以会重新查询数据。因此，User2 不会用这个字段的旧数据覆盖 User1 的修改。

一旦成功地锁定了这一行，应用就会绑定新值，发出更新命令后，提交所做的修改：

```
scott@ORA10G> update emp
```

```
  2 set ename = :ename, sal = :sal

  3 where empno = :empno;

1 row updated.



scott@ORA10G> commit;

Commit complete.
```

现在可以非常安全地修改这一行。我们不可能覆盖其他人所做的修改，因为已经验证了在最初读出数据之后以及对数据锁定之前数据没有改变。

6.2.3 乐观锁定

第二种方法称为乐观锁定（optimistic locking），即把所有锁定都延迟到即将执行更新之前才做。换句话说，我们会修改屏幕上的信息而不要锁。我们很乐观，认为数据不会被其他用户修改；因此，会等到最后一刻才去看我们的想法对不对。

这种锁定方法在所有环境下都行得通，但是采用这种方法的话，执行更新的用户“失败”的可能性会加大。这说明，这个用户要更新他的数据行时，发现数据已经修改过，所以他必须从头再来。

可以在应用中同时保留旧值和新值，然后在更新数据时使用如下的更新语句，这是乐观锁定的一种流行实现：

```
Update table

      Set column1 = :new_column1, column2 = :new_column2, ....

Where primary_key = :primary_key

      And column1 = :old_column1

      And column2 = :old_column2

...
```

在 此，我们乐观地认为数据没有修改。在这种情况下，如果更新语句更新了一行，那我们很幸运；这说明，在读数据和提交更新之间，数据没有改变。但是如果更新了 零行，我们就会失败；另外一个人已经修改了数据，现在我们必须确定应用中下一步要做什么。是让最终用户查询这一行现在的新值，然后再重新开始事务呢（这可 能会让用户很受打击，因为这一行有可能又被修改了）？还是应该根据业务规则解决更新冲突，试图合并两个更新的值（这需要大量的代码）？

实际上，前面的 UPDATE 能避免丢失更新，但是确实有可能被阻塞，在等待另一个会话执行对这一行的 UPDATE 时，它会挂起。如果所有应用（会话）都使用乐观锁定，那么

使用直接的 UPDATE 一般没什么问题，因为执行更新并提交时，行只会被锁定很短的时间。不过，如果某些应用使用了悲观锁定，它会在一段相对较长的时间内持有行上的锁，你可能就会考虑使用 SELECT FOR UPDATE NOWAIT，以此来验证行是否未被修改，并在即将 UPDATE 之前锁定来避免被另一个会话阻塞。

实现乐观并发控制的方法有很多种。我们已经讨论了这样的一种方法，即应用本身会存储行的所有“前”（before）映像。在后几节中，我们将介绍另外三种方法，分别是：

- 使用一个特殊的列，这个列由一个数据库触发器或应用程序代码维护，可以告诉我们记录的“版本”
- 使用一个校验和或散列值，这是使用原来的数据计算得出的
- 使用新增的 Oracle 10g 特性 ORA_ROWSCN。

1. 使用版本列的乐观锁定

这是一个简单的实现，如果你想保护数据库表不出现丢失更新问题，应对每个要保护的表增加一列。这一列一般是 NUMBER 或 DATE/TIMESTAMP 列，通常通过表上的一个行触发器来维护。每次修改行时，这个触发器要负责递增 NUMBER 列中的值，或者更新 DATE/TIMESTAMP 列。

如果应用要实现乐观并发控制，只需要保存这个附加列的值，而不需要保存其他列的所有“前”映像。应用只需验证请求更新那一刻，数据库中这一列的值与最初读出的值是否匹配。如果两个值相等，就说明这一行未被更新过。

下面使用 SCOTT.DEPT 表的一个副本来看看乐观锁定的实现。我们可以使用以下数据定义语言（Data Definition Language，DDL）来创建这个表：

```
ops$tkyte@ORA10G> create table dept
  2        ( deptno number(2),
  3        dname varchar2(14),
  4        loc varchar2(13),
  5       last_mod timestamp with time zone
  6             default systimestamp
  7            not null,
  8        constraint dept_pk primary key(deptno)
  9        )
 10 /

Table created.
```

然后向这个表 INSERT（插入）DEPT 数据的一个副本：

```
ops$tkyte@ORA10G> insert into dept( deptno, dname, loc )
  2 select deptno, dname, loc
  3 from scott.dept;
4 rows created.



ops$tkyte@ORA10G> commit;
Commit complete.
```

以上代码会重建 DEPT 表，但是将有一个附加的 LAST_MOD 列，这个列使用 TIMESTAMP WITH TIME ZONE 数据类型（Oracle9i 及以上版本中才有这个数据类型）。我们将这个列定义为 NOT NULL，以保证这个列必须填有数据，其默认值是当前的系统时间。

这个 TIMESTAMP 数据类型在 Oracle 中精度最高，通常可以精确到微秒（百万分之一秒）。如果应用要考虑到用户的思考时间，这种 TIMESTAMP 级的精度实在是绰绰有余，而且数据库获取一行后，人看到这一行，然后修改，再向数据库发回更新，一般不太可能在不到 1 秒钟的片刻时间内执行整个过程。两个人在同样短的时间内（不到 1 秒钟）读取和修改同一行的几率实在太小了。

接下来，需要一种方法来维护这个值。我们有两种选择：可以由应用维护这一列，更新记录时将 LAST_MOD 列的值设置为 SYSTIMESTAMP；也可以由触发器/存储过程来维护。如果让应用维护 LAST_MOD，这比基于触发器的方法表现更好，因为触发器会代表 Oracle 对修改增加额外的处理。不过这并不是说：无论什么情况，你都要依赖所有应用在表中经过修改的所有位置上一致地维护 LAST_MOD。所以，如果要由各个应用负责维护这个字段，就需要一致地验证 LAST_MOD 列未被修改，并把 LAST_MOD 列设置为当前的 SYSTIMESTAMP。例如，如果应用查询 DEPTNO=10 这一行：

```
ops$tkyte@ORA10G> variable deptno number
ops$tkyte@ORA10G> variable dname varchar2(14)
ops$tkyte@ORA10G> variable loc varchar2(13)
ops$tkyte@ORA10G> variable last_mod varchar2(50)
ops$tkyte@ORA10G> begin
  2        :deptno := 10;
  3        select dname, loc, last_mod
```

```
  4                 into :dname,:loc,:last_mod
  5          from dept
  6          where deptno = :deptno;
  7 end;
  8 /

PL/SQL procedure successfully completed.
```

目前我们看到的是：

```
ops$tkyte@ORA10G> select :deptno dno, :dname dname, :loc loc, :last_mod lm
  2 from dual;

DNO         DNAME          LOC            LM
---------   ----------     --------       ---------------------------------

10          ACCOUNTING     NEW YORK       25-APR-05 10.54.00.493380 AM
-04:00
```

再使用下面的更新语句来修改信息。最后一行执行了一个非常重要的检查，以确保时间戳没有改变，并使用内置函数 TO_TIMESTAMP_TZ(TZ 是 TimeZone 的缩写，即时区）将以上 select（选择）得到的串转换为适当的数据类型。另外，如果发现行已经更新，以下更新语句中的第 3 行会把 LAST_MOD 列更新为当前时间：

```
ops$tkyte@ORA10G> update dept
  2          set dname = initcap(:dname),
  3          last_mod = systimestamp
  4 where deptno = :deptno
  5          and last_mod = to_timestamp_tz(:last_mod);

1 row updated.
```

可以看到，这里更新了一行，也就是我们关心的那一行。在此按主键（DEPTNO）更新了这一行，并验证从最初读取记录到执行更新这段时间，LAST_MOD 列未被其他会话修改。如果我们想尝试再更新这个记录，仍然使用同样的逻辑，不过没有获取新的 LAST_MOD 值，就会观察到以下情况：

```
ops$tkyte@ORA10G> update dept
```

```
  2          set dname = upper(:dname),

  3          last_mod = systimestamp

  4 where deptno = :deptno

  5          and last_mod = to_timestamp_tz(:last_mod);

0 rows updated.
```

注意到这一次报告称“0 rows updated”（更新了 0 行），因为关于 LAST_MOD 的谓词条件不能满足。尽管 DEPTNO 10 还存在，但是想要执行更新的那个时刻的 LAST_MOD 值与查询行时的时间戳值不再匹配。所以，应用知道，既然未能修改行，就说明数据库中的数据已经（被别人）改变，现在它必须得出下一步要对此做什么。

不 能总是依赖各个应用来维护这个字段，原因是多方面的。例如，这样会增加应用程序代码，而且只要是表中需要修改的地方，都必须重复这些代码，并正确地实现。 在一个大型应用中，这样的地方可能很多。另外，将来开发的每个应用也必须遵循这些规则。应用程序代码中很可能会“遗漏”某一处，未能适当地使用这个字段。 因此，如果应用程序代码本身不负责维护这个 LAST_MOD 字段，我相信应用也不应负责检查这个 LAST_MOD 字段（如果它确实能执行检查，当然也能执行更新!）。所以在这种情况下，我建议把更新逻辑封装到一个存储过程中，而不要让应用直接更新表。如果无法相信应用 能维护这个字段的值，那么也无法相信它能正确地检查这个字段。存储过程可以取以上更新中使用的绑定变量作为输入，执行同样的更新。当检测到更新了 0 行时，存储过程会向客户返回一个异常，让客户知道更新实际上失败了。

还有一种实现是使用一个触发器来维护这个 LAST_MOD 字段，但是对于这么简单的工作，我建议还是避免使用触发器，而让 DML 来负责。触发器会引入大量开销，而且在这种情况下没有必要使用它们。

2. 使用校验和的乐观锁定

这与前面的版本列方法很相似，不过在此要使用基数据本身来计算一个“虚拟的”版本列。为了帮助解释有关校验和或散列函数的目标和概念，以下引用了 Oracle 10g PL/SQL Supplied Packages Guide 中的一段话（尽管现在还没有介绍如何使用 Oracle 提供的任何一个包）：

单向散列函数取一个变长输入串（即数据），并把它转换为一个定长的输出串（通常更小），这个输出称为散列值（hash value）。散列值充当输入数据的一个惟一标识符（就像指纹一样）。可以使用散列值来验证数据是否被修改。

需要注意，单向散列函数只能在一个方向上应用。从输入数据计算散列值很容易，但是要生成能散列为某个特定值的数据却很难。

散列值或校验和并非真正惟一。只能说，通过适当地设计，能使出现冲突的可能性相当小，也就是说，两个随机的串有相同校验和或散列值的可能性极小，足以忽略不计。

与 使用版本列的做法一样，我们可以采用同样的方法使用这些散列值或校验和，只需把从数据库读出数据时得到的散列或校验和值与修改数据前得到的散列或校验和值 进行比

较。在我们读出数据之后，但是在修改数据之前，如果有人在这段时间内修改了这一行的值，散列值或校验和值往往会大不相同。

有很多方法来计算散列或校验和。这里列出其中的 3 种方法，分别在以下 3 个小节中介绍。所有这些方法都利用了 Oracle 提供的数据库包：

- OWA_OPT_LOCK.CHECKSUM：这个方法在 Oracle8i 8.1.5 及以上版本中提供。给定一个串，其中一个函数会返回一个 16 位的校验和。给定 ROWID 时，另一个函数会计算该行的 16 位校验和，而且同时将这一行锁定。出现冲突的可能性是 65 536 分之一（65 536 个串中有一个冲突，这是假警报的最大几率）。
- DBMS_OBFUSCATION_TOOLKIT.MD5：这个方法在 Oracle8i 8.1.7 及以上版本中提供。它会计算一个 128 位的消息摘要。冲突的可能性是 3.4028E+38 分之一（非常小）。
- DBMS_CRYPTO.HASH：这个方法在 Oracle 10g Release 1 及以上版本中提供。它能计算一个 SHA-1（安全散列算法 1，Secure Hash Algorithm 1）或 MD4/MD5 消息摘要。建议你使用 SHA-1 算法。

注意 很多编程语言中都提供了一些散列和校验和函数，所以还可以使用数据库之外的散列和校验和函数。

下面的例子显示了如何使用 Oracle 10g 中的 DBMS_CRYPTO 内置包来计算这些散列/校验和。这个技术也适用于以上所列的另外两个包；逻辑上差别不大，但是调用的 API 可能不同。

下面在某个应用中查询并显示部门 10 的信息。查询信息之后，紧接着我们使用 DBMS_CRYPTO 包计算散列。这是应用中要保留的“版本”信息：

```
ops$tkyte@ORA10G> begin
  2      for x in ( select deptno, dname, loc
  3                 from dept
  4                 where deptno = 10 )
  5      loop
  6          dbms_output.put_line( 'Dname: ' || x.dname );
  7          dbms_output.put_line( 'Loc: ' || x.loc );
  8          dbms_output.put_line( 'Hash: ' ||
  9                  dbms_crypto.hash
 10                  ( utl_raw.cast_to_raw(x.deptno||'/'||x.dname||'/'||x.loc),
```

```
  11                      dbms_crypto.hash_sh1 ) );
  12        end loop;
  13 end;
  14 /
Dname: ACCOUNTING
Loc: NEW YORK
Hash: C44F7052661CE945D385D5C3F911E70FA99407A6
PL/SQL procedure successfully completed.
```

可以看到，散列值就是一个很大的 16 进制位串。DBMS_CRYPTO 的返回值是一个 RAW 变量，显示时，它会隐式地转换为 HEX。这个值会在更新前使用。为了执行更新，需要在数据库中获取这一行，并按其现在的样子锁定，然后计算所获取的行的散列值，将这个新散列值与从数据库读出数据时计算的散列值进行比较。上述逻辑表示如下（当然，在实际中，可能使用绑定变量而不是散列值直接量）：

```
ops$tkyte@ORA10G> begin
  2       for x in ( select deptno, dname, loc
  3                  from dept
  4                  where deptno = 10
  5                  for update nowait )
  6       loop
  7                  if ( hextoraw( 'C44F7052661CE945D385D5C3F911E70FA99407A6' )
<>
  8                      dbms_crypto.hash
  9                       ( utl_raw.cast_to_raw(x.deptno||'/'||x.dname||'/'||x.loc),
  10                     dbms_crypto.hash_sh1 ) )
  11                 then
  12                     raise_application_error(-20001, 'Row was modified' );
```

```
13              end if;
14      end loop;
15      update dept
16              set dname = lower(dname)
17      where deptno = 10;
18      commit;
19 end;
20 /
PL/SQL procedure successfully completed.
```

更新后，重新查询数据，并再次计算散列值，此时可以看到散列值大不相同。如果有人抢在我们前面先修改了这一行，我们的散列值比较就不会成功：

```
ops$tkyte@ORA10G> begin
2       for x in ( select deptno, dname, loc
3               from dept
4               where deptno = 10 )
5       loop
6               dbms_output.put_line( 'Dname: ' || x.dname );
7               dbms_output.put_line( 'Loc: ' || x.loc );
8               dbms_output.put_line( 'Hash: ' ||
9                       dbms_crypto.hash
10                      ( utl_raw.cast_to_raw(x.deptno||'/'||x.dname||'/'||x.loc),
11                      dbms_crypto.hash_sh1 ) );
12      end loop;
13 end;
```

```
  14 /

Dname: accounting

Loc: NEW YORK

Hash: F3DE485922D44DF598C2CEBC34C27DD2216FB90F

PL/SQL procedure successfully completed.
```

这个例子显示了如何利用散列或校验和来实现乐观锁定。要记住，计算散列或校验和是一个 CPU 密集型操作（相当占用 CPU），其计算代价很昂贵。如果系统上 CPU 是稀有资源，在这种系统上就必须充分考虑到这一点。不过，如果从“网络友好性”角度看，这种方法会比较好，因为只需在网络上传输相当小的散列值，而不是行的完整的前映像和后映像（以便逐列地进行比较），所以消耗的资源会少得多。下面最后一个例子会使用一个新的 Oracle 10g 函数 ORA_ROWSCN，它不仅很小（类似于散列），而且计算时不是 CPU 密集的（不会过多占用 CPU）。

3. 使用 ORA_ROWSCN 的乐观锁定

从 Oracle 10g Release 1 开始，你还可以使用内置的 ORA_ROWSCN 函数。它的工作与前面所述的版本列技术很相似，但是可以由 Oracle 自动执行，而不需要在表中增加额外的列，也不需要额外的更新/维护代码来更新这个值。

ORA_ROWSCN 建立在内部 Oracle 系统时钟（SCN）基础上。在 Oracle 中，每次提交时，SCN 都会推进（其他情况也可能导致 SCN 推进，要注意，SCN 只会推进，绝对不会后退）。这个概念与前面在获取数据时得到 ORA_ROWSCN 的方法是一样的，更新数据时要验证 SCN 未修改过。之所以我会强调这一点（而不是草草带过），原因是除非你创建表时支持在行级维护 ORA_ROWSCN，否则 Oracle 会在块级维护。也就是说，默认情况下，一个块上的多行会共享相同的 ORA_ROWSCN 值。如果更新一个块上的某一行，而且这个块上还有另外 50 行，那么这些行的 ORA_ROWSCN 也会推进。这往往会导致许多假警报，你认为某一行已经修改，但实际上它并没有改动。因此，需要注意这一点，并了解如何改变这种行为。

我们想查看这种行为，然后进行修改，为此还要使用前面的小表 DEPT：

```
ops$tkyte@ORA10G> create table dept

  2       (deptno, dname, loc, data,

  3       constraint dept_pk primary key(deptno)

  4       )

  5       as

  6       select deptno, dname, loc, rpad('*',3500,'*')
```

```
  7          from scott.dept;

Table created.
```

现在可以观察到每一行分别在哪个块上（在这种情况下，可以假设它们都在同一个文件中，所以如果块号相同，就说明它们在同一个块上）。我使用的块大小是 8 KB，一行的宽度大约 3 550 字节，所以我预料这个例子中每块上有两行：

```
ops$tkyte@ORA10G> select deptno, dname,
  2                 dbms_rowid.rowid_block_number(rowid) blockno,
  3                 ora_rowscn
  4          from dept;

DEPTNO     DNAME              BLOCKNO     ORA_ROWSCN
---------- -------------- ---------- ----------
10         ACCOUNTING                  20972       34676029
20         RESEARCH       20972          34676029
30         SALES          20973          34676029
40         OPERATIONS 20973          34676029
```

不错，我们观察的结果也是这样，每块有两行。所以，下面来更新块 20972 上 DEPTNO = 10 的那一行：

```
ops$tkyte@ORA10G> update dept
  2 set dname = lower(dname)
  3 where deptno = 10;

1 row updated.



ops$tkyte@ORA10G> commit;

Commit complete.
```

接下来观察到，ORA_ROWSCN 的结果在块级维护。我们只修改了一行，也只提交了这一行的修改，但是块 20972 上两行的 ORA_ROWSCN 值都推进了：

```
ops$tkyte@ORA10G> select deptno, dname,
  2 dbms_rowid.rowid_block_number(rowid) blockno,
  3 ora_rowscn
  4 from dept;
DEPTNO     DNAME              BLOCKNO     ORA_ROWSCN
---------- -------------- ---------- ----------
10         accounting     20972       34676046
20         RESEARCH       20972       34676046
30         SALES          20973       34676029
40         OPERATIONS     20973       34676029
```

如果有人读取 DEPTNO=20 这一行，看起来这一行已经修改了，但实际上并非如此。块 20973 上的行是“安全”的，我们没有修改这些行，所以它们没有推进。不过，如果更新其中任何一行，两行都将推进。所以现在的问题是：如何修改这种默认行为。遗憾的是，我们必须启用 ROWDEPENDENCIES 再重新创建这个段。

Oracle9i 为数据库增加了行依赖性跟踪，可以支持推进复制，以便更好地并行传播修改。在 Oracle 10g 之前，这个特性只能在复制环境中使用；但是从 Oracle 10g 开始，还可以利用这个特性用 ORA_ROWSCN 来实现一种有效的乐观锁定技术。它会为每行增加 6 字节的开销（所以与自己增加版本列的方法（即 DIY 版本列方法）相比，并不会节省空间），而实际上，也正是因为这个原因，所以需要重新创建表，而不只是简单地 ALTER TABLE：必须修改物理块结构来适应这个特性。

下面重新建立我们的表，启用 ROWDEPENDENCIES。可以使用 DBMS_REDEFINITION 中（Oracle 提供的另一个包）的在线重建功能来执行，但是对于一个这么小的任务，我们还是从头开始更好一些：

```
ops$tkyte@ORA10G> drop table dept;
Table dropped.

ops$tkyte@ORA10G> create table dept
  2                 (deptno, dname, loc, data,
  3                 constraint dept_pk primary key(deptno)
```

```
  4          )
  5        ROWDEPENDENCIES
  6          as
  7                  select deptno, dname, loc, rpad('*',3500,'*')
  8                  from scott.dept;
Table created.


ops$tkyte@ORA10G> select deptno, dname,
  2 dbms_rowid.rowid_block_number(rowid) blockno,
  3 ora_rowscn
  4 from dept;
DEPTNO     DNAME              BLOCKNO     ORA_ROWSCN
---------       --------------          ----------          ----------
10              ACCOUNTING                    21020       34676364
20              RESEARCH       21020          34676364
30              SALES              21021          34676364
40              OPERATIONS 21021          34676364
```

又回到前面：两个块上有 4 行，它们都有相同的 ORA_ROWSCN 值。现在，更新 DEPTNO=10 的那一行时：

```
ops$tkyte@ORA10G> update dept
  2 set dname = lower(dname)
  3 where deptno = 10;
1 row updated.
```

```
ops$tkyte@ORA10G> commit;

Commit complete.
```

查询 DEPT 表时应该能观察到以下结果：

```
ops$tkyte@ORA10G> select deptno, dname,
  2         dbms_rowid.rowid_block_number(rowid) blockno,
  3         ora_rowscn
  4 from dept;


DEPTNO     DNAME              BLOCKNO     ORA_ROWSCN
----------       --------------          ----------          ----------
10               accounting         21020           34676381
20               RESEARCH       21020           34676364
30               SALES              21021           34676364
40               OPERATIONS 21021           34676364
```

此时，只有 DEPTNO = 10 这一行的 ORA_ROWSCN 改变，这正是我们所希望的。现在可以依靠 ORA_ROWSCN 来为我们检测行级修改了。

将 SCN 转换为墙上时钟时间

使用透明的 ORA_ROWSCN 列还有一个好处：可以把 SCN 转换为近似的墙上时钟时间（有+/－3 秒的偏差），从而发现行最后一次修改发生在什么时间。例如，可以执行以下查询：

```
ops$tkyte@ORA10G> select deptno, ora_rowscn, scn_to_timestamp(ora_rowscn) ts
  2 from dept;


DEPTNO ORA_ROWSCN TS
----------       ----------              ------------------------------
```

```
10          34676381        25-APR-05 02.37.04.000000000 PM

20          34676364        25-APR-05 02.34.42.000000000 PM

30          34676364        25-APR-05 02.34.42.000000000 PM

40          34676364        25-APR-05 02.34.42.000000000 PM
```

在此可以看到，在表的最初创建和更新 DEPTNO = 10 行之间，我等了大约 3 分钟。不过，从 SCN 到墙上时钟时间的这种转换有一些限制：数据库的正常运行时间只有 5 天左右。例如，如果查看一个“旧”表，查找其中最旧的 ORA_ROWSCN（注意，在此我作为 SCOTT 登录；没有使用前面的新表）：

```
scott@ORA10G> select min(ora_rowscn) from dept;

MIN(ORA_ROWSCN)

---------------

364937
```

如果我试图把这个 SCN 转换为一个时间戳，可能看到以下结果（取决于 DEPT 表有多旧 ）：

```
scott@ORA10G> select scn_to_timestamp(min(ora_rowscn)) from dept;

select scn_to_timestamp(min(ora_rowscn)) from dept

                *

ERROR at line 1:

ORA-08181: specified number is not a valid system change number

ORA-06512: at "SYS.SCN_TO_TIMESTAMP", line 1

ORA-06512: at line 1
```

所以从长远看不能依赖这种转换。

6.2.4 乐观锁定还是悲观锁定？

那么哪种方法最好呢？根据我的经验，悲观锁定在 Oracle 中工作得非常好（但是在其他数据库中可能不是这样），而且与乐观锁定相比，悲观锁定有很多优点。不过，它需要与数据库有一条有状态的连接，如客户/服务器连接，因为无法跨连接持有锁。正是因为这一点，在当前的许多情况下，悲观锁定不太现实。过去，客户/服务器应用可能只有数十个或

数百个用户，对于这些应用，悲观锁定是我的不二选择。不过，如今对大多数应用来说，我都建议采用乐观并发控制。要在整个事务期间保持连接，这个代价太大了，一般无法承受。

在这些可用的方法中，我使用哪一种呢？我喜欢使用版本列方法，并增加一个时间戳列（而不只是一个 NUMBER）。从长远看，这样能为我提供一个额外的信息：“这一行最后一次更新发生在什么时间？”所以意义更大。而且与散列或校验和方法相比，计算的代价不那么昂贵，在处理 LONG、LONG RAW、CLOB、BLOB 和其他非常大的列时，散列或校验和方法可能会遇到一些问题，而版本列方法则没有这些问题。

如果必须向一个表增加乐观并发控制，而此时还在利用悲观锁定机制使用这个表（例如，客户/服务器应用都在访问这个表，而且还在通过 Web 访问），我则倾向于选择 ORA_ROWSCN 方法。这是因为，在现有的遗留应用中，可能不希望出现一个新列，或者即使我们另外增加一步把这个额外的列隐藏起来，为了维护这个列，可能需要一个必要的触发器，而这个触发器的开销非常大，这是我们无法承受的。ORA_ROWSCN 技术没有干扰性，而且在这个方面是轻量级的（当然，这是指我们执行表的重建之后）。

散列/校验和方法在数据库独立性方面很不错，特别是如果我们在数据库之外计算散列或校验和，则更是如此。不过，如果在中间层而不是在数据库中执行计算，从 CPU 使用和网络传输方面来看，就会带来更大的资源使用开销。

6.2.5 阻塞

如果一个会话持有某个资源的锁，而另一个会话在请求这个资源，就会出现阻塞（blocking）。这样一来，请求的会话会被阻塞，它会“挂起”，直至持有锁的会话放弃锁定的资源。几乎在所有情况下，阻塞都是可以避免的。实际上，如果你真的发现会话在一个交互式应用中被阻塞，就说明很有可能同时存在着另一个 bug，即丢失更新，只不过你可能没有意识到这一点。也就是说，你的应用逻辑有问题，这才是阻塞的根源。

数据库中有 5 条常见的 DML 语句可能会阻塞，具体是：INSERT、UPDATE、DELETE、MERGE 和 SELECT FOR UPDATE。对于一个阻塞的 SELECT FOR UPDATE，解决方案很简单：只需增加 NOWAIT 子句，它就不会阻塞了。这样一来，你的应用会向最终用户报告，这一行已经锁定。另外 4 条 DML 语句才有意思。我们会分别分析这些 DML 语句，看看它们为什么不应阻塞，如果真的阻塞了又该如何修正。

1. 阻塞的 INSERT

INSERT 阻塞的情况不多见。最常见的情况是，你有一个带主键的表，或者表上有惟一的约束，但有两个会话试图用同样的值插入一行。如果是这样，其中一个会话就会阻塞，直到另一个会话提交或者回滚为止：如果另一个会话提交，那么阻塞的会话会收到一个错误，指出存在一个重复值；倘若另一个会话回滚，在这种情况下，阻塞的会话则会成功。还有一种情况，可能多个表通过引用完整性约束相互链接。对子表的插入可能会阻塞，因为它所依赖的父表正在创建或删除。

如果应用允许最终用户生成主键/惟一列值，往往就会发生 INSERT 阻塞。为避免这种情况，最容易的做法是使用一个序列来生成主键/惟一列值。序列（sequence）设计为一种高度并发的方法，用在多用户环境中生成惟一键。如果无法使用序列，那你可以使用以下技术，也就是使用手工锁来避免这个问题，这里的手工锁通过内置的 DBMS_LOCK 包来实现。

注意 会话可能因为主键或惟一约束而遭遇插入阻塞，下面的例子展示了如何避免这种情

况。需要强调一点，这里所示的“修正方法”只能算是一个短期的解决方案，因为这 个应用的体系结构本身就存在问题。这种方法显然会增加开销，而且不能轻量级地实现。如果应用设计得好，就不会遇到这个问题。只能把这当作最后一道防线，千 万不要因为“以防万一”而对应用中的每个表都采用这种技术。

对于插入，不会选择现有的行，也不会对现有的行锁定[1]。没有办法避免其他人插入值相同的行，如果别人真的插入了具有相同值的行，这会阻塞我们的会话，而导致我们无休止地等待。此时，DBMS_LOCK就能派上用场了。为了介绍这个技术，下面创建一个带主键的表，还有一个触发器，它会防止两个（或更多）会话同时插入相同的值。这个触发器使用DBMS_UTILITY.GET_ HASH_VALUE来计算主键的散列值，得到一个 0～1 073 741 823 之间的数（这也是Oracle允许我们使用的锁ID号的范围）。在这个例子中，我选择了一个大小为 1 024 的散列表，这说明我们会把主键散列到 1 024 个不同的锁ID。然后使用DBMS_LOCK.REQUEST根据这个ID分配一个排他锁（也称独占锁，exclusive lock）。一次只有一个会话能做这个工作，所以，如果有人想用相同的主键值向表中插入一条记录，这个人的锁请求就会失败（并且会产生resource busy（资源忙）错误）：

注意 为了成功地编译这个触发器，必须直接给你的模式授予 DBMS_LOCK 的执行权限。执行 DBMS_LOCK 的权限不能从角色得来：

```
scott@ORA10G> create table demo ( x int primary key );

Table created.



scott@ORA10G> create or replace trigger demo_bifer
  2          before insert on demo
  3          for each row
  4          declare
  5                  l_lock_id number;
  6                  resource_busy exception;
  7                  pragma exception_init( resource_busy, -54 );
  8          begin
  9                  l_lock_id :=
 10                          dbms_utility.get_hash_value( to_char( :new.x ), 0, 1024 );
 11                 if ( dbms_lock.request
```

```
12                      ( id => l_lock_id,
13                              lockmode => dbms_lock.x_mode,
14                              timeout => 0,
15                              release_on_commit => TRUE ) <> 0 )
16              then
17                      raise resource_busy;
18              end if;
19      end;
20 /

Trigger created.
```

现在，如果在两个单独的会话中执行下面的插入：

```
scott@ORA10G> insert into demo values ( 1 );

1 row created.
```

第一个会话会成功，但是紧接着第二个会话中会得出以下错误：

```
scott@ORA10G> insert into demo values ( 1 );

insert into demo values ( 1 )

                *

ERROR at line 1:

ORA-00054: resource busy and acquire with NOWAIT specified

ORA-06512: at "SCOTT.DEMO_BIFER", line 14

ORA-04088: error during execution of trigger 'SCOTT.DEMO_BIFER'
```

这里的思想是：为表提供的主键值要受触发器的保护，并把它放入一个字符串中。然后可以使用 DBMS_UTILITY.GET_HASH_VALUE 为这个串得出一个“几乎惟一”的散列值。只要使用小于 1 073 741 823 的散列表，就可以使用 DBMS_LOCK 独占地“锁住”这个值。

计算散列之后，取得这个散列值，并使用 DBMS_LOCK 来请求将这个锁 ID 独占地锁住（超时时间为 ZERO，这说明如果已经有人锁住了这个值，它会立即返回）。如果超时或者由于某种原因失败了，将产生 ORA-54 Resource Busy（资源忙）错误。否则什么也不做，

完全可以顺利地插入，我们不会阻塞。

当然，如果表的主键是一个 INTEGER，而你不希望这个主键超过 1 000 000 000，那么可以跳过散列，直接使用这个数作为锁 ID。要适当地设置散列表的大小（在这个例子中，散列表的大小是 1 024），以避免因为不同的串散列为同一个数（这称为散列冲突）而人工地导致资源忙消息。散列表的大小与特定的应用（数据）有关，并发插入的数量也会影响散列表的大小。最后，还要记住，尽管 Oracle 有无限多个行级锁，但是 enqueue 锁（这是一种队列锁）的个数则是有限的。如果在会话中插入大量行，而没有提交，可能就会发现创建了太多的 enqueue 队列锁，而耗尽了系统的队列资源（超出了 ENQUEUE_RESOURCES 系统参数设置的最大值），因为每行都会创建另一个 enqueue 锁。如果确实发生了这种情况，就需要增大 ENQUEUE_RESOURCES 参数的值。还可以向触发器增加一个标志，允许打开或关闭这种检查。例如，如果我准备插入数百条或数千条记录，可能就不希望启用这个检查。

2. 阻塞的 Merge、Update 和 Delete

在一个交互式应用中，可以从数据库查询某个数据，允许最终用户处理这个数据，再把它“放回”到数据库中，此时如果 UPDATE 或 DELETE 阻塞，就说明你的代码中可能存在一个丢失更新问题（如果真是这样，按我的说法，就是你的代码中存在 bug）。你试图 UPDATE（更新）其他人正在更新的行（换句话说，有人已经锁住了这一行）。通过使用 SELECT FOR UPDATE NOWAIT 查询可以避免这个问题，这个查询能做到：

- ☐ 验证自从你查询数据之后数据未被修改（防止丢失更新）。
- ☐ 锁住行（防止 UPDATE 或 DELETE 被阻塞）。

如前所述，不论采用哪一种锁定方法都可以这样做。不论是悲观锁定还是乐观锁定都可以利用 SELECT FOR UPDATE NOWAIT 查询来验证行未被修改。悲观锁定会在用户有意修改数据那一刻使用这条语句。乐观锁定则在即将在数据库中更新数据时使用这条语句。这样不仅能解决应用中的阻塞问题，还可以修正数据完整性问题。

由于 MERGE 只是 INSERT 和 UPDATE（如果在 10g 中采用改进的 MERGE 语法，还可以是 DELETE），所以可以同时使用这两种技术。

6.2.6 死锁

如果你有两个会话，每个会话都持有另一个会话想要的资源，此时就会出现死锁（deadlock）。例如，如果我的数据库中有两个表 A 和 B，每个表中都只有一行，就可以很容易地展示什么是死锁。我要做的只是打开两个会话（例如，两个 SQL*Plus 会话）。在会话 A 中更新表 A，并在会话 B 中更新表 B。现在，如果我想在会话 B 中更新表 A，就会阻塞。会话 A 已经锁定了这一行。这不是死锁；只是阻塞而已。我还没有遇到过死锁，因为会话 A 还有机会提交或回滚，这样会话 B 就能继续。

如果我再回到会话 A，试图更新表 B，这就会导致一个死锁。要在这两个会话中选择一个作为“牺牲品”，让它的语句回滚。例如，会话 B 中对表 A 的更新可能回滚，得到以下错误：

```
update a set x = x+1
```

```
*

ERROR at line 1:

ORA-00060: deadlock detected while waiting for resource
```

想要更新表 B 的会话 A 还阻塞着，Oracle 不会回滚整个事务。只会回滚与死锁有关的某条语句。会话 B 仍然锁定着表 B 中的行，而会话 A 还在耐心地等待这一行可用。收到死锁消息后，会话 B 必须决定将表 B 上未执行的工作提交还是回滚，或者继续走另一条路，以后再提交。一旦这个会话执行提交或回滚，另一个阻塞的会话就会继续，好像什么也没有发生过一样。

Oracle 认为死锁很少见，而且由于如此少见，所以每次出现死锁时它都会在服务器上创建一个跟踪文件。这个跟踪文件的内容如下：

```
*** 2005-04-25 15:53:01.455

*** ACTION NAME:() 2005-04-25 15:53:01.455

*** MODULE NAME:(SQL*Plus) 2005-04-25 15:53:01.455

*** SERVICE NAME:(SYS$USERS) 2005-04-25 15:53:01.455

*** SESSION ID:(145.208) 2005-04-25 15:53:01.455

DEADLOCK DETECTED

Current SQL statement for this session:

update a set x = 1

The following deadlock is not an ORACLE error. It is a

deadlock due to user error in the design of an application

or from issuing incorrect ad-hoc SQL. The following

information may aid in determining the deadlock:...
```

显然，Oracle 认为这些应用死锁是应用自己导致的错误，而且在大多数情况下，Oracle 的这种看法都是正确的。不同于许多其他的 RDBMS，Oracle 中极少出现死锁，甚至可以认为几乎不存在。通常情况下，必须人为地提供条件才会产生死锁。

根据我的经验，导致死锁的头号原因是外键未加索引（第二号原因是表上的位图索引遭到并发更新，这个内容将在第 11 章讨论）。在以下两种情况下，Oracle 在修改父表后会对子表加一个全表锁：

- 如果更新了父表的主键（倘若遵循关系数据库的原则，即主键应当是不可变的，这种情况就很少见），由于外键上没有索引，所以子表会被锁住。

□ 如果删除了父表中的一行，整个子表也会被锁住（由于外键上没有索引）。

在Oracle9i及以上版本中，这些全表锁都是短期的，这意味着它们仅在DML操作期间存在，而不是在整个事务期间都存在。即便如此，这些全表锁还是可能（而且确实会）导致很严重的锁定问题。下面说明第二点[2]，如果用以下命令建立了两个表：

```
ops$tkyte@ORA10G> create table p ( x int primary key );

Table created.



ops$tkyte@ORA10G> create table c ( x references p );

Table created.



ops$tkyte@ORA10G> insert into p values ( 1 );

1 row created.



ops$tkyte@ORA10G> insert into p values ( 2 );

1 row created.

ops$tkyte@ORA10G> commit;

Commit complete.
```

然后执行以下语句：

```
ops$tkyte@ORA10G> insert into c values ( 2 );

1 row created.
```

到目前为止，还没有什么问题。但是如果再到另一个会话中，试图删除第一条父记录：

```
ops$tkyte@ORA10G> delete from p where x = 1;
```

此时就会发现，这个会话立即被阻塞了。它在执行删除之前试图对表C加一个全表锁。现在，别的会话都不能对C中的任何行执行DELETE、INSERT或UPDATE（已经开始的会话可以继续[3]，但是新会话将无法修改C）。

更新主键值也会发生这种阻塞。因为在关系数据库中，更新主键是一个很大的禁忌，所以更新在这方面一般没有什么问题。在我看来，如果开发人员使用能生成 SQL 的工具，而且这些工具会更新每一列，而不论最终用户是否确实修改了那些列，此时更新主键就会成为一个严重的问题。例如，假设我们使用了 Oracle Forms，并为表创建了一个默认布局。默

认情况下，Oracle Forms 会生成一个更新，对我们选择要显示的表中的每一列进行修改。如果在 DEPT 表中建立一个默认布局，包括 3 个字段，只要我们修改了 DEPT 表中的任何列，Oracle Forms 都会执行以下命令：

```
update dept set deptno=:1,dname=:2,loc=:3 where rowid=:4
```

在这种情况下，如果 EMP 表有 DEPT 的一个外键，而且在 EMP 表的 DEPTNO 列上没有任何索引，那么更新 DEPT 时整个 EMP 表都会被锁定。如果你使用了能生成 SQL 的工具，就一定要当心这一点。即便主键值没有改变，执行前面的 SQL 语句后，子表 EMP 也会被锁定。如果使用 Oracle Forms，解决方案是把这个表的 UPDATE CHANGED COLUMNS ONLY 属性设置为 YES。这样一来，Oracle Forms 会生成一条 UPDATE 语句，其中只包含修改过的列（而不包括主键）。

删除父表中的一行可能导致子表被锁住，由此产生的问题更多。我已经说过，如果删除表 P 中的一行，那么在 DML 操作期间，子表 C 就会锁定，这样能避免事务期间对 C 执行其他更新（当然，这有一个前提，即没有人在修改 C；如果确实已经有人在修改 C，删除会等待）。此时就会出现阻塞和死锁问题。通过锁定整个表 C，数据库的并发性就会大幅下降，以至于没有人能够修改 C 中的任何内容。另外，出现死锁的可能性则增加了，因为我的会话现在“拥有”大量数据，直到提交时才会交出。其他会话因为 C 而阻塞的可能性也更大；只要会话试图修改 C 就 会被阻塞。因此，我开始注意到，数据库中大量会话被阻塞，这些会话持有另外一些资源的锁。实际上，如果其中任何阻塞的会话锁住了我的会话需要的资源，就会 出现一个死锁。在这种情况下，造成死锁的原因是：我的会话不允许别人访问超出其所需的更多资源（在这里就是一个表中的所有行）。如果有人抱怨说数据库中存 在死锁，我会让他们运行一个脚本，查看是不是存在未加索引的外键，而且在 99%的情况下都会发现表中确实存在这个问题。只需对外键加索引，死锁（以及大量其他的竞争问题）都会烟消云散。下面的例子展示了如何使用这个脚本来找出表 C 中未加索引的外键：

```
ops$tkyte@ORA10G> column columns format a30 word_wrapped

ops$tkyte@ORA10G> column tablename format a15 word_wrapped

ops$tkyte@ORA10G> column constraint_name format a15 word_wrapped



ops$tkyte@ORA10G> select table_name, constraint_name,

  2                 cname1 || nvl2(cname2,','||cname2,null) ||

  3                 nvl2(cname3,','||cname3,null) || nvl2(cname4,','||cname4,null) ||

  4                  nvl2(cname5,','||cname5,null) || nvl2(cname6,','||cname6,null) ||

  5                  nvl2(cname7,','||cname7,null) || nvl2(cname8,','||cname8,null)

  6                         columns
```

```
7       from ( select b.table_name,
8                    b.constraint_name,
9                    max(decode( position, 1, column_name, null )) cname1,
10                   max(decode( position, 2, column_name, null )) cname2,
11                   max(decode( position, 3, column_name, null )) cname3,
12                   max(decode( position, 4, column_name, null )) cname4,
13                   max(decode( position, 5, column_name, null )) cname5,
14                   max(decode( position, 6, column_name, null )) cname6,
15                   max(decode( position, 7, column_name, null )) cname7,
16                   max(decode( position, 8, column_name, null )) cname8,
17                   count(*) col_cnt
18             from (select substr(table_name,1,30) table_name,
19                          substr(constraint_name,1,30)  constraint_name,
20                          substr(column_name,1,30) column_name,
21                          position
22                   from user_cons_columns ) a,
23                          user_constraints b
24                   where a.constraint_name = b.constraint_name
25                          and b.constraint_type = 'R'
26                   group by b.table_name, b.constraint_name
27                   ) cons
28             where col_cnt > ALL
29                   ( select count(*)
```

```
   30                  from user_ind_columns i
   31                  where i.table_name = cons.table_name
   32                        and i.column_name in (cname1, cname2, cname3, cname4,
   33                              cname5, cname6, cname7, cname8 )
   34                        and i.column_position <= cons.col_cnt
   35                  group by i.index_name
   36        )
   37 /

TABLE_NAME                CONSTRAINT_NAME     COLUMNS
------------------------------  ---------------     ------------------------------
C                         SYS_C009485         X
```

这个脚本将处理外键约束，其中最多可以有 8 列（如果你的外键有更多的列，可能就得重新考虑一下你的设计了）。首先，它在前面的查询中建立一个名为 CONS 的内联视图（inline view）。这个内联视图将约束中适当的列名从行转置到列，其结果是每个约束有一行，最多有 8 列，这些列分别取值为约束中的列名。另外，这个视图中还有一个列 COL_CNT，其中包含外键约束本身的列数。对于这个内联视图中返回的每一行，我们要执行一个关联子查询（correlated subquery），检查当前所处理表上的所有索引。它会统计出索引中与外键约束中的列相匹配的列数，然后按索引名分组。这样，就能生成一组数，每个数都是该表某个索引中匹配列的总计。如果原来的 COL_CNT 大于所有这些数，那么表中就没有支持这个约束的索引。如果 COL_CNT 小于所有这些数，就至少有一个索引支持这个约束。注意，这里使用了 NVL2 函数，我们用这个函数把列名列表“粘到”一个用逗号分隔的列表中。这个函数有 3 个参数：A、B 和 C。如果参数 A 非空，则返回 B；否则返回参数 C。这个查询有一个前提，假设约束的所有者也是表和索引的所有者。如果另一位用户对表加索引，或者表在另一个模式中（这两种情况都很少见），就不能正确地工作。

所以，这个脚本展示出，表 C 在列 X 上有一个外键，但是没有索引。通过对 X 加索引，就可以完全消除这个锁定问题。除了全表锁外，在以下情况下，未加索引的外键也可能带来问题：

- 如果有ON DELETE CASCADE，而且没有对子表加索引：例如，EMP是DEPT的子表，DELETE DEPTNO = 10 应该CASCADE（级联）至EMP[4]。如果EMP中的DEPTNO没有索引，那么删除DEPT表中的每一行时都会对EMP做一个全表扫描。这个全表扫描可能是不必要的，而且如果从父表删除多行，父表中每删除一行就要扫描一次子表。
- 从父表查询子表：再次考虑 EMP/DEPT 例子。利用 DEPTNO 查询 EMP 表是相

当常见的。如果频繁地运行以下查询（例如，生成一个报告），你会发现没有索引会使查询速度变慢：

- select * from dept, emp
- where emp.deptno = dept.deptno and dept.deptno = :X;

那么，什么时候不需要对外键加索引呢？答案是，一般来说，当满足以下条件时不需要加索引：

- 没有从父表删除行。
- 没有更新父表的惟一键/主键值（当心工具有时会无意地更新主键!）。
- 没有从父表联结子表（如 DEPT 联结到 EMP）。

如果满足上述全部 3 个条件，那你完全可以跳过索引，不需要对外键加索引。如果满足以上的某个条件，就要当心加索引的后果。这是一种少有的情况，即 Oracle“过分地锁定了”数据。

6.2.7 锁升级

出现锁升级（lock escalation）时，系统会降低锁的粒度。举例来说，数据库系统可以把一个表的 100 个行级锁变成一个表级锁。现在你用的是“能锁住全部的一个锁”，一般而言，这还会锁住以前没有锁定的大量数据。如果数据库认为锁是一种稀有资源，而且想避免锁的开销，这些数据库中就会频繁使用锁升级。

注意 Oracle 不会升级锁，从来不会。

Oracle 从来不会升级锁，但是它会执行锁转换（lock conversion）或锁提升（lock promotion），这些词通常会与锁升级混淆。

注意 “锁转换”和“锁提升”是同义词。Oracle 一般称这个过程为“锁转换”。

Oracle 会尽可能地在最低级别锁定（也就是说，限制最少的锁），如果必要，会把这个锁转换为一个更受限的级别。例如，如果用 FOR UPDATE 子句从表中选择一行，就会创建两个锁。一个锁放在所选的行上（这是一个排他锁；任何人都不能以独占模式锁定这一行）。另一个锁是 ROW SHARE TABLE 锁，放在表本身上。这个锁能防止其他会话在表上放置一个排他锁，举例来说，这样能相应地防止这些会话改变表的结构。另一个会话可以修改这个表中的任何其他行，而不会有冲突。假设表中有一个锁定的行，这样就可以成功执行尽可能多的命令。

锁升级不是一个数据库“特性”。这不是我们想要的性质。如果数据库支持锁升级，就说明这个数据库的锁定机制中存在某些内部开销，而且管理数百个锁需要做大量的工作。在 Oracle 中，1 个锁的开销与 1 000 000 个锁是一样的，都没有开销。

6.3 锁类型

Oracle 中主要有 3 类锁，具体是：

- DML 锁（DML lock）：DML 代表数据操纵语言（Data Manipulation Language）。

一般来讲，这表示 SELECT、INSERT、UPDATE、MERGE 和 DELETE 语句。DML 锁机制允许并发执行数据修改。例如，DML 锁可能是特定数据行上的锁，或者是锁定表中所有行的表级锁。

- DDL 锁（DDL lock）：DDL 代表数据定义语言（Data Definition Language），如 CREATE 和 ALTER 语句等。DDL 锁可以保护对象结构定义。
- 内部锁和闩：Oracle 使用这些锁来保护其内部数据结构。例如，Oracle 解析一个查询并生成优化的查询计划时，它会把库缓存“临时闩”，将计划放在那里，以供其他会话使用。闩（latch）是 Oracle 采用的一种轻量级的低级串行化设备，功能上类似于锁。不要被“轻量级”这个词搞糊涂或蒙骗了，你会看到，闩是数据库中导致竞争的一个常见原因。轻量级指的是闩的实现，而不是闩的作用。

下面将更详细地讨论上述各个特定类型的锁，并介绍使用这些锁有什么影响。除了我在这里介绍的锁之外，还有另外一些锁类型。这一节以及下一节介绍的锁是最常见的，而且会保持很长时间。其他类型的锁往往只保持很短的一段时间。

6.3.1 DML 锁

DML 锁（DML Lock）用于确保一次只有一个人能修改某一行，而且你正在处理一个表时别人不能删除这个表。在你工作时，Oracle 会透明程度不一地为你加这些锁。

1. TX 锁（事务锁）

事务发起第一个修改时会得到 TX 锁（事务锁），而且会一直持有这个锁，直至事务执行提交（COMMIT）或回滚（ROLLBACK）。TX 锁用作一种排队机制，使得其他会话可以等待这个事务执行。事务中修改或通过 SELECT FOR UPDATE 选择的每一行都会“指向”该事务的一个相关 TX 锁。听上去好像开销很大，但实际上并非如此。要想知道这是为什么，需要从概念上对锁“居住”在哪里以及如何管理锁有所认识。在 Oracle 中，闩为数据的一个属性（第 10 章会给出 Oracle 块格式的一个概述）。Oracle 并 没有一个传统的锁管理器，不会用锁管理器为系统中锁定的每一行维护一个长长的列表。不过，其他的许多数据库却是这样做的，因为对于这些数据库来说，锁是一 种稀有资源，需要对锁的使用进行监视。使用的锁越多，系统要管理的方面就越多，所以在这些系统中，如果使用了“太多的”锁就会有问题。

如果数据库中有一个传统的基于内存的锁管理器，在这样一个数据库中，对一行锁定的过程一般如下：

(1) 找到想锁定的那一行的地址。

(2) 在锁管理器中排队（锁管理器必须是串行化的，因为这是一个常见的内存中的结构 。

(3) 锁定列表。

(4) 搜索列表，查看别人是否已经锁定了这一行。

(5) 在列表中创建一个新的条目，表明你已经锁定了这一行。

(6) 对列表解锁。

既然已经锁定了这一行，接下来就可以修改它了。之后，在你提交修改时，必须继续这个过程，如下：

(7) 再次排队。

(8) 锁住锁的列表。

(9) 在这个列表中搜索，并释放所有的锁。

(10) 对列表解锁。

可以看到，得到的锁越多，这个操作所花的时间就越多，修改数据前和修改数据之后耗费的时间都会增加。Oracle 不是这样做的。Oracle 中的锁定过程如下：

(1) 找到想锁定的那一行的地址。

(2) 到达那一行。

(3) 锁定这一行（如果这一行已经锁定，则等待锁住它的事务结束，除非使用了 NOWAIT 选项）。

仅此而已。由于闩为数据的一个属性，Oracle不需要传统的锁管理器。事务只是找到数据[5]，如果数据还没有被锁定，则对其锁定。有意思的是，找到数据时，它可能看上去被锁住了，但实际上并非如此。在Oracle中对数据行锁定时，行指向事务ID的一个副本，事务ID存储在包含数据的块中，释放锁时，事务ID却会保留下来。这个事务ID是事务所独有的，表示了回滚段号、槽和序列号。事务ID留在包含数据行的块上，可以告诉其他会话：你“拥有”这个数据（并非块上的所有数据都是你的，只是你修改的那一行“归你所有”）。另一个会话到来时，它会看到锁ID，由于锁ID表示一个事务，所以可以很快地查看持有这个锁的事务是否还是活动的。如果锁不活动，则允许会话访问这个数据。如果锁还是活动的，会话就会要求一旦释放锁就得到通知。因此，这就有了一个排队机制：请求锁的会话会排队，等待目前拥有锁的事务执行，然后得到数据。

以下是一个小例子，展示了这到底是怎么回事，这里使用了 3 个 V$ 表：

- V$TRANSACTION，对应每个活动事务都包含一个条目。
- V$SESSION，显示已经登录的会话。
- V$LOCK，对应持有所有 enqueue 队列锁以及正在等待锁的会话，都分别包含一个条目。这并不是说，对于表中被会话锁定的每一行，这个视图中就有相应的一行。你不会看到这种情况。如前所述，不存在行级锁的一个主列表。如果某个会话将 EMP 表中的一行锁定，V$LOCK 视图中就有对应这个会话的一行来指示这一事实。如果一个会话锁定了 EMP 表中的数百万行，V$LOCK 视图中对应这个会话还是只有一行。这个视图显示了各个会话有哪些队列锁。

首先启动一个事务（如果你没有 DEPT 表的一个副本，只需使用 CREATE TABLE AS SELECT 来建立一个副本）：

```
ops$tkyte@ORA10G> update dept set deptno = deptno+10;

4 rows updated.
```

下面来看看此时系统的状态。这个例子假设是一个单用户系统；否则，在 V$TRANSACTION 中可以看到多行。即使在一个单用户的系统中，如果看到 V$TRANSACTION 中有多行也不要奇怪，因为许多后台 Oracle 进程可能也会执行事务。

```
ops$tkyte@ORA10G> select username,
  2 v$lock.sid,
  3 trunc(id1/power(2,16)) rbs,
  4 bitand(id1,to_number('ffff','xxxx'))+0 slot,
  5 id2 seq,
  6 lmode,
  7 request
  8 from v$lock, v$session
  9 where v$lock.type = 'TX'
  10 and v$lock.sid = v$session.sid
  11 and v$session.username = USER;

USERNAME SID RBS      SLOT       SEQ      LMODE       REQUEST
--------- ---- --- ---- ------ ----- -------
OPS$TKYTE          145      4            12        16582     6          0

ops$tkyte@ORA10G> select XIDUSN, XIDSLOT, XIDSQN
  2 from v$transaction;
XIDUSN     XIDSLOT    XIDSQN
---------- ---------- ----------
4              12             16582
```

这里有几点很有意思：

- $LOCK 表中的 LMODE 为 6，REQUEST 为 0。如果在 Oracle Server Reference 手册中查看 V$LOCK 表的定义，会发现 LMODE=6 是一个排他锁。请求（REQUEST）值为 0 则意味着你没有发出请求；也就是说，你拥有这个锁。

- 这个表中只有一行。V$LOCK表更应算是一个队列表而不是一个锁表。许多人都认为V$LOCK中会有 4 行，因为我们锁定了 4 行。不过，你要记住，Oracle不会在任何地方存储行级锁的列表（也就是说，不会为每一个被锁定的行维护一个主列表）。要查看某一行是否被锁定，必须直接找到这一行[6]。

- □ 我选择了ID1 和ID2 列，并对它们执行了一些处理。Oracle需要保存 3 个 16 位的数，但是对此只有两个列。所以，第一个列ID1 保存着其中两个数。通过用 trunc(id1/power (2,16))rbs除以 2^16[7]，并用 bitand(id1,to_number('ffff','xxxx'))+0 slot 把高位屏蔽[8]，就能从这个数中找回隐藏的两个数。

- RBS、SLOT 和 SEQ 值与 V$TRANSACTION 信息匹配。这就是我的事务 ID。

下面使用同样的用户名启动另一个会话，更新 EMP 中的某些行，并希望试图更新 DEPT：

```
ops$tkyte@ORA10G> update emp set ename = upper(ename);

14 rows updated.



ops$tkyte@ORA10G> update dept set deptno = deptno-10;
```

现在这个会话会阻塞。如果再次运行 V$查询，可以看到下面的结果：

```
ops$tkyte@ORA10G> select username,
  2                 v$lock.sid,
  3                 trunc(id1/power(2,16)) rbs,
  4                 bitand(id1,to_number('ffff','xxxx'))+0 slot,
  5                 id2 seq,
  6                 lmode,
  7                 request
  8          from v$lock, v$session
  9          where v$lock.type = 'TX'
 10                and v$lock.sid = v$session.sid
```

```
  11              and v$session.username = USER;


USERNAME SID      RBS     SLOT     SEQ      LMODE REQUEST

---------          ----       ---       ----       ------      -----          -------

OPS$TKYTE          144     4          12         16582       0 6

OPS$TKYTE          144     5          34         1759        6 0

OPS$TKYTE          145     4          12         16582       6 0


ops$tkyte@ORA10G> select XIDUSN, XIDSLOT, XIDSQN

  2 from v$transaction;


XIDUSN                XIDSLOT          XIDSQN

----------    ----------       ----------

5             34             1759

4             12             16582
```

这里可以看到开始了一个新的事务，事务 ID 是(5,34,1759)。这一次，这个新会话（SID=144）在 V$LOCK 中有两行。其中一行表示它所拥有的锁（LMODE=6）。另外还有一行，显示了一个值为 6 的 REQUEST。这是一个对排他锁的请求。有意思的是，这个请求行的 RBS/SLOT/SEQ 值正是锁持有者的事务 ID。SID=145 的事务阻塞了 SID=144 的事务。只需执行 V$LOCK 的一个自联结，就可以更明确地看出这一点：

```
ops$tkyte@ORA10G> select

  2              (select username from v$session where sid=a.sid) blocker,

  3              a.sid,

  4              ' is blocking ',

  5              (select username from v$session where sid=b.sid) blockee,
```

```
  6                 b.sid
  7         from v$lock a, v$lock b
  8         where a.block = 1
  9                 and b.request > 0
 10                and a.id1 = b.id1
 11                and a.id2 = b.id2;

BLOCKER       SID      'ISBLOCKING'                        BLOCKEE         SID
---------             ----       -------------          ---------              ----
OPS$TKYTE             145                 is blocking         OPS$TKYTE     144
```

现在，如果提交原来的事务（SID=145），并重新运行锁查询，可以看到请求行不见了：

```
ops$tkyte@ORA10G> select username,
  2                 v$lock.sid,
  3                 trunc(id1/power(2,16)) rbs,
  4                 bitand(id1,to_number('ffff','xxxx'))+0 slot,
  5                 id2 seq,
  6                 lmode,
  7                 request
  8          from v$lock, v$session
  9          where v$lock.type = 'TX'
 10                and v$lock.sid = v$session.sid
 11                and v$session.username = USER;


USERNAME SID      RBS     SLOT SEQ     LMODE     REQUEST
```

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

OPS$TKYTE            144     5          34        1759        6       0



ops$tkyte@ORA10G> select XIDUSN, XIDSLOT, XIDSQN

    2 from v$transaction;

XIDUSN XIDSLOT          XIDSQN

----------     ----------      ----------

5              34              1759
```

另一个会话一旦放弃锁，请求行就会消失。这个请求行就是排队机制。一旦事务执行，数据库会唤醒被阻塞的会话。当然，利用各种 GUI 工具肯定能得到更“好看”的显示，但是，必要时对你要查看的表有所了解还是非常有用的。

不过，我们还不能说自己已经很好地掌握了 Oracle 中行锁定是如何工作的，因为还有最后一个主题需要说明：如何用数据本身来管理锁定和事务信息。这是块开销的一部分。在第 9 章中，我们会详细分析块的格式，但是现在只需知道数据库块的最前面有一个“开销”空间，这里会存放该块的一个事务表，了解这一点就足够了。对于锁定了该块中某些数据的各个“实际”事务，在这个事务表中都有一个相应的条目。这个结构的大小由创建对象时 CREATE 语句上的两个物理属性参数决定：

- INITRANS：这个结构初始的预分配大小。对于索引和表，这个大小默认为 2（不过我已经提出，Oracle SQL Reference 手册中与此有关的说明有问题）。
- MAXTRANS：这个结构可以扩缩到的最大大小。它默认为 255，在实际中，最小值为 2。在 Oracle 10g 中，这个设置已经废弃了，所以不再使用。这个版本中的 MAXTRANS 总是 255。

默认情况下，每个块最开始都有两个事务槽。一个块上同时的活动事务数受 MAXTRANS 值的约束，另外也受块上空间可用性的限制。如果没有足够的空间来扩大这个结构，块上就无法得到 255 个并发事务。

我们可以创建一个具有受限 MAXTRANS 的表，来专门展示这是如何工作的。为此，需要使用 Oracle9i 或以前的版本，因为 Oracle 10g 中会忽略 MAXTRANS。在 Oracle 10g 中，只要块上的空间允许，即使设置了 MAXTRANS，Oracle 也会不受约束地扩大事务表。在 Oracle9i 及以前的版本中，一旦块达到了 MAXTRANS 值，事务表就不会再扩大了，例如：

```
ops$tkyte@ORA9IR2> create table t ( x int ) maxtrans 2;

Table created.
```

```
ops$tkyte@ORA9IR2> insert into t select rownum from all_users;

24 rows created.



ops$tkyte@ORA9IR2> commit;

Commit complete.



ops$tkyte@ORA9IR2> select distinct dbms_rowid.rowid_block_number(rowid) from t;

DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)

---------------------------------

18
```

因此，我们有 24 行，而且经验证，它们都在同一个数据库块上。现在，在一个会话中发出以下命令：

```
ops$tkyte@ORA9IR2> update t set x = 1 where x = 1;

1 row updated.
```

在另一个会话中，发出下面的命令：

```
ops$tkyte@ORA9IR2> update t set x = 2 where x = 2;

1 row updated.
```

最后，在第三个会话中，发出如下命令：

```
ops$tkyte@ORA9IR2> update t set x = 3 where x = 3;
```

现在，由于这 3 行在同一个数据库块上，而且我们将 MAXTRANS（该块的最大并发度）设置为 2，所以第 3 个会话会被阻塞。

注意 要记住，在 Oracle 10g 中，不会发生上例中出现的阻塞，不管怎样，MAXTRANS 都会设置为 255。在这个版本中，只有当块上没有足够的空间来扩大事务表时，才会看到这种阻塞。

从这个例子可以看出，如果多个MAXTRANS事务试图同时访问同一个块时会发生什么情况[9]。类似地，如果INITRANS设置得很低，而且块上没有足够的空间来动态地扩缩事务，也会出现阻塞。大多数情况下，INITRANS的默认值 2 就足够了，因为事务表会动态扩大（只要空间允许）。但是在某些环境中，可能需要加大这个设置来提高并发性，并减少等待。比如，在频繁修改的表上就可能要增加INITRANS设置，或者更常见的是，对于频繁修改的索引也可能需要这么做，因为索引块中的行一般比表中的行多。你可能需要增加PCTFREE（见

第 10 章的讨论）或INITRANS，从而在块上提前预留足够的空间以应付可能的并发事务数。尤其是，如果你预料到块开始时几乎是满的（这说明块上没有空间来动态扩缩事务结构），则更需要增加PCTFREE或INITRANS。

2. TM (DML Enqueue)锁

TM 锁（TM lock）用于确保在修改表的内容时，表的结构不会改变。例如，如果你已经更新了一个表，会得到这个表的一个 TM 锁。这会防止另一个用户在该表上执行 DROP 或 ALTER 命令。如果你有表的一个 TM 锁，而另一位用户试图在这个表上执行 DDL，他就会得到以下错误消息：

```
drop table dept

                *

ERROR at line 1:

ORA-00054: resource busy and acquire with NOWAIT specified
```

初看上去，这是一条让人摸不着头脑的消息，因为根本没有办法在 DROP TABLE 上指定 NOWAIT 或 WAIT。如果你要执行的操作将要阻塞，但是这个操作不允许阻塞，总是会得到这样一条一般性的消息。前面已经看到，如果在一个锁定的行上发出 SELECT FOR UPDATE NOWAIT 命令，也会得到同样的消息。

以下显示了这些锁在 V$LOCK 表中是什么样子：

```
ops$tkyte@ORA10G> create table t1 ( x int );

Table created.



ops$tkyte@ORA10G> create table t2 ( x int );

Table created.



ops$tkyte@ORA10G> insert into t1 values ( 1 );

1 row created.



ops$tkyte@ORA10G> insert into t2 values ( 1 );

1 row created.
```

```
ops$tkyte@ORA10G> select (select username
  2                 from v$session
  3                where sid = v$lock.sid) username,
  4                 sid,
  5                 id1,
  6                 id2,
  7                 lmode,
  8                 request, block, v$lock.type
  9          from v$lock
 10        where sid = (select sid
 11                              from v$mystat
 12                              where rownum=1)
 13 /

USERNAME SID     ID1        ID2        LMODE      REQUEST    BLOCK    TYPE
---------          ----      -------    ------      -----         -------          -----        ----
OPS$TKYTE          161       262151              16584         6          0     0
TX
OPS$TKYTE          161       62074    0          3              0          0     TM
OPS$TKYTE          161       62073    0          3              0          0     TM

ops$tkyte@ORA10G> select object_name, object_id
  2 from user_objects
```

```
  3 where object_name in ('T1','T2')

  4 /

OBJECT_NAME        OBJECT_ID

------------        ----------

T1                  62073

T2                  62074
```

尽管每个事务只能得到一个 TX 锁，但是 TM 锁则不同，修改了多少个对象，就能得到多少个 TM 锁。在此，有意思的是，TM 锁的 ID1 列就是 DML 锁定对象的对象 ID，所以，很容易发现哪个对象持有这个锁。

关于 TM 锁还有另外一个有意思的地方：系统中允许的 TM 锁总数可以由你来配置（有关细节请见 Oracle Database Reference 手册中的 DML_LOCKS 参数定义）。实际上，这个数可能设置为 0。但这并不是说你的数据库变成了一个只读数据库（没有锁），而是说不允许 DDL。在非常专业的应用（如 RAC 实现）中，这一点就很有用，可以减少实例内可能发生的协调次数。通过使用 ALTER TABLE TABLENAME DISABLE TABLE LOCK 命令，还可以逐对象地禁用 TM 锁。这是一种快捷方法，可以使意外删除表的“难度更大”，因为在删除表之前，你必须重新启用表锁。还能用它来检测由于外键未加索引而导致的全表锁（前面已经讨论过）。

6.3.2 DDL 锁

在 DDL 操作中会自动为对象加 DDL 锁（DDL Lock），从而保护这些对象不会被其他会话所修改。例如，如果我执行一个 DDL 操作 ALTERTABLE T，表 T 上就会加一个排他 DDL 锁，以防止其他会话得到这个表的 DDL 锁和 TM 锁。在 DDL 语句执行期间会一直持有 DDL 锁，一旦操作执行就立即释放 DDL 锁。实际上，通常会把 DDL 语句包装在隐式提交（或提交/回滚对）中来执行这些工作。由于这个原因，在 Oracle 中 DDL 一定会提交。每条 CREATE、ALTER 等语句实际上都如下执行（这里用伪代码来展示）：

```
Begin
    Commit;
    DDL-STATEMENT
    Commit;
Exception
    When others then rollback;
End;
```

因此，DDL 总会提交（即使提交不成功也会如此）。DDL 一开始就提交，一定要知道这一点。它首先提交，因此如果必须回滚，它不会回滚你的事务。如果你执行了 DDL，它会使你所执行的所有未执行的工作成为永久性的，即使 DDL 不成功也会如此。如果你需要执行 DDL，但是不想让它提交你现有的事务，就可以使用一个自治事务（autonomous transaction）。

有 3 种类型的 DDL 锁：

- 排他 DDL 锁（Exclusive DDL lock）：这会防止其他会话得到它们自己的 DDL 锁或 TM（DML）锁。这说明，在 DDL 操作期间你可以查询一个表，但是无法以任何方式修改这个表。
- 共享 DDL 锁（Share DDL lock）：这些锁会保护所引用对象的结构，使之不会被其他会话修改，但是允许修改数据。
- 可中断解析锁（Breakable parse locks）：这些锁允许一个对象（如共享池中缓存的一个查询计划）向另外某个对象注册其依赖性。如果在被依赖的对象上执行 DDL，Oracle 会查看已经对该对象注册了依赖性的对象列表，并使这些对象无效。因此，这些锁是“可中断的”，它们不能防止 DDL 出现。

大多数 DDL 都带有一个排他 DDL 锁。如果发出如下一条语句：

```
Alter table t add new_column date;
```

在执行这条语句时，表 T 不能被别人修改。在此期间，可以使用 SELECT 查询这个表，但是大多数其他操作都不允许执行，包括所有 DDL 语句。在 Oracle 中，现在有些 DDL 操作没有 DDL 锁也可以发生。例如，可以发出以下语句：

```
create index t_idx on t(x) ONLINE;
```

ONLINE 关键字会改变具体建立索引的方法。Oracle 并不是加一个排他 DDL 锁来防止数据修改，而只会试图得到表上的一个低级（mode 2）TM 锁。这会有效地防止其他 DDL 发生，同时还允许 DML 正常进行。Oracle 执行这一“壮举”的做法是，为 DDL 语句执行期间对表所做的修改维护一个记录，执行 CREATE 时再把这些修改应用至新的索引。这样能大大增加数据的可用性。

另外一类 DDL 会获得共享 DDL 锁。在创建存储的编译对象（如过程和视图）时，会对依赖的对象加这种共享 DDL 锁。例如，如果执行以下语句：

```
Create view MyView

as

    select *

    from emp, dept

    where emp.deptno = dept.deptno;
```

表 EMP 和 DEPT 上都会加共享 DDL 锁，而 CREATE VIEW 命令仍在处理。可以修改这些表的内容，但是不能修改它们的结构。

最后一类 DDL 锁是可中断解析锁。你的会话解析一条语句时，对于该语句引用的每一个对象都会加一个解析锁。加这些锁的目的是：如果以某种方式删除或修改了一个被引用的对象，可以将共享池中已解析的缓存语句置为无效（刷新输出）。

有一个意义非凡的视图可用于查看这个信息，即 DBA_DDL_LOCKS 视图。对此没有相应的 V$视图。DBA_DDL_LOCKS 视图建立在更神秘的 X$表基础上，而且默认情况下，你的数据库上不会安装这个视图。可以运行[ORACLE_HOME]/rdbms/admin 目录下的 catblock.sql 脚本来安装这个视图以及其他锁视图。必须作为用户 SYS 来执行这个脚本才能成功。一旦执行了这个脚本，可以对视图运行一个查询。例如，在一个单用户数据库中，我看到以下结果：

```
ops$tkyte@ORA10G> select session_id sid, owner, name, type,
  2 mode_held held, mode_requested request
  3 from dba_ddl_locks;

SID    OWNER           NAME                      TYPE                      HELD
   REQUEST
---    ---------       ----------                ----------                -
---    ---------
161    SYS             DBMS_UTILITY              Body                      Null
 None
161    SYS             DBMS_UTILITY              Body                      Null
 None
161    SYS             DBMS_APPLICATION_INFO     Table/Procedure/Type      Null
  None
161    OPS$TKYTE       OPS$TKYTE                 18                        Null
 None
161    SYS             DBMS_OUTPUT               Body                      Null
 None
161    SYS             DATABASE                  18                        Null
None
161    SYS             DBMS_UTILITY              Table/Procedure/Type      Null
 None
161    SYS             DBMS_UTILITY              Table/Procedure/Type      Null
```

```
None

161     SYS                 PLITBLM                     Table/Procedure/Type      Null
 None

161     SYS                 DBMS_APPLICATION_INFO       Body                      Null
  None

161     SYS                 DBMS_OUTPUT                 Table/Procedure/Type      Null
 None

11 rows selected.
```

这些就是我的会话“锁定”的所有对象。我对一组DBMS_*包加了可中断解析锁。这是使用SQL*Plus的副作用；例如，它会调用DBMS_APPLICATION_INFO[10]。可以看到不同的对象可能有不止一个副本，这是正常的，这只是表明，共享池中有多个“事物”引用了这些对象。需要指出有意思的一点，在这个视图中，OWNER列不是锁的所有者；而是所锁定对象的所有者。正是由于这个原因，所以你会看到多个SYS行。SYS拥有这些包，但是它们都属于我的会话。

要看到一个实际的可中断解析锁，下面先创建并运行存储过程 P：

```
ops$tkyte@ORA10G> create or replace procedure p as begin null; end;

  2 /

Procedure created.



ops$tkyte@ORA10G> exec p

PL/SQL procedure successfully completed.
```

过程 P 现在会出现在 DBA_DDL_LOCKS 视图中。我们有这个过程的一个解析锁：

然后重新编译这个过程，并再次查询视图：

```
ops$tkyte@ORA10G> select session_id sid, owner, name, type,

  2 mode_held held, mode_requested request

  3 from dba_ddl_locks

  4 /

SID     OWNER               NAME                        TYPE                      HELD
REQUEST
```

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

161 OPS$TKYTE    P                      Table/Procedure/Type       Null    None

161    SYS             DBMS_UTILITY              Body                    Null
 None

161    SYS             DBMS_UTILITY              Body                    Null
 None

...

161    SYS             DBMS_OUTPUT               Table/Procedure/Type    Null
 None

12 rows selected.
```

可以看到，现在这个视图中没有 P 了。我们的解析锁被中断了。

这个视图对开发人员很有用，发现测试或开发系统中某段代码无法编译时，将会挂起并最终超时。这说明，有人正在使用这段代码（实际上在运行这段代码），你可以使用这个视图来查看这个人是谁。对于 GRANTS 和对象的其他类型的 DDL 也是一样。例如，无法对正在运行的过程授予 EXECUTE 权限。可以使用同样的方法来发现潜在的阻塞者和等待者。

6.3.3 闩

闩（latch）是轻量级的串行化设备，用于协调对共享数据结构、对象和文件的多用户访问。

闩就是一种锁，设计为只保持极短的一段时间（例如，修改一个内存中数据结构所需的时间）。闩用于保护某些内存结构，如数据库块缓冲区缓存或共享池中的库缓存。一般会在内部以一种“愿意等待”（willing to wait）模式请求闩。这说明，如果闩不可用，请求会话会睡眠很短的一段时间，并在以后再次尝试这个操作。还可以采用一种“立即”（immediate）模式请求其他闩，这与 SELECT FOR UPDATE NOWAIT 的 思想很相似，说明这个进程会做其他事情（如获取另一个与之相当的空闲闩），而不只是坐而等待这个闩直到它可用。由于许多请求者可能会同时等待一个闩，你会 看到一些进程等待的时间比其他进程要长一些。闩的分配相当随机，这要看运气好坏了。闩释放后，紧接着不论哪个会话请求闩都会得到它。等待闩的会话不会排 队，只是一大堆会话在不断地重试。

Oracle 使用诸如“测试和设置”（test and set）以及“比较和交换”（compare and swap）之类的原子指令来处理闩。由于设置和释放闩的指令是原子性的，尽管可能有多个进程在同时请求它，但操作系统本身可以保证只有一个进程能测试和设置闩。指令 仅仅是一个指令而已，它执行得可能非常快。闩只保持很短的时间，而且提供了一种清理机制，万一某个闩持有者在持有闩时异常地“死掉了”，就能执行清理。这 个清理过程由 PMON 执行。

队列锁（enqueue）在前面已经讨论过，这也是一种更复杂的串行化设备，例如，在更

新数据库表中的行时就会使用队列锁。与闩的区别在于，队列锁允许请求者“排队”等待资源。对于闩请求，请求者会话会立即得到通知是否得到了闩。而对于队列锁，请求者会话会阻塞，直至真正得到锁。

注意 使用 SELECT FOR UPDATE NOWAIT 或 WAIT [n]，你还可以决定倘若会话被阻塞，则并不等待一个队列锁，但是如果确实阻塞并等待，就会在一个队列中等待。

因此，队列锁没有闩快，但是它确实能提供闩所没有的一些功能。可以在不同级别上得到队列锁，因此可以有多个共享锁以及有不同程度共享性的锁。

1. 闩“自旋”

关于闩还要了解一点：闩是一种锁，锁是串行化设备，而串行化设备会妨碍可扩缩性。如果你的目标是构建一个能在 Oracle 环境中很好地扩缩的应用，就必须寻找合适的方法和解决方案，尽量减少所需执行的闩定的量。

有些活动尽管看上去很简单（如解析一条 SQL 语句），也会为共享池中的库缓存和相关结构得到并释放数百个或数千个闩。如果我们有一个闩，可能会有另外某个人在等待这个闩。而当我们想要得到一个闩时，也许我们自己也必须等待（因为别人拥有着这个闩）。

等待闩可能是一个代价很高的操作。如果闩不是立即可用的，我们就得等待（大多数情况下都是如此），在一台多 CPU 机器上，我们的会话就会自旋（spin），也就是说，在循环中反复地尝试来得到闩。出现自旋的原因是，上下文切换（context switching）的开销很大（上下文切换是指被“踢出”CPU，然后又必须调度回 CPU）。所以，如果进程不能立即得到闩，我们就会一直呆在 CPU 上，并立即再次尝试，而不是先睡眠，放弃 CPU，等到必须调度回 CPU 时才再次尝试。之所以呆在 CPU 上，是因为我们指望闩的持有者正在另一个 CPU 上忙于处理（由于闩设计为只保持很短的时间，所以一般是这样），而且会很快放弃闩。如果出现自旋并不断地尝试想得到闩，但是之后还是得不到闩，此时我们的进程才会睡眠，或者让开 CPU，而让其他工作进行。得到闩的伪代码如下所示：

```
Attempt to get Latch

If Latch gotten

Then

    return SUCCESS

Else

    Misses on that Latch = Misses+1;

    Loop

        Sleeps on Latch = Sleeps + 1

        For I in 1 .. 2000
```

```
        Loop
            Attempt to get Latch
            If Latch gotten
            Then
                Return SUCCESS
            End if
        End loop
        Go to sleep for short period
    End loop
End if
```

其逻辑是，尝试得到闩，如果失败，则递增未命中计数（miss count），这个统计结果可以在 Statspack 报告中看到，或者直接查询 V$LATCH 视图也可以看到。一旦进程未命中，它就会循环一定的次数（有一个参数能控制这个次数，通常设置为 2 000，但是这个参数在文档中未做说明），反复地试图得到闩。如果某次尝试成功，它就会返回，我们能继续处理。如果所有尝试都失败了，这个进程就会将该闩的睡眠计数（sleep count）递增，然后睡眠很短的一段时间。醒来时，整个过程会再重复一遍。这说明，得到一个闩的开销不只是“测试和设置”操作这么简单，我们尝试得到闩时，可能会耗费大量的 CPU 时间。系统看上去非常忙（因为消耗了很多 CPU 时间），但是并没有做多少实际的工作。

2. 测量闩定共享资源的开销

举个例子，我们来研究闩定共享池的开销。我们会把一个编写得很好的程序和一个编写得不太好的程序进行比较，前者使用了绑定变量，而在编写得不好的程序中，每条语句使用了 SQL 直接量或各不相同的 SQL。为此，我们使用了一个很小的 Java 程序，它只是登录 Oracle，关掉自动提交（所有 Java 程序在连接数据库后紧接着都应这么做），并通过一个循环执行 25 000 条不同的 INSERT 语句。我们会执行两组测试：在第一组测试中，程序不使用绑定变量；在第二组测试中，程序会使用绑定变量。

要评估这些程序以及它们在多用户环境中的行为，我喜欢用 Statspack 来收集度量信息，如下：

(1) 执行一个 Statspack 快照来收集系统的当前状态。

(2) 运行程序的 N 个副本，每个程序向其自己的数据库表中插入（INSERT），以避免所有程序都试图向一个表中插入而产生的竞争。

(3) 在最后一个程序副本执行后，紧接着取另一个快照。

然后只需打印出 Statspack 报告，并查看完成 N 个程序副本需要多长时间，使用了多少

CPU 时间，主要的等待事件是什么，等等。

这些测试在一台双 CPU 机器上执行，并启用了超线程（看上去就好像有 4 个 CPU）。给定两个物理 CPU，你可能以为能线性扩缩，也就是说，如果一个用户使用了一个 CPU 单位来处理其插入，那么两个客户可能需要两个 CPU 单位。你会发现，这个假设尽管听上去好像是正确的，但可能并不正确（随后将会看到，不正确的程度取决于你的编程水平）。如果所要执行的处理不需要共享资源，这么说可能是正确的，但是我们的进程确实会使用一个共享资源，即共享池（Shared pool）。我们需要闩定共享池来解析 SQL 语句，为什么要闩定共享池呢？因为这是一个共享数据结构，别人在读取这个共享资源时，我们不能对其进行修改，另外如果别人正在修改它，我们就不能读取。

注意 我分别使用 Java、PL/SQL、Pro*C 和其他语言执行过这些测试。每一次的最终结果基本上都一样。这里所展示和讨论的内容适用于所有语言和所有数据库接口。这个例子之所以选择 Java，是因为我发现处理 Oracle 数据库时，Java 和 Visual Basic 应用最有可能不使用绑定变量。

□ 不使用绑定变量

在第一个实例中，我们的程序不使用绑定变量，而是使用串连接来插入数据：

```
import java.sql.*;

public class instest

{

    static public void main(String args[]) throws Exception

    {

        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        Connection

        conn = DriverManager.getConnection

            ("jdbc:oracle:thin:@dellpe:1521:ora10gr1",

            "scott","tiger");

        conn.setAutoCommit( false );

        Statement stmt = conn.createStatement();

        for( int i = 0; i < 25000; i++ )
```

```
            {
                stmt.execute
                    ("insert into "+ args[0] +
                    " (x) values(" + i + ")" );
            }
            conn.commit();
            conn.close();
        }
}
```

我以“单用户”模式运行这个测试，Statspack 报告返回了以下信息：

```
Elapsed:          0.52 (mins)
Cache Sizes (end)
~~~~~~~~~~~~~~~~~
            Buffer Cache:          768M      Std Block Size:         8K
            Shared Pool Size:      244M      Log Buffer:         1,024K
Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                ---------------  ---------------
...
            Parses:                      810.58          12,564.00
            Hard parses:                 807.16          12,511.00
....
```

```
Top 5 Timed Events

~~~~~~~~~~~~~~~~~~~                                    % Total

Event                                  Waits    Time (s)    Call Time

----------------------------------------  ----------  ----------  ---------

CPU time                                                   26       55.15

class slave wait                            2            10       21.33

Queue Monitor Task Wait                     2            10       21.33

log file parallel write                    48             1        1.35

control file parallel write                14             0         .51
```

这里加入了 SGA 配置以供参考，不过其中最重要的统计信息是：

- 耗用时间大约是 30 秒
- 每秒有 807 次硬解析
- 使用了 26 秒的 CPU 时间

现在，如果要同时运行这样的两个程序，你可能会认为硬解析会跃升至每秒 1 600 个（毕竟，我们有两个可用的 CPU），并认为 CPU 时间会加倍为大约 52 秒。下面看一下：

```
Elapsed:          0.78 (mins)

Load Profile

~~~~~~~~~~~~                        Per Second            Per Transaction

                                    ---------------       ---------------

                Parses:                1,066.62              16,710.33

                Hard parses:           1,064.28              16,673.67


Top 5 Timed Events

~~~~~~~~~~~~~~~~~~~                                    % Total

Event                                  Waits    Time (s)    Call Time
```

CPU time	74		97.53
log file parallel write	53	1	1.27
latch: shared pool	406	1	.66
control file parallel write	21	0	.45
log file sync	6	0	.04

可以发现，硬解析数比预计的稍微多了一点，但是 CPU 时间是原来的 3 倍而不是两倍！怎么会这样呢？答案就在于 Oracle 的闩定实现。在这台多 CPU 机器上，无法立即得到一个闩时，我们就会“自旋”。自旋行为本身会消耗 CPU 时间。进程 1 多次尝试想要得到共享池的闩，但最终只是一再地发现进程 2 持有着这个闩，所以进程 1 必须自旋并等待（这会消耗 CPU 时间）。反过来对进程 2 也是一样，通过多次尝试，它发现进程 1 正持有着所需资源的闩。所以，很多处理时间都没有花在正事上，只是在等待某个资源可用。如果把 Statspack 报告向下翻页到“Latch Sleep Breakdown”报告部分，可以发现：

Latch Name	Requests	Misses	Sleeps	Sleeps 1->3+
shared pool	1,126,006	229,537	406	229135/398/4/0
library cache	1,108,039	45,582	7	45575/7/0/0

注意到这里 SLEEPS 列怎么出现了一个 406 呢？这个 406 对应于前面“Top 5 Timed Events”报告中报告的等待数。这个报告显示了自旋循环中尝试得到闩并且失败的次数。这说明，“Top 5”报告只是显示了闩定问题的冰山一角，而没有给出共有 229 537 次未命中这一信息（这说明我们尝试得到闩时陷入了自旋）。尽管这里存在一个严重的硬解析问题，但是分析“Top 5”报告后，我们可能想不到：“这里有一个硬解析问题”。为了完成两个单位的工作，这里需要使用 3 个单位的 CPU 时间。其原因就在于：我们需要一个共享资源（即共享池），这正是闩定的本质所在。不过，除非我们知道闩定实现的原理，否则可能很难诊断与闩定相关的问题。简单地看一下 Statspack 报告，从“Top 5”部分我们可能会漏掉这样一个事实：此时存在很糟糕的扩缩问题。只有更深入地研究 Statspack 报告的闩定部分才会发现这个问题。

另外，由于存在这种自旋，通常不可能确定系统使用多少 CPU 时间，从这个两用户的测试所能知道的只是：我们使用了 74 秒的 CPU 时间，而且力图得到共享池闩时未命中次数共有 229 537 次。我们不知道每次得不到闩时要尝试多少次 ，所以没有具体的办法来度量有多少 CPU 时间花在自旋上，而有多少 CPU 时间用于处理。要得到这个信息，我们需要多个数据点。

在我们的测试中，由于有一个单用户例子可以对照比较，因此可以得出结论：大约 22

秒的 CPU 时间花费在闩的自旋上，这些 CPU 时间白白浪费在等待资源上。

□ 使用了绑定变量

现在来看与上一节相同的情况，不过，这一次使用的程序在处理时使用的闩要少得多。还是用原来的 Java 程序，但把它重写为要使用绑定变量。为此，把 Statement 改为 PreparedStatement，解析一条 INSERT 语句，然后在循环中反复绑定并执行这个 PreparedStatement：

```
import java.sql.*;

public class instest

{

    static public void main(String args[]) throws Exception

    {

    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

    Connection

        conn = DriverManager.getConnection

            ("jdbc:oracle:thin:@dellpe:1521:ora10gr1",

            "scott","tiger");

    conn.setAutoCommit( false );

    PreparedStatement pstmt =

        conn.prepareStatement

            ("insert into "+ args[0] + " (x) values(?)" );

    for( int i = 0; i < 25000; i++ )

    {

        pstmt.setInt( 1, i );

        pstmt.executeUpdate();

    }
```

```
        conn.commit();

        conn.close();

        }

}
```

与前面“不使用绑定变量”的例子一样，下面来看所生成的单用户情况和两个用户情况下的 Statspack 报告。可以看到这里有显著的差别。以下是单用户情况下的报告：

```
Elapsed:          0.12 (mins)

Load Profile

~~~~~~~~~~~~                    Per Second              Per Transaction

                                ---------------         ---------------

...

                Parses:                 8.43                    29.50

                Hard parses:            0.14                    0.50


Top 5 Timed Events

~~~~~~~~~~~~~~~~~~                                      % Total

Event                                   Waits       Time (s)    Call Time

----------------------------------------    ------------    -----------     ---------

CPU time                                                    4       86.86

log file parallel write                     49              0       10.51

control file parallel write                 4               0       2.26

log file sync                               4               0       .23

control file sequential read                542             0       .14
```

差别确实很大，不使用绑定变量的例子中需要 26 秒的 CPU 时间，现在只需 4 秒。原来每秒钟有 807 次硬解析，现在仅为每秒 0.14 次。甚至耗用时间也从 45 秒大幅下降到 8 秒。没有使用绑定变量时，我们的 CPU 时间中有 5/6 的时间都用于解析 SQL。这并非都是闩导

致的，因为没有使用绑定变量时，解析和优化 SQL 也需要许多 CPU 时间。解析 SQL 是 CPU 密集型操作（需要耗费大量 CPU 时间），不过如果大幅增加 CPU 时间，但其中 5/6 的 CPU 时间都只是用来执行我们并不需要的解析，而不是做对我们有用的事情，这个代价实在太昂贵了。

再来看两个用户情况下的测试，结果看上去更好：

```
Elapsed:          0.20 (mins)

Load Profile

~~~~~~~~~~~~~                 Per Second            Per Transaction

                                ---------------       ---------------

              Parses:                 6.58                  26.33

              Hard parses:            0.17                   0.67



Top 5 Timed Events

~~~~~~~~~~~~~~~~~~                                    % Total

Event                                 Waits         Time (s)     Call Time

----------------------------------------------      ------------  -----------  ---------

CPU time                                                    11        89.11

log file parallel write                    48               1         9.70

control file parallel write                 4               0          .88

log file sync                               5               0          .23

log buffer space                            2               0          .05
```

CPU 时间大约是单用户测试用例所报告 CPU 时间的 2～2.5 倍。

注意　由于取整，4 秒的 CPU 时间实际上是指 3.5～4.49 秒之间，11 实际上表示 10.5～11.49 秒。

另外，使用绑定变量时，与不使用绑定变量的一个用户所需的 CPU 时间相比，两个用户使用的 CPU 时间还不到前者的一半！查看这个 Statspack 报告中的闩部分时，我发现，如果使用了绑定变量，则根本没有闩等待，对共享池和库缓存的竞争太少了，所以甚至没有相关的报告。实际上，再进一步挖掘还可以发现，使用绑定变量时，两用户测试中请求共享池

闩的次数是 50 367 次，而在前面不使用绑定变量的两用户测试中，请求次数超过 1 000 000 次。

□ 性能/可扩缩性比较

表 6-1 总结了随着用户数的增加（超过 2 个），各个实现所用的 CPU 时间以及相应的闩定结果。可以看到，随着用户负载的增加，使用较少闩的方案能更好地扩缩。

表 6-1 使用和不使用绑定变量时 CPU 使用情况的比较

用户数	CPU 时间（秒）/耗用时间（分钟）		共享池闩请求		对闩等待的度量（等待数/等待时间（秒））	
	不使用绑定变量	使用绑定变量	不使用绑定变量	使用绑定变量	不使用绑定变量	使用绑定变量
1	26/0.52	4/0.10	563	883	25	232
0/0						
2	74/0.78	11/0.20	1 126	006	50	367
406/1						
3	155/1.13	29/0.37	1 712 280	75 541	2	830/4
4	272/1.50	44/0.45	2 298 179	100 682	9	400/5
5	370/2.03	64/0.62	2 920 219	125 933	13	800/20
6	466/2.58	74/0.72	3 526 704	150 957	30	800/80
17/0						
7	564/3.15	95/0.92	4 172 492	176 085	40	800/154
8	664/3.57	106/1.00	4 734 793	201 351	56	300/240
120/1						
9	747/4.05	117/1.15	5 360 188	230 516	74	600/374
230/1						
10	822/4.42	137/1.30	5 901 981	251 434	60	000/450
354/1						

对我来说，我观察到很有意思的一点，如果 10 个用户使用绑定变量（所以闩请求很少），使用的硬件资源与不使用绑定变量的 2～2.5 个用户（也就是说，过量使用了闩，或者执行了本不需要的处理）所需的硬件资源相当。检查 10 个用户的结果时，可以看到，倘若未使用绑定变量，与使用了绑定变量的方案相比，所需的 CPU 时间是后者的 6 倍，执行时间也是后者的 3.4 倍。随着增加更多的用户，每个用户等待闩所花费的时间就更长。当有 5 个用户时，对闩的等待时间平均为 4 秒/会话，等到有 10 个用户时，平均等待时间就是 45 秒/会话。不过，如果采用了能避免过量使用闩的实现，则用户规模的扩大不会带来不好的影响。

6.3.4 手动锁定和用户定义锁

到此为止，前面主要了解了 Oracle 为我们所加的锁，这些锁定工作对我们来说都是透明的。更新一个表时，Oracle 会为它加一个 TM 锁，以防止其他会话删除这个表（实际上，也会防止其他会话对这个表执行大多数 DDL）。在我们修改的各个块上会加上 TX 锁，这样就能告诉别人哪些数据是“我们的”。数据库采用 DDL 锁来保护对象，这样当我们正在修改这些对象时，别人不会同时对它们进行修改。数据库在内部使用了闩和锁（lock）来保护自己的结构。

接下来，我们来看看如何介入这种锁定活动。有以下选择：

- 通过一条 SQL 语句手动地锁定数据。
- 通过 DBMS_LOCK 包创建我们自己的锁。

在后面的小节中，我们将简要地讨论这样做的目的。

1. 手动锁定

我们可能想使用手动锁定（manual locking），实际上，前面已经见过这样的几种情况了。SELECT...FOR UPDATE 语句就是手动锁定数据的一种主要方法。在前面的例子中，曾经用过这个语句来避免丢失更新问题（也就是一个会话可能覆盖另一个会话所做的修改）。我们已经看到，可以用这种方法来串行访问详细记录，从而执行业务规则（例如，第 1 章中的资源调度程序示例）。

还可以使用 LOCK TABLE 语句手动地锁定数据。这个语句实际上很少使用，因为锁的粒度太大。它只是锁定表，而不是对表中的行锁定。如果你开始修改行，它们也会被正常地“锁定”。所以，这种方法不能节省资源（但在其他 RDBMS 中可以用这个方法节省资源）。如果你在编写一个大批量的更新，它会影响给定表中的大多数行，而且你希望保证没有人能“阻塞”你，就可以使用 LOCK TABLE IN EXCLUSIVE MODE 语句。通过以这种方式锁定表，就能确保你的更新能够执行所有工作，而不会被其他事务所阻塞。不过，有 LOCK TABLE 语句的应用确实很少见。

2. 创建你自己的锁

通过 DBMS_LOCK 包，Oracle 实际上向开发人员公开了它在内部使用的队列锁（enqueue lock）机制。你可能会奇怪，为什么想创建你自己的锁呢？答案通常与应用有关。例如，你可能要使用这个包对 Oracle 外部的一些资源进行串行访问。假设你在使用

UTL_FILE 例程，它允许你写至服务器文件系统上的一个文件。你可能已经开发了一个通用的消息例程，每个应用都能调用这个例程来记录消息。由于这个文件是外部的，Oracle 不会对试图同时修改这个文件的多个用户进行协调。现在，由于有了 DBMS_LOCK 包，在你打开、写入和关闭文件之前，可以采用排他模式请求一个锁（以文件命名），一次只能有一个人向这个文件写消息。这样所有人都会排队。通过利用 DBMS_LOCK 包，等你用完了锁之后能手动地释放这个锁，或者在你提交时自动放弃这个锁，甚至也可以在你登录期间一直保持这个锁。

6.4 小结

这一章介绍的内容很多，可能会让你觉得很难，不时地抓耳挠腮。尽管锁定本身相当直接，但是它的一些副作用却不是这样。关键是你要理解这些锁定问题。例如，倘若没有对外键加索引，Oracle 会使用表锁来保证外键关系，如果你不知道这一点，你的应用就会性能很差。如果你不知道如何查看数据字典来得出谁锁住了什么，可能永远也发现不了到底是怎么回 事。你可能只是认为数据库有时会“挂起”。有时，面对一个看上去无法解决的挂起问题，我只是运行一个查询来检测外键是不是没有索引，并建议对导致问题的外键加上索引，就能很好地解决问题。这种情况太常见了，我想如果每次解决这样一个问题就能得到 1 美元的报酬的话，我肯定会成为一个富翁。

第 7 章 并发与多版本

上一章曾经说过，开发多用户的数据库驱动应用时，最大的难题之一是：一方面要力争最大的并发访问，与此同时还要确保每个用户能以一致的方式读取和修改数据。这一章我们将进一步详细地讨论 Oracle 如何获得多版本读一致性（multi-version read consistency），并说明这对于开发人员来说意味着什么。我还会介绍一个新概念，即写一致性（write consistency），并用这个概念来说明 Oracle 不仅能在提供读一致性的读环境中工作，还能在混合读写环境中工作。

7.1 什么是并发控制？

并发控制（concurrency control）是数据库提供的函数集合，允许多个人同时访问和修改数据。前一章曾经说过，锁（lock）是 Oracle 管理共享数据库资源并发访问并防止并发数据库事务之间“相互干涉”的核心机制之一。总结一下，Oracle 使用了多种锁，包括：

- TX 锁：修改数据的事务在执行期间会获得这种锁。
- TM 锁和 DDL 锁：在你修改一个对象的内容（对于 TM 锁）或对象本身（对应 DDL 锁）时，这些锁可以确保对象的结构不被修改。
- 闩（latch）：这是 Oracle 的内部锁，用来协调对其共享数据结构的访问。

不论是哪一种锁，请求锁时都存在相关的最小开销。TX 锁在性能和基数方面可扩缩性极好。TM 锁和 DDL 锁要尽可能地采用限制最小的模式。闩和队列锁（enqueue）都是轻量级的，而且都很快（在这二者中，队列锁相对“重”一些，不过，它的功能也更丰富）。如果应用设计不当，不必要地过长时间保持锁，而导致数据库中出现阻塞，就会带来问题。如果能很好地设计代码，利用 Oracle 的锁定机制就能建立可扩缩的高度并发的应用。

但是 Oracle 对并发的支持不只是高效的锁定。它还实现了一种多版本（multi-versioning）体系结构（见第 1 章的介绍），这种体系结构提供了一种受控但高度并发的数据访问。多版本是指，Oracle 能同时物化多个版本的数据，这也是 Oracle 提供数据读一致视图的机制（读一致视图即 read-consistent view，是指相对于某个时间点有一致的结果）。多版本有一个很好的副作用，即数据的读取器（reader）绝对不会被数据的写入器（writer）所阻塞。换句话说，写不会阻塞读。这是 Oracle 与其他数据库之间的一个根本区别。在 Oracle 中，如果一个查询只是读取信息，那么永远也不会被阻塞。它不会与其他会话发生死锁，而且不可能得到数据库中根本不存在的答案。

注意 在分布式 2PC（两段提交）的处理期间，在很短的一段时间内 Oracle 不允许读信息。因为这种处理相当少见，而且属于例外情况（只有查询在准备阶段和提交阶段之间开始，而且试图在提交之前读取数据，此时才会存在这个问题），所以我不打算详细介绍这种情况。

默认情况下，Oracle 的读一致性多版本模型应用于语句级（statement level），也就是说，应用于每一个查询；另外还可以应用于事务级（transaction level）。这说明，至少提交到数据库的每一条 SQL 语句都会看到数据库的一个读一致视图，如果你希望数据库的这种读一致视图是事务级的（一组 SQL 语句），这也是可以的。

数据库中事务的基本作用是将数据库从一种一致状态转变为另一种一种状态。ISO SQL

标准指定了多种事务隔离级别（transaction isolation level），这些隔离级别定义了一个事务对其他事务做出的修改有多“敏感”。越是敏感，数据库在应用执行的各个事务之间必须提供的隔离程度就越高。在下一节中我们就看到，Oracle 如何利用多版本体系结构和绝对最小锁定（absolutely minimal locking）来支持 SQL 标准定义的各种隔离级别。

7.2 事务隔离级别

ANSI/ISO SQL 标准定义了 4 种事务隔离级别，对于相同的事务，采用不同的隔离级别分别有不同的结果。也就是说，即使输入相同，而且采用同样的方式来完成同样的工作，也可能得到完全不同的答案，这取决于事务的隔离级别。这些隔离级别是根据 3 个“现象”定义的，以下就是给定隔离级别可能允许或不允许的 3 种现象：

- 脏读（dirty read）：这个词不仅不好听，实际上也确实是贬义的。你能读取未提交的数据，也就是脏数据。只要打开别人正在读写的一个 OS 文件（不论文件中有什么数据），就可以达到脏读的效果。如果允许脏读，将影响数据完整性，另外外键约束会遭到破坏，而且会忽略惟一性约束。
- 不可重复读（nonrepeatable read）：这意味着，如果你在 T1 时间读取某一行，在 T2 时间重新读取这一行时，这一行可能已经有所修改。也许它已经消失，有可能被更新了，等等。
- 幻像读（phantom read）：这说明，如果你在 T1 时间执行一个查询，而在 T2 时间再执行这个查询，此时可能已经向数据库中增加了另外的行，这会影响你的结果。与不可重复读的区别在于：在幻像读中，已经读取的数据不会改变，只是与以前相比，会有更多的数据满足你的查询条件。

注意 ANSI/ISO SQL 标准不只是定义了单个的语句级特征，还定义了事务级特征。在后面几页的介绍中，我们将分析事务级隔离，而不只是语句级隔离。

SQL 隔离级别是根据以下原则定义的，即是否允许上述各个现象。我发现有一点很有意思，SQL 标准并没有强制采用某种特定的锁定机制或硬性规定的特定行为，而只是通过这些现象来描述隔离级别，这就允许多种不同的锁定/并发机制存在（见表 7-1）

表 7-1 ANSI 隔离级别

隔离级别	脏读	不可重复	幻像读
READ UNCOMMITTED	允许	允许	允许
READ COMMITTED		允许	允许
REPEATABLE READ			允许
SERIALIZABLE			

Oracle 明确地支持 READ COMMITTED（读已提交）和 SERIALIZABLE（可串行化）隔离级别，因为标准中定义了这两种隔离级别。不过，这还不是全部。SQL 标准试图建立多种隔离级别，从而允许在各个级别上完成的查询有不同程度的一致性。REPEATABLE

READ（可重复读）也是 SQL 标准定义的一个隔离级别，可以保证由查询得到读一致的（read-consistent）结果。在 SQL 标准的定义中，READ COMMITTED 不能提供一致的结果，而 READ UNCOMMITTED（读未提交）级别用来得到非阻塞读（non-blocking read）。

不过，在 Oracle 中，READ COMMITTED 则有得到读一致查询所需的所有属性。在其他数据库中，READ COMMITTED 查询可能（而且将会）返回数据库中根本不存在的答案（即实际上任何时间点上都没有这样的结果）。另外，Oracle 还秉承了 READ UNCOMMITTED 的“精神”。（有些数据库）提供脏读的目的是为了支持非阻塞读，也就是说，查询不会被同一个数据的更新所阻塞，也不会因为查询而阻塞同一数据的更新。不过，Oracle 不需要脏读来达到这个目的，而且也不支持脏读。但在其他数据库中必须实现脏读来提供非阻塞读。

除了 4 个已定义的 SQL 隔离级别外，Oracle 还提供了另外一个级别，称为 READ ONLY（只读）。READ ONLY 事务相对于无法在 SQL 中完成任何修改的 REPEATABLE READ 或 SERIALIZABLE 事务。如果事务使用 READ ONLY 隔离级别，只能看到事务开始那一刻提交的修改，但是插入、更新和删除不允许采用这种模式（其他会话可以更新数据，但是 READ ONLY 事务不行）。如果使用这种模式，可以得到 REPEATABLE READ 和 SERIALIZABLE 级别的隔离性。

下面再来讨论多版本及读一致性如何用于实现隔离机制，并说明不支持多版本的数据库怎样得到同样的结果。如果你曾经用过其他数据库，自认为很清楚隔离级别是如何工作的，就会发现这一部分的介绍对你很有帮助。还有一点很有意思，你会看到，尽管 ANSI/ISO SQL 标准原本力图消除数据库之间的差别，而实际上却允许各个数据库有不同的具体做法。这个标准尽管非常详细，但是可以采用完全不同的方式来实现。

7.2.1 READ UNCOMMITTED

READ UNCOMMITTED 隔离级别允许脏读。Oracle 没有利用脏读，甚至不允许脏读。READ UNCOMMITTED 隔离级别的根本目标是提供一个基于标准的定义以支持非阻塞读。我们已经看到了，Oracle 会默认地提供非阻塞读。在数据库中很难阻塞一个 SELECT 查询（如前所述，只是在分布式事务中对此有一个特殊的例外情况）。每个查询都以一种读一致的方式执行，而不论是 SELECT、INSERT、UPDATE、MERGE，还是 DELETE。这里把 UPDATE 语句称为查询可能很可笑，不过，它确实是一个查询。UPDATE 语句有两个部分：一个是 WHERE 子句定义的读部分，另一个是 SET 子句定义的写部分。UPDATE 语句会对数据库进行读写，就像所有 DML 语句一样。对此只有一个例外：使用 VALUES 子句的单行 INSET 是一个特例，因为这种语句没有读部分，而只有写部分。

在第 1 章中，我们通过一个简单的单表查询说明了 Oracle 如何得到读一致性，那个例子中获得了在游标打开后所删除的行。下面我们再分析一个实际的例子，来看看使用多版本的 Oracle 中会发生什么，并介绍在其他多种数据库中又会怎么样。

先来看一个简单的表和查询（还是与以前一样）：

```
create table accounts

( account_number number primary key,
```

```
  account_balance number not null

);

select sum(account_balance) from accounts;
```

查询开始前，数据如表 7-2 所示。

表 7-2　修改前的 ACCOUNTS 表

行	帐号	账户金额
1	123	$500.00
2	456	$240.25
. . .	. . .	. . .
342,023	987	$100.00

下面，select 语句开始执行，读取第 1 行、第 2 行等。在查询中的某一点上，一个事务将$400.00 从账户 123 转到账户 987。这个事务完成了两个更新，但是并没有提交。现在的数据表如表 7-3 所示。

所以，其中两行已经锁定。如果有人试图更新这两行，该用户就会被阻塞。到目前为止，我们所看到的在所有数据库上基本上都是一致的。只是如果有人要查询锁定的数据，此时不同的数据库就会表现出不同的行为。

表 7-3　修改期间的 ACCOUNTS 表

行	帐号	账户金额	是否?
1	123	($500.00) changed to $100.00	X
2	456	$240.25	
. . .	. . .	. . .	
342,023	987	($100.00) changed to $500.00	X

如果我们执行的查询要访问某一块，其中包含表“最后”已锁定的行（第 342,023 行），则会注意到，这一行中的数据在开始执行之后有所改变。为了提供一个一致（正确）的答案，Oracle 在这个时刻会创建该块的一个副本，其中包含查询开始时行的“本来面目”。也就是说，它会读取值$100.00，这就是查询开始时该行的值。这样一来，Oracle 就有效地绕过了已修改的数据，它没有读修改后的值，而是从 undo 段（也称为回滚（rollback）段，详细内容见第 9 章）重新建立原数据。因此可以返回一致而且正确的答案，而无需等待事务提交。

再来看允许脏读的数据库，这些数据库只会返回读取那一刻在账户 987 中看到的值，在这里就是$500.00。这个查询会把转账的$400 重复统计两次。因此，它不仅会返回错误的答案，而且会返回表中根本不存在的一个总计（任何时间点都没有这样一个总计）。在多用户数据库中，脏读可能是一个危险的特性，就我个人来看，我实在看不出脏读有什么用处。例如，如果不是转账，而是事务要向账户 987 中存入$400.00 会怎么样呢？脏读会计入$400.00，得到“正确”的答案，是这样吗？假设未提交的事务回滚了，那么我们就计入了数据库中并没有的$400.00。

这里的关键是，脏读不是一个特性：而是一个缺点。Oracle 中根本不需要脏读。Oracle 完全可以得到脏读的所有好处（即无阻塞），而不会带来任何不正确的结果。

7.2.2 READ COMMITTED

READ COMMITTED 隔离级别是指，事务只能读取数据库中已经提交的数据。这里没有脏读，不过可能有不可重复读（也就是说，在同一个事务中重复读取同一行可能返回不同的答案）和幻像读（与事务早期相比，查询不光能看到已经提交的行，还可以看到新插入的行）。在数据库应用中，READ COMMITTED 可能是最常用的隔离级别了，这也是 Oracle 数据库的默认模式，很少看到使用其他的隔离级别。

不过，得到 READ COMMITTED 隔离并不像听起来那么简单。看看表 7-1，你可能认为这看上去很直接。显然，根据先前的规则，在使用 READ COMMITTED 隔离级别的数据库中执行的查询肯定就会有相同的表现，真的是这样吗？这是不对的。如果在一条语句中查询了多行，除了 Oracle 外，在几乎所有其他的数据库中，READ COMMITTED 隔离都可能“退化”得像脏读一样，这取决于具体的实现。

在 Oracle 中，由于使用多版本和读一致查询，无论是使用 READ COMMITTED 还是使用 READ UNCOMMITTED，从 ACCOUNTS 查询得到的答案总是一样的。Oracle 会按查询开始时数据的样子对已修改的数据进行重建，恢复其“本来面目”，因此会返回数据库在查询开始时的答案。

下面来看看其他数据库，如果采用 READ COMMITTED 模式，前面的例子又会怎样。你可能看到答案很让人吃惊。我们从表所述的那个时间点开始：

- 现在正处在表的中间。已经读取并合计了前 N 行。
- 另一个事务将$400.00 从账户 123 转到账户 987。
- 事务还没有提交，所以包含账户 123 和 987 信息的行被锁定。

我们知道 Oracle 中到达账户 987 那一行时会发生什么，它会绕过已修改的数据，发现它原本是$100.00，然后完成工作。表 7-4 显示了其他数据库（非 Oracle）采用默认的 READ COMMITTED 模式运行时可能得到的答案。

表 7-4　非 Oracle 数据库使用 READ COMMITTED 隔离级别时的时间表

时间	查询	转账事务
T1	读取第 1 行。到目前为止 Sum=$500.00	

T2　读取第 2 行。到目前为止 Sum=$740.25

T3　更新第 1 行，并对第 1 行加一个排他锁，防止出现其他更新和读取。第 1 行现在的值为$100.00

T4　读取第 N 行。Sum=…

T5　更新第 342,023 行，对这一行加一个排他锁。现在第 342,023 行的值是$500.00

T6　试图读取第 342,023 行，发现这一行被锁定。会话会阻塞，并等待这一行重新可用。对这个查询的所有处理都停止

T7　提交事务

T8　读取第 342,023 行，发现值为$500.00，提供一个最终答案，可惜这里把$400.00 重复计入了两次

首先要注意到，在这个数据库中，到达账户 987 时， 我们的查询会被阻塞。这个会话必须等待这一行，真正持有排它锁的事务提交。这是因为这个原因，所以很多人养成一种坏习惯，在执行每条语句后都立即提交，而 不是处理一个合理的事务，其中包括将数据库从一种一致状态转变为另一种一致状态所需的所有语句。在大多数其他数据库中，更新都会干涉读取。在这种情况下， 还有一个不好的消息，我们不仅会让用户等待，而且他们苦苦等待的最后结果居然还是不正确的。我们会到数据库中从来没有过的答案，这就像脏读一样，但是与脏 读不同的是，这一次还需要用户等待这个错误的答案。在下一节中，我们将了解这些数据库要得到读一致的正确结果需要做些什么。

从这里可以得到一个重要的教训，不同的数据库尽管采用相同的、显然安全的隔离级别，而且在完全相同的环境中执行，仍有可能返回完全不同的答案。要知道的重要的一点是，在 Oracle 中，非阻塞读并没有以答案不正确作为代价。有时，鱼和熊掌可以兼得。

7.2.3 REPEATABLE READ

REPEATABLE READ 的目标是提供这样一个隔离级别，它不仅能给出一致的正确答案，还能避免丢失更新。我们会分别给出例子，来看看在 Oracle 中为达到这些目标需要做些什么，而在其他系统中又会发生什么。

1. 得到一致的答案

如果隔离级别是 REPEATABLE READ，从给定查询得到的结果相对于某个时间点来说应该是一致的。大多数数据库（不包括 Oracle）都通过使用低级的共享读锁来实现可重复读。共享读锁会防止其他会话修改我们已经读取的数据。当然，这会降低并发性。Oracle 则采用

了更具并发性的多版本模型来提供读一致的答案。

在 Oracle 中，通过使用多版本，得到的答案相对于查询开始执行那个时间点是一致的。在其他数据库中，通过使用共享读锁，可以得到相对于查询完成那个时间点一致的答案，也就是说，查询结果相对于我们得到的答案的那一刻是一致的（稍后会对这个问题做更多的说明）。

在一个采用共享读锁来提供可重复读的系统中，可以观察到，查询处理表中的行时，这些行都会锁定。所以，仍使用前面的例子，也就是我们的查询要读取 ACCOUNTS 表，那么每一行上都会有共享读锁，如表 7-5 所示。

表 7-5　非 Oracle 数据库使用 READ REPEATABLE 隔离级别时的时间表 1

时间	查询	转账事务
T1	读取第 1 行。到目前为止 Sum=$500.00。	
	块 1 上有一个共享读锁	
T2	读取第 2 行。到目前为止 Sum=$740.25。	
	块 2 上有一个共享读锁	
T3		试图更新第 1 行，但是被阻塞。这个事务被挂起，直至可以得到一个排他锁
T4	读取第 N 行。Sum=…	
T5	读取第 342,023 行，看到$100.00，提供最后的答案	
T6	提交事务	
T7		更新第 1 行，并对这一块加一个排他锁。现在第 1 行有$100.00
T8		更新第 342,023 行，对这一块加一个排它锁。第 342,023 行现在的值为$500.00。提交事务

从表 7-5 可 以看出，现在我们得到了正确的答案，但是这是有代价的：需要物理地阻塞一个事务，并且顺序执行两个事务。这是使用共享读锁来得到一致答案的副作用之一：数据的读取器会阻塞数据的写入器。不仅如此，在这些系统这，数据的写入器还会阻塞数据读取器。想像一下，如果实际生活中自动柜员机（ATM）这样工作会是什么情况。

由此可以看到，共享读锁会妨碍并发性，而且还导致有欺骗性的错误。在表 7-6 中，我们先从原来的表开始，不过这一次的目标是把$50.00 从账户 987 转账到账户 123。

表 7-6　非 Oracle 数据库使用 READ REPEATABLE 隔离级别时的时间表 2

时间	查询	转账事务
T1	读取第 1 行。到目前为止 Sum=$500.00。	
	块 1 上有一个共享读锁	
T2	读取第 2 行。到目前为止 Sum=$740.25。	
	块 2 上有一个共享读锁	
T3		更新第 342,023 行，对块 342,023 加一个排他锁，防止出现其他更新和共享读锁。现在这一行的值为$50.00
T4	读取第 N 行。Sum=…	
T5		试图更新第 1 行，但是被阻塞。这个事务被挂起，直至可以得到一个排他锁
T6	试图读取第 342,023 行，但是做不到，	
	因为该行已经有一个排他锁	

我 们陷入了经典的死锁条件。我们的查询拥有更新需要的资源，而更新也持有着查询所需的资源。查询与更新事务陷入死锁。要把其中一个作为牺牲品，将其中止。这 样说来，我们可能会花大量的时间和资源，而最终只是会失败并回滚。这是共享读锁的另一个副作用：数据的读取器和写入器可能而且经常相互死锁。

可以看到，Oracle 中可以得到语句级的读一致性，而不会带来读阻塞写的现象，也不会导致死锁。Oracle 从不使用共享读锁，从来不会。Oracle 选择了多版本机制，尽管更难实现，但绝对更具并发性。

2.　丢失更新：另一个可移植性问题

在采用共享读锁的数据库中，REPEATABLE READ 的一个常见用途是防止丢失更新。

注意　丢失更新问题的丢失更新检测及解决方案在第 6 章已经讨论过。

在一个采用共享读锁（而不是多版本）的数据库中，如果启用了 REPEATABLE READ，则不会发生丢失更新错误。这些数据库中之所以不会发生丢失更新，原因是：这样选择数据

就会在上面加一个锁，数据一旦由一个事务读取，就不能被任何其他事务修改。如此说来，如果你的应用认为 REPEATABLE READ 就意味着“丢失更新不可能发生”，等你把应用移植到一个没有使用共享读锁作为底层并发控制机制的数据库时，就会痛苦地发现与你预想的并不一样。

尽管听上去使用共享读锁好像不错，但你必须记住，如果读取数据时在所有数据上都加共享读锁，这肯定会严重地限制并发读和修改。所以，尽管在这些数据库中这个隔离级别可以防止丢失更新，但是与此同时，也使得完成并发操作的能力化为乌有！对于这些数据库，你无法鱼和熊掌兼得。

7.2.4 SEAIALIZABLE

一般认为这是最受限的隔离级别，但是它也提供了最高程度的隔离性。SERIALIZABLE 事务在一个环境中操作时，就好像没有别的用户在修改数据库中的数据一样。我们读取的所有行在重新读取时都肯定完全一样，所执行的查询在整个事务期间也总能返回相同的结果。例如，如果执行以下查询：

```
Select * from T;

Begin dbms_lock.sleep( 60*60*24 ); end;

Select * from T;
```

从 T 返回的答案总是相同的，就算是我们睡眠了 24 小时也一样（或者会得到一个 ORA-1555:snapshot too old 错误，这将在第 8 章讨论）。这个隔离级别可以确保这两个查询总会返回相同的结果。其他事务的副作用（修改）对查询是不可见的，而不论这个查询运行了多长时间。

Oracle 中是这样实现 SERIALIZABLE 事务的：原本通常在语句级得到的读一致性现在可以扩展到事务级。

注意 前面提到过，Oracle 中还有一种称为 READ ONLY 的隔离级别。它有着 SERIALIZABLE 隔离级别的所有性质，另外还会限制修改。需要指出，SYS 用户（或作为 SYSDBA 连接的用户）不能有 READ ONLY 或 SERIALIZABLE 事务。在这方面，SYS 很特殊。

结果并非相对于语句开始的那个时间点一致，而是在事务开始的那一刻就固定了。换句话说，Oracle 使用回滚段按事务开始时数据的原样来重建数据，而不是按语句开始时的样子重建。

这里有一点很深奥——在你问问题之前，数据库就已经知道了你要问的问题的答案。

这种隔离性是有代价的，可能会得到以下错误：

```
ERROR at line 1:

ORA-08177: can't serialize access for this transaction
```

只要你试图更新某一行，而这一行自事务开始后已经修改，你就会得到这个消息。

注意 Oracle试图完全在行级得到这种隔离性，但是即使你想修改的行尚未被别人修改后，也可能得到一个 ORA-01877 错误。发生 ORA-01877 错误的原因可能是：包含这一行的块上有其他行正在被修改。

Oracle 采用了一种乐观的方法来实现串行化，它认为你的事务想要更新的数据不会被其他事务所更新，而且把宝押在这上面。一般确实是这样的，所以说通常这个宝是押对了，特别是在事务执行得很快的 OLTP 型系统中。尽管在其他系统中这个隔离级别通常会降低并发性，但是在 Oracle 中，倘若你的事务在执行期间没有别人更新你的数据，则能提供同等程度的并发性，就好像没有 SERIALIZABLE 事务一样。另一方面，这也是有缺点的，如果宝押错了，你就会得到 ORA_08177 错误。不过，可以再想想看，冒这个险还是值得的。如果你确实要更新信息，就应该使用第 1 章所述的 SELECT … FOR UPDATE，这会实现串行访问。所以，如果使用 SERIALIZABLE 隔离级别，只要保证以下几点就能很有成效：

- ☐ 一般没有其他人修改相同的数据
- ☐ 需要事务级读一致性
- ☐ 事务都很短（这有助于保证第一点）

Oracle发现这种方法的可扩缩性很好，足以运行其所有TPC-C（这是一个行业标准OLTP基准：有关详细内容请见www.tpc.org）。在许多其他的实现中，你会发现这种隔离性都是利用共享读锁达到的，相应地会带来死锁和阻塞。而在Oracle中，没有任何阻塞，但是如果其他会话修改了我们也想修改的数据，则会得到ORA-08177 错误。不过，与其他系统中得到死锁和阻塞相比，我们得到的错误要少得多。

但是，凡事总有个但是，首先必须了解存在这样一些不同的隔离级别，而且要清楚它们带来的影响。要记住，如果隔离级别设置为 SERIALIZABEL，事务开始之后，你不会看到数据库中做出任何修改，直到提交事务为止。如果应用试图保证其数据完整性约束，如第 1 章介绍的资源调度程序，就必须在这方面特别小心。如果你还记得，第 1 章中的问题是：我们无法保证一个多用户系统中的完整性约束，因为我们看不到其他未提交会话做出的修改。通过使用 SERIALIZABLE。这些未提交的修改还是看不到，但是同样也看不到事务开始后执行的已提交的修改！

还有最后一点要注意，SERIALIZABLE 并不意味着用户执行的所有事务都表现得好像是以一种串行化方式一个接一个地执行。SERIALIZABLE 不代表事务有某种串行顺序并且总能得到相同的结果。尽管按照 SQL 标准来说，这种模式不允许前面所述的几种现象，但不能保证事务总按串行方式顺序执行。最后这个概念经常被误解，不过只需一个小小的演示例子就能澄清。下表表示了在一段时间内完成工作的两个会话。数据库表 A 和 B 开始时为空，并创建如下：

```
ops$tkyte@ORA10G> create table a ( x int );

Table created.
```

```
ops$tkyte@ORA10G> create table b ( x int );

Table created.
```

现在可以得到表 7-7 所示的一系列事件。

表 7-7　SERIALIZABLE 事务例子

时间	会话 1 执行	会话 2 执行
T1	Alter session set isolation_level= serializable;	
T2		Alter session set isolation_level= serializable;
T3	Insert into a select count(*) from b;	
T4		Insert into b select count(*) from a;
T5	Commit;	
T6		Commit;

现在，一起都完成后，表 A 和 B 中都有一个值为 0 的行。如果事务有某种“串行”顺序，就不可能得到两个都包含 0 值的表。如果会话 1 在会话 2 之前执行，表 B 就会有一个值为 1 的行。如果会话 2 在会话 1 之前执行，那么表 A 则有一个值为 1 的行。不过，按照这里的执行方式，两个表中的行都有值 0，不论是哪个事务，执行时就好像是此时数据库中只有它一个事务一样。不管会话 1 查询多少次表 B，计数（count）都是对 T1 时间数据库中已提交记录的计数。类似地，不论会话 2 查询多少次表 A，都会得到与 T2 时间相同的计数。

7.2.5 READ ONLY

READ ONLY 事务与 SERIALIZABLE 事务很相似，惟一的区别是 READ ONLY 事务不允许修改，因此不会遭遇 ORA-08177 错误。READ ONLY 事务的目的是支持报告需求，即相对于某个时间点，报告的内容应该是一致的。在其他系统中，为此要使用 REPEATABLE READ，这就要承受共享读锁的相关影响。在 Oracle 中，则可以使用 READ ONLY 事务。采用这种模式，如果一个报告使用 50 条 SELECT 语句来收集数据，所生成的结果相对于某个时间点就是一致的，即事务开始的那个时间点。你可以做到这一点，而无需在任何地方锁定数据。

为达到这个目标，就像对单语句一样，也使用了同样的多版本机制。会根据需要从回滚段重新创建数据，并提供报告开始时数据的原样。不过，READ ONLY 事务也不是没有问

题。在 SERIALIZABLE 事务中你可能会遇到 ORA-08177 错误，而在 READ ONLY 事务中可能会看到 ORA-1555：snapshot too old 错误。如果系统上有人正在修改你读取的数据，就会发生这种情况。对这个信息所做的修改（undo 信息）将记录在回滚段中。但是回滚段以一种循环方式使用，这与重做日志非常相似。报告运行的时间越长，重建数据所需的 undo 信息就越有可能已经不在那里了。回滚段会回绕，你需要的那部分回滚段可能已经被另外某个事务占用了。此时，就会得到 ORA-1555 错误，只能从头再来。

对于这个棘手的问题，惟一的解决方案就是为系统适当地确定回滚段的大小。我多次看到，人们为了节省几 MB 的磁盘空间而把回滚段设置得尽可能小（这些人的想法是："为什么要在我不需要的东西上'浪费'空间呢？）。问题在于，回滚段是完成数据库工作的一个关键组件，除非有合适的大小，否则就会遭遇这个错误。在使用 Oracle 6、7 和 8 的 16 年间，我可以自豪地说，除了测试或开发系统之外，我从来没有在生产系统中遇到过 ORA-1555 错误。如果真的遇到了这个错误，这就说明你没有正确地设置回滚段的大小，需要适当地加以修正。我们将在第 9 章再来讨论这个问题。

7.3 多版本读一致性的含义

到此为止，我们已经看到了多版本机制如何提供非阻塞读，我强调了多版本是一个好东西，它不仅能提供一致（正确）的答案，还有高度的并发性。能还有什么不妥吗？这么说吧，除非你了解到存在多版本机制，也知道多版本的含义，否则事务完成就有可能完成得不正确。应该记得，在第 1 章的调度资源例子中，我们必须采用某种手动锁定技术（通过 SELECT FOR UPDATE 对 SCHEDULES 表中的资源调度修改进行串行化）。但是它会在其他方面带来影响吗？答案是当然会。下面几节将具体谈谈这些问题。

7.3.1 一种会失败的常用数据仓库技术

我看到，许多人都喜欢用这样一种常用的数据仓库技术：

（1） 他们使用一个触发器维护源表中的一个 LAST_UPDATED 列，这与上一章的 6.2.3 节中讨论的方法很相似。

（2） 最初要填充数据仓库表时，他们要记住当前的时间，为此会选择源系统上的 SYSDATE。例如，假设现在刚好是上午 9:00。

（3） 然后他们从事务系统中拉（pull）出所有行，这是一个完整的 SELECT * FROM TABLE 查询，可以得到最初填充的数据仓库。

（4） 要刷新这个数据仓库，他们要再次记住现在的时间。例如，假设已经过去了 1 个小时，现在源系统上的时间就是 10:00.他们会记住这一点。然后拉出自上午 9:00（也就是第一次拉出数据之前的那个时刻）以来修改过的所有记录，并把这些修改合并到数据仓库中。

注意 这种技术可能会在两次连续的刷新中将相同的记录"拉出"两次。由于时钟的粒度所致，这是不可避免的。MERGE 操作不会受此影响（即更新数据仓库中现有的记录，或插入一个新记录）。

他 们相信，现在数据仓库中有了自第一次执行拉出操作以来所修改的所有记录。他们确实可能有所有记录，但是也有可能不是这样。对于其他采用锁定系统的数据库来 说，这种技术确实能很好地工作，在这些数据库中读会被写阻塞，反之写也会被读阻塞。但是在默

认支持非阻塞读的系统中，这个逻辑是有问题的。

要看这个例子有什么问题，只需假设上午 9:00 至少有一个打开的未提交事务。例如，假设在上午 8:59:30 时，这个事务已经更新了表中我们想复制的一行。在上午 9:00， 开始拉数据时，会读取这个表中的数据，但是我们看不到对这一行做的修改；只能看到它的最后一个已提交的版本。如果在查询中到达这一行时它已经锁定，我们就 会绕过这个锁。如果在到达它之前事务已经提交，我们还是会绕过它读取查询开始时的数据，因为读一致性只允许我们读取语句开始时数据库中已经提交的数据。在 上午 9:00 第一次拉数据期间我们读不到这一行的新版本，在上午 10:00 刷新期间也读不到这个修改过的行。为什么呢？上午 10:00 的刷新只会拉出自那天早上上午 9:00 以后修改的记录，但是这个记录是在上午 8:59:30 时修改的，我们永远也拉不到这个已修改的记录。

在许多其他的数据库中，其中读会被写阻塞，可以完成已提交但不一致的读，那么这个刷新过程就能很好地工作。如果上午 9:00（第一次拉数据时）我们到达这一行，它已经上锁，我们就会阻塞，等待这一行可用，然后读取已提交的版本。如果这一行未锁定，那么只需读取就行，因为它们都是已提交的。

那么，这是否意味着前面的逻辑就根本不能用呢？也不是，这只是说明我们需要用稍微不同的方式来得到“现在”的时间。应该查询 V$TRANSACTION，找出最早的当前时间是什么，以及这个视图中 START_TIME 列记录的时间。我们需要拉出自最老事务开始时间（如果没有活动事务，则取当前的 SYSDATE 值）以来经过修改的所有记录：

```
select nvl( min(to_date(start_time,'mm/dd/rr hh24:mi:ss')),sysdate)

from v$transaction;
```

在这个例子中，这就是上午 8:59:30，即修改这一行的事务开始的那个时间。我们在上午 10:00 刷新数据时，会拉出自那个时间以来发生的所有修改，把这些修改合并到数据仓库中，这就能得到需要的所有东西。

7.3.2 解释热表上超出期望的 I/O

在另外一种情况下很有必要了解读一致性和多版本，这就是生产环境中在一个大负载条件下，一个查询使用的 I/O 比你在测试或开发系统时观察到的 I/O 要多得多，而你无法解释这一现象。你查看查询执行的 I/O 时，注意到它比你在开发系统中看到的 I/O 次数要多得多，多得简直不可想像。然后，你再在测试环境中恢复这个实例，却发现 I/O 又 降下来了。但是到了生产环境中，它又变得非常高（但是好像还有些变化：有时高，有时低，有时则处于中间）。可以看到，造成这种现象的原因是：在你测试系统 中，由于它是独立的，所以不必撤销事务修改。不过，在生产系统中，读一个给定的块时，可能必须撤销（回滚）多个事务所做的修改，而且每个回滚都可能涉及 I/O 来获取 undo 信息并应用于系统。

可能只是要查询一个表，但是这个表上发生了多个并发修改，因此你看到 Oracle 正在读 undo 段，从而将块恢复到查询开始时的样子。通过一个会话就能很容易地看到由此带来的结果，从而了解到底发生了什么。我们先从一个非常小的表开始：

```
ops$tkyte@ORA10GR1> create table t ( x int );

Table created.
```

```
ops$tkyte@ORA10GR1> insert into t values ( 1 );

1 row created.



ops$tkyte@ORA10GR1> exec dbms_stats.gather_table_stats( user, 'T' );

PL/SQL procedure successfully completed.



ops$tkyte@ORA10GR1> select * from t;

X

----------

1
```

下面，将会话设置为使用 SERIALIZABLE 隔离级别，这样无论在会话中运行多少次查询，都将得到事务开始时刻的查询结果：

```
ops$tkyte@ORA10GR1> alter session set isolation_level=serializable;

Session altered.
```

下面查询这个小表，并观察执行的 I/O 次数：

```
ops$tkyte@ORA10GR1> set autotrace on statistics

ops$tkyte@ORA10GR1> select * from t;

X

----------

1



Statistics

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

```
0 recursive calls

0 db block gets

3 consistent gets

...
```

由此可见，完成这个查询用了 3 个 I/O（一致获取，consistent get）。在另一个会话中，我们将反复修改这个表：

```
ops$tkyte@ORA10GR1> begin

  2 for i in 1 .. 10000

  3 loop

  4 update t set x = x+1;

  5 commit;

  6 end loop;

  7 end;

  8 /

PL/SQL procedure successfully completed
```

再返回到前面的 SERIALIZABLE 会话，重新运行同样的查询：

```
ops$tkyte@ORA10GR1> select * from t;

X

---------

1

Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets
```

```
10004 consistent gets

...
```

这一次执行了 10,004 次 I/O，简直有天壤之别。那么，所有这些 I/O 是从哪里来的呢？这是因为 Oracle 回滚了对该数据库块的修改。在运行第二个查询时，Oracle 知道查询获取和处理的所有块都必须针对事务开始的那个时刻。到达缓冲区缓存时，我们发现，缓存中的块“太新了”，另一个会话已经把这个块修改了 10,000 次。查询无法看到这些修改，所以它开始查找 undo 信息，并撤销上一次所做的修改。它发现这个回滚块还是太新了，然后再对它做一次回滚。这个工作会复发进行，直至最后发现事务开始时的那个版本（即事务开始时数据库中的已提交块）。这才是我们可以使用的块，而且我们用的就是这个块。

注意 需要指出，有一点很有意思，如果你想再次运行 SELECT * FROM T，可能会看到 I/O 再次下降到 3；不再是 10,004。为什么呢？Oracle 能把同一个块的多个版本保存在缓冲区缓存中。你撤销对这个块的修改时，也就把相应的版本留在缓存中了，这样以后执行查询时就可以直接访问。

那么，是不是只在使用 SERIALIZABLE 隔离级别时才会遇到这个问题呢？不，绝对不是。可以考虑一个运行 5 分钟的查询。在查询运行的这 5 分钟期间，它从缓冲区缓存获取块。每次从缓冲区缓存获取一个块时，都会完成这样一个检查：“这个块是不是太新了？如果是，就将其回滚。”另外，要记住，查询运行的时间越长，它需要的块在此期间被修改的可能性就越大。

现在，数据库希望进行这个检查(也就是说，查看块是不是“太新”，并相应地回滚修改)。正是由于这个原因，缓冲区缓存实际上可能在内存中包含同一个块的多个版本。通过这种方式，很有可能你需要的版本就在缓存中，已经准备好，正等着你使用，而无需使用 undo 信息进行物化。请看以下查询：

```
select file#, block#, count(*)

from v$bh

group by file#, block#

having count(*) > 3

order by 3

/
```

可以用这个查询查看这些块。一般而言，你会发现在任何时间点上缓存中一个块的版本大约不超过 6 个，但是这些版本可以由需要它们的任何查询使用。

通常就是这些小的“热表”会因为读一致性遭遇 I/O 膨胀问题。另外，如果查询需要针对易失表长时间运行，也经常受到这个问题的影响。运行的时间越长，“它们也就会运行得越久”，因为过一段时间，它们可能必须完成更多的工作才能从缓冲区缓存中获取一个块。

7.4 写一致性

到此为止，我们语句了解了读一致性：Oracle 可以使用 undo 信息来提供非阻塞的查询和一致（正确）的读。我们了解到，查询时，Oracle 会从缓冲区缓存中读出块，它能保证这个块版本足够“旧”，能够被该查询看到。

但是，这又带来了以下的问题：写/修改会怎么样呢？如果运行以下 UPDATE 语句，会发生什么：

```
Update t set x = 2 where y = 5;
```

在该语句运行时，有人将这条语句已经读取的一行从 Y=5 更新为 Y=6，并提交，如果是这样会发生什么情况？也就是说，在 UPDATE 开始时，某一行有值 Y=5。在 UPDATE 使用一致读来读取表时，它看到了 UPDATE 开始时这一行是 Y=5。但是，现在 Y 的当前值是 6，不再是 5 了，在更新 X 的值之前，Oracle 会查看 Y 是否还是 5。现在会发生什么呢？这会对更新有什么影响？

显然，我们不能修改块的老版本，修改一行时，必须修改该块的当前版本。另外，Oracle 无法简单地跳过这一行，因为这将是不一致读，而且是不可预测的。在这种情况下，我们发现 Oracle 会从头重新开始写修改。

7.4.1 一致读和当前读

Oracle 处理修改语句时会完成两类块获取。它会执行：

- 一致读（Consistent read）：“发现”要修改的行时，所完成的获取就是一致读。
- 当前 读（Current read）：得到块来实际更新所要修改的行时，所完成的获取就是当前读。

使用 TKPROF 可以很容易地看到这一点。请考虑以下这个很小的单行例子，它从先前的表 T 读取和更新一行：

```
ops$tkyte@ORA10GR1> alter session set sql_trace=true;

Session altered.

ops$tkyte@ORA10GR1> select * from t;

X

----------

10001

ops$tkyte@ORA10G> update t t1 set x = x+1;

1 row updated.
```

```
ops$tkyte@ORA10G> update t t2 set x = x+1;

1 row updated.
```

运行 TKPROF 并查看结果时，可以看到如下的结果（需要注意，我去掉了报告中的 ELAPSED、CPU 和 DISK 列）：

```
select * from t

call     count       query      current        rows
-------  ------  ----------  ----------  ----------
Parse        1          0           0           0
Execute      1          0           0           0
Fetch        2          3           0           1
-------  ------  ----------  ----------  ----------
total        4          3           0           1

update t t1 set x = x+1

call     count       query      current        rows
-------  ------  ----------  ----------  ----------
Parse        1          0           0           0
Execute      1          3           3           1
Fetch        0          0           0           0
-------  ------  ----------  ----------  ----------
total        2          3           3           1

update t t2 set x = x+1

call     count       query      current        rows
-------  ------  ----------  ----------  ----------
Parse        1          0           0           0
```

Execute	1	3	1	1
Fetch	0	0	0	0
-------	------	------	----------	----------
total	2	3	1	1

因此，在一个正常的查询中，我们会遇到 3 个查询模式获取（一致模式获取，query（consistent）mode get）。在第一个 UPDATE 期间，会遇到同样的 3 个当前模式获取（current mode get）。完成这些当前模式获取是为了分别得到现在的表块（table block），也就是包含待修改行的块；得到一个 undo 段块（undo segment block）来开始事务；以及一个 undo 块（undo block）。第二个更新只有一个当前模式获取，因为我们不必再次完成撤销工作，只是要利用一个当前获取来得到包含待更新行的块。既然存在当前模式获取，这就什么发生了某种修改。在 Oracle 用新信息修改一个块之前，它必须得到这个块的当前副本。

那么，读一致性对修改有什么影响呢？这么说吧，想像一下你正在对某个数据库表执行以下 UPDATE 语句：

```
Update t set x = x+1 where y = 5;
```

我们知道，查询的 WHERE Y=5 部分（即读一致阶段）会使用一个一致读来处理（TKPROF 报告中的查询模式获取）。这个语句开始执行时表中已提交的 WHERE Y=5 记录集就是它将看到的记录（假设使用 READ COMMITED 隔离级别，如果隔离级别是 SERIALIZABLE，所看到的则是事务开始是存在的 WHERE Y=5 记录集）。这说明，如果 UPDATE 语句从开始到结束要花 5 分钟来进行处理，而有人在此期间向表中增加并提交了一个新记录，其 Y 列值为 5，那么 UPDATE 看不到这个记录，因为一致读是看不到新记录的。这在预料之中，也是正常的。但问题是，如果两个会话按顺序执行以下语句会发生什么情况呢？

```
Update t set y = 10 where y = 5;

Update t Set x = x+1 Where y = 5;
```

表 7-8 展示了这个时间表。

表 7-8　更新序列

时间	会话 1	会话 2	注释
T1	Update t set y = 10 where y = 5;		这会更新与条件匹配的一行
T2		Update t	使用一致读，这会找到会话 1 修改的记录，

但是无法更新这个记录，因为会话 1 已经

将其阻塞。会话 2 将被阻塞，并等待这一行可用

set x = x+1

where y = 5;

T3 Commit; 这会解放会话 2；会话 2 不再阻塞。他终于可以在

包含这一行（会话 1 开始更新时 Y 等于 5 的那一行）

的块上完成当前读

因此开始 UPDATE 时 Y=5 的记录不再是 Y=5 了。UPDATE 的一致读部分指出：“你想更新这个记录，因为我们开始时 Y 是 5”，但是根据块的当前版本，你会这样想：”噢，不行，我不能更新这一行，因为 Y 不再是 5 了，这可能不对。“

如 果我们此时只是跳过这个记录，并将其忽略，就会有一个不确定的更新。这可能会破坏数据一致性和完整性。更新的结果（即修改了多少行，以及修改了哪些行）将 取决于以何种顺序命中（找到）表中的行以及此时刚好在做什么活动。在两个不同的数据库中，取同样的一个行集，每个数据库都以相同的顺序运行事务，可能会观 察到不同的结果，这只是因为这些行在磁盘上的位置不同。

在这种情况下，Oracle 会选择重启动更新。如果开始时 Y=5 的行现在包含值 Y=10，Oracle 会悄悄地回滚更新，并重启动（假设使用的是 READ COMMITTED 隔离级别）。如果你使用了 SERIALIZABLE 隔离级别，此时这个事务就会收到一个 ORA-08177： can’t serialize access 错误。采用 READ COMMITTED 模式，事务回滚你的更新后，数据库会重启动更新（也就是说，修改更新相关的时间点），而且它并非重新更新数据，而是进入 SELECT FOR UPDATE 模式，并试图为你的会话锁住所有 WHERE Y=5 的行。一旦完成了这个锁定，它会对这些锁定的数据运行 UPDATE，这样可以确保这一次就能完成而不必（再次）重启动。

但是再想想“会发生什么……“，如果重启动更新，并进入 SELECT FOR UPDATE 模式（与 UPDATE 一样，同样有读一致块获取（read-consistent block get）和读当前块获取（read current block get）），开始 SELECT FOR UPDATE 时 Y=5 的一行等到你得到它的当前版本时却发现 Y=11，会发生什么呢？SELECT FOR UPDATE 会重启动，而且这个循环会再来一遍。

这里要解决两个问题，这两个问题对我来说很有意思。第一个问题是，我们能观察到这种情况吗？可以看到具体是如何发生的吗？第二个问题是，出现这种情况有怎么样呢？这对于我们这些开发人员来说到底有什么意义？下面将分别解决这些问题。

7.4.2 查看重启动

查看重启动比你原来想象的要容易。实际上，可以用一个单行表来观察。我们将用下表进行测试：

```
ops$tkyte@ORA10G> create table t ( x int, y int );

Table created.



ops$tkyte@ORA10G> insert into t values ( 1, 1 );

1 row created.



ops$tkyte@ORA10G> commit;

Commit complete.
```

为了观察重启动，只需要一个触发器打印出一些信息。我们会使用一个 BEFORE UPDATE FOR EACH ROW 触发器打印出行的前映像和作为更新结果的后映像：

```
ops$tkyte@ORA10G> create or replace trigger t_bufer

  2 before update on t for each row

  3 begin

  4 dbms_output.put_line

  5 ( 'old.x = ' || :old.x ||

  6 ', old.y = ' || :old.y );

  7 dbms_output.put_line

  8 ( 'new.x = ' || :new.x ||

  9 ', new.y = ' || :new.y );

  10 end;

  11 /

Trigger created.
```

下面可以更新这一行：

```
ops$tkyte@ORA10G> set serveroutput on
```

```
ops$tkyte@ORA10G> update t set x = x+1;

old.x = 1, old.y = 1

new.x = 2, new.y = 1

1 row updated.
```

到此为止，一切都不出所料：触发器每触发一次，我们都可以看到旧值和新值。不过，要注意，此时还没有提交，这一行仍被锁定。在另一个会话中，执行以下更新：

```
ops$tkyte@ORA10G> set serveroutput on

ops$tkyte@ORA10G> update t set x = x+1 where x > 0;
```

当然，这会立即阻塞，因为第一个会话将这一行锁住了。如果现在回到第一个会话，并提交，会看到第二个会话中有以下输出（为清楚起见，这里把更新语句再写一遍）：

```
ops$tkyte@ORA10G> update t set x = x+1 where x > 0;

old.x = 1, old.y = 1

new.x = 2, new.y = 1

old.x = 2, old.y = 1

new.x = 3, new.y = 1

1 row created.
```

可以看到，行触发器看到这一行有两个版本。行触发器会触发两次：一次提供了行原来的版本以及我们想把原来这个版本修改成什么，另一次提供了最后实际更新的行。由于这是一个 BEFORE FOR EACH ROW 触发器，Oracle 看到了记录的读一致版本，以及我们想对它做的修改。不过，Oracle 以当前模式获取块，从而在 BEFORE FOR EACH ROW 触发器触发之后具体执行更新。它会等待触发器触发后再以当前模式得到块，因为触发器可能会修改:NEW 值。因此 Oracle 在触发器执行之前无法修改这个块，而且触发器的执行可能要花很长时间。由于一次只有一个会话能以当前模式持有一个块；所以 Oracle 需要对处于当前模式下的时间加以限制。

触发器触发后，Oracle 以当前模式获取这个块，并注意到用来查找这一行的 X 列已经修改过。由于使用了 X 来定位这条记录，而且 X 已经修改，所以数据库决定重启动查询。注意，尽管 X 从 1 更新到 2，但这并不会使该行不满足条件（X>0);这 条 UPDATE 语句还是会更新这一行。而且，由于 X 用于定位这一行，而 X 的一致读值（这里是 1）不同于 X 的当前模式读值（2），所以在重启动查询时，触发器把值 X=2（被另一个会话修改之后）看作是:OLD 值，而把 X=3 看作是:NEW 值。

由此就可以看出发生了重启动。要用触发器查看实际的情况；否则，重启动一般是“不

可检测的“。这并不是说无法看到重启动的其他症状，例如更新多行时，发现某一行会导致重启动，而导致一个很大的 UPDATE 语句回滚工作，这也可能是一个症状，但是很难明确地指出”这个症状是重启动造成的“。

可以观察一个有意思的情况，即使语句本身不一定导致重启动，触发器本身也可能导致发生重启动。一般来讲，UPDATE 或 DELETE 语句的 WHERE 子句中引用的列能用于确定修改是否需要重启动。Oracle 使用这些列完成一个一致读，然后以当前模式获取块时，如果检测到任何列有修改，就会重启动这个语句。一般来讲，不会检查行中的其他列。例如，下面重新运行前面的例子，这里使用 WHERE Y>0 来查找行：

```
ops$tkyte@ORA10G> update t set x = x+1 where y > 0;

old.x = 1, old.y = 1

new.x = 2, new.y = 1

old.x = 2, old.y = 1

new.x = 3, new.y = 1

1 row updated.
```

你开始可能会奇怪，“查看 Y 值时，Oracle 为什么会把触发器触发两次？它会检查整个行吗？“从输出结果可以看到，尽管我们在搜索 Y>0，而且根本没有修改 Y，但是更新确实重启动了，触发器又触发了两次。不过，倘若重新创建触发器，只打印出它已触发这个事实就行了，而不再引用:OLD 和:NEW 值：

```
ops$tkyte@ORA10G> create or replace trigger t_bufer

    2 before update on t for each row

    3 begin

    4 dbms_output.put_line( 'fired' );

    5 end;

    6 /

Trigger created.



ops$tkyte@ORA10G> update t set x = x+1;

fired

1 row updated.
```

再到第二个会话中，运行更新后，可以观察到它会阻塞（当然会这样）。提交阻塞会话（即第一个会话）后，可以看到以下输出：

```
ops$tkyte@ORA10G> update t set x = x+1 where y > 0;

fired

1 row updated.
```

这一次触发器只触发了一次，而不是两次。这说明，:NEW 和:OLD 列值在触发器中引用时，也会被 Oracle 用于完成重启动检查。在触发器中引用:NEW.X 和:OLD.X 时，会比较 X 的一致读值和当前读值，并发现二者不同。这就会带来一个重启动。从触发器将这一列的引用去掉后，就没有重启动了。

所以，对此的原则是：WHERE 子句中查找行所用的列集会与行触发器中引用的列进行比较。行的一致读版本会与行的当前读版本比较，只要有不同，就会重启动修改。

注意 根据这些信息，我们可以进一步理解为什么使用 AFTER FOR EACH ROW 触发器比使用 BEFORE FOR EACH ROW 更高效。AFTER 触发器不存在这些问题。

下面再来看另一个问题：“我们为什么要关心重启动？“

7.4.3 为什么重启动对我们很重要？

首先应该注意到“我们的触发器触发了两次！“表中只有一行，而且只有一个 BEFORE FOR EACH ROW 触发器。我们更新了一行，但触发器却触发了两次。

想 想看这会有什么潜在的影响。如果你有一个触发器会做一些非事务性的事情，这可能就是一个相当严重的问题。例如，考虑这样一个触发器，它要发出一个更新（电 子邮件），电子邮件的正文是“这是数据以前的样子，它已经修改成现在这个样子“。如果从触发器直接发送这个电子邮件，（在 Oracle9i 中使用 UTL_SMTY，或者在 Oracle 10g 及以上版本中使用 UTL_MAIL），用户就会收到两个电子邮件，而且其中一个报告的更新从未实际发生过。

如果在触发器中做任何非事务性的工作，就会受到重启动的影响。考虑以下影响：

- 考虑一个触发器，它维护着一些 PL/SQL 全局变量，如所处理的个数。重启动的语句回滚时，对 PL/SQL 变量的修改不会“回滚“。
- 一般认为，以 UTL_开头的几乎所有函数（UTL_FILE、UTL_HTTP、UTL_SMTP 等）都会受到语句重启动的影响。语句重启动时，UTL_FILE 不会“取消“对所写文件的写操作。
- 作为自治事务一部分的触发器肯定会受到影响。语句重启动并回滚时，自治事务无法回滚。

所有这些后果都要小心处理，要想到对于每一个触发器可能会触发多次，或者甚至对根本未被语句更新的行也会触发。

之所以要当心可能的重启动，还有一个原因，这与性能有关。我们一直在使用单行的例子，但是如果你开始一个很大的批更新，而且它处理了前 100,000 条记录后重启动了会怎么样？它会回滚前 100,000 行的修改，以 SELECT FOR UPDATE 模式重启动，在此之后再

完成那 100,000 行的修改。

你可能注意到了，放上那个简单的审计跟踪触发器后（即读取:NEW 和:OLD 值的触发器），即使除了增加了这些新触发器外什么也没有改变，但性能可能会突然差到你无法解释的地步。你可能在重启动过去从未用过的查询。或者你增加了一个小程序，它只更新某处的一行，确使过去只运行 1 个小时的批处理突然需要几个小时才能运行完，其原因只是重启动了过去从未发生过工作。

这不是 Oracle 的一个新特性，从 4.0 版本起，Oracle 数据库就已经有了这个特性，也正是在这个版本中开始引入读一致性。我自己原先就根本没有注意到这是如何工作的，直到 2003 年 的夏天，等我发现了重启动的影响，终于能回答以前困扰我的大量“怎么会发生这种事情？“之类的问题。了解了这一点后，我几乎完全戒除在触发器里使用自治事 务，另外开始重新考虑我的一些应用应该如何实现。例如，我不再从触发器直接发送电子邮件；相反，肯定会在我的事务提交之后用 DBMS_JOB 或新的 Oracle 10g 调度工具发送电子邮件。这样能是电子邮件的发送是”事务性的“，也就是说，如果导致触发器触发和发送电子邮件的语句重启动了，它完成的回滚就会回滚 DBMS_JOB 请求。我修改了在触发器里做的几乎所有非事务性工作，使之在事后的作业中完成，从而得到事务一致性。

7.5 小结

在 这一章中，我介绍了不是内容，难度也很小，可能会让你不时地挠头。不过，理解这些问题至关重要。例如，如果你不知道语句级重启动，可能就不会明白某些情况 怎么会发生。也就是说，你无法解释实验中观察到的一些情况。实际上，如果你不知道有这些重启动，可能会错误地认为错误是环境造成的，或者是最终用户的错 误。这是一个不可重复的问题，因为必须以某种特定顺序做许多事情后才能观察到。

我们介绍了 SQL 标准中定义的隔离级别的含义，并分析了 Oracle 如何实现隔离级别，另外还将 Oracle 的实现与其他数据库的实现做了对照。可以看到，在其他实现中（也就是说，采用读锁提供一致数据的实现），并发性和一致性之间存在着巨大的折衷。为了对数据实现高度并发的访问，就必须对一致性答案减低要求。要想得到一致、正确的答案，则需要忍受并发性的下降。我们看到，在 Oracle 中不是这样的，因为它提供了多版本特性。

表 7-9 对采用读锁的数据库实现与使用的 Oracle 多版本做了一个比较。

表 7-9 Oracle 与采用读锁定机制的数据库之间事务、并发性和锁定行为的比较

隔离级别	实现	写阻塞读	读阻塞写	死锁	不正确的	丢失更新	锁升级	敏感读	查询结果	或限制
READ UNCOMMITTED	非 Oracle	否	否	否	是	是	是			
READ COMMITTED	非 Oracle	是	否	否	是	是	是			

READ COMMITTED Oracle 否 否 否 否 否 * 否

REPEATABLE READ 非 Oracle 是 是 是 否 否 是

SERIALIZABLE 非 Oracle 是 是 是 否 否 是

SERIALIZABLE Oracle 否 否 否 否 否 否

*利用 SELECT FOR UPDATE NOWAIT。

必须很好地掌握并发控制以及数据库如何实现并发控制。我一直在大唱多版本和读一致性的赞歌，但是与世界上的所有事物一样，它们也是双刃剑。如果你不了解存在多版本特性，以及它是如何工作的，在应用设计中就会犯错误。请考虑第 1 章中资源调度程序的例子。在一个不支持多版本及相应非阻塞读的数据库中，该程序原来采用的逻辑可以很好地工作。不过，搬到 Oracle 中实现时，这个逻辑就有麻烦了，它会使数据完整性遭到破坏。除非你知道它如何工作，否则就会写出可能破坏数据的程序，犯这种错误实在太轻而易举了。

第 8 章 事务

事务（Transaction）是数据库区别于文件系统的特性之一。在文件系统中，如果你正把文件写到一半，操作系统突然崩溃了，这个文件就很可能会被破坏。不错，确实还有一些“日报式”（journaled）之类的文件系统，它们能把文件恢复到某个时间点。不过，如果需要保证两个文件同步，这些文件系统就无能为力了。倘若你更新了一个文件，在更新完第二个文件之前，系统突然失败了，你就会有两个不同步的文件。

这是数据库中引入事务的主要目的：事务会把数据库从一种一致状态转变为另一种一致状态。这就是事务的任务。在数据库中提交工作时，可以确保要么所有修改都已经保存，要么所有修改都不保存。另外，还能保证实现了保护数据完整性的各种规则和检查。

在上一章中，我们从并发控制角度讨论了事务，并说明了在高度并发的数据访问条件下，根据 Oracle 的多版本读一致模型，Oracle 事务每次如何提供一致的数据。Oracle 中的事务体现了所有必要的 ACID 特征。ACID 是以下 4 个词的缩写：

- 原子性（atomicity）：事务中的所有动作要么都发生，要么都不发生。
- 一致性（consistency）：事务将数据库从一种一致状态转变为下一种一致状态。
- 隔离性（isolation）：一个事务的影响在该事务提交前对其他事务都不可见。
- 持久性（durability）：事务一旦提交，其结果就是永久性的。

上一章讨论过 Oracle 如何得到一致性和隔离性。这里我们主要关注原子性的概念，并说明 Oracle 中是如何应用这个概念的。

这一章我们将讨论原子性的含义，以及 Oracle 中原子性对语句有什么影响。首先会介绍 COMMIT、SAVEPOINT 和 ROLLBACK 等事务控制语句，并讨论事务中如何保证完整性约束。我们会读到，如果你原来一直在其他数据库中开发，可能在事务方面养成一些坏习惯

（即后面所说的“不好的事务习惯”）。这里还将介绍分布式事务和两段提交（two-phase commit，2PC）。最后会分析自治事务，指出什么是自治事务以及自治事务所扮演的角色。

8.1 事务控制语句

Oracle 中不需要专门的语句来“开始事务”。隐含地，事务会在修改数据的第一条语句处开始（也就是得到 TX 锁的第一条语句）。也可以使用 SET TRANSACTION 或 DBMS_TRANSACTION 包来显示地开始一个事务，但是这一步并不是必要的，这与其他的许多数据库不同，因为那些数据库中都必须显式地开始事务。如果发出 COMMIT 或 ROLLBACK 语句，就会显式地结束一个事务。

注意 ROLLBACK TO SAVEPOINT 命令不会结束事务！正确地写为 ROLLBACK（只有这一个词）才能结束事务。

一定要显式地使用 COMMIT 或 ROLLBACK 来终止你的事务；否则，你使用的工具/环境就会从中挑一个来结束事务。如果正常地退出 SQL*Plus 会话，而没有提交或回滚事务，SQL*Plus 就会认为你希望提交前面做的工作，并为你完成提交。另一方面，如果你只是退出一个 Pro*C 程序，就会发生一个隐式的回滚。不要过分依赖这些隐式行为，因为将来这些行为可能会有改变。一定要显式地 COMMIT 或 ROLLBACK 你的事务。

Oracle 中 的事务是原子性的。这说明无非两种情况：构成事务的每条语句都会提交（成为永久），或者所有语句都回滚。这种保护还延伸到单个的语句。一条语句要么完全成 功，要么这条语句完全回滚。注意，我说的是“语句”回滚。如果一条语句失败，并不会导致先前已经执行的语句自动回滚。它们的工作会保留，必须由你来提交或 回滚。这里谈到了语句和事务是原子性的，在具体介绍其含义之前，先来看看我们可以使用哪些事务控制语句：

- COMMIT：要想使用这个语句的最简形式，只需发出 COMMIT。也可以更详细一些，写为 COMMIT WORK，不过这二者是等价的。COMMIT 会结束你的事务，并使得已做的所有修改成为永久性的（持久保存）。COMMIT 语句还有一些扩展用于分布式事务中。利用这些扩展，允许增加一些有意义的注释为 COMMIT 加标签（对事务加标签），以及强调提交一个可疑的分布式事务。

- ROLLBACK：要想使用这个语句的最简形式，只需发出 ROLLBACK。同样地，你也可以罗嗦一些，写为 ROLLBACK WORK，但是二者是等价的。回滚会结束你的事务，并撤销正在进行的所有未提交的修改。为此要读取存储在回滚段/undo 段中的信息，并把数据库块恢复到事务开始之前的状态（后面我将把回滚段/undo 段统称为 undo 段，Oracle 10g 中都喜欢用这个词）。

- SAVEPOINT：SAVEPOINT 允许你在事务中创建一个“标记点”（marked point），一个事务中可以有多个 SAVEPOINT。

- ROLLBACK TO <SAVEPOINT>：这个语句与 SAVEPOINT 命令一起使用。可以把事务回滚到标记点，而不回滚在此标记点之前的任何工作。所以，可以发出两条 UPDATE 语句，后面跟一个 SAVEPOINT，然后又是两条 DELETE 语句。如果执行 DELETE 语句期间出现了某种异常情况，而且你捕获到这个异常，并发出 ROLLBACK TO SAVEPOINT 命令，事务就会回滚到指定的 SAVEPOINT，撤销 DELETE 完成的所有工作，而 UPDATE 语句完成的工作不受影响。

- SET TRANSACTION：这条语句允许你设置不同的事务属性，如事务的隔离级

别以及事务是只读的还是可读写的。使用手动 undo 管理时，还可以使用这个 来指示事务使用某个特定的 undo 段，不过不推荐这种做法。我们将在第 9 章更详细地讨论手动和自动 undo 管理。

就这么多，没有别的事务控制语句了。最常用的控制语句就是 COMMIT 和 ROLLBACK。SAVEPOINT 语句的用途有点特殊。Oracle 在内部频繁地使用了这个语句，你会发现这语句在你的应用中可能也有用。

8.2 原子性

前面对事务控制语句做了一个简要的概述后，下面可以看看语句原子性、过程原子性和事务原子性到底有什么含义。

8.2.1 语句级原子性

考虑以下语句：

```
Insert into t values ( 1 );
```

看上去很明显，如果它由于一个约束冲突而失败，这一行就不会插入。不过，再考虑下面的例子，这里表 T 上的一个 INSERT 或 DELETE 会触发一个触发器，它将适当地调整表 T2 中的 CNT 列：

```
ops$tkyte@ORA10G> create table t2 ( cnt int );

Table created.



ops$tkyte@ORA10G> insert into t2 values ( 0 );

1 row created.



ops$tkyte@ORA10G> commit;

Commit complete.



ops$tkyte@ORA10G> create table t ( x int check ( x>0 ) );

Table created.



ops$tkyte@ORA10G> create trigger t_trigger
```

```
2 before insert or delete on t for each row

3 begin

4 if ( inserting ) then

5 update t2 set cnt = cnt +1;

6 else

7 update t2 set cnt = cnt -1;

8 end if;

9 dbms_output.put_line( 'I fired and updated ' ||

10 sql%rowcount || ' rows' );

11 end;

12 /

Trigger created.
```

在这种情况下，会发生什么就不那么显而易见了。如果触发器触发之后出现了错误，触发器的影响是否还存在？也就是说，如果触发器被触发，并且更新了 T2，但是这一行没有插入到 T 中，结果会是什么？显然答案应该是，如果并没有真正在 T 中插入一行，我们就希望 T2 中的 CNT 列递增。幸运的是，在 Oracle 中，客户最初发出的语句（在这里就是 INSERT INTO T）会完全成功或完全失败。这个语句是原子性的。以下可以验证这一点：

```
ops$tkyte@ORA10G> set serveroutput on

ops$tkyte@ORA10G> insert into t values (1);

I fired and updated 1 rows

1 row created.

ops$tkyte@ORA10G> insert into t values(-1);

I fired and updated 1 rows

insert into t values(-1)

*
```

```
ERROR at line 1:

ORA-02290: check constraint (OPS$TKYTE.SYS_C009597) violated

ops$tkyte@ORA10G> select * from t2;

CNT

----------

1
```

注意 使用 Oracle9i Release 2 及以前版本的 SQL*Plus 时，要想看到触发器被触发，需要在第二个插入后面增加一行代码：exec null。这是因为，在这些版本中，SQL*Plus 不会在失败的 DML 语句之后获取和显示 DBMS_OUTPUT 信息。Oracle 10g 版本则不然，这个版本的 SQL*Plus 确实会显示 DBMS_OUTPUT 信息。

这样一来，T 中成功地插入一行，而且我们也适当地接收到信息：I fired and updated 1 row。下一个 INSERT 语句违反了 T 上的完整性约束。此时出现了 DBMS_OUTPUT 信息——T 上的触发器确实触发了，这个 DBMS_OUTPUT 信息就是证据。触发器成功地完成了 T2 的更新。我们可能认为现在 T2 中 CNT 的值是 2，但是可以看到它的值实际上为 1。Oracle 保证最初的 INSET（即导致触发器触发的插入语句）是原子性的，这个 INSERT INTO T 是语句，所以 INSERT INTO T 的任何副作用都被认为是语句的一部分。

为了得到这种语句级原子性，Oracle 悄悄地在每个数据库调用外面包了一个 SAVEPOINT。前面的两个 INSERT 实际上处理如下：

```
Savepoint statement1;

Insert into t values ( 1 );

If error then rollback to statement1;

Savepoint statement2;

Insert into t values ( -1 );

If error then rollback to statement2;
```

对于习惯于使用 Sybase 或 SQL Server 的 程序员来说，刚开始可能会有点摸不着头脑。在这些数据库中，情况恰恰相反。这些系统中的触发器会独立于触发语句执行。如果触发器遇到一个错误，它必须显式 地回滚自己的工作，然后产生另外一个错误来回滚触发语句。否则，即使触发语句（或该语句的另外某个部分）最终会失败，触发器完成的工作也会持久保留。

在 Oracle 中，这种语句级原子性可以根据需要延伸。在前面的例子中，如果 INSERT INTO T 触发了一个触发器，这个触发器会更新另一个表，而那个表也有一个触发器，它会删除第三个表（以此类推），那么要么所有工作都成功，要么无一成功。为保证这一点，无

需你编写任何特殊的代码，Oracle 本来就会这么做。

8.2.2 过程级原子性

有意思的是，Oracle 把 PL/SQL 匿名块也当作是语句。请考虑以下存储过程：

```
ops$tkyte@ORA10G> create or replace procedure p
  2 as
  3 begin
  4 insert into t values ( 1 );
  5 insert into t values (-1 );
  6 end;
  7 /
Procedure created.


ops$tkyte@ORA10G> select * from t;
no rows selected


ops$tkyte@ORA10G> select * from t2;
CNT
----------
0
```

以上创建了一个过程，而且我们知道这个过程不会成功。在这个过程中，第二个 INSERT 总会失败。下面看运行这个存储过程时会发生什么情况：

```
ops$tkyte@ORA10G> begin
  2 p;
  3 end;
```

```
  4 /

I fired and updated 1 rows

I fired and updated 1 rows

begin

*

ERROR at line 1:

ORA-02290: check constraint (OPS$TKYTE.SYS_C009598) violated

ORA-06512: at "OPS$TKYTE.P", line 5

ORA-06512: at line 2



ops$tkyte@ORA10G> select * from t;

no rows selected



ops$tkyte@ORA10G> select * from t2;

CNT

----------

0
```

可以看到，Oracle 把这个存储过程调用处理为一个原子语句。客户提交了一个代码块 BEGIN P; END;，Oracle 在它外面包了一个 SAVEPOINT。由于 P 失败了，Oracle 将数据库恢复到调用这个存储过程之前的时间点。下面，如果提交一个稍微不同的代码块，会得到完全不同的结果：

```
ops$tkyte@ORA10G> begin

  2 p;

  3 exception

  4 when others then null;
```

```
     5 end;

     6 /

I fired and updated 1 rows

I fired and updated 1 rows



PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select * from t;

X

----------

1



ops$tkyte@ORA10G> select

CNT

----------

1
```

在此，我们运行的代码块会忽略所有错误，这两个代码块的输出结果有显著的差别。尽管前面第一个 P 调用没有带来任何改变，但在这里的 P 调用中，第一个 INSERT 会成功，而且 T2 中的 CNT 列会相应地递增。

注意 如果代码中包含一个 WHEN OTHERS 异常处理器，但其中没有一个 RAISE 来重新引发异常，我认为这样的代码都是有 bug 的。它会悄悄地忽略错误，这就改变了事务的语义。如果捕获 WHEN OTHERS，并把异常转换为旧式的返回码，这会改变数据库本该有的表现。

Oracle 把客户提交的代码块认为是“语句”。这个语句之所以会成功，因为它自行捕获并忽略了错误，所以 If error then rollback…没有起作用，而且执行这个语句后 Oracle 没有回滚到 SAVEPOINT。因此，这就保留了 P 完成的部分工作。为什么会保留这一部分工作呢？首要的原因是 P 中有语句级原子性：P 中的每条语句都具有原子性。P 提交其两条 INSERT 语句时就成为 Oracle 的客户。每个 INSERT 要么完全成功，要么完全失败。从以下事实就可以证明这一点：可以看到，T 上的触发器触发了两次，而且将 T2 更新了两次，不过 T2 中的计数只反映了一个 UPDATE。P 中执行的第二个 INSERT 外包着一个隐式的 SAVEPOINT。

以上两个代码块的差别很微妙，但是你在应用中必须考虑到这些问题。向一个 PL/SQL 块增加异常处理器可能会显著地改变它的行为。对此可以用另一种方式编写代码，将语句级原子性恢复为整个 PL/SQL 块级原子性，如下所示：

```
ops$tkyte@ORA10G> begin

2 savepoint sp;

3 p;

4 exception

5 when others then

6 rollback to sp;

7 end;

8 /

I fired and updated 1 rows

I fired and updated 1 rows

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select * from t;

no rows selected



ops$tkyte@ORA10G> select * from t2;

CNT

----------

0
```

警告　前面的代码代表着一种极其糟糕的实践。一般来讲，不应该捕获 WHEN OTHERS，另外对于事务语义而言，也不应该为 Oracle 已经提供的特性显式编写代码。

在此模仿了 Oracle 通常用 SAVEPOINT 所做的工作，这样一来，我们不仅仍然能捕获

和“忽略”错误，还能恢复原来的行为。这个例子只作说明之用，这是一种非常糟糕的编码实践。

8.2.3 事务级原子性

最后，还有一种事务级原子性的概念。事务（也就是一组 SQL 语句作为一个工作单元一同执行）的总目标是把数据库从一种一致状态转变为另一种一致状态。

为了实现这个目标，事务也是原子性的，事务完成的所有工作要么完全提交并成为永久性的，要么会回滚并撤销。像语句一样，事务是一个原子性的工作单元。提交一个事务后，接收到数据库返回的“成功”信息后，你就能知道事务完成的所有工作都已经成为永久性的。

8.3 完整性约束和事务

需要指出到底什么时候检查完整性约束。默认情况下，完整性约束会在整个 SQL 语句得到处理之后才进行检查。也有一些可延迟的约束允许将完整性约束的验证延迟到应用请求时（发出一个 SET CONSTRAINTS ALL IMMEDIATE 命令）才完成，或者延迟到发出 COMMIT 时再检查。

8.3.1 IMMEDIATE 约束

在讨论的前一部分，我们假设约束都是 IMMEDIATE 模式，这也是一般情况。在这种情况下，完整性约束会在整个 SQL 语句得到处理之后立即检查。注意，这里我用的是“SQL 语句”而不只是“语句”。如果一个 PL/SQL 存储过程中有多条 SQL 语句，那么在每条 SQL 语句执行之后都会立即验证其完整性约束，而不是在这个存储过程完成后才检查它。

那么，为什么约束要在 SQL 语句执行之后才验证呢？为什么不是在 SQL 语句执行期间验证？这是因为，一条语句可能会使表中的各行暂时地“不一致”，这是很自然的。尽管一条语句全部完成后的最终结果是对的，但如果查看这条语句所做的部分工作，会导致 Oracle 拒绝这个结果。例如，假设有下面这样一个表：

```
ops$tkyte@ORA10G> create table t ( x int unique );

Table created.



ops$tkyte@ORA10G> insert into t values ( 1 );

1 row created.



ops$tkyte@ORA10G> insert into t values ( 2 );

1 row created.
```

现在，我们想执行一个多行 UPDATE：

```
ops$tkyte@ORA10G> update t set x = x+1;

2 rows updated.
```

如果 Oracle 每更新一行之后都检查约束，那么无论什么时候，UPDATE 都有一半的可能性（50%的机会）会失败。由于会以某种顺序来访问 T 中的行，如果 Oracle 先更新 X=1 这一行，那么 X 就会临时地有一个重复的值，这就会拒绝 UPDATE。由于 Oracle 会耐心等待语句结束（而不是在语句执行期间检查约束），所以这条语句最后会成功，因为等到语句完成时已经不存在重复值了。

8.3.2 DEFERRABLE 约束和级联更新

从 Oracle8.0 开始，我们还能够延迟约束检查，对于许多操作来说，这很有好处。首先能想到的是，可能需要将一个主键的 UPDATE 级联到子键。也许很多人会说：这没有必要，因为主键是不可变的（我就是这些人之一），但是还有人坚持要有级联 UPDATE。有了可延迟的约束，就使得级联更新成为可能。

注意 一 般认为，完成更新级联来修改主键是很不好的做法。这会破坏主键的意图。如果你必须做一次级联更新来修正不对的信息，这倒是可以的；但是如果你发现自己在不 停地完成级联更新，并把这当做应用的一部分，那就是另一码事了，你应该退一步，重新考虑一下这个过程。倘若真是这样，能你就是错把鸡毛当令箭了！

在以前的版本中，确实也可以完成 CASCADE UPDATE，但是为此需要做大量的工作，而且存在某些限制。有了可延迟的约束后，这就变得易如反掌了。代码如下：

```
ops$tkyte@ORA10G> create table p
  2 ( pk int primary key )
  3 /

Table created.



ops$tkyte@ORA10G> create table c
  2 ( fk constraint c_fk
  3 references p(pk)
  4 deferrable
  5 initially immediate
  6 )
```

```
      7 /

Table created.



ops$tkyte@ORA10G> insert into p values ( 1 );

1 row created.



ops$tkyte@ORA10G> insert into c values ( 1 );

1 row created.
```

我们有一个父表 P，还有一个子表 C。表 C 引用了表 P，保证这个规则的约束是 C_FK（子外键）。这个约束创建为一个 DEFERRABLE 约束，但是设置为 INITIALLY IMMEDIATE。这说明，可以把这个约束延迟到 COMMIT 或另外某个时间才检查。不过，默认情况下，这个约束在语句级验证。这是可延迟约束最常见的用法。大多数现有的应用不会在 COMMIT 语句上检查约束冲突，你最好也不要这么做。根据定义，表 C 与一般的表一样有正常的表现，不过我们可以显式地改变它的行为。下面，在这些表上尝试一些 DML，看看会发生什么：

```
ops$tkyte@ORA10G> update p set pk = 2;

update p set pk = 2

*

ERROR at line 1:

ORA-02292: integrity constraint (OPS$TKYTE.C_FK) violated - child record found
```

由于约束是 IMMEDIATE 模式，这个 UPDATE 会失败。下面换个模式再试一次：

```
ops$tkyte@ORA10G> set constraint c_fk deferred;

Constraint set.



ops$tkyte@ORA10G> update p set pk = 2;

1 row updated.
```

现 在更新成功了。为了便于说明，下面将显示如何在提交前显式地检查了一个延迟约束，才中了解我们所做的修改与业务规则是否一致（换句话说，检查目前确实没有 违反约

束）。应该在提交之前或者在把控制交给程序的另外某个部分（这一部分可能不希望有延迟约束）之前做这个工作，这是一个很好的主意：

```
ops$tkyte@ORA10G> set constraint c_fk immediate;

set constraint c_fk immediate

*

ERROR at line 1:

ORA-02291: integrity constraint (OPS$TKYTE.C_FK) violated - parent key not found
```

不出所料，它会失败，并立即返回一个错误，因为我们知道以上更新会违反约束。对 P 的 UPDATE 没有回滚（否则会违反语句级原子性）；它仍在进行。还要注意，我们的事务仍把 C_FK 当作延迟约束，因为 SET CONSTRAINT 命令失败了。下面继续将 UPDATE 级联到 C：

```
ops$tkyte@ORA10G> update c set fk = 2;

1 row updated.



ops$tkyte@ORA10G> set constraint c_fk immediate;

Constraint set.



ops$tkyte@ORA10G> commit;

Commit complete.
```

这就是级联更新的做法。注意，要延迟一个约束，必须这样来创建它们：先将其删除，再重新创建约束，这样才能把不可延迟的约束改变为可延迟约束。

8.4 不好的事务习惯

许多开发人员在事务方面都有一些不好的习惯。如果开发人员使用过另外某个数据库，其中只是“支持”事务，而没有“提升”事务的使用，执行开发人员就常常有这样的一些坏习惯。例如，在 Informix(默认设置)、Sybase 和 SQL Server 中，必须显式地 BEGIN（开始）一个事务；否则，每条单个的语句本身就是一个事务。Oracle 在具体的语句外包了一个 SAVEPOINT，采用类似的方式，那些数据库则在各条语句外包了一个 BEGIN WORK/COMMIT 或 ROLLBACK。这是因为，在这些数据库中，锁是稀有资源，另外读取器会阻塞写入器，反之，写入器也会阻塞读取器。为了提高并发性，这些数据库希望你的事务越小越好，有时甚至会以数据完整性为代价来做到这一点。

Oracle 则采用了完全不同的方法。事务总是隐式的，没有办法“自动提交”事务，除非应用专门实现（更多详细内容请见 8.4.2 节 。在 Oracle 中， 每个事务都应该只在必要时才提交，而在此之前不能提交。事务的大小要根据需要而定。锁、阻塞等问题并不是决定事务大小的关键，数据完整性才是确定事务大小 的根本。锁不是稀有资源，并发的数据读取器和数据写入器之间不存在竞争问题。这样在数据库中就能有健壮的事务。这些事务不必很短，而要根据需求有足够长的 持续时间（但是不能不必要地太长）。事务不是为了方便计算机及其软件，而是为了保护你的数据。

8.4.1 在循环中提交

如果交给你一个任务，要求更新多行，大多数程序员都会力图找出一种过程性方法，通过循环来完成这个任务，这样就能提交多行。据我听到的，这样做的两个主要原因是：

- □ 频繁地提交大量小事务比处理和提交一个大事务更快，也更高效。
- □ 没有足够的 undo 空间。

这 两个结论都存在误导性。另外，如果提交得太过频繁，很容易让你陷入危险，倘若更新做到一半的时候失败了，这会使你的数据库处于一种“未知”的状态。要编写 一个过程从而在出现失败的情况下能平滑地重启动，这需要复杂的逻辑。到目前为止，最好的方法是按业务过程的要求以适当的频度提交，并且相应地设置 undo 段大小。

下面将更详细地分析这些问题。

1. 性能影响

如果频繁地提交，通常并不会更快。一般地，在一个 SQL 语句中完成工作几乎总是更快一些。可以通过一个小例子来说明，假设我们有一个表 T，其中有大量的行，而且我们希望为该表中的每一行更新一个列值。这里使用两个表 T1 和 T2 来进行演示：

```
ops$tkyte@ORA10G> create table t1 as select * from all_objects;

Table created.



ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T1' );

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> create table t2 as select * from t1;

Table created.
```

```
ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T2' );

PL/SQL procedure successfully completed.
```

这样一来，更新时，只需简单地在一条 UPDATE 语句中完成，如下：

```
ops$tkyte@ORA10G> set timing on

ops$tkyte@ORA10G> update t1 set object_name = lower(object_name);

48306 rows updated.



Elapsed: 00:00:00.31
```

不过大多数人更喜欢像下面这样做（不管由于什么原因）：

```
ops$tkyte@ORA10G> begin
  2      for x in ( select rowid rid, object_name, rownum r
  3                 from t2 )
  4      loop
  5          update t2
  6             set object_name = lower(x.object_name)
  7          where rowid = x.rid;
  8          if ( mod(x.r,100) = 0 ) then
  9             commit;
 10          end if;
 11     end loop;
 12     commit;
 13 end;
 14 /

PL/SQL procedure successfully completed.
```

```
Elapsed: 00:00:05.38
```

对于这个小例子，倘若在循环中频繁地提交，就会慢上好几倍。如果能在一条 SQL 语句中完成，就要尽量这么做，因为这样几乎总是更快。即使我们“优化”了过程性代码，也要使用批处理来完成更新，如下：

```
ops$tkyte@ORA10G> declare
  2          type ridArray is table of rowid;
  3          type vcArray is table of t2.object_name%type;
  4
  5          l_rids ridArray;
  6          l_names vcArray;
  7
  8          cursor c is select rowid, object_name from t2;
  9  begin
 10          open c;
 11          loop
 12                  fetch c bulk collect into l_rids, l_names LIMIT 100;
 13                  forall i in 1 .. l_rids.count
 14                  update t2
 15                          set object_name = lower(l_names(i))
 16                  where rowid = l_rids(i);
 17                  commit;
 18                  exit when c%notfound;
 19          end loop;
 20          close c;
```

```
21      end;

22 /

PL/SQL procedure successfully completed.

Elapsed: 00:00:02.36
```

这确实要块一些，但是本来还可以更快的。不仅如此，你还应该注意到这段代码变得越来越复杂。从极其简单的一条 UPDATE 语句，到过程性代码，再到更复杂的过程性代码，我们正沿着错误的反向越走越远！

下面再对这个讨论做个补充，给出一个对应的例子。应该记得在第 7 章中，我们讨论过写一致性的概念，并介绍了 UPDATE 语句如何导致重启动。如果要针对一个行子集（有一个 WHERE 子句）执行先前的 UPDATE 语句，而其他用户正在修改这个 UPDATE 在 WHERE 子句中使用的列，就可能需要使用一系列较小的事务而不是一个大事务，或者更适合在执行大量更新之前先锁定表。这样做的目标是减少出现重启动的机会。如果要 UPDATE 表中的大量行，这会导致我们使用 LOCK TABLE 命 令。不过，根据我的经验，这种大量更新或大量删除（只有这些语句才可能遭遇重启动）都是独立完成的。一次性的大量更新或清除旧数据通常不会在活动高发期间 完成。实际上，数据的清除根本不应受此影响，因为我们一般会使用某个日期字段来定位要清除的信息，而其他应用不会修改这个日期数据。

2. Snapshot Too Old 错误

下面来看开发人员喜欢在过程循环中提交更新的第二个原因，这是因为他们可能被误导，试图节俭地使用“受限资源”（undo 段）。这是一个配置问题；你需要确保有足够的 undo 空间来适当地确定事务的大小。如果在循环中提交，一般会更慢，不仅如此，这也是导致让人胆战心惊的 ORA-01555 错误的最常见的原因。下面将更详细地说明。

如果你阅读过第 1 章和第 7 章，就会知道，Oracle 的多版本模型会使用 undo 段数据依照语句或事务开始时的原样来重建块（究竟是语句还是事务，这取决于隔离模式）。如果必要的 undo 信息不再存在，你就会收到一个 ORA-01555：snapshot too old 错误消息，查询也不会完成。所以，如果像前面的那个例子一样，你一边在读取表，一边在修改这个表，就会同时生成查询所需的 undo 信息。UPDATE 生成了 undo 信息，你的查询可能会利用这些 undo 信息来得到待更新数据的读一致视图。如果提交了所做的更新，就会允许系统重用刚刚填写的 undo 段空间。如果系统确实重用了 undo 段空间，擦除了旧的 undo 数据（查询随后要用到这些 undo 信息），你就有大麻烦了。SELECT 会失败，而 UPDATE 也会中途停止。这样就有了一个部分完成的逻辑事务，而且可能没有什么好办法来重启动（对此稍后还会更多说明）。

下面通过一个小演示例子来看看这个概念具体是怎样的。我在一个很小的测试数据库中建立了一个表：

```
ops$tkyte@ORA10G> create table t as select * from all_objects;

Table created.
```

```
ops$tkyte@ORA10G> create index t_idx on t(object_name);

Index created.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T', cascade=>true );

PL/SQL procedure successfully completed.
```

然后创建一个非常小的 undo 表空间，并修改系统，要求使用这个 undo 表空间。注意，通过将 AUTOEXTEND 设置为 off，已经把这个系统中全部 UNDO 空间的大小限制为 2MB 或更小：

```
ops$tkyte@ORA10G> create undo tablespace undo_small
  2 datafile size 2m
  3 autoextend off
  4 /

Tablespace created.


ops$tkyte@ORA10G> alter system set undo_tablespace = undo_small;

System altered.
```

现在只能用这个小 undo 表空间，我运行了以下代码块来完成 UPDATE：

```
ops$tkyte@ORA10G> begin
  2      for x in ( select /*+ INDEX(t t_idx) */ rowid rid, object_name, rownum r
  3                 from t
  4                 where object_name > ' ' )
  5      loop
  6          update t
  7             set object_name = lower(x.object_name)
  8          where rowid = x.rid;
```

```
 9              if ( mod(x.r,100) = 0 ) then
10                      commit;
11              end if;
12      end loop;
13      commit;
14 end;
15 /
begin
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number with name "" too small
ORA-06512: at line 2
```

我收到了这个错误。应该指出，这里向查询增加了一个索引提示以及一个 WHERE 子句，以确保随机地读取这个表（这两方面加在一起，就能使基于代价的优化器读取按索引键“排序”的表）。通过索引来处理表时，往往会为某一行读取一个块，可是我们想要的下一行又在另一个块上。最终，我们确实会处理块 1 上的所有行，只不过不是同时处理。假设块 1 可能包含 OBJECT_NAME 以字母 A、M、N、Q 和 Z 开头的所有行的数据。这样我们就会多次命中（读取）这个块，因为我们在读取按 OBJECT_NAME 排序的数据，而且可能有很多行的 OBJECT_NAME 以 A~M 之间的字母开头。由于我们正在频繁地提交和重用 undo 空间，最终再次访问一个块时，可能已经无法再回滚到查询开始的那个时间点，此时就会得到这个错误。

这是一个特意构造的例子，纯粹是为了说明如何以一种可靠的方式发生这个错误。UPDATE 语句正在生成 undo 信息。我只能用一个很小的 undo 表空间（大小为 2MB）。这里多次在 undo 段中回绕，因为 undo 段要以一种循环方式使用。每次提交时，都允许 Oracle 覆盖前面生成的 undo 数据。最终，可能需要某些已生成的 undo 数据，但是它已经不在了（即已经被覆盖），这样就会收到 ORA-01555 错误。

你可能会指出，在这种情况下，如果没有在上一个例子的第 10 行提交，就会收到以下错误：

```
begin
*
```

```
ERROR at line 1:

ORA-30036: unable to extend segment by 8 in undo tablespace 'UNDO_SMALL'

ORA-06512: at line 6
```

你说的没错。这两个错误之间的主要区别在于：

报 ORA-01555 错误的例子会使更新处于一种完全未知的状态。有些工作已经做了，而有些还没有做。

如果在游标的 FOR 循环中提交，要想避免 ORA-01555，我绝对是无计可施。

ORA-30036 错误是可以避免的，只需在系统中分配适当的资源。通过设置正确的大小就可以避免这个错误；但是第一个错误（ORA-01555）则不然。另外，即使我未能避免 ORA-30036 错误，至少更新会回滚，数据库还是处于一种已知的一致状态，而不会半路停在某个大更新的中间。

这里的关键是，无法通过频繁提交来“节省”undo 空间——你会需要这些 undo 信息。收到 ORA-01555 错误时，我运行的是单用户系统。只需一个会话就能导致这个错误，在实际中，很多情况下甚至各个会话都能导致自己的 ORA-01555 错误。开发人员和 DBA 需要协作来适当地确定这些段的大小，从而完成所需完成的任务。这里没有捷径。你必须通过分析系统来发现最大的事务是什么，并适当地为之确定段大小。动态性能视图 V$UNDOSTAT 对于监视所生成的 undo 数量可能非常有用，你可以用来监视运行时间最长的查询的持续时间。许多人认为像临时段、undo 和 redo 都是“开销”，应该分配尽可能小的存储空间。这与计算机行业 2000 年 1 月 1 日遭遇的千年虫问题同出一辙，所有问题都只是因为想在日期字段中节省 2 个字节。数据库的这些组件不是开销，而是系统的关键组件。必须适当地设置大小（不要太大，也不要太小）。

3. 可重启动的过程需要复杂的逻辑

如果采用“在逻辑事务结束之前提交”的方法，最验证的问题是：如果 UPDATE 半截失败了，这会经常将你的数据库置于一种未知的状态中。除非你提取对此做了规划，否则很难重启动这个失败的过程，让它从摔倒的地方再爬起来。例如，假设我们不是像上一个例子那样对列应用 LOWER()函数，而是应用了以下的列函数：

```
last_ddl_time = last_ddl_time + 1;
```

如果 UPDATE 循环半路停止了，怎么重启动呢？我们不能简单地重新运行，因为这样有可能导致某些日期加 2，而另外一些只加了 1.如果我们再次失败，可能会对某些日期加 3，另外一些加 2，还有一些加 1，依此类推。我们还需要更复杂的逻辑，必须有办法对数据“分区”。例如，可以处理以 A 开头的每一个 OBJECT_NAME，然后是以 B 开头的，依此类推：

```
ops$tkyte@ORA10G> create table to_do
  2 as
  3       select distinct substr( object_name, 1,1 ) first_char
```

```
  4          from T
  5 /

Table created.


ops$tkyte@ORA10G> begin
  2 for x in ( select * from to_do )
  3 loop
  4          update t set last_ddl_time = last_ddl_time+1
  5          where object_name like x.first_char || '%';
  6
  7          dbms_output.put_line( sql%rowcount || ' rows updated' );
  8          delete from to_do where first_char = x.first_char;
  9
  10                 commit;
  11         end loop;
  12 end;
  13 /
22257 rows updated
1167 rows updated
135 rows updated
1139 rows updated
2993 rows updated
691 rows updated
```

```
...

2810 rows updated

6 rows updated

10 rows updated

2849 rows updated

1 rows updated

2 rows updated

7 rows updated



PL/SQL procedure successfully completed.
```

现在，如果这个过程失败了，我们就能重启动，因为不会再处理已经得到成功处理的任何对象名。不过，这种方法也是有问题的，除非有某个属性能均匀地划分数据。否则最终行的分布就会差异很大在这里，第一个 UPDATE 比所有其他 UPDATE 加在一起完成的工作还多。另外，如果其他会话正在访问这个表，并且在修改数据，则它们可能也会更新 OBJECT_NAME 字段。假设我们已经处理完 A 对象，此后另外某个会话把名为 Z 的对象更新为 A，我们就会漏掉这个记录。更进一步，与 UPDATE T SET LAST_DDL_TIME = LAST_DDL_TIME+1 相比，这个过程效率非常低。我们可能使用索引来读取表中的每一行，或者我们要对它做 n 次全扫描，不论哪种情况这个过程都不能让人满意。这种方法的缺点太多了。

最好的方法还是我在第 1 章刚开始时推荐的做法：力求简单。如果能在 SQL 中完成，那就在 SQL 里完成。如果不能在 SQL 中完成，就用 PL/SQL 实 现。要用尽可能少的代码来完成，另外应当分配充分的资源。一定要考虑到万一出现错误会怎么样。有些人编写了更新循环，对测试数据做了大量工作，但是把这个 更新循环应用到实际数据上时，却中途失败了，这种情况我见得实在太多了。此时他们确实很为难，因为不知道处理是在哪里停止的。应当正确地设置 undo 段的大小，比起编写一个可重启动的程序来说，前者要容易得多。如果你有非常大的表需要更新，就应该使用分区(有关的更多内容见第 10 章)，这样就能单独地更新各个分区。甚至可以使用并行 DML 来执行更新。

8.4.2 使用自动提交

关于不好的事务习惯，最后要说的是由于使用流行的编程 API（ODBC 和 JDBC）所带来的问题。这些 API 会默认地”自动提交“（autocommit）。考虑以下语句，它把$1,000 从一个支票账户转到一个储蓄账户：

```
update accounts set balance = balance - 1000 where account_id = 123;
```

```
update accounts set balance = balance + 1000 where account_id = 456;
```

如果提交这些语句时你的程序在使用 JDBC，那么 JDBC 会（悄悄地）在每个 UPDATE 之后插入一个提交。如果在第一个 UPDATE 之后并在第二个 UPDATE 之前系统失败了，请考虑这个自动提交所带来的影响。你会凭空失去$1,000！

我很清楚为什么 ODBC 会这样做。这是因为，ODBC 是 SQL Server 的开发人员设计的，而 SQL Server 数据库要求使用非常短的事务，这是有其并发模型造成的（SQL Server 数据库的并发模型是：写会阻塞读，读会阻塞写，而且锁是稀有资源）。但我不能理解的是，这一点为什么会传承到 JDBC，要知道这个 API 本来是要支持“企业“的。在我看来，在 JDBC 中打开一个连接之后紧接着应该有下面这样几行代码：

```
connection conn = DriverManager.getConnection

("jdbc:oracle:oci:@database","scott","tiger");

conn.setAutoCommit (false);
```

这会把事务的控制权返回给你（开发人员），这才是合适的。这样，你就能安全地编写转账事务，并在两个语句都成功之后再提交。在这种情况下，如果对 API 缺乏了解，结果将是致命的。我曾经见过很多开发人员没有注意到这个自动提交“特性“，等到出现错误时应用就会遇到大麻烦。

8.5 分布式事务

Oracle 有很多很好的特性，其中之一就是能够透明地处理分布式事务。在一个事务的范围内，可以更新多个不同数据库中的数据。提交时，要么提交所有实例中的更新，要么一个都不提交（它们都会回滚）。为此，我不需要另外编写任何代码：只是“提交“就行了。

Oracle 中分布式事务的关键是数据库链接（database link）。数据库链接是一个数据库对象，描述了如何从你的实例登录到另一个实例。不过，这一节的目的不是介绍数据库链接命令的语法（在文档中有全面的说明），而是要展示这些数据库链接是存在的。一旦建立了一个数据库链接，访问远程对象就很简单了，如下：

```
select * from T@another_database;
```

这会从数据库链接 ANOTHER_DATABASE 所定义数据库实例的表 T 中选择。一般地，你会创建表 T 的一个视图（或一个同义词），来“隐藏“T 是一个远程表的事实。例如，可以发出以下命令，然后就可以像访问本地表一样地访问 T 了：

```
create synonym T for T@another_database;
```

既然建立了这个数据库链接，而且能读取一些表，还能够修改这些表（当然，假设有适当的权限）。现在执行一个分布式事务与执行一个本地事务没有什么两样。我要做的只是：

```
update local_table set x = 5;

update remote_table@another_database set y = 10;
```

```
commit;
```

就这么简单。Oracle 会完成所有数据库中的提交，或者都不提交。它使用了一个 2PC 协议来做到这一点。2PC 是一个分布式协议，如果一个修改影响到多个不同的数据库，2PC 允许原子性地提交这个修改。它会在提交之前尽可能地关闭分布式失败窗口。在多个数据库之间的一个 2PC 事务中，其中一个数据库（通常是客户最初登录的那个数据库）会成为分布式事务的协调器。这个站点会询问其他站点是否已经准备好提交。实际上，这个站点会转向其他站点，问它们是否准备就绪。其他的每个站点会报告它的“就绪状态“（YES 或 NO）。如果只要有一个站点投票 NO，整个事务就会回滚。如果所有站点都投票 YES，站点协调器会广播一条消息，使每个站点上的提交成为永久性的。

2PC 会限制可能出现的严重错误的窗口（时间窗）。在 2PC 上“投票“之前，任何分布式错误都会导致所有这点回滚。对于事务的结果来说，这里不存在疑义。在提交或回滚之后，分布式事务的结果同样没有疑义。只有一个非常短的时间窗除外，此时协调器要收集投票结果，只有在这个时候如果失败，结果可能有疑义。

例如，假设有 3 个站点参与一个事务，其中站点 1 是协调器。站点 1 问站点 2 是否准备好提交，站点 2 报告说是。站点 1 再问站点 3 是否准备好提交，站点 3 也说准备好了。在这个时间点，站点 1 就是惟一知道事务结果的站点，它现在要负责把这个结果广播给其他站点。如果现在出现一个错误，比如说网络失败了，站点 1 掉电，或者其他某个原因，站点 2 和站点 3 就会“挂起“它们就会有所谓的可疑分布式事务（in-doubt distributed transaction）。2PC 协议力图尽可能地关闭这个错误窗口，但是无法完全将其关闭。站点 2 和站点 3 必须保持事务打开，等待站点 1 发出的结果通知。如果还记得第 5 章讨论的体系结构，应该知道这个问题要由 RECO 进程来解决。有 FORCE 选项的 COMMIT 和 ROLLBACK 在这里就有了用武之地。如果问题的原因是站点 1、2 和 3 之间的网络故障，站点 2 和站点 3 的 DBA 可以打电话给站点 1 的 DBA，问他结果是什么，并相应地手动应用提交或回滚。

对于分布式事务中能做的事情，还存在一些限制（不过并不多），这些限制是合理的（在我看来，它们确实是合理的）。其中重要的限制如下：

- 不能在数据库链接上发出 COMMIT。也就是说，不能发出 COMMIT@remote_site。只能从发起事务的那个站点提交。

- 不能在数据库链接上完成 DDL。这是上一个问题带来的直接结果。DDL 会提交，而除了发起事务的站点外，你不能从任何其他站点提交，所以不能在数据库链接上完成 DDL。

- 不能在数据库链接上发出 SAVEPOINT。简单地说，不能在数据库链接发出任务事务控制语句。所有事务控制都有最初打开数据库链接的会话继承得来；对于事务中的分布式实例，不能有不同的事务控制。

尽管数据库链接上缺乏事务控制，但是这是合理的，因为只有发起事务的站点才有参与事务的所有站点的一个列表。在我们的 3 站点配置中，如果站点 2 试图提交，它无从知道站点 3 也参与了这个事务。在 Oracle 中，只有站点 1 可以发出提交命令。此时，允许站点 1 把分布式事务控制的责任委托给另一个站点。

我们可以设置站点的 COMMIT_POINT_STRENGTH（这是一个参数），从而改变具体

的提交站点。提交点能力（commit-point strength） 会为分布式事务中的服务器关联一个相对的重要性级别。服务器越重要（要求这个服务器上的数据有更大的可用性），它就越有可能协调这个分布式事务。如果需要 在生产主机和测试主机之间完成一个分布式事务，你可能就希望这样做。由于事务协调器对于事务的结果绝对不会有疑义，最好是由生产主机协调分布式事务。你并 不关心测试主机是否有一些打开的事务和锁定的资源。但是如果生产系统上有这种情况，你肯定会很关心。

不能在数据库链接上执行 DDL，实际上这并不太糟糕。首先，DDL 很“少见“。只会在安装或升级期间执行一次 DDL。生产系统不会执行 DDL（应该说，生产系统不应该执行 DDL）。其次，要在数据库链接上执行 DDL 也是有办法的，只是要采用另一种方式。可以使用作业队列工具 DBMS_JOB，或者在 Oracle 10g 中可以使用调度工具包 DBMS_SCHEDULER。你不用试图在链接上执行 DDL，而是可以使用链接来调度一个远程作业，一旦提交就执行这个远程作业。采用这种方式，作业在远程主机上运行，这不是一个分布式事务，可以执行 DDL。实际上，Oracle Replication Services（远程服务）就采用这种方法执行分布式 DDL 来实现模式复制。

8.6 自治事务

自治事务（autonomous transaction）允许你创建一个“事务中的事务），它能独立于其父事务提交或回滚。利用自治事务，可以挂起当前执行的事务，开始一个新事务，完成一些工作，然后提交或回滚，所有这些都不影响当前执行事务的状态。自治事务提供了一种用 PL/SQL 控制事务的新方法，可以用于：

- ☐ 顶层匿名块；
- ☐ 本地（过程中的过程）、独立或打包的函数和过程；
- ☐ 对象类型的方法；
- ☐ 数据库触发器。

在 介绍自治事务如何工作之前，我想再强调一下，自治事务是一种功能非常强大的工具，但是如果使用不当又会相当危险。真正需要自治事务的情况实际上极其少见。 我对使用了自治事务的代码都持怀疑态度，这些代码需要多看几眼，要更仔细地审查。在使用自治事务的系统中偶然引入逻辑数据完整性问题实在是太容易了。在下 面几节中，我们首先介绍自治事务如何工作，再讨论什么情况下可以安全地使用自治事务。

8.6.1 自治事务如果工作？

要展示自治事务的动作和结果，最好的办法是通过例子来说明。我们将创建一个简单的表来保存消息：

```
ops$tkyte@ORA10G> create table t ( msg varchar2(25) );

Table created.
```

接下来创建两个过程，每个过程只是将其名字插入到消息表中，然后提交。不过，其中一个过程是正常的过程，另一个编写为自治事务。我们将使用这些对象来显示在各种情况下哪些工作会在数据库中持久保留（被提交）。

首先是 AUTONOMOUS_INSERT 过程：

```
ops$tkyte@ORA10G> create or replace procedure Autonomous_Insert

2 as

3 pragma autonomous_transaction;

4 begin

5 insert into t values ( 'Autonomous Insert' );

6 commit;

7 end;

8 /

Procedure created.
```

注意这里使用了 pragma AUTONOMOUS_TRANSACTION。这个指令告诉数据库：执行这个过程时要作为一个新的自治事务来执行，而且独立于其父事务。

注意 pragma 是一个编译器指令，这是一种编辑器执行某种编译选项的方法。还有其他一些 pragma。参考 PL/SQL 编程手册，可以看到其索引中有 pragma 的一个列表。

以下是“正常”的 NONAUTONOMOUS_INSERT 过程：

```
ops$tkyte@ORA10G> create or replace procedure NonAutonomous_Insert

2 as

3 begin

4 insert into t values ( 'NonAutonomous Insert' );

5 commit;

6 end;

7 /

Procedure created.
```

下面来观察 PL/SQL 代码匿名块中非自治（nonautonomous）事务的行为：

```
ops$tkyte@ORA10G> begin
```

```
  2 insert into t values ( 'Anonymous Block' );

  3 NonAutonomous_Insert;

  4 rollback;

  5 end;

  6 /

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select * from t;

MSG

------------------------

Anonymous Block

NonAutonomous Insert
```

可以看到，匿名块执行的工作（INSERT）由 NONAUTONOMOUNS_INSERT 过程提交。两个数据行都已提交，所以 ROLLBACK 命令没有什么可以回滚。把这个过程与自治事务过程的行为加以比较：

```
ops$tkyte@ORA10G> delete from t;

2 rows deleted.



ops$tkyte@ORA10G> commit;

Commit complete.



ops$tkyte@ORA10G> begin

  2          insert into t values ( 'Anonymous Block' );

  3           Autonomous_Insert;
```

```
  4          rollback;

  5 end;

  6 /

PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select * from t;

MSG

------------------------

Autonomous Insert
```

在此，只有自治事务中完成并已提交的工作会持久保留。匿名块中完成的 INSERT 由第 4 行的回滚语句回滚。自治事务过程的 COMMIT 对匿名块中开始的父事务没有影响。本质上讲，中就抓住了自治事务的精髓，并能从中了解到自治事务做什么。

总结一下，如果在一个“正常”的过程中 COMMIT，它不仅会持久保留自己的工作，也会使该会话中未完成的工作成为永久性的。不过，如果在一个自治事务过程中完成 COMMIT，只会让这个过程本身的工作成为永久性的。

8.6.2 何时使用自治事务？

Oracle 数据库在内部支持自治事务时间已经不短了。我们看到的一直都是以递归 SQL 形式出现的自治事务。例如，从一个序列选择时就可以完成一个递归事务，从而在 SYS.SEQ$表中立即递增序列。为支持你的序列，SYS.SEQ$表的更新会立即提交，并对其他事务可见，但是此时你的事务（即父事务）尚未提交。另外，如果回滚你的事务，序列的递增仍会保留；它没有随父事务回滚，因为这部分工作已经提交。空间管理、审计以及其他内部操作都是以类似的递归方式完成的。

这个特性现在已经公开，任何人都能使用。不过我发现，在实际世界中，自治事务的合理使用实在很有限。我多次看到，人们往往把自治事务当作某些问题的迂回解决方法，如触发器的变异表约束（变异表也称为变化表，mutating table）。 不过，这几乎总会导致数据完整性问题，因为变异表就是为了读取触发了触发器的表。不错，通过使用自治事务，确实可以查询表，但是你现在只是查询表，并不能 看到你的修改（而这是变异表约束本来要做的事情；表正在修改当中，所以查询结果将不一致）。根据这个触发器的查询所做的任何决定可能都有问题，你只是在读 取那个时间点的“旧”数据。

自治事务的一种可能合法的使用是用于定制审计，不过我要强调这只是“可能合法”。数据库中要对信息完成审计，与编写定制的触发器相比，还有许多更高效的方法。例如，可以使用 DBMS_FGA 包或者只是使用 AUDIT 命令本身。

对 此，应用开发人员经常问我：“怎么能对每一个修改安全信息的企图进行审计，并记录他们试图修改的值呢？”他们不只是想避免修改企图的发生，还想为这些企图 建立一个永久的记录。在自治事务出现之前，许多开发人员尝试着使用标准触发器（没有自治事务）

来达到这个目的，但是失败了。触发器能检测到 UPDATE，而且发现一个用户在修改他不该修改的数据，此时就会创建一个审计记录，并使 UPDATE 失败。遗憾的是，触发器让 UPDATE 失败时，它也会回滚审计记录，这是一个“全有或全无”性质的失败。有了自治事务，现在就可以安全地捕获到企图完成的没有权限修改的数，而且已经对这个企图建立了一个记录。

有意思的是，Oracle 的 AUDIT 命令也提供了这个能力，多年前就可以使用自治事务捕获未成功的信息修改企图。现在这个特性已经向 Oracle 开发人员公开，这样我们就能创建自己的更为定制的审计。

下 面是一个小例子。先在表上放一个自治事务触发器，它能捕获一个审计跟踪记录，详细地指出谁试图更新表，这个人什么时候想更新表，另外还会提供一个描述性消 息指出这个人想要修改什么数据。这个触发器的基本逻辑是：对于不向你直接或间接报告的员工，要防止更新这些员工记录的任何企图。

首先，从 SCOTT 模式建立 EMP 表的一个副本，以此作为本例使用的表：

```
ops$tkyte@ORA10G> create table emp

  2 as

  3 select * from scott.emp;

Table created.

ops$tkyte@ORA10G> grant all on emp to scott;

Grant succeeded.
```

还要创建一个 AUDIT_TAB 表，在这个表中存储审计信息。注意，我们使用了列的 DEFAULT 属性，从而默认具有当前登录的用户名以及登记审计跟踪信息的当前日期/时间：

```
ops$tkyte@ORA10G> create table audit_tab

  2 ( username varchar2(30) default user,

  3 timestamp date default sysdate,

  4 msg varchar2(4000)

  5 )

  6 /

Table created.
```

接下来，创建一个 EMP_AUDIT 触发器对 EMP 表上的 UPDATE 活动进行审计：

```
ops$tkyte@ORA10G> create or replace trigger EMP_AUDIT
  2 before update on emp
  3 for each row
  4 declare
  5      pragma autonomous_transaction;
  6      l_cnt number;
  7 begin
  8
  9      select count(*) into l_cnt
 10       from dual
 11       where EXISTS ( select null
 12                      from emp
 13                      where empno = :new.empno
 14                      start with mgr = ( select empno
 15                                         from emp
 16                                         where ename = USER )
 17                      connect by prior empno = mgr );
 18       if ( l_cnt = 0 )
 19       then
 20            insert into audit_tab ( msg )
 21            values ( 'Attempt to update ' || :new.empno );
 22            commit;
 23
```

```
24                 raise_application_error( -20001, 'Access Denied' );

25         end if;

26 end;

27 /

Trigger created.
```

注意，这里使用了 CONNECT BY 查询。这会根据当前用户分析整个（员工）层次结构。它会验证我们试图更新的记录是某个下属员工的记录，即这个人会在某个层次上向我们报告。

关于这个触发器的要点，主要如下：

- PRAGMA AUTONOMOUS_TRANSACTION 应用于触发器定义。整个触发器是一个“自治事务”，因此它独立于父事务（即企图完成更新的事务）。
- 触发器在查询中从它保护的表（EMP 表） 中具体读取。如果这不是一个自治事务，它本身在运行时就会导致一个变异表错误。自治事务使我们绕开了这个问题，它允许我们读取表，但是也带来了一个缺点， 我们无法看到自己对表做的修改。在这种情况下需要特别小心，这个逻辑必须仔细审查。如果我们完成的事务是对员工层次结构本身的一个更新会怎么样？我们不会 在触发器中看到这些修改，在评估触发器的正确性时也要把这考虑在内。
- 触发器提交。这在以前不可能的，触发器以前从来不能提交工作。这个触发器并不是提交父事务的工作（实际触发器触发的工作，即更新员工记录），而只是提交了触发器所完成的工作（审计记录）。

在此，我们建立了 EMP 表，其中一个妥善的层次结构（EMPNO-MGR 递归关系）。另外还有一个 AUDIT_TAB 表，要在其中记录修改信息的失败企图。我们的触发器可以保证这样一个规则：只有我们的经理或经理的经理（依此类推）可以修改我们的记录。

下面尝试在 EMP 表中更新一条记录，来看看这是如何工作的：

```
ops$tkyte@ORA10G> update emp set sal = sal*10;

update emp set sal = sal*10

       *

ERROR at line 1:

ORA-20001: Access Denied

ORA-06512: at "OPS$TKYTE.EMP_AUDIT", line 21
```

```
ORA-04088: error during execution of trigger 'OPS$TKYTE.EMP_AUDIT'



ops$tkyte@ORA10G> select * from audit_tab;



USERNAME TIMESTAMP MSG

--------- --------- ----------------------------------------

OPS$TKYTE 27-APR-05 Attempt to update 7369
```

触发器发现了情况，能防止 UPDATE 发生，而与此同时，会为这个企图创建一个永久记录（注意它在 AUDIT_TAB 表的 CREATE TABLE 语句上如何使用 DEFAULT 关键字来自动插入 USER 和 SYSDATE 值）。接下来，假设我们作为一个用户登录，想实际完成一个 UPDATE，并做一些尝试：

```
ops$tkyte@ORA10G> connect scott/tiger

Connected.

scott@ORA10G> set echo on

scott@ORA10G> update ops$tkyte.emp set sal = sal*1.05 where ename = 'ADAMS';



1 row updated.



scott@ORA10G> update ops$tkyte.emp set sal = sal*1.05 where ename = 'SCOTT';

update ops$tkyte.emp set sal = sal*1.05 where ename = 'SCOTT'

                       *

ERROR at line 1:

ORA-20001: Access Denied

ORA-06512: at "OPS$TKYTE.EMP_AUDIT", line 21

ORA-04088: error during execution of trigger 'OPS$TKYTE.EMP_AUDIT'
```

在演示表 EMP 的默认安装中，员工 ADAMS 是 SCOTT 的下属，所以第一个 UPDATE

成功。再看第二个 UPDATE，SCOTT 试图给自己加薪，但是由于 SCOTT 不向 SCOTT 报告（SCOTT 不是自己的下属），所以这个更新失败了。再登录回到包括 AUDIT_TAB 表的模式，可以看到以下结果：

```
scott@ORA10G> connect /

Connected.

ops$tkyte@ORA10G> set echo on

ops$tkyte@ORA10G> select * from audit_tab;



USERNAME TIMESTAMP         MSG

---------           ---------              ----------------------------------------

OPS$TKYTE 27-APR-05          Attempt to update 7369

SCOTT           27-APR-05          Attempt to update 7788
```

SCOTT 试图完成的这个 UPDATE 已经被记录下来。

8.7 小结

在这一章中，我们了解了 Oracle 事务管理的许多方面。事务是数据库区别于文件系统的主要特性之一。要了解事务如何工作以及如何使用事务，在任何数据库中这对于正确地实现应用都是必要的。要知道，在 Oracle 中，所有语句都具有原子性（包括副作用），而且要知道这种原子性延伸到了存储过程，这一点很重要。我们看到，如果在 PL/SQL 块中放置一个 WHEN OTHERS 异常处理器，可能会显著地影响数据库中发生的改变。作为数据库开发人员，要对事务如何工作有一个很好的了解，这至关重要。

我们介绍了完整性约束（惟一键、检查约束等）与 Oracle 中事务之间的交互，这种交互有些复杂。我们讨论了 Oracle 通常在事务执行之后立即处理完整性约束，但是如果我们愿意，也可以把这个约束验证延迟到事务结束时进行。实现复杂的多表更新时，如果所修改的表彼此相互依赖，这种延迟特性就非常重要，级联更新就是这样一个例子。

如 果人们使用过其他数据库，这些数据库只是“支持”事务而没有“提升”事务的使用，就会沿袭下来一个不好的事务习惯，接下来我们就考虑了这些坏习惯。在此介 绍了事务的根本原则：一方面，事务应该尽可能短（也就是不应该不必要地建立大事务）；另一方面，要根据需要是事务足够大。决定事务大小的关键是数据完整 性，这是本章阐述的一个关键概念。能决定事务大小的惟一因素就是控制系统的业务规则。记住，不是 undo 空间，不是锁，而是业务规则。

我们还介绍了分布式事务以及分布式事务与单个数据库事务有什么区别。另外分析了分布式事务中存在的限制，并讨论了为什么会有这些限制。在建立分布式系统之前，你要了解这些限制，在单实例中能做的事情到了分布式数据库中可能并不能做。

这一章的最后介绍的是自治事务，分析了自治事务是什么，更重要的是，指出了什么时候应该使用自治事务，什么时候不该使用。我想再强调一遍，实际中自治事务的合理使用少之又少。如果发现你把它当作一个特性在经常使用，可能就得花些时间好好看看为什么会这样。

第 9 章 redo 与 undo

这一章将介绍 Oracle 数据库中最重要的两部分数据：redo 与 undo。redo（重做信息）是 Oracle 在在线（或归档）重做日志文件中记录的信息，万一出现失败时可以利用这些数据来“重放”（或重做）事务。undo（撤销信息）是 Oracle 在 undo 段中记录的信息，用于取消或回滚事务。

在这一章中，我们讨论的内容很多，包括 redo 和 undo（回滚信息）如何生成，以及事务、恢复等方面如何应用 redo 和 undo。首先我们给出一个高层概述，说明 undo 和 redo 分别是什么，它们如何协作。然后向下细化，更深入地介绍各个主题，并讨论作为开发人员需要了解哪些内容。

这一章主要面向开发人员，在此没有涵盖应由 DBA 完全负责确定和调整的问题。例如，如何找到 RECOVERY_PARALLELISM 或 FAST_START_MTTR_TARGET 参数的最优设置？这个问题要由 DBA 确定，本章就没有涉及。但是，redo 和 undo 则是 DBA 和开发人员都关心的主题，它们是 DBA 和开发人员之间的桥梁。不论是 DBA 还是开发人员，都需要对 redo 和 undo 的作用有很好的基本了解，知道它们如何工作，并且知道如何避免与 redo 和 undo 的使用有关的潜在问题。如果掌握了 redo 和 undo 的相关知识，这还有助于 DBA 和开发人员更好地理解数据库一般如何操作。

在这一章中，我将针对 Oracle 的 这些机制提供伪代码，并从概念上解释到底会发生什么。这里不会详尽地介绍所有内部细节，如会用哪些数据字节更新哪些文件等详细内容并不会谈到。具体发生的 情况可能比我们介绍的更复杂，但是不管怎样，如果能很好地理解这些机制的工作流程，将很有意义，这有助于理解你的动作会带来怎样的影响。

9.1 什么是 redo？

重做日志文件（redo log file）对 Oracle 数据库来说至关重要。它们是数据库的事务日志。Oracle 维护着两类重做日志文件：在线（online）重做日志文件和归档（archived）重做日志文件。这两类重做日志文件都用于恢复；其主要目的是，万一实例失败或介质失败，它们就能派上用场。

如果数据库所在主机掉电，导致实例失败，Oracle 会使用在线重做日志将系统恰好恢复到掉电之前的那个时间点。如果磁盘驱动器出现故障（这是一个介质失败），Oracle 会 使用归档重做日志以及在线重做日志将该驱动器上的数据备份恢复到适当的时间点。另外，如果你“无意地”截除了一个表，或者删除了某些重要的信息，然后提交 了这个操作，那么可以恢复受影响数据的一个备份，并使用在线和归档重做日志文件把它恢复到这个“意外”发生前的时间点。

归档重做日志文件实际上就是已填满的“旧”在线重做日志文件的副本。系统将日志文件填满时，ARCH 进程会在另一个位置建立在线重做日志文件的一个副本，也可以在本地和远程位置上建立多个另外的副本。如果由于磁盘驱动器损坏或者其他物理故障而导致失败，就会用这些归档重做日志文件来执行介质恢复。Oracle 拿到这些归档重做日志文件，并把它们应用于数据文件的备份，使这些数据文件能“赶上”数据库的其余部分。归档重做日志文件是数据库的事务历史。

注意 随着 Oracle 10g 的到来，我们现在还有了一种闪回技术（flashback）。利用闪回技术，可以执行闪回查询（也就是说，查询过去某个时间点的数据），取消数据库表的删除，将表置回到以前某个时间的状态，等等。因此，现在使用备份和归档重做日志文件来完成传统恢复的情况越来越少。不过，执行恢复是 DBA 最重要的任务，而且 DBA 在数据库恢复方面绝对不能犯错误。

每个 Oracle 数据库都至少有两个在线重做日志组，每个组中至少有一个成员（重做日志文件）。这些在线重做日志组以循环方式使用。Oracle 会写组 1 中的日志文件，等写到组 1 中文件的最后时，将切换到日志文件组 2，开始写这个组中的文件。等到把日志文件组 2 写满时，会再次切换回日志文件组 1（假设只有两个重做日志文件组；如果有 3 个重做日志文件组，Oracle 当然会继续写第 3 个组）。

数据库之所以成为数据库（而不是文件系统等其他事物），是因为它有自己独有的一些特征，重做日志或事务日志就是其中重要的特性之一。重做日志可能是数据库中最重要的恢复结构，不过，如果没有其他部分（如 undo 段、分布式事务恢复等），但靠重做日志什么也做不了。重做日志是数据库区别于传统文件系统的一个主要因素。Oracle 正写到一半的时候有可能发生掉电，利用在线重做日志，我们就能有效地从这个掉电失败中恢复。归档重做日志则允许我们从介质失败中恢复，如硬盘损坏，或者由于人为错误而导致数据丢失。如果没有重做日志，数据库提供 id 保护就比文件系统多不了多少。

9.2 什么是 undo？

从概念上讲，undo 正好与 redo 相对。你对数据执行修改时，数据库会生成 undo 信息，这样万一你执行的事务或语句由于某种原因失败了，或者如果你用一条 ROLLBACK 语句请求回滚，就可以利用这些 undo 信息将数据放回到修改前的样子。redo 用于在失败时重放事务（即恢复事务），undo 则用于取消一条语句或一组语句的作用。与 redo 不同，undo 在数

据库内部存储在一组特殊的段中，这称为 undo 段（undo segment）。

注意 “回滚段”（rollback segment）和“undo 段“（undo segment）一般认为是同义词。使用手动 undo 管理时，DBA 会创建”回滚段“。使用自动 undo 管理时，系统将根据需要自动地创建和销毁”undo 段“。对于这里的讨论来说，这些词的意图和作用都一样。

通常对 undo 有一个误解，认为 undo 用 于数据库物理地恢复到执行语句或事务之前的样子，但实际上并非如此。数据库只是逻辑地恢复到原来的样子，所有修改都被逻辑地取消，但是数据结构以及数据库 块本身在回滚后可能大不相同。原因在于：在所有多用户系统中，可能会有数十、数百甚至数千个并发事务。数据库的主要功能之一就是协调对数据的并发访问。也 许我们的事务在修改一些块，而一般来讲往往会有许多其他的事务也在修改这些块。因此，不能简单地将一个块放回到我们的事务开始前的样子，这样会撤销其他人 （其他事务）的工作！

例如，假设我们的事务执行了一个 INSERT 语句，这条语句导致分配一个新区段（也就是说，导致表的空间增大）。通过执行这个 INSET，我们将得到一个新的块，格式化这个块以便使用，并在其中放上一些数据。此时，可能出现另外某个事务，它也向这个块中插入数据。如果要回滚我们的事务，显然不能取消对这个块的格式化和空间分配。因此，Oracle 回滚时，它实际上会做与先前逻辑上相反的工作。对于每个 INSERT，Oracle 会完成一个 DELETE。对于每个 DELETE，Oracle 会执行一个 INSERT。对于每个 UPDATE，Oracle 则会执行一个“反 UPDATE“，或者执行另一个 UPDATE 将修改前的行放回去。

注意 这种 undo 生成对于直接路径操作（direct path operation）不适用，直接路径操作能够绕过表上的 undo 生成。稍后将更详细地讨论这些问题。

怎么才能看到 undo 生成（undo generation）具体是怎样的呢？也许最容易的方法就是遵循以下步骤：

（1） 创建一个空表。

（2） 对它做一个全部扫描，观察读表所执行的 I/O 数量。

（3） 在表中填入许多行（但没有提交）。

（4） 回滚这个工作，并撤销。

（5） 再次进行全表扫描，观察所执行的 I/O 数量。

首先，我们创建一个空表：

```
ops$tkyte@ORA10G> create table t
  2 as
  3 select *
```

```
  4 from all_objects

  5 where 1=0;

Table created.
```

然后查询这个表，这里在 SQL*Plus 中启用了 AUTOTRACE，以便能测试 I/O。

注意 在这个例子中，每次（即每个用例）都会做两次全表扫描。我们的目标只是测试每个用例中第二次完成的 I/O。这样可以避免统计在解析和优化期间优化器可能完成的额外 I/O。

最初，这个查询需要 3 个 I/O 来完成这个表的全表扫描：

```
ops$tkyte@ORA10G> select * from t;

no rows selected



ops$tkyte@ORA10G> set autotrace traceonly statistics

ops$tkyte@ORA10G> select * from t;

no rows selected



Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets

3 consistent gets

...

ops$tkyte@ORA10G> set autotrace off
```

接下来，向表中增加大量数据。这会使它“扩大“，不过随后再将其回滚：

```
ops$tkyte@ORA10G> insert into t select * from all_objects;

48350 rows created.
```

```
ops$tkyte@ORA10G> rollback;

Rollback complete.
```

现在，如果再次查询这个表，会发现这一次读表所需的 I/O 比先前多得多：

```
ops$tkyte@ORA10G> select * from t;

no rows selected



ops$tkyte@ORA10G> set autotrace traceonly statistics

ops$tkyte@ORA10G> select * from t;

no rows selected



Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets

689 consistent gets

...

ops$tkyte@ORA10G> set autotrace off
```

前面的 INSERT 导致将一些块增加到表的高水位线（high-water mark，HWM）之下，这些块没有因为回滚而消失，它们还在那里，而且已经格式化，只不过现在为空。全表扫描必须读取这些块，看看其中是否包含行。这说明，回滚只是一个“将数据库还原“的逻辑操作。数据库并非真的还原成原来的样子，只是逻辑上相同而已。

9.2.1 redo 和 undo 如何协作？

在这一节中，我们来看看在不同场景中 redo 和 undo 如何协作。例如，我们会讨论处理 INSERT 时关于 redo 和 undo 生成会发生什么情况，另外如果在不同时间点出现失败，Oracle 将如何使用执行信息。

有意思的是，尽管 undo 信息存储在 undo 表空间或 undo 段中，但也会受到 redo 的保护。

换句话说，会把 undo 数据当成是表数据或索引数据一样，对 undo 的修改会生成一些 redo，这些 redo 将计入日志。为什么会这样呢？稍后在讨论系统崩溃时发生的情况时将会解释它，到时你会明白了。将 undo 数据增加到 undo 段中，并像其他部分的数据一样在缓冲区缓存中得到缓存。

INSERT-UPDATE-DELETE 示例场景

作为一个例子，我们将分析对于下面这组语句可能发生什么情况：

```
insert into t (x,y) values (1,1);

update t set x = x+1 where x = 1;

delete from t where x = 2;
```

我们会沿着不同的路径完成这个事务，从而得到以下问题的答案：

- 如果系统在处理这些语句的不同时间点上失败，会发生什么情况？
- 如果在某个时间点上 ROLLBACK，会发生什么情况？
- 如果成功并 COMMIT，会发生什么情况？

1. INSERT

对于第一条 INSERT INTO T 语句，redo 和 undo 都会生成。所生成的 undo 信息足以使 INSERT“消失“。INSERT INTO T 生成的 redo 信息则足以让这个插入”再次发生“。

插入发生后，系统状态如图 9-1 所示。

图 9-1 INSERT 之后的系统状态

这里缓存了一些已修改的 undo 块、索引块和表数据块。这些块得到重做日志缓冲区中相应条目的“保护“。

- **假想场景：系统现在崩溃**

即使系统现在崩溃也没有关系。SGA 会被清空，但是我们并不需要 SGA 里的任何内容。重启动时就好像这个事务从来没有发生过一样。没有将任何已修改的块刷新输出到磁盘，也没有任何 redo 刷新输出到磁盘。我们不需要这些 undo 或 redo 信息来实现实例失败恢复。

□ **假想场景：缓冲区缓存现在已满**

在这种情况下，DBWR 必须留出空间，要把已修改的块从缓存刷新输出。如果是这样，DBWR 首先要求 LGWR 将保护这些数据库块的 redo 条目刷新输出。DBWR 将任何有修改的块写至磁盘之前，LGWR 必须先刷新输出与这些块相关的 redo 信息。这是有道理的——如果我们要刷新输出表 T 中已修改的块，但没有刷新输出与 undo 块关联的 redo 条目，倘若系统失败了，此时就会有一个已修改的表 T 块，而没有与之相关的 redo 信息。在写出这些块之前需要先刷新输出重做日志缓存区，这样就能重做（重做）所有必要的修改，将 SGA 放回到现在的状态，从而能发生回滚。

从第二个场景还可以预见到一些情况。这里描述的条件是“如果刷新输出了表 T 的块，但没有刷新输出 undo 块的相应 redo，而且此时系统失败了“，这个条件开始变得有些复杂。随着增加更多用户、更多的对象，再加上并发处理等因素，条件还会更复杂。

此时的情况如图 9-1 所示。我们生成了一些已修改的表和索引块。这些块有一些与之关联的 undo 段块，这 3 类块都会生成 redo 来保护自己。如果还记得第 4 章中对重做日志缓冲区的讨论，应该知道，它会在以下情况刷新输出：每 3 秒一次；缓冲区 1/3 满时或者包含了 1MB 的缓冲数据；或者是只要发生提交就会刷新输出。重做日志缓冲区还有可能会在处理期间的某一点上刷新输出。在这种情况下，其状态如图 9-2 所示。

图 9-2　重做日志缓冲区刷新输出后的系统状态

2. UPDATE

UPDATE 所带来的工作与 INSERT 大体一样。不过 UPDATE 生成的 undo 量更大；由于存在更新，所以需要保存一些“前“映像。系统状态如图 9-3 所示。

图 9-3　UPDATE 后的系统状态

块缓冲区缓存中会有更多新的 undo 段块。为了撤销这个更新，如果必要，已修改的数据库表和索引块也会放在缓存中。我们还生成了更多的重做日志缓存区条目。下面假设前面的插入语句生成了一些重做日志，其中有些重做日志已经刷新输出到磁盘上，有些还放在缓存中。

□　**假想场景：系统现在崩溃**

启动时，Oracle 会读取重做日志，发现针对这个事务的一些重做日志条目。给定系统的当前状态，利用重做日志文件中对应插入的 redo 条目，并利用仍在缓冲区中对应插入的 redo 信息，Oracle 会“前滚”插入。最后到与图 9-1 类似的状态。现在有一些 undo 块（用以撤销插入）、已修改的表块（刚插入后的状态），以及已修改的索引块（刚插入后的状态）。由于系统正在进行崩溃恢复，而且我们的会话还不再连接（这是当然），Oracle 发现这个事务从未提交，因此会将其回滚。它取刚刚在缓冲区缓存中前滚得到的 undo，并将这些 undo 应用到数据和索引块，使数据和索引块“恢复”为插入发生前的样子。现在一切都回到从前。磁盘上的块可能会反映前面的 INSERT，也可能不反映（这取决于在崩溃前是否已经将块刷新输出）。如果磁盘上的块确实反映了插入，而实际上现在插入已经被撤销，当从缓冲区缓存刷新输出块时，数据文件就会反映出插入已撤销。如果磁盘上的块本来就没有反映前面的插入，就不用去管它——这些块以后肯定会被覆盖。

这个场景涵盖了崩溃恢复的基本细节。系统将其作为一个两步的过程来完成。首先前滚，把系统放到失败点上，然后回滚尚未提交的所有工作。这个动作会再次同步数据文件。它会重放已经进行的工作，并撤销尚未完成的所有工作。

□　**假想场景：应用回滚事务**

此时，Oracle 会发现这个事务的 undo 信息可能在缓存的 undo 段块中（基本上是这样），也可能已经刷新输出到磁盘上（对于非常大的事务，就往往是这种情况）。它会把 undo 信息应用到缓冲区缓存中的数据和索引块上，或者倘若数据和索引块已经不在缓存中，则要从磁盘将数据和索引块读入缓存，再对其应用 undo。这些块会恢复为其原来的行值，并刷新输出到数据文件。

这个场景比系统崩溃更常见。需要指出，有一点很有用：回滚过程中从不涉及重做日志。只有恢复和归档时会当前重做日志。这对于调优是一个很重要的概念：重做日志是用来写的（而不是用于读）。Oracle 不会在正常的处理中读取重做日志。只要你有足够的设备，使得 ARCH 读文件时，LGWR 能写到另一个不同的设备，那么就不存在重做日志竞争。许多其他的数据库（非 Oracle）都把日志文件处理为“事务日志”。这些数据库没有把 redo 和 undo 分开。对于这些系统，回滚可能是灾难性的，回滚进程必须读取日志，而日志写入器正在试图写这个日志。这就向系统中最薄弱的环节引入了竞争。Oracle 的目标是：可以顺序地写日志，而且在写日志时别人不会读日志。

3. DELETE

同样，DELETE 会生成 undo，块将被修改，并把 redo 发送到重做日志缓冲区。这与前面没有太大的不同。实际上，它与 UPDATE 如此类似，所以我们不再啰嗦，直接来介绍 COMMIT。

4. COMMIT

我们已经看到了多种失败场景和不同的路径，现在终于到 COMMIT 了。在此，Oracle 会把重做日志缓冲区刷新输出到磁盘，系统状态如图 9-4 所示。

图 9-4 COMMIT 后的系统状态

已修改的块放在缓冲区缓存中；可能有一些块已经刷新输出到磁盘上。重做这个事务所需的全部 redo 都安全地存放在磁盘上，现在修改已经是永久的了。如果从数据文件直接读取数据，可能会看到块还是事务发生前的样子，因为很有可能 DBWR 还没有（从缓冲区缓存）写出这些块。这没有关系，如果出现失败，可以利用重做日志文件来得到最新的块。undo 信息会一直存在，除非 undo 段回绕重用这些 undo 块。如果某些对象受到影响，Oracle 会使用这个 undo 信息为需要这些对象的会话提供对象的一致读。

9.3 提交和回滚处理

有一点很重要，我们要知道重做日志文件对开发人员有什么影响。下面介绍编写代码的不同方法会对重做日志的利用有怎样的影响。在本章前面已经了解了 redo 的原理，接下

来介绍一些更特定的问题。作为开发人员，你能检测到许多这样的场景，但是它们要由 DBA 来修正，因为这些场景会影响整个数据库实例。我们先来介绍 COMMIT 期间会发生什么，然后讨论有关在线重做日志的一些经常被问到的问题。

9.3.1 COMMIT 做什么？

作为一名开发人员，你应该深入了解 COMMIT 期间会做些什么。在这一节中，我们将分析 Oracle 中处理 COMMIT 语句期间发生的情况。COMMIT 通常是一个非常快的操作，而不论事务大小如何。

你可能认为，一个事务越大（换句话说，它影响的数据越多），COMMIT 需要的时间就越长。不是这样的。不论事务有多大，COMMIT 的响应时间一般都很“平”（flat，可以理解为无高低变化）。这是因为 COMMIT 并没有太多的工作去做，不过它所做的确实至关重要。

这 一点很重要，之所以要了解并掌握这个事实，原因之一是：这样你就能心无芥蒂地让事务有足够的大小。在上一章曾经讨论过，许多开发人员会人为地限制事务的大 小，分别提交太多的行，而不是一个逻辑工作单元完成后才提交。这样做主要是出于一种错误的信念，即认为可以节省稀有的系统资源，而实际上这只是增加了资源 的使用。如果一行的 COMMIT 需要 X 个时间单位，1,000 次 COMMIT 也同样需要 X 个时间单位，倘若采用以下方式执行工作，即每行提交一次共执行 1,000 次 COMMIT，就会需要 1000*X 各时间单位才能完成。如果只在必要时才提交（即逻辑工作单元结束时），不仅能提高性能，还能减少对共享资源的竞争（日志文件、各种内部闩等）。通过一个简单的例子就能展示出过多的提交要花费更长的时间。这里将使用一个 Java 应用，不过对于大多数其他客户程序来说，结果可能都是类似的，只有 PL/SQL 除外（在这个例子后面，我们将讨论为什么会这样）。首先，下面是我们要插入的示例表：

```
scott@ORA10G> desc test

Name                  Null?            Type

----------------      --------          ------------

ID                                      NUMBER

CODE                                    VARCHAR2(20)

DESCR                                   VARCHAR2(20)

INSERT_USER                            VARCHAR2(30)

INSERT_DATE                             DATE
```

Java 程序要接受两个输入：要插入（INSERT）的行数（iters），以及两次提交之间插入的行数（commitCnt）。它先连接到数据库，将 autocommit（自动提交）设置为 off（所有 Java 代码都应该这么做），然后将 doInserts()方法共调用 3 次：

- 第一次调用只是“热身”（确保所有类都已经加载）。

- 第二次调用指定了要插入（INSERT）的行数，并指定一次提交多少行（即每 N 行提交一次）。
- 最后一次调用将要插入的行数和一次提交的行数设置为相同的值（也就是说，所有行都插入之后才提交）。

然后关闭连接，并退出。其 main 方法如下：

```
import java.sql.*;

import oracle.jdbc.OracleDriver;

import java.util.Date;

public class perftest

{

        public static void main (String arr[]) throws Exception

        {

                DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

                Connection con = DriverManager.getConnection

                        ("jdbc:oracle:thin:@localhost.localdomain:1521:ora10g",

                        "scott", "tiger");

                Integer iters = new Integer(arr[0]);

                Integer commitCnt = new Integer(arr[1]);

                on.setAutoCommit(false);

                doInserts( con, 1, 1 );

                doInserts( con, iters.intValue(), commitCnt.intValue() );

                doInserts( con, iters.intValue(), iters.intValue() );

                con.commit();

                con.close();

}
```

现在，doInserts()方法相当简单。首先准备（解析）一条 INSERT 语句，以便多次反复绑定/执行这个 INSERT：

```
static void doInserts(Connection con, int count, int commitCount )

throws Exception

{

    PreparedStatement ps =

        con.prepareStatement

        ("insert into test " +

        "(id, code, descr, insert_user, insert_date)"

        + " values (?,?,?, user, sysdate)");
```

然后根据要插入的行数循环，反复绑定和执行这个 INSERT。另外，它会检查一个行计数器，查看是否需要 COMMIT，或者是否已经不在循环范围内。还要注意，我们分别在循环之前和循环之后获取了当前时间，从而监视并报告耗用的时间：

```
    int rowcnt = 0;

    int committed = 0;

    long start = new Date().getTime();

    for (int i = 0; i < count; i++ )

    {

        ps.setInt(1,i);

        ps.setString(2,"PS - code" + i);

        ps.setString(3,"PS - desc" + i);

        ps.executeUpdate();

        rowcnt++;

        if ( rowcnt == commitCount )

        {
```

```
                        con.commit();
                        rowcnt = 0;
                        committed++;
                    }
                }
                con.commit();
                long end = new Date().getTime();
                System.out.println
                    ("pstatement " + count + " times in " +
                    (end - start) + " milli seconds committed = "+committed);
        }
}
```

下面根据不同的输入发放运行这个代码：

```
$ java perftest 10000 1
pstatement 1 times in 4 milli seconds committed = 1
pstatement 10000 times in 11510 milli seconds committed = 10000
pstatement 10000 times in 2708 milli seconds committed = 1

$ java perftest 10000 10
pstatement 1 times in 4 milli seconds committed = 1
pstatement 10000 times in 3876 milli seconds committed = 1000
pstatement 10000 times in 2703 milli seconds committed = 1
```

```
$ java perftest 10000 100

pstatement 1 times in 4 milli seconds committed = 1

pstatement 10000 times in 3105 milli seconds committed = 100

pstatement 10000 times in 2694 milli seconds committed = 1
```

可以看到，提交得越多，花费的时间就越长（你的具体数据可能与这里报告的不同）。这只是单用户的情况，如果有多个用户在做同样的工作，所有这些用户都过于频繁地提交，那么得到的数字将飞速增长。

在其他类似的情况下，我们也不止一次地听到过同样的“故事”。例如，我们已经知道，如果不使用绑定变量，而且频繁地完成硬解析，这会严重地降低并发性，原因是存在库缓存竞争和过量的 CPU 占用。即使转而使用绑定变量，如果过于频繁地软解析，也会带来大量的开销（导致过多软解析的原因可能是：执意地关闭游标，尽管稍后就会重用这些游标）。必须在必要时才完成操作，COMMIT 就是这样的一种操作。最好根据业务需求来确定事务的大小，而不是错误地为了减少数据库上的资源使用而“压缩”事务。

在这个例子中，COMMIT 的开销存在两个因素：

- 显然会增加与数据库的往返通信。如果每个记录都提交，生成的往返通信量就会大得多。
- 每次提交时，必须等待 redo 写至磁盘。这会导致“等待”。在这种情况下，等待称为“日志文件同步”（log file sync）。
- 只需对这个 Java 应用稍做修改就可以观察到后面这一条。我们将做两件事情：
- 增加一个 DBMS_MONITOR 调用，启用对等待事件的 SQL 跟踪。在 Oracle9i 中，则要使用 alter session set events ‘10046 trace name context forever, level 12’，因为 DBMS_MONITOR 是 Oracle 10g 中新增的。
- 把 con.commit()调用改为一条完成提交的 SQL 语句调用。如果使用内置的 JDBC commit()调用，这不会向跟踪文件发出 SQL COMMIT 语句，而 TKPROF（用于格式化跟踪文件的工具）也不会报告完成 COMMIT 所花费的时间。

因此，我们将 doInserts()方法修改如下：

```
  doInserts( con, 1, 1 );

  Statement stmt = con.createStatement ();

  stmt.execute

  ( "begin dbms_monitor.session_trace_enable(waits=>TRUE); end;" );

  doInserts( con, iters.intValue(), iters.intValue() );
```

对于 main 方法，要增加以下代码：

```
PreparedStatement commit =
    con.prepareStatement
    ("begin /* commit size = " + commitCount + " */ commit; end;" );
  int rowcnt = 0;
  int committed = 0;
  ...
if ( rowcnt == commitCount )
  {
    commit.executeUpdate();
      rowcnt = 0;
      committed++;
```

如果运行这个应用来插入 10,000 行，每行提交一次，TKPROF 报告显示的结果如下：

```
begin /* commit size = 1 */ commit; end;
....
Elapsed times include waiting on following events:
  Event waited on                          Times          Max. Wait          Total Waited
  ----------------------------------------  Waited          ----------          ------------
  SQL*Net message to client                10000          0.00                0.01
  SQL*Net message from client              10000          0.00                0.04
  log file sync                             8288          0.06                2.00
```

如果还是插入 10,000 行，但是只在插入了全部 10,000 行时才提交，就会得到如下的结果：

```
begin /* commit size = 10000 */ commit; end;
....
```

```
Elapsed times include waiting on following events:
  Event waited on                          Times           Max. Wait             Total Waited
  ----------------------------------------  Waited          ----------            ------------
-
  log file sync                             1               0.00                  0.00
  SQL*Net message to client                 1               0.00                  0.00
  SQL*Net message from client               1               0.00                  0.00
```

如果在每个 INSERT 之后都提交，几乎每次都要等待。尽管每次只等待很短的时间，但是由于经常要等待，这些时间就会累积起来。运行时间中整整 2 秒都用于等待 COMMIT 完成，换句话说，等待 LGWR 将 redo 写至磁盘。与之形成鲜明对比，如果只提交一次，就不会等待很长时间（实际上，这个时间实在太短了，以至于简直无法度量）。这说明，COMMIT 是一个很快的操作；我们希望响应时间或多或少是“平”的，而不是所完成工作量的一个函数。

那么，为什么 COMMIT 的响应时间相当“平”，而不论事务大小呢？在数据库中执行 COMMIT 之前，困难的工作都已经做了。我们已经修改了数据库中的数据，所以 99.9%的工作都已经完成。例如，已经发生了以下操作：

- 已经在 SGA 中生成了 undo 块。
- 已经在 SGA 中生成了已修改数据块。
- 已经在 SGA 中生成了对于前两项的缓存 redo。
- 取决于前三项的大小，以及这些工作花费的时间，前面的每个数据（或某些数据）可能已经刷新输出到磁盘。
- 已经得到了所需的全部锁。

执行 COMMIT 时，余下的工作只是：

- 为事务生成一个 SCN。如果你还不熟悉 SCN，起码要知道，SCN 是 Oracle 使用的一种简单的计时机制，用于保证事务的顺序，并支持失败恢复。SCN 还用于保证数据库中的读一致性和检查点。可以把 SCN 看作一个钟摆，每次有人 COMMIT 时，SCN 都会增 1.
- LGWR 将所有余下的缓存重做日志条目写到磁盘，并把 SCN 记录到在线重做日志文件中。这一步就是真正的 COMMIT。如果出现了这一步，即已经提交。事务条目会从 V$TRANSACTION 中“删除”，这说明我们已经提交。
- V$LOCK 中记录这我们的会话持有的锁，这些所都将被释放，而排队等待这些锁的每一个人都会被唤醒，可以继续完成他们的工作。
- 如果事务修改的某些块还在缓冲区缓存中，则会以一种快速的模式访问并“清理”。块清除（Block cleanout）是指清除存储在数据库块首部的与锁相关的信息。

实质上讲，我们在清除块上的事务信息，这样下一个访问这个块的人就不用再这么做了。我们采用一种无需生成重做日志信息的方式来完成块清除，这样可以省去以后的大量工作（在下面的“块清除”一节中将更全面地讨论这个问题）。

可以看到，处理 COMMIT 所要做的工作很少。其中耗时最长的操作要算 LGWR 执行的活动（一般是这样），因为这些磁盘写是物理磁盘 I/O。不过，这里 LGWR 花费的时间并不会太多，之所以能大幅减少这个操作的时间，原因是 LGWR 一直在以连续的方式刷新输出重做日志缓冲区的内容。在你工作期间，LGWR 并非缓存这你做的所有工作；实际上，随着你的工作的进行，LGWR 会在后台增量式地刷新输出重做日志缓冲区的内容。这样做是为了避免 COMMIT 等待很长时间来一次性刷新输出所有的 redo。

因此，即使我们有一个长时间运行的事务，但在提交之前，它生成的许多缓存重做日志已经刷新输出到磁盘了（而不是全部等到提交时才刷新输出）。这也有不好的一面，COMMIT 时，我们必须等待，直到尚未写出的所有缓存 redo 都已经安全写到磁盘上才行。也就是说，对 LGWR 的调用是一个同步（synchronous）调用。尽管 LGWR 本身可以使用异步 I/O 并行地写至日志文件，但是我们的事务会一直等待 LGWR 完成所有写操作，并收到数据都已在磁盘上的确认才会返回。

前面我提高过，由于某种原因，我们用的是一个 Java 程序而不是 PL/SQL，这个原因就是 PL/SQL 提供了提交时优化（commit-time optimization）。我说过，LGWR 是一个同步调用，我们要等待它完成所有写操作。在 Oracle 10g Release 1 及以前版本中，除 PL/SQL 以外的所有编程语言都是如此。PL/SQL 引擎不同，要认识到直到 PL/SQL 例程完成之前，客户并不知道这个 PL/SQL 例程中是否发生了 COMMIT，所以 PL/SQL 引擎完成的是异步提交。它不会等待 LGWR 完成；相反，PL/SQL 引擎会从 COMMIT 调用立即返回。不过，等到 PL/SQL 例程完成，我们从数据库返回客户时，PL/SQL 例程则要等待 LGWR 完成所有尚未完成的 COMMIT。因此，如果在 PL/SQL 中提交了 100 次，然后返回客户，会发现由于存在这种优化，你只会等待 LGWR 一次，而不是 100 次。这是不是说可以在 PL/SQL 中频繁地提交呢？这是一个很好或者不错的主意吗？不是，绝对不是，在 PL/SQ;中频繁地提交与在其他语言中这样做同样糟糕。指导原则是，应该在逻辑工作单元完成时才提交，而不要在此之前草率地提交。

注意 如果你在执行分布式事务或者以最大可能性模式执行 Data Guard，PL/SQL 中的这种提交时优化可能会被挂起。因为此时存在两个参与者，PL/SQL 必须等待提交确实完成后才能继续。

为了说明 COMMIT 是一个“响应时间很平”的操作，下面将生成不同大小的 redo，并测试插入（INSERT）和提交（COMMIT）的时间。为此，还是在 SQL*Plus 中使用 AUTOTRACE。首先创建一个大表（要把其中的测试数据插入到另一个表中），再创建一个空表：

```
ops$tkyte@ORA10G> @big_table 100000

ops$tkyte@ORA10G> create table t as select * from big_table where 1=0;

Table created.
```

然后在 SQL*Plus 中运行以下命令：

```
ops$tkyte@ORA10G> set timing on

ops$tkyte@ORA10G> set autotrace on statistics;

ops$tkyte@ORA10G> insert into t select * from big_table where rownum <= 10;

ops$tkyte@ORA10G> commit;
```

在此监视 AUTOTRACE 提供的 redo size（redo 大小）统计，并通过 set timing on 监视计时信息。我执行了这个测试，并尝试插入不同数目的行（行数从 10 到 100,000，每次增加一个数量级）。表 9-1 显示了我的观察结果。

表 9-1　随事务大小得到的提交时间*

插入行数	插入时间（秒）	redo 大小（字节）	提交时间（秒）
10	0.05	116	0.06
100	0.08	3,594	0.04
1,000	0.07	372,924	0.06
10,000	0.25	3,744,620	0.06
100,000	1.94	37,843,108	0.07

*这个测试在一个单用户主机上完成，这个主机有一个 8MB 的日志缓冲区和两个 512MB 的在线重做日志文件。

可以看到，使用一个精确度为百分之一秒的计数器度量时，随着生成不同数量的 redo（从 116 字节到 37MB），却几乎测不出 COMMIT 时间的差异。在我们处理和生成重做日志时，LGWR 也没有闲着，它在后台不断地将缓存的重做信息刷新输出到磁盘上。所以，我们生成 37MB 的重做日志信息时，LGWR 一直在忙着，可能每 1MB 左右刷新输出一次。等到 COMMIT 时，剩下的重做日志信息（即尚未写出到磁盘的 redo）已经不多了，可能与创建 10 行数据生成的重做日志信息相差无几。不论生成了多少 redo，结果应该是类似的（但可能不完全一样）。

9.3.2 ROLLBACK 做什么？

把 COMMIT 改为 ROLLBACK，可能会得到完全不同的结果。回滚时间绝对是所修改数据量的一个函数。修改上一节中的脚本，要求完成一个 ROLLBACK（只需把 COMMIT 改成 ROLLBACK），计时信息将完全不同（见表 9-2）。

表 9-2　随事务大小得到的回滚时间

插入行数	回滚时间（秒）	提交时间（秒）
10	0.04	0.06

100	0.05	0.04
1,000	0.06	0.06
10,000	0.22	0.06
100,000	1.6	0.07

这是可以想见的，因为 ROLLBACK 必须物理地撤销我们所做的工作。类似于 COMMIT，必须完成一系列操作。在到达 ROLLBACK 之前，数据库已经做了大量的工作。再复习一遍，可能已经发生的操作如下：

- 已经在 SGA 中生成了 undo 块。
- 已经在 SGA 中生成了已修改数据块。
- 已经在 SGA 中生成了对于前两项的缓存 redo。
- 取决于前三项的大小，以及这些工作花费的时间，前面的每个数据（或某些数据）可能已经刷新输出到磁盘。
- 已经得到了所需的全部锁。

ROLLBACK 时，要做以下工作：

- 撤销已做的所有修改。其完成方式如下：从 undo 段读回数据，然后实际上逆向执行前面所做的操作，并将 undo 条目标记为已用。如果先前插入了一行，ROLLBACK 会将其删除。如果更新了一行，回滚就会取消更新。如果删除了一行，回滚将把它再次插入。
- 会话持有的所有锁都将释放，如果有人在排队等待我们持有的锁，就会被唤醒。

与此不同，COMMIT 只是将重做日志缓冲区中剩余的数据刷新到磁盘。与 ROLLBACK 相比，COMMIT 完成的工作非常少。这里的关键是，除非不得已，否则不会希望回滚。回滚操作的开销很大，因为你花了大量的时间做工作，还要花大量的时间撤销这些工作。除非你有把握肯定会 COMMIT 你的工作，否则干脆什么也别做。听上去这好像是一个常识，这是当然的了，既然不想 COMMIT，又何苦去做所有这些工作！不过，我经常看到这样一些情况：开发人员使用一个“真正”的表作为临时表，在其中填入数据，得到这个表的报告，如何回滚，并删除表中的临时数据。下一节我会讨论真正的临时表，以及如何避免这个问题。

9.4 分析 redo

作为一名开发人员，应该能够测量你的操作生成了多少 redo，这往往很重要。生成的 redo 越多，你的操作花费的时间就越长，整个系统也会越慢。你不光在影响你自己的会话，还会影响每一个会话。redo 管理是数据库中的一个串行点。任何 Oracle 实例都只有一个 LGWR，最终所有事务都会归于 LGWR，要求这个进程管理它们的 redo，并 BOMMIT 其事务，LGWR 要做的越多，系统就会越慢。通过查看一个操作会生成多少 redo，并对一个问题的多种解决方法进行测试，可以从中找出最佳的方法。

9.4.1 测量 redo

要查看生成的 redo 量相当简单，这在本章前面已经见过。我使用了 SQL*Plus 的内置特性 AUTOTRACE。不过 AUTOTRACE 只能用于简单的 DML，对其他操作就力所不能及了，例如，它无法查看一个存储过程调用做了什么。为此，我们需要访问两个动态性能视图：

- V$MYSTAT，其中有会话的提交信息。
- V$STATNAME，这个视图能告诉我们 V$MYSTAT 中的每一行表示什么（所查看的统计名）。

因为我经常要做这种测量，所以使用了两个脚本，分别为 mystat 和 mystat2。mystat.sql 脚本把我感兴趣的统计初始值（如 redo 大小）保存在一个 SQL*Plus 变量中：

```
set verify off

column value new_val V

define S="&1"



set autotrace off

select a.name, b.value

from v$statname a, v$mystat b

where a.statistic# = b.statistic#

and lower(a.name) like '%' || lower('&S')||'%'

/
```

mystat2.sql 脚本只是打印出该统计的初始值和结束值之差：

```
set verify off

select a.name, b.value V, to_char(b.value-&V,'999,999,999,999') diff

from v$statname a, v$mystat b

where a.statistic# = b.statistic#

and lower(a.name) like '%' || lower('&S')||'%'

/
```

下面，可以测量一个给定事务会生成多少 redo。我们只需这样做：

```
@mystat "redo size"

...process...

@mystat2
```

例如：

```
ops$tkyte@ORA10G> @mystat "redo size"

NAME                                 VALUE

-----------------------------        ----------

redo size                            496



ops$tkyte@ORA10G> insert into t select * from big_table;

100000 rows created.



ops$tkyte@ORA10G> @mystat2

NAME                          V                      DIFF

------------------------------        ----------          ----------------

redo size                     37678732               37,678,236
```

如上所示，这个 INSERT 生成了大约 37MB 的 redo。你可能想与一个直接路径 INSERT 生成的 redo 做个比较，如下：

注意 这一节的例子在一个 NOARCHIVELOG 模式的数据库上执行。如果你的数据库采用 ARCHIVELOG 模式，要想观察到这种显著的差异，必须把表置为 NOLOGGING。稍后的“SQL 中设置 NOLOGGING”一节中会更详细地分析 NOLOGGING 属性。不过，在一个“实际”的系统上，对于所有“非日志”（nonlogged）操作，一定要与你的 DBA 协调好。

```
ops$tkyte@ORA10G> @mystat "redo size"

NAME                                 VALUE

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

```
redo size                                 37678732



ops$tkyte@ORA10G> insert /*+ APPEND */ into t select * from big_table;

100000 rows created.



ops$tkyte@ORA10G> @mystat2

ops$tkyte@ORA10G> set echo off

NAME                            V                       DIFF

------------------------------  ----------              ----------------

redo size                       37714328                35,596
```

以上方法使用了 V$MYSTAT 视图，这个方法对于查看各个选项的副作用通常很有用。mystat.sql 脚本适用于有一两个操作的小测试，但是如果我们想完成很大的一系列测试呢？在此可以用到一个很小的测试工具。在下一节中，我们将建立和使用这个测试工具，并利用一个表来记录我们的结果，从而分析 BEFORE 触发器生成的 redo。

9.4.2 redo 生成和 BEFORE/AFTER 触发器

经常有人问我：“除了可以在 BEFORE 触发器中修改一行的值外，BEFORE 和 AFTER 触发器之间还有没有其他的区别？”嗯，对于这个问题，答案是当然有。BEFORE 触发器要额外的 redo 信息，即使它根本没有修改行中的任何值。实际上，这是一个很有意思的案例研究，使用上一节介绍的技术，我们会发现：

- BEFORE 或 AFTER 触发器不影响 DELETE 生成的 redo。
- 在 Oracle9i Release 2 及以前版本中，BEFORE 或 AFTER 触发器会使 INSERT 生成同样数量的额外 redo。在 Oracle 10g 中，则不会生成任何额外的 redo。
- 在 Oracle9i Release 2 及以前的所有版本中，UPDATE 生成的 redo 只受 BEFORE 触发器的影响。AFTER 触发器不会增加任何额外的 redo。不过，在 Oracle 10g 中，情况又有所变化。具体表现为：
  - 总的来讲，如果一个表没有触发器，对其更新期间生成的 redo 量总是比 Oracle9i 及以前版本中要少。看来这是 Oracle 着力解决的一个关键问题：对于触发器的表，要减少这种表更新所生成的 redo 量。
  - 在 Oracle 10g 中，如果表有一个 BEFORE 触发器，则其更新期间生成的 redo 量比 9i 中更大。
  - 如果表有 AFTER 触发器，则更新所生成的 redo 量与 9i 中一样。

为了完成这个测试，我们要使用一个表 T，定义如下：

```
create table t ( x int, y char(N), z date );
```

但是，创建时 N 的大小是可变的。在这个例子中，将使用 N=30、100、500、1,000 和 2,000 来得到不同宽度的行。针对不同大小的 Y 列运行测试，再来分析结果。我使用了一个很小的日志表来保存多次运行的结果：

```
create table log ( what varchar2(15), -- will be no trigger, after or before

        op varchar2(10), -- will be insert/update or delete

        rowsize int, -- will be the size of Y

        redo_size int, -- will be the redo generated

        rowcnt int ) -- will be the count of rows affected
```

这里使用以下 DO_WORK 存储过程来生成事务，并记录所生成的 redo。子过程 REPORT 是一个本地过程（只在 DO_WORK 过程中可见），它只是在屏幕上报告发生了什么，并把结果保存到 LOG 表中：

```
ops$tkyte@ORA10G> create or replace procedure do_work( p_what in varchar2 )
  2  as
  3          l_redo_size number;
  4          l_cnt number := 200;
  5
  6          procedure report( l_op in varchar2 )
  7          is
  8          begin
  9                  select v$mystat.value-l_redo_size
 10                          into l_redo_size
 11                  from v$mystat, v$statname
 12                  where v$mystat.statistic# = v$statname.statistic#
 13                          and v$statname.name = 'redo size';
```

```
14
15              dbms_output.put_line(l_op || ' redo size = ' || l_redo_size ||
16                      ' rows = ' || l_cnt || ' ' ||
17                      o_char(l_redo_size/l_cnt,'99,999.9') ||
18                      ' bytes/row' );
19              insert into log
20                      select p_what, l_op, data_length, l_redo_size, l_cnt
21                       from user_tab_columns
22                      where table_name = 'T'
23                      and column_name = 'Y';
24      end;
```

本地过程 SET_REDO_SET 会查询 V$MYSTAT 和 V$STATNAME，来获取到目前为止会话已生成的当前 redo 量。它将过程中的变量 L_REDO_SIZE 设置为这个值：

```
25      procedure set_redo_size
26      as
27      begin
28              select v$mystat.value
29                      into l_redo_size
30              from v$mystat, v$statname
31              where v$mystat.statistic# = v$statname.statistic#
32                      and v$statname.name = 'redo size';
33      end;
```

接下来是主例程。它收集当前的 redo 大小，运行一个 INSERT/UPDATE/DELETE，然后把该操作生成的 redo 保存到 LOG 表中：

```
34      begin
```

```
35              set_redo_size;
36              insert into t
37                      select object_id, object_name, created
38                      from all_objects
39                      where rownum <= l_cnt;
40              l_cnt := sql%rowcount;
41              commit;
42              report('insert');
43
44              set_redo_size;
45              update t set y=lower(y);
46              l_cnt := sql%rowcount;
47              commit;
48               report('update');
49
50              set_redo_size;
51              delete from t;
52              l_cnt := sql%rowcount;
53              commit;
54              report('delete');
55      end;
56 /
```

一旦有了这个例程，下面将 Y 列的宽度设置为 2,000，然后运行以下脚本来测试 3 种场景：没有触发器、有 BEFORE 触发器，以及有 AFTER 触发器。

```
ops$tkyte@ORA10G> exec do_work('no trigger');

insert redo size = 505960 rows = 200 2,529.8 bytes/row

update redo size = 837744 rows = 200 4,188.7 bytes/row

delete redo size = 474164 rows = 200 2,370.8 bytes/row

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> create or replace trigger before_insert_update_delete

  2 before insert or update or delete on T for each row

  3 begin

  4          null;

  5 end;

  6 /

Trigger created.

ops$tkyte@ORA10G> truncate table t;

Table truncated.



ops$tkyte@ORA10G> exec do_work('before trigger');

insert redo size = 506096 rows = 200 2,530.5 bytes/row

update redo size = 897768 rows = 200 4,488.8 bytes/row

delete redo size = 474216 rows = 200 2,371.1 bytes/row

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> drop trigger before_insert_update_delete;
```

```
Trigger dropped.

ops$tkyte@ORA10G> create or replace trigger after_insert_update_delete
  2 after insert or update or delete on T
  3 for each row
  4 begin
  5          null;
  6 end;
  7 /

Trigger created.



ops$tkyte@ORA10G> truncate table t;

Table truncated.



ops$tkyte@ORA10G> exec do_work( 'after trigger' );

insert redo size = 505972     rows = 200 2,529.9 bytes/row

update redo size = 856636        rows = 200 4,283.2 bytes/row

delete redo size = 474176 rows = 200 2,370.9 bytes/row

PL/SQL procedure successfully completed.
```

前面的输出是在把 Y 大小设置为 2,000 字节时运行脚本所得到的。完成所有运行后，就能查询 LOG 表，并看到以下结果：

```
ops$tkyte@ORA10G> break on op skip 1

ops$tkyte@ORA10G> set numformat 999,999

ops$tkyte@ORA10G> select op, rowsize, no_trig,
                                  before_trig-no_trig, after_trig-no_trig
```

```
  2 from
  3 ( select op, rowsize,
  4          sum(decode( what, 'no trigger', redo_size/rowcnt,0 ) ) no_trig,
  5          sum(decode( what, 'before trigger', redo_size/rowcnt, 0 ) ) before_trig,
  6          sum(decode( what, 'after trigger', redo_size/rowcnt, 0 ) ) after_trig
  7 from log
  8 group by op, rowsize
  9 )
  10 order by op, rowsize
  11 /

OP              ROWSIZE              NO_TRIG          BEFORE_TRIG-NO_TRIG
     AFTER_TRIG-NO_TRIG

---------       ----------           ----------       ----------------------------------
         --------------------------------

delete                30                   291                            0
               0

                     100                   364                           -1
              -0

                     500                   785                           -0
               0

                   1,000                 1,307                           -0
              -0

                   2,000                 2,371                            0
              -0

insert                30                   296                            0
              -0

                     100                   367                            0
```

```
                0

                  500                822                  1
                1

                1,000              1,381                -0
               -0

                2,000              2,530                0
                0

update            30                147                358
              152

                  100                288                363
              157

                  500              1,103                355
              150

                1,000              2,125                342
              137

                2,000              4,188                300
               94

15 rows selected.
```

现在，我想知道日志模式（ARCHIVELOG 和 NOARCHIVELOG 模式）是否会影响这些结果。我发现答案是否定的，这两种模式得到的结果数一样。我很奇怪为什么这个结果与 Expert One-on-One Oracle 第 1 版中的结果有很大的差异。你现在读的这本书或多或少就是以那本书为基础的。出版那本书时，Oracle 的最新版本是 Oracle8i8.1.7。前面所示的 Oracle 10g 结果与 Oracle8i 得到的结果大不相同，但是对于 Oracle9i，这个表中所示的结果则与 Oracle8i 的结果很接近：

我发现，在 Oracle9i Release 2 和 Oracle 10g 这两个版本之间，触发器对事务实际生成的 redo 存在不同的影响。可以很容易地看到这些执行：

- 是否存在触发器对 DELETE 没有影响（DELETE 还是不受触发器的影响）。
- 在 Oracle9i Release 2 及以前版本中，INSERT 会受到触发器的影响。初看上去，你可能会说，Oracle 10g 优化了 INSERT，所以它不会受到影响，但是再看看 Oracle 10g 中无触发器时生成的 redo 总量，你会看到，这与 Oracle9i Release 2 及以前版本中有触发器时生成的 redo 量是一样的。所以，并不是 Oracle 10g 减少了有触发器时 INSERT 生成的 redo 量，而是所生成的 redo 量是常量（有无触发器都会生成同样多的 redo），无触发器时，Oracle 10g 中的 INSERT 比 Oracle9i 中生成的 redo

要多。

- 在 9i 中，UPDATE 会受 BEFORE 触发器的影响，但不受 AFTER 触发器的影响。初看上去，似乎 Oracle 10g 中改成了两个触发器都会影响 UPDATE。但是通过进一步的分析，可以看到，实际上 Oracle 10g 中无触发器是 UPDATE 生成的 redo 有所下降，下降的量正是有触发器时 UPDATE 生成的 redo 量。所用与 9i 和 10g 中 INSERT 的情况恰恰相反，与 9i 相比，没有触发器时 Oracle 10g 中 UPDATE 生成的 redo 量会下降。

表 9-3 对此做了一个总结，这里列出了 Oracle9i 及以前版本与 Oracle 10g 中触发器的 DML 操作生成的 redo 量分别有怎样的影响。

表 9-3　触发器对 redo 生成的影响

DML 操作	AFTER 触发器（10g 以前）	BEFORE 触发器（10g 以前）	AFTER 触发器（10g）	BEFORE 触发器（10g）
DELETE	不影响	不影响	不影响	不影响
INSERT	增加 redo	增加 redo	常量 redo	常量 redo
UPDATE	增加 redo	不影响	增加 redo	增加 redo

测试用例的重要性

更新这本书的第一版时，我切切实实地感受到，这是一个绝好的例子，可以充分说明为什么要用测试用例展示一件事物到底是好是坏。如果我在第一版中只是下了个结论：“触发器会如此这般影响 INSERT、UPDATE 和 DELETE”，而没有提供一种方法来加以度量，另外倘若在这里也没有提供测试用例，那我很有可能还会沿袭同样的结论。在 Oracle9i 和 Oracle 10g 中同样地运行这些测试用例，却会得到不同的结果，所以现在我能很容易地展示出这两个版本间的差异，并且知道 Oracle 数据库确实“发生了变化”。在对第一版更新的过程中，我一次次地发现，如果没有这些测试用例，我可能完全依据过去的经验妄下断言，得出许多错误的结论。

现在你应该知道怎么来估计 redo 量，这是每一个开发人员应该具备的能力。你可以：

- 估计你的“事务”大小（你要修改多少数据）。
- 在要修改的数据量基础上再加 10%～20%的开销，具体增加多大的开销取决于你要修改的行数。修改行越多，增加的开销就越小。
- 对于 UPDATE，要把这个估计值加倍。

在大多数情况下，这将是一个很好的估计。UPDATE 的估计值加倍只是一个猜测，实际上这取决于你修改了多少数据。之所以加倍，是因为在此假设要取一个 X 字节的行，并把它更新（UPDATE）为另一个 X 字节的行。如果你取一个小行（数据量较少的行），要把它更新为一个大行（数据量较多的行），就不用对这个值加倍（这更像是一个 INSERT）。如果取一个大行，而把它更新为一个小行，也不用对这个值加倍（这更像是一个 DELETE）。

加倍只是一种“最坏情况”，因为许多选项和特性会对此产生影响，例如，存在索引或者没有索引（我这里就没有索引）也会影响这个底线。维护索引结构所必须的工作量对不同的 UPDATE 来说是不同的，此外还有一些影响因素。除了前面所述的固定开销外，还必须把触发器的副作用考虑在内。另外要考虑到 Oracle 为你执行的隐式操作（如外键上的 ON DELETE CASCADE 设置）。有了这些考虑，你就能适当地估计 redo 量以便调整事务大小以及实现性能优化。不过，只有通过实际的测试才能得到确定的答案。给定以上脚本，你应该已经知道如何对任何对象和事务自行测量 redo。

9.4.3 我能关掉重做日志生成吗？

这个问题经常被问到。答案很简单：不能。因为重做日志对于数据库至关重要；它不是开销，不是浪费。不论你是否相信，重做日志对你来说确确实实必不可少。这是无法改变的事实，也是数据库采用的工作方式。如果你真的“关闭了 redo”，那么磁盘驱动器的任何暂时失败、掉电或每个软件崩溃都会导致整个数据库不可用，而且不可恢复。不过需要指出，有些情况下执行某些操作时确实可以不生成重做日志。

注意 对于 Oracle9i Release 2，DBA 可能把数据库置于 FORCE LOGGING 模式。在这种情况下，所有操作都会计入日志。查询 SELECT FORCE_LOGGING FROM V$DATABASE 可以查看是否强制为日志模式。这个特性是为了支持 Data Guard, Data Guard 是 Oracle 的一个灾难恢复特性，它依赖于 redo 来维护一个备用数据库（standby database）备份。

1. 在 SQL 中设置 NOLOGGING

有些 SQL 语句和操作支持使用 NOLOGGING 子句。这并不是说：这个对象的所有操作在执行时都不生成重做日志，而是说有些特定操作生成的 redo 会比平常（即不使用 NOLOGGING 子句时）少得多。注意，我只是说“redo”少得多，而不是“完全没有 redo“。所有操作都会生成一些 redo——不论日志模式是什么，所有数据字典操作都会计入日志。只不过使用 NOLOGGING 子句后，生成的 redo 量可能会显著减少。下面是使用 NOLOGGING 子句的一个例子。为此先在采用 ARCHIVELOG 模式运行的一个数据库中运行以下命令：

```
ops$tkyte@ORA10G> select log_mode from v$database;



LOG_MODE

------------

ARCHIVELOG



ops$tkyte@ORA10G> @mystat "redo size"

ops$tkyte@ORA10G> set echo off
```

```
NAME                           VALUE
---------                      ----------
redo size                      5846068


ops$tkyte@ORA10G> create table t
  2 as
  3 select * from all_objects;
Table created.


ops$tkyte@ORA10G> @mystat2
ops$tkyte@ORA10G> set echo off
NAME                           V                DIFF
-------------              -------------              ----------------
redo size          11454472         5,608,404
```

这个 CREATE TABLE 生成了大约 5.5MB 的 redo 信息。接下来删除这个表，再重建，不过这一次采用 NOLOGGING 模式：

```
ops$tkyte@ORA10G> drop table t;
Table dropped.


ops$tkyte@ORA10G> @mystat "redo size"
ops$tkyte@ORA10G> set echo off


NAME                           VALUE
```

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

redo size                     11459508



ops$tkyte@ORA10G> create table t

  2 NOLOGGING

  3 as

  4 select * from all_objects;

Table created.



ops$tkyte@ORA10G> @mystat2

ops$tkyte@ORA10G> set echo off



NAME                          V                 DIFF

----------                    ----------        ----------------

redo size                     11540676          81,168
```

这一次，只生成了 80KB 的 redo 信息。

可以看到，差距很悬殊：原来有 5.5MB 的 redo，现在只有 80KB。5.5MB 是实际的表数据本身；现在它直接写至磁盘，对此没有生成重做日志。

如果对一个 NOARCHIVELOG 模式的数据库运行这个测试，就看不到什么差别。在 NOARCHIVELOG 模式的数据库中，除了数据字典的修改外，CREATE TABLE 不会记录日志。如果你想在 NOARCHIVELOG 模式的数据库上看到差别，可以把对表 T 的 DROP TABLE 和 CREATE TABLE 换成 DROP INDEX 和 CREATE INDEX。默认情况下，不论数据库以何种模式运行，这些操作都会生成日志。从这个例子可以得出一个很有意义的提示：要按生产环境中所采用的模式来测试你的系统，因为不同的模式可能导致不同的行为。你的生产系统可能采用 AUCHIVELOG 模式运行；倘若你执行的大量操作在 ARCHIVELOG 模式下会生成 redo，而在 NOARCHIVELOG 模式下不会生成 redo，你肯定想在测试时就发现这一点，而不要等到系统交付给用户时才暴露出来！

这么说，好像所有工作都应该尽可能采用 NOLOGGING 模式，是这样吗？实际上，答案很干脆：恰恰相反。必须非常谨慎地使用这种模式，而且要与负责备份和恢复的人沟通之后才能使用。下面假设你创建了一个非日志模式的表，并作为应用的一部分（例如，升级脚

本中使用了 CREATE TABLE AS SELECT NOLOGGING）。用户白天修改了这个表。那天晚上，表所在的磁盘出了故障。“没关系“，DBA 说”数据库在用 ARCHIVELOG 模式运行，我们可以执行介质恢复“。不过问题是，现在无法从归档重做日志恢复最初创建的表，因为根本没有生成日志。这个表将无法恢复。由此可以看出使用 NOLOGGIG 操作最重要的一点是：必须与 DBA 和整个系统协调。如果你使用了 NOLOGGING 操作，而其他人不知道这一点，你可能就会拖 DBA 的后退，使得出现介质失败后 DBA 无法全面地恢复数据库。必须谨慎而且小心地使用这些 NOLOGGING 操作。

关于 NOLOGGING 操作，需要注意以下几点：

- 事实上，还是会生成一定数量的 redo。这些 redo 的作用是保护数据字典。这是不可避免的。与以前（不使用 NOLOGGING）相比，尽管生成的 redo 量要少多了，但是确实会有一些 redo。

- NOLOGGING 不能避免所有后续操作生成 redo。在前面的例子中，我创建的并非不生成日志的表。只是创建表（CREATE TABLE）这一个操作没有生成日志。所有后续的“正常“操作（如 INSERT、UPDATE 和 DELETE）还是会生成日志。其他特殊的操作（如使用 SQL*Loader 的直接路径加载，或使用 INSERT /*+ APPEND */语法的直接路径插入）不生成日志（除非你 ALTER 这个表，再次启用完全的日志模式）。不过，一般来说，应用对这个表执行的操作都会生成日志。

- 在一个 ARCHIVELOG 模式的数据库上执行 NOLOGGING 操作后，必须尽快为受影响的数据文件建立一个新的基准备份，从而避免由于介质失败而丢失对这些对象的后续修改。实际上，我们并不会丢失后来做出的修改，因为这些修改确实在重做日志中；我们真正丢失的只是要应用这些修改的数据（即最初的数据）。

2. 在索引上设置 NOLOGGING

使用 NOLOGGING 选项有两种方法。你已经看到了前一种，也就是把 NOLOGGING 关键字潜在 SQL 命令中。另一种方法是在段（索引或表）上设置 NOLOGGING 属性，从而隐式地采用 NOLOGGING 模式来执行操作。例如，可以把一个索引或表修改为默认采用 NOLOGGING 模式。这说明，以后重建这个索引不会生成日志（其他索引和表本身可能还会生成 redo，但是这个索引不会）：

```
ops$tkyte@ORA10G> create index t_idx on t(object_name);

Index created.



ops$tkyte@ORA10G> @mystat "redo size"

ops$tkyte@ORA10G> set echo off



NAME                           VALUE
```

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

redo size                      13567908
```

```
ops$tkyte@ORA10G> alter index t_idx rebuild;

Index altered.

ops$tkyte@ORA10G> @mystat2

ops$tkyte@ORA10G> set echo off

NAME                           V                DIFF

-------------              --------------              ----------------

redo size          15603436          2,035,528
```

这个索引采用 LOGGING 模式（默认），重建这个索引会生成 2MB 的重做日志。不过，可以如下修改这个索引：

```
ops$tkyte@ORA10G> alter index t_idx nologging;

Index altered.



ops$tkyte@ORA10G> @mystat "redo size"

ops$tkyte@ORA10G> set echo off



NAME                           VALUE

----------                     ----------

redo size                      15605792



ops$tkyte@ORA10G> alter index t_idx rebuild;

Index altered.
```

```
ops$tkyte@ORA10G> @mystat2

ops$tkyte@ORA10G> set echo off


NAME                           V                DIFF

-------------          -------------          ----------------

redo size          15668084          62,292
```

现在它只生成 61KB 的 redo。但是，现在这个索引没有得到保护（unprotected），如果它所在的数据文件失败而必须从一个备份恢复，我们就会丢失这个索引数据。了解这一点很重要。现在索引是不可恢复的，所以需要做一个备份。或者，DBA 也可以干脆创建索引，因为完全可以从表数据直接创建索引。

3. NOLOGGING 小结

可以采用 NOLOGGING 模式执行以下操作：

- 索引的创建和 ALTER（重建）。
- 表的批量 INSERT（通过/*+APPEND */提示使用“直接路径插入“。或采用 SQL*Loader 直接路径加载）。表数据不生成 redo，但是所有索引修改会生成 redo，但是所有索引修改会生成 redo（尽管表不生成日志，但这个表上的索引却会生成 redo！）。
- LOB 操作（对大对象的更新不必生成日志）。
- 通过 CREATE TABLE AS SELECT 创建表。
- 各种 ALTER TABLE 操作，如 MOVE 和 SPLIT。

在一个 ARCHIVELOG 模式的数据库上，如果 NOLOGGING 使用得当，可以加快许多操作的速度，因为它能显著减少生成的重做日志量。假设你有一个表，需要从一个表空间移到另一个表空间。可以适当地调度这个操作，让它在备份之后紧接着发生，这样就能把表 ALTER 为 NOLOGGING 模式，移到表，创建索引（也不生成日志），然后再把表 ALTER 回 LOGGING 模式。现在，原先需要 X 小时才能完成的操作可能只需要 X/2 小时（运行是会不会真的减少 50%的时间，这一点我不敢打保票！）。要想适当地使用这个特性，需要 DBA 的参与，或者必须与负责数据库备份和恢复（或任何备用数据库）的人沟通。如果这个人不知道使用了这个特性，一旦出现介质失败，就可能丢失数据，或者备用数据库的完整性可能遭到破坏。对此一定要三思。

9.4.4 为什么不能分配一个新日志？

老是有人问我这个问题。这样做会得到一条警告消息（可以在服务器上的 alert.log 中看到 ：

```
Thread 1 cannot allocate new log, sequence 1466

Checkpoint not complete

Current log# 3 seq# 1465 mem# 0: /home/ora10g/oradata/ora10g/redo03.log
```

警告消息中也可能指出 Archival required 而不是 Checkpoint not complete，但是效果几乎都一样。DBA 必须当心这种情况。如果数据库试图重用一个在线重做日志文件，但是发现做不到，就会把这样一条消息写到服务器上的 alert.log 中。如果 DBWR 还没有完成重做日志所保护数据的检查点（checkpointing）,或者 ARCH 还没有把重做日志文件复制到归档目标，就会发生这种情况。对最终用户来说，这个时间点上数据库实际上停止了。它会原地不动。DBWR 或 ARCH 将得到最大的优先级以将 redo 块刷新输出的磁盘。完成了检查点或归档之后，一切又回归正常。数据库之所以暂停用户的活动，这是因为此时已经没地方记录用户所做的修改了。Oracle 试图重用一个在线重做日志文件，但是由于归档进程尚未完成这个文件的复制（Archival required），所以 Oracle 必须等待（相应地，最终用户也必须等待），直到能安全地重用这个重做日志文件为止。

如果你看到会话因为一个“日志文件切换”、“日志缓冲区空间”或“日志文件切换检查点或归档未完成”等待了很长时间，就很可能遇到了这个问题。如果日志文件大小不合适，或者 DBWR 和 ARCH 太慢（需要由 DBA 或系统管理员调优），在漫长的数据库修改期间，你就会注意到这个问题。我经常看到未定制的“起始”数据库就存在这个问题。“起始”数据库一般会把重做日志的大小定得太小，不适用较大的工作量（包括数据字典本身的起始数据库构建）。一旦启动数据库的加载，你会注意到，前 1,000 行进行得很快，然后就会呈喷射状进行：1,000 进行得很快，然后暂停，接下来又进行得很快，然后又暂停，如此等等。这些就是很明确的提示，说明你遭遇了这个问题。

要解决这个问题，有几种做法：

- 让 DBWR 更快一些。让你的 DBA 对 DBWR 调优，为此可以启用 ASYNC I/O、使用 DBWR I/O 从属进程，或者使用多个 DBWR 进程。看看系统产生的 I/O，查看是否有一个磁盘（或一组磁盘）“太热”，相应地需要将数据散布开。这个建议对 ARCH 也适用。这种做法的好处是，你不用付出什么代价就能有所收获，性能会提高，而且不必修改任何逻辑/结构/代码。这种方法确实没有缺点。

- 增加更多重做日志文件。在某些情况下，这会延迟 Checkpoint not complete 的出现，而且过一段时间后，可以把 Checkpoint not complete 延迟得足够长，使得这个错误可能根本不会出现（因为你给 DBWR 留出了足够的活动空间来建立检查点）。这个方法也同样适用于 Archival required 消息。这种方法的好处是可以消除系统中的“暂停”。其缺点是会消耗更多的磁盘空间，但是在此利远远大于弊。

- 重新创建更大的日志文件。这会扩大填写在线重做日志与重用这个在线重做日志文件之间的时间间隔。如果重做日志文件的使用呈“喷射状”，这种方法同样适用于 Archival required 消息。倘若一段时间内会大量生成日志（如每晚加载、批处理等），其后一段数据却相当平静，如果有更大的在线重做日志，就能让 ARCH 在平静的期间有足够的时间“赶上来”。这种方法的优缺点与前面增加更多文件的方法是一样的。另外，它可能会延迟检查点的发生，由于（至少）每个日志切换都会发生检查点，而现在日志切换间隔会更大。

- 让检查点发生得更频繁、更连续。可以使用一个更小的块缓冲区缓存（不太好），或者使用诸如 FAST_START_MTTR_TARGET、LOG_CHECKPOINT_INTERVAL 和 LOG_CHECKPOINT_TIMEOUT 之类的参数设置。这会强制 DBWR 更 频繁地刷新输出脏块。这种方法的好处是，失败恢复的时间会减少。在线重做日志中应用的工作肯定更少。其缺点是，如果经常修改块，可能会更频繁地写至磁盘。 缓冲区缓存本该更有效的，但由于频繁地写磁盘，会导致缓冲区缓存不能充分发挥作用，这可能会影响下一节将讨论的块清除机制。

究竟选择哪一种方法，这取决于你的实际环境。应该在数据库级确定它，要把整个实例都考虑在内。

9.4.5 块清除

在这一节中，我们将讨论块清除（block cleanout），即生成所修改数据库块上与“锁定”有关的信息。这个概念很重要，必须充分理解，在下一节讨论讨厌的 ORA-01555:snapshot too old 错误时会用到这个概念。

在第 6 章中，我们曾经讨论过数据锁以及如何管理它们。我介绍了数据锁实际上是数据的属性，存储在块首部。这就带来一个副作用，下一次访问这个块时，可能必须“清理”这个块，换句话说，要将这些事务信息删除。这个动作会生成 redo，并导致变脏（原本并不脏，因为数据本身没有修改），这说明一个简单的 SELECT 有可能生成 redo，而且可能导致完成下一个检查点时将大量的块写至磁盘。不过，在大多数正常的情况下，这是不会发生的。如果系统中主要是小型或中型事务（OLTP），或者数据仓库会执行直接路径加载或使用 DBMS_STATS 在加载操作后分析表，你会发现块通常已经得到“清理”。如果还记得前面“COMMIT 做什么？”一节中介绍的内容，应该知道，COMMIT 时处理的步骤之一是：如果块还在 SGA 中，就要再次访问这些块，如果可以访问（没有别人在修改这些块），则对这些块完成清理。这个 活动称为提交清除（commit cleanout），即清除已修改块上事务信息。最理想的是，COMMIT 可以完成块清除，这样后面的 SELECT（读）就不必再清理了。只有块的 UPDATE 才会真正清除残余的事务信息，由于 UPDATE 已经在生成 redo，所用注意不到这个清除工作。

可以强制清除不发生来观察它的副作用，并了解提交清除是怎么工作的。在与我们的事务相关的提交列表中，Oracle 会记录已修改的块列表。这些列表都有 20 个块，Oracle 会根据需要分配多个这样的列表，直至达到某个临界点。如果我们修改的块加起来超过了块缓冲区缓存大小的 10%，Oracle 会停止为我们分配新的列表。例如，如果缓冲区缓存设置为可以缓存 3,000 个块，Oracle 会为我们维护最多 300 个块（3,000 的 10%）。COMMIT 时，Oracle 会处理这些包含 20 个块指针的列表，如果块仍可用，它会执行一个很快的清理。所以，只要我们修改的块数没有超过缓存中总块数的 10%，而且块仍在缓存中并且是可用的，Oracle 就会在 COMMIT 时清理这些块。否则，它只会将其忽略（也就是说不清理）。

有了上面的理解，可以人为地建立一些条件来查看这种块清除是怎么工作的。我把 DB_CACHE_SIZE 设置为一个很低的值 4MB，这足以放下 512 个 8KB 的块（我的块大小是 8KB）。然后创建一个表，其中每行刚好能在一个块中放下（我不会在每块里放两行）。接下来在这个表中填入了 500 行，并 COMMIT。我要测量到此为止生成的 redo 量。然后运行一个 SELECT，它会访问每个块，最后测量这个 SELECT 生成的 redo 量。

让许多人奇怪的是，SELECT 居然会生成 redo。不仅如此，它还会把这些修改块“弄

脏”，导致 DBWR 再次将块写入磁盘。这是因为块清除的缘故。接下来，我会再一次运行 SELECT，可以看到这回没有生成 redo。这在意料之中，因为此时块都已经“干净”了。

```
ops$tkyte@ORA10G> create table t
  2 ( x char(2000),
  3 y char(2000),
  4 z char(2000)
  5 )
  6 /
Table created.
ops$tkyte@ORA10G> set autotrace traceonly statistics
ops$tkyte@ORA10G> insert into t
  2 select 'x', 'y', 'z'
  3 from all_objects
  4 where rownum <= 500;
500 rows created.
Statistics
----------------------------------------------------------
...
          3297580  redo  size
...
          500          rows processed
ops$tkyte@ORA10G> commit;
Commit complete.
```

以上就是我的表，每个块中一行（我的数据库中块大小为 8KB）。现在测量读数据是生成的 redo 量：

```
ops$tkyte@ORA10G> select *

2 from t;

500 rows selected.

Statistics

----------------------------------------------------------

...

          36484          redo    size

...

          500      rows processed
```

可见，这个 SELECT 在处理期间生成了大约 35KB 的 redo。这表示对 T 进行全表扫描时修改了 35KB 的块首部。DBWR 会在将来某个时间把这些已修改的块写回到磁盘上。现在，如果再次运行这个查询：

```
ops$tkyte@ORA10G> select *

2 from t;

500 rows selected.

Statistics

----------------------------------------------------------

...

          0          redo    size

...

          500      rows    processed

ops$tkyte@ORA10G> set autotrace off
```

可以看到，这一次没有生成 redo，块都是干净的。

如果把缓冲区缓存设置为能保存至少 5,000 个块，再次运行前面的例子。你会发现，无论哪一个 SELECT，生成的 redo 都很少甚至没有——我们不必在其中任何一个 SELECT 语句期间清理脏块。这是因为，我们修改的 500 个块完全可以在缓冲区缓存的 10%中放下，而且我们是独家用户。别人不会动数据，不会有人导致我们的数据刷新输出到磁盘，也没有

人在访问这些块。在实际系统中，有些情况下，至少某些块不会进行清除，这是正常的。

如果执行一个大的 INSERT（如上所述）、UPDATE 或 DELETE，这种块清除行为的影响最大，它会影响数据库中的许多块（缓存中 10%以上的块都会完成块清除）。你会注意到，在此之后，第一个“接触”块的查询会生成少量的 redo，并把块弄脏，如果 DBWR 已经将块刷新输出或者实例已经关闭，可能就会因为这个查询而导致重写这些块，并完全清理缓冲区缓存。对此你基本上做不了什么。这是正常的，也在意料之中。如果 Oracle 不对块完成这种延迟清除，那么 COMMIT 的处理就会与事务本身一样长。COMMIT 必须重新访问每一个块，可能还要从磁盘将块再次读入（它们可能已经刷新输出）。

如果你不知道块清除，不明白块清除如果工作，在你看来中可能就是一种好像毫无来由的神秘事务。例如，假设你更新（UPDATE）了大量数据，然后 COMMIT。现在对这些数据运行一个查询来验证结果。看上去查询生成了大量写 I/O 和 redo。倘若你不知道存在块清除，这似乎是不可能的；对我来说，第一次看到这种情况时就是这样认为的，实在是不可思议。然后你请别人一起来观察这个行为，但这是不可再生的，因为在第二次查询时块又是“干净的”了。这样一来，你就会把它当成是数据库的奥秘之一。

在一个 OLTP 系统中，可能从来不会看到这种情况发生，因为 OLTP 系统的特点是事务都很短小，只会影响为数不多的一些块。根据设计，所有或者大多数事务都短而精。只是修改几个块，而且这些块都会得到清理。在一个数据仓库中，如果加载之后要对数据执行大量 UPDATE，就要把块清除作为设计中要考虑的一个因素。有些操作会在“干净”的块上创建数据。例如，CREATE TABLE AS SELECT、直接路径加载的数据以及直接路径插入的数据都会创建“干净”的块。UPDATE、正常的 INSERT 或 DELETE 创建的块则可能需要在第一次读时完成块清除。如果你有如下的处理，就会受到块清除的影响：

- ☐ 将大量新数据批量加载到数据仓库中；
- ☐ 在刚刚加载的所有数据上运行 UPDATE（产生需要清理的块）；
- ☐ 让人们查询这些数据。

必须知道，如果块需要清理，第一接触这个数据的查询将带来一些额外的处理。如果认识到这一点，你就应该在 UPDATE 之 后自己主动地“接触”数据。你刚刚加载或修改了大量的数据；现在至少需要分析这些数据。可能要自行运行一些报告来验证数据已经加载。这些报告会完成块清 除，这样下一个查询就不必再做这个工作了。更好的做法是：由于你刚刚批量加载了数据，现在需要以某种方式刷新统计。通过运行 DBMS_STATS 实用程序来收集统计，就能很好地清理所有块，这是因为它只是使用 SQL 来查询信息，会在查询当中很自然地完成块清除。

9.4.6 日志竞争

与 cannot allocate new log 信息一样，日志竞争（log contention）也是 DBA 必须修改的问题，一般要与系统管理员联手。不过，如果 DBA 检查得不够仔细，开发人员也可以检测到这个问题。

如果你遭遇到日志竞争，可能会看到对“日志文件同步”事件的等待时间相当长，另外 Statspack 报告的“日志文件并行写”事件中写次数（写 I/O 数）可能很大。如果观察到这种情况，就说明你遇到了重做日志的竞争；重做日志写得不够快。发生这种情况可能有许多原因。其中一个应用原因（所谓应用原因是指 DBA 无法修正这个问题，而必须由开发人

员解决）是：提交得太过频繁，例如在重复执行 INSERT 的循环中反复提交。在“COMMIT 做什么？”一节中我们讲过，如果提交得太频繁，这不仅是不好的编程实践，肯定还会引入大量日志文件同步等待。假设你的所有事务都有适当的大小（完全遵从业务规则的要求，而没有过于频繁地提交），但还是看到了这种日志文件等待，这就有其他原因了。其中最常见的原因如下：

- ☐ redo 放在一个慢速设备上：磁盘表现不佳。该购买速度更快的磁盘了。
- ☐ redo 与其他频繁访问的文件放在同一个设备上。redo 设计为要采用顺序写，而且要放在专用的设备上。如果系统的其他组件（甚至其他 Oracle 组件）试图与 LGWR 同时读写这个设备，你就会遭遇某种程度的竞争。在此，只要有可能，你就会希望确保 LGWR 拥有这些设备的独占访问权限。
- ☐ 已缓冲方式装载日志设备。你在使用一个“cooked”文件系统（而不是 RAW 磁盘）。操作系统在缓冲数据，而数据库也在缓冲数据（重做日志缓冲区）。这种双缓冲会让速度慢下来。如果可能，应该以一种“直接”方式了装载设备。具体操作依据操作系统和设备的不同而有所变化，但一般都可以直接装载。
- ☐ redo 采用了一种慢速技术，如 RAID-5。RAID-5 很合适读，但是用于写时表现则很差。前面已经了解了 COMMIT 期间会发生什么，我们必须等待 LGWR 以确保数据写到磁盘上。倘若使用的技术会导致这个工作变慢，这就不是一个好主意。

只有有可能，实际上你会希望至少有 5 个专用设备来记录日志，最好还有第 6 个设备来镜像归档日志。由于当前往往使用 9GB、20GB、36GB、200GB、300GB 和更大的磁盘，要想拥有这么多专用设备变得更加困难。但是如果能留出 4 块你能找到的最小、最快的磁盘，再有一个或两个大磁盘，就可以很好地促进 LGWR 和 ARCH 的工作。安排这些磁盘时，可以把它们分为 3 组（见图 9-5）：

- ☐ 重做日志组 1：磁盘 1 和磁盘 3
- ☐ 重做日志组 2：磁盘 2 和磁盘 4
- ☐ 归档：磁盘 5，可能还有磁盘 6（大磁盘）

图 9-5 最优的重做日志配置

将重做日志组 1（包括成员 A 和 B）放在磁盘 1 和磁盘 3 上。把重做日志组 2（包括成员 C 和 D）放在磁盘 2 和磁盘 4 上。如果还有组 3、4 等，将分别放在相应的奇数和偶数磁盘组上。这样做的作用是，数据库当前使用组 1 时，LGWR 会同时写至磁盘 1 和 3.这一组填满时，LGWR 会转向磁盘 2 和 4.等这一组再填满时，LGWR 会回到磁盘 1 和 3.与此同时，ARCH 会处理完整的在线重做日志，并讲其写至磁盘 5 和 6（即大磁盘）。最终的效果是，不论是 ARCH 还是 LGWR 都不会读正在有别人写的磁盘，也不会写正在由别人读的磁盘，所以在此没有竞争（见图 9-6）。

图 9-6　重做日志流

因此，当 LGWR 写组 1 时，ARCH 在读组 2，并写至归档磁盘。当 LGWR 写组 2 时，ARCH 在读组 1，并写至归档磁盘。采用这种方式，LGWR 和 ARCH 都有各自的专用设备，不会与别人竞争，甚至不会相互竞争。

在线重做日志文件是一组 Oracle 文件，最适合使用 RAW 磁盘（原始磁盘）。如果说哪种类型的文件可以考虑使用原始分区（RAW），首先其冲地便是日志文件。关于使用原始分区和 cooked 文件系统的优缺点，这方面的讨论很复杂。由于这不是一本有关 DBA/SA 认为的书，所以我不打算过分深入。但是要指出，如果你要使用 RAW 设备，在线重做日志文件就是最佳候选。在线重做日志文件不用备份，所以将在线重做日志文件放在 RAW 分区上而不是 cooked 文件系统上，这不会影响你的任何备份脚本。ARCH 总能把 RAW 日志转变为 cooked 文件系统文件（不能使用一个 RAW 设备来建立归档），在这种情况下，就大大减少了 RAW 设备的“神秘感”。

9.4.7 临时表和 redo/undo

一般认为临时表（temporary table）还是 Oracle 中一个相当新的特性，只是在 Oracle8i 8.1.5 版本中才引入。因此，有关临时表还存在一些困惑，特别是在日志方面。我们将在第 10 章介绍如何以及为什么使用临时表。这一节只是要回答这样一个问题：“关于生成修改日志，临时表是怎样做的？”

临时表不会为它们的块生成 redo。因此，对临时表的操作不是“可恢复的”。修改临时表中的一个块时，不会将这个修改记录到重做日志文件中。不过，临时表确实会生成 undo，

而且这个 undo 会计入日志。因此，临时表也会生成一些 redo。初看上去好像没有道理：为什么需要生成 undo？这是因为你能回滚到事务中的一个 SAVEPOINT。可以擦除对临时表的后 50 个 INSERT，而只留下前 50 个。临时表可以有约束，正常表有的一切临时表都可以有。可能有一条 INSERT 语句要向临时表中插入 500 行，但插入到第 500 行时失败了，这就要求回滚这条语句。由于临时表一般表现得就像“正常”表一样，所以临时表必须生成 undo。由于 undo 数据必须建立日志，因此临时表会为所生成的 undo 生成一些重做日志。

这样似乎很不好，不过没有你想像中那么糟糕。在临时表上运行的 SQL 语句主要是 INSERT 和 SELECT。幸运的是，INSERT 只生成极少的 undo（需要把块恢复为插入前的“没有”状态，而存储“没有”不需要多少空间），另外 SELECT 根本不生成 undo。因此，如果只使用临时表执行 INSERT 和 SELECT，这一节对你来说意义不大。仅当要对临时表执行 UPDATE 或 DELETE 时，才需要关心这一节的内容。

我建立了一个小测试来演示使用临时表时生成的 redo 量，同时这也暗示了临时表生成的 undo 量，因为对于临时表，只会为 undo 生成日志。为了说明这一点，我采用了配置相同的“永久”表和“临时”表，然后对各个表执行相同的操作，测量每次生成的 redo 量。这里使用的表如下：

```
ops$tkyte@ORA10G> create table perm
  2 ( x char(2000) ,
  3 y char(2000) ,
  4 z char(2000) )
  5 /
Table created.



ops$tkyte@ORA10G> create global temporary table temp
  2 ( x char(2000) ,
  3 y char(2000) ,
  4 z char(2000) )
  5 on commit preserve rows
  6 /
Table created.
```

我建立了一个小的存储过程，它能执行任意的 SQL，并报告 SQL 生成的 redo 量。我会使用这个例程分别在临时表和永久表上执行 INSERT、UPDATE 和 DELETE：

```
ops$tkyte@ORA10G> create or replace procedure do_sql( p_sql in varchar2 )
2       as
3               l_start_redo number;
4               l_redo number;
5       begin
6               select v$mystat.value
7                       into l_start_redo
8               from v$mystat, v$statname
9               where v$mystat.statistic# = v$statname.statistic#
10                      and v$statname.name = 'redo size';
11
12              execute immediate p_sql;
13              commit;
14
15              select v$mystat.value-l_start_redo
16                      into l_redo
17              from v$mystat, v$statname
18              where v$mystat.statistic# = v$statname.statistic#
19                      and v$statname.name = 'redo size';
20
21              dbms_output.put_line
22                      ( to_char(l_redo,'9,999,999') ||' bytes of redo generated for "' ||
23                      substr( replace( p_sql, chr(10), ' '), 1, 25 ) || '"...' );
```

```
24      end;

25 /

Procedure created.
```

接下来，对 PERM 表和 TEMP 表运行同样的 INSERT、UPDATE 和 DELETE：

```
ops$tkyte@ORA10G> set serveroutput on format wrapped

ops$tkyte@ORA10G> begin

  2          do_sql( 'insert into perm

  3                  select 1,1,1

  4                  from all_objects

  5                  where rownum <= 500' );

  6

  7          do_sql( 'insert into temp

  8                  select 1,1,1

  9                  from all_objects

 10                   where rownum <= 500' );

 11          dbms_output.new_line;

 12

 13           do_sql( 'update perm set x = 2' );

 14          do_sql( 'update temp set x = 2' );

 15          dbms_output.new_line;

 16

 17          do_sql( 'delete from perm' );

 18          do_sql( 'delete from temp' );
```

```
 19 end;

 20 /

3,297,752 bytes of redo generated for "insert into perm "...

66,488 bytes of redo generated for "insert into temp "...

2,182,200 bytes of redo generated for "update perm set x = 2"...

1,100,252 bytes of redo generated for "update temp set x = 2"...



3,218,804 bytes of redo generated for "delete from perm"...

3,212,084 bytes of redo generated for "delete from temp"...



PL/SQL procedure successfully completed.
```

可以看到：

- 对“实际”表（永久表）的 INSERT 生成了大量 redo。而对临时表几乎没有生成任何 redo。这是有道理的，对临时表的 INSERT 只会生成很少的 undo 数据，而且对于临时表只会为 undo 数据建立日志。
- 实际表的 UPDATE 生成的 redo 大约是临时表更新所生成 redo 的两倍。同样，这也是合理的。必须保存 UPDATE 的大约一半（即“前映像”）。对于临时表来说，不必保存“后映像”（redo）。
- DELETE 需要几乎相同的 redo 空间。这是有道理的，因为对 DELETE 的 undo 很大，而对已修改块的 redo 很小。因此，对临时表的 DELETE 与对永久表的 DELETE 几乎相同。

注意 你看到 INSERT 语句在临时表上生成的 redo 比在永久表上生成的 redo 还多，这实际上是数据库产品本身的问题，这个问题至少在 Oracle9.2.0.6 和 10.1.0.4 补丁版中（编写这本书时发布的当前版本）中得到了修正。

因此，关于临时表上的 DML 活动，可以得出以下一般结论：

- INSERT 会生成很少甚至不生成 undo/redo 活动。
- DELETE 在临时表上生成的 redo 与正常表上生成的 redo 同样多。
- 临时表的 UPDATE 会生成正常表 UPDATE 一半的 redo。

对于最后一个结论，需要指出有一些例外情况。例如，如果我用 2,000 字节的数据 UPDATE（更新）完全为 NULL 的一列，生成的 undo 数据就非常少。这个 UPDATE 表现得

就像是 INSERT。另一方面，如果我把有 2,000 字节数据的一列 UPDATE 为全 NULL，对 redo 生成来说，这就表现得像是 DELETE。平均来讲，可以这样认为：临时表 UPDATE 与实际表 UPDATE 生成的 undo/redo 相比，前者是后者的 50%。

一般来讲，关于创建的 redo 量有一个常识。如果你完成的操作导致创建 undo 数据，则可以确定逆向完成这个操作（撤销操作）的难易程度。如果 INSERT2,000 字节，逆向操作就很容易，只需回退到无字节即可。如果删除了（DELETE）2,000 字节，逆向操作就是要插入 2,000 字节。在这种情况下，redo 量就很大。

有了以上了解，你可能会避免删除临时表。可以使用 TRUNCATE（当然要记住，TRUNCATE 是 DDL，而 DDL 会提交事务，而且在 Oracle9i 及以前版本中，TRUNCATE 还会使你的游标失效），或者只是让临时表在 COMMIT 之后或会话终止时自动置空。执行方法不会生成 undo，相应地也不会生成 redo。你可能会尽量避免更新临时表，除非由于某种原因必须这样做。你会把临时表主要用于插入（INSERT）和选择（SELECT）。采用这种方式，就能更优地使用临时表不生成 redo 的特有能力。

9.5 分析 undo

我们已经讨论了许多有关 undo 段的主题，介绍了恢复时如何使用 undo 段，undo 段与重做日志如何交互，以及 undo 段如何用于数据的一致性、非阻塞读等。在这一节中，我们将分析有关 undo 段的一些常被问到的问题。

我们主要讨论讨厌的 ORA-01555:anapshot too old 错误，因为这个问题所引发的困惑比其他任何数据库主题带来的困惑都多。不过，在此之前，下一节先分析另一个与 undo 相关的问题：哪些类型的 DML 操作会生成最多和最少的 undo（根据前面临时表的有关例子，可能你自己已经能回答这个问题了）。

9.5.1 什么操作会生成最多和最少的 undo？

这是一个常常问到的问题，不过很容易回答。如果存在索引（或者实际上表就是索引组织表），这将显著地影响生成的 undo 量，因为索引是一种复杂的数据结构，可能会生成相当多的 undo 信息。

也就是说，一般来讲，INSERT 生成的 undo 最少，因为 Oracle 为此需记录的只是要“删除”的一个 rowid（行 ID）。UPDATE 一般排名第二（在大多数情况下）。对于 UPDATE，只需记录修改的字节。你可能只更新（UPDATE）了整个数据行中很少的一部分，这种情况最常见。因此，必须在 undo 中记录行的一小部分。前面的许多例子都与这条经验相左，不过这是因为那些列更新的行很大（有固定大小），而且它们更新了整个行。更常见的是 UPDATE 一行，并修改整行中的一小部分。一般来讲，DELETE 生成的 undo 最多。对于 DELETE，Oracle 必须把整行的前映像记录到 undo 段中。在 redo 生成方面，前面的临时表例子展示了这样一个事实：DELETE 生成的 redo 最多，而且由于临时表的 DML 操作只会把 undo 记入日志，这实际上也表明 DELETE 会生成最多的 undo。INSERT 只生成需要建立日志的很少的 undo。UPDATE 生成的 undo 量等于所修改数据的前映像大小，DELETE 会生成整个数据集写至 undo 段。

前面已经提到，必须把索引执行的工作也考虑在内。你会发现，与加索引列的更新相比，对一个未加索引的列进行更新不仅执行得更快，生成的 undo 也会好得多。例如，下面创建一个有两列的表，这两列包含相同的数据，但是其中一列加了索引：

```
ops$tkyte@ORA10G> create table t
  2 as
  3 select object_name unindexed,
  4 object_name indexed
  5 from all_objects
  6 /
Table created.

ops$tkyte@ORA10G> create index t_idx on t(indexed);
Index created.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats(user,'T');
PL/SQL procedure successfully completed.
```

下面更新这个表，首先，更新未加索引的列，然后更新加索引的列。我们需要一个新的 V$查询来测量各种情况下生成的 undo 量。以下查询可以完成这个工作。它先从 V$MYSTAT 得到我们的会话 ID（SID），在使用这个会话 ID 在 V$SESSION 视图中找到相应的会话记录，并获取事务地址（TADDR）。然后使用 TADDR 拉出（查出）我们的 V$TRANSACTION 记录（如果有），选择 USED_UBLK 列，即已用 undo 块的个数。由于我们目前不在一个事务中，这个查询现在应该返回 0 行：

```
ops$tkyte@ORA10G> select used_ublk
  2       from v$transaction
  3       where addr = (select taddr
  4                     from v$session
  5                     where sid = (select sid
  6                                  from v$mystat
  7                                  where rownum = 1
```

```
  8                                          )

  9                                  )

  10 /

no rows selected
```

然后在每个 UPDATE 之后再使用这个查询，不过在正文中不再重复这个查询，下面只会显示查询的结果。

现在我们准备好执行更新，并测试各个更新使用的 undo 块数：

```
ops$tkyte@ORA10G> update t set unindexed = lower(unindexed);

48771 rows updated.



ops$tkyte@ORA10G> select used_ublk

...

10 /

USED_UBLK

----------

401



ops$tkyte@ORA10G> commit;

Commit complete.
```

这个 UPDATE 使用了 401 个块存储其 undo。提交会“解放”这些块，或者将其释放，所以如果再次对 V$TRANSACTION 运行这个查询，它还会显示 no rows selected。更新同样的数据时，不过这一次是加索引的列，会观察到下面的结果：

```
ops$tkyte@ORA10G> update t set indexed = lower(indexed);

48771 rows updated.



ops$tkyte@ORA10G> select used_ublk
```

```
...

10 /

USED_UBLK

----------

1938
```

可以看到，在这个例子中，更新加索引的列会生成几乎 5 倍的 undo。这是因为索引结构本身所固有的复杂性，而且我们更新了这个表中的每一行，移动了这个结构中的每一个索引键值。

9.5.2 ORA-01555:snapshot too old 错误

在上一章中，我们简要分析了 ORA-01555 错误，并了解了导致这个错的一个原因：提交得太过频繁。这一节我们将更详细地分析 ORA-01555 错误的起因和解决方案。ORA-01555 是最让人讨厌的错误之一。这是许多神话、谬误和不当推测的基础。

注意 ORA-01555 与数据破坏或数据丢失毫无关系。在这方面，这是一个“安全”的错误；惟一的影响是：接收到这个错误的查询无法继续处理。

这个错误实际上很直接，其实只有两个原因，但是其中之一有一个特例，而且这种特例情况发生得如此频繁，所以我要说存在 3 个原因：

- undo 段太小，不足以在系统上执行工作。
- 你的程序跨 COMMIT 获取（实际上这是前一点的一个变体）。我们在上一章讨论了这种情况。
- 块清除。

前两点与 Oracle 的读一致性模型直接相关。从第 7 章可以了解到，查询的结果是预定的，这说明在 Oracle 去获取第一行之前，结果就已经定好了。Oracle 使用 undo 段来回滚自查询开始以来有修改的块，从而提供数据库的一致时间点“快照”。例如执行以下语句：

```
update t set x = 5 where x = 2;

insert into t select * from t where x = 2;

delete from t where x = 2;

select * from t where x = 2;
```

执行每条语句时都会看到 T 的一个读一致视图以及 X=2 的行集，而不论数据库中还有哪些并发的活动。

注意 其他语句也可以看到 T 的读一致视图，这里所示的 4 条语句只是这样的一个例子。它们不作为数据库中单独的事务来运行，因为第一个更新（如果作为单独的事务）可能导致后面 3 条语句看不到记录。这几条语句纯粹是为了说明之用，没有实际意

义。

所有“读”这个表的语句都利用了这种读一致性。在上面所示的例子中，UPDATE 读这个表，找到 X=2 的行（然后 UPDATE 这些行）。INSERT 也要读表，找到 X=2 的行，然后 INSERT，等等。由于两个语句都使用了 undo 段，都是为了回滚失败的事务并提供读一致性，这就导致了 ORA-01555 错误。

前面列的第三项也会导致 ORA-01555，而且这一点更阴险，因为它可能在只有一个会话的数据库中发生，而且这个会话并没有修改出现 ORA-01555 错误时所查询的表！看上去好像不太可能，既然表肯定不会被修改，为什么还需要这个表的 undo 数据呢？稍后将会解释。

在充分说明这三种情况之前，我想先与你分享 ORA-01555 错误的几种解决方案，一般来说可以采用下面的方法：

- 适当地设置参数 UNDO_RETENTION（要大于执行运行时间最长的事务所需的时间）。可以用 V$UNDOSTAT 来确定长时间运行的查询的持续时间。另外，要确保磁盘上已经预留了足够的空间，使 undo 段能根据所请求的 UNDO_RETENTION 增大。
- 使用手动 undo 管理时加大或增加更多的回滚段。这样在长时间运行的查询执行期间，覆盖 undo 数据的可能性就能降低。这种方法可以解决上述的所有 3 个问题。
- 减少查询的运行时间（调优）。如果可能的话，这绝对是一个好办法，所以应该首先尝试这种方法。这样就能降低对 undo 段的需求，不需求太大的 undo 段。这种方法可以解决上述的所有 3 个问题。
- 收集相关对象的统计信息。这有助于避免前面所列的第三点。由于大批量的 UPDATE 或 INSERT 会导致块清除（block cleanout），所以需要在大批量 UPDATE 或大量加载之后以某种方式收集统计信息。

我们还会详细讨论这些方案，因为这些都是必须掌握的重要内容。在真正开始介绍这些解决方案之前，最好先来看看具体情况是怎样的。

1. undo 段确实太小

一种场景是：你的系统中事务很小。正因如此，只需要分配非常少的 undo 段空间。假如，假设存在以下情况：

- 每个事务平均生成 8KB 的 undo。
- 平均每秒完成其中 5 个事务（每秒生成 40KB 的 undo，每分钟生成 2,400KB 的 undo）。
- 有一个生成 1MB undo 的事务平均每分钟出现一次。总的说来，每分钟会生成大约 3.5MB 的 undo。
- 你为系统配置了 15MB 的 undo。

处理事务时，相对于这个数据库的 undo 需求，这完全够了。undo 段会回绕，平均每 3～4 分钟左右会重用一次 undo 段空间。如果要根据执行修改的事务确定 undo 段的大小，那你做得没错。

不过，在同样的环境中，可能有一些报告需求。其中一些查询需要运行相当长的时间，可能是 5 分钟。这就有问题了。如果这些查询需要执行 5 分钟，而且它们需要查询开始时的一个数据视图，你就极有可能遭遇 ORA-01555 错误。由于你的 undo 段会在这个查询执行期间回绕，要知道查询开始以来生成的一些 undo 信息已经没有了，这些信息已经被覆盖。如果你命中了一个块，而这个块几乎在查询开始的同时被修改，这个块的 undo 信息就会因为 undo 段回绕而丢掉，你将收到一个 ORA-01555 错误。

以下是一个小例子。假设我们有一个表，其中有块 1、2、3、…、1,000,000。表 9-4 显示了可能出现的事件序列。

表 9-4 长时间运行的查询时间表

时间（分:秒） 动作

0:00 查询开始

0:01 另一个会话更新（UPDATE）块 1,000,000。将块 1,000,000 的 undo 信息记录到某个 undo 段

0:01 这个 UPDATE 会话提交（COMMIT）。它生成的 undo 数据还在 undo 段中，但是倘若我们需要空间，选择

允许覆盖这个信息

1:00 我们的查询还在运行。现在更新到块 200,000

1:01 进行了大量活动。现在已经生成了稍大于 14MB 的 undo

3:00 查询还在兢兢业业地工作着。现在处理到块 600,000 左右

4:00 undo 段开始回绕，并重用查询开始时（0:00）活动的空间。具体地讲，我们已经重用了原先 0:01 时刻 UPDATE

块 1,000,000 时所用的 undo 段空间

5:00 查询终于到了块 1,000,000。它发现自查询开始以来这个块已经修改过。它找到 undo 段，试图发现对应这

一块的 undo 来得到一个一致读。此时，它发现所需要的信息已经不存在了。这就产生了 ORA-01555 错误，

查询失败

具体就是这样的。如果如此设置 undo 段大小，使得很有可能在执行查询期间重用这些 undo 段，而且查询要访问被修改的数据，那就也有可能不断地遭遇 ORA-01555 错误。此时必须把 UNDO_RETENTION 参数设置得高一些，让 Oracle 负责确定要保留多少 undo 段的大小，让它们更大一些（或者有更多的 undo 段）。你要配置足够的 undo，在长时间运行的

查询期间应当能够维持。在前面的例子中，只是针对修改数据的事务来确定系统 undo 段的大小，而忘记了还有考虑系统的其他组件。

对于 Oracle9i 和以上版本，管理系统中的 undo 有两种方法：

- 自动 undo 管理（Automatic undo management）：采用这种方法，通过 UNDO_RETENTION 参数告诉 Oracle 要把 undo 保留多长时间。Oracle 会根据并发工作负载来确定要创建多少个 undo 段，以及每个 undo 段应该多大。数据库甚至在运行时可以在各个 undo 段之间重新分配区段，以满足 DBA 设置的 UNDO_RETENTION 目标。这是 undo 管理的推荐方法。

- 手动 undo 管理（Manual undo management）：采用这种方法的话，要由 DBA 来完成工作。DBA 要根据估计或观察到的工作负载，确定要手动地创建多少个 undo 段。DBA 根据事务量（生成多少 undo）和长时间运行查询的长度来确定这些 undo 段应该多大。

在手动 undo 管理的情况下，DBA 要确定有多少个 undo 段，以及各个 undo 段有多大，这就产生了一个容易混淆的问题。有人说：“那好，我们已经配置了 XMB 的 undo，但是它们可以增长。我们把 MAXEXTENTS 设置为 500，而且每个区段是 1MB，所以 undo 可以相当大。”问题是，倘若手动地管理 undo 段，undo 段从来不会因为查询而扩大；只有 INSERT、UPDATE 和 DELETE 才会让 undo 段增长。事实上，如果执行一个长时间运行的查询，Oracle 不会因此扩大手动回滚段（即手动管理的回滚段）来保留数据，以备以后可能需要用到这些数据。只有当执行一个长时间运行的 UPDATE 事务时才会扩大手动回滚段。在前面的例子中，即使手动回滚段有增长的潜力，但它们并不会真正增长。对于这样一个系统，你需要有更大的手动回滚段（尽管它们已经很大了）。你要永久地为回滚段分配空间，而不是只给它们自行增长的机会。

对于这个问题，惟一的解决方案只能是适当地设置手动回滚段的大小，从而每 6～10 分钟才回绕，或者让查询执行时间不能超过 2～3 分钟。在这种情况下，DBA 要让永久分配的 undo 量再扩大 2～3 倍。第二种建议也同样适用（也相当有效）。只要能让查询运行得更快，就应该尽力为之。如果自查询开始以来生成的 undo 从未被覆盖，就可以避免 ORA-01555。

在自动 undo 管理的情况下，从 ORA-01555 角度看，问题则要容易得多。无需自行确定 undo 空间有多大并完成预分配，DBA 只有告诉数据库运行时间至少在这段时间内保留 undo。如果已经分配了足够的空间可以扩展，Oracle 就会扩展 undo 段，而不是回绕，从而满足 UNDO_RETENTION 保持时间的要求。这与手动管理的 undo 截然相反，手动管理是会回绕，并尽可能块地重用 undo 空间。这是由于这个原因（即自动 undo 管理支持 UNDO_RETENTION 参数），所以我强烈建议尽可能采用自动 undo 管理。这个参数可以大大降低遭遇 ORA-01555 错误的可能性（只要进行适当地设置！）。

使用手动 undo 管理时，还要记住重要的一点，遇到 ORA-01555 错误的可能性是由系统中最小的回滚段指示的（而非最大的回滚段，也并非平均大小的回滚段）。增加一个“大”回滚段不能解决这个问题。处理查询时只会让最小的回滚段回绕，这个查询就有可能遇到 ORA-01555 错误。使用遗留的回滚段时我主张回滚段大小要相等，以上就是原因所在。如果回滚段的大小都相等，那么每个回滚段即是最小的，也是最大的。这也是我为什么避免使用“最优大小”回滚段的原因。如果你收缩一个此前被扩大的回滚段，就要丢掉以后可能还需要的大量 undo。倘若这么做，会丢掉最老的回滚数据，从而力图使风险最小，但是风险还是存在。我喜欢尽可能在非高峰期间手动地收缩回滚段。

在这方面我有些过于深入了，有介入 DBA 角色之嫌，所以下面讨论另一个话题。重要的是，你要知道出现这种情况下的 ORA-01555 错 误是因为系统没有根据工作负载适当地确定大小。解决方案只有一个，那就是针对工作负载正确地设置大小。这不是你的过错，但是既然遇到了，那就是你的问题 了。这与查询期间临时空间耗尽的情况是一样的。对此可以为系统分配足够的临时空间；或者重写查询，使得所用的查询计划不需要临时空间。

为了演示这种效果，可以建立一个有些人为的小测试。我们将创建一个非常小的 undo 表空间，并有一个生成许多小事务的会话，实际上这能确保这个 undo 表空间回绕，多次重用所分配的空间，而不论 UNDO_RETENTION 设置为多大，因为我们不允许 undo 表空间增长。使用这个 undo 段的会话将修改一个表 T。它使用 T 的一个全表扫描，自顶向下地读表。在另一个会话中，我们将执行一个查询，它通过一个索引读表 T。采用这种方式，这个查询会稍微有些随机地读表：先读第 1 行，然后是第 1,000 行，接下来是第 500 行，再后面是第 20,001 行，如此等等。这样一来，我们可能会非常随机地访问块，并在查询的处理期间多次访问块。这种情况下得到 ORA-01555 错误的机率几乎是 100%。所以，在一个会话中首先执行以下命令：

```
ops$tkyte@ORA10G> create undo tablespace undo_small

  2 datafile size 2m

  3 autoextend off

  4 /

Tablespace created.



ops$tkyte@ORA10G> alter system set undo_tablespace = undo_small;

System altered.
```

现在，我们将建立表 T 来查询和修改。注意我们在这个表中随机地对数据排序。CREATE TABLE AS SELECT 力图按查询获取的顺序将行放在块中。我们的目的只是把行弄乱，使它们不至于认为地有某种顺序，从而得到随机的分布：

```
ops$tkyte@ORA10G> create table t

  2 as

  3 select *

  4 from all_objects

  5 order by dbms_random.random;

Table created.
```

```
ops$tkyte@ORA10G> alter table t add constraint t_pk primary key(object_id)
2 /
Table altered.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T', cascade=> true );
PL/SQL procedure successfully completed.
```

现在可以执行修改了：

```
ops$tkyte@ORA10G> begin
  2        for x in ( select rowid rid from t )
  3        loop
  4                update t set object_name = lower(object_name) where rowid = x.rid;
  5                commit;
  6        end loop;
  7 end;
  8 /
```

在运行这个修改的同时，我们在另一个会话中运行一个查询。这个查询要读表 T，并处理每个记录。获取下一个记录之前处理每个记录所花的时间大约为 1/100 秒（使用 DBMS_LOCK.SLEEP(0.01)来模拟）。在查询中使用了 FIRST_ROWS 提示，使之使用前面创建的索引，从而通过索引（按 OBJECT_ID 排序）来读出表中的行。由于数据是随机地插入到表中的，我们可能会相当随机地查询表中的块。这个查询只运行几秒就会失败：

```
ops$tkyte@ORA10G> declare
  2                cursor c is
  3                        select /*+ first_rows */ object_name
  4                        from t
  5                        order by object_id;
```

```
  6
  7                 l_object_name t.object_name%type;
  8                 l_rowcnt number := 0;
  9          begin
 10                   open c;
 11                   loop
 12                            fetch c into l_object_name;
 13                            exit when c%notfound;
 14                            dbms_lock.sleep( 0.01 );
 15                            l_rowcnt := l_rowcnt+1;
 16                   end loop;
 17                   close c;
 18          exception
 19                   when others then
 20                   dbms_output.put_line( 'rows fetched = ' || l_rowcnt );
 21                   raise;
 22          end;
 23 /
rows fetched = 253
declare
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number 23 with name "_SYSSMU23$"
```

```
too small

ORA-06512: at line 21
```

可以看到，在遭遇 ORA-01555:snapshot too old 错误而失败之前，它只处理了 253 个记录。要修正这个错误，我们要保证做到两点：

- 数据库中 UNDO_RETENTION 要设置得足够长，以保证这个读进程完成。这样数据库就能扩大 undo 表空间来保留足够的 undo，使我们能够完成工作。
- undo 表空间可以增长，或者为之手动分配更多的磁盘空间。

对于这个例子，我认为这个长时间运行的进程需要大约 600 秒才能完成。我的 UNDO_RETENTION 设置为 900（单位是秒，所以 undo 保持大约 15 分钟）。我修改了 undo 表空间的数据文件，使之一次扩大 1MB，直到最大达到 2GB：

```
ops$tkyte@ORA10G> column file_name new_val F

ops$tkyte@ORA10G> select file_name

  2 from dba_data_files

  3 where tablespace_name = 'UNDO_SMALL';



FILE_NAME

------------------------------

/home/ora10g/oradata/ora10g/OR

A10G/datafile/o1_mf_undo_sma_1

729wn1h_.dbf



ops$tkyte@ORA10G> alter database

  2 datafile '&F'

  3 autoextend on

  4 next 1m

  5 maxsize 2048m;
```

```
old       2: datafile '&F'

new       2: datafile '/home/ora10g/.../o1_mf_undo_sma_1729wn1h_.dbf'

Database altered.
```

再次并发地运行这些进程时，两个进程都能顺利完成。这一次 undo 表空间的数据文件扩大了，因为在此允许 undo 表空间扩大，而且根据我设置的 undo 保持时间可知：

```
ops$tkyte@ORA10G> select bytes/1024/1024

  2 from dba_data_files

  3 where tablespace_name = 'UNDO_SMALL';



BYTES/1024/1024

---------------

11
```

因此，这里没有收到错误，我们成功地完成了工作，而且 undo 扩大得足够大，可以满足我们的需要。在这个例子中，之所以会得到错误只是因为我们通过索引来读表 T，而且在全表上执行随机读。如果不是这样，而是执行全表扫描，在这个特例中很可能不会遇到 ORA-01555 错误。原因是 SELECT 和 UPDATE 都要对 T 执行全表扫描，而 SELECT 扫描很可能在 UPDATE 之前进行（SELECT 只需要读，而 UPDATE 不仅要读还有更新，因此可能更慢一些）。如果执行随机读，SELECT 就更有可能要读已修改的块（即块中的多行已经被 UPDATE 修改而且已经提交）。这就展示了 ORA-01555 的“阴险”，这个错误的出现取决于并发会话如何访问和管理底层表。

2. 延迟的块清除

块清除是导致 ORA-01555 错误错误的原因，尽管很难完全杜绝，不过好在毕竟并不多见，因为可能出现块清除的情况不常发生（至少在 Oracle8i 及 以上版本中是这样）。我们已经讨论过块清除机制，不过这里可以做一个总结：在块清除过程中，如果一个块已被修改，下一个会话访问这个块时，可能必须查看最 后一个修改这个块的事务是否还是活动的。一旦确定该事务不再活动，就会完成块清除，这样另一个会话访问这个块时就不必再历经同样的过程。要完成块清除，Oracle 会从块首部确定前一个事务所用的 undo 段，然后确定从 undo 首部能不能看出这个块是否已经提交。可以用以下两种方式完成这种确认。一种方式是 Oracle 可以确定这个事务很久以前就已经提交，它在 undo 段事务表中的事务槽已经被覆盖。另一种情况是 COMMIT SCN 还在 undo 段的事务表中，这说明事务只是稍早前刚提交，其事务槽尚未被覆盖。

要从一个延迟的块清除收到 ORA-01555 错误，以下条件都必须满足：

- 首先做了一个修改并 COMMIT，块没有自动清理（即没有自动完成“提交清除”，例如修改了太多的块，在 SGA 块缓冲区缓存的 10%中放不下）。

- 其他会话没有接触这些块，而且在我们这个“倒霉”的查询（稍后显示）命中这些块之前，任何会话都不会接触它们。
- 开始一个长时间运行的查询。这个查询最后会读其中的一些块。这个查询从 SCN t1 开始，这就是读一致 SCN，必须将数据回滚到这一点来得到读一致性。开始查询时，上述修改事务的事务条目还在 undo 段的事务表中。
- 查询期间，系统中执行了多个提交。执行事务没有接触执行已修改的块（如果确实接触到，也就不存在问题了）。
- 由于出现了大量的 COMMIT，undo 段中的事务表要回绕并重用事务槽。最重要的是，将循环地重用原来修改事务的事务条目。另外，系统重用了 undo 段的区段，以避免对 undo 段首部块本身的一致读。
- 此外，由于提交太多，undo 段中记录的最低 SCN 现在超过了 t1（高于查询的读一致 SCN）。

如果查询到达某个块，而这个块在查询开始之前已经修改并提交，就会遇到麻烦。正常情况下，会回到块所指的 undo 段，找到修改了这个块的事务的状态（换句话说，它会找到事务的 COMMIT SCN）。如果这个 COMMIT SCN 小于 t1，查询就可以使用这个块。如果该事务的 COMMIT SCN 大于 t1，查询就必须回滚这个块。不过，问题是，在这种特殊的情况下，查询无法确定块的 COMMIT SCN 是大于还是小于 t1。相应地，不清楚查询能否使用这个块映像。这就导致了 ORA-01555 错误。

为 了真正看到这种情况，我们将在一个表中创建多个需要清理的块。然后在这个表上打开一个游标，并允许对另外某个表完成许多小事务（不是那个刚更新并打开了游 标的表）。最后尝试为该游标获取数据。现在，我们认为游标需要的数据每问题，应该能看到所有数据，因为我们是在打开游标之前完成并提交了表修改。倘若此时 得到 ORA-01555 错误，就说明存在前面所述的问题。要建立这个例子，我们将使用：

- 2MB UNDO_SMALL undo 表空间（还是这个 undo 表空间）。
- 4MB 的缓冲区缓存，足以放下大约 500 个块。这样我们就可以将一些脏块刷新输出到磁盘来观察这种现象。

首先创建要查询的大表：

```
ops$tkyte@ORA10G> create table big
  2 as
  3 select a.*, rpad('*',1000,'*') data
  4 from all_objects a;
Table created.
```

```
ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'BIG' );

PL/SQL procedure successfully completed.
```

由于使用了这么大的数据字段，每个块中大约有 6～7 行，所以这个表中有大量的块。接下来，我们创建将由多个小事务修改的小表：

```
ops$tkyte@ORA10G> create table small ( x int, y char(500) );

Table created.



ops$tkyte@ORA10G> insert into small select rownum, 'x' from all_users;

38 rows created.

ops$tkyte@ORA10G> commit;

Commit complete.



ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'SMALL' );

PL/SQL procedure successfully completed.
```

下面把那个大表“弄脏”。由于 undo 表空间非常小，所以希望尽可能多地更新这个大表的块，同时生成尽可能少的 undo。为此，将使用一个有意思的 UPDATE 语句来执行该任务。实质上讲，下面的子查询要找出每个块上的“第一个”行 rowid。这个子查询会返回每一个数据库块的一个 rowid（标识了这个块是的一行）。我们将更新这一行，设置一个 VARCHAR2(1)字段。这样我们就能更新表中的所有块（在这个例子中，块数大约比 8,000 稍多一点），缓冲区缓存中将会充斥着必须写出的脏块（现在只有 500 个块的空间）。仍然必须保证只能使用那个小 undo 表空间。为做到这一点，而且不超过 undo 表空间的容量，下面构造一个 UPDATE 语句，它只更新每个块上的“第一行”。ROW_NUMBER()内置分析函数是这个操作中使用的一个工具；它把数字 1 指派给表中的数据库块的“第 1 行”，在这个块上只会更新这一行：

```
ops$tkyte@ORA10G> alter system set undo_tablespace = undo_small;

System altered.



ops$tkyte@ORA10G> update big

  2                 set temporary = temporary
```

```
  3          where rowid in
  4                  (
  5                  select r
  6                  from (
  7                          select rowid r, row_number() over
                                     (partition   by   dbms_rowid.rowid_block_number(rowid)
order by rowid) rn
  8                          from big
  9                          )
  10                  where rn = 1
  11          )
  12 /
8045 rows updated.



ops$tkyte@ORA10G> commit;
Commit complete.
```

现在我们知道磁盘上有大量脏块。我们已经写出了一些，但是没有足够的空间把它们都放下。接下来打开一个游标，但是尚未获取任何数据行。要记住，打开游标时，结果集是预定的，所以即使 Oracle 并没有具体处理一行数据，打开结果集这个动作本身就确定了“一致时间点”，即结果集必须相对于那个时间点一致。现在要获取刚刚更新并提交的数据，而且我们知道没有别人修改这个数据，现在应该能获取这些数据行而不需要如何 undo。但是此时就会“冒出”延迟块清除。修改这些块的事务太新了，所以 Oracle 必须验证在我们开始之前这个事务是否已经提交，如果这个信息（也存储在 undo 表空间中）被覆盖，查询就会失败。以下打开了游标：

```
ops$tkyte@ORA10G> variable x refcursor
ops$tkyte@ORA10G> exec open :x for select * from big;
PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10G> !./run.sh
```

run.sh 是一个 shell 脚本。其中使用一个命令启动 9 个 SQL*Plus 会话：

```
$ORACLE_HOME/bin/sqlplus / @test2 1 &
```

这里为每个 SQL*Plus 会话传递一个不同的数字（这里是数字 1，还有 2、3 等）。每个会话运行的脚本 test2.sql 如下：

```
begin
        for i in 1 .. 1000
        loop
                update small set y = i where x= &1;
                commit;
        end loop;
end;
/
exit
```

这样一来，就有了 9 个会话分别在一个循环中启动多个事务。run.sh 脚本等待这 9 个 SQL*Plus 会话完成其工作，然后返回我们的会话，也就是打开了游标的会话。在视图大约时，我们观察到下面的结果：

```
ops$tkyte@ORA10G> print x

ERROR:

ORA-01555: snapshot too old: rollback segment number 23 with name "_SYSSMU23$"

too small

no rows selected
```

前面已经说过，以上是一种很少见的情况。它需要许多条件，所有这些条件必须同时存在才会出现这种情况。首先要有需要清理的块，而这种块在 Oracle8i 及以前的版本中很少见。收集统计信息的 DBMS_STATS 调用就能消除这种块。尽管大批量的更新和大量加载是造成块清除最常见的理由，但是利用 DBMS_STATS 调用的话，这些操作就不再成为问题，因为在这种操作之后总要对表执行分析。大多数事务只会接触很少的块，而不到块缓冲区缓存的 10%；因此，它们不会生成需要清理的块。万一你发现遭遇了这个问题，即选择

（SELECT）一个表时（没有应用其他 DML 操作）出现了 ORA-01555 错误，能你可以试试以下解决方案：

- 首先，保证使用的事务“大小适当”。确保没有不必要地过于频繁地提交。
- 使用 DBMS_STATS 扫描相关的对象，加载之后完成这些对象的清理。由于块清除是极大量的 UPDATE 或 INSERT 造成的，所以很有必要这样做。
- 允许 undo 表空间扩大，为之留出扩展的空间，并增加 undo 保持时间。这样在长时间运行查询期间，undo 段事务表中的事务槽被覆盖的可能性就会降低。针对导致 ORA-01555 错误的另一个原因（undo 段太小），也同样可以采用这个解决方案（这两个原因有紧密的关系；块清除问题就是因为处理查询期间遇到了 undo 段重用，而 undo 段大小正是重用 undo 段的一个根本原因）。实际上，如果把 undo 表空间设置为一次自动扩展 1MB，而且 undo 保持时间为 900 秒，再运行前面的例子，对表 BIG 的查询就能成功地完成了。
- 减少查询的运行时间（调优）。如果可能的话，这总是件好事，所以应该首先尝试这样做。

9.6 小结

在这一章中，我们介绍了 redo 和 undo，并说明了 redo 和 undo 对开发人员有什么意义。这里我主要强调的是作为开发人员要当心的问题，因为有关 redo 和 undo 的问题实际上应该有 DBA 或 SA 负责修正。重要的是，从这一章应该了解到 redo 和 undo 的重要性，要知道它们绝对不是开销，而是数据库的组成部分，它们是必要的，甚至是必不可少的。一旦很好地了解了 redo 和 undo 如何工作，以及它们做些什么，你就能更好地加以利用。要知道，如果不必要过于频繁地提交，你不仅不会因此“节省”任何资源，实际上反而会浪费资源，这才是重点；因为这会占用更多的 CPU 时间、更多的磁盘，还有做更多的编程工作。应该清楚数据库需要做什么，然后才让数据库真正地去做。

第 10 章 数据库表

在这一章中，我们将讨论各种类型的数据库表，并介绍什么情况下想用哪种类型的数据库表（也就是说，在哪些情况下某种类型的表比其他类型更适用）。我们会强调表的物理存储特征：即数据如何组织和存储。

从前只有一种类型的表，这千真万确，原先确实只有一种“普通”表。管理这种表就像管理“一个堆”一样（下一节会给出有关的定义）。后来，Oracle 又增加 了几类更复杂的表。如今，除了堆组织表外，还有聚簇表（共有 3 种类型的聚簇表）、索引组织表、嵌套表、临时表和对象表。每种类型的表都有不同的特征，因此 分别适用于不同的应用领域。

10.1 表类型

在深入讨论细节之前，我们先对各种类型的表给出定义。Oracle 中主要有 9 种表类型：

- 堆 组织表（heap organized table）：这些就是“普通”的标准数据库表。数据以堆的方式管理。增加数据时，会使用段中找到的第一个能放下此数据的自由空间。从表中删除数据时，则 允许以后的 INSERT 和 UPDATE 重用这部分空间。这就是这种表类型中的“堆”这个名词的由来。堆（heap）是一组空间，以一种有些随机的方式使 用。

- 索引组织表（index organized table）：这些表按索引结构存储。这就强制要求行本身有某种物理顺序。在堆中，只要放得下，数据可以放在任何位置；而索引组织表（IOT）有所不同，在 IOT 中，数据要根据主键有序地存储。

- 索 引聚簇表（index clustered table）：聚簇（cluster）是指一个或多个表组成的组，这些表物理地存储在相同的数据库块上，有相同聚簇键值的所有行会相邻地物理存储。这种结 构可以实现两个目标。首先，多个表可以物理地存储在一起。一般而言，你可能认为一个表的数据就在一个数据库块上，但是对于聚簇表，可能把多个表的数据存储 在同一个块上。其次，包含相同聚簇键值（如 DEPTNO=10）的所有数据会物理地存储在一起。这些数据按聚簇键值“聚簇”在一起。聚簇键使用 B*树索引 建立。

- 散 列聚簇表（hash clustered table）：这些表类似于聚簇表，但是不使用 B*树索引聚簇键来定位数据，散列聚簇将键散列到聚簇上，从而找到数据应该在哪个数据库块上。在散列聚簇 中，数据就是索引（这是隐喻的说法）。如果需要频繁地通过键的相等性比较来读取数据，散列聚簇表就很适用。

- 有 序散列聚簇表（sorted hash clustered table）：这种表类型是 Oracle 10g 中新增的，它结合了散列聚簇表的某些方面，同时兼有 IOT 的一些方面。其概念如下：你的行按某个键值（如 CUSTOMER_ID）散列，而与该键相 关的一系列记录以某种有序顺序到达（因此这些记录是基于时间戳的记录），并按这种有序顺序处理。例如，客户在你的订单输入系统中下订单，这些订单会按先进 先出（first in, first out, FIFO）的方式获取和处理。在这样一个系统中，有序散列聚簇就是适用的数据结构。

- 嵌 套表（nested table）：嵌套表是 Oracle 对象关系扩展的一部分。它们实际上就是系统生成和维护的父/子关系中的子表。嵌套表的工作类似于 SCOTT 模式中的 EMP 和 DEPT。可以认为 EMP 是 DEPT 表的子表，因为 EMP 表有一个指向 DEPT 的外键 DEPTNO。嵌套表与子表的主要区别是：嵌套表不像子表 （如 EMP）那样是“独立”表。

- 临时表（temporary table）：这些表存储的是事务期间或会话期间的“草稿”数据。临时表要根据需要从当前用户的临时表空间分配临时区段。每个会话只能看到这个会话分配的区段；它从不会看到其他任何会话中创建的任何数据。

- 对象表（object table）：对象表基于某种对象类型创建。它们拥有非对象表所没有的特殊属性，如系统会为对象表的每一行生成 REF（对象标识符）。对象表实际上是堆组织表、索引组织表和临时表的特例，还可以包含嵌套表作为其结构的一部分。

- 外 部表（external table）：这些表并不存储在数据库本身中，而是放在数据库之外，即放在平常的操作系统文件中。在 Oracle9i 及以上版本中，利用外部表可以查询数 据库之外的一个文件，就好像这个文件也是数据库中平常的表一样。外部表对于向数据库加载数据最有用（外部表是非常强大的数据加载工具）。Oracle 10g 则更进一步，还引入了一个外部表卸载功能，在不使用数据库链接的情况下，这为在 Oracle 数据库之间移动数据提供了一种简单的方法。我们将在第 15 章更详细地讨论外部表。

- 不论哪种类型的表，都有以下一般信息：
- 一 个表最多可以有 1000 列，不过我不鼓励设计中真的包含这么多列，除非存在某个硬性需求。表中的列数远远少于 1000 列时才最有效。Oracle 在内部会 把列数大于 254 的行存储在多个单独的行段（row piece）中，这些行段相互指向，而且必须重新组装为完整的行影像。
- 表 的行数几乎是无限的，不过你可能会遇到另外某个限制，使得这种“无限”并不实际。例如，一般来讲，一个表空间最多有 1022 个文件（不过，Oracle 10g 中有一些新的 BIGFILE 表空间，这些表空间可以超出上述文件大小限制）。假设你有一些 32GB 的文件，也就是说，每个表空间有 32,704GB，就会有 2,143,289,344 个块，每个块大小为 16KB。你可能在每个块上放 160 行（每行大约 80～100 字节 ）。这样就会有 342,926,295,040 行。不过，如果对这个表分区，这个行数还能很容易地翻倍。例如，假设一个表有 1024 个散列分区，则能有 1024× 342,926,295,040 行。确实存在着上限，但是在接近这些上限之前，你肯定会遇到另外某个实际限制。
- 表中的列有多少种排列（以及这些列的函数有多少种排列），表就可以有多少个索引。随着基于函数的索引的出现，理论上讲，说能创建的索引数是无限的！不过，同样由于存在一些实际的限制，这会影响真正能创建和维护的索引数。
- 即使在一个数据库中也可以有无限多个表。不过，还是同样的道理，实际的限制会使数据库中的表数在一个合理的范围内。不可能有数百万个表（这么多表对于创建和管理来说都是不实际的），但是有数千个表还是允许的。

在下一节中，我们将讨论与表相关的一些参数和术语。在此之后，我们再转而讨论基本的堆组织表，然后介绍其他类型的表。

10.2 术语

在 这一节中，我们将介绍与表相关的各种存储参数和术语。并非每种表类型都会用到所有参数。例如，PCTUSED 参数在 IOT 环境中就没有意义。具体讨论各种 表类型时还会分别介绍与之相关的参数。这一节的目标时介绍这些术语，并给出定义。在后面几节中，还会在适当的时候介绍使用特定参数的更多信息。

10.2.1 段

Oracle 中的段（segment）是占用磁盘上存储空间的一个对象。尽管有多种类型，不过最常见的段类型如下：

- 聚簇（cluster）：这种段类型能存储表。有两种类型的聚簇：B*树聚簇和散列聚簇。聚簇通常用于存储多个表上的相关数据，将其“预联结”存储到同一个数据库块上；还可以用于存储一个表的相关信息。“聚簇”这个词是指这个段能把相关的信息物理的聚在一起。
- 表（table）：表段保存一个数据库表的数据，这可能是最常用的段类型，通常与索引段联合使用。
- 表 分区（table partition）或子分区（subpartition）：这种段类型用于分区，与表段很相似。分区表由一个或多个分区段（table partition segment）组成，组合分区表则由一个或多个表子分区段（table subpartition segment）组成。

- 索引（index）：这种段类型可以保存索引结构。
- 索引分区（index partition）：类似与表分区，这种段类型包含一个索引的某个片。分区索引由一个或多个索引分区段（index partition segment）组成。
- Lob 分 区（lob partition）、lob 子分区（lob subpartition）、lob 索引（lobindex）和 lob 段（lobsegment）：lobindex 和 lobsegment 段保存大对象（large object 或 LOB）的结构。对包含 LOB 的表分区时，lobsegment 也会分区，lob 分区段（lob partition segment）正是用于此。有意思的是，并没有一种 lobindex 分区段（lobindex partition segment）类型——不论出于什么原因，Oracle 将分区 lobindex 标记为一个索引分区（有人很奇怪为什么要另外给 lobindex 取一个特 殊的名字！）。
- 嵌套表（nested table）：这是为嵌套表指定的段类型，它是主/明细关系中一种特殊类型的“子”表，这种关系随后将详细讨论。
- 回滚段（rollback）和 Type2 undo 段：undo 数据就存储在这里。回滚段是 DBA 手动创建的段。Type2 undo 段由 Oracle 自动创建和管理。

举 例来说，一个表可以是一个段。索引有可能是一个段。这里我强调了“可能”，这是因为，我们可以把一个索引划分到不同的段中。所以，索引对象本身只是一个定 义，而不是一个物理段，索引可能由多个索引分区组成，而每个索引分区（index partition）是一个段。表可能是一个段，也可能不是。由于同样的原因，由于表分区，一个表可以有多个表段：或者可以在一个称为聚簇的段中创建一个 表，此时这个表可能与其他表同在一个聚簇段中。

不 过，最常见的情况是，表是一个段，索引也是一个段。对现在来说，这样考虑最简单。创建一个表时，通常就是创建一个新的表段，而且如第 3 章所述，这个段包含 区段，区段则包含块。这是平常的存储层次结构。但是要指出重要的一点，只在“通常”情况下才有这种一对一的关系。例如，考虑以下这个简单的 CREATE TABLE 语句：

```
Create table t ( x int primary key, y clob, z blob );
```

这个语句创建 6 个段。如果在一个初始为空（什么也没有）的模式中发出下面的 CREATE TABLE 语句，会观察到以下结果：

```
ops$tkyte@ORA10G> select segment_name, segment_type

      2 from user_segments;

no rows selected


ops$tkyte@ORA10G> create table t ( x int primary key, y clob, z blob );

Table created.


ops$tkyte@ORA10G> select segment_name, segment_type

      2 from user_segments;

SEGMENT_NAME                   SEGMENT_TYPE
```

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

SYS_IL0000063631C00002$$     LOBINDEX
SYS_LOB0000063631C00003$$   LOBSEGMENT
SYS_C009783                    INDEX
SYS_IL0000063631C00003$$     LOBINDEX
SYS_LOB0000063631C00002$$   LOBSEGMENT
T                                 TABLE
6 rows selected.
```

在这个例子中，表本身创建了一个段：如输出中最后一行所示。这里主键约束创建了一个索引段，以保证惟一性。

注意 惟一约束或主键可能创建一个新索引，也可能不创建。如果约束列上已经有一个索引，而且这些列处于索引的前几列，这个约束就会（而且将会）使用这些列（而不再创建创建新索引）。

另外，每个 LOB 列分别创建了两个段：一个段用于存储字符大对象（character large object, CLOB）或二进制大对象（binary large object, BLOB）所指的实际数据块，另一个段用于“组织”这些数据块。LOB 为非常大块的信息提供了支持，可以多达几 GB。LOB 存储在 lobsegment 的块中，lobindex 用于跟踪这些 LOB 块在哪里，以及应该以何种顺序来访问它们。

10.2.2 段空间管理

从 Oracle 9i 开始，管理段空间有两种方法：

- 手 动段空间管理（Manual Segment Space Management）：由你设置 FREELISTS、FREELIST GROUPS、PCTUSED 和其他参数来控制如何分配、使用和重用段中的空间。在这一章中我会把这种空间管理方法称为 MSSM，但是要记住，这是一个我 自造的缩写，Oracle 文档中没有这个缩写。

- 自动段空间管理（Automatic Segment Space Management, ASSM）：你只需控制与空间使用相关的一个参数：PCTFREE。创建段时也可以接受其他参数，但是这些参数将被忽略。

MSSM 是 Oracle 的遗留实现。它已经存在多年，许多版本都支持 MSSM。ASSM 则在 Oracle 9i Release 1 中 才首次引入。原先用于控制空间分配和提供高并发性的参数数不胜数，并且需要对这些参数进行调整，人们不希望还要这么做，这正是设计 ASSM 的出发点。例 如，倘若将 FREELISTS 参数设置为默认值 1，可能会出现，如果你的段是插入/更新新密集的（有大量插入/更新操作），对自由空间的分配就会存在竞 争。Oracle 要在表中插入一行，或更新一个索引键条目，或者由于更新一行而导致这一行迁移时（稍后还会更多地介绍这方面的内容），可能需要从与这个段 相关的自由块列表中得到一个块。如果只有一个自由块列表，一次就只有一个事务能查看和修改这个列表，事务之间必须相互等待。在这种情况下，如果有多个 FREELISTS 和 FREELIST GROUPS，就能提高并发性，因为事务可以分别查看不同的列表，而不会相互竞争。

稍后讨论存储设置时，我还会提到哪些参数用于手工段空间管理，而哪些参数用于自动段空间管理，不过需要指出，在存储/段特征这方面，应用于 ASSM 段的存储设置只有：

- BUFFER_POOL
- PCTFREE
- INITRANS
- MAXTRANS（仅用于 9i；在 10g 中，所有段都会忽略这个参数）

其他存储和物理属性参数都不适用于 ASSM 段。

段空间管理是从段的表空间（而且段从不会跨表空间）继承来的一个属性。段要使用 ASSM，就必须位于支持 ASSM 空间管理的表空间中。

10.2.3 高水位线

存储在数据库中的表段使用了这个术语。例如，如果把表想象成一个“平面”结构，或者想象成从左到右依次排开的一系列块，高水平线（high-water mark，HWM）就是包含了数据的最右边的块，如图 10－1 所示。

图 10-1 HWM 示意图

图 10 －1 显示了 HWM 首先位于新创建表的第一个块中。过一段时间后，随着在这个表中放入数据，而且使用了越来越多的块，HWM 会升高。如果我们删除了表中的一 些（甚至全部）行，可能就会有许多块不再包含数据，但是它们仍在 HWM 之下，而且这些块会一直保持在 HWM 之下，直到重建、截除或收缩这个对象（将段收缩 是 Oracle 10g 的一个新特性，只有当段在一个 ASSM 表空间中时才支持这个特性）。

HWM 很重要，因为 Oracle 在全面扫描段时会扫描 HWM 之下的所有块，即使其中不包含任何数据。这会影响全面扫描的性能，特别是当 HWM 之下的大多数块都为空时。要查看这种情况，只需创建一个有 1,000,000 行的表（或者创建其他有大量行的表），然后对这个表执行一个 SELECT COUNT(*)。 下面再删除（DELETE）这个表中的每一行，你会发现尽管 SELECT COUNT(*)统计出 0 行，但是它与统计出 1,000,000 所花的时间一样长（如

果需要完成块清除，时间可能还会更长：有关内容请参加 9.5.5 节）。这是因为 Oracle 在忙于读取 HWM 之下的所有块，查看其中是否包含数据。如果对这个表使用 TRUNCATE 而不是删除其中的每一行，你可以比较 一下结果有什么不同。TRUNCATE 会把表的 HWM 重置回“0”，还会截除表上的相关索引。由于以上原因，如果你打算删除表中的所有行，就应该选择使用 TRUNCATE（如果可以使用的话）。

在 一个 MSSM 表空间中，段只有一个 HWM。不过，在 ASSM 表空间中，除了一个 HWM 外，还有一个低 HWM（见图 10－2）。在 MSSM 中，HWM 推进时 （例如，插入行时），所有块都会并立即有效，Oracle 可以安全地读取这些块。不过，对于 ASSM，HWM 推进时，Oracle 并不会立即格式化所有 块，只有在第一次使用这些块时才会完成格式化，以便安全地读取。所以，全面扫描一个段时，必须知道要读取的块是否“安全”或是否格式化，（这说明，其中不 包含有意义的信息，不能对其进行处理）。为了避免表中每一个块都必须经过这种安全/不安全检查，Oracle 同时维护了一个低 HWM 和一个 HWM。 Oracle 会全表扫描至 HWM，对于低 HWM 以下的所有块会直接读取并加以处理。而对介于低 HWM 和 HWM 之间的块，则必须更加小心，需要参考管理这些 块所用的 ASSM 位图信息来查看应该读取哪些块，而哪些块应该被忽略。

图 10-2 低 HWM 示意图

10.2.4 freelists

使用 MSSM 表空间时，Oracle 会在自由列表（freelist）中为有自由空间的对象维护 HWM 一些的块。

注意 freelists 组和 freelist 组在 ASSM 表空间中根本就没有；仅 MSSM 表空间使用这个技术。

每 个对象都至少有一个相关的 freelist，使用块时，可能会根据需要把块放在 freelist 上或者从 freelist 删除。需要说明的重要一点是，只 有位于 HWM 以下的对象块才会出现在 freelist 中。仅当 freelist 为空时才会使用 HWM 之上的块，此时 Oracle 会推进 HWM，并把这些块 增加到 freelist 中，采用这种方式，Oracle 会延迟到不得已时才增加对象的 HWM。

一个对象可以有多个 freelist。如果预计到会有多个并发用户在一个对象上执行大量的 INSERT 或 UPDATE 活动，就可以配置多个 freelist，这对性能提升很有好处（但是可能要以额外的存储空间为代价）。根据需要配置足够多的 freelist 非常重要。

如果存在多个并发的插入和更新，在这样一个环境中，FREELISTS 可能对性能产生巨大的影响（可能是促进，也可能是妨碍）。通过一个极其简单的测试就能看出正确地设置 FREELISTS 有什么好处。请考虑下面这个相对简单的表：

```
ops$tkyte@ORA10GR1> create table t ( x int, y char(50) ) tablespace MSSM;

Table created.
```

接 下来使用 5 个并发会话，开始“疯狂地”对这个表执行插入。如果分别测量插入前和插入后与块相关的系统级等待事件，就会发现长时间的等待，特别是对数据块的 等待（试图插入数据）。这通常是因为表（以及索引）上的 freelist 不足造成的（不过关于索引的有关内容将在下一章更详细介绍）。为此我使用了 Statspack，首先取一个 statspace.snap，接下来执行一个脚本开始 5 个并发的 SQL*Plus 会话，等这些会话退出后再取另一个 statspace.snap。这些会话运行的脚本很简单，如下：

```
begin
        for i in 1 .. 100000
        loop
                insert into t values ( i, 'x' );
        end loop;
        commit;
end;
/
exit;
```

这 是一个非常简单的代码块，此时我是数据库中惟一的用户。按理说，应该得到最佳的性能，因为我配置了充足的缓冲区缓存，重做日志大小很合适，另外索引也不会 减慢速度，而且这是在有两个超线程 Xeon CPU 的主机上运行，这个主机应该能运行得很快。不过，我看到的结果却是：

```
                   Snap Id Snap Time                Sessions Curs/Sess Comment

                   --------- ------------------      -------- --------- ---------------
-----

Begin Snap:   793          29-Apr-05 13:45:36      15              3.9

End Snap:         794          29-Apr-05 13:46:34      15              5.7

Elapsed:                     0.97 (mins)

Top 5 Timed Events

~~~~~~~~~~~~~~~~~~                                                   % Total

Event                                          Waits        Time (s)       Call Time

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

```
CPU time                                    165          53.19

buffer busy waits                 368,698   119          38.43

log file parallel write             1,323    21           6.86
latch: cache buffers chains           355     2            .67
enq: HW - contention                2,828     1            .24
```

对 buffer busy waits 总共等待了 119 秒，也就是每个会话大约 24 秒。导致这些等待的原因完全是：表中没有配置足够的 freelist 来应付发生的这种并发活动。不过，只需将表创建为有多个 freelist，就能轻松地消除大部分等待时间：

```
ops$tkyte@ORA10GR1> create table t ( x int, y char(50) )

        2 storage( freelists 5 ) tablespace MSSM;
Table created.
```

或者也可以通过修改对象达到目的：

```
ops$tkyteORA10GR1> alter table t storage ( FREELISTS 5 );

Table altered.
```

你会看到，buffer busy waits 大幅下降，而且所需的 CPU 时间也随着耗用时间的下降而减少（因为这里做的工作更少；对闩定数据结构的竞争确实会让 CPU 焦头烂额）：

```
                  Snap Id Snap Time                Sessions Curs/Sess Comment

                  ---------      ------------------            --------        ---------        ----------
-----

Begin Snap:   809          29-Apr-05 14:04:07      15              4.0

End Snap:         810          29-Apr-05 14:04:41      14              6.0

Elapsed:                       0.57 (mins)


Top 5 Timed Events

~~~~~~~~~~~~~~~~~~                                                 % Total

Event                                          Waits         Time (s)       Call Time

---------------------------------------------          ------------         -----------          ---------

CPU time                                          122                          74.66

buffer busy waits                              76,538              24           14.94
```

log file parallel write	722	14	8.45
latch: cache buffers chains	144	1	.63
enq: HW - contention	678	1	.46

对 于一个表来说，你可能想确定最多能有多少个真正的并发插入或更新（这需要更多空间）。这里我所说的“真正的并发”是指，你认为两个人在同一时刻请求表中一 个自由块的情况是否频繁。这不是对重叠事务的一种量度；而是量度多少个会话在同时完成插入，而不论事务边界是什么。你可能希望对表的并发插入有多少， freelist 就有多少，以此来提高并发性。

只 需把 freelist 设置得相当高，然后就万事大吉了，是这样吗？当然不是，哪有这么容易。使用多个 freelist 时，有一个主 freelist，还有 一些进程 freelist。如果一个段只有一个 freelist，那么主 freelist 和进程 freelist 就是这同一个自由列表。如果你有两个 freelist，实际上将有一个主 freelist 和两个进程 freelist。对于一个给定的会话，会根据其会话 ID 的散列值为之指定一个进程 freelist。目前，每个进程 freelist 都只有很少的块，余下的自由块都在主 freelist 上。使用一个进程 freelist 时，它会根据需 要从主 freelist 拉出一些块。如果主 freelist 无法满足空间需求，Oracle 就会推进 HWM，并向主 freelist 中增加空块。过一段时 间后，主 freelist 会把其存储空间分配多个进程 freelist（再次说明，每个进程 freelist 都只有为数不多的块）。因此，每个进程会使用 一个进程 freelist。它不会从一个进程 freelist 到另一个进程 freelist 上寻找空间。这说明，如果一个表上有 10 个进程 freelist，而且你的进程所用的进程 freelist 已经用尽了该列表中的自由缓冲区，它不会到另一个进程 freelist 上寻找空间，即使另外 9 个进程 freelist 都分别有 5 块（总共有 45 个块），此时它还是会去求助主 freelist。假设主 freelist 上的空间无法满足这样一个自由块 请求，就会导致表推进 HWM，或者如果表的 HWM 无法推进（所有空间都已用），就要扩展表的空间（得到另一个区段）。然后这个进程仍然只使用其 freelist 上的空间（现在不再为空）。使用多个 freelist 时要有所权衡。一方面，使用多个 freelist 可以大幅度提升性能。另一方面，有 可能导致表不太必要地使用稍多的磁盘空间。你必须想清楚在你的环境中哪种做法麻烦比较小。

不 要低估了 FREELISTS 参数的用处，特别是在 Oracle 8.1.6 及以后版本中，你可以根据意愿自由地将其改大或改小。可以把它修改为一个大数，从而与采用传统路径模式的 SQL*Loader 并行完成数据的加 载。这样可以获得高度并发的加载，而只有最少的等待。加载之后，可以再把这个值降低为某个更合理的平常的数。将空间改小时，现有的多个 freelist 上 的块要合并为一个主 freelist。

要解决前面提到的缓冲区忙等待问题，还有一种方法，这就是使用一个 ASSM 管理的表空间。还是前面的例子，但在 ASSM 管理的表空间中要如下创建表 T：

```
ops$tkyte@ORA10GR1> create tablespace assm

    2 datafile size 1m autoextend on next 1m

    3 segment space management auto;

Tablespace created.


ops$tkyte@ORA10GR1> create table t ( x int, y char(50) ) tablespace ASSM;

Table created.
```

你会看到，在这种情况下，缓冲区忙等待、CPU 时间和耗用时间都会下降，在此不必确定最好要有多少个 freelist：

```
                  Snap Id Snap Time            Sessions Curs/Sess Comment
                  ------- ------------------   -------- --------- -------------
Begin Snap:   812 29-Apr-05 14:12:37      15       3.9
End Snap:     813 29-Apr-05 14:13:07      15       5.6
Elapsed:          0.50 (mins)

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                       % Total
Event                                            Waits   Time (s)   Call Time
---------------------------------------------- --------- ---------- ---------
CPU time                                           107                78.54
log file parallel write                            705         12      9.13
buffer busy waits                               12,485         12      8.52
latch: library cache                               68          1       .70
LGWR wait for redo copy                         3,794          1       .47
```

这就是 ASSM 的主要作用之一：不必手动地确定许多关键存储参数的正确设置。

10.2.5 PCTFREE 和 PCTUSED

一 般而言，PCTFREE 参数用来告诉 Oracle 应该在块上保留多少空间来完成将来的更新。默认情况下，这个值是 10%。如果自由空间的百分比高于 PCTFREE 中的指定值，这个块就认为是“自由的”。PCTUSED 则告诉 Oracle 当前不“自由”的一个块上自由空间百分比需要达到多大才能使它再 次变为自由的。默认值是 40%[1]。

如 前所述，对于一个表（而不是一个 IOT，有关内容稍后再介绍），PCTFREE 会告诉 Oracle：块上应该保留多大的空间来完成将来的更新。这说明，如 果我们使用的块大小为 8KB，只要向块中增加一个新行，就会导致块上的自由空间下降大约 800 字节，Oracle 会使用 FREELIST 的另一个块，而不 是现有的块。块上这 10%的数据空间会预留出来，以便更新该块上的行。

1. 实际上 PCTUSED 的含义是，如果块上不自由的空间到达或小于 PCTUSED 参数指定的百分比时，这个块将重新变为自由，如倘若 PCTUSED 为 40％，那么块上不自由的空间小于 40％时，即自由空间达到 60％时，这个块就重新变为自由。——译者注。

注意 对 于不同的表类型，PCTFREE 和 PCTUSED 的实现有所不同。对于某些表类型，这两个参数都要使用，而另外一些表类型只使用 PCTFREE，而且对于 这些表类型，仅当创建对象时才会使用 PCTFREE。IOT 在创建时可以使用 PCTFREE 在表中预览空间来完成将来的更新，但是在其他方面并不使用 PCTFREE，例如，PCTFREE 不用于决定何时停止向一个给定块中插入行。

根 据你使用的是 ASSM 表空间还是 MSSM 表空间，这两个参数的实际作用会有所不同。使用 MSSM 时，这些参数设置控制着块何时放入 freelist 中，以 及何时从 freelist 中取出。如果使用默认值：PCTFREE 为 10，PCTUSED 为 40，那么在块到达 90%满之前（有 10%以上的自由空间）， 这个块会一直在 freelist 上。一旦到底 90%，就会从 freelist 中取出，而且直到块上的自由空间超过了块的 60%时，才会重新回到 freelist 上，在此之前，这个块一直不在 freelist 上。

使用 ASSM 时，PCTFREE 仍然会限制能否将一个新行插入到一个块中，但是它不会控制一个块是否在 freelist 上，因为 ASSM 根本不使用 freelist。在 ASSM 中，PCTUSED 将被忽略。

PCTFREE 有 3 种设置：太高、太低好刚好。如果把块的 PCTFREE 设置得过高，就会浪费空间。如果把 PCTFREE 设置为 50%，而你从未更新数据，那么每个块都会浪 费 50%的空间。不过，在另一个表上，50%可能非常合理。如果行初始很小，现在想将行的大小加倍，但是倘若 PCTFREE 设置得太小，更新行时就会导致 行迁移。

1. 行迁移

到底什么是行迁移？行迁移（row migration）是指由于某一行变得太大，无法再与其余的行一同放在创建这一行的块中（块中已经放不下这一行），这就要求这一行离开原来的块。这一节将分析行迁移。首先来看一个块，如同 10-3 所示。

图 10-3 更新前的数据块

这 个块上大约 1/7 是自由空间。不过，我们想通过一个 UPDATE 将第 4 行所有的空间加倍（第 4 行现在占用了块上 1/7 的空间）。在这种情况下，即使 Oracle 合并了块上的空间（如同 10-4 所示），还是没有足够的空间将第 4 行的大小加倍，因为自由空间小于第 4 行的当前大小。

图 10-4　合并自由空间之后可能得到的数据块

如 果这一行能在合并的空间中放下，自然就会这么做。不过，在此 Oracle 没有完成这个合并，块还是保持原样。因为第 4 行如果还呆在这个块上，它就必须跨 块，所以 Oracle 会移动或迁移这一行。不过，Oracle 不能简单地移动这一行，它必须留下一个“转发地址”。可能有一些索引物理地指向第 4 行的这个 地址。简单的更新不会同时修改这些索引（注意对于分区表则有一个特例：更新分区表时，rowid 即行地址会改变。这种情况将在第 13 章介绍）。因此， Oracle 迁移这一行时，它会留下一个指针，指示这一行实际上在什么位置。更新之后，块可能如图 10-5 所示。

图 10-5　迁移行示意图

因 此，迁移行（migrated row）就是这一行从最初所插入的块上移到另外的某个块上。为什么这会带来问题？你的应用绝对不会知道存在行迁移；你使用的 SQL 也没有任何不同。行迁移 只会影响性能。如果你通过一个索引来读这一行，索引会指向原来的块，那个块再指向这个新块。要得到具体的行数据，一般并不是执行两个左右的 I/O 就可以得 到行数据。单独来看，这不是大问题，甚至根本注意不到。不过，如果这种行所占的比例相当大，而且有大量用户在访问这些行，你就会注意到这种副作用了。访问 这些数据的速度开始变慢（额外的 I/O 以及与 I/O 相关的闩定都会增加访问时间），缓冲区缓存的效率开始下降（需要缓存两个块，而如果行没有迁移就只需要 缓存一个块），另外表的大小好复杂性都有所增加。由于这些原因，你可能不希望迁移行。

有 意思的是，如果一行从左边的块迁移到右边的块，如同 10-5 所示，而且它在将来某个时间点还要再迁移，Oracle 会这样做呢？造成这种又一次迁移的原因 可能是：在这一行迁移到的“目标”块上又增加了其他的行，然后这一行再次更新，变得更大。Oracle 实际上会把这一行迁移回原来的块，如果有足够的空 间，仍放回原地（这么一来，这一行可能变得“未

迁移”）。如果没有足够的空间，Oracle 会把这一行迁移到另外的某个块上，并修改原来块上的转发地址。 因此，行迁移总是涉及一层间接性。

所以，现在我们再回到 PCTFREE，来说明这个参数的作用：如果设置得当，这个参数可以帮助你尽量减少行串链。

2. 设置 PCTFREE 和 PCTUSED 值

设 置 PCTFREE 和 PCTUSED 是一个很重要的主题，不过往往被忽视。总的来说，使用 MSSM 时，PCTUSED 和 PCTFREE 都很重要；对于 ASSM，只有 PCTFREE 是重要的。一方面，你要使用这些参数来避免迁移过多的行。另一方面，要使用这些参数避免浪费太多的空间。你需要查看对象，描 述这些对象要如何使用，然后为设置这些值得出一个逻辑计划。设置这些参数时，如果主观地采用一般经验很可能招致失败；必须根据具体的使用设置。可以考虑以 下做法（要记住，这里的“高”和“低”都是相对的；而且使用 ASSM 时仅 PCTFREE 适用）：

- 高 PCTFREE，低 PCTUSED：如果你插入了将要更新的大量数据，而且这些更新会频繁地增加行的大小，此时就适合采用这种设置。这种设置在插入后会在块上预留大量的空间（高 PCTFREE），并使得将块放回到 freelist 之前必须几乎为空（低 PCTUSED）。

- 低 PCTFREE，高 PCTUSED：如果你只想对表完成 INSERT 或 DELETE，或者如果你确实要完成 UPDATE，但 UPDATE 只是缩小行的大小，此时这种设置就很适合。

10.2.6 LOGGING 和 NOLOGGING

通 常对象都采用 LOGGING 方式创建，这说明对象上完成的操作只要能生成 redo 就都会生成 redo。NOLOGGING 则允许该对象完成某些操作时可以 不生成 redo；这个内容在上一章详细介绍过。NOLOGGING 只影响几个特定的操作，如对象的初始创建，或使用 SQL*Loader 的直接路径加载， 或者重建（请参考 Oracle SQL Reference 手册来了解你使用的数据库对象可以应用哪些操作）。

这个选项并不会完全禁用对象的重做日志生成，只是几个特定的操作不生成日志而已。例如，如果把一个表创建为 SELECT NOLOGGING，然后 INSERT INTO THAT_TABLE VALUES(1)，这个 INSERT 就会生成日志，但是表创建可能不生成 redo（DBA 可以在数据库或表空间级强制生成日志）。

10.2.7 INITRANS 和 MAXTRANS

段 中每个块都有一个块首部。这个块首部中有一个事务表。事务表中会建立一些条目来描述哪些事务将块上的哪些行/元素锁定。这个事务表的初始大小由对象的 INITRANS 设置指定。对于表，这个值默认为 2（索引的 INITRANS 也默认为 2）。事务表会根据需要动态扩展，最大达到 MAXTRANS 个条目 （假设块上有足够的自由空间）。所分配的每个事务条目需要占用块首部中的 23～24 字节的存储空间。注意，对于 Oracle 10g，MAXTRANS 则会忽略，所有段的 MAXTRANS 都是 255。

10.3 堆组织表

应 用中 99%（或者更多）的情况下使用的可能都是堆组织表，不过随着 IOT 的出现，这种状况以后可能会有所改观，因为 IOT 本身就可以加索引。执行 CREATE TABLE 语句

时，默认得到的表类型就是堆组织表。如果你想要任何其他类型的表结构，就需要在 CREATE 语句本身中指定它。

堆 （heap）是计算机科学领域中得到深入研究的一种经典数据结构。它实际上就是一个很大的空间、磁盘或内存区（当然，这里所说的磁盘是指数据库表的相应磁 盘），会以一种显然随机的方式管理。数据会放在最合适的地方，而不是以某种特定顺序来放置。许多人希望能按数据放入表中的顺序从表中取出数据，但是对于 堆，这是无法保证的。这一点很容易说清楚。

在以下的例子中，我将建立一个表，使得在我的数据库中每块刚好能放一个整行（我使用的块大小是 8KB）。不一定非得每块上有一行，我只是想利用这一点来展示一种可预测的事务序列。不论数据库使用多大的块大小，也不论表的大小如何，都可以观察到以下行为（行没有次序）：

```
ops$tkyte@ORA10GR1> create table t

  2 ( a int,

  3 b varchar2(4000) default rpad('*',4000,'*'),

  4 c varchar2(3000) default rpad('*',3000,'*')
  5 )
  6 /
Table created.


ops$tkyte@ORA10GR1> insert into t (a) values ( 1);

1 row created.


ops$tkyte@ORA10GR1> insert into t (a) values ( 2);

1 row created.


ops$tkyte@ORA10GR1> insert into t (a) values ( 3);

1 row created.


ops$tkyte@ORA10GR1> delete from t where a = 2 ;

1 row deleted.


ops$tkyte@ORA10GR1> insert into t (a) values ( 4);

1 row created.
```

```
ops$tkyte@ORA10GR1> select a from t;

A
----------
1
4
3
```

如 果你想再试试（得到同样的结果），可以根据你的块大小来调整 B 和 C 列。例如，如果你的块大小为 2KB，则不需要 C 列，而且 B 列应该是一个 VARCHAR2 (1500)，默认 有 1,500 个星号。在这样一个表中，由于数据在堆中管理，只要有空间变为可用，就会重用这个空间。

注意 使用 ASSM 或 MSSM 时，你会发现行最后会在“不同的位置上”。底层的空间管理例程有很大差别，在 ASSM 和 MSSM 中，对同一个表执行同样的操作很可能得到不同的物理顺序。尽管数据逻辑是相同的，但是它们会以不同的方式存储。

全 部扫描时，会按命中的顺序来获取数据，而不是以插入的顺序。这是一个必须了解的重要的数据库表概念：一般来讲，数据库表本质上是无序的数据集合。还应该注 意到，要观察到这种效果，不必在 INSERT 后接下来再使用 DELETE；只需使用 INSERT 就可以得到同样的结果。如果我插入一个小行，那么观察到的 结果很可能是：取出行时默认的顺序为“小行、小行、大行”。这些行并不按插入的顺序获取。Oracle 会把数据放在能放下的任何地方，而不是按照日期或事 务的某种顺序来存放。

如 果你的查询需要按插入的顺序来获取数据，就必须向表中增加一列，以便获取数据时使用这个列对数据排序。例如，这可以是一个数字列，有一个递增的序列（使用 Oracle SEQUENCE 对象）。只需使用一个 SELECT，其 ORDER BY 子句对这个列完成排序，这样就可以模拟插入顺序。这个顺序可能只是近似的，因为序号为 55 的行很可能在序号为 54 的行之前提交，因此，数据库中序号为 55 的行可能放在前面。

应该把堆组织表看作一个很大的无序行集合。这些行会以一种看来随机的顺序取出，而且取出的顺序还取决于所用的其他选项（并行查询、不同的优化器模式，等待），同一个查询可能会以不同的顺序取出数据。不要过分依赖查询得到的顺序，除非查询中有一个 ORDER BY 语句！

除 此之外，关于堆表还有什么重要的内容需要了解？要知道，Oracle SQL Reference 手册中介绍 CREATE TABLE 语法时足足用了 72 页，所以有关的选项当然多。由于存在如此之多的选项，所以很难全部掌握。“线路图”（或“轨迹”图）本身就用了 18 页来介 绍。要了解一个给定表的 CREATE TABLE 语句中主要有哪些可用的选项，我用了一个技巧。首先，尽可能简单地创建表，例如：

```
ops$tkyte@ORA10GR1> create table t

  2 ( x int primary key,

  3 y date,
  4 z clob
  5 )
  6 /
Table created.
```

然后，使用标准内置包 DBMS_METADATA，查询这个表的定义，并查看详细语法：

```
ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE', 'T' ) from dual;


DBMS_METADATA.GET_DDL('TABLE','T')
--------------------------------------------------------------------------------

CREATE TABLE "OPS$TKYTE"."T"

(       "X" NUMBER(*,0),
        "Y" DATE,
        "Z" CLOB,
        PRIMARY KEY ("X")

USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645

PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)

TABLESPACE "USERS" ENABLE

) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS
LOGGING

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645

PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)

TABLESPACE "USERS"

LOB ("Z") STORE AS (

TABLESPACE "USERS" ENABLE STORAGE IN ROW CHUNK 8192 PCTVERSION
10

NOCACHE

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645

PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT))
```

这个技巧的好处是，它显示了 CREATE TABLE 语句的许多选项。我只需要提供数据类型，Oracle 就会为我生成详细的“版本”（CREATE TABLE 版本）。现在我可以定制这个详细的版本，可能把 ENABLE STORAGE IN ROW 改 成 DISABLE STORAGE IN ROW，这样会禁用随结构化数据在行中存储 LOB 数据，而把 LOB 数据存储在另外一个段中。我一直都在使用这个技巧来节省我的时间，因为要从那个庞大的线 路图中找出该使用哪个选项很让人犯愁，如果不采用这个技巧，可能就会为此浪费好几分钟。使用这个技术还可以了解不同的情况下 CREATE TABLE 语句有哪些可用的选项。

既然你知道了如何查看一个给定的 CREATE TABLE 语句可用的大多数选项，那么对于堆表来说，需要注意哪些重要的选项呢？在我看来，对于 ASSM 有两个重要选项，对于 MSSM，重要选项有 4 个：

FREELISTS： 仅适用于 MSSM。每个表都会在一个 freelist 上管理堆中分配的块。一个表可以有多个 freelist。如果你认定会有多个并发用户对表执行大量的 插入，配置多个 freelist 可能会大大地改善性能（可能要以额外的存储空间为代价）。这个设置对性能可能产生的影响请参见“FREELISTS”一节 中的讨论和有关例子。

PCTFREE：ASSM 和 MSSM 都适用。在 INSERT 过程中，会测量块的充满程度。如前所示，根据块当前充满的程度，这个参数用于控制能否将一行增加到一个块上。这个选项还可以控制因后续更新所导致的行迁移，要根据将如何使用表来适当地设置。

PCTUSED：仅适用于 MSSM。度量一个块必须为多空才允许再次插入行。如果块中已用的空间小于 PCTUSED，就可以插入新行了。同样地，类似于 PCTFREE，必须考虑你将如何使用表，从而适当地设置这个选项。

INITRANS：ASSM 和 MSSM 都适合。为块初始分配的事务槽数。如果这个选项设置得太低（默认值为 2，这也是最小值），可能导致多个用户访问的一个块上出现并发问题。如果一 个数据块机会已满，而且事务表无法动态扩展，会话就会排队等待这个块，因为每个并发事务都需要一个事务槽。如果你认为会对同样的块完成多个并发更新，就应 该考虑增大这个值。

注意 单独存储在 LOB 段中的 LOB 数据并不使用表的 PCTFREE/PCTUSED 参数设置。这些 LOB 块以不同的方式管理：它们总是会填入，直至达到最大容量，而且仅当完全为空时才返回 freelist。

这些参数要特别注意。随着本地管理表空间的引入（这也是强烈推荐的做法），我发现其余的参数（如 PCTINCREASE、NEXT 等）已经没有什么意义了。

10.4 索引组织表

索引组织表（index organized table，IOT） 就是存储在一个索引结构中的表。存储在堆中的表是无组织的（也就是说，只要有可用的空间，数据可以放在任何地方），IOT 中的数据则按主键存储和排序。对 你的应用来说，IOT 表现得与一个“常规”表并无二致；还是要使用 SQL 正常地访问这些表。IOT 对信息获取、空间应用和 OLAP 应用特别有用。

IOT 有 什么意义？实际上，可以反过来问：堆组织表有什么意义？由于一般认为关系数据库中的所有表都有一个主键，堆组织表难道不是在浪费空间吗？使用堆组织表时， 我们必须为表和表主键上的索引分别留出空间。而 IOT 则不存在主键的空间开销，因为索引就是数据，数据就是索引，两者已经合二为一。事实上，索引是一个复 杂的数据结构，需要大量的工作来管理和维护，而且随着存储的行宽度有所增加，维护的需求也会增加。另一方面，相比之下，堆管理起来则很容易。对组织表在某 些方面的效率要比 IOT 高。一般认

为，比起堆组织表来说，IOT 有一些突出的优点。例如，记得曾经有一次，我在一些文本数据上建立一个反向表索引（那时还 没有引入 interMedia 和相关的技术）。我有一个表，其中放满了文档，并发现其中的单词。我的表如下所示：

```
create table keywords
(       word varchar2(50),
        position int,
        doc_id int,
        primary key(word,position,doc_id)
);
```

在 此，我的表完全由主键组成。因此有超过 100%的（主键索引）开销；表的大小与主键索引的大小相当（实际上，主键索引更大，因为它物理地存储了所指向的行 的 rowid；而表中并不存储 rowid，表中的行 ID 是推断出来的）。使用这个表时，WHERE 子句只选择了 WORD 列或 WORD 和 POSITION 列。也就是说，我并没有使用表，而只是使用了表上的索引，表本身完全是开销。我想找出包含某个给定单词的所有文档（或者满足“接近”每个词等匹配条件）。 此时，堆表是没有用的，它只会在维护 KEYWORDS 表时让应用变慢，并使存储空间的需求加倍。这个应用就非常适合采用 IOT。

另一个适于使用 IOT 的实现是代码查找表。例如，可能要从 ZIP_CODE 查找 STATE。此时可以不要堆表，而只使用 IOT 本身。如果你只会通过主键来访问一个表，这个表就非常适合实现为 IOT。

如 果你想保证数据存储在某个位置上，或者希望数据以某种特定的顺序物理存储，IOT 就是一种合适的结构。如果是 Sybase 和 SQL Server 的用户，你可能会使用一个聚簇索引，但是 IOT 比聚簇索引更好。这些数据库中的聚簇索引可能有多达 110%的开销（与前面的 KEYWORDS 表例子类似）。而使用 IOT 的话，我们的开销则是 0%，因为数据只存储一次。有些情况下，你可能希望数据像这样物理地共同存储在一处，父/子关系就是这样 一个典型的例子。假设 EMP 表有一个包含地址的子表。员工最初递交求职信时，你可能向系统中（地址比表中）输出一个家庭地址。过一段时间后，他搬家了，就 要把家庭地址修改为原地址，并增加一个新的家庭地址。然后，他可能还会回去读学位，此时可能还要增加一个学校地址，等等。也就是说，这个员工有 3～4 个 （或者更多）的（地址）详细记录，但是这些详细记录是随机到来的。在一个普通的基于堆的表中，这些记录可以放在任何地方。两个或更多地址记录放在堆表的同 一个数据库块上的概率接近于 0.不过，你查询员工的信息时，总会把所有地址详细记录都取出来。在一段时间内分别到达的这些行总会被一并获取得到。为了让这 种获取更为高效，可以对子表使用 IOT，使得子表将对应某个给定员工的所有记录都插入到相互“靠近”的地方，这样在反复获取这些记录时，就可以减少工作 量。

使用一个 IOT 将子表信息物理地存储在同一个位置上有什么作用？这一点通过一个例子就能很容易地说明。下面创建并填充一个 EMP 表：

```
ops$tkyte@ORA10GR1> create table emp

  2 as

  3         select object_id empno,

  4         object_name ename,

  5         created hiredate,
```

```
    6          owner job

    7          from all_objects

    8 /
Table created.


ops$tkyte@ORA10GR1> alter table emp add constraint emp_pk primary key(empno)

    2 /
Table altered.

ops$tkyte@ORA10GR1> begin

    2 dbms_stats.gather_table_stats( user, 'EMP', cascade=>true );

    3 end;
    4 /

PL/SQL procedure successfully completed.
```

接下来，将这个子表实现两次：一次作为传统的堆表，另一次实现为 IOT：

```
ops$tkyte@ORA10GR1> create table heap_addresses

    2 (        empno references emp(empno) on delete cascade,

    3          addr_type varchar2(10),

    4          street varchar2(20),

    5          city varchar2(20),

    6          state varchar2(2),

    7          zip number,

    8          primary key (empno,addr_type)

    9 )
    10 /
Table created.


ops$tkyte@ORA10GR1> create table iot_addresses
```

```
  2 (        empno references emp(empno) on delete cascade,
  3          addr_type varchar2(10),
  4          street varchar2(20),
  5          city varchar2(20),
  6          state varchar2(2),
  7          zip number,
  8          primary key (empno,addr_type)
  9 )
 10 ORGANIZATION INDEX
 11 /
Table created.
```

我 如下填充这些表，首先为每个员工插入一个工作地址，其次插入一个家庭地址，再次是原地址，最后是一个学校地址。堆表很可能把数据放在表的“最后”；数据到 来时，堆表只是把它增加到最后，因为此时只有数据到来，而没有数据被删除。过一段时间后，如果有地址被删除，插入就开始变得更为随机，会随机地插入到整个 表中的每个位置上。不过，有一点是肯定的，堆表中员工的工作地址与家庭地址同在一个块上的机率几乎为 0.不过，对于 IOT，由于键在 EMPNO， ADDR_TYPE 上，完全可以相信：对应一个给定 EMPNO 的所有地址都会放在同一个（或者两个）索引块上。填充这些数据的插入语句如下：

```
ops$tkyte@ORA10GR1> insert into heap_addresses
  2 select empno, 'WORK', '123 main street', 'Washington', 'DC', 20123
  3 from emp;
48250 rows created.

ops$tkyte@ORA10GR1> insert into iot_addresses
  2 select empno, 'WORK', '123 main street', 'Washington', 'DC', 20123
  3 from emp;
48250 rows created.
```

我把这个插入又做了 3 次，依次将 WORK 分别改为 HOME、PREV 和 SCHOOL。然后收集统计信息：

```
ops$tkyte@ORA10GR1> exec dbms_stats.gather_table_stats( user,
'HEAP_ADDRESSES' );
```

```
PL/SQL procedure successfully completed.



ops$tkyte@ORA10GR1> exec dbms_stats.gather_table_stats( user, 'IOT_ADDRESSES' );

PL/SQL procedure successfully completed.
```

现在可以看看我们预料到的显著差别。通过使用 AUTOTRACE，可以了解到改变有多大：

```
ops$tkyte@ORA10GR1> set autotrace traceonly

ops$tkyte@ORA10GR1> select *

  2 from emp, heap_addresses
  3 where emp.empno = heap_addresses.empno
  4 and emp.empno = 42;

Execution Plan
----------------------------------------------------------

    0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=8 Card=4
Bytes=336)

    1    0      NESTED LOOPS (Cost=8 Card=4 Bytes=336)

    2    1        TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)
(Cost=2 Card=1...

    3    2          INDEX (UNIQUE SCAN) OF 'EMP_PK' (INDEX (UNIQUE))
(Cost=1 Card=1)

    4    1        TABLE ACCESS (BY INDEX ROWID) OF 'HEAP_ADDRESSES'
(TABLE) (Cost=6...

    5    4          INDEX (RANGE SCAN) OF 'SYS_C008078' (INDEX
(UNIQUE)) (Cost=2 Card=4)


Statistics
----------------------------------------------------------
...
       11  consistent gets
...
        4  rows processed
```

这是一个相对常见的计划：按主键访问 EMP 表；得到行；然后使用这个 EMPNO 访问地址表；接下来使用索引找出子记录。获取这个数据执行了 11 次 I/O。下面再运行同样的查询，不过这一次地址表实现为 IOT：

```
ops$tkyte@ORA10GR1> select *

  2 from emp, iot_addresses

  3 where emp.empno = iot_addresses.empno

  4           and emp.empno = 42;


Execution Plan
----------------------------------------------------------

    0         SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=4
Bytes=336)

    1     0         NESTED LOOPS (Cost=4 Card=4 Bytes=336)

    2     1               TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)
(Cost=2 Card=1...

    3     2                     INDEX (UNIQUE SCAN) OF 'EMP_PK' (INDEX (UNIQUE))
(Cost=1 Card=1)

    4     1               INDEX (RANGE SCAN) OF 'SYS_IOT_TOP_59615' (INDEX
(UNIQUE)) (Cost=2...


Statistics
----------------------------------------------------------
...

       7       consistent gets

...
       4       rows processed


ops$tkyte@ORA10GR1> set autotrace off
```

这里少做了 4 次 I/O（这个 4 应该能推测出来）；我们跳过了 4 个 TABLE ACCESS (BY INDEX ROWID)步骤。子表记录越多，所能跳过的 I/O 就越多。

那 么，这 4 个 I/O 是什么呢？在这个例子中，这是查询所完成 I/O 的 1/3 还多，如果反复执行这个查询，这就会累积起来。每个 I/O 和每个一致获取需要访问 缓冲区缓存，尽管从缓存区缓存读数据要比从磁盘读快得多，但是要知道，缓存区缓存获取并不是“免费”

的，而且也绝对不是“廉价”的。每个缓冲区缓存获取都 需要缓冲区缓存的多个闩，而闩是串行化设备，会限制我们的扩展能力。通过运行以下 PL/SQL 块，可以测量出 I/O 和闩定的减少：

```
ops$tkyte@ORA10GR1> begin
  2 for x in ( select empno from emp )
  3 loop
  4         for y in ( select emp.ename, a.street, a.city, a.state, a.zip
  5                    from emp, heap_addresses a
  6                    where emp.empno = a.empno
  7                          and emp.empno = x.empno )
  8         loop
  9                  null;
 10         end loop;
 11 end loop;
 12 end;
 13 /

PL/SQL procedure successfully completed.
```

这里只是模拟我们很忙，在此将查询运行大约 45,000 次，对应各个 EMPNO 运行一次。如果对 HEAP_ADRESSES 和 IOT_ADDRESSES 表分别运行这个代码，TKPROF 会显示如下结果：

```
SELECT EMP.ENAME, A.STREET, A.CITY, A.STATE, A.ZIP
FROM EMP, HEAP_ADDRESSES A
WHERE EMP.EMPNO = A.EMPNO AND EMP.EMPNO = :B1

call     count       cpu    elapsed       disk      query    current
        rows
-------  ------  --------  ----------  ----------  ----------  -
---------  ----------
Parse        1      0.00       0.00          0          0          0
```

0

Execute 48244 7.66 7.42 0 0 0
0

Fetch 48244 6.29 6.56 0 483393 0
192976

------- ------ -------- ---------- ---------- ---------- -
--------- ----------

total 96489 13.95 13.98 0 483393 0
192976

Rows Row Source Operation

---------- ----------

192976 NESTED LOOPS (cr=483393 pr=0 pw=0 time=5730335 us)

48244 TABLE ACCESS BY INDEX ROWID EMP (cr=144732 pr=0 pw=0
time=1594981 us)

48244 INDEX UNIQUE SCAN EMP_PK (cr=96488 pr=0 pw=0 time=926147 us)...

192976 TABLE ACCESS BY INDEX ROWID HEAP_ADDRESSES (cr=338661 pr=0
pw=0 time=...

192976 INDEX RANGE SCAN SYS_C008073 (cr=145685 pr=0 pw=0 time=1105135
us)...

****
******

SELECT EMP.ENAME, A.STREET, A.CITY, A.STATE, A.ZIP

FROM EMP, IOT_ADDRESSES A

WHERE EMP.EMPNO = A.EMPNO AND EMP.EMPNO = :B1

call count cpu elapsed disk query current

```
        rows

-------  ------  --------  ----------  ----------  ----------  -
---------  ----------

Parse        1      0.00        0.00           0           0           0
         0

Execute  48244      8.17        8.81           0           0           0
         0

Fetch    48244      4.31        4.12           0      292918           0
    192976

-------  ------  --------  ----------  ----------  ----------  -
---------  ----------

total    96489     12.48       12.93           0      292918           0
    192976



      Rows   Row Source Operation

    -----------  ---------------------------------------------------------------------------------------------
-----------------

 192976   NESTED LOOPS (cr=292918 pr=0 pw=0 time=3429753 us)

     48244   TABLE ACCESS BY INDEX ROWID EMP (cr=144732 pr=0 pw=0
time=1615024 us)

     48244   INDEX UNIQUE SCAN EMP_PK (cr=96488 pr=0 pw=0 time=930931 us)...

 192976   INDEX RANGE SCAN SYS_IOT_TOP_59607 (cr=148186 pr=0 pw=0
time=1417238 us)...
```

两 个查询获取的行数同样多，但是 HEAP 表完成的逻辑 I/O 显著增加。随着系统并发度的增加，可以想见，堆表使用的 CPU 时间也会增长得更快，而查询耗费 CPU 时间的原因可能只是在等待缓冲区缓存的闩。使用 runstats（我自己设计的一个工具），可以测量出两种实现的闩定之差。在我的系统上，观察到的 结果是：

```
STAT...consistent gets                  484,065 293,566   -190,499

STAT...no work - consistent re          194,546 4,047        -190,499
STAT...consistent gets from ca          484,065 293,566   -190,499
STAT...session logical reads            484,787 294,275   -190,512
STAT...table fetch by rowid              241,260 48,260       -193,000
```

```
STAT...buffer is not pinned co          337,770 96,520      -241,250
LATCH.cache buffers chains              732,960 349,380   -383,580


Run1 latches total versus runs -- difference and pct

Run1          Run2          Diff              Pct

990,344      598,750       -391,594       165.40%
```

在此 Run1 是 HEAP_ADDRESSES 表，Run2 是 IOT_ADDRESSES 表。可以看到，在发生的闩定方面，存在显著的下降，而且这种下降可以重复验证，这主要是因为缓冲区缓存存在闩的串链（即保护缓冲区缓存的闩）。在这种情况下，IOT 提供了以下好处：

- 提供缓冲区缓存效率，因为给定查询的缓存中需要的块更少。
- 减少缓冲区缓存访问，这会改善可扩缩性。
- 获取数据的工作总量更少，因为获取数据更快。
- 每个查询完成的物理 I/O 更少，因为对于任何给定的查询，需要的块更少，而且对地址记录的一个物理 I/O 很可能可以获取所有地址（而不只是其中一个地址，但堆表实现就只是获取一个地址）。

如 果经常在一个主键或惟一键上使用 BETWEEN 查询，也是如此。如果数据有序地物理存储，就能提升这些查询的性能。例如，我在数据库中维护了一个股价表。 每天我要收集数百支股票的股价记录、日期、收盘价、当日最高价、当日最低价、买入卖出量和其他相关信息。这个表如下所示：

```
ops$tkyte@ORA10GR1> create table stocks
  2 (       ticker varchar2(10),
  3         day date,
  4         value number,
  5         change number,
  6         high number,
  7         low number,
  8         vol number,
  9         primary key(ticker,day)
 10 )
 11 organization index
 12 /
```

```
Table created.
```

我 经常一次查看一支股票几天内的表现（例如，计算移动平均数）。如果我使用一个堆组织表，那么对于股票记录 ORCL 的两行在同一个数据库块上的可能性几乎为 0.这是因为，每天晚上我都会插入当天所有股票的记录。这至少会填满一个数据库块（实际上，可能会填满多个数据库块）。因此，每天我都会增加一个新的 ORCL 记录，但是它总在另一个块上，与表中已有的其他 ORCL 记录不在同一个块上。如果执行如下查询：

```
Select * from stocks
where ticker = 'ORCL'
      and day between sysdate-100 and sysdate;
```

Oracle 会 读取索引，然后按 rowid 来访问表，得到余下的行数据。由于我加载表所采用的方式，获取的每 100 行会在一个不同的数据库块上，所有每获取 100 行可能 就是一个物理 I/O。下面考虑 IOT 中有同样的数据。这是这个查询，不过现在只需要读取相关的索引块，这个索引块中已经有所有的数据。在此不仅不存在表访 问，而且一段时期内对于 ORCL 的所有行物理存储在相互“邻近”的位置。因此引入的逻辑 I/O 和物理 I/O 都更少。

现在你已经知道了什么时候想使用 IOT，以及如何使用 IOT。接下来需要了解这些表有哪些选项。有哪些需要告诫的方面？IOT 的选项与堆组织表的选项非常相似。我们还是使用 DBMS_METADATA 来显示详细选项。先从 IOT 的 3 个基本变体开始：

```
ops$tkyte@ORA10GR1> create table t1

  2 (       x int primary key,

  3         y varchar2(25),

  4         z date

  5 )
  6 organization index;
Table created.


ops$tkyte@ORA10GR1> create table t2

  2 (       x int primary key,

  3         y varchar2(25),

  4         z date

  5 )
  6 organization index
  7 OVERFLOW;
Table created.
```

```
ops$tkyte@ORA10GR1> create table t3
  2 (        x int primary key,
  3          y varchar2(25),
  4          z date
  5 )
  6 organization index
  7 overflow INCLUDING y;
ole created.
```

后面会介绍 OVERFLOW 和 INCLUDING 会为我们做什么，不过首先来看第一个所需的详细 SQL：

```
ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE', 'T1' ) from dual;

BMS_METADATA.GET_DDL('TABLE','T1')
--------------------------------------------------------------------------------

  CREATE TABLE "OPS$TKYTE"."T1"
   "X" NUMBER(*,0),
   "Y" VARCHAR2(25),
   "Z" DATE,
   PRIMARY KEY ("X") ENABLE
RGANIZATION INDEX
OCOMPRESS
  PCTFREE 10 INITRANS 2 MAXTRANS 255 LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
       2147483645
                PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
       DEFAULT)
LBLESPACE "USERS"
TTHRESHOLD 50
```

这个表引入了两个新的选项：NOCOMPRESS 和 PCTTHRESHOLD，稍后将介绍它们。你可能已经注意到了，与前面的 CREATE TABLE 语 法相比，这里好像少了点什么：没有 PCTUSED 子句，但是这里有一个 PCTFREE。这是因为，索引是一个复杂的数据结构，它不像堆那样随机组织，所以 数据必须按部就班地存放到它该去的地方去。在堆中，块只是

有时能插入新行，而索引则不同，块总是可以插入新的索引条目。如果每个数据（根据它的值）属于一 个给定块，在总会放在那个块上，而不论这个块多满或者多空。另外，只是在索引结构中创建对象和填充数据时才会使用 PCTFREE。其用法与堆组织表中的用 法不同。PCTFREE 会在新创建的索引上预留空间，但是对于以后对索引的操作不预留空间，这与不使用 PCTUSED 的原因是一样的。堆组织表上关于 freelist 的考虑同样完全适用于 IOT。

现 在来讨论新发现的选项 NOCOMPRESS。这个选项对索引一般都可用。它告诉 Oracle 把每个值分别存储在各个索引条目中（也就是不压缩）。如果对象 的主键在 A、B 和 C 列上，A、B 和 C 的每一次出现都会物理地存储。NOCOMPRESS 反过来就是 COMPRESS N，在此 N 是一个整数，表示要压缩的列数。这样可以避免重复值，并在块级提取“公因子”（factor out）。这样在 A 的值（以及 B 的值）重复出现时，将不再物理地存储它们。例如，请考虑如下创建的一个表：

```
ops$tkyte@ORA10GR1> create table iot

  2 (         owner, object_type, object_name,

  3 primary key(owner,object_type,object_name)

4 )
5 organization index
6 NOCOMPRESS
7 as

  8 select owner, object_type, object_name from all_objects

9 /
ble created.
```

可 以想想看，每个模式（作为 OWNER）都拥有大量对象，所有 OWNER 值可能会重复数百次。甚至 OWNER，OBJECT_TYPE 值对也会重复多次，因 为给定模式可能有数十个表、数十个包等。只是这 3 列合在一起不会重复。可以让 Oracle 压缩这些重复的值。索引块不是包含表 10-1 所示的值，而是可以 使用 COMPRESS 2（提取前两列），包含表 10-2 所示的值。

表 10-1 索引叶子块，NOCOMPRESS

Sys,table,t1	Sys,table,t2	Sys,table,t3	Sys,table,t4
Sys,table,t5	Sys,table,t6	Sys,table,t7	Sys,table,t8
. . .	. . .	. . .	. . .
Sys,table,t100	Sys,table,t101	Sys,table,t102	Sys,table,t103

表 10-2 索引叶子块，COMPRESS 2

Sys,table	t1	t2	t3

t4 t5 . . . . . .

. . . t103 t104 . . .

t300 t301 t302 t303

也 就是说，值 SYS 和 TABLE 只出现一次，然后存储第三列。采用这种方式，每个索引块可以有更多的条目（否则这是不可能的）。这不会降低并发性，因为我们 仍在行级操作；另外也不会影响功能。它可能会稍微多占用一些 CPU 时间，因为 Oracle 必须做更多的工作将键合并在一起。另一方面，这可能会显著地减少 I/O，并允许更多的数据在缓冲区缓存中缓存，原因是每个块上能有更多的数据。这笔交易很划得来。

下面做一个快速的测试，对前面 CREATE TABLE 的 SELECT 分别采用 NOCOMPRESS、COMPRESS 1 和 COMPRESS 2 选项，来展示能节省多少空间。先来创建 IOT，但不进行压缩：

```
ops$tkyte@ORA10GR1> create table iot
  2 (       owner, object_type, object_name,
  3         constraint iot_pk primary key(owner,object_type,object_name)
  4 )
  5 organization index
  6 NOCOMPRESS
  7 as
  8         select distinct owner, object_type, object_name
  9         from all_objects
 10 /
ble created.
```

现在可以测量所用的空间。为此我们将使用 ANALYZE INDEX VALIDATE STRUCTURE 命令。这个命令会填写一个名为 INDEX_STATS 的动态性能视图，其中最多只包含一行，即这个 ANALYZE 命令最后一次执行的信息：

```
ops$tkyte@ORA10GR1> analyze index iot_pk validate structure;
lex analyzed.

ops$tkyte@ORA10GR1> select lf_blks, br_blks, used_space,
  2 opt_cmpr_count, opt_cmpr_pctsave
  3 from index_stats;
```

```
 LF_BLKS   BR_BLKS     USED_SPACE     OPT_CMPR_COUNT     OPT_CMPR_PCT
SAVE

     ----------       ----------     --------------------      ----------------------------      ----------------
----------------

       284                3             2037248                              2
       33
```

由 此显示出，我们的索引目前使用了 284 个叶子块（即数据所在的块），并使用了 3 个分支块（Oracle 在索引结构中导航所用的块）来找到这些叶子块。使用 的空间大约是 2MB（2,038,248 字节）。另外两列名字有些奇怪，这两列是要告诉我们一些信息。OPT_CMPR_COUNT（最优压缩数）列要说 的是："如果你把这个索引置为 COMPRESS 2，就会得到最佳的压缩"。OPT_CMPR_PCTSAVE（最优的节省压缩百分比）则是说，如果执行 COMPRESS 2，就能节省大约 1/3 的存储空间，索引只会使用现在 2/3 的磁盘空间。

注意 下一章将更详细地介绍索引结构。

为了测试上述理论，我们先用 COMPRESS 1 重建这个 IOT：

```
ops$tkyte@ORA10GR1> alter table iot move compress 1;

ble altered.


ops$tkyte@ORA10GR1> analyze index iot_pk validate structure;

lex analyzed.


ops$tkyte@ORA10GR1> select lf_blks, br_blks, used_space,

   2                 opt_cmpr_count, opt_cmpr_pctsave
   3          from index_stats;


 LF_BLKS   BR_BLKS     USED_SPACE     OPT_CMPR_COUNT     OPT_CMPR_PCT
SAVE

     ----------       ----------              ----------                 --------------
  ----------------

          247                1             1772767                              2
          23
```

可以看到，索引确实更小了：大约 1.7MB，叶子块和分支块都更少。但是，现在它说“你还能再节省另外 23%的空间”，因为我们没有充分地压缩。下面用 COMPRESS 2 再来重建 IOT：

```
ops$tkyte@ORA10GR1> alter table iot move compress 2;

ble altered.


ops$tkyte@ORA10GR1> analyze index iot_pk validate structure;

dex analyzed.


ops$tkyte@ORA10GR1> select lf_blks, br_blks, used_space,
  2 opt_cmpr_count, opt_cmpr_pctsave
  3 from index_stats;


 LF_BLKS   BR_BLKS    USED_SPACE    OPT_CMPR_COUNT    OPT_CMPR_PCT
SAVE

     ----------        ----------              ----------                  --------------
 ---------------

         190               1            1359357                           2
           0
```

现在大小有了显著减少，不论是叶子块数还是总的使用空间都大幅下降，现在使用的空间大约是 1.3MB。再来看原来的数字：

```
ops$tkyte@ORA10GR1> select (2/3) * 2037497 from dual;

3)*2037497
----------
58331.33
```

可以看到 OPT_CMPR_PCTSAVE 真是精准无比。上一个例子指出，关于 IOT 有一点很有意思：IOT 是表，但是只是有其名而无其实。IOT 段实际上是一个索引段。

现 在我先不讨论 PCTTHRESHOLD 选项，因为它与 IOT 的下面两个选项有关：OVERFLOW 和 INCLUDING。如果查看以下两组表（T2 和 T3）的完整 SQL，可以看到如下内容（这里我使用了一个 DBMS_METADATA 例程来避免 STORAGE 子句，因为它们对这个例子没有意义）：

```
ops$tkyte@ORA10GR1> begin
  2 dbms_metadata.set_transform_param
```

```
     3 ( DBMS_METADATA.SESSION_TRANSFORM, 'STORAGE', false );

     4 end;


ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE', 'T2' ) from dual;


BMS_METADATA.GET_DDL('TABLE','T2')
--------------------------------------------------------------------------------

  CREATE TABLE "OPS$TKYTE"."T2"

    "X" NUMBER(*,0),
    "Y" VARCHAR2(25),
    "Z" DATE,
    PRIMARY KEY ("X") ENABLE

  ) ORGANIZATION INDEX NOCOMPRESS PCTFREE 10 INITRANS 2 MAXTRANS
      255 LOGGING

    TABLESPACE "USERS"
CTTHRESHOLD 50 OVERFLOW

  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING

    TABLESPACE "USERS"


ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE', 'T3' ) from dual;


BMS_METADATA.GET_DDL('TABLE','T3')
--------------------------------------------------------------------------------

  CREATE TABLE "OPS$TKYTE"."T3"

    "X" NUMBER(*,0),
    "Y" VARCHAR2(25),
    "Z" DATE,
    PRIMARY KEY ("X") ENABLE

  ) ORGANIZATION INDEX NOCOMPRESS PCTFREE 10 INITRANS 2 MAXTRANS
      255 LOGGING
```

```
  TABLESPACE "USERS"

PCTTHRESHOLD 50 INCLUDING "Y" OVERFLOW

PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING

  TABLESPACE "USERS"
```

所 以，现在只剩下 PCTTHRESHOLD、OVERFLOW 和 INCLUDING 还没有讨论。这三个选项有点“绕”，其目标是让索引叶子块（包含具体索引 数据的块）能够高效地存储数据。索引一般在一个列子集上。通常索引块上的行数比堆表块上的行数会多出几倍。索引指望这每块能得到多行。否则，Oracle 会花费大量的时间来维护索引，因为每个 INSERT 或 UPDATE 都可能导致索引块分解，以容纳新数据。

OVERFLOW 子句允许你建立另一个段（这就使得 IOT 成为一个多段对象，就像有一个 CLOB 列一样），如果 IOT 的行数据变得太大，就可以溢出到这个段中。

注意 构成主键的列不能溢出，它们必须直接放在叶子块上。

注意，使用 MSSM 时，OVERFLOW 再次为 IOT 引入了 PCTUSED 子句。对于 OVERFLOW 段和堆表来说，PCTFREE 和 PCTUSED 的含义都相同。使用溢出段的条件可以采用两种方式来指定：

- PCTTHRESHOLD：行中的数据量超过块的这个百分比时，行中余下的列将存储在溢出段中。所以，如果 PCTTHRESHOLD 是 10%，而块大小是 8KB，长度大于 800 字节的行就会把其中一部分存储在别处，而不能在索引块上存储。
- INCLUDING：行中从第一列直到 INCLUDING 子句所指定列（也包括这一列）的所有列都存储在索引块上，余下的列存储在溢出段中。

假设有以下表，块大小为 2KB：

```
ops$tkyte@ORA10GR1> create table iot

  2 ( x int,
  3 y date,

  4 z varchar2(2000),

  5 constraint iot_pk primary key (x)

  6 )
  7 organization index
  8 pctthreshold 10
  9 overflow
  10 /
ole created.
```

用图来说明，则如图 10-6 所示。

图 10-6　有溢出段的 IOT，使用 PCTTHRESHOLD 子句

灰 框是索引条目，这是一个更大索引结构的一部分（在第 11 章中，你将看到一个更大的图，其中会展示索引是什么样子）。简单地说，索引结构是一棵树，叶子块 （存储数据的块）实际上构成一个双向链表，这样一来，一旦我们发现想从索引中的哪个位置开始，就能更容易地按顺序遍历这些节点。白框表示一个 OVERFLOW 段。超出 PCTTHRESHOLD 设置的数据就会存储在这里。Oracle 会从最后一列开始向前查找，直到主键的最后一列（但不包括主键 的最后一列），得出哪些列需要存储在溢出段中。在这个例子中，数字列 X 和日期列 Y 在索引块中总能放下。最后一列 Z 的长度不定。如果它小于大约 190 字节 （2KB 块的 10%是大约 200 字节；再减去 7 字节的日期和 3～5 字节的数字），就会存储在索引块上。如果超过了 190 字节，Oracle 将把 Z 的数据存 储在溢出段中，并建立一个指向它的指针（实际上是一个 rowid）。

另一种做法是使用 INCLUDING 子句。在此要明确地说明希望把哪些列存储在索引块上，而哪些列要存储在溢出段中。给出以下的 CREATE TABLE 语句：

```
ops$tkyte@ORA10GR1> create table iot
  2 (         x int,
  3           y date,
  4           z varchar2(2000),
  5           constraint iot_pk primary key (x)
  6 )
  7 organization index
  8 including y
  9 overflow
  10 /
ble created.
```

我们可能看到图 10-7 所示的情况。

图 10-7 有 OVERFLOW 段的 IOT，使用 INCLUDING 子句

在这种情况下，不论 Z 中存储的数据大小如何，Z 都会“另行”存储在溢出段中。

那 么究竟使用 PCTTHRESHOLD、INCLUDING 还是二者的某种组合呢？这些方法中哪一个更好？这取决于你的实际需求。如果你的应用总是（或者几 乎总是）使用表的前 4 列，而很少访问后 5 列，使用 INCLUDING 会更合适。可以包含至第 4 列，而让另外 5 列另行存储。运行时，如果需要这 5 列，可以采 用行迁移或串链的方式获取这些列。Oracle 将读取行的“首部”，找到行余下部分的指针，然后读取这些部分。另一方面，如果无法清楚地指出哪些列总会被 访问而哪些列一般不会被访问，就可以考虑使用 PCTTHRESHOLD。一旦确定了平均每个索引块上可能存储多少行，设置 PCTTHRESHOLD 就会很 容易。假设你希望每个索引块上存储 20 行。那好，这说明每行应该是 1/20（5%）。你的 PCTTHRESHOLD 就是 5，而且索引叶子块上的每个行块都 不能占用对于块中 5%的空间。

对于 IOT 最后要考虑的是建立索引。IOT 本身可以有一个索引，就像在索引之上再加索引，这称为二次索引（secondary index）。 正常情况下，索引包含了所指向的行的物理地址，即 rowid。而 IOT 二次索引无法做到这一点；它必须使用另外某种方法来指示行的地址。这是因为 IOT 中 的行可以大量移动，而且它不像堆组织表中的行那样“迁移”。IOT 中的行肯定在索引结构中的每个位置上，这取决于它的主键值；只有当索引本身的大小和形状 发生改变时行才会移动（下一章将更详细地讨论索引结构如何维护）。为了适应这种情况，Oracle 引入了一个逻辑 rowid（logical rowid）。 这些逻辑 rowid 根据 IOT 主键建立。对于行的当前位置还可以包含一个“猜测”，不过这个猜测几乎是错的，因为稍过一段时间后，IOT 中的数据可能就会 移动。这个猜测是行第一次置于二次索引结构中时在 IOT 中的物理地址。如果 IOT 中的行必须移动到另外一个块上，二次索引中的猜测就会变得“过时”。因 此，与常规表相比，IOT 上的索引效率稍低。在一个常规表上，索引访问通常需要完成一个 I/O 来扫描索引结构，然后需要一个读来读取表数据。对于 IOT， 通常要完成两个扫描；一次扫描二次结构，另一次扫描 IOT 本身。除此之外，IOT 上的索引可以使用非主键列提供 IOT 数据的快速、高效访问。

索引组织表小结

在 建立 IOT 时，最关键的是适当地分配数据，即哪些数据存储在索引块上，哪些数据存储在溢出段上。对溢出条件不同的各种场景进行基准测试，查看对 INSERT、UPDATE、DELETE 和 SELECT 分别有怎样的影响。如果结构只建立一次，而且要频繁读取，就应该尽可能地把数据放在索引块上（最 合适获取），要么频繁地组织索引中的数据（不适于修改）。堆表的 freelist 相关考虑对 IOT 也同样适用。PCTFREE 和 PCTUSED 在 IOT 中 是两

个重要的角色。不过，PCTFREE 对于 IOT 不像对于堆表那么重要，另外 PCTUSED 一般不起作用。不过，考虑 OVERFLOW 段时， PCTFREE 和 PCTUSED 对于 IOT 的意义将与对于堆表一样重大；要采用与堆表相同的逻辑为溢出段设置这两个参数。

10.5 索引聚簇表

我常常发现，人们对 Oracle 中聚簇的理解是不正确的。许多人都把聚簇与 SQL Server 或 Sybase 中的“聚簇索引”相混淆。但它们并不一样。聚簇（cluster）是指：如果一组表有一些共同的列，则将这样一组表存储在相同 的数据库块中；聚簇还表示把相关的数据存储在同一个块上。SQL Server 中的聚簇索引（clustered index）则要求行按索引键有序的方式存储，这类似于前面所述的 IOT。利用聚簇，一个块可能包含多个表的数据。从概念上讲，这是将数据“预联结”地存 储。聚簇还可以用于单个表，可以按某个列将数据分组存储。例如，部门 10 的所有员工都存储在同一个块上（或者如果一个块放不下，则存储在尽可能少的几个块 上）。聚簇并不是有序地存储数据（这是 IOT 的工作），它是按每个键以聚簇方式存储数据，但数据存储在堆中。所以，部门 100 可能挨在部门 1 旁边，而与部 门 101 和 99 离得很远（这是指磁盘上的物理位置）。

如 同 10-8 所示，图的左边使用了传统的表，EMP 会存储在它的段中。DEPT 也存储在自己的段中。它们可能位于不同的文件和不同的表空间，而且绝对在单独 的区段中。从图的右边可以看到将这两个表聚簇起来会是什么情况。方框表示数据库块。现在将值 10 抽取出来，只存储一次。这样聚簇的所有表中对应部门 10 的 所有数据都存储在这个块上。如果部门 10 的所有数据在一个块上放不下，可以为原来的块串链另外的块，来包含这些溢出的部分，这与 IOT 的溢出块所用的方式 类似。

图 10-8 索引聚簇数据

因 此，下面来看如何创建一个聚簇对象。在对象中创建表的一个聚簇很直接。对象的存储定义（PCTFREE、PCTUSED、INITIAL 等）与 CLUSTER 相关，而不是与表相关。这是有道理的，因为聚簇中会有多个表，而且它们在同一个块上。有多个不同的 PCTFREE 没有意义。因此， CREATE CLUSTER 非常类似于只有很少几个列的 CREATE TABLE（只有聚簇键列）：

```
ops$tkyte@ORA10GR1> create cluster emp_dept_cluster

  2 (         deptno number(2) )
```

```
  3          size 1024

  4 /
ıster created.
```

在此，我们创建了一个索引聚簇（index cluster， 还有一种类型是散列聚簇（hash cluster），将在下一节介绍）。这个聚簇的聚簇列是 DEPTNO 列。表中的列不必非得叫 DEPTNO，但是必须是 NUMBER(2)，这样才能与定 义匹配。我们在这个聚簇定义中加一个 SIZE 1024 选项。这个选项原来告诉 Oracle：我们希望与每个聚簇键值关联大约 1024 字节的数据，Oracle 会在用这个数据库块上设置来计算每个块最 多能放下多少个聚簇键。假设块大小为 8KB，Oracle 会在每个数据库块上放上最多 7 个聚簇键（但是如果数据比预想的更大，聚簇键可能还会少一些）。也 就是说，对应部门 10、20、30、40、50、60 和 70 的数据会放在一个块上，一旦插入部门 80，就会使用一个新块。这并不是说数据按一种有序的方式 存储，而是说如果按这种顺序插入部门，它们会很自然地放在一起。如果按下面的顺序插入部门，即先插入 10、80、20、30、40、50、60，然后插入 70，那么最后一个部门（70）将放在新增的块上。稍后会看到，数据的大小以及数据插入的顺序都会影响每个块上都存储的聚簇键个数。

因 此，SIZE 测试控制着每块上聚簇键的最大个数。这是对聚簇空间利用率影响最大的因素。如果把这个 SIZE 设置得太高，那么每个块上的键就会很少，我们会 不必要地使用更多的空间。如果设置得太低，又会导致数据过分串链，这又与聚簇本来的目的不符，因为聚簇原本是为了把所有相关数据都存储在一个块上。对于聚 簇来说，SIZE 是最重要的参数。

下 面来看聚簇上的聚簇索引。向聚簇中放数据之前，需要先对聚簇建立索引。可以现在就在聚簇中创建表，但是由于我们想同时创建和填充表，而有数据之前必须有一 个聚簇索引，所以我们先来建立聚簇索引。聚簇索引的任务是拿到一个聚簇键值，然后返回包含这个键的块的块地址。实际上这是一个主键，其中每个聚簇键值指向 聚簇本身中的一个块。因此，我们请求部门 10 的数据时，Oracle 会读取聚簇键，确定相应的块地址，然后读取数据。聚簇键索引如下创建：

```
ops$tkyte@ORA10GR1> create index emp_dept_cluster_idx

  2 on cluster emp_dept_cluster
  3 /
lex created.
```

对 于索引平常有的存储参数，聚簇索引都可以有，而且聚簇索引可以存储在另一个表空间中。它就像是一个常规的索引，所以同样可以在多列上建立；聚簇索引只不过 恰好是一个聚簇的索引，另外可以包含对应完全 null 值的条目（这很有意思，之所以要指出这一点，原因将在第 11 章解释）。注意，在这个 CREATE INDEX 语句中，并没有指定列的一个列表，索引列可以由 CLUSTER 定义本身得出。现在我们可以在聚簇中创建表了：

```
ops$tkyte@ORA10GR1> create table dept

  2 (          deptno number(2) primary key,

  3          dname varchar2(14),
```

```
  4          loc varchar2(13)

  5 )

  6 cluster emp_dept_cluster(deptno)

  7 /
ble created.


ops$tkyte@ORA10GR1> create table emp

  2 (          empno number primary key,

  3          ename varchar2(10),

  4          job varchar2(9),

  5          mgr number,

  6          hiredate date,

  7          sal number,

  8          comm number,

  9          deptno number(2) references dept(deptno)

 10 )

 11 cluster emp_dept_cluster(deptno)

 12 /
ble created.
```

在 此，与“正常”表惟一的区别是，我们使用了 CLUSTER 关键字，并告诉 Oracle 基表的哪个列会映射到聚簇本身的聚簇键。要记住，这里的段是聚簇，因 此这个表不会有诸如 TABLESPACE、PCTFREE 等段属性，它们都是聚簇段的属性，而不是我们所创建的表的属性。现在可以向这些表加载初始数据 集：

```
ops$tkyte@ORA10GR1> begin

  2    for x in ( select * from scott.dept )

  3 loop

  4          insert into dept
```

```
  5                 values ( x.deptno, x.dname, x.loc );

  6         insert into emp

  7                 select *

  8                 from scott.emp

  9                 where deptno = x.deptno;

 10 end loop;
 11 end;
 12 /

PL/SQL procedure successfully completed.
```

你 可能会奇怪，为什么不是插入所有 DEPT 数据，然后再插入所有 EMP 数据呢？或者反之，先插入所有 EMP 数据，然后插入所有 DEPT 数据？为什么要像这样 按 DEPTNO 逐个地加载数据呢？原因就在于聚簇的设计。我们在模拟一个聚簇的大批量初始加载。如果写加载所有 DEPT 行，每个块上就会有 7 个键（根据前 面指定的 SIZE 1024 设置），这是因为 DEPT 行非常小（只有几个字节）。等到加载 EMP 行时，可能会发现有些部门的数据远远超过了 1024 字节。这样就会在那些聚簇 键块上导致过度的串链。Oracle 会把包含这些信息的一组块串链或链接起来。如果同时加载对应一个给定聚簇键的所有数据，就能尽可能紧地塞满块，等空间 用完时再开始一个新块。Oracle 并不是在每个块中放最多 7 个聚簇键值，而是会适当地尽可能多地放入聚簇键值。

下 面给出一个小例子，从中可以看出这两种方法的区别。我们将向 EMP 表增加一个很大的列：CHAR(1000)。加这个列是为了让 EMP 行远远大于现在的大 小。我们将以两种方式加载聚簇表：先加载 DEPT，再加载 EMP。第二次加载时，则会按部门编号来加载：先是一个 DEPT 行，然后是与之相关的所有 EMP 行，然后又是一个 DEPT 行。我们将查看给定情况下每一行最后在哪个块上，从而得出哪种方法最好，能最好地实现将数据按 DEPTNO 共同存储的目标。我们 的 EMP 表如下：

```
ops$tkyte@ORA10GR1> create table emp

  2 (       empno number primary key,

  3         ename varchar2(10),

  4         job varchar2(9),

  5         mgr number,

  6         hiredate date,

  7         sal number,

  8         comm number,
```

```
  9          deptno number(2) references dept(deptno),

 10          data char(1000)

 11 )
 12 cluster emp_dept_cluster(deptno)
 13 /
ble created.
```

向 DEPT 和 EMP 表中加载数据时，可以看到许多 EMP 行与 DEPT 行不在同一个块上（DBMS_ROWID）是一个内置包，可以用于查看行 ID 的内容）：

```
ops$tkyte@ORA10GR1> insert into dept

  2          select * from scott.dept;

ows created.


ops$tkyte@ORA10GR1> insert into emp

  2          select emp.*, '*' from scott.emp;

rows created.


ops$tkyte@ORA10GR1> select dept_blk, emp_blk,

  2                 case when dept_blk <> emp_blk then '*' end flag,

3                   deptno

  4          from (

  5                 select dbms_rowid.rowid_block_number(dept.rowid) dept_blk,

  6                        dbms_rowid.rowid_block_number(emp.rowid) emp_blk,

7                     dept.deptno

  8                 from emp, dept

  9                 where emp.deptno = dept.deptno

10                )

 11         order by deptno
```

```
12 /

 DEPT_BLK     EMP_BLK      F      DEPTNO

---------- ---------- -         ----------

      4792       4788 *              10

      4792       4788 *              10

      4792       4791 *              10

      4792       4788 *              20

      4792       4788 *              20

      4792       4792                20

      4792       4792                20

      4792       4791 *              20

      4792       4788 *              30

      4792       4792                30

      4792       4792                30

      4792       4792                30

      4792       4792                30

      4792       4788 *              30

rows selected.
```

一半以上的 EMP 行与 DEPT 行不在同一个块上。如果使用聚簇键而不是表键来加载数据，会得到以下结果：

```
ops$tkyte@ORA10GR1> begin

  2          for x in ( select * from scott.dept )

3          loop

  4                  insert into dept
```

```
  5                     values ( x.deptno, x.dname, x.loc );
  6               insert into emp
  7                     select emp.*, 'x'
  8                     from scott.emp
  9                     where deptno = x.deptno;
 10       end loop;
11       end;
12 /

 PL/SQL procedure successfully completed.


 ops$tkyte@ORA10GR1> select dept_blk, emp_blk,
  2          case when dept_blk <> emp_blk then '*' end flag,
3          deptno
4 from (
  5          select dbms_rowid.rowid_block_number(dept.rowid) dept_blk,
  6                 dbms_rowid.rowid_block_number(emp.rowid) emp_blk,
7                  dept.deptno
  8          from emp, dept
  9          where emp.deptno = dept.deptno
10 )
11 order by deptno
12 /

    DEPT_BLK     EMP_BLK      F       DEPTNO

    ----------   ----------   -       ----------

          12           12                   10
```

```
        12         12                 10
        12         12                 10
        11         11                 20
        11         11                 20
        11         11                 20
        11         12      *          20
        11         11                 20
        10         10                 30
        10         10                 30
        10         10                 30
        10         10                 30
        10         10                 30
        10         11      *          30

rows selected.
```

注意 你 看到的结果可能与此不同，因为从 SCOTT.DEPT 表获取行的顺序可能会（而且将会）改变这个结果，另外使用 ASSM 或是 MSSM 也会带来影响。不过， 概念应该很清楚：如果把对应 DEPTNO=n 的行放在一个给定块上，然后再加载对应 DEPTNO=n 的员工行，就能得到最佳的聚簇。

大 多数 EMP 行都与 DEPT 行在同一个块上。这个例子少有些技巧，因为我故意把 SIZE 参数设置得很小以便得出结论，不过这里建议的方法对于聚簇的初始加载 确实是正确可行的。由此可以确保，如果某些聚簇键超过了估计的 SIZE，最后大多数数据都会聚簇到同一个块上。如果一次加载一个表，则做不到这一点。

这种技术只适用于聚簇的初始加载，在此之后，只有在事务认为必要的时候才应使用这个技术。你不会为了专门使用聚簇去调整应用。

这里存在一个很让人诧异的困惑。许多人错误地认为一个 rowid 能惟一地标识数据库中的一个行，给定一个 rowid，就能得出这一行来自哪个表。实际上，这是做不到的。从聚簇可以得到（而且将得到）重复的 rowid。例如，执行以上代码后，你会发现：

```
ops$tkyte@ORA10GR1> select rowid from emp

2 intersect
3 select rowid from dept;

)WID
```

```
----------
AAOniAAJAAAAAAKAAA
AAOniAAJAAAAAAKAAB
AAOniAAJAAAAAALAAA
AAOniAAJAAAAAAMAAA
```

DEPT 中为各行分配的每个 rowid 也同时分配给了 EMP 中的行。这是因为，要由表和行 ID 共同地惟一标识一行。Rowid 伪列只是在一个表中惟一。

我还发现，许多人认为聚簇对象是一种神秘的对象，以为没有人用它，所有人都只是在使用普通表。事实上，每次你使用 Oracle 的时候都会使用聚簇。例如，许多数据字典就存储在各个聚簇中：

```
sys@ORA10GR1> break on cluster_name

sys@ORA10GR1> select cluster_name, table_name

  2  from user_tables

  3  where cluster_name is not null

  4  order by 1;


USTER_NAME                     TABLE_NAME

------------------------------ ------------------------------

COBJ#                          CCOL$
                                 CDEF$
FILE#_BLOCK#                   UET$
                                 SEG$
MLOG#                          MLOG$
                                 SLOG$
OBJ#                           ICOL$
                                 CLU$
                                 COL$
                                 TYPE_MISC$
                                 VIEWTRCOL$
                                 ATTRCOL$
                                 SUBCOLTYPE$
                                 COLTYPE$
                                 LOB$
                                 TAB$
                                 IND$
                                 ICOLDEP$
                                 OPQTYPE$
                                 REFCON$
```

```
                               LIBRARY$
                               NTAB$
OBJ#_INTCOL#                   HISTGRM$

                                    C_RG# RGROUP$

                               RGCHILD$
TOID_VERSION#                  TYPE$
                               COLLECTION$
                               METHOD$
                               RESULT$
                               PARAMETER$
                               ATTRIBUTE$
TS#                            TS$
                               FET$
USER#                          USER$
                               TSQ$
ION_SCN_TO_TIME                SMON_SCN_TIME
rows selected.
```

可 以看到，与对象相关的大多数数据都存储在一个聚簇（C_OBJ#聚簇）中：16 个表都在同一个块中。这里存储的主要是与列相关的信息，所以关于表或索引列 集的所有信息都物理地存储在同一个块上。这是有道理的：Oracle 解析一个查询时，它希望访问所引用的表中所有列的数据。如果这些数据分布得到处都是， 就要花一些时间才能把它们收集起来。如果数据都在一个块上，通常就能很容易地得到。

什么时候要使用聚簇呢？可能反过来回答什么时候不应该使用聚簇会更容易一些：

- 如果预料到聚簇中的表会大量修改：必须知道，索引聚簇会对 DML 的性能产生某种负面影响（特别是 INSERT 语句）。管理聚簇中的数据需要做更多的工作。
- 如果需要对聚簇中的表执行全表扫描：不只是必须对你的表中的数据执行全面扫描，还必须对（可能的）多个表中的数据进行全面扫描。由于需要扫描更多的数据，所以全表扫描耗时更久。
- 如果你认为需要频繁地 TRUNCATE 和加载表：聚簇中的表不能截除。这是显然的，因为聚簇在一个块上存储了多个表，必须删除聚簇表中的行。

因 此，如果数据主要用于读（这并不表示“从来不写”；聚簇表完全可以修改），而且要通过索引来读（可以是聚簇键索引，也可以是聚簇表上的其他索引），另外会 频繁地把这些信息联结在一起，此时聚簇就很适合。应用找出逻辑上相关而且总是一起使用的表，设计 Oracle 数据字典的人就是这样做的，他们把与列相关的 所有信息都聚簇在一起。

索引聚簇表小结

利用聚簇表，可以物理地“预联结”数据。使用聚簇可以把多个表上的相关数据存储在同一个数据库块上。聚簇有助于完成总是把数据联结在一起或者访问相关数据集（例如，部门 10 中的每一个人）的读密集型操作。

聚簇表可以减少 Oracle 必须缓存的块数，从而提供缓存区缓存的效率。不好的一面是，除非你能正确地计算出 SIZE 参数设置，否则聚簇在空间利用方面可能效率低下，而且可能会使有大量 DML 的操作变慢。

10.6 散列聚簇表

散列聚簇表（Hash clustered table） 在概念上与前面介绍的索引聚簇表非常相似，只有一个主要区别：聚簇键索引被一个散列函数所取代。表中的数据就是索引；这里没有物理索引。Oracle 会取 得一行的键值，使用每个内部函数或者你提供的每个函数对其计算散列，然后使用这个散列值得出数据应该在磁盘上的哪个位置。不过，使用散列算法来定位数据有一个副作用，如果不向表增加一个传统的索引，将无法对散列聚簇中的表完成区间扫描。在一个索引聚簇中，如果有以下查询：

```
select * from emp where deptno between 10 and 20
```

它就能利用聚簇键索引来找到这些行。在一个散列聚簇中，这个查询会导致一个全表扫描，除非 DEPTNO 列上已经有一个索引。如果没有使用一个支持区间扫描的索引，就只能在散列键上完成精确搜索（包括列表和子查询）。

理 想情况下，散列键值均匀分布，并且有一个散列函数可以将这些散列键值均匀地分布到为散列聚簇分配的所有块上，从查询利用一个 I/O 就能直接找到数据。但在 实际中，最后可能会有多个散列键值散列到同一个数据库块地址，而且这个块上放不下这么多散列键值。这就会导致块串链，Oracle 必须用一个链表把块串起 来，来保存散列到这个块的所有行。现在，当需要获取与某个散列键匹配的行时，可能必须访问多个块。

类似于编程语言中的散列表，数据库中的散列表有固定的“大小”。创建表时，必须确定这个表中将有多少个散列键（而且这个数永远不变）。但散列表的大小并不限制其中能放的行数。

图 10 -9 是一个散列聚簇的图形表示，这里创建了表 EMP。客户发出一个查询，其中的谓词条件中使用了散列聚簇键，Oracle 会应用散列函数确定数据应该在哪 个块中。然后读这个块来找到数据。如果存在许多冲突，或者 CREATE CLUSTER 的 SIZE 参数估计过低，Oracle 会分配溢出块与原来的块串链起来。

图 10-9 散列聚簇示意图

创 建散列聚簇时，还是使用创建索引聚簇时所用的同样的 CREATE CLUSTER 语句，不过选项不同。这里只是要增加一个 HASHKEYS 选项来指定散列表的大小。Oracle 得到你的 HASHKEYS 值，将其 “舍入” 为与之最接近的质数（散列键数总是一个质数）。然后 Oracle 再将 SIZE 参数乘以修改后的 HASHKEYS 值，计算出一个值。再根据这个值为聚簇分配 空间，也就是说，至少要分配这么多字节的空间。这与前面的索引聚簇有很大差别，索引聚簇

会在需要时动态地分配空间，散列聚簇则要预留足够的空间来保存（HASHKEYS/trunc(blocksize/SIZE)）字节的数据。例如，如果将 SIZE 设置为 1,500 字节，而且块大小为 4KB， Oracle 会在每个块上存储两个键。如果你计划有 1,000 个 HASHKEY，Oracle 就分配 500 个块。

有 意思的是，不同于计算机语言中的传统散列表，这里允许有散列冲突，实际上，许多情况下还需要有冲突。还是用前面的 DEPT/EMP 例子，可以根据 DEPTNO 列建立一个散列聚簇。显然，多个行会散列到同一个值，这正是你希望的（因为它们有相同的 DEPTNO）。这就反映了聚簇某些方面的特点：要把 类似的数据聚簇在一起。正是由于这个原因，所以 Oracle 要求你指定 HASHKEY（你预计一段时间会有多少个部门号）和 SIZE（与各个部门号相关联 的数据量）。Oracle 会分配一个散列表来保存 HASHKEY 个部门，每个部门有 SIZE 字节的数据。你想避免的是无意的散列冲突。显而易见，如果就散 列表的大小设置为 1,000（实际上是 1,099，因为散列表的大小总是质数，而且 Oracle 会为你找出与之最接近的质数），而你在表中放入了 1,010 个部门，就至少会存在一个冲突（两个不同部门散列到同一个值）。无意的散列冲突是要避免的，因为它们会增加开销，使块串链的可能性增加。

要 查看散列聚簇会用哪种空间，下面就使用一个小工具——存储过程 SHOW_SPACE（有关这个过程的详细介绍，请参见本书最前面的“环境配置”一节），这 一章和下一章都就使用这个小工具。这个例程只是使用 DBMS_SPACE 提供的包来得到数据库中段所用存储空间的详细信息。

如果发出以下 CREATE CLUSTER 语句，可以看到它分配的存储空间如下：

```
ops$tkyte@ORA10GR1> create cluster hash_cluster

2 ( hash_key number )
3 hashkeys 1000
4 size 8192
5 tablespace mssm
6 /
uster created.


ops$tkyte@ORA10GR1> exec show_space( 'HASH_CLUSTER', user, 'CLUSTER' )

e Blocks.....                  0
tal Blocks.....              1,024
tal Bytes.....           8,388,608
tal MBytes.....                8
used Blocks.....              14
used Bytes.....          114,688
st Used Ext FileId.....        9
st Used Ext BlockId.....   1,033
st Used Block.....           114

PL/SQL procedure successfully completed.
```

可 以看到，为表分配的总块数为 1,024。其中 14 个块未用（空闲）。另外有 1 个块用于维护表开销，以管理区段。因此，有 1,099 个块在这个对象的 HWM 之下，这些是聚簇使用的块。1,099 是一个质数，而且正好是大于 1000 的最小质数，由于块大小为 8KB，可

以看到，Oracle 实际上会分配 （8,192×1,099）字节。由于区段的”舍入“而且/或者通过使用本地管理的表空间（区段的大小一致），实际分配的空间（8,388,608）比这 个数稍高一些。

这 个例子指出，关于散列聚簇需要注意以下问题。一般地，如果创建一个空表，该表在 HWM 下的块数为 0.如果对它执行全表扫描，达到 HWM 就会停止。对于一个 散列聚簇，表一开始就很大，需要花更长的时间创建，因为 Oracle 必须初始化各个块（而对于一般的表，这个动作通常在数据增加到表时才发生）。散列聚簇 表有可能把数据放在第一个块和最后一个块中，而中间什么都没有。对一个几乎为空的散列聚簇进行前面扫描与全面扫描一个满的散列聚簇所花的时间是一样的。这 不一定是件坏事：建立散列聚簇的本来目的是为了根据散列键查找从而非常快地访问数据。而不是为了频繁地对它进行全面扫描。

现在可以开始把表放在散列聚簇中，仍采用前面索引聚簇所用的方法：

```
Ops$tkyte@ORA10GR1> create table hashed_table

  2 ( x number, data1 varchar2(4000), data2 varchar2(4000) )

  3 cluster hash_cluster(x);
Table created.
```

为 了看出散列聚簇可能有哪些区别，我建立了一个小测试。首先创建一个散列聚簇，在其中加载一些数据，再将这些数据复制到一个有传统索引的“常规“表中，然后 对各个表完成一些随机读（对每个完成的随机读是一样的）。通过使用 runstats、SQL_TRACE 和 TKPPOF，可以确定各个表的特征。以下先完 成散列聚簇和表的建立，其后是分析：

```
ops$tkyte@ORA10GR1> create cluster hash_cluster
  2 ( hash_key number )
  3 hashkeys 75000
  4 size 150
  5 /
Cluster created.

ops$tkyte@ORA10GR1> create table t_hashed

  2 cluster hash_cluster(object_id)

  3 as
  4 select *
  5 from all_objects
  6 /
Table created.


ops$tkyte@ORA10GR1> alter table t_hashed add constraint

  2 t_hashed_pk primary key(object_id)

  3 /
Table altered.
```

```
ops$tkyte@ORA10GR1> begin

  2 dbms_stats.gather_table_stats( user, 'T_HASHED', cascade=>true );

  3 end;
  4 /
PL/SQL procedure successfully completed.
```

在此创建了一个 SIZE 为 150 字节的散列聚簇。这是因为，我认为我的表中一行的平均大小大约是 100 字节，但是根据实际数据，具体的行大小可能会上下浮动。然后在这个聚簇中创建并填充一个表，作为 ALL_OBJECTS 的一个副本。

接下来，创建这个表的传统版本的“克隆“（即相应的堆组织表）：

```
ops$tkyte@ORA10GR1> create table t_heap
  2 as
  3 select *
  4 from t_hashed
  5 /
Table created.



ops$tkyte@ORA10GR1> alter table t_heap add constraint

  2 t_heap_pk primary key(object_id)

  3 /
Table altered.

ops$tkyte@ORA10GR1> begin
  2 dbms_stats.gather_table_stats( user, 'T_HEAP', cascade=>true );
  3 end;
  4 /
PL/SQL procedure successfully completed.
```

现 在，我需要一些“随机“的数据，用来从各个表中抽取行。为此，我只是把所有 OBJECT_ID 选择到一个数组中，然后随机地排序，从而以一种分散的方式命 中表的各个块。我使用了一个 PL/SQL 包来定义和声明这个数组，并使用一些 PL/SQL 代码来”准备“这个数组，填入随机数据：

```
ops$tkyte@ORA10GR1> create or replace package state_pkg

  2 as
  3         type array is table of t_hashed.object_id%type;
  4         g_data array;
  5 end;
  6 /
Package created.

ops$tkyte@ORA10GR1> begin
```

```
  2          select object_id bulk collect into state_pkg.g_data
  3          from t_hashed
  4          order by dbms_random.random;
  5 end;
  6 /
PL/SQL procedure successfully completed.
```

要看到各个表完成的工作，我使用了以下代码块（如果把这里出现的 HASHED 都代之以 HEAP，就可以得到另一个要测试的代码块）：

```
ops$tkyte@ORA10GR1> declare
  2          l_rec t_hashed%rowtype;
  3 begin
  4          for i in 1 .. state_pkg.g_data.count
  5          loop
  6                  select * into l_rec from t_hashed
  7                  where object_id = state_pkg.g_data(i);
  8          end loop;
  9 end;
  10 /
PL/SQL procedure successfully completed.
```

接 下来，就前面的代码块（以及用 HEAP 取代 HASHED 得到的代码块）运行 3 次。第一次运行是系统“热身“，以避免以后再完成硬解析。第二次运行这个代码 块时，我使用 runstats 来查看二者的主要差别：先运行散列实现，然后运行堆实现。第三次运行代码块时，我启用了 SQL_TRACE，从而能看到一个 TKPPOF 报告。runstats 运行的报告如下：

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_stop(10000);

Run1 ran in 263 hsecs
Run2 ran in 268 hsecs
run 1 ran in 98.13% of the time


Name                                  Run1          Run2          Diff

LATCH.cache buffers chains          99,891       148,031        48,140

STAT...Cached Commit SCN refer      48,144             0       -48,144

STAT...no work - consistent re      48,176             0       -48,176

STAT...cluster key scans            48,176             0       -48,176

STAT...cluster key scan block       48,176             0       -48,176

STAT...table fetch by rowid              0        48,176        48,176
```

```
STAT...rows fetched via callba                0         48,176       48,176

STAT...buffer is not pinned co               48,176         96,352       48,176

STAT...index fetch by key                     0         48,176       48,176
STAT...session logical reads                 48,901        145,239       96,338
STAT...consistent gets                       48,178        144,530       96,352

STAT...consistent gets from ca               48,178        144,530       96,352

STAT...consistent gets - exami                1        144,529      144,528



Run1 latches total versus runs -- difference and pct

 Run1         Run2          Diff          Pct
 347,515      401,961       54,446        86.45%
```

现 在，从墙上的时钟来看，两个仿真运行的时间是一样的。因为我有一个足够大的缓存区缓存来缓存这些结果，所以这在预料之中。不过，要注意一个重要差别，缓存 区缓存链的闩大幅减少。第一个实现（散列实现）使用的闩少得多，这说明在一个读密集型环境中，散列实现应该能更好地扩缩，因为它需要的串行化资源（这些资 源要求某种程度的串行化）更少。其原因完全在于，与 HEAP 表相比，散列实现需要的 I/O 显著减少，可以看到，报告中的一致获取统计就能反映出这点。 TKPPOF 反映得更清楚：

```
SELECT * FROM T_HASHED WHERE OBJECT_ID = :B1

call             count          cpu      elapsed          disk         query       current        rows

-------          ------         --------      ----------      ----------      ----------      ----------
----------

Parse                1         0.00          0.00             0             0             0
 0

Execute          48174         4.77          4.83             0             2             0
0

Fetch            48174         1.50          1.46      0 48174              0         48174

-------          ------         --------      ----------      ----------      ----------      ----------
----------

total            96349         6.27          6.30             0         48176             0      48174
```

```
Rows        Row Source Operation

-------         ---------------------------------------------------

48174       TABLE ACCESS HASH T_HASHED (cr=48174 pr=0 pw=0 time=899962 us)

********************************************************************************
SELECT * FROM T_HEAP WHERE OBJECT_ID = :B1

call              count           cpu     elapsed           disk         query       current        rows

-------           ------        --------      ----------      ----------      ----------      ----------
----------

Parse                 1          0.00          0.00              0              0              0
 0

Execute       48174           5.37          5.02              0              0              0
0

Fetch           48174           1.36          1.32              0      144522              0     48174

-------           ------        --------      ----------      ----------      ----------      ----------
----------

total             96349           6.73          6.34              0      144522              0     48174

Rows        Row Source Operation

-------         ---------------------------------------------------

48174       TABLE ACCESS BY INDEX ROWID T_HEAP (cr=144522 pr=0 pw=0
time=1266695 us)

48174       INDEX UNIQUE SCAN T_HEAP_PK (cr=96348 pr=0 pw=0 time=700017
us)(object ...
```

HASHED 实 现只是把传递到查询的 OBJECT_ID 转换为要读取的一个 FILE/BLOCK，并且直接读，这里没有索引。不过，HEAP 表则不同，它必须对每一行在 索引上完成两个 I/O。TKPROF Row Source Operation 行中的 cr=96348 清楚地显示了对索引做了多少次一致读。每次查找 OBJECT_ID = ：B1 时，Oracle 必须得到索引的根块，然后找出包含该行位置的叶子块。接下来必须得到叶子块信息，其中包含行的 ROWID，再利用第 3 个 I/O 在表 中访问这个行。HEAP 表完成的 I/O 是 HASHED 实现的 3 倍。

这里的要点是：

- 散 列聚簇完成的 I/O（查询列）少得多。这正是我们期望的。查询只是取随机的 OBJECT_ID，对其完成散列，然后找到块。散列聚簇至少要做一次 I/O 来 得到数据。有索引的传统表则必须完成索引扫描，然后要根据 rowid 访问表，才能得到同样的答案。在这个例子中，索引表必须至少完成 3 个 I/O 才能得到数 据。

- 不 论用于什么目的，散列聚簇查询与索引查询所用的 CPU 是一样的，尽管它访问缓存区缓存的次数只是后者的 1/3。同样，这也在预料之中。执行散列是一个 CPU 相当密集的操作，执行索引查询则是一个 I/O 密集的操作，这里要做个权衡。不过，随着用户数的增加，可以想见，散列聚簇查询能更好地扩缩，因为要想 很好地扩缩，就不能太过频繁地访问缓存区缓存。

最 后一点很重要。使用计算机时，我们所关心的就是资源及其利用。如果存在大量 I/O，并且像这里一样，所执行的查询要根据键做大量的读操作，此时散列聚簇就 能提供性能。如果已经占用了大量 CPU 时间，再采用散列聚簇反而会降低性能，因为它需要更多的 CPU 时间来执行散列。不过，如果耗用更多 CPU 时间的原因 是缓冲区缓存的闩存在自旋，那么散列聚簇就能显著减少所需的 CPU 时间。这就说明了为什么有些经验在实际系统中不适用；有些经验对于你来说也许很合适，但 是在类似但不同的条件下，这些经验可能并不可行。

散列聚簇有一个特例，称为单表散列聚簇（single table hash cluster）。 这是前面介绍的一般散列聚簇的优化版本。它一次只支持聚簇中的一个表（必须 DROP（删除）单表散列聚簇中现有的表，才能在其中创建另一个表）。另外，如 果散列键和数据行之间存在一对一的映射，访问行还会更快一些。这种散列聚簇是为以下情况设计的：如果你想按主键来访问一个表，但是不关心其他表是否与这个 表聚簇在一起存储。如果你需要按 EMPNO 快速地访问员工记录，可能就需要一个单表散列聚簇。我在一个单表散列聚簇上执行了前面的测试，发现性能比一般的 散列聚簇还要好。不过，甚至还可以将这个例子更进一步，由于 Oracle 允许你编写自己的散列函数（而不是使用 Oracle 提供的默认散列函数），所以能 利用这一点。不过，你只能使用表中可用的列，而且编写自己的散列函数时只能使用 Oracle 的内置函数（例如，不能有 PL/SQL 代码）。由于上例中 OBJECT_ID 是一个介于 1～75,000 之间的数，充分利用这一点，我建立了自己的“散列函数”：就是 OBJECT_ID 本身。采用这种方式，可以 保证绝对不会有散列冲突。综合在一起，我如下创建一个单表散列聚簇（有我自己的散列函数）：

```
ops$tkyte@ORA10GR1> create cluster hash_cluster
  2 ( hash_key number(10) )
  3 hashkeys 75000
  4 size 150
  5 single table
  6 hash is HASH_KEY
  7 /
Cluster created.
```

这里只是增加了关键字 SINGLE TABLE，使之作为一个单步散列聚簇。在这种情况下，我的散列函数就是 HASH_KEY 聚簇键本身。这是一个 SQL 函数，所以如果我愿意，也可以使用 trunc (mod(hash_key/324+278,555)/abs(hash_key+1))（并不是说这是一个好的散列函数，这只是说明，只要我们愿意， 完全可以使用一个复杂的函数）。我使用了 NUMBER(10) 而不是 NUMBER，这是因为散列值必须是一个整数，所以不能有任何小数部分。下面，在这个 表单散列聚簇中创建表：

```
ops$tkyte@ORA10GR1> create table t_hashed
  2         cluster hash_cluster(object_id)
  3         as
  4         select OWNER, OBJECT_NAME, SUBOBJECT_NAME,

  5            cast( OBJECT_ID as number(10) ) object_id,

  6                DATA_OBJECT_ID, OBJECT_TYPE, CREATED,
  7                LAST_DDL_TIME, TIMESTAMP, STATUS, TEMPORARY,
  8                GENERATED, SECONDARY
  9         from all_objects
 10 /
Table created.
```

以上建立了散列表。注意这里使用了 CAST 内置函数将 OBJECT_ID 强制转换为它本来的数据类型。像前面一样运行测试（每个代码块运行 3 次），这一次 runstats 的输出表明情况更好了：

```
Run1 ran in 224 hsecs
Run2 ran in 269 hsecs
run 1 ran in 83.27% of the time

Name                                  Run1          Run2          Diff
STAT...index fetch by key                0        48,178        48,178

STAT...buffer is not pinned co      48,178        96,356        48,178

STAT...table fetch by rowid              0        48,178        48,178
STAT...cluster key scans            48,178             0       -48,178
STAT...session logical reads        48,889       145,245        96,356
STAT...consistent gets              48,184       144,540        96,356

STAT...consistent gets from ca      48,184       144,540        96,356

STAT...consistent gets - exami      48,184       144,540        96,356

LATCH.cache buffers chains          51,663       148,019        96,356


Run1 latches total versus runs -- difference and pct

Run1        Run2            Diff            Pct

298,987   402,085         103,098          74.36%

PL/SQL procedure successfully completed.
```

这个单表散列聚簇需要更少的缓冲区缓存闩来完成处理（它能更快地结束数据查找，而且能得到更多的信息）。因此，TKPROF 报告显示出这一次的 CPU 使用量大幅减少：

```
SELECT * FROM T_HASHED WHERE OBJECT_ID = :B1


call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  48178      4.45       4.52          0          2          0           0
Fetch    48178      0.67       0.82          0      48178          0       48178
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total    96357      5.12       5.35          0      48180          0       48178


Rows     Row Source Operation
-------  ---------------------------------------------------
48178    TABLE ACCESS HASH T_HASHED (cr=48178 pr=0 pw=0 time=551123 us)

********************************************************************************
SELECT * FROM T_HEAP WHERE OBJECT_ID = :B1


call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  48178      5.38       4.99          0          0          0
```

```
0

Fetch        48178      1.25       1.65          0     144534          0     48178

-------      ------   --------   ----------  ----------  ----------  ----------
----------

total        96357      6.63       6.65          0     144534          0     48178


Rows     Row Source Operation

-------  ---------------------------------------------------

48178    TABLE ACCESS BY INDEX ROWID T_HEAP (cr=144534 pr=0 pw=0
time=1331049 us)

48178    INDEX UNIQUE SCAN T_HEAP_PK (cr=96356 pr=0 pw=0 time=710295
us)(object...
```

散列聚簇表小结

以上就是散列聚簇的核心。散列聚簇在概念上与索引聚簇很相似，只不过没有使用聚簇索引。在这里，数据就是索引。聚簇键散列到一个块地址上，数据应该就在那个位置上。关于散列聚簇，需要了解以下要点：

- 散 列聚簇一开始就要分配空间。Oracle 根据你的 HASHKEYS 和 SIZE 来计算 HASHKEYS/trunc（blocksize/SIZE），立即 分配空间，并完成格式化，一旦将第一个表放入这个聚簇中，任何全面扫描都会命中每一个已分配的块。在这方面，它与其他的所有表都不同。

- 散列聚簇中的 HASHKEY 数是固定大小的。除非重新聚簇，否则不能改变散列表的大小。这并不会限制聚簇中能存储的数据量，它只是限制了能为这个聚簇生成的惟一散列键的个数。如果 HASHKEY 值设置得太低，可能因为无意的散列冲突影响性能。

- 不 能在聚簇键上完成区间扫描。诸如 WHERE cluster_key BETWEEN 50 AND 60 谓词条件不能使用散列算法。介于 50～60 之间的可能值有无限多个，服务器必须生成所有可能的值，并分别计算散列，来查看相应位置是否有数据。这是不 可能的。如果你在一个聚簇键上使用区间扫描，而且没有使用传统索引，实际上会全面扫描这个聚簇。

散列聚簇适用于以下情况：

你很清楚表中会有多少行，或者你知道一个合理的上界。HASHKEY 和 SIZE 参数的大小要正确，这对于避免聚簇重建至关重要。

与 获取操作相比，DML（特别是插入）很轻。这说明必须在数据获取的优化和新数据的创建之间有所权衡。有多少个插入算轻，对此没有定论，对某些人来说，每个 单位时间有 100,000 个插入就算轻，而在另一个人来看，可能每单位时间 100 个插入才算轻——这取

决于具体的数据获取模式。更新不会引入严重的开销， 除非更新了 HASHKEY（不过这可不是一个好主意，因为更新 HASHKEY 会导致行迁移）。

经常按 HASHKEY 值访问数据。例如，假如有一个零件表，并按零件号来访问这些零件。查找表特别适合采用散列聚簇。

10.7 有序散列聚簇表

有序散列聚簇是 Oracle 10g 中新增的。其中不仅有前面所述的散列聚簇的有关性质，还结合了 IOT 的一些性质。如果经常使用类似于以下的查询来获取数据，有序散列聚簇就最适合：

```
Select *
From t
Where KEY=:x
Order by SORTED_COLUMN
```

也就是说，要按某个键获取数据，但要求这些数据按另外每个列排序。通过使用有序散列聚簇，Oracle 可以返回数据而根据不用执行排序。这是通过插入时按键的有序物理存储数据做到的。假设有一个客户订单表：

```
ops$tkyte@ORA10G> select cust_id, order_dt, order_number

  2 from cust_orders
  3 order by cust_id, order_dt;

CUST_ID   ORDER_DT                                ORDER_NUMBER

----------  ---------------------------------------      ------------------------

         1   31-MAR-05 09.13.57.000000 PM                  21453

             11-APR-05 08.30.45.000000 AM                  21454

             28-APR-05 06.21.09.000000 AM                  21455

         2   08-APR-05 03.42.45.000000 AM                  21456

             19-APR-05 08.59.33.000000 AM                  21457

             27-APR-05 06.35.34.000000 AM                  21458

             30-APR-05 01.47.34.000000 AM                  21459

7 rows selected.
```

这个表存储在一个有序散列聚簇中，在此 HASH 键是 CUST_ID，并按 ORDER_DT 字段排序。如图 10-10 所示，这里 1、2、3、4、…分别表示每个块上依序存储的记录。

图 10-10 有序散列聚簇示意图

创建有序散列聚簇与创建其他聚簇基本相同。要建立一个能存储以上数据的有序散列聚簇，可以使用下面的语句：

```
ops$tkyte@ORA10G> CREATE CLUSTER shc

  2 (
  3         cust_id NUMBER,
  4         order_dt timestamp SORT
  5 )
  6 HASHKEYS 10000
  7 HASH IS cust_id
  8 SIZE 8192
  9 /
Cluster created.
```

这 里引入了一个新的关键字：SORT。创建聚簇时，我们标识了 HASH IS CUST_ID，而且用关键字 SORT 增加了一个时间戳（timestamp）类型的 ORDER_DT。这说明，数据将按 CUST_ID 查找（查找条件是 CUST_ID = :X），而按 ORDER_DT 物理地获取和排序。从技术上讲，实际上这表示我们存储的数据将通过一个 NUMBER 列获取，但按一个 TIMESTAMP 列排 序。这里的列名并不重要，因为列名并不出现在 B*树或 HASH 聚簇中，不过一般约定是根据所表示的内容来命名。

这个 CUST_ORDERS 的相应 CREATE TABLE 语句如下所示：

```
ops$tkyte@ORA10G> CREATE TABLE cust_orders

  2 (       cust_id number,
  3         order_dt timestamp SORT,
  4         order_number number,
  5         username varchar2(30),
  6         ship_addr number,
  7         bill_addr number,
  8         invoice_num number
  9 )
 10 CLUSTER shc ( cust_id, order_dt )
```

```
  11 /
Table created.
```

我们就这个表的 CUST_ID 列映射到有序散列聚簇的散列键，并把 ORDER_DT 列映射到 SORT 列。使用 SQL*Plus 中的 AUTOTRACE，可以观察到，访问有序散列聚簇时，原本以为有的正常排序操作不见了：

```
ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> variable x number

ops$tkyte@ORA10G> select cust_id, order_dt, order_number

  2          from cust_orders
  3          where cust_id = :x
  4          order by order_dt;

Execution Plan
----------------------------------------------------------

0              SELECT STATEMENT Optimizer=ALL_ROWS (Cost=0 Card=4 Bytes=76)

1      0               TABLE ACCESS (HASH) OF 'CUST_ORDERS' (CLUSTER (HASH))



ops$tkyte@ORA10G> select job, hiredate, empno

  2          from scott.emp
  3          where job = 'CLERK'
  4          order by hiredate;

Execution Plan
----------------------------------------------------------

0              SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=3 Bytes=60)

1      0               SORT (ORDER BY) (Cost=3 Card=3 Bytes=60)

2      1                       TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)
(Cost=2 Card=3 ...

3      2                               INDEX (RANGE SCAN) OF 'JOB_IDX' (INDEX) (Cost=1
Card=3)



ops$tkyte@ORA10G> set autotrace off
```

我 对平常的 SCOTT.EMP 表做了一个查询（为了便于说明，这里现在 JOB 列上加了索引），这样可以看到我们预想的情况，以便做个比较：在上面的例子中， 后面展示的是 SCOTT.EMP 查询计划，前面则展示了希望以 FIFO 模式（像队列一样）访问数据时有序散列聚簇会为我们做什么。可以看到，有序散列聚簇 只完成了一步：它取得 CUST_ID = :X，对输入执行散列，找到第一行，如何开始读取这一行，因为有序散列聚簇中行已经是有序的。常规表则有很大不同：它会查找所有 JOB='CLERK'行 （可能分布在堆表中的各个地方），对它们进行排序，再返回第一行。

所 以，有序散列聚簇具备散列聚簇在获取方面的所有特点，因为它能得到数据而不必遍历索引；另外有序散列聚簇还拥有 IOT 的许多特性，因为数据可以按你选择的 每个字段（键）有序地存储。当输入数据按排序字段（键）的顺序到达时，这种数据结构就能很好地工作。也就是说，在一段时间内，数据按某个给定键值的递增有 序顺序到达。股票信息就满足这个要求。每天晚上你会得到一个新文件，其中填满了股票代码、日期以及相关的信息（日期是排序键，股票代码是散列键）。你按排 序键顺序地接收好加载这些数据。对于股票代码 ORCL，昨天的股票数据不会在今天之后才到达，你会先加载昨天的值，然后才是今天的值，之后是明天的值。如 果信息随机地到达（不按有序的顺序到来），插入过程中，这个数据结构很快就会受不了，因为必须移动大量的数据使得这些行在磁盘上物理有序。在这种情况下， 不建议采用有序散列聚簇（此时采用 IOT 可能很合适）。

使用这个结构时，应当考虑到散列聚簇同样的问题，另外还要考虑到一个约束条件，即数据应该按键值的有序顺序到达。

10.8 嵌套表

嵌 套表（nested table）是 Oracle 对象关系扩展的一部分。嵌套表是 Oracle 中的两种集合类型之一，它与关系模型中传统的"父/子表对"里的子表很相似。这是 数据元素的一个无序集，所有数据元素的数据类型都相同，可以是一个内置数据类型，也可以是一个对象数据类型。不过，还不仅如此，因为设计嵌套表是为了制造 一个假象，好像父表中的每一行都有其自己的子表。如果父表中有 100 行，那么就有 100 个虚拟的嵌套表。但实际来讲，物理上只有一个父表和一个子表。在嵌 套表和父/子表之间还存在一些显著的语法和语义差别，这一节就会详细介绍这些内容。

使用嵌套表有两种方法。一种方法是在 PL/SQL 代码中使用，用来扩展 PL/SQL 语言。另一种方法作为一种物理存储机制，持久地存储集合。我个人总是在 PL/SQL 中使用嵌套表，而从未将嵌套表用作持久存储机制。

在这一节中，我将简要地介绍创建、查询和修改嵌套表的语法。然后介绍一些实现细节，并说明关于 Oracle 如何存储嵌套表有哪些要点需要知道。

10.8.1 嵌套表语法

要创建一个有嵌套表的表相当简单，只是管理这个表的语法有点复杂。下面使用简单的 EMP 和 DEPT 表来说明。我们对用关系模式实现的这个小数据模型已经很熟悉了，如下：

```
ops$tkyte@ORA10GR1> create table dept
  2         (deptno number(2) primary key,
  3         dname varchar2(14),
  4         loc varchar2(13)
  5         );
Table created.
```

```
ops$tkyte@ORA10GR1> create table emp
  2          (empno number(4) primary key,
  3          ename varchar2(10),
  4          job varchar2(9),
  5          mgr number(4) references emp,
  6          hiredate date,
  7          sal number(7, 2),
  8          comm number(7, 2),
  9          deptno number(2) references dept
 10         );
Table created.
```

这里有主键和外键。下面建立与之对应的等价实现，不过这里 EMP 表实现为嵌套表：

```
ops$tkyte@ORA10GR1> create or replace type emp_type

  2     as object
  3     (empno number(4),
  4     ename varchar2(10),
  5     job varchar2(9),
  6     mgr number(4),
  7     hiredate date,
  8     sal number(7, 2),
  9     comm number(7, 2)
 10 );
 11 /
Type created.



ops$tkyte@ORA10GR1> create or replace type emp_tab_type

  2 as table of emp_type
  3 /
Type created.
```

要 创建一个带嵌套表的表，需要有一个嵌套表类型。以上代码创建了一个复杂的对象类型 EMP_TYPE，并创建了它的一个嵌套表类型 EMP_TAB_TYPE。在 PL/SQL 中，会像处理数组一样处理这种类型。在 SQL 中，这会导致创建一个物理的嵌套表。以下简单的 CREATE TABLE 语句使用了这个类型：

```
ops$tkyte@ORA10G> create table dept_and_emp

  2     (deptno number(2) primary key,
  3     dname varchar2(14),
  4     loc varchar2(13),
  5     emps emp_tab_type
  6 )
```

```
  7 nested table emps store as emps_nt;

Table created.

ops$tkyte@ORA10G> alter table emps_nt add constraint

  2     emps_empno_unique unique(empno)
  3 /
Table altered.
```

这 个 CREATE TABLE 语句中重要的是：其中包括列 EMPS（类型为 EMP_TAB_TYPE），还有相应的 NESTED TABLE EMPS STORE AS EMPS_NT。这样除了表 DEPT_AND_EMP 之外，还会创建一个真正的物理表 EMPS_NT，这个表与 DEPT_AND_EMP 是分开的。我们的 嵌套表的 EMPNO 列上直接加了一个约束，使 EMPNO 像在原来的关系模型中一样是惟一的。利用嵌套表无法实现前面完整的数据模型：关系模型中存在自引用 约束，倘若在嵌套表上增加同样的约束，则有：

```
ops$tkyte@ORA10G> alter table emps_nt add constraint mgr_fk

  2 foreign key(mgr) references emps_nt(empno);

alter table emps_nt add constraint mgr_fk
*
ERROR at line 1:

ORA-30730: referential constraint not allowed on nested table column
```

这是不行的。嵌套表不支持引用完整性约束，因为它们不能引用任何表，甚至它们自己。因此，现在先不管它。接下来，用现有的 EMP 和 DEPT 数据来填充这个表：

```
ops$tkyte@ORA10G> insert into dept_and_emp

  2     select dept.*,

  3          CAST( multiset( select empno, ename, job, mgr, hiredate, sal, comm

  4                  from SCOTT.EMP

  5                  where emp.deptno = dept.deptno ) AS emp_tab_type )

  6     from SCOTT.DEPT
  7 /
4 rows created.
```

这里有两点要注意：

q 只创建了“4”行。DEPT_AND_EMP 表中确实只有 4 行。没有独立地存在 14 个 EMP 行。

q 这 个语法开始有点奇怪了。CAST 和 MULTISET 是大多数人从来没有用过的语法。处理数据库中的对象关系组件时，你会看到很多奇怪的语法。 MULTISET 关

键字用于告诉 Oracle：这个子查询返回多行（SELECT 列表中的子查询先前限制为只返回一行）。CAST 用于指示 Oracle： 要把返回的结果集处理为一个集合类型，在这里，我们将 MULTISET 强制转换（CAST）为一个 EMP_TAB_TYPE。CAST 是一个通用的例程， 并不仅限于在集合中使用。例如，如果想从 EMP 中将 EMPNO 列获取为 VARCHAR2(20)而不是 NUMBER(4)类型，可以使用以下查询：SELECT CAST(EMPNO AS VARCHAR2(20)) E FROM EMP。

现在可以查询数据了。下面来看一行是什么样子：

```
ops$tkyte@ORA10G> select deptno, dname, loc, d.emps AS employees

  2 from dept_and_emp d
  3 where deptno = 10
  4 /


DEPTNO     DNAME          LOC            EMPLOYEES(EMPNO, ENAME, JOB,

---------  -------------  -------------  ------------------------------------------------
10         ACCOUNTING     NEW YORK EMP_TAB_TYPE(EMP_TYPE(7782,

                                         'CLARK', 'MANAGER', 7839, '0

                                         9-JUN-81', 2450, NULL), EMP_

                                         TYPE(7839, 'KING', 'PRESIDEN

                                         T', NULL, '17-NOV-81', 5000,

                                         NULL), EMP_TYPE(7934, 'MILL

                                          ER', 'CLERK', 7782, '23-JAN-

                                         82', 1300, NULL))
```

现 有数据都在这里，都放在一个利中。大多数应用都不能处理这个特殊的列，除非是专门针对对象关系特性编写的。例如，ODBC 没有办法处理嵌套表（JDBC、 OCI、Pro*C、PL/SQL 和大多数其他 API 和语言则可以）。针对这些情况，Oracle 提供了一种方法，可以取消集合的嵌套，把它当成一个关系 表来处理：

```
ops$tkyte@ORA10G> select d.deptno, d.dname, emp.*

  2 from dept_and_emp D, table(d.emps) emp
  3 /


DEPTNO     DNAME          EMPNO  ENAME  JOB              MGR  HIREDATE
    SAL       COMM
```

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

    10 ACCOUNTING      7782 CLARK     MANAGER       7839 09-JUN-81
  2450

    10 ACCOUNTING      7839 KING      PRESIDENT           17-NOV-81
  5000

    10 ACCOUNTING      7934 MILLER    CLERK         7782 23-JAN-82
  1300

    20 RESEARCH        7369 SMITH     CLERK         7902 17-DEC-80
   800

    20 RESEARCH        7566 JONES     MANAGER       7839 02-APR-81
  2975

    20 RESEARCH        7788 SCOTT     ANALYST       7566 09-DEC-82
  3000

    20 RESEARCH        7876 ADAMS     CLERK         7788 12-JAN-83
  1100

    20 RESEARCH        7902 FORD      ANALYST       7566 03-DEC-81
  3000

    30 SALES           7499 ALLEN     SALESMAN      7698 20-FEB-81
  1600        300

    30 SALES           7521 WARD      SALESMAN      7698 22-FEB-81
  1250        500

    30 SALES           7654 MARTIN    SALESMAN      7698 28-SEP-81
1250       1400

    30 SALES           7698 BLAKE     MANAGER       7839 01-MAY-81
  2850

    30 SALES           7844 TURNER
SALESMAN      7698 08-SEP-81       1500           0

    30 SALES           7900 JAMES     CLERK         7698 03-DEC-81
   950

14 rows selected.
```

可以把 EMPS 列强制转换为一个表，它会自然地为我们完成联结，这里不需要联结条件。实际上，由于我们的 EMP 类型根本没有 DEPTNO 列，所以无法明确地在哪个列上完成联结。这些杂事都由 Oracle 为我们做。

那么，怎么更新数据呢？假设我们想给部门 10 发$100 的奖金。可以如下编写代码：

```
ops$tkyte@ORA10G> update

  2     table( select emps

  3     from dept_and_emp

  4     where deptno = 10

  5     )

  6     set comm = 100

  7 /
3 rows updated.
```

前 面说过“每一行都有一个虚拟的表”，这句话在这里就得到了应验。在前面所示的 SELECT 谓词中，每行有一个表可能还不太明显：特别是前面没有联结之类的 工作，所有看上去有点“神奇”。不过，UPDATE 语句能很清楚地显示出每行有一个表。我们选择一个具体的表来更新（UPDATE）——这个表没有名字， 只是用一个查询来标识。如果使用的这个查询不是刚好选择（SELECT）一个表，我们就会得到以下错误：

```
ops$tkyte@ORA10G> update

  2     table( select emps

  3                 from dept_and_emp

  4                 where deptno = 1

  5     )
  6 set comm = 100
  7 /
update
*
ERROR at line 1:
ORA-22908: reference to NULL table value


ops$tkyte@ORA10G> update

  2     table( select emps

  3           from dept_and_emp
```

```
  4          where deptno > 1

  5    )

  6    set comm = 100

  7 /
table( select emps
*
ERROR at line 2:

ORA-01427: single-row subquery returns more than one row
```

如 果返回至少一行（一个嵌套表实例也没有），更新就会失败。正常情况下，更新 0 行是可以的，但在这里不行，它会返回一个错误，就好像我们的更新中漏写了表名 一样。如果返回了多行（不止一个嵌套表实例），更新也会失败。正常情况下，更新多行是完全可以的。但是这里显示出，Oracle 认为 DEPT_AND_EMP 表中的每一行指向另一个表，而不是像关系模型中那样指定另外一个行集。

这 就是嵌套表和父/子关系表之间的语义差别。在嵌套表模型中，每个父行有一个表。而关系模型中，每个父行有一个行集。这种差别使用嵌套表有时使用起来有些麻 烦。考虑我们使用的这个模型，它从单个部门的角度提供了一个很好的数据视图。如果我们想问“KIND 为哪个部门工作？”“有多少在职的会计”等待，这个模 型就很糟糕了。这些问题最好去问 EMP 关系表，而在嵌套表模型中，我们只能通过 DEPT 数据来访问 EMP 数据。总是必须联结，而无法单独地查询 EMP 数 据。在这方面，Oracle 没有提供公开支持的方法（也没有相关的文档说明），但是我们可以使用一个技巧来完成（关于这个技巧，稍后还会更多地介绍）。如 果需要更新 EMPS_NT 中的每一行，我们必须完成 4 个更新：分别更新 DEPT_AND_EMP 中的各行，从而更新与之关联的虚拟表。

更 新部门 10 的员工数据时，还有考虑一个问题，我们是在语义上更新 DEPT_AND_EMP 表中的 EMPS 列。要知道，尽管在物理上涉及两个表，但是语义上 只有一个表。即使我们没有更新部门表中的数据，包含所修改嵌套表的行也会被锁定，不允许其他会话更新。传统的父/子表关系中则不是这样。

正是由于这些原因，我不主张把嵌套表用作一种持久存储机制。作为子表，如果不用单独查询，这种情况实在是少见。在前面的例子中，EMP 表应该是一个强实体。它是独立的，所以需要单独查询。我发现一般都是这种情况，所以打算通过关系表上的视图来使用嵌套表。

既然我们了解了如何更新一个嵌套表实例，插入和删除就相当容易了。下面向嵌套表实例部门 10 增加一行，再从部门 20 删除一行：

```
ops$tkyte@ORA10G> insert into table

  2    ( select emps from dept_and_emp where deptno = 10 )

  3    values

  4    ( 1234, 'NewEmp', 'CLERK', 7782, sysdate, 1200, null );

1 row created.
```

```
ops$tkyte@ORA10G> delete from table
  2     ( select emps from dept_and_emp where deptno = 20 )
  3     where ename = 'SCOTT';
1 row deleted.

ops$tkyte@ORA10G> select d.dname, e.empno, ename
  2     from dept_and_emp d, table(d.emps) e
  3     where d.deptno in ( 10, 20 );

DNAME          EMPNO      ENAME
-------------  ---------  ----------
ACCOUNTING      7782      CLARK
ACCOUNTING      7839      KING
ACCOUNTING      7934      MILLER
RESEARCH        7369      SMITH
RESEARCH        7566      JONES
RESEARCH        7876      ADAMS
RESEARCH        7902      FORD
ACCOUNTING      1234      NewEmp
8 rows selected.
```

这就是查询和修改嵌套表的基本语法。你往往会发现，必须像我们刚才那样取消这些表的嵌套（特别是在查询中），才能使用这些嵌套表。一旦从概念上了解了“每行一个虚拟表”的概念，使用嵌套表就会容易得多。

前 面我说过，“总是必须联结；而无法单独地查询 EMP 数据”，但是然后我又告诫说“如果你确实需要，（利用一个技巧）这也是能办到的”。这种方法没有得到公 开支持，也没有相关的文档说明，所以只能把它作为最后一个杀手锏。如果你确实需要大批量地更新嵌套表（记住，要利用联结通过 DEPT 表来做到），此时这种 方法才能最好地发挥作用。Oracle 中有一个无文档说明的提示（只是简单地提了一下，而没有充分地说明）：NESTED_TABLE_GET_REFS，许多工具都使用了这个提示，如 EXP 和 IMP 就利用了这个提示来处理嵌套表。利用这种方法还可以查看嵌套表 物理结构的更多信息。使用这个提示，可以完成查询来得到一些“神奇”的结果。EXP（一个数据卸载工具）从嵌套表中抽取数据时就使用了以下查询：

```
ops$tkyte@ORA10G> SELECT /*+NESTED_TABLE_GET_REFS*/
  2 NESTED_TABLE_ID,SYS_NC_ROWINFO$
  3 FROM "OPS$TKYTE"."EMPS_NT"
```

```
  4 /


NESTED_TABLE_ID                          SYS_NC_ROWINFO$(EMPNO, EN

--------------------------------         ------------------------
F60DEEE0FF7D7BC1E030007F01001321 EMP_TYPE(7782, 'CLARK', '

                                         MANAGER', 7839, '09-JUN-8

                                         1', 2450, 100)

F60DEEE0FF7D7BC1E030007F01001321 EMP_TYPE(7839, 'KING', 'P

                                         RESIDENT', NULL, '17-NOV-

                                         81', 5000, 100) ...
```

但是，如果你描述这个嵌套表，会有点奇怪：

```
ops$tkyte@ORA10G> desc emps_nt

Name                          Null?            Type

----------------------------- --------         --------------------
EMPNO                                          NUMBER(4)
ENAME                                          VARCHAR2(10)
JOB                                            VARCHAR2(9)
MGR                                            NUMBER(4)
HIREDATE                                       DATE
SAL                                            NUMBER(7,2)
COMM                                           NUMBER(7,2)
```

前 面查询的两列居然没有出现。它们是嵌套表隐藏实现的一部分。NESTED_TABLE_ID 实际上是父表 DEPT_AND_EMP 的一个外键。 DEPT_AND_EMP 中确实有一个隐藏列，用于联结至 EMPS_NT。SYS_NC_ROWINFO$“列”是一个神奇的列；它更应该算是一个函数而 不是一个列。这里的嵌套表实际上是一个对象表（由一个对象类型组成），而 SYS_NC_ROWINFO$正是 Oracle 将行引用为对象所采用的内部方 法，而并非引用行的各个标量列。在底层，Oracle 会用系统生成的主键和外键实现一个父/子表。如果更进一步挖掘，可以查询“真正”的数据字典来查看 DEPT_AND_EMP 表中的所有列：

```
sys@ORA10G> select name

  2     from sys.col$

  3     where obj# = ( select object_id

  4          from dba_objects
```

```
  5          where object_name = 'DEPT_AND_EMP'

  6                and owner = 'OPS$TKYTE' )

  7 /

NAME
------------------------------
DEPTNO
DNAME
EMPS
LOC
SYS_NC0000400005$
```

从嵌套表中选出这一列，我们会看到下面的结果：

```
ops$tkyte@ORA10G> select SYS_NC0000400005$ from dept_and_emp;



SYS_NC0000400005$
--------------------------------
F60DEEE0FF887BC1E030007F01001321
F60DEEE0FF897BC1E030007F01001321
F60DEEE0FF8A7BC1E030007F01001321
F60DEEE0FF8B7BC1E030007F01001321
```

这 个列名看上去很古怪（SYS_NC0000400005$），这是放在 DEPT_AND_EMP 表中的系统生成的键。如果更深层次地挖掘，会发现 Oracle 已经在这个列上放上了惟一一个索引。不过，遗憾的是，它没有对 EMP_NT 中的 NESTED_TABLE_ID 加索引。这个列确实需要加索 引，因为我们总是要从 DEPT_AND_EMP 联结到 EMPS_NT。如果像刚才那样使用默认值，必须记住关于嵌套表的一个要点：一定要对嵌套表中的 NESTED_TABLE_ID 加索引!

不过现在我有些离题了，我要说的是如何把嵌套表当成一个真正的表来进行处理。NESTED_TABLE_GET_REFS 提示就为我们做了这个工作。可以使用如下的提示：

```
ops$tkyte@ORA10G> select /*+ nested_table_get_refs */ empno, ename

  2 from emps_nt where ename like '%A%';

EMPNO   ENAME
----------      ----------
7782        CLARK
7876        ADAMS
7499        ALLEN
7521        WARD
7654        MARTIN
7698        BLAKE
7900        JAMES
7 rows selected.
```

```
ops$tkyte@ORA10G> update /*+ nested_table_get_refs */ emps_nt
  2 set ename = initcap(ename);
14 rows updated.

ops$tkyte@ORA10G> select /*+ nested_table_get_refs */ empno, ename
  2 from emps_nt where ename like '%a%';

EMPNO   ENAME
---------     ----------
7782        Clark
7876        Adams
7521        Ward
7654        Martin
7698        Blake
7900        James
6 rows selected.
```

再 次重申，这个特性没有相关的文档说明，没有得到公开支持。它是一个保障 EXP 和 IMP 工作的特定功能，而且只有在 EXP 和 IMP 环境中才能保证这种方法一 定可行。你自己使用时会有风险，而且千万不要用在生成代码中。实际上，如果你发现确实需要使用这个特性，那么从定义来讲，这说明根本就不应该使用嵌套表！ 这个结构对你完全不适合。可以用这种方法对数据完成一次性修正，或者你对嵌套表很好奇，可以通过这个技巧来看看嵌套表里有什么。要报告嵌套表中的数据，公 开支持的方法是消除嵌套，如下：

```
ops$tkyte@ORA10G> select d.deptno, d.dname, emp.*
  2 from dept_and_emp D, table(d.emps) emp
  3 /
```

查询和生产代码中应该使用这种方法。

10.8.2 嵌套表存储

我们已经了解了嵌套表结构的一些存储问题。在这一节中，我们将输入地分析 Oracle 默认创建的结构，并说明我们对这种结构能有怎样的控制。还是用前面的 CREATE 语句：

```
ops$tkyte@ORA10G> create table dept_and_emp
  2     (deptno number(2) primary key,
  3     dname varchar2(14),
  4     loc varchar2(13),
```

```
  5     emps emp_tab_type

  6     )

  7     nested table emps store as emps_nt;

Table created.


ops$tkyte@ORA10G> alter table emps_nt add constraint emps_empno_unique

  2 unique(empno)
  3 /
Table altered.
```

我们知道 Oracle 实际上会创建一个图 10-11 所示的结构。

图 10-11 嵌套表的物理实现

这 个代码创建了两个实际的表。这里确实会创建我们请求创建的表，但是它有额外的一个隐藏列（默认情况下，对于表中的每一个嵌套表列，都会有一个额外的隐藏 列）。它还在这个隐藏列上创建了一个惟一约束。Oracle 为我们创建了嵌套表 EMPS_NT。这个表有两个隐藏列，其中的 SYS_NC_ROWINFO $列并不真正算是一个列，而应该是一个虚拟列，它会把所有标量元素返回为一个对象。另一个隐藏列是名为 NESTED_TABLE_ID 的外键，通过这个外 键可以联结回父表。注意这个列上没有索引。最后，Oracle 在 DEPT_AND_EMP 表的 DEPTNO 列上增加了一个索引，以保证主键。所以，我们本 来只请求创建一个表，得到的却远不止一个表。如果查看这个表，与创建父/子关系时所看到的情况非常相似，但是你要使用 DEPTNO 上的现有主键作为 EMPS_NT 的外键，而不是生成一个代理键 RAW(16)。

如果查看嵌套表示例的 DBMS_METADATA.GET_DDL 转储信息，可以看到以下内容：

```
ops$tkyte@ORA10G> begin

  2           dbms_metadata.set_transform_param
```

```
  3          ( DBMS_METADATA.SESSION_TRANSFORM, 'STORAGE', false );

  4      end;
  5      /
PL/SQL procedure successfully completed.


ops$tkyte@ORA10G> select dbms_metadata.get_ddl( 'TABLE', 'DEPT_AND_EMP' ) from
dual;


DBMS_METADATA.GET_DDL('TABLE','DEPT_AND_EMP')
--------------------------------------------------------------------------------

CREATE TABLE "OPS$TKYTE"."DEPT_AND_EMP"

(         "DEPTNO" NUMBER(2,0),

         "DNAME" VARCHAR2(14),

         "LOC" VARCHAR2(13),

         "EMPS" "OPS$TKYTE"."EMP_TAB_TYPE" ,

         PRIMARY KEY ("DEPTNO")

USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
TABLESPACE "USERS" ENABLE

) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING

TABLESPACE "USERS"

NESTED TABLE "EMPS" STORE AS "EMPS_NT"

(PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING

TABLESPACE "USERS" ) RETURN AS VALUE
```

到此为止，只有一个新内容，即 RETURN AS VALUE。 这个选项用于描述如何向客户应用返回嵌套表。默认情况下，Oracle 会按值把嵌套表返回给客户：具体数据会随各行传输。这个选项也可以设置为 RETURN AS LOCATOR，这说明客户会得到指向数据的一个指针，而不是数据本身。当且仅当客户对这个指针解除引用（dereference）时，才会把数据传输给 客户。因此，如果你相信客户通常不会查看对应各个父行的嵌套表（不会查看这些嵌套表中的行），就可以返回一个 locator 而不是值，这样可以节省网络往 返通信开销。例如，

如果你的客户应用要显示部门列表，当用户双击一个部门时，客户应用会显示出员工信息，你就可以考虑使用 locator。这是因为通常都 不会查看详细信息，查看详细信息的情况是例外，而不是一般情况。

那 么，我们还能对嵌套表做些什么呢？首先，NESTED_TABLE_ID 列必须建立索引。因为我们总是从父表联结到子表来访问嵌套表，我们确实需要这个索 引。可以使用 CREATE INDEX 对该列建立索引，但是最好的解决方案是使用一个 IOT 来存储嵌套表。嵌套表也是一个适用 IOT 的绝好例子。它会按 NESTED_TABLE_ID 将子行物理地共同存储在一块（所以用最少的物理 I/O 就能完成表的获取），这样就不必在 RAW(16)列上建立冗余的索 引。再前进一步，由于 NESTED_TABLE_ID 就是 IOT 主键的第一列，还应该加入索引键压缩来避免冗余的 NESTED_TABLE_ID（否则会 重复存储）。另外，我们还可以在 CREATE TABLE 命令中加入 EMPNO 列的 UNIQUE 和 NOT NULL 约束。因此，对于前面的 CREATE TABLE，可以稍作修改，如下所示：

```
ops$tkyte@ORA10G> CREATE TABLE "OPS$TKYTE"."DEPT_AND_EMP"
  2    ("DEPTNO" NUMBER(2, 0),
  3    "DNAME" VARCHAR2(14),
  4    "LOC" VARCHAR2(13),
  5    "EMPS" "EMP_TAB_TYPE")
  6    PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
  7    STORAGE(INITIAL 131072 NEXT 131072
  8          MINEXTENTS 1 MAXEXTENTS 4096
  9          PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
 10          BUFFER_POOL DEFAULT)
 11 TABLESPACE "USERS"
 12 NESTED TABLE "EMPS"
 13          STORE AS "EMPS_NT"
 14          ( (empno NOT NULL, unique (empno), primary key(nested_table_id,empno))
 15                organization index compress 1 )
 16          RETURN AS VALUE
 17 /
```

```
Table created.
```

现在可以得到以下的一组对象。这里没有一个传统表 EMP_NT，而是一个 IOT EMPS_NT，由图 10-12 中表上的索引结构指示。

图 10-12 嵌套表实现为 IOT

如果 EMPS_NT 是一个使用压缩的 IOT，它比原来默认嵌套表占用的存储空间更少，而且其中有我们非常需要的索引。

10.8.3 嵌套表小结

我自己并不把嵌套表用作一种持久存储机制，原因如下：

- 这样会增加不必要的 RAW(16)列存储开销。父表和子表都有这个额外的列。父表对于其中的各个嵌套表列都有一个额外的 16 字节 RAW 字段。由于父表通常已经有一个主键（在我们这个例子中就是 DEPTNO），所以完全可以在子表中使用这个键，而不必使用一个系统生成的键。
- 这会在父表上增加另一个惟一约束（相应地带领不必要的开销），而父表中通常已经有一个惟一约束。
- 如果不使用 NESTED_TABLE_GET_REFS（这个方法未得到公开支持），嵌套表本身使用起来并不 2.如果是查询，可以通过消除嵌套来访问嵌套表，但是如果是大批量更新，则无法简单地消除嵌套。在实际中，表往往都会“自己”查询自己，我还没有见过这方面的例外。

不 过作为一个编程构造，我确实大量使用了嵌套表，并把嵌套表用于视图中。我认为这才是嵌套表应有的位置。作为一个存储机制，我更倾向于我自己创建父/子表。 创建了父/子表之后，实际上，我们可以再创建一个视图，使之看上去就好像我们有一个真正的嵌套表一样。也就是说，这样做可以得到嵌套表构造的所有好处，而 不会引入嵌套表的开销。

如 果你确实把嵌套表用作一个存储机制，一定要保证将嵌套表建立为一个 IOT，以避免 NESTED_TABLE_ID 上索引的开销以及嵌套表本身的开销。请参 考前面有关 IOT 的一节，了解基于溢出段和其他选项来建立 IOT 时有哪些建议。如果你没有使用 IOT，则要确保在嵌套表的 NESTED_TABLE_ID 列上创建一个索引，来避免为查找子行而执行全表扫描。

10.9 临时表

临 时表（Temporary table）用于保存事务或会话期间的中间结果集。临时表中保存的数据只对当前会话可见，所有会话都看不到其他会话的数据；即使当前会话已经提交（COMMIT）了数据，别的会话也看不到它的数据。对于临时表，不存在多用户并发问题，因为一个会话不会因为使用一个临时表而阻塞另一个会话。即使我们 “锁住”了临时表，也不会妨碍其他会话使用它们自己的临时表。我们在第 9 章了解到，临时表比常规表生成的 redo 少得多。不过，由于临时表必须为其中包含 的数据生成 undo 信息，所以也会生成一定的 redo。UPDATE 和 DELETE 会生成最多的 undo；INSERT 和 SELECT 生成的 undo 最 少。

临 时表会从当前登录用户的临时表空间分配存储空间，或者如果从一个定义者权限（definer right）过程访问临时表，就会使用该过程所有者的临时表空间。全局临时表实际上是表本身的一个模板。创建临时表的动作不涉及存储空间分配；不会为此分 配初始（INITIAL）区段，这与常规表有所不同。对于临时表，运行时当一个会话第一次在临时表中放入数据时，才会为该会话创建一个临时段。由于每个会 话会得到其自己的临时段（而不是一个现有段的一个区段），每个用户可能在不同的表空间为其临时表分配空间。USER1 的临时表空间可能设置为 TEMP1， 因此他的临时表会从这个表空间分配。USER2 可能把 TEMP2 作为其临时表空间，他的临时表就会从那里分配。

Oracle 的 临时表与其他关系数据库中的临时表类似，这样区别只是：Oracle 的临时表是“静态”定义的。每个数据库只创建一次临时表，而不是为数据库中的每个存储 过程都创建一次。在 Oracle 中，临时表一定存在，它们作为对象放在数据字典中，但是在会话向临时表中放入数据之前，临时表看上去总是空。由于临时表是 静态定义的，所以你能创建引用临时表的视图，还可以创建存储过程使用静态 SQL 来引用临时表，等等。临时表可以是基于会话的（临时表中的数据可以跨提交存 在，即提交之前仍然存在，但是断开连接后再连接后再连接时数据就没有了），也可以是基于事务的（提交之后数据就消失）。下面这个例子显示了这两种不同的临 时表。我使用 SCOTT.EMP 表作为一个模板：

```
ops$tkyte@ORA10G> create global temporary table temp_table_session

  2    on commit preserve rows

  3    as

  4    select * from scott.emp where 1=0

  5    /
Table created.
```

ON COMMIT PRESERVE ROWS 子句使得这是一个基于会话的临时表。在我的会话断开连接之前，或者我通过一个 DELETE 或 TRUNCATE 物理地删除行之前，这些行会一直存在于这个临时表中。只有我的会话能看到这些行；即使我已经提交（COMMIT），其他会话也无法看到“我的”行。

```
ops$tkyte@ORA10G> create global temporary table temp_table_transaction

  2    on commit delete rows

  3    as
```

```
  4   select * from scott.emp where 1=0
  5   /
Table created.
```

ON COMMIT DELETE ROWS 子句使得这是一个基于事务的临时表。我的会话提交时，临时表中的行就不见了。只需把分配给这个表的临时区段交回，这些行就会消失，在临时表的自动清除过程中不存在开销。

下面来看这两种类型之间的区别：

```
ops$tkyte@ORA10G> insert into temp_table_session select * from scott.emp;

14 rows created.


ops$tkyte@ORA10G> insert into temp_table_transaction select * from scott.emp;

14 rows created.
```

我们在各个 TEMP 表中方便放了 14 行，以下显示出我们可以“看到”这些行：

```
ops$tkyte@ORA10G> select session_cnt, transaction_cnt
  2   from ( select count(*) session_cnt from temp_table_session ),
  3        ( select count(*) transaction_cnt from temp_table_transaction );

 SESSION_CNT   TRANSACTION_CNT
-------------  -----------------
          14                14

ops$tkyte@ORA10G> commit;
```

由于我们已经提交，所以仍可以看到基于会话的临时表行，但是看不到基于事务的临时表行：

```
ops$tkyte@ORA10G> select session_cnt, transaction_cnt
  2   from ( select count(*) session_cnt from temp_table_session ),
  3        ( select count(*) transaction_cnt from temp_table_transaction );

 SESSION_CNT   TRANSACTION_CNT
   -----------        ---------------
          14                 0

ops$tkyte@ORA10G>
```

```
ops$tkyte@ORA10G> disconnect

Disconnected from Oracle Database 10g Enterprise Edition Release 10.1.0.3.0

With the Partitioning, OLAP and Data Mining options

ops$tkyte@ORA10G> connect /

Connected.
```

由于此时已经开始了一个新的会话，所以两个表中的行都看不到了：

```
ops$tkyte@ORA10G> select session_cnt, transaction_cnt

  2     from ( select count(*) session_cnt from temp_table_session ),

  3          ( select count(*) transaction_cnt from temp_table_transaction );


 SESSION_CNT      TRANSACTION_CNT
 -----------      ---------------
           0                    0
```

如果你曾在 SQL Server 和/或 Sybase 中用过临时表，现在所要考虑的主要问题是：不应该执行 SELECT X, Y, Z INTO #TEMP FROM SOME_TABLE 来动态创建和填充一个临时表，而应该：

- 将所有全局临时表只创建一次，作为应用安装的一部分，就像是创建永久表一样。
- 在你的过程中，只需执行 INSERT INTO TEMP(X, Y, Z) SELECT X, Y, Z FROM SOME_TABLE。

归根结底，这里的目标是：不要在运行时在你的存储过程中创建表。这不是 Oracle 中使用临时表的正确做法。DDL 是一种代价昂贵的操作：你要全力避免在运行时执行这种操作。一个应用的临时表应该在应用安装期间创建，绝对不要在运行时创建。

临时表可以有永久表的许多属性。它们可以有触发器、检查约束、索引等。但永久表的某些特性在临时表中并不支持，这包括：

- 不能有引用完整性约束。临时表不能作为外键的目标，也不能在临时表中定义外键。
- 不能有 NESTED TABLE 类型的列。在 Oracle 9i 及以前版本中，VARRAY 类型的列也不允许；不过 Oracle 10g 中去掉了这个限制。
- 不能是 IOT。
- 不能在任何类型的聚簇中。
- 不能分区。
- 不能通过 ANALYZE 表命令生成统计信息。

在 所有数据库中，临时表的缺点之一是优化器不能正常地得到临时表的真实统计。使用基于代价的优化器（cost-based optimizer，CBO）时，有效的统计对于优化器的成败至关重要。如果没有统计信息，优化器就只能对数据的分布、数据量以及索引的选择性作出猜测。如果这些猜测是错的，为查询生成的查询计划（大量使用临时表）可能就不是最优的。在许多情况下，正确的解决方案是根本不使用临时表，而是使用一个 INLINE VIEW（要看 INLINE VIEW 的例子，可以查看前面运行的 SELECT，它就有两个内联视图）。采用这种方式，Oracle 可以访问一个表的所有相关统计信息，而且得出一个最 优计划。

我 经常发现，人们之所以使用临时表，是因为他们在其他数据库中了解到一个查询中联结太多的表是一件“不好的事情”。但在 Oracle 开发中，必须把这个知识 忘掉。不要想着你比优化器要聪明，把本来一个查询分解成 3 个或 4 个查询，将其子结果存储在临时表中，然后再合并这些临时表；正确的做法是应该编写一个查 询，直接回答最初的问题。在一个查询中引用多个表是可以的；Oracle 中在这个方面不需要临时表的帮助。

不 过在其他情况下，可以在进程中使用临时表，这是一种正确的做法。例如，我曾经编写过一个 PALM 同步应用程序，将 Palm Pilot 上的日期簿与 Oracle 中存储的日历信息同步。Palm 会为我提供自最后一次热同步以来修改的所有记录的列表，我必须取得这些记录，把它们与 数据库中的当前数据相比较，更新数据库记录，然后生成一个修改列表，应用到 Palm。这是一个展示临时表用处的绝好例子。我使用一个临时表在数据库中存储 Palm 上所做的修改。然后运行一个存储过程，它将 Palm 生成的修改与当前的永久表（非常大）相比较，发现需要对 Oracle 数据做哪些修改，然后找出 Oracle 数据库中的哪些修改需要再应用到 Palm 上。我必须对这个数据做两趟处理。首先，要发现仅在 Palm 上修改的记录，并在 Oracle 中做相应 的修改。接下来，要发现自最后一次同步和修改以来 Palm 和数据库中都经过修改的所有记录。如何发现仅在数据库中经过修改的所有记录。并将其修改放在临时 表中。最后，Palm 同步应用程序从临时表拉出这些修改，把它们应用于 Palm 设备本身，断开连接时，临时表会消失。

不 过，我遇到的问题是，由于会分析永久表，所以使用了 CBO。但是临时表上没有统计信息（尽管可以分析临时表，但不会收集统计信息），因此 CBO 会对它做出 很多“猜测”。作为一名开发人员，我知道可能的平均行数、数据的分布、查询选择的列等。我需要一种方法来告诉优化器这些更准确的猜测。可以有 3 中种方法向 优化器提供关于全局临时表的统计信息。一种方法是通过动态采样（只是 Oracle9i Release 2 及以上版本中新增的特性），另一种方法是使用 DBMS_STATS 包，它有两种做法。下面首先来看动态采样。

动 态采样（dynamic sampling）是优化器的一种功能，硬解析一个查询时，会扫描数据库中的段（采样），收集有用的统计信息，来完成这个特定查询的优化。这与硬解析期间完成一个“缩型收集统计”命令很类似。Oracle 10g 中大量使用了动态采样，因为默认设置已经从 level 1 提升到 level 2，采用 level 2， 优化器在结算查询计划之前，会对优化器处理的查询中引用的所有未分析的对象完成动态采样。9i Release2 中则设置为 level 1，所以动态采样的使用少得多。在 Oracle9i Release 2 中可以使用一个 ALTER SESSION|SYSTEM 命令，从而能有 Oracle 10g 默认行为，或者可以使用动态采样提示，如下：

```
ops$tkyte@ORA9IR2> create global temporary table gtt
  2    as
  3    select * from scott.emp where 1=0;
```

```
Table created.


ops$tkyte@ORA9IR2> insert into gtt select * from scott.emp;

14 rows created.


ops$tkyte@ORA9IR2> set autotrace traceonly explain

ops$tkyte@ORA9IR2> select /*+ first_rows */ * from gtt;


Execution Plan
----------------------------------------------------------

0       SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=17
Card=8168 Bytes...

1    0       TABLE ACCESS (FULL) OF 'GTT' (Cost=17 Card=8168
Bytes=710616)



ops$tkyte@ORA9IR2> select /*+ first_rows dynamic_sampling(gtt 2) */ * from gtt;


Execution Plan
----------------------------------------------------------

0       SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=17 Card=14
Bytes=1218)

1    0       TABLE ACCESS (FULL) OF 'GTT' (Cost=17 Card=14 Bytes=1218)


ops$tkyte@ORA9IR2> set autotrace off
```

在 此，我们在这个查询中把表 GTT 的动态采样设置为 level 2.在此之前，优化器猜测会从表 GTT 返回 8,168 行。通过使用动态采样，估计的基数会与实际更为接近（这会得到总体上更好的查询计划）。使用 level 2 设置，优化器会很快地扫描表，对表的真实大小得出更实际的估计。在 Oracle 10g 中，这应该不成问题，因为默认就会发生动态采样：

```
ops$tkyte@ORA10G> create global temporary table gtt

  2 as
  3 select * from scott.emp where 1=0;
```

```
Table created.


ops$tkyte@ORA10G> insert into gtt select * from scott.emp;

14 rows created.


ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> select * from gtt;


Execution Plan
----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=14 Bytes=1218)

1 0           TABLE ACCESS (FULL) OF 'GTT' (TABLE (TEMP)) (Cost=2 Card=14
Bytes=1218)

ops$tkyte@ORA10G> set autotrace off
```

在此不必要求，就能得到正确的基数。不过，动态采样不是免费的，由于必须在查询解析时完成，所以存在相当大的代价。如果能提前收集适当的代表性统计信息，就可以避免在硬解析时执行动态采样。为此可以使用 DBMS_STATS。

使 用 DBMS_STATS 收集代表性统计信息有 3 种方法。第一种方法是利用 GATHER_SCHEMA_STATS 或 GATHER_DATABASE_STATS 调用来使用 DBMS_STATS。这些过程允许你传入一个参数 GATHER_TEMP，这是一个布尔值，默认 为 FALSE。设置为 TRUE 时，所有 ON COMMIT PRESERVE ROWS 全局临时表都会收集和存储统计信息（这个技术在 ON COMMIT DELETE ROWS 表上不可行）。考虑以下情况（注意这在一个空模式中完成：除了你创建的对象之外没有其他对象）：

```
ops$tkyte@ORA10G> create table emp as select * from scott.emp;

Table created.


ops$tkyte@ORA10G> create global temporary table gtt1 ( x number )

2 on commit preserve rows;
Table created.


ops$tkyte@ORA10G> create global temporary table gtt2 ( x number )

2 on commit delete rows;
Table created.
```

```
ops$tkyte@ORA10G> insert into gtt1 select user_id from all_users;

38 rows created.

ops$tkyte@ORA10G> insert into gtt2 select user_id from all_users;

38 rows created.

ops$tkyte@ORA10G> exec dbms_stats.gather_schema_stats( user );

PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select table_name, last_analyzed, num_rows from user_tables;

TABLE_NAME                     LAST_ANAL     NUM_ROWS

------------------------------ ---------     ----------

EMP                            01-MAY-05             14
GTT1
GTT2
```

可以看到，在这种情况下，只会分析 EMP 表：两个全局临时表将被忽略。可以如下调用 GATHER_SCHEMA_STATS（带 GATHER_TEMP => TRUE）来改变这种行为：

```
ops$tkyte@ORA10G> insert into gtt2 select user_id from all_users;

38 rows created.

ops$tkyte@ORA10G> exec dbms_stats.gather_schema_stats( user, gather_temp=>TRUE );

PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select table_name, last_analyzed, num_rows from user_tables;

TABLE_NAME                     LAST_ANAL     NUM_ROWS

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

```
EMP                    01-MAY-05              14

GTT1                   01-MAY-05              38

GTT2                   01-MAY-05               0
```

注意，ON COMMIT PRESERVE ROWS 表 会有正确的统计，但是 ON COMMIT DELETE ROWS 表没有。DBMS_STATS 将提交，而这会擦除 ON COMMIT DELETE ROWS 表中的所有信息。不过，要注意，现在 GTT2 确实有统计信息了，这本身并不好，因为统计信息太离谱了！运行时表居然只有 0 行，这实在是让人怀疑。 所以，如果使用这种方法，要注意两点：

- 要保证在收集统计信息的会话中用代表性数据填充全局临时表。如果做不到，在 DBMS_STATS 看来它们就是空的。
- 如果有 ON COMMIT DELETE ROWS 全局临时表，就不应该使用这种方法，因为这样会收集到不正确的值。

对于 ON COMMIT PRESERVE ROWS 全局临时表，还可以采用第二种技术：直接在表上使用 GATHER_TABLE_STATS。你要像我们刚才那样填充全局临时表，然后在这个全局临时表上执行 GATHER_TABLE_STATS。注意还是像前面一样，对于 ON COMMIT DELETE ROWS 全局临时表，这种技术还是不能用，同样是因为存在前面所述的问题。

使用 DBMS_STATS 的最后一种技术是通过一个手动过程用临时表的代表性统计信息填充数据字典。例如，如果平均来讲临时表中的行数是 500，而且行的平均大小是 100 字节，块数为 7，则只需如下使用 DBMS_STATS：

```
ops$tkyte@ORA10G> create global temporary table t ( x int, y varchar2(100) );

Table created.

ops$tkyte@ORA10G> begin

  2          dbms_stats.set_table_stats( ownname => USER,

  3          tabname => 'T',
  4          numrows => 500,
  5          numblks => 7,
  6          avgrlen => 100 );
  7 end;
  8 /

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select table_name, num_rows, blocks, avg_row_len

  2 from user_tables
```

```
  3 where table_name = 'T';

TABLE_NAME                     NUM_ROWS     BLOCKS AVG_ROW_LEN
------------------------------ ---------- ---------- -----------
T                                     500          7         100
```

现在，优化器不会使用它自己的最优猜测，而会使用我们给出的最优猜测。

临时表小结

如 果应用中需要临时存储一个行集由其他表处理（可能对应一个会话，也可能对应一个事务），临时表就很有用。不要把临时表作为一个分解大查询的方法，即拿到一 个大查询，把它“分解”为几个较小的结果集，然后再把这些结果集合并在一起（这看来是其他数据库中最常见的临时表用法）。实际上，你会发现，在几乎所有的 情况下。Oracle 中如果将一个查询分解为较小的临时表查询，与原来的一个查询相比，只会执行得更慢。我就经常看到人们这样做，如果有可能把对临时表的 一系列 INSERT 重写为一个大查询（SELECT），所得到的单个查询会比原来的多步过程快得多。

临 时表会生成少量的 redo，但是确实还是会生成 redo，而且没有办法避免。这些 redo 是为回滚数据生成的，而且在最典型的情况下，可以忽略不计。如果 只是对临时表执行 INSERT 和 SELETE，生成的 redo 量几乎注意不到。只有对临时表执行大量 DELETE 和 UPDATE 时，才会看到生成大量的 redo。

如果精心设计，可以在临时表上生成 CBO 使用的统计信息；不过，可以使用 DBMS_STATS 包对临时表上的统计给出更好的猜测，或者由优化器使用动态采样在硬解析时动态收集。

10.10 对象表

我们已经看到了一个不完整的对象表（带嵌套表）例子。对象表（object table）是基于一个 TYPE 创建的表，而不是作为一个列集合。正常情况下，CREATE TABLE 可能如下所示：

```
create table t ( x int, y date, z varchar2(25) );
```

对象表创建语句则如下所示：

```
create table t of Some_Type;
```

T 的属性（列）由 SOME_TYPE 的定义得出。下面来简单地看一个例子，这里用到了几种类型，然后查看所得到的数据结构：

```
ops$tkyte@ORA10G> create or replace type address_type

  2 as object
  3         ( city varchar2(30),
  4         street varchar2(30),
  5         state varchar2(2),
  6         zip number
  7 )
  8 /
Type created.
```

```
ops$tkyte@ORA10G> create or replace type person_type

  2 as object
  3       ( name varchar2(30),
  4       dob date,
  5       home_address address_type,
  6       work_address address_type
  7 )
  8 /
Type created.


ops$tkyte@ORA10G> create table people of person_type

  2 /
Table created.


ops$tkyte@ORA10G> desc people

Name                                    Null?    Type

----------------------------------------- --------  ----------------------------

NAME                                              VARCHAR2(30)
DOB                                               DATE
HOME_ADDRESS                                      ADDRESS_TYPE
WORK_ADDRESS                                      ADDRESS_TYPE
```

就这么多。我们创建了一些类型定义，接下来可以创建这种类型的表。这个表有 4 列，表示所创建的 PERSON_TYPE 的 4 个属性。现在我们可以在这个对象表上执行 DML 来创建和查询数据了：

```
ops$tkyte@ORA10G> insert into people values ( 'Tom', '15-mar-1965',

  2       address_type( 'Reston', '123 Main Street', 'Va', '45678' ),

  3       address_type( 'Redwood', '1 Oracle Way', 'Ca', '23456' ) );

1 row created.


ops$tkyte@ORA10G> select * from people;

NAME                 DOB       HOME_ADDRESS(CITY, S
```

```
WORK_ADDRESS(CITY, S

-----      ---------          --------------------        --------------------

Tom      15-MAR-65 ADDRESS_TYPE('Reston ADDRESS_TYPE('Redwoo

                                ', '123 Main Street' d', '1 Oracle Way',

                                , 'Va', 45678) 'Ca', 23456)



ops$tkyte@ORA10G> select name, p.home_address.city from people p;

NAME      HOME_ADDRESS.CITY
-----     ------------------------------
Tom       Reston
```

从 现在开始，可以看到处理对象类型必需的一些对象语法了。例如，在 INSERT 语句中，必须把 HOME_ADDRESS 和 WORK_ADDRESS 用一个 CAST 包装起来。我们将标量值强制转换为一个 ADDRESS_TYPE。对此也可以用另一种说法来解释，我们使用 ADDRESS_TYPE 对象的默认构 造函数为这一行创建了一个 ADDRESS_TYPE 实例。

现 在，从外部来看这个表，表中只有 4 个列。由于我们已经了解到嵌套表内部有神奇的隐藏列，所以可以猜测到，对象表可能也不会这么简单，也许还会做其他的事 情。Oracle 把所有对象关系数据都存储在普通的关系表中，最终还是存储在行和列中。如果挖掘“真正”的数据字典，可以看到这个表实际上是什么样子：

```
ops$tkyte@ORA10G> select name, segcollength

  2 from sys.col$

  3 where obj# = ( select object_id

  4                from user_objects

  5                where object_name = 'PEOPLE' )

  6 /
NAME                           SEGCOLLENGTH

------------------------       ------------

SYS_NC_OID$                              16
SYS_NC_ROWINFO$                           1
NAME                                     30
DOB                                       7
HOME_ADDRESS                              1
SYS_NC00006$                             30
```

```
SYS_NC00007$                    30
SYS_NC00008$                     2
SYS_NC00009$                    22
WORK_ADDRESS                     1
SYS_NC00011$                    30
SYS_NC00012$                    30
SYS_NC00013$                     2
SYS_NC00014$                    22
14 rows selected.
```

看上去这与 DESCRIBE 告诉我们的结果完全不同。显然，这个表中有 14 列，而不是 4 列。在这个例子中，这些列分别是：

- SYS_NC_OID$：这是表中系统生成的对象 ID。这是一个惟一的 RAW(16)列。这个列上有一个惟一约束，而且在这个列上还创建了一个相应的惟一索引。

- SYS_NC_ROWINFO$：这是嵌套表中已经研究过的那个“神奇”函数。如果从表中选择这个列，它会把整行作为一列返回：

```
ops$tkyte@ORA10G> select sys_nc_rowinfo$ from people;



SYS_NC_ROWINFO$(NAME, DOB, HOME_ADDRESS(CITY,STREET,STATE,ZIP),
WORK_ADDRESS

--------------------------------------------------------------------------------

PERSON_TYPE('Tom', '15-MAR-65', ADDRESS_TYPE('Reston', '123 Main Street',

'Va', 45678), ADDRESS_TYPE('Redwood', '1 Oracle Way', 'Ca', 23456))
```

- NAME.DOB：这些是对象表的标量属性。与我们预期的一样，它们存储为常规的列。

- HOME_ADDRESS，WORK_ADDRESS：这些也是“神奇的”函数。它们把所表示的列集返回为一个对象。这些不占用实际空间，只是为实际指示 NULL 或 NOT NULL。

- SYS_NCnnnnn$： 这些是嵌入的对象类型的标量实现。由于 PERSON_TYPE 中嵌入了 ADDRESS_TYPE，Oracle 需要留出空间将它们存储在适当类型的列中。 系统生成的名字是必要的，因为列名必须惟一，我们很可能多次使用同一个对象类型（像这里一样），如果不采用系统生成的名字，最后就完全有可能重复地使用相 同的名字，如有两个 ZIP 列。

所 以，就像嵌套表一样，这里完成了很多工作。首先增加了一个 16 字节的伪主键，而且为我们创建了一个索引。关于如何为对象指定对象标识符的值，默认行为是可 以修改的，稍后将介绍。首先来看生成这个表的完整 SQL。同样，这是使用 EXP/IMP 生成的，因为我想轻松地看到依赖对象，包括重建这个对象所需的全部 SQL。这是如下得到的：

```
[tkyte@localhost tkyte]$ exp userid=/ tables=people rows=n

Export: Release 10.1.0.3.0 - Production on Sun May 1 14:04:16 2005

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


Connected to: Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options

Export done in WE8ISO8859P1 character set and AL16UTF16 NCHAR character set

Note: table data (rows) will not be exported


About to export specified tables via Conventional Path ...

. . exporting table PEOPLE
Export terminated successfully without warnings.

[tkyte@localhost tkyte]$ imp userid=/ indexfile=people.sql full=y

Import: Release 10.1.0.3.0 - Production on Sun May 1 14:04:33 2005

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


Connected to: Oracle Database 10g Enterprise Edition Release 10.1.0.3.0 - Production

With the Partitioning, OLAP and Data Mining options


Export file created by EXPORT:V10.01.00 via conventional path

import done in WE8ISO8859P1 character set and AL16UTF16 NCHAR character set

Import terminated successfully without warnings.
```

分析 people.sql 文件，可以看到以下结果：

```
CREATE TABLE "OPS$TKYTE"."PEOPLE"

OF "PERSON_TYPE" OID 'F610318AC3D8981FE030007F01001464'

OIDINDEX (PCTFREE 10 INITRANS 2 MAXTRANS 255

      STORAGE(INITIAL 131072 NEXT 131072
```

```
    MINEXTENTS 1 MAXEXTENTS 4096

    PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1

    BUFFER_POOL DEFAULT)

TABLESPACE "USERS")

PCTFREE 10 PCTUSED 40

INITRANS 1 MAXTRANS 255

LOGGING STORAGE(INITIAL 131072 NEXT 131072

    MINEXTENTS 1 MAXEXTENTS 4096

    PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1

    BUFFER_POOL DEFAULT) TABLESPACE "USERS" NOCOMPRESS

/


ALTER TABLE "OPS$TKYTE"."PEOPLE" MODIFY

("SYS_NC_OID$" DEFAULT SYS_OP_GUID())

/
```

由 此我们可以更深入地了解到这里到底发生了什么，现在明显地看到了 OIDINDEX 子句，而且看到了对 SYS_NC_OID$列的一个引用。这是这个表的隐 藏主键。函数 SYS_OP_GUID 与 SYS_GUID 相同，他们都返回一个全局惟一的标识符，这是一个 16 字节的 RAW 字段。

OID’<big hex number>’语法在 Oracle 文档中没有相关说明。它的作用是在 EXP 和后续的 IMP 期间，确保底层类型 PERSON_TYPE 确实是相同的类型。这样当我们完成以下步骤时就能避免可能出现的错误：

（1） 创建 PEOPLE 表。

（2） 导出这个表。

（3） 删除这个表和底层 PERSON_TYPE。

（4） 用不同的属性创建一个新的 PERSON_TYPE。

（5） 导入原来的 PEOPLE 数据。

显然，原来导出的表无法导入到新结构中，因为结构不符。以上检查就能避免这种情况的出现。

如 果你还记得，我曾经提到过：为对象实例分配对象标识符的行为是可以修改的。可以不让系统为我们生成一个伪主键，而是使用对象的自然键。最初这看上去有些弄 巧成拙——SYS_NC_OID$仍然出现在 SYS.COL$的表定义中，相对于系统生成的列来说，似乎这会占用更多的存储空间。不过，这里神奇再现。如 果对象表所基于的是一个主键而不是系统生成的键，这个对象表的 SYS_NC_OID$列就是虚拟列，并不占用磁盘上的任何实际存储空间。

下面的例子显示了数据字典中发生了什么，并展示出 SYS_NC_OID$没有占用物理存储空间。先来分析系统生成的 OID 表：

```
ops$tkyte@ORA10G> create table people of person_type

  2 /
Table created.


ops$tkyte@ORA10G> select name, type#, segcollength

  2 from sys.col$

  3 where obj# = ( select object_id

  4          from user_objects
  5          where object_name = 'PEOPLE' )

  6                and name like 'SYS\_NC\_%' escape '\'

  7 /
NAME                          TYPE#       SEGCOLLENGTH

------------------------      ----------                  ------------

SYS_NC_OID$                     23                          16
SYS_NC_ROWINFO$           121                           1


ops$tkyte@ORA10G> insert into people(name)

  2 select rownum from all_objects;

48217 rows created.


ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'PEOPLE' );

PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10G> select table_name, avg_row_len from user_object_tables;

TABLE_NAME                     AVG_ROW_LEN

------------------------------ -----------

PEOPLE                                  23
```

在此我们看到，行平均长度为 23 字节：16 字节用于 SYS_NC_OID$，7 字节用于 NAME。还是做同样的事情，不过这一次使用 NAME 列上的一个主键作为对象标识符：

```
ops$tkyte@ORA10G> CREATE TABLE "PEOPLE"

  2 OF "PERSON_TYPE"

  3         ( constraint people_pk primary key(name) )

  4         object identifier is PRIMARY KEY
  5 /
Table created.


ops$tkyte@ORA10G> select name, type#, segcollength

  2 from sys.col$

  3 where obj# = ( select object_id

  4                 from user_objects

  5                 where object_name = 'PEOPLE' )

  6                       and name like 'SYS\_NC\_%' escape '\'
  7 /
NAME                           TYPE#       SEGCOLLENGTH

------------------------------ ----------  ------------

SYS_NC_OID$                    23                    81
SYS_NC_ROWINFO$                121                    1
```

根据结果来看，现在 SYS_NC_OID$列不是只有 16 字节，而变成一个 81 字节的大列！实际上，这里没有存储任何数据，它是空的。系统会根据对象表、其底层类型和行本身中的值生成一个惟一 ID。如下所示：

```
ops$tkyte@ORA10G> insert into people (name)
```

```
    2 values ( 'Hello World!' );

1 row created.


ops$tkyte@ORA10G> select sys_nc_oid$ from people p;

SYS_NC_OID$
--------------------------------------------------------------------------------
F610733A48F865F9E030007F0100149A00000017260100010001002900000000000C0700
1E01000
02A00078401FE000000140C48656C6C6F20576F726C642100000000000000000000000000000
0000000
0000


ops$tkyte@ORA10G> select utl_raw.cast_to_raw( 'Hello World!' ) data

    2 from dual;
DATA
--------------------------------------------------------------------------------
48656C6C6F20576F726C6421


ops$tkyte@ORA10G> select utl_raw.cast_to_varchar2(sys_nc_oid$) data

    2 from people;
DATA
--------------------------------------------------------------------------------
<garbage bits and bytes..>Hello World!
```

如 果选择 SYS_NC_OID$列，查看所插入串的 HEX 转储信息，可以看到行数据本身已经嵌入到对象 ID 中。将对象 ID 转换为一个 VARCHAR2，可以 更清楚地确认这一点。这是不是表示我们的数据要存储两次，而且存在大量开销？不，并非如此，这正是神奇之处，只是在获取时才有这样的 SYS_NC_OID $列。Oracle 从表中选择 SYS_NC_OID$时会合成数据。

下 面来表明我的观点。对象关系组件（嵌套表和对象表）就是我所说的“语法迷药”（syntactic sugar）。嵌套表和对象表总会转换为原来的关系行和列。我个人不把它们用作物理存储机制。”神奇之处“太多了，而且它们的副作用并不是很清楚。你会得 到隐藏列、额外的索引、奇怪的伪列等。这并不是说对象关系组件毫无用处，恰恰相反，我就经常在 PL/SQL 中使用这些组件。我会利用对象视图来得到对象关 系组件的功能。这么一来，不仅可以得到嵌套表结构的优点（同样能表示主表/明细表关系，但通过网络返回（传输）的数据较少；概念上也更容易于使用，等 等），而且不存在任何物理存储问题。这是因为我可以使用对象视图，从关系数据合成对象。这就解决了对象表/嵌套表的大多数问题，因为物理存储由我来指定， 联结条件也由我建立，而且这些表可以很自然地用作关系表（这一点是许多第三方工具和应用所要求的）。如果有人需要关系数据的对象视图，这是可以做到的；倘 若

有人需要关系视图，也同样可以达到目的。由于对象表实际上就是伪装的关系表，所以 Oracle 在底层为我们做的事情我们自己也可以做，而且还能更高效地 完成工作，因为我们不必像对象表那样采用一般性的方式。例如，使用前面定义的类型，可以很容易地使用以下命令创建对象视图：

```
ops$tkyte@ORA10G> create table people_tab

  2 ( name varchar2(30) primary key,

  3 dob date,

  4 home_city varchar2(30),

  5 home_street varchar2(30),

  6 home_state varchar2(2),
  7 home_zip number,

  8 work_city varchar2(30),

  9 work_street varchar2(30),

  10 work_state varchar2(2),

  11 work_zip number
  12 )
  13 /
Table created.


ops$tkyte@ORA10G> create view people of person_type

  2 with object identifier (name)

  3 as
  4       select name, dob,

  5           address_type(home_city,home_street,home_state,home_zip) home_adress,

  6           address_type(work_city,work_street,work_state,work_zip) work_adress

  7       from people_tab
  8 /
View created.
```

```
ops$tkyte@ORA10G> insert into people values ( 'Tom', '15-mar-1965',
  2          address_type( 'Reston', '123 Main Street', 'Va', '45678' ),
  3          address_type( 'Redwood', '1 Oracle Way', 'Ca', '23456' ) );
1 row created.
```

无论如何，效果完全相同，我很清楚存储什么、如何存储，以及在哪里存储。对于更复杂的对象，可能必须在对象视图上编写 INSTEAD OF 触发器，以允许通过视图修改数据。

对象表小结

Oracle 中 的对象表用于实现对象关系模型。一个对象表一般会创建多个物理的数据库对象，并向模式增加一些额外的列完成管理。对象表存在一些“神奇“的方面。利用对象 视图，你可以利用”对象“的语法和语义，与此同时，还能对数据的物理存储有完全的控制，并允许对底层数据进行关系型访问。采用这种方式，就能同时得到关系 世界和对象关系世界中最棒的特点。

10.11 小结

读完这一章后，希望你已经得出这样一个结论：并非所有的表都创建得完全一样。Oracle 提供了多种表类型可供使用。在这一章中，我们介绍了一般情况下表的一些突出的方面，并分析了 Oracle 为我们提供的各种表类型。

首 先，我们介绍了与表相关的一些术语以及存储参数。这里讨论了 freelist 在多用户环境中的作用，如果多个人同时频繁地插入/更新一个表， freelist 将产生很大影响；另外说明了如果使用 ASSM 表空间，就不必再考虑 freelist。我们研究了 PCTFREE 和 PCTUSED 的含义， 并为如何正确地设置这些信息提供了一些指导原则。

然 后开始介绍各种类型的表，先从最常见的堆表开始。堆组织表是目前大多数 Oracle 应用中最常用的表，这也是默认的表类型。接下来介绍索引组织表 （IOT），利用 IOT，可以把表数据存储在索引结构中而不是堆表中。我们了解了这些表适用于哪些情况（如查询表和反向表），在这些情况下，堆表就不合适 了，它将只是数据的一个冗余副本。在本章的后面，我们还指出了结合使用 IOT 与其他表类型确实很有用，特别是嵌套表类型。

我们还介绍了聚簇对象，Oracle 中有 3 种聚簇：索引聚簇、散列聚簇和有序散列聚簇。聚簇有两方面的目标：

- 使我们能够把多个表的数据共同存储在同一个（多个）数据库块上。
- 是我们能够强制把类似的数据根据某个聚簇键物理地存储在“一起“。例如，采用这种方式，部门 10 的所有数据（来自多个表）可以存储在一起。

基于这些特性，我们可以非常快地访问相关的数据，而且只有最少的物理 I/O。我们研究了索引聚簇和散列聚簇之间的主要区别，并讨论了各种聚簇分别在何种情况下适用（和不适用）。

接 下来转向嵌套表。我们介绍了嵌套表的语法、语义和用法，了解到嵌套表实际上就是系统生成和维护的父/子表对，并且知道了 Oracle 在物理上是如何做到这 一点的。我们分析了可以使用不同的表类型来实现嵌套表，默认会使用基于堆的表。我们发现，如果不使用 IOT 而使用堆表来实现嵌套表，一般这都是说不过去 的。

接下来我们介绍了临时表的有关细节，包括如何创建、从哪里得到存储空间，还说明了临时表在运行时不会引入与并发性相关的任何问题。我们分析了会话级和事务级临时表之间的区别，讨论了 Oracle 数据库中使用临时表的适当方法。

在这一章的最后，我们讨论了对象表的内部工作原则。我们发现，与嵌套表一样，Oracle 的对象表在底层也有很多“动作“。在此指出：利用关系表上的对象视图可以得到对象表的功能，与此同时还能轻松地访问底层关系数据。

第 11 章 索引

索引是应用设计和开发的一个重要方面。如果有太多的索引，DML 的性能就会受到影响。如果索引太少，又会影响查询（包括插入、更新和删除）的性能。要找到一个合适的平衡点，这对于应用的性能至关重要。

我 常常发现，人们在应用开发中总是事后才想起索引。我坚持认为这是一种错误的做法。如果你知道数据将如何使用，从一开始就应该能提出应用中要使用怎样的索 引，即具有一组代表性的索引。不过，一般的做法却往往是随应用“放任自流“，过后才发现哪里需要索引，这种情况实在太多了。这说明，你没有花时间来了解数 据将如何使用以及最终要处理多少行。经过一段时间后，随着数据量的增长，你会不停地向系统增加索引（也就是说，你所执行的是一种反应式调优）。你就有一些 冗余而且从不使用的索引，这不仅会浪费空间，还会浪费计算资源。磨刀不误砍柴工，如果刚开始的时候花几个小时好好地考虑何时为数据加索引，以及如何加索 引，这肯定能在以后的”调优“中节省更多的时间（注意，我所说

的是”肯定能“节省更多时间，而不只是”可能“节省更多时间)。

这一章的主旨是对 Oracle 中可用的索引提供一个概述，讨论什么时候以及在哪里可以使用索引。这一章的风格和格式与本书其他章有所不同。索引是一个很宽泛的主题，光是介绍索引就可以单独写一本书，其部分原因是：索引是开发人员和 DBA 角色之间的一个桥梁。一方面，开发人员必须了解索引，清楚如何在应用中使用索引，而且知道何时使用索引（以及何时不使用索引）等。另一方面，DBA 则要考虑索引的增长、索引中存储空间的使用以及其他物理特性。我们将主要从应用角度来考虑，也就是从索引的实际使用来介绍索引。这一章前半部分提供了一些基本知识。这一章的后半部分回答了关于索引的一些最常问到的问题。

这一章中的各个例子分别需要不同的 Oracle 版本中的特性。如果每个例子需要 Oracle 企业版或个人版的某些特性（而标准版中不支持），我会明确地指出来。

11.1 Oracle 索引概述

Oracle 提供了多种不同类型的索引以供使用。简单地说，Oracle 中包括如下索引：

B*树索引：这些是我所说的“传统“索引。到目前为止，这是 Oracle 和大多数其他数据库中最常用的索引。B*树的构造类似于二叉树，能根据键提供一行或一个行集的快速访问，通常只需很少的读操作就能找到正确的行。不过，需要注意重要的一点，”B*树“中的”B“不代表二叉（binary），而代表平衡（balanced）。B*树索引并不是一颗二叉树，这一点在介绍如何在磁盘上物理地存储 B*树时就会了解到。B*树索引有以下子类型：

索引组织表（index organized table）：索引组织表以 B*树结构存储。堆表的数据行是以一种无组织的方式存储的（只要有可用的空间，就可以放数据），而 IOT 与之不同，IOT 中的数据要按主键的顺序存储和排序。对应用来说，IOT 表现得与“常规“表并无二致；需要使用 SQL 来正确地访问 IOT。IOT 对信息获取、空间系统和 OLAP 应用最为有用。IOT 在上一章已经详细地讨论过。

B*树聚簇索引（B*tree cluster index）这些是传统 B*树索引的一个变体（只是稍有变化）。B*树聚簇索引用于对聚簇键建立索引（见第 11.章中“索引聚簇表“一节），所以这一章不再讨论。在传统 B*树中，键都指向一行；而 B*树聚簇不同，一个聚簇键会指向一个块，其中包含与这个聚簇键相关的多行。

降序索引（descending index）：降序索引允许数据在索引结构中按“从大到小“的顺序（降序）排序，而不是按”从小到大“的顺序（升序）排序。我们会解释为什么降序索引很重要，并说明降序索引如何工作。

反向键索引（reverse key index）：这也是 B*树索引，只不过键中的字节会“反转“。利用反向键索引，如果索引中填充的是递增的值，索引条目在索引中可以得到更均匀的分布。例如，如果使用一个序列来生成主键，这个序列将生成诸如 987500、987501、987502 等值。这些值是顺序的，所以倘若使用一个传统的 B*树索引，这些值就可能放在同一个右侧块上，这就加剧了对这一块的竞争。利用反向键，Oracle 则会逻辑地对 205789、105789、005789 等建立索引。Oracle 将数据放在索引中之前，将先把所存储数据的字节反转，这样原来可能在索引中相邻放置的值在字节反转之后就会相距很远。通过反转字节，对索引的插入就会分布到多个块上。

位图索引（bitmap index）：在一颗 B*树中，通常索引条目和行之间存在一种一对一的关系：一个索引条目就指向一行。而对于位图索引，一个索引条目则使用一个位图同时指向

多行。位图索引适用于高度重复而且通常只读的数据（高度重复是指相对于表中的总行数，数据只有很少的几个不同值）。考虑在一个有 100 万行的表中，每个列只有 3 个可取值：Y、N 和 NULL。举例来说，如果你需要频繁地统计多少行有值 Y，这就很适合建立位图索引。不过并不是说如果这个表中某一列有 11.000 个不同的值就不能建立位图索引，这一列当然也可以建立位图索引。在一个 OLTP 数据库中，由于存在并发性相关的问题，所以不能考虑使用位图索引（后面我们就会讨论这一点）。注意，位图索引要求使用 Oracle 企业版或个人版。

位图联结索引（bitmap join index）：这为索引结构（而不是表）中的数据提供了一种逆规范化的方法。例如，请考虑简单的 EMP 和 DEPT 表。有人可能会问这样一个问题：“多少人在位于波士顿的部门工作？“EMP 有一个指向 DEPT 的外键，要想统计 LOC 值为 Boston 的部门中的员工人数，通常必须完成表联结，将 LOC 列联结至 EMP 记录来回答这个问题。通过使用位图联结索引，则可以在 EMP 表上对 LOC 列建立索引。

基于函数的索引（function-based index）：这些就是 B*树索引或位图索引，它将一个函数计算得到的结果存储在行的列中，而不是存储列数据本身。可以把基于函数的索引看作一个虚拟列（或派生列）上的索引，换句话说，这个列并不物理存储在表中。基于函数的索引可以用于加快形如 SELECT * FROM T WHERE FUNCTION(DATABASE_COLUMN) = SAME_VALUE 这样的查询，因为值 FUNCTION(DATABASE_COLUMN)已经提前计算并存储在索引中。

应用域索引（application domain index）：应用域索引是你自己构建和存储的索引，可能存储在 Oracle 中，也可能在 Oracle 之外。你要告诉优化器索引的选择性如何，以及执行的开销有多大，优化器则会根据你提供的信息来决定是否使用你的索引。Oracle 文本索引就是应用域索引的一个例子；你也可以使用构建 Oracle 文本索引所用的工具来建立自己的索引。需要指出，这里创建的“索引“不需要使用传统的索引结构。例如，Oracle 文本索引就使用了一组表来实现其索引概念。

可以看到，可供选择的索引类型有很多。在下面几节中，我将提供有关的一些技术细节来说明某种索引如何工作，以及应该何时使用这些索引。需要重申一遍，在此不会涵盖某些与 DBA 相关的主题。例如，我们不打算讨论在线重建索引的原理；而会把重点放在与应用相关的实用细节上。

11.2 B*树索引

B*树索引就是我所说的“传统“索引，这是数据库中最常用的一类索引结构。其实现与二叉查找树很相似。其目标是尽可能减少 Oracle 查找数据所花费的时间。不严格地说，如果在一个数字列上有一个索引，那么从概念上讲这个结构可能如图 11.-1 所示。

注意 也许会有一些块级优化和数据压缩，这些可能会使实际的块结构与图 11.-1 所示并不同。

图 11.-1　典型的 B*树索引布局

这个树最底层的块称为叶子节点（leaf node）或叶子块（leaf block），其中分别包含各个索引键以及一个 rowid（指向所索引的行）。叶子节点之上的内部块称为分支块（branch block）。这些节点用于在结构中实现导航。例如，如果想在索引中找到值 42，要从树顶开始，找到左分支。我们要检查这个块，并发现需要找到范围在“42..50“的块。这个块将是叶子块，其中会指示包含数 42 的行。有意思的是，索引的叶子节点实际上构成了一个双向链表。一旦发现要从叶子节点中的哪里”开始“（也就是说，一旦发现第一个值），执行值的有序扫描（也称为索引区间扫描（index range scan））就会很容易。我们不用再在索引结构中导航；而只需根据需要通过叶子节点向前或向后扫描就可以了。所以要满足诸如以下的谓词条件将相当简单：

```
where x between 20 and 30
```

Oracle 发现第一个最小键值大于或等于 20 的索引叶子块，然后水平地遍历叶子节点链表，直到最后命中一个大于 30 的值。

B*树索引中不存在非惟一（nonunique）条目。在一个非惟一索引中，Oracle 会把 rowid 作为一个额外的列（有一个长度字节）追加到键上，使得键惟一。例如，如果有一个 CREATE INDEX I ON T(X,Y)索引，从概念上讲，它就是 CREATE UNIQUE INDEX I ON T(X,Y,ROWID)。在一个惟一索引中，根据你定义的惟一性，Oracle 不会再向索引键增加 rowid。在非惟一索引中，你会发现，数据会首先按索引键值排序（依索引键的顺序）。然后按 rowid 升序排序。而在惟一索引中，数据只按索引键排序。

B*树的特点之一是：所有叶子块都应该在树的同一层上。这一层也称为索引的高度（height），这说明所有从索引的根块到叶子块的遍历都会访问同样数目的块。也就是说，对于形如”SELECT INDEXED_COL FROM T WHERE INDEXED_COL = :X“的索引，要到达叶子块来获取第一行，不论使用的:X 值是什么，都会执行同样数目的 I/O。换句话说，索引是高度平衡的（height balanced）。大多数 B*树索引的高度都是 2 或者 3，即使索引中有数百万行记录也是如此。这说明，一般来讲，在索引中找到一个键只需要执行 2 或 3 次 I/O，这

倒不坏。

注意 Oracle在表示从索引根块到叶子块遍历所涉及的块数时用了两个含义稍有不同的术语。第一个是高度（HEIGHT），这是指从根块到叶子块遍历所需的块数。使用ANALYZE INDEX <name> VALIDATE STRUCTURE 命令分析索引后，可以从INDEX_STATS视图找到这个高度（HEIGHT）值。另一个术语是BLEVEL，这是指分支层数，与HEIGHT相差1（BLEVEL不把叶子块层算在内）。收集统计信息后，可以在诸如USER_INDEXES之类的常规字典表中找到BLEVEL值。

例如，假设有一个11.,000,000行的表，其主键索引建立在一个数字列上：

```
big_table@ORA9IR2> select index_name, blevel, num_rows

    2 from user_indexes

    3 where table_name = 'BIG_TABLE';

INDEX_NAME                    BLEVEL       NUM_ROWS

------------------------------        ----------              ----------

BIG_TABLE_PK                          2           10441513
```

BLEVEL为2，这说明HEIGHT为3，要找到叶子需要两个I/O（访问叶子本身还需要第三个I/O）。所以，要从这个索引中获取任何给定的键值，共需要3个I/O：

```
big_table@ORA9IR2> select id from big_table where id = 42;

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=11.Bytes=6)

11.0            INDEX (UNIQUE SCAN) OF 'BIG_TABLE_PK' (UNIQUE) (Cost=2 Card=11.Bytes=6)

Statistics

----------------------------------------------------------

...

3 consistent gets

...

11.rows processed
```

```
big_table@ORA9IR2> select id from big_table where id = 12345;

Statistics

----------------------------------------------------------

... 3 consistent gets

... 11.rows processed

big_table@ORA9IR2> select id from big_table where id = 1234567;

Statistics

----------------------------------------------------------

...

3 consistent gets

...

11.rows processed
```

B*树是一个绝佳的通用索引机制，无论是大表还是小表都很适用，随着底层表大小的增长，获取数据的性能只会稍有恶化（或者根本不会恶化）。

11.2.1 索引键压缩

对于 B*树索引，可以做的一件有意思的工作是“压缩”。这与压缩 ZIP 文件的方式不同，它是指从串联（多列）索引去除冗余。

在第 11.章的“索引组织表”一节中我们曾经详细地讨论压缩键索引，这里再简要地做个说明。压缩键索引（compressed key index）的基本概念是，每个键条目分解为两个部分：“前缀”和“后缀”。前缀建立在串联索引（concatenated index）的前几列上，这些列有许多重复的值。后缀则在索引键的后几列上，这是前缀所在索引条目中的惟一部分（即有区别的部分）。

下面通过一个例子来说明，我们将创建一个表和一个串联索引，并使用 ANALYZE INDEX 测量无压缩时所用的空间。然后利用索引压缩创建这个索引，分别压缩不同数目的键条目，查看有什么差别。下面先来看这个表和索引：

```
ops$tkyte@ORA10G> create table t

  2 as

  3 select * from all_objects;
```

```
Table created.



ops$tkyte@ORA10G> create index t_idx on

  2 t(owner,object_type,object_name);

Index created.



ops$tkyte@ORA10G> analyze index t_idx validate structure;

Index analyzed.
```

然后创建一个 INX_STATS 表，在这里存放 INDEX_STATS 信息，我们把表中的行标记为“未压缩”（noncompressed）：

```
ops$tkyte@ORA10G> create table idx_stats

  2 as

  3 select 'noncompressed' what, a.*

  4 from index_stats a;

Table created.
```

现在可以看到，OWNER 部分重复了多次，这说明，这个索引中的一个索引块可能有数十个条目，如图 11.-2 所示。

图 11.-2 有重复 OWNER 列的索引块

可以从中抽取出重复的 OWNER 列，这会得到如同 11.-3 所示的块。

图 11.-3　抽取了 OWNER 列的索引块

在图 11.-3 中，所有者（owner）名在叶子块上只出现了一次，而不是在每个重复的条目上都出现一次。运行以上脚本，传入数字 1 作为参数来重新创建这个索引，在此索引使用了第一列的压缩：

```
drop index t_idx;

create index t_idx on

t(owner,object_type,object_name)

compress &1;

analyze index t_idx validate structure;

insert into idx_stats

select 'compress &1', a.*

from index_stats a;
```

为了进行比较，我们不仅在压缩一列的基础上运行了这个脚本，还分别使用了两个和 3 个压缩列来运行这个脚本，查看会发生什么情况。最终，我们将查询 IDX_STATS，应该能观察到以下信息：

```
ops$tkyte@ORA10G> select what, height, lf_blks, br_blks,

  2 btree_space, opt_cmpr_count, opt_cmpr_pctsave

  3 from idx_stats

  4 /

WHAT                          HEIGHT          LF_BLKS                   BR_BLKS
BTREE_SPACE    OPT_CMPR_COUNT OPT_CMPR_PCTSAVE

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

```
noncompressed       3        337              3          2718736                    2
          28

compress
1                   3        300              3          2421684                    2
11.

compress
2                   2        240              1          1926108                    2
 0

compress
3                   3        375              3          3021084                    2
35
```

可以看到，COMPRESS 1 索引的大小大约是无压缩索引的 89%（通过比较 BTREE_SPACE 得出）。叶子块数大幅度下降。更进一步，使用 COMPRESS 2 时，节省的幅度更为显著。所得到的索引大约是原索引（无压缩索引）的 70%，而且由于数据量减少，这些数据能放在单个块上，相应地索引的高度就从 3 降为 2.实际上，利用列 OPT_CMPR_PCTSAVE 的信息（这代表最优的节省压缩百分比（optimum compression percent saved）或期望从压缩得到的节省幅度）。我们可以猜测出 COMPRESS 2 索引的大小：

```
ops$tkyte@ORA10G> select 2718736*(11.0.28) from dual;



  2718736*(11.0.28)

          ----------

          1957489.92
```

注意 对无压缩索引执行 ANALYZE 命令时，会填写 OPT_CMPR_PCTSAVE/OPT_CMPR_COUNT 列，并估计出：利用 COMPRESS 2，可以节省 28%的空间；而事实确实如此，我们果真节省了大约这么多的空间。

不过，再看看 COMPRESS 3 会怎么样。如果压缩 3 列，所得到的索引实际上会更大：是原来索引大小的 110%。这是因为：每删除一个重复的前缀，能节省 N 个副本的空间，但是作为压缩机制的一部分，这会在叶子块上增加 4 字节的开销。把 OBJECT_NAME 列增加到压缩键后，则使得这个键是惟一的；在这种情况下，则说明没有重复的副本可以提取。因此，最后的结果就是：我们只是向每个索引键条目增加了 4 个字节，而不能提取出任何重复的数据。IDX_STATS 中的 OPT_CMPR_COUNT 列真是精准无比，确实给出了可用的最佳压缩数，OPT_COMPR_PCTSAVE 则指出了可以得到多大的节省幅度。

对现在来说，这种压缩不是免费的。现在压缩索引比原来更复杂了。Oracle 会花更多的时间来处理这个索引结构中的数据，不光在修改期间维护索引更耗时，查询期间搜索索引也更花时间。利用压缩，块缓冲区缓存比以前能存放更多的索引条目，缓冲命中率可能会上升，物理 I/O 应该下降，但是要多占用一些 CPU 时间来处理索引，还会增加块竞争的可能

性。在讨论散列聚簇时，我们曾经说过，对于散列聚簇，获取 100 万个随机的行可能占用更多的 CPU 时间，但是 I/O 数会减半；这里也是一样，我们必须清楚存在的这种折中。如果你现在已经在大量占用 CPU 时间，在增加压缩键索引只能适得其反，这会减慢处理的速度。另一方面，如果目前的 I/O 操作很多，使用压缩键索引就能加快处理速度。

11.2.2 反向键索引

B*树索引的另一个特点是能够将索引键“反转”。首先，你可以问问自己“为什么想这么做？” B*树索引是为特定的环境、特定的问题而设计的。实现 B*树索引的目的是为了减少“右侧”索引中对索引叶子块的竞争，比如在一个 Oracle RAC 环境中，某些列用一个序列值或时间戳填充，这些列上建立的索引就属于“右侧”（right-hand-side）索引。

注意 我们在第 2 章讨论过 RAC。

RAC 是一种 Oracle 配置，其中多个实例可以装载和打开同一个数据库。如果两个实例需要同时修改同一个数据块，它们会通过一个硬件互连（interconnect）来回传递这个块来实现共享，互连是两个（或多个）机器之间的一条专用网络连接。如果某个利用一个序列填充，这个列上有一个主键索引（这是一种非常流行的实现），那么每个人插入新值时，都会视图修改目前索引结构右侧的左块（见图 11.-1，其中显示出索引中较高的值都放在右侧，而较低的值放在左侧）。如果对用序列填充的列上的索引进行修改，就会聚集在很少的一组叶子块上。倘若将索引的键反转，对索引进行插入时，就能在索引中的所有叶子键上分布开（不过这往往会使索引不能得到充分地填充）。

注意 你可能还会注意到，反向键可以用作一种减少竞争的方法（即使只有一个 Oracle 实例）。不过重申一遍，如这一节所述，反向键主要用于缓解忙索引右侧的缓冲区忙等待。

在介绍如何度量反向键索引的影响之前，我们先来讨论物理上反向键索引会做什么。反向键索引只是将索引键中各个列的字节反转。如果考虑 90101、90102 和 90103 这样几个数，使用 Oracle DUMP 函数查看其内部表示，可以看到这几个数的表示如下：

```
ops$tkyte@ORA10GR1> select 90101, dump(90101,11.) from dual
  2 union all
  3 select 90102, dump(90102,11.) from dual
  4 union all
  5 select 90103, dump(90103,11.) from dual
  6 /

     90101       DUMP(90101,11.)
----------       --------------------
```

```
90101      Typ=2 Len=4: c3,a,2,2

90102      Typ=2 Len=4: c3,a,2,3

90103      Typ=2 Len=4: c3,a,2,4
```

每个数的长度都是 4 字节，它们只是最后一个字节有所不同。这些数最后可能在一个索引结构中向右依次放置。不过，如果反转这些数的字节，Oracle 就会插入以下值：

```
ops$tkyte@ORA10GR1> select 90101, dump(reverse(90101),11.) from dual

  2 union all

  3 select 90102, dump(reverse(90102),11.) from dual

  4 union all

  5 select 90103, dump(reverse(90103),11.) from dual

  6 /



    90101      DUMP(REVERSE(90101),1

---------      ---------------------

    90101      Typ=2 Len=4: 2,2,a,c3

    90102      Typ=2 Len=4: 3,2,a,c3

    90103      Typ=2 Len=4: 4,2,a,c3
```

注意　REVERSE 函数没有相关的文档说明，因此，使用是当心。我不建议在“实际“代码中使用 REVERSE，因为它没有相关的文档，这说明这个函数未得到公开支持。

这些数彼此之间最后会“相距很远“。这样访问同一个块（最右边的块）的 RAC 实例个数就能减少，相应地，在 RAC 实例之间传输的块数也会减少。反向键索引的缺点之一是：能用常规索引的地方不一定能用反向键索引。例如，在回答以下谓词时，X 上的反向键索引就没有：

```
where x > 5
```

存储之前，数据不是按 X 在索引中排序，而是按 REVERSE(X)排序，因此，对 X>5 的区间扫描不能使用这个索引。另一方面，有些区间扫描确实可以在反向键索引上完成。如果在(X,Y)上有一个串联索引，以下谓词就能够利用反向键索引，并对它执行“区间扫描“：

```
where x = 5
```

这是因为，首先将 X 的字节反转，然后再将 Y 的字节反转。Oracle 并不是将（X||Y）的字节反转，而是会存储(REVERSE(X) || REVERSE(Y))。这说明， X = 5 的所有值会存储

在一起，所以 Oracle 可以对这个索引执行区间扫描来找到所有这些数据。

下面，假设在一个用序列填充的表上有一个代理主键（surrogate primary key），而且不需要在这个（主键）索引上使用区间扫描，也就是说，不需要做 MAX(primary_key)、MIN(primary_key)、WHERE primary_key < 100 等查询，在有大量插入操作的情况下，即使只有一个 Oracle 实例，也可以考虑使用反向键索引。我建立了两个不同的测试，一个是在纯 PL/SQL 环境中进行测试，另一个使用了 Pro*C，我想通过这两个测试来展示反向键索引和传统索引对插入的不同影响，即如果一个表的主键上有一个反向键索引，与有一个传统索引的情况相比，完成插入时会有什么差别。在这两种情况下，所用的表都是用以下 DDL 创建的（这里使用了 ASSM 来避免表块的竞争，这样可以把索引块的竞争隔离开）：

```
create table t tablespace assm
    as
    select 0 id, a.*
    from all_objects a
    where 11.0;


    alter table t
    add constraint t_pk
    primary key (id)
    using index (create index t_pk on t(id) &indexType tablespace assm);


create sequence s cache 1000;
```

在此如果把&indexType 替换为关键字 REVERSE，就会创建一个反向键索引，如果不加&indexType（即替换为“什么也没有“），则表示使用一个”常规“索引。要运行的 PL/SQL 代码如下，将分别由 1、2、5、11.或 11.个用户并发运行这个代码：

```
create or replace procedure do_sql
as
begin
    for x in ( select rownum r, all_objects.* from all_objects )
```

```
    loop
            insert into t
                ( id, OWNER, OBJECT_NAME, SUBOBJECT_NAME,
                OBJECT_ID, DATA_OBJECT_ID, OBJECT_TYPE, CREATED,
                LAST_DDL_TIME, TIMESTAMP, STATUS, TEMPORARY,
                GENERATED, SECONDARY )
            values
                ( s.nextval, x.OWNER, x.OBJECT_NAME, x.SUBOBJECT_NAME,
                x.OBJECT_ID, x.DATA_OBJECT_ID, x.OBJECT_TYPE, x.CREATED,
                x.LAST_DDL_TIME, x.TIMESTAMP, x.STATUS, x.TEMPORARY,
                x.GENERATED, x.SECONDARY );
            if ( mod(x.r,100) = 0 )
            then
                commit;
            end if;
    end loop;
    commit;
end;
/
```

我们已经在第 9 章讨论过 PL/SQL 提交时优化，所以现在我想运行使用另一种环境的测试，以免被这种提交时优化所误导。我使用了 Pro*C 来模拟一个数据仓库抽取、转换和加载（extract, transform, load, ETL）例程，它会在提交之间一次成批地处理 100 行（即每次提交前都处理 100 行）：

```
    exec sql declare c cursor for select * from all_objects;
    exec sql open c;
```

```
exec sql whenever notfound do break;

for(;;)

{

        exec sql

        fetch c into :owner:owner_i,

        :object_name:object_name_i, :subobject_name:subobject_name_i,

        :object_id:object_id_i, :data_object_id:data_object_id_i,

        :object_type:object_type_i, :created:created_i,

        :last_ddl_time:last_ddl_time_i, :timestamp:timestamp_i,

        :status:status_i, :temporary:temporary_i,

        :generated:generated_i, :secondary:secondary_i;



        exec sql

        insert into t

            ( id, OWNER, OBJECT_NAME, SUBOBJECT_NAME,

            OBJECT_ID, DATA_OBJECT_ID, OBJECT_TYPE, CREATED,

            LAST_DDL_TIME, TIMESTAMP, STATUS, TEMPORARY,

            GENERATED, SECONDARY )

        values

            ( s.nextval, :owner:owner_i, :object_name:object_name_i,

            :subobject_name:subobject_name_i, :object_id:object_id_i,

            :data_object_id:data_object_id_i, :object_type:object_type_i,

            :created:created_i, :last_ddl_time:last_ddl_time_i,
```

```
                :timestamp:timestamp_i, :status:status_i,
                :temporary:temporary_i, :generated:generated_i,
                :secondary:secondary_i );
            if ( (++cnt%100) == 0 )
            {
                exec sql commit;
            }
    }
    exec sql whenever notfound continue;
    exec sql commit;
    exec sql close c;
```

Pro*C 预编译时 PREFETCH 设置为 100，使得这个 C 代码与上述 PL/SQL 代码（要求 Oracle 版本为 Oracle 10g）是相当的，它们有同样的表现。

注意 在 Oracle 10g Release 1 及以上版本中，PL/SQL 中简单的 FOR X IN(SELECT * FROM T)会悄悄地一次批量获取 100 行，而在 Oracle9i 及以前版本中，只会一次获取一行。因此，如果想在 Oracle9i 及以前版本中执行这个测试，就需要修改 PL/SQL 代码，利用 BULK COLLECT 语法成批地进行获取。

两种代码都会一次获取 100 行，然后将数据逐行插入到另一个表中。下表总结了各次运行之间的差别，先从单用户测试开始，如表 11.-1 所示。

表 11.-1 利用 PL/SQL 和 Pro*C 对使用反向键索引进行性能测试：单用户

	反向 PL/SQL	**无反向 PL/SQL**	**反向 Pro*C**	**无反向 Pro*C**
事务/秒	38.24	43.45	11..35	11..08
CPU 时间（秒）	25	22	33	31
缓冲区忙等待数/时间	0/0	0/0	0/0	0/0
耗用时间（分钟）	0.42	0.37	0.92	0.83
日志文件同步数/时间		6/0	11.940/7	11.940/7

从第一个单用户测试可以看到，PL/SQL 代码在执行这个操作时比 Pro*C 代码高效得多，随着用户负载的增加，我们还将继续看到这种趋势。Pro*C 之所以不能像 PL/SQL 那样很好地扩展，部分原因在于 Pro*C 必须等待日志文件同步等待，而 PL/SQL 提供了一种优化可以避免这种日志文件同步等待。

从这个单用户测试来看，似乎方向键索引会占用更多的 CPU 时间。这是有道理的，因为数据库必须执行额外的工作来反转键中的字节。不过，随着用户数的增加，我们会看到这种情况不再成立。随着竞争的引入，方向键索引的开销会完全消失。实际上，甚至在执行两个用户的测试时，这种开销就几乎被索引右侧的竞争所抵消，如表 11.-2 所示。

表 11.-2 利用 PL/SQL 和 Pro*C 对使用反向键索引进行性能测试：两个用户

	反向 PL/SQL	**无反向 PL/SQL**	**反向 Pro*C**	**无反向 Pro*C**
事务/秒	46.59	49.03	20.07	20.29
CPU 时间（秒）	77	73	104	101
缓冲区忙等待数/时间	4,267/2	133,644/2	3,286/0	23,688/1
耗用时间（分钟）	0.68	0.65	11.58	11.57
日志文件同步数/时间	11./0	11./0	3,273/29	2,132/29

从这个两用户的测试中可以看到，PL/SQL 还是要优于 Pro*C。另外在 PL/SQL 这一边，使用方向键索引已经开始显示出某种好处，而在 Pro*C 一边还没有明显的反映。这种趋势也会延续下去。方向键索引可以解决由于索引结构中对最右边的块的竞争而导致的缓冲区忙等待问题，不过，对于影响 Pro*C 程序的日志文件同步等待问题则无计可施。这正是我们同时执行一个 PL/SQL 测试和一个 Pro*C 测试的主要原因：我们就是想看看这两种环境之间有什么差别。由此产生了一个问题：在这种情况下，为什么方向键索引对 PL/SQL 明显有好处，而对 Pro*C 似乎没有作用？归根结底就是因为日志文件同步等待事件。PL/SQL 能不断地插入，而很少在提交时等待日志文件同步等待事件，而 Pro*C 不同，它必须每 100 行等待一次。因此，在这种情况下，与 Pro*C 相比，PL/SQL 更多地要受缓冲区忙等待的影响。在 PL/SQL 中如果能缓解缓冲区忙等待，它就能处理更多事务，所以方向键索引对 PL/SQL 很有好处。但是对于 Pro*C，缓冲区忙等待并不是根本问题，这不是主要的性能瓶颈，所以消除缓冲区忙等待对于总体性能来说没有说明影响。

下面来看 5 个用户的测试，如表 11.-3 所示

表 11.-3 利用 PL/SQL 和 Pro*C 对使用反向键索引进行性能测试：5 个用户

	反向 PL/SQL	**无反向 PL/SQL**	**反向 Pro*C**	**无反向 Pro*C**
事务/秒	43.84	39.78	11..22	11..11.

CPU 时间（秒）	389	395	561	588
缓冲区忙等待数/时间	11.,259/45	221,353/153	11.,118/9	157,967/56
耗用时间（分钟）	11.82	2.00	4.11.	4.38
日志文件同步数/时间		691/11.	6,655/73	5,391/82

这里的结果似曾相识。PL/SQL 程序运行时几乎没有日志同步等待，所以会显著地受缓冲区忙等待的影响。倘若采用一个传统索引，如果 5 个用户都试图插入索引结构的右侧，PL/SQL 受到缓冲区忙等待的影响最大，相应地，如果能减少这种缓冲区忙等待，所得到的好处也最明显。

下面来看 11.个用户测试，如表 11.-4 所示，可以看到这种趋势还在延续。

表 11.-4 利用 PL/SQL 和 Pro*C 对使用反向键索引进行性能测试：11.个用户

	反向 PL/SQL	**无反向 PL/SQL**	**反向 Pro*C**	**无反向 Pro*C**
事务/秒	45.90	35.38	11..88	11..05
CPU 时间（秒）	781	789	11.256	11.384
缓冲区忙等待数/时间	26,846/279	456,231/11.382	25,871/134	364,556/11.702
耗用时间（分钟）	3.47	4.50	8.90	9.92
日志文件同步数/时间		2,602/72	11.,032/196	11.,653/141

PL/SQL 程序中完全没有日志文件同步等待，通过消除缓冲区忙等待事件，会大为受益。尽管 Pro*C 程序出现遭遇到更多的缓冲区忙等待竞争，但是由于它还在频繁地等待日志文件同步事件，所以方向键索引对它的好处不大。对于一个有常规索引的 PL/SQL 实现，要改善它的性能，一种方法是引入一个小等待。这会减少对索引右侧的竞争，并提高总体性能。由于篇幅有限，这里不再给出 11.个和 20 个用户测试的情况，但是可以确保一点，这一节观察到的趋势还会延续。

在这个演示中，我们得出了两个结论。方向键索引有助于缓解缓冲区忙等待问题：但是取决于其他的一些因素，你的投资可能会得到不同的回报。查看 11.用户测试的表 11.-4 时，可以看到，通过消除缓冲区忙等待（在这里，这是等待最多的等待事件），将对事务吞吐量稍有影响；同时也确实显示出：随着并发程度的提高，可扩展性会增加。而对 PL/SQL 做同样的工作时，对性能的影响则有很大不同：通过消除这个瓶颈，吞吐量会有大幅提升。

11.2.3 降序索引

降序索引（descending index）是 Oracle8i 开始引入的，用以扩展 B*树索引的功能。它

允许在索引中以降序（从大到小的顺序）存储一列，而不是升序（从小到大）存储。在之前的 Oracle 版本（即 Oracle8i 以前的版本）中，尽管语法上也支持 DESC（降序）关键字，但是一般都会将其忽略，这个关键字对于索引中数据如何存储或使用没有任何影响。不过，在 Oracle8i 及以上版本中，DESC 关键字确实会改变创建和使用索引的方式。

Oracle 能往前读索引，这种能力已不算新，所以你可能会奇怪我们为什么会兴师动众地说这个特性很重要。例如，如果使用先前的表 T，并如下查询这个表：

```
ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> select owner, object_type

  2 from t

  3 where owner between 'T' and 'Z'

  4       and object_type is not null

  5 order by owner DESC, object_type DESC;

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=82 Card=11395
Bytes=170925)
```

Oracle 会往前读索引。这个计划最后没有排序步骤：数据已经是有序的。不过，如果你有一组列，其中一些列按升序排序（ASC），另外一些列按降序排序（DESC），此时这种降序索引就能派上用场了，例如：

```
ops$tkyte@ORA10G> select owner, object_type

  2 from t

  3 where owner between 'T' and 'Z'

  4       and object_type is not null

  5 order by owner DESC, object_type ASC;

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=85 Card=11395
Bytes=170925)
```

```
11.0          SORT (ORDER BY) (Cost=85 Card=11395 Bytes=170925)

2 1               INDEX (RANGE SCAN) OF 'T_IDX' (INDEX) (Cost=82 Card=11395 ...
```

Oracle 不能再使用（OWNER, OBJECT_TYPE, OBJECT_NAME）上的索引对数据排序。它可以往前读得到按 OWNER DESC 排序的数据，但是现在还需要“向后读“来得到按 OBJET_TYPE 顺序排序（ASC）数据。此时 Oracle 的实际做法是，它会把所有行收集起来，然后排序。但是如果使用 DESC 索引，则有：

```
ops$tkyte@ORA10G> create index desc_t_idx on t(owner desc,object_type asc);

Index created.

ops$tkyte@ORA10G> exec dbms_stats.gather_index_stats( user, 'DESC_T_IDX' );

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select owner, object_type

  2 from t

  3 where owner between 'T' and 'Z'

  4       and object_type is not null

  5 order by owner DESC, object_type ASC;

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=11395
Bytes=170925)

11.0          INDEX (RANGE SCAN) OF 'DESC_T_IDX' (INDEX) (Cost=2
Card=11395 ...
```

现在，我们又可以读取有序的数据了，在这个计划的最后并没有额外的排序步骤。应当注意，除非 init.ora 中的 compatible 参数设置为 8.11.0 或更高，否则 CREATE INDEX 上的 DESC 选项会被悄悄地忽略，没有警告，也不会产生错误，因为这是先前版本的默认行为。

注意 查询中最好别少了 ORDER BY。即使你的查询计划中包含一个索引，但这并不表示数据会以“某种顺序“返回。要想从数据库以某种有序的顺序获取数据，惟一的办法就是在查询中包括一个 ORDER BY 子句。ORDER BY 是无可替代的。

11.2.4 什么情况下应该使用 B*树索引？

我并不盲目地信息“经验“（所有规则都有例外），所以，对于什么时候该使用（和不该使用）B*树索引，我没有什么经验可以告诉你。为了说明为什么在这方面不能提供经验，下面给出两种做法，这两种做法同等有效：

- 仅当要通过索引访问表中很少的一部分行（只占一个很小的百分比）时，才使用 B*树在列上建立索引。
- 如果要处理表中的多行，而且可以使用索引而不用表，就可以使用一个 B*树索引。
- 这两个规则看上去彼此存在冲突，不过在实际中，它们并不冲突，只是涵盖了两种完全不同的情况。根据以上建议，有两种使用索引的方法：
- 索引用于访问表中的行：通过读索引来访问表中的行。此时你希望访问表中很少的一部分行（只占一个很小的百分比）。
- 索引用于回答一个查询：索引包含了足够的信息来回答整个查询，我根本不用去访问表。在这种情况下，索引则用作一个“较瘦“版本的表。

此 外，还有其他的一些做法，例如我们还可以使用一个索引来获取表中的所有行，包括索引本身中没有的列。这看上去好像与前面的经验相左。交互式应用中可能就是 这种情况，你要得到并显示一些行，然后再得到一些行，如此继续。为此，你可能希望优化查询以使最初的响应时间很短，而不是针对吞吐量进行优化。

第一种情况（也就是说，为了访问表中很少的一部分行而使用索引）是指，如果你有一个表 T（还是使用前面的表 T），并有如下的一个查询计划：

```
ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> select owner, status

  2 from t

  3 where owner = USER;

Execution Plan

----------------------------------------------------------

0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1947 Bytes=25311)

11.0          TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE) (Cost=3 Card=1947 ....

2 1                INDEX (RANGE SCAN) OF 'DESC_T_IDX' (INDEX) (Cost=2 Card=8)
```

那你就应该只访问这个表的很少一部分行（只占一个很小的百分比）。这里要注意 TABLE ACCESS BY INDEX ROWID 后面的 INDEX (RANGE SCAN)。这说明，Oracle 会读索引，然后会对索引条目执行一个数据库块读（逻辑或物理 I/O）来得到行数据。如果你要

通过索引访问 T 中的大量行（占很大的百分比），这就不是最高效的方法了（稍后我们将定义多少才算是大百分比）。

在第二种情况下（也就是说，可以用索引而不必用表），你可以通过索引处理表中 100%的行（或者实际上可以是任何比例）。使用索引可以只是为了创建一个“较瘦“版本的表。以下查询演示了这个概念：

```
ops$tkyte@ORA10G> select count(*)

  2 from t

  3 where owner = user;

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11. Card=11.Bytes=6)

11.0            SORT (AGGREGATE)

2 1                     INDEX (RANGE SCAN) OF 'T_IDX' (INDEX) (Cost=11. Card=1947
```

在此，只使用了索引来回答查询，现在访问多少行都没关系，因为我们只会使用索引。从查询计划可以看出，这里从未访问底层表；我们只是扫描了索引结构本身。

重要的是，要了解这两个概念之间的区别。如果必须完成 TABLE ACCESS BY INDEX ROWID，就必须确保只访问表中很少的一部分块（只占很小的百分比），这通常对应为很少的一部分行，或者需要能够尽可能快地获取前面的几行（最终用户正在不耐烦地等着这几行）。如果我们访问的行太多（所占百分比过大，不在总行数的 1%~20%之间），那么与全表扫描相比，通过 B*树索引来访问这些数据通常要花更长的时间。

对 于第二种类型的查询，即答案完全可以在索引中找到，情况就完全不同了。我们会读一个索引块，选出许多“行“来处理，然后再到另一个索引块，如此继续，在此 从不访问表。某些情况下，还可以在索引上执行一个快速全面扫描，从而更快地回答这类查询。快速全面扫描是指，数据库不按特定的顺序读取索引块，而只是开始 读取它们。这里不再是把索引只用作一个索引，此时索引更像是一个表。如果采用快速全面扫描，将不再按索引条目的顺序来得到行。

一般来讲，B*树索引会放在频繁使用查询谓词的列上，而且我们希望从表中只返回少量的数据（只占很小的百分比），或者最终用户请求立即得到反馈。在一个瘦（thin）表（也就是说，只有很少的几个列，或者列很小）上，这个百分比可能相当小。使用这个索引的查询应该只获取表中 2%～3%（或者更少）的行。在一个胖（fat）表中（也就是说，这个表有很多列，或者列很宽），百分比则可能会上升到表的 20%～25%。以上建议不一定直接适用于每一个人；这个比例并不直观，但很精确。索引按索引键的顺序存储。索引会按键的有序顺序进行访问。索引指向的块则随机地存储在堆中。因此，我们通过索引访问表时，会执行大量分散、随机的 I/O。这里“分散“（scattered）是指，索引会告诉我们读取块 1，然后是块 11.000、块 205、块 321、块 1、块 11.032、块 1，等等，它不会要求我们按一种连续的方式读取块 1、然后是块 2，接着是块 3.我们将以一种非常随意的方式读取和重新读取块。这

种块 I/O 可能非常慢。

下面来看这样一个简化的例子，假设我们通过索引读取一个瘦表，而且要读取表中 20%的行。若这个表中有 100,000 行，其中的 20%就是 2,000 行。如果行大小约为 80 字节，在一个块大小为 8KB 的数据库中，每个块上则有大约 100 行。这说明，这个表有大约 11.000 个块。了解了执行情况，计算起来就非常容易了。我们要通过索引读取 20,000 行；这说明，大约是 20,000 个 TABLE ACCESS BY ROWID 操作。为此要处理 20,000 个表块来执行这个查询。不过，整个表才有大约 11.000 个块！最后会把表中的每一个块读取好处理 20 次。即使把行的大小提高一个数量级，达到每行 800 字节，这样每块有 11.行，现在表中就有 11.,000 个块。要通过索引访问 20,000 行，仍要求我们把每一个块平均读取 2 次。在这种情况下，全表扫描就比使用索引高效得多，因为每个块只会命中一次。如果查询使用这个索引来访问数据，效率都不会高，除非对于 800 字节的行，平均只访问表中不到 5%的数据（这样一来，就只会访问大约 5,000 个块），如果是 80 字节的行，则访问的数据应当只占更小的百分比（大约 0.5%或更少）。

1. 物理组织

数据在磁盘上如何物理地组织，对上述计算会有显著影响，因为这会大大影响索引访问的开销有多昂贵（或者有多廉价）。假设有一个表，其中的行主键由一个序列来填充。向这个表增加数据时，序列号相邻的行一般存储位置也会彼此“相邻“。

注意 如果使用诸如 ASSM 或多个 freelist/freelist 组等特性，也会影响数据在磁盘上的组织。这些特性力图将数据发布开，这样可能就观察不到按主键组织的这种自然聚簇。

表会很自然地按主键顺序聚簇（因为数据或多或少就是已这种属性增加的）。当然，它不一定严格按照键聚簇（要想做到这一点，必须使用一个 IOT），但是，一般来讲，主键值彼此接近的行的物理位置也会“靠“在一起。如果发出以下查询：

```
select * from T where primary_key between :x and :y
```

你想要的行通常就位于同样的块上。在这种情况下，即使要访问大量的行（占很大的百分比），索引区间扫描可能也很有用。原因在于：我们需要读取和重新读取的数据库块很可能会被缓存，因为数据共同放置在同一个位置（co-located）。另一方面，如果行并非共同存储在一个位置上，使用这个索引对性能来讲可能就是灾难性的。只需一个小小的演示就能说明这一点。首先创建一个表，这个表主要按其主键排序：：

```
ops$tkyte@ORA10G> create table colocated ( x int, y varchar2(80) );

Table created.



ops$tkyte@ORA10G> begin

  2        for i in 1 .. 100000

  3        loop
```

```
  4                 insert into colocated(x,y)
  5                 values (i, rpad(dbms_random.random,75,'*') );
  6           end loop;
  7 end;
  8 /
PL/SQL procedure successfully completed.


ops$tkyte@ORA10G> alter table colocated
  2 add constraint colocated_pk
  3 primary key(x);
Table altered.


ops$tkyte@ORA10G> begin
  2 dbms_stats.gather_table_stats( user, 'COLOCATED', cascade=>true );
  3 end;
  4 /
PL/SQL procedure successfully completed.
```

这个表正好满足前面的描述，即在块大小为 8KB 的一个数据库中，每块有大约 100 行。在这个表中，，X = 11.2,3 的行极有可能在同一个块上。仍取这个表，但有意地使它“无组织“。在 COLOCATED 表中，我们创建一个 Y 列，它带有一个前导随机数，现在利用这一点使得数据”无组织“，即不再按主键排序：

```
ops$tkyte@ORA10G> create table disorganized
  2 as
  3 select x,y
  4 from colocated
```

```
    5 order by y;

Table created.



ops$tkyte@ORA10G> alter table disorganized

    2 add constraint disorganized_pk

    3 primary key (x);

Table altered.



ops$tkyte@ORA10G> begin

    2 dbms_stats.gather_table_stats( user, 'DISORGANIZED', cascade=>true );

    3 end;

    4 /

PL/SQL procedure successfully completed.
```

可以证明，这两个表是一样的——这是一个关系数据库，所以物理组织对于返回的答案没有影响（至少关于数据库理论的课程中就是这么教的）。实际上，尽管会返回相同的答案，但这两个表的性能特征却有着天壤之别。给定同样的问题，使用同样的查询计划，查看 TKPROF（SQL 跟踪）输出，可以看到以下报告：

```
select * from colocated where x between 20000 and 40000

call     count       cpu    elapsed       disk      query    current        rows

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

Parse        5      0.00       0.00          0          0          0           0

Execute      5      0.00       0.00          0          0          0           0

Fetch     6675      0.59       0.60          0      14495          0      100005

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

total     6685      0.59       0.60          0      14495          0      100005
```

```
Rows Row Source Operation

-------  ------------------------------------------------

20001  TABLE ACCESS BY INDEX ROWID COLOCATED (cr=2899 pr=0 pw=0
time=120134 us)

20001  INDEX RANGE SCAN COLOCATED_PK (cr=1374 pr=0 pw=0 time=40081
us)(object id...

********************************************************************************

select /*+ index( disorganized disorganized_pk ) */* from disorganized

where x between 20000 and 40000

call     count       cpu    elapsed       disk      query    current        rows

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

Parse        5      0.00        0.00           0           0           0           0

Execute      5      0.00        0.00           0           0           0           0

Fetch     6675      0.85        0.87           0      106815           0      100005

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

total     6685      0. 85       0.87           0      106815           0      100005


Rows Row Source Operation

-------  ------------------------------------------------

20001  TABLE ACCESS BY INDEX ROWID DISORGANIZED (cr=21363 pr=0 pw=0
time=220144 ...

20001  INDEX RANGE SCAN DISORGANIZED_PK (cr=1374 pr=0 pw=0
time=40311 us)(...
```

注意　我把每个查询运行了 5 次，从而得到每个查询的“平均“运行时间。

简直是难以置信。物理数据布局居然会带来这么大的差异！表 11.-5 对这些结果做了一个总结。

表 11.-5 研究物理数据布局对索引访问开销的影响

表	CPU 时间	逻辑 I/O
共同放置（Co-located）	0.59 秒	11.,495
无组织（Disorganized）	0.85 秒	106,815
共同放置表与无组织表的开销相对百分比	70%	11.%

在我的数据库中（块大小为 8KB），这些表的总块数如下：

```
ops$tkyte@ORA10G> select table_name, blocks

2 from user_tables

3 where table_name in ( 'COLOCATED', 'DISORGANIZED' );



TABLE_NAME                BLOCKS

------------------------------        ----------

COLOCATED                    1252

DISORGANIZED                 1219
```

对无组织表的查询正如先前计算的一样：我们做了 20,000 次以上的逻辑 I/O（这里总共查询了 100,000 个块，因为查询运行了 5 次，所以每次查询做了 100,000/5 = 20,000 次逻辑 I/O）。每个块被处理了 20 次！另一方面，对于物理上共同放置的数据（COLOCATED），逻辑 I/O 次 数则大幅下降。由此很好地说明了为什么很难提供经验：在某种情况下，使用这个索引可能很不错，但在另外一种情况下它却不能很好地工作。从生产系统转储数据 并加载到开发系统时也可以考虑这一点，因为这至少能从一定程度上回答如下问题：“为什么在这台机器上运行得完全不同？难道它们不一样吗？“不错，他们确实 不一样。

注意 第 6 章曾说过，逻辑 I/O 会增加，但这还只是冰山一角。每个逻辑 I/O 都涉及缓冲区缓存的一个或多个锁存器。在一个多用户/CPU 情况下，在我们自旋并等待锁存器时，与第一个查询相比，第二个查询所用的 CPU 时间无疑会高出几倍。第二个示例查询不仅要完成更多的工作，而且无法像第一个查询那样很好地扩展。

ARRAYSIZE 对逻辑 I/O 的影响
有一个问题很有意思，这就是 ARRAYSIZE 对所执行逻辑 I/O 的影响。ARRAYSIZE 是客户请求下一行时 Oracle 向客户返回的行数。客户将缓存这些行，在向数据库请求下一个行集之前会先使用缓存的这些行，ARRAYSIZE 对查询执行的逻辑 I/O 可能有非常重要的影

响，这是因为，如果必须跨数据库调用反复地访问同一个块（也就是说，通过多个数据库调用反复访问同一个块，这里特别是指跨获取调用），Oracle 就必须一而再、再而三地从缓冲区缓存获取这个块。因此，如果一个调用从数据库请求 100 行，Oracle 可能就能够处理完一个数据库块，而无需再次获取这个块。如果你一次请求 11.行，Oracle 就必须反复地获得同一个块来获取同样的行集。

在这一节前面的例子中，我们使用了 SQL*Plus 的默认批量获取大小（11.行，如果把获取的总行数除以获取调用的个数，所得到的结果将非常接近 11.）。我们在每次获取 11.行和每次获取 100 行的情况下执行前面的查询，从而做一个比较，会观察到对于 COLOCATED 表有以下结果：

```
select * from colocated a15 where x between 20000 and 40000

Rows Row Source Operation

------- ---------------------------------------------------

20001 TABLE ACCESS BY INDEX ROWID COLOCATED (cr=2899 pr=0 pw=0 time=120125...

20001 INDEX RANGE SCAN COLOCATED_PK (cr=1374 pr=0 pw=0 time=40072 us)(...



select * from colocated a100 where x between 20000 and 40000

Rows Row Source Operation

------- ---------------------------------------------------

20001 TABLE ACCESS BY INDEX ROWID COLOCATED (cr=684 pr=0 pw=0 ...)

20001 INDEX RANGE SCAN COLOCATED_PK (cr=245 pr=0 pw=0 ...
```

执行第一个查询时 ARRAYSIZE 为 11.，Row Source Operation 中（cr-nnnn）值显示出，对这个索引执行了 11.374 个逻辑 I/O，并对表执行了 11.625 个逻辑 I/O（2,899-11.374；这些数在 Row Source Operation 步骤中加在了一起，即 2,899）。把 ARRAYSIZE 从 11.增加到 100 时，对索引的逻辑 I/O 数减至 245，这是因为，不必每 11.行就从缓冲区缓存重新读取索引叶子块，而是每 100 行才读取一次。为了说明这一点，假设每个叶子块上能存储 200 行。如果扫描索引时每次只能读取 11.行，则必须把第一个叶子块获取 11.次才能得到其中的全部 200 行。另一方面，如果每次批量获取 100 行，只需要将这个叶子块从缓冲区缓存中获取两次就能得到其中的所有行。

对于表块也存在这种情况。由于表按索引键同样的顺序排序，所以会更少地获取各个表块，原因是每个获取调用能从表中得到更多的行。

这么说来，如果增加 ARRAYSIZE 对 COLOCATED 表很合适，那么这对于

DISORGANIZED 表也同样会很好，是这样吗？并非如此。DISORGANIZED 表的相应结果如下所示：

```
select /*+ index( a15 disorganized_pk ) */ *

from disorganized a15 where x between 20000 and 40000



Rows Row Source Operation

------- ---------------------------------------------------

20001 TABLE ACCESS BY INDEX ROWID DISORGANIZED (cr=21357 pr=0 pw=0 ...

20001 INDEX RANGE SCAN DISORGANIZED_PK (cr=1374 pr=0 pw=0 ...



select /*+ index( a100 disorganized_pk ) */ *

from disorganized a100 where x between 20000 and 40000



Rows Row Source Operation

------- ---------------------------------------------------

20001 TABLE ACCESS BY INDEX ROWID OBJ#(75652) (cr=20228 pr=0 pw=0 ...

20001 INDEX RANGE SCAN OBJ#(75653) (cr=245 pr=0 pw=0 time=20281 us)(...
```

索引有关的结果并无不同，这是有道理的，因为不论表如何组织，索引中存储的数据都一样，执行一次查询时，如果从 11.增加到 100，对索引执行的逻辑 I/O 总次数在调整 ARRAYSIZE 前后并没有太大差别：分别是 21,357 和 20,281。为什么呢？原因是对表执行的逻辑 I/O 次数根本没有变化，如果把每个查询执行的逻辑 I/O 总次数减去对索引执行的逻辑 I/O 次数，就会发现这两个查询对表执行的逻辑 I/O 此时都是 11.,983。这是因为，每次我们希望从数据库得到 N 行时，其中的任何两行在同一个块上的机率非常小，所以通过一个调用就从一个表块中得到多行是不可能的。

我见过的每一种与 Oracle 交互的专业编程语言都实现了这种批量获取（array fetching）的概念。在 PL/SQL 中，可以使用 BULK COLLECT，也可以依靠为隐式游标 for 循环执行的隐式批量获取（一次 100 行）来实现。在 Java/JDBC 中，连接（connect）或语句（statement）对象上有一个预获取（prefetch）方法。Oracle 调用接口（Oracle Call Interface，即 OCI，这是一个 C API）与 Pro*C 类似，允许在程序中设置预获取大小。可以看到，这对查询执行的

逻辑 I/O 次数会有重要而且显著的影响，值得注意。

在这个例子的最后，来看一下全面扫描 DISORGANIZED 表时会发生什么：

```
select * from disorganized where x between 20000 and 40000


call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        5      0.00       0.00          0          0          0           0
Execute      5      0.00       0.00          0          0          0           0
Fetch     6675      0.53       0.54          0      12565          0      100005
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total     6685      0.53       0. 54 0    12565          0      100005

Rows Row Source Operation
-------  ---------------------------------------------------
20001 TABLE ACCESS FULL DISORGANIZED (cr=2513 pr=0 pw=0 time=60115 us)
```

由此显示出，在这个特殊的例子中，根据数据在磁盘上的物理存储的方式，非常合适采用全表扫描。这就带来一个问题：“为什么优化器不先对这个查询执行全面扫描呢？“这么说吧，如果按照它自己本来的设计，确实会先执行全面扫描，但是在对 DISORGANIZED 的第一个示例查询中，我有意地为查询提供了提示，告诉优化器要构造一个使用这个索引的计划。在第二个例子中，则是让优化器自己来选择最佳的字体计划。

2. 聚簇因子

接下来，我们来看 Oracle 所用的一些信息。我们要特别查看 USER_INDEXES 视图中的 CLUSTERING_FACTOR 列。Oracle reference 手册指出了这个列有以下含义：

根据索引的值指示表中行的有序程度：

- ☐ 如果这个值与块数接近，则说明表相当有序，得到了很好的组织，在这种情况下，同一个叶子块中的索引条目可能指向同一个数据块上的行。
- ☐ 如果这个值与行数接近，表的次序可能就是非常随机的。在这种情况下，同一个叶子块上的索引条目不太可能指向同一个数据块上的行。

可以把聚簇因子（clustering factor）看作是通过索引读取整个表时对表执行的逻辑 I/O 次数。也就是说，CLUSTERING_FACTOR 指示了表相对于索引本身的有序程度，查看这些索引时，会看到以下结果：

```
ops$tkyte@ORA10G> select a.index_name,
  2          b.num_rows,
  3          b.blocks,
  4          a.clustering_factor
  5 from user_indexes a, user_tables b
  6 where index_name in ('COLOCATED_PK', 'DISORGANIZED_PK' )
  7          and a.table_name = b.table_name
  8 /

INDEX_NAME          NUM_ROWS       BLOCKS      CLUSTERING_FACTOR
---------------                  ----------     ----------                  -----------------
COLOCATED_PK          100000          1252                          1190
DISORGANIZED_PK       100000          1219                         99932
```

注意 对于这一节的例子，我使用了一个 ASSM 管理的表空间，由此可以解释为什么 COLOCATED 表的聚簇因子小于表中的块数。COLOCATED 表中在 HWM 之下有一些未格式化的块，其中未包含数据，而且 ASSM 本身也使用了一些块来管理空间，索引区间扫描中不会读取这些块。第 11.章更详细地解释了 HWM 和 ASSM。

所以数据库说：“如果通过索引 COLOCATED_PK 从头到尾地读取 COLOCATED 表中的每一行，就要执行 11.190 次 I/O。不过，如果我们对 DISORGANIZED 表做同样的事情，则会对这个表执行 99,932 次 I/O。“之所以存在这么大的区别，原因在于，当 Oracle 对索引结构执行区间扫描时，如果它发现索引中的下一行于前一行在同一个数据库块上，就不会再执行另一个 I/O 从缓冲区缓存中获得表块。它已经有表块的一个句柄，只需直接使用就可以了。不过，如果下一行不在同一个块上，就会释放当前的这个块，而执行另一个 I/O 从缓冲区缓存获取要处理的下一个块。因此，在我们对索引执行区间扫描时，COLOCATED_PK 索引会发现下一行几乎总于前一行在同一个块上。DISORGANIZED_PK 索引发现的情况则恰好相反。实际上，你会看到这个测量相当准确。通过使用提示，让优化器使用索引全面扫描来读取整个表，再统计非 NULL 的 Y 值个数，就能看到通过索引读取整个表需要执行多少次 I/O：

```
select count(Y)
from
    (select /*+ INDEX(COLOCATED COLOCATED_PK) */ * from colocated)
```

```
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      0.11.      0.11.         0       1399          0           1
-------  ------  --------  ---------- ----------  ----------  ----------  ----------
total        4      0.11.      0.11.         0       1399          0           1

  Rows     Row Source Operation
  -------  ---------------------------------------------------
       1   SORT AGGREGATE (cr=1399 pr=0 pw=0 time=160325 us)
  100000   TABLE ACCESS BY INDEX ROWID COLOCATED (cr=1399 pr=0 pw=0 time=500059 us)
  100000   INDEX FULL SCAN COLOCATED_PK (cr=209 pr=0 pw=0 time=101057 us)(object ...

********************************************************************************

select count(Y)
from
    (select /*+ INDEX(DISORGANIZED DISORGANIZED_PK) */ * from disorganized)


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      0.34       0.40          0     100141          0          1

-------  ------  --------   ----------  ----------  ----------  ----------  ----------

total        4      0.34       0.40          0     100141          0          1

  Rows     Row Source Operation

  -------  ---------------------------------------------------

        1  SORT AGGREGATE (cr=100141 pr=0 pw=0 time=401109 us)

100000  TABLE ACCESS BY INDEX ROWID OBJ#(66615) (cr=100141 pr=0 pw=0
time=800058...

100000   INDEX FULL SCAN OBJ#(66616) (cr=209 pr=0 pw=0 time=101129
us)(object...
```

在这两种情况下，索引都需要执行 209 次逻辑 I/O（Row Source Operation 行中的 cr-209）。如果将一致读（consistent read）总次数减去 209，只测量对表执行的 I/O 次数，就会发现所得到的数字与各个索引的聚簇因子相等。COLOCATED_PK 是“有序表“的一个经典例子，DISORGANIZE_PK 则是一个典型的”表次序相当随机“的例子。现在来看看这对优化器有什么影响。如果我们想获取 25,000 行，Oracle 对两个索引都会选择全表扫描（通过索引获取 25%的行不是最优计划，即使是对很有序的表也是如此）。不过，如果只选择表数据的 11.%，就会观察到以下结果：

```
ops$tkyte@ORA10G> select * from colocated where x between 20000 and 30000;



Execution Plan

----------------------------------------------------------

0        SELECT STATEMENT Optimizer=ALL_ROWS (Cost=143 Card=10002
Bytes=800160)

11.0          TABLE ACCESS (BY INDEX ROWID) OF 'COLOCATED' (TABLE)
(Cost=143 ...

2 1                INDEX (RANGE SCAN) OF 'COLOCATED_PK' (INDEX (UNIQUE))
(Cost=22 ...
```

```
ops$tkyte@ORA10G> select * from disorganized where x between 20000 and 30000;



Execution Plan

----------------------------------------------------------

0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=337 Card=10002
Bytes=800160)

11.0      TABLE ACCESS (FULL) OF 'DISORGANIZED' (TABLE) (Cost=337
Card=10002 ...
```

这里的表结构和索引与前面完全一样，但聚簇因子有所不同。在这种情况下，优化器为 COLOCATED 表选择了一个索引访问计划，而对 DISORGANIZED 表选择了一个全面扫描访问计划。要记住，11.%并不是一个阀值，它只是一个小于 25%的数，而且在这里会导致对 COLOCATED 表的一个索引区间扫描。

以 上讨论的关键点是，索引并不一定总是合适的访问方法。优化器也许选择不使用索引，而且如前面的例子所示，这种选择可能很正确。影响优化器是否使用索引的因 素有很多，包括物理数据布局。因此，你可能会矫枉过正，力图重建所有的表来使所有索引有一个好的聚簇因子，但是在大多数情况下这可能只会浪费时间。只有当 你在对表中的大量数据（所占百分比很大）执行索引区间扫描时，这才会产生影响。另外必须记住，对于一个表来说，一般只有一个索引能有合适的聚簇因子！表中 的行可能只以一种方式排序。在前面所示的例子中，如果 Y 列上还有一个索引，这个索引在 COLOCATED 表中可能就不能很好地聚簇，而在 DISORGANIZED 表中则恰好相反。如果你认为数据物理聚簇很重要，可以考虑使用一个 IOT、B*树聚簇，或者在连续地重建表时考虑散列聚簇。

11.2.5 B*树小结

B*树索引是到目前为止 Oracle 数据库中最常用的索引结构，对 B*树索引的研究和了解也最为深入。它们是绝好的通用索引机制。在访问时间方面提供了很大的可扩缩性，从一个 11.000 行的索引返回数据所用的时间与一个 100,000 行的索引结构中返回数据的时间是一样的。

什么时候建立索引，在哪些列上建立索引，你的设计中必须注意这些问题。索引并不一定就意味着更快的访问；实际上你会发现，在许多情况下，如果 Oracle 使 用索引，反而会使性能下降。这实际上两个因素的一个函数，其中一个因素是通过索引需要访问表中多少数据（占多大的百分比），另一个因素是数据如何布局。如 果能完全使用索引“回答问题“（而不用表），那么访问大量的行（占很大的百分比）就是有意义的，因为这样可以避免读表所带来的额外的分散 I/O。如果使用索引来访问表，可能就要确保只处理整个表中的很少一部分（只占很小的百分比）。

应该在应用的设计期间考虑索引的设计和实现，而不要事后才想起来（我就经常见到这种情况）。如果对如何访问数据做了精心的计划和考虑，大多数情况下就能清楚地知道需要什么索引。

11.3 位图索引

位图索引（bitmap index）是从 Oracle7.3 版本开始引入的。目前 Oracle 企业版和个人版都支持位图索引，但标准版不支持。位图索引是为数据仓库/即席查询环境设计的，在此所有查询要求的数据在系统实现时根本不知道。位图索引特别不适用于 OLTP 系统，如果系统中的数据会由多个并发会话频繁地更新，这种系统也不适用位图索引。

位图索引是这样一种结构，其中用一个索引键条目存储指向多行的指针；这与 B*树结构不同，在 B*树结构中，索引键和表中的行存在着对应关系。在位图索引中，可能只有很少的索引条目，每个索引条目指向多行。而在传统的 B*树中，一个索引条目就指向一行。

下面假设我们要在 EMP 表的 JOB 列上创建一个位图索引，如下：

```
Ops$tkyte@ORA10G> create BITMAP index job_idx on emp(job);

Index created.
```

Oracle 在索引中存储的内容如表 11.-6 所示。

表 11.-6 Oracle 如何存储 JOB-IDX 位图索引

值/行	1	2	3	4	5	6	7	8	9	11.	11.	11.	11.	11.
ANALYST	0	0	0	0	0	0	0	1	0	1	0	0	1	0
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
PRESIDENT	0	0	0	0	0	0	0	0	1	0	0	0	0	0
SALESMAN	0	1	1	0	1	0	0	0	0	0	0	0	0	0

表 11.-6 显示了第 8、11.和 11.行的值为 ANALYST，而第 4、6 和 7 行的值为 MANAGER。在此还显示了所有行都不为 null（位图索引可以存储 null 条目；如果索引中没有 null 条目，这说明表中没有 null 行）。如果我们想统计值为 MANAGER 的行数，位图索引就能很快地完成这个任务。如果我们想找出 JOB 为 CLERK 或 MANAGER 的所有行，只需根据索引合并它们的位图，如表 11.-7 所示。

表 11.-7 位 OR 的表示

值/行	1	2	3	4	5	6	7	8	9	11.	11.	11.	11.	11.
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
CLERK 或	1	0	0	1	0	1	1	0	0	0	1	1	0	1

MANAGER

表 11.-7 清楚地显示出，第 1、4、6、7、11.、11.还 11.行满足我们的要求。Oracle 如下为每个键值存储位图，使得每个位置表示底层表中的一个 rowid，以后如果确实需要访问行时，可以利用这个 rowid 进行处理。对于以下查询：

```
select count(*) from emp where job = 'CLERK' or job = 'MANAGER'
```

用位图索引就能直接得出答案。另一方面，对于以下查询：

```
select * from emp where job = 'CLERK' or job = 'MANAGER'
```

则需要访问表。在此 Oracle 会应用一个函数把位图中的第 i 位转换为一个 rowid，从而可用于访问表。

11.3.1 什么情况下应该使用位图索引？

位图索引对于相异基数（distinct cardinality）低的数据最为合适（也就是说，与职工数据集的基数相比，这个数据只有很少几个不同的值）。对此做出量化是不太可能的——换句话说，很难定义低相异基数到底是多大。在一个有几千条记录的数据集中，2 就是一个低相异基数，但是在一个只有两行的表中，2 就不能算是低相异基数了。而在一个有上千万或上亿条记录的表中，甚至 100,000 都能作为一个低相异基数。所以，多大才算是低相异基数，这要相对于结果集的大小来说。这是指，行集中不同项的个数除以行数应该是一个很小的数（接近于 0）。例如，GENDER 列可能取值为 M、F 和 NULL。如果一个表中有 20,000 条员工记录，那么 3/20000=0.00015。类似地，如果有 100,000 个不同的值，与 11.,000,000 条结果相比，比值为 0.01，同样这也很小（可算是低相异基数）。这些列就可以建立位图索引。它们可能不适合建立 B*树索引，因为每个值可能会获取表中的大量数据（占很大百分比）。如前所述，B*树索引一般来讲应当是选择性的。与之相反，位图索引不应是选择性的，一般来讲它们应该“没有选择性“。

如果有大量即席查询，特别是查询以一种即席方式引用了多列或者会生成诸如 COUNT 之类的聚合，在这样的环境中，位图索引就特别有用。例如，假设你有一个很大的表，其中有 3 列：GENDER、LOCATION 和 AGE_GROUP。在这个表中，GENDER 只有两个可取值：M 或 F，LOCATION 可取值为 11.50，AGE_GROUP 是一个代码，分别表示 11. and under（11. 及以下）、11.-25、26-30、31-40 和 41 and over（41 及以上）。你必须支持大量即席查询，形式如下：

```
Select count(*)

from T

where gender = 'M'

        and location in ( 1, 11., 30 )

        and age_group = '41 and over';
```

```
select *

from t

where ( ( gender = 'M' and location = 20 )

                or ( gender = 'F' and location = 22 ))

      and age_group = '11. and under';



select count(*) from t where location in (11.,20,30);



select count(*) from t where age_group = '41 and over' and gender = 'F';
```

你会发现，这里使用传统的 B*树索引机制是不行的。如果你想使用一个索引来得到答案，就需要组合至少 3～6 个可能的 B*树索引，才能通过索引访问数据。由于这 3 列或它们的任何子集都有可能出现，所以需要在以下列上建立很大的串联 B*树索引：

- GENDER、LOCATION 和 AGE_GROUP：对应使用了这 3 列、使用了 GENDER 和 LOCATION 或者仅使用 GENDER 的查询。
- LOCATION、AGE_GROUP：对应使用了 LOCATION 和 AGE_GROUP 或者仅使用 LOCATION 的查询。
- AGE_GROUP、GENDER：对应使用了 AGE_GROUP 和 GENDER 或者仅使用 LOCATION 的查询。

要减少搜索的数据量，还可以有其他排列，以减少所扫描索引结构的大小。这是因为在此忽略了这样一个重要事实：对这种低基数数据建立 B*树索引并不明智。

这里位图索引就能派上用场了。利用分别建立在各个列上的 3 个较小的位图索引，就能高效地满足前面的所有谓词条件。对于引用了这 3 列（其中任何一例及任何子集）的任何谓词，Oracle 只需对 3 个索引的位图使用函数 AND、OR 和 NOT，就能得到相应的结果集。它会得到合并后的位图，如果必要还可以将位图中的“1”转换为 rowid 来访问数据（如果只是统计与条件匹配的行数，Oracle 就只会统计“1”位的个数）。下面来看一个例子。首先，生成一些测试数据（满足我们指定的相异基数），建立索引，并收集统计。我们将利用 DBMS_RANDOM 包来生成满足我们的分布要求的随机数据：

```
ops$tkyte@ORA10G> create table t

  2 ( gender not null,

  3 location not null,

  4 age_group not null,
```

```
  5 data
  6 )
  7 as
  8 select decode( ceil(dbms_random.value(11.2)),
  9             1, 'M',
 11.            2, 'F' ) gender,
 11.            ceil(dbms_random.value(11.50)) location,
 11.            decode( ceil(dbms_random.value(11.5)),
 11.            1,'11. and under',
 11.            2,'11.-25',
 11.            3,'26-30',
 11.            4,'31-40',
 11.            5,'41 and over'),
 11.            rpad( '*', 20, '*')
 11. from big_table.big_table
 20 where rownum <= 100000;

Table created.

ops$tkyte@ORA10G> create bitmap index gender_idx on t(gender);

Index created.

ops$tkyte@ORA10G> create bitmap index location_idx on t(location);

Index created.
```

```
ops$tkyte@ORA10G> create bitmap index age_group_idx on t(age_group);

Index created.



ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T', cascade=>true );

PL/SQL procedure successfully completed.
```

现在来看前面各个即席查询的相应查询计划：

```
ops$tkyte@ORA10G> Select count(*)

  2 from T

  3 where gender = 'M'

  4       and location in ( 1, 11., 30 )

  5       and age_group = '41 and over';



Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=5 Card=11.Bytes=11.)

11.0          SORT (AGGREGATE)

2 1             BITMAP CONVERSION (COUNT) (Cost=5 Card=11.Bytes=11.)

3 2               BITMAP AND

4  3                BITMAP INDEX (SINGLE VALUE) OF 'GENDER_IDX' (INDEX
(BITMAP))

5 3                BITMAP OR

6 5                  BITMAP INDEX (SINGLE VALUE) OF 'LOCATION_IDX' (INDEX
(BITMAP))

7 5                  BITMAP INDEX (SINGLE VALUE) OF 'LOCATION_IDX' (INDEX
```

```
(BITMAP))

8 5                    BITMAP INDEX (SINGLE VALUE) OF 'LOCATION_IDX' (INDEX
(BITMAP))

9 3                BITMAP INDEX (SINGLE VALUE) OF 'AGE_GROUP_IDX' (INDEX
(BITMAP))
```

这个例子展示出了位图索引的强大能力。Oracle 能看到 location in (11.11.,30)，知道要读取这 3 个位置（对于这 3 个值的位置）上的索引，并在位图中对这些“位”执行逻辑 OR。然后将得到的位图与 AGE_GROUP=’41 AND OVER’和 GENDER=’M’的相应位图执行逻辑 AND。再统计”1”的个数，这就得到了答案：

```
ops$tkyte@ORA10G> select *

  2 from t

  3 where ( ( gender = 'M' and location = 20 )

  4           or ( gender = 'F' and location = 22 ))

  5           and age_group = '11. and under';



Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=77 Card=507 Bytes=16731)

11.0         TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE) (Cost=77 Card=507 ...

2 1            BITMAP CONVERSION (TO ROWIDS)

3 2               BITMAP AND

4 3                BITMAP INDEX (SINGLE VALUE) OF 'AGE_GROUP_IDX' (INDEX
(BITMAP))

5 3                BITMAP OR

6 5                   BITMAP AND

7 6                      BITMAP INDEX (SINGLE VALUE) OF 'LOCATION_IDX'
(INDEX (BITMAP))
```

```
8 6                    BITMAP INDEX (SINGLE VALUE) OF 'GENDER_IDX' (INDEX
(BITMAP))

9 5                    BITMAP AND

11. 9                  BITMAP INDEX (SINGLE VALUE) OF 'GENDER_IDX'
(INDEX (BITMAP))

11. 9                  BITMAP INDEX (SINGLE VALUE) OF 'LOCATION_IDX'
(INDEX (BITMAP))
```

这个逻辑与前面是类似的，由计划显示：这里执行逻辑 OR 的两个条件是通过 AND 适当的位图逻辑计算得到的，然后再对这些结果执行逻辑 OR 得到一个位图。再加上另一个 AND 条件（以满足 AGE_GROUP=’11.’ AND UNDER），我们就找到了满足所有条件的结果。由于这一次要请求具体的行，所以 Oracle 会把位图中的各个“1“和”0“转换为 rowid，来获取源数据。

在一个数据仓库或支持多个即席 SQL 查询的大型报告系统中，能同时合理地使用尽可能多的索引确实非常有用。这里不常使用传统的 B*树索引，甚至不能使用 B*树索引，随着即席查询要搜索的列数的增加，需要的 B*树索引的组合也会飞速增长。

不过，在某些情况下，位图并不合适。位图索引在读密集的环境中能很好地工作，但是对于写密集的环境则极不适用。原因在于，一个位图索引键条目指向多行。如果一个会话修改了所索引的数据，那么在大多数情况下，这个索引条目指向的所有行都会被锁定。Oracle 无 法锁定一个位图索引条目中的单独一位；而是会锁定这个位图索引条目。倘若其他修改也需要更新同样的这个位图索引条目，就会被“关在门外“。这样将大大影响 并发性，因为每个更新都有可能锁定数百行，不允许并发地更新它们的位图列。在此不是像你所想的那样锁定每一行，而是会锁定很多行。位图存储在块（chunk）中，所以，使用前面的 EMP 例子就可以看到，索引键 ANALYST 在索引中出现了多次，每一次都指向数百行。更新一行时，如果修改了 JOB 列，则需要独占地访问其中两个索引键条目：对应老值的索引键条目和对应新值的索引键条目。这两个条目指向的数百行就不允许其他会话修改，直到 UPDATE 提交。

11.3.2 位图联结索引

Oracle9i 引入了一个新的索引类型：位图联结索引（bitmap join index）。通常都是在一个表上创建索引，而且只使用这个表的列。位图联结索引则打破了这个规则，它允许使用另外某个表的列对一个给定表建立索引。实际上，这就允许对一个索引结构（而不是表本身）中的数据进行逆规范化。

考虑简单的 EMP 表和 DEPT 表。EMP 有指向 DEPT 的一个外键（DEPTNO 列）。DEPT 表有一个 DNAME 属性（部门名）。最终用户会频繁地问这样的问题：“销售部门有多少人？“”谁在销售部门工作？“”可以告诉我销售部门中业绩最好的前 N 个人吗？“注意他们没有这样问：”DEPTNO 为 30 的部门中有多少人？“他们没有用这些键值；而是用了人可读的部门名。因此，最后运行的查询如下所示：

```
select count(*)
```

```
from emp, dept

where emp.deptno = dept.deptno

and dept.dname = 'SALES'

/



select emp.*

from emp, dept

where emp.deptno = dept.deptno

and dept.dname = 'SALES'

/
```

使用传统索引的话，这些查询中 DEPT 表和 EMP 表都必须访问。我们可以使用 DEPT.DNAME 上的一个索引来查找 SALES 行，并获取 SALES 的 DEPTNO 值，然后使用 EMP.DEPTNO 上的一个索引来查找匹配的行，但是如果使用一个位图联结索引，就不需要这些了。利用位图联结索引，我们能对 DEPT.DNAME 列建立索引，但这个索引不是指向 DEPT 表，而是指向 EMP 表。这是一个全新的概念：能从其他表对某个表的属性建立索引，而这可能会改变你的报告系统中实现数据模型的方式。实际上，可以鱼和熊掌兼得，一方面保持规范化数据结构不变，与此同时还能得到逆规范化的好处。

以下是我们为这个例子创建的索引：

```
ops$tkyte@ORA10G> create bitmap index emp_bm_idx

  2 on emp( d.dname )

  3 from emp e, dept d

  4 where e.deptno = d.deptno

  5 /

Index created.
```

注意，这个 CREATE INDEX 开始看上去很“正常“，它会在表上创建索引 INDEX_NAME。但之后就不那么”正常“了。可以看到在此引用了 DEPT 表中的一列：D.DNAME。这里有一个 FROM 子句。使这个 CREATE INDEX 语句有些像查询。另外，多个表之间有一个联结条件。这个 CREATE INDEX 语句对 DEPT.DNAME 列建立了索引，但这在 EMP 表的上下文中。对于前面提到的那些问题，我们会发现数据库根本不会访问 DEPT，而且也不需要访问 DEPT，因为 DNAME 列现在是在指向 EMP 表中的行的索引中。为了便

于说明，我们把 EMP 表和 DEPT 表制作得看上去很”大“（以避免 CBO 认为它们很小，以至于选择执行全面扫描，而不是使用索引）：

```
ops$tkyte@ORA10G> begin
  2      dbms_stats.set_table_stats( user, 'EMP',
  3      numrows => 1000000, numblks => 300000 );
  4      dbms_stats.set_table_stats( user, 'DEPT',
  5      numrows => 100000, numblks => 30000 );
  6 end;
  7 /
PL/SQL procedure successfully completed.
```

然后再执行查询：

```
ops$tkyte@ORA10G> set autotrace traceonly explain
ops$tkyte@ORA10G> select count(*)
  2 from emp, dept
  3 where emp.deptno = dept.deptno
  4      and dept.dname = 'SALES'
  5 /
Execution Plan
----------------------------------------------------------
0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11.Card=11.Bytes=11.)
11.0         SORT (AGGREGATE)
2 1             BITMAP CONVERSION (COUNT) (Cost=11.Card=10000 Bytes=130000)
3 2                 BITMAP INDEX (SINGLE VALUE) OF 'EMP_BM_IDX' (INDEX (BITMAP))
```

可以看到，要回答这个特定的问题，我们不必真正去访问 EMP 表或 DEPT 表，答案全部来自索引本身。回答这个问题所需的全部信息都能在索引结构中找到。

另外，我们还能避免访问 DEPT 表，使用 EMP 上的索引就能从 DEPT 合并我们需要的数据，直接访问我们所需的行：

```
ops$tkyte@ORA10G> select emp.*

  2 from emp, dept

  3 where emp.deptno = dept.deptno

  4       and dept.dname = 'SALES'

  5 /

Execution Plan

----------------------------------------------------------

0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6145 Card=10000 Bytes=870000)

11.0          TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=6145 Card=10000 ...

2 1             BITMAP CONVERSION (TO ROWIDS)

3 2                BITMAP INDEX (SINGLE VALUE) OF 'EMP_BM_IDX' (INDEX (BITMAP))
```

位图联结索引确实有一个先决条件。联结条件必须联结到另一个表中的主键或惟一键。在前面的例子中，DEPT.DEPTNO 就是 DEPT 表的主键，而且这个主键必须合适，否则就会出现一个错误：

```
ops$tkyte@ORA10G> create bitmap index emp_bm_idx

  2 on emp( d.dname )

  3 from emp e, dept d

  4 where e.deptno = d.deptno

  5 /

from emp e, dept d

                *

ERROR at line 3:
```

ORA-25954: missing primary key or unique constraint on dimension

11.3.3 位图索引小结

如果还犹豫不定，你可以亲自试一试。向一个表（或一组表）增加位图索引很容易，你可以自己看看位图索引会做些什么。另外，创建位图索引通常比创建 B*树索引快得多。要查看位图索引是否适用于你的环境，最好的办法就是动手实验。经常有人问我：“怎么定义低基数？“对这个问题并没有直截了当的答案。有时，在 100,000 行的表中 3 就是一个低基数；而有时在 11.000,000 行的表中 11.,000 也是一个低基数。低基数并不是说不同的值有几位数字。要想知道你的应用中是否适合使用位图，最好做个实验。一般而言，如果是一个很大的环境，主要是只读操作，并且有大量即席查询，你所需要的可能正是一组位图索引。

11.4 基于函数的索引

基于函数的索引（function-based index）是 Oracle8.11.5 中增加的一种索引。现在这已经是标准版的一个特性，但在 Oracle9i Release 2 之前的版本中，这还只是企业版的一个特性。

利用基于函数的索引，我们能够对计算得出的列建立索引，并在查询中使用这些索引。简而言之，利用这种能力，你可以做很多事情，如执行大小写无关的搜索或排序；根据复杂的公式进行搜索；还可以实现你自己的函数和运算符，然后在此之上执行搜索从而高效地扩展 SQL 语言。

使用基于函数的索引可能有很多原因，其中主要的原因如下：

- 使用索引很容易实现，并能立即提交一个值。
- 可以加快现有应用的速度，而不用修改任何逻辑或查询。

11.4.1 重要的实现细节

在 Oracle9i Release 1 中，要创建和使用基于函数的索引，需要一些初始设置，这与 B*树和位图索引有所不同。

注意 以下内容只适用于 Oracle9i Release 1 及以前的版本。在 Oracle9i Release2 和以后的版本中，基于函数的索引无需初始设置就可以使用。Oracle9i Release 2 的 Oracle SQL Reference 手册在这方面说得不对，它称你需要这些权限，但实际上并不需要。

你必须使用一些系统参数或会话设置，而且必须能创建这些系统参数或会话设置，这需要有以下权限：

- 必须有系统权限 QUERY REWRITE，从而在你自己的模式中的表上创建基于函数的索引。
- 必须有系统权限 GLOBAL QUERY REWRITE，从而在其他模式中的表上创建基于函数的索引。
- 要让优化器使用基于函数的索引，必须设置以下会话或系统变量：QUERY_REWRITE_ENABLED=TRUE 和 QUERY_REWRITE_INTEGRITY=RUSTED。可以在会话级用 ALTER SESSION 来完成设置，也可以在系统级通过 ALTER SYSTEM 来设置，还可以在 init.ora 参数文件中设置。QUERY_REWRITE_ENABLED 允许优化器重写查询来使用基于函数的索引。QUERY_REWRITE_INTEGRITY 告 诉优化器要“相信“程序员标记为确

定性的代码确实是确定性的（下一节会给出一些例子来说明什么是确定性代码，并指出确定性代码的含义）。如果代码实际上不 是确定性的（也就是说，给定相同的输入，它会返回不同的输出），通过索引获取得到的行可能是不正确的。你必须负责确保定义为确定性的函数确实是确定性的。

在所有版本中，以下结论都适用：

- 使用基于代价的优化器（cost-based optimizer，CBO）。在基于函数的索引中，虚拟列（应用了函数的列）只对 CBO 可见，而基于规则的优化器（rule-based optimizer， RBO）不能使用这些虚拟列。RBO 可以利用基于函数的索引中未应用函数的前几列。
- 对于返回 VARCHAR2 或 RAW 类型的用户编写的函数，使用 SUBSTR 来约束其返回值，也可以把 SUBSTR 隐藏在一个视图中（这是推荐的做法）。同样，下一节会给出这样一个例子。

一旦满足前面的条件，基于函数的索引就很容易使用。只需使用 CREATE INDEX 命令来创建索引，优化器会在运行时发现并使用你的索引。

11.4.2 一个简单的基于函数的索引例子

考虑以下例子。我们想在 EMP 表的 ENAME 列上执行一个大小写无关的搜索。在基于函数的索引引入之前，我们可能必须采用另外一种完全不同的方式来做到。可能要为 EMP 表增加一个额外的列，例如名为 UPPER_ENAME 的列。这个列由 INSERT 和 UPDATE 上的一个数据库触发器维护；这个触发器只是设置 NEW.UPPER_NAME := UPPER(:NEW.ENAME)。另外要在这个额外的列上建立索引。但是现在有了基于函数的索引，就根本不用再增加额外的列了。

首先在 SCOTT 模式中创建演示性 EMP 表的一个副本，并在其中增加一些数据：

```
ops$tkyte@ORA10G> create table emp
  2 as
  3 select *
  4 from scott.emp
  5 where 11.0;
Table created.


ops$tkyte@ORA10G> insert into emp
  2      (empno,ename,job,mgr,hiredate,sal,comm,deptno)
  3      select rownum empno,
```

```
  4          initcap(substr(object_name,11.11.)) ename,

  5          substr(object_type,11.9) JOB,

  6          rownum MGR,

  7          created hiredate,

  8          rownum SAL,

  9          rownum COMM,

  11.      (mod(rownum,4)+1)*11. DEPTNO

  11. from all_objects

  11. where rownum < 10000;

9999 rows created.
```

接下来，在 ENAME 列的 UPPER 值上创建一个索引，这就创建了一个大小写无关的索引：

```
ops$tkyte@ORA10G> create index emp_upper_idx on emp(upper(ename));

Index created.
```

最后，前面已经提到了，我们要分析这个表。这是因为，需要利用 CBO 来使用基于函数的索引。在 Oracle 10g 中，从技术上讲这一步不是必须的，因为就会默认使用 CBO，而且动态采样会收集所需的信息，但是最好还是自行收集统计信息。

```
ops$tkyte@ORA10G> begin

  2      dbms_stats.gather_table_stats

  3      (user,'EMP',cascade=>true);

  4 end;

  5 /

PL/SQL procedure successfully completed.
```

现在就在一个列的 UPPER 值上建立了一个索引。执行“大小写无关“查询（如以下查询）的任何应用都能利用这个索引：

```
ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> select *
```

```
  2 from emp

  3 where upper(ename) = 'KING';

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=2 Bytes=92)

11.0        TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=2 Card=2
Bytes=92)

2 1             INDEX (RANGE SCAN) OF 'EMP_UPPER_IDX' (INDEX) (Cost=11.Card=2)
```

这样就能得到索引所能提供的性能提升。在有这个特性之前，EMP 表中的每一行都要扫描、改为大写并进行比较。与之不同，利用 UPPER(ENAME)上的索引，查询为索引提供了常量 KING，然后只对少量数据执行区间扫描，并按 rowid 访问表来得到数据。这是相当快的。

对列上的用户编写的函数建立索引时，能更清楚地看到这种性能提升。Oracle 7.1 开始允许在 SQL 中使用用户编写的函数，所以我们可以做下面的工作：

```
SQL> select my_function(ename)

  2 from emp

  3 where some_other_function(empno) > 11.

  4 /
```

这很棒，因为现在我们能很好地扩展 SQL 语言，可以包括应用特定的函数。不过，遗憾的是，有时前面这个查询的性能并不让人满意。假设 EMP 表中有 11.000 行。查询期间函数 SOME_OTHER_FUNCTION 就会执行 11.000 次，每行执行一次。另外，假设这个函数执行时需要百分之一秒的时间，尽管这个查询相对简单，现在也至少需要 11.秒的时间才能完成。

下面来看一个实际的例子，我们在 PL/SQL 中实现了一个例程，它对 SOUNDEX 例程稍有修改。另外，我们将使用一个包全局变量作为过程中的一个计数器，这样我们就能执行使用了 MY_SOUNDEX 函数的查询，并查看这个函数会被调用多少次：

```
ops$tkyte@ORA10G> create or replace package stats

  2 as

  3       cnt number default 0;

  4 end;
```

```
  5 /

Package created.

ops$tkyte@ORA10G> create or replace
  2       function my_soundex( p_string in varchar2 ) return varchar2
  3      deterministic
  4       as
  5           l_return_string varchar2(6) default substr( p_string, 1, 1 );
  6           l_char varchar2(1);
  7           l_last_digit number default 0;
  8
  9           type vcArray is table of varchar2(11.) index by binary_integer;
 11.       l_code_table vcArray;
 11.
 11. begin
 11.       stats.cnt := stats.cnt+1;
 11.
 11.       l_code_table(1) := 'BPFV';
 11.       l_code_table(2) := 'CSKGJQXZ';
 11.       l_code_table(3) := 'DT';
 11.       l_code_table(4) := 'L';
 11.       l_code_table(5) := 'MN';
 20       l_code_table(6) := 'R';
 21
```

```
22
23      for i in 1 .. length(p_string)
24      loop
25          exit when (length(l_return_string) = 6);
26          l_char := upper(substr( p_string, i, 1 ) );
27
28          for j in 1 .. l_code_table.count
29          loop
30              if (instr(l_code_table(j), l_char ) > 0 AND j <> l_last_digit)
31              then
32                l_return_string := l_return_string || to_char(j,'fm9');
33                l_last_digit := j;
34              end if;
35          end loop;
36      end loop;
37
38       return rpad( l_return_string, 6, '0' );
39 end;
40 /
Function created.
```

注意在这个函数中，我们使用了一个新的关键字 DETERMINISTIC。这就声明了：前面这个函数在给定相同的输入时，总会返回完全相同的输出。要在一个用户编写的函数上创建索引，这个关键字是必要的。我们必须告诉 Oracle 这个函数是确定性的（DETERMINISTIC），而且在给定相同输入的情况下总会返回一致的结果。通过这个关键字，就是在告诉 Oracle：可以相信这个函数，给定相同的输入，不论做多少次调用，它肯定能返回相同的值。如果不是这样，通过索引访问数据时就会得到与全表扫描不同的答案。这种确定性设置表明在有些函数上是不能建立索引的，例如，我们无法在函数

DBMS_RANDOM.RANDOM 上创建索引，因为这是一个随机数生成器。函数 DBMS_RANDOM.RANDOM 的结果不是确定性的；给定相同的输入，我们会得到随机的输出。另一方面，第一个例子中所用的内置 SQL 函数 UPPER 则是确定性的，所有可以在列的 UPPER 值上创建一个索引。

既然有了函数 MY_SOUNDEX，下面来看没有索引时表现如何。在此使用了前面创建的 EMP 表（其中有大约 11.,000 行 ：

```
ops$tkyte@ORA10G> set timing on

ops$tkyte@ORA10G> set autotrace on explain

ops$tkyte@ORA10G> select ename, hiredate

  2 from emp

  3 where my_soundex(ename) = my_soundex('Kings')

  4 /



ENAME             HIREDATE

----------        ---------

Ku$_Chunk_    11.-AUG-04

Ku$_Chunk_    11.-AUG-04

Elapsed: 00:00:01.07



Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=32 Card=100 Bytes=1900)

11.0        TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=32 Card=100 Bytes=1900)

ops$tkyte@ORA10G> set autotrace off

ops$tkyte@ORA10G> set timing off

ops$tkyte@ORA10G> set serveroutput on
```

```
ops$tkyte@ORA10G> exec dbms_output.put_line( stats.cnt );

19998

PL/SQL procedure successfully completed.
```

可以看到这个查询花了一秒多的时间执行，而且必须执行全表扫描。函数 MY_SOUNDEX 被调用了几乎 20,000 次（根据计数器得出），每行要调用两次。

下面来看对这个函数建立索引后，速度会有怎样的提高。首先如下创建索引：

```
ops$tkyte@ORA10G> create index emp_soundex_idx on

  2 emp( substr(my_soundex(ename),11.6) )

  3 /

Index created.
```

在这个 CREATE INDEX 命令中，有意思的是在此使用了 SUBSTR 函数。这是因为，我们在对一个返回串的函数建索引。如果对一个返回数字或日期的函数建索引，就没有必须使用这个 SUBSTR。如果用户编写的函数返回一个串，之所以要对这样一个函数使用 SUBSTR，原因是这种函数会返回 VARCHAR2(4000)类型。这就太大了，无法建立索引，索引条目必须能在块大小的 3/4 中放得下。如果尝试对这么大的返回值建索引，就会收到以下错误（在一个块大小为 4KB 的表空间中）：

```
ops$tkyte@ORA10G> create index emp_soundex_idx on

  2 emp( my_soundex(ename) ) tablespace ts4k;

emp( my_soundex(ename) ) tablespace ts4k

                         *

ERROR at line 2:

ORA-01450: maximum key length (3118) exceeded
```

这并不是说索引中确实包含那么大的键，而是说对数据库而言键可以有这么大。但是数据库“看得懂“SUBSTR。它看到 SUBSTR 的输入参数为 1 和 6，知道最大的返回值是 6 个字符；因此，它允许创建索引。你很可能会遇到这种大小问题，特别是对于串联索引。以下是一个例子，在此表空间的块大小为 8KB：

```
ops$tkyte@ORA10G> create index emp_soundex_idx on

  2 emp( my_soundex(ename), my_soundex(job) );

emp( my_soundex(ename), my_soundex(job) )
```

```
                                                *

ERROR at line 2:

ORA-01450: maximum key length (6398) exceeded
```

在此，数据库认为最大的键为 6,398，所以 CREATE 再一次失败。因此，如果用户编写的函数要返回一个串，要对这样一个函数建立索引，应当在 CREATE INDEX 语句中对返回类型有所限制。在这个例子中，由于知道 MY_SOUNDEX 最多返回 6 个字符，所以取前 6 个字符作为字串。

有了这个索引后，现在来测试表的性能。我们想监视索引对 INSERT 的影响，并观察它能怎样加快 SELECT 的执行速度。在没有索引的测试用例中，我们的查询用了 1 秒多的时间，如果在插入期间运行 SQL_TRACE 和 TKPROF，会观察到：在没有索引的情况下，插入 9,999 条记录耗时约 0.5 秒：

```
insert into emp NO_INDEX

    (empno,ename,job,mgr,hiredate,sal,comm,deptno)

    select rownum empno,

           initcap(substr(object_name,11.11.)) ename,

           substr(object_type,11.9) JOB,

           rownum MGR,

           created hiredate,

           rownum SAL,

           rownum COMM,

           (mod(rownum,4)+1)*11. DEPTNO

    from all_objects

    where rownum < 10000


call     count       cpu    elapsed       disk      query    current        rows

-------  ------  --------  ---------- ----------  ----------  ----------  ----------

Parse        1      0.03          0.06          0           0           0           0
```

```
Execute      1      0.46       0.43          0      15439        948        9999
Fetch        0      0.00       0.00          0          0          0           0
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        2      0.49       0.50          0      15439        948        9999
```

但是如果有索引，则需要大约 11.2 秒：

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.03       0.04          0          0          0           0
Execute      1     11.11.     11.11.          2      15650       7432        9999
Fetch        0      0.00       0.00          0          0          0           0
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        2     11.11.     11.11.          2      15650       7432        9999
```

原因在于管理 MY_SOUNDEX 函数上的新索引会带来开销，这一方面是因为只要有索引就存在相应的性能开销（任何类型的索引都会影响插入的性能）；另一方面是因为这个索引必须把一个存储过程调用 9,999 次。

下面测试这个查询，只需再次运行查询：

```
ops$tkyte@ORA10G> REM reset our counter
ops$tkyte@ORA10G> exec stats.cnt := 0
PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> set timing on
ops$tkyte@ORA10G> set autotrace on explain
ops$tkyte@ORA10G> select ename, hiredate
  2 from emp
  3 where substr(my_soundex(ename),11.6) = my_soundex('Kings')
```

```
  4 /


ENAME HIREDATE

---------- ---------

Ku$_Chunk_ 11.-AUG-04

Ku$_Chunk_ 11.-AUG-04

Elapsed: 00:00:00.02


Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=11.Bytes=11.)

11.0          TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=2
Card=11.Bytes=11.)

2  1              INDEX (RANGE SCAN) OF 'EMP_SOUNDEX_IDX' (INDEX)
(Cost=11.Card=35)



ops$tkyte@ORA10G> set autotrace off

ops$tkyte@ORA10G> set timing off

ops$tkyte@ORA10G> set serveroutput on

ops$tkyte@ORA10G> exec dbms_output.put_line( stats.cnt );

2


PL/SQL procedure successfully completed.
```

如果对这两个例子做个比较（无索引和有索引），会发现插入受到的影响是：其运行时间是原来的两倍还多。不过，选择操作则不同，原来需要 1 秒多的时间，现在几乎是“立即”完成。这里的要点是：

- 有索引时，插入 9,999 条记录需要大约两倍多的时间。对用户编程的函数建立索引绝对会影响插入（和一些更新）的性能。当然，你应该意识到任何索引都会影响性能。例如，我做了一个简单的测试（没有 MY_SOUNDEX 函数），其中只是对 ENAME 列本身加索引。这就导致 INSERT 花了大约 1 秒的时间执行，因此整个开销并不能全部归咎于 PL/SQL 函数。由于大多数应用都只是插入和更新单个的条目，而且插入每一行只会花不到 11.11.,000 秒的时间，所以在一个经典的应用中，你可能注意不到这种开销。由于我们一次只插入一行，代价就只是在列上执行一次函数，而不是像查询数据时那样可能执行数千次。

- 尽管插入的运行速度慢了两倍多，但查询运行的速度却快了几倍。它只是把 MY_SOUNDEX 函数计算了几次，而不是几乎 20,000 次。这里有索引和无索引时查询性能的差异相当显著。另外，表越大，全面扫描查询执行的时间就会越来越长。基于索引的查询则不同，随着表的增大，基于索引的查询总是有几乎相同的执行性能。

- 在我们的查询中必须使用 SUBSTR。这好像不太好，不如只是写 WHERE MY_SOUNDEX(ename)=MY_SOUNDEX(‘King’)那么直接，但是这个问题可以很容易地得到解决，稍后就会看到。

因此，插入会受到影响，但是查询运行得快得多。尽管插入/更新性能稍有下降，但是回报是丰厚的。另外，如果从不更新 MY_SOUNDEX 函数调用中涉及到的列，更新就根本没有开销（仅当修改了 ENAME 列而且其值确实有改变时，才会调用 MY_SOUNDEX）。

现在来看如何让查询不使用 SUBSTR 函数调用。使用 SUBSTR 调用可能很容易出错，最终用户必须知道要从第 1 个字符起取 6 个字符作为子串（SUBSTR）。如果使用的子串大小不同，就不会使用这个索引。另外，我们可能希望在服务器中控制要索引的字节数。因此我们可以重新实现 MY_SOUNDEX 函数，如果愿意还可以索引 7 个字节而不是 6 个。利用一个视图就能非常简单地隐藏 SUBSTR，如下所示：

```
ops$tkyte@ORA10G> create or replace view emp_v
  2 as
  3 select ename, substr(my_soundex(ename),11.6) ename_soundex, hiredate
  4 from emp
  5 /

View created.



ops$tkyte@ORA10G> exec stats.cnt := 0;

PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10G> set timing on
ops$tkyte@ORA10G> select ename, hiredate
  2 from emp_v
  3 where ename_soundex = my_soundex('Kings')
  4 /
ENAME          HIREDATE
----------     ---------
Ku$_Chunk_     11.-AUG-04
Ku$_Chunk_     11.-AUG-04

Elapsed: 00:00:00.03
ops$tkyte@ORA10G> set timing off
ops$tkyte@ORA10G> exec dbms_output.put_line( stats.cnt )
2
PL/SQL procedure successfully completed.
```

可以看到这个查询计划与对基表的查询计划是一样的。这里所做的只是将SUBSTR(F(X)),11.6)隐藏在视图本身中。优化器会识别出这个虚拟列实际上是加了索引的列，并采取“正确”的行动。我们能看到同样的性能提升和同样的查询计划。使用这个视图与使用基表是一样的，甚至还更好一些，因为它隐藏了复杂性，并允许我们以后改变 SUBSTR 的大小。

11.4.3 只对部分行建立索引

基于函数的索引除了对使用内置函数（如 UPPER、LOWER 等）的查询显然有帮助之外，还可以用来有选择地只是对表中的某些行建立索引。稍后会讨论，B*树索引对于完成为 NULL 的键没有相应的条目。也就是说，如果在表 T 上有一个索引 I：

```
Create index I on t(a,b);
```

而且行中 A 和 B 都为 NULL，索引结构中就没有相应的条目。如果只对表中的某些行建立索引，这就能用得上。

考虑有一个很大的表，其中有一个 NOT NULL 列，名为 PROCESSED_FLAG，它有两个可取值：Y 或 N，默认值为 N。增加新行时，这个值为 N，指示这一行未得到处理，等到处理了这一行后，则会将其更新为 Y 来指示已处理。我们可能想对这个列建立索引，从而能快速地获取值为 N 的记录，但是这里有数百万行，而且几乎所有行的值都为 Y。所得到的 B*树索引将会很大，如果我们把值从 N 更新为 Y，维护这样一个大索引的开销也相当高。这个表听起来很适合采用位图索引（毕竟基数很低！），但这是一个事务性系统，可能有很多人在同时插入记录（新记录的“是否处理”列设置为 N），前面讨论过，位图索引不适用于并发修改。如果考虑到这个表中会不断地将 N 更新为 Y，那位图就更不合适了，根本不应考虑，因为这个过程会完全串行化。

所以，我们真正想做的是，只对感兴趣的记录建立索引（即该列值为 N 的记录）。我们会介绍如何利用基于函数的索引来做到这一点，但是在此之前，先来看如果只是一个常规索引会发生什么。使用本书最前面“环境设置”一节中描述的标准 BIG_TABLE 脚本，下面更新 TEMPORARY 列，在此将 Y 变成 N，以及 N 变成 Y：

```
ops$tkyte@ORA10G> update big_table set temporary = decode(temporary,'N','Y','N');

1000000 rows updated.
```

现在检查 Y 与 N 地比例：

```
ops$tkyte@ORA10G> select temporary, cnt,
  2      round( (ratio_to_report(cnt) over ()) * 100, 2 ) rtr
  3 from (
  4          select temporary, count(*) cnt
  5          from big_table
  6          group by temporary
  7 )
  8 /


T            CNT              RTR
-         ----------          ----------
N           1779               .11.
Y         998221             99.82
```

可以看到，在表的 11.000,000 条记录中，只有 0.2%的数据应当加索引。如果在 TEMPORARY 列上使用传统索引（相对于这个例子中 PROCESSED_FLAG 列的角色），会

发现这个索引有 11.000,000 个条目，占用了超过 14MB 的空间，其高度为 3：

```
ops$tkyte@ORA10G> create index processed_flag_idx

  2 on big_table(temporary);

Index created.



ops$tkyte@ORA10G> analyze index processed_flag_idx

  2 validate structure;

Index analyzed.



ops$tkyte@ORA10G> select name, btree_space, lf_rows, height

  2 from index_stats;

NAME                           BTREE_SPACE     LF_ROWS     HEIGHT

------------------------------                 -----------         ----------       ----------

PROCESSED_FLAG_IDX          14528892       1000000              3
```

通过这个索引获取任何数据都会带来 3 个 I/O 才能达到叶子块。这个索引不仅很“宽”，还很“高”。要得到第一个未处理的记录，必须至少执行 4 个 I/O（其中 3 个是对索引的 I/O，另外一个是对表的 I/O）。

怎么改变这种情况呢？我们要让索引更小一些，而且要更易维护（更新期间的运行时开销更少）。采用基于函数的索引，我们可以编写一个函数，如果不想对某个给定行加索引，则这个函数就返回 NULL；而对想加索引的行则返回一个非 NULL 值。例如，由于我们只对列值为 N 的记录感兴趣，所以只对这些记录加索引：

```
ops$tkyte@ORA10G> drop index processed_flag_idx;

Index dropped.



ops$tkyte@ORA10G> create index processed_flag_idx

  2 on big_table( case temporary when 'N' then 'N' end );

Index created.
```

```
ops$tkyte@ORA10G> analyze index processed_flag_idx
  2 validate structure;
Index analyzed.

ops$tkyte@ORA10G> select name, btree_space, lf_rows, height
  2 from index_stats;
NAME                          BTREE_SPACE     LF_ROWS     HEIGHT
----------------------------- ----------- ---------- ----------
PROCESSED_FLAG_IDX            40012           1779           2
```

这就有很大不同，这个索引只有大约 40KB，而不是 11..5MB。高度也有所降低。与前面那个更高的索引相比，使用这个索引能少执行一个 I/O。

11.4.4 实现有选择的惟一性

要利用基于函数的索引，还有一个有用的技术，这就是使用这种索引来保证某种复杂的约束。例如，假设有一个带版本信息的表，如项目表。项目有两种状态：要么为 ACTIVE，要么为 INACTIVE。需要保证以下规则：“活动的项目必须有一个惟一名；而不活动的项目无此要求。”也就是说，只有一个活动的“项目 X”，但是如果你愿意，可以有多个名为 X 的不活动项目。

开发人员了解到这个需求时，第一反应往往是：“我们只需运行一个查询来查看是否有活动项目 X，如果没有，就可以创建一个活动项目 X。”如果你读过第 7 章（介绍并发控制和多版本的内容），就会知道，这种简单的实现在多用户环境中是不可行的。如果两个人想同时创建一个新的活动项目 X，他们都会成功。我们需要将项目 X 的创建串行化，但是对此惟一的做法是锁住这个项目表（这样做并发性就不太好了），或者使用一个基于函数的索引，让数据库为我们做这个工作。

由于可以在函数上创建索引，而且 B*树索引中对于完全为 NULL 的行没有相应的条目，另外我们可以创建一个 UNIQUE 索引，基于这几点，可以很容易做到：

```
Create unique index active_projects_must_be_unique
On projects ( case when status = 'ACTIVE' then name end );
```

这就行了。状态（status）列是 ACTIVE 时，NAME 列将建立惟一的索引。如果试图创建同名的活动项目，就会被检测到，而且这根本不会影响对这个表的并发访问。

11.4.5 关于 CASE 的警告

某些 Oracle 版本中有一个 bug，其中基于函数的索引中引用的函数会以某种方式被重写，以至于索引无法被透明地使用。例如，前面的 CASE 语句

```
Case when temporary = 'N' then 'N' end
```

会悄悄地重写为以下更高效的语句：

```
CASE "TEMPORARY" WHEN 'N' THEN 'N' END
```

但是这个函数与我们创建的那个函数不再匹配，所以查询无法使用此函数。如果在 11..11.0.3 中执行这个简单的测试用例，然后再在 11..11.0.4（该版本修正了这个 bug）中执行它，结果如下（在 11..11.0.3 中）：

```
ops$tkyte@ORA10GR1> create table t ( x int );

Table created.



ops$tkyte@ORA10GR1> create index t_idx on

  2 t( case when x = 42 then 11.end );

Index created.



ops$tkyte@ORA10GR1> set autotrace traceonly explain



ops$tkyte@ORA10GR1> select /*+ index( t t_idx ) */ *

  2 from t

  3 where (case when x = 42 then 11.end ) = 1;



Execution Plan

----------------------------------------------------------

0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=11.Bytes=11.)

11.0        TABLE ACCESS (FULL) OF 'T' (TABLE) (Cost=2 Card=11.Bytes=11.)
```

看上去，基于函数的索引不仅不会工作，而且不可用。但是这个 FBI（基于函数的索引）

其实是可用的，只不过这里底层函数被重写了，我们可以查看视图 USER_IND_EXPRESSIONS 来看看 Oracle 是如何重写它的，从而验证这一点：

```
ops$tkyte@ORA10GR1> select column_expression

  2 from user_ind_expressions

  3 where index_name = 'T_IDX';

COLUMN_EXPRESSION

--------------------------------------------------------------------------------

CASE "X" WHEN 42 THEN 11.END
```

在 Oracle11..11.0.4 中，基于函数的索引中也会发生重写，但是索引会使用重写后的函数：

```
ops$tkyte@ORA10G> set autotrace traceonly explain

ops$tkyte@ORA10G> select /*+ index( t t_idx ) */ *

  2 from t

  3 where (case when x = 42 then 11.end ) = 1;



Execution Plan

----------------------------------------------------------

0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11.Card=11.Bytes=11.)

11.0           TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE)
(Cost=11.Card=11.Bytes=11.)

2 1            INDEX (RANGE SCAN) OF 'T_IDX' (INDEX) (Cost=11.Card=1)
```

这是因为数据库选择不仅重写了 CREATE INDEX 中的函数，还重写了查询本身使用的函数，因此二者是匹配的。

在以前的版本中，对此的解决办法有以下几种：

- 使用 DECODE 而不是 CASE，因为 DECODE 不会被重写，即所谓的“所见即所得”。
- 使用最先搜索到的 CASE 语法（预计到可能会发生的优化）。

但是，倘若优化器没有使用你的基于函数的索引，而且你对此无法做出解释，不知道为什么没有使用你的函数，在这种情况下，就可以检查 USER_IND_EXPRESSIONS 视图，

验证使用的函数是否正确。

11.4.6 关于 ORA-01743 的警告

对于基于函数的索引，我注意到这样一个奇怪的现像，如果你要在内置函数 TO_DATE 上创建一个索引，某些情况下并不能成功创建，例如：

```
ops$tkyte@ORA10GR1> create table t ( year varchar2(4) );

Table created.



ops$tkyte@ORA10GR1> create index t_idx on t( to_date(year,'YYYY') );

create index t_idx on t( to_date(year,'YYYY') )

                                *

ERROR at line 1:

ORA-01743: only pure functions can be indexed
```

这看上去很奇怪，因为有时使用 TO_DATE 创建索引确实是可以的，例如：

```
ops$tkyte@ORA10GR1> create index t_idx on t( to_date('01'||year,'MMYYYY') );

Index created.
```

相关的错误信息也很模糊，让人摸不着头脑：

```
ops$tkyte@ORA10GR1> !oerr ora 1743

01743, 00000, "only pure functions can be indexed"

// *Cause: The indexed function uses SYSDATE or the user environment.

// *Action: PL/SQL functions must be pure (RNDS, RNPS, WNDS, WNPS). SQL

// expressions must not use SYSDATE, USER, USERENV(), or anything

// else dependent on the session state. NLS-dependent functions

// are OK.
```

我们并没有使用 SYSDATE；也没有使用“用户环境”（难道我们用了吗？）。这里并没有使用 PL/SQL 函数，而且没有涉及任何与会话状态有关的方面。问题只是在于我们使用的格式：YYYY。给定完全相同的输入，这种格式可能返回不同的答案，这取决于调用的函数时输入的月份。例如，对于 5 月的每个时间：

```
ops$tkyte@ORA10GR1> select to_char( to_date('2005','YYYY'),
  2                                    'DD-Mon-YYYY HH24:MI:SS' )
  3 from dual;
TO_CHAR(TO_DATE('200
--------------------
01-May-2005 00:00:00
```

YYYY 格式会返回 5 月 1 日，在 6 月它会返回 6 月 1 日，以此类推。这就说明，如果用到 YYYY，TO_DATE 就不是确定性的！这这是无法创建索引的原因：只有在创建一行（或插入/更新一行）的多月它能正确工作。所以，这个错误确实归根于用户环境，其中包含当前日期本身。

要在一行基于函数的索引中使用 TO_DATE，必须使用一种无歧义的确定性日期格式，而不论当前是哪一天。

11.4.7 基于函数的索引小结

基 于函数的索引很容易使用和实现，他们能提供立即值。可以用基于函数的索引来加快现有应用的速度，而不用修改应用中的任何逻辑或查询。通过使用基于函数的索 引，可以观察到性能会呈数量级地增长。使用这种索引能提前计算出复杂的值，而无需使用触发器。另外，如果在基于函数的索引中物化表达式，优化器就能更准确 度估计出选择性。可以使用基于函数的索引有选择地只对感兴趣的几行建立索引（如前面关于 PROCESSED_FLAG 的例子所示）。实际上，使用这种就是可以对 WHERE 子句加索引。最后，我们研究了如何使用基于函数的索引来实现某种完整性约束：有选择的惟一性（例如，“每个条件成立时字段 X、Y 和 Z 必须惟一”）。

基 于函数的索引会影响插入和更新的性能。不论这一点对你是否重要，都必须有所考虑。如果你总是插入数据，而不经常查询，基于函数的索引可能对你并不适用。另 一方面，要记住，一般插入时都是一次插入一行，查询却会完成数千次。所以插入方面的性能下降（最终用户可能根本注意不到）能换来查询速度数千倍的提高。一 般来说，在这种情况下利远大于弊。

11.5 应用域索引

应用域索引（application domain index）即 Oracle 所谓的可扩展索引（extensible indexing）。利用应用域索引，你可以创建自己的索引结构，使之像 Oracle 提供的索引一样工作。有人使用你的索引类型发出一个 CREATE INDEX 语句时，Oracle 会运行你的代码来生成这个索引。如果有人分析索引来计算统计信息，Oracle 会执行你的代码来生成统计信息（采用你要求的存储格式）。Oracle 解析查询并开发查询计划时，如果查询计划中可能使用你的索引，Oracle 会问你：这个函数的计算不同的计划时会有怎样的开销。简单地说，利用应用域索引，你能实现数据库中原本没有的一个新的索引类型。例如，如果你开发一个软件来分析数据库中存储的图像，而且生成了关于图像的信息（如图像中的颜色），就可以创建你自己的图像（image）索引。向数据库中增加图象时，会调用你的代码，从图像中抽取颜

色，并将其存储在某个地方（你想存储图像索引的任何地方）。查询时，用户请求所有“蓝色图像”时，Oracle 就会在合适的时候从索引提供答案。

对此最好的例子是 Oracle 自己的文本索引（text index）。这个索引用于对大量的文本项提供关键字搜索。可以如下创建一个简单的文本索引：

```
ops$tkyte@ORA10G> create index myindex on mytable(docs)
  2 indextype is ctxsys.context
  3 /

Index created.
```

这个索引的创建者向 SQL 语言中引入了一些文本运算符，接下来使用这些文本运算符：

```
select * from mytable where contains( docs, 'some words' ) > 0;
```

它甚至能对如下的命令做出响应：

```
ops$tkyte@ORA10GR1> begin
  2 dbms_stats.gather_index_stats( user, 'MYINDEX' );
  3 end;
  4 /

PL/SQL procedure successfully completed.
```

它会与优化器合作，在运行时确定使用文本索引（而不是其他某个索引或前面扫描）的相对开销。有意思的是，任何人（包括你和我）都可以开发这样一个索引。文本索引的实现无需你了解“内部核心知识”。这是使用专用的 API 完成的，这些 API 有文档说明而且已经公开提供。Oracle 数据库内核并不关心文本索引如果存储（对于创建的每个索引，API 会把它存储在多个物理数据库表中）。Oracle 也不知道插入新行时会做怎样的处理。Oracle 文本实际上是建立在数据库之上的一个应用，但采用了一种完全集成的方式。对于你和我来说，这看上去就像是如何其他 Oracle 数据库内核函数一样，但事实上它并不是内核函数。

我个人认为，没有必要去构建一个标新立异的索引结构类型，在我看来，这种特定的特性大多由第三方解决方案提供者使用（他们有一些创新性的索引技术）。

我认为，应用域索引最有意思的一点是：利用应用域索引，这就允许其他人提供新的索引技术，而我可以在自己的应用中使用这些技术。大多数人从来都没有用过这种特定的 API 来构建新的索引类型，但是我们大多都用到过某种非内置的新索引类型。我参与的几乎每一个应用都有一些与之相关的文本（text）、待处理的 XML 或者要存储和分类的图像（image）。这些功能通过一个 interMedia 功能集（利用了应用域索引特性来实现）就能提供。随着时间的推移，可用的索引类型越来越多。我们将在下一章更深入地分析文本索引。

11.6 关于索引的常见问题和神话

在本书的引言中曾经说过，我回答过大量关于Oracle的问题。我就是Oracle Magazine上

“Ask Tom”专栏和http://asktom.oracle.com上的Tom，在这个专栏和网站上我一直在回答大家提出的关于Oracle数据库和工具的问题。根据我的经验，其中关于索引的问题最多。这一节中，我将回答问得最多的一些问题。有些答案就像是常识一样，很直接；但是有些答案可能会让你很诧异。可以这么说，关于索引存在的许多神话和误解。

11.6.1 视图能使用索引吗？

与这个问题相关的另一个问题是：“能对视图加索引吗？”视图实际上就是一个存储查询（stored query）。Oracle 会把查询中访问视图的有关文本代之以视图定义本身。视图只是为了方便最终用户或程序员，优化器还是会对基表使用查询。使用视图时，完全可以考虑使用为基表编写的查询中所能使用的所有索引。“对视图建立索引”实际上就是对基表建立索引。

11.6.2 Null 和索引能协作吗？

B*树索引（除了聚簇 B*树索引这个特例之外）不会存储完全为 null 的条目，而位图好聚簇索引则不同。这个副作用可能会带来一些混淆，但是如果你理解了不存储完全为 null 的键是什么含义，就能很好地利用这一点。

要看到不存储 null 值所带来的影响，请考虑下面这个例子：

```
ops$tkyte@ORA10GR1> create table t ( x int, y int );

Table created.



ops$tkyte@ORA10GR1> create unique index t_idx on t(x,y);

Index created.



ops$tkyte@ORA10GR1> insert into t values ( 1, 1 );

11.row created.



ops$tkyte@ORA10GR1> insert into t values ( 1, NULL );

11.row created.



ops$tkyte@ORA10GR1> insert into t values ( NULL, 1 );
```

```
11.row created.



ops$tkyte@ORA10GR1> insert into t values ( NULL, NULL );

11.row created.



ops$tkyte@ORA10GR1> analyze index t_idx validate structure;

Index analyzed.



ops$tkyte@ORA10GR1> select name, lf_rows from index_stats;

NAME                           LF_ROWS

------------------------------ ----------

T_IDX                               3
```

这个表有 4 行，而索引只有 3 行。前三行（索引键元素中至少有一个不为 null）都在索引中。最后一行的索引键是（NULL,NULL）,所以这一行不在索引中。倘若索引是一个惟一索引（如上所示），这就是可能产生混淆的一种情况。考虑以下 3 个 INSERT 语句的作用：

```
ops$tkyte@ORA10GR1> insert into t values ( NULL, NULL );

11.row created.



ops$tkyte@ORA10GR1> insert into t values ( NULL, 1 );

insert into t values ( NULL, 1 )

*

ERROR at line 1:

ORA-00001: unique constraint (OPS$TKYTE.T_IDX) violated



ops$tkyte@ORA10GR1> insert into t values ( 1, NULL );
```

```
insert into t values ( 1, NULL )

*

ERROR at line 1:

ORA-00001: unique constraint (OPS$TKYTE.T_IDX) violated
```

这里并不认为新的（NULL,NULL）行与原来的（NULL,NULL）行相同：

```
ops$tkyte@ORA10GR1> select x, y, count(*)

  2 from t

  3 group by x,y

  4 having count(*) > 1;

X          Y          COUNT(*)

---------- ---------- ----------

                               2
```

看上去好像不可能的，如果考虑到所有 null 条目，这就说明我们的惟一键并不惟一。事实上，在 Oracle 中，考虑惟一性时（NULL,NULL）与（NULL,NULL）并不相同，这是 SQL 标准要求的。不过对于聚集来说（NULL,NULL）和（NULL,NULL）则认为是相同的。两个（NULL,NULL）在比较时并不相同，但是对 GROUP BY 子句来说却是一样的。所以应当考虑到：每个惟一约束应该至少有一个确实惟一的 NOT NULL 列。

关于索引和 null 值还会提出这样一个疑问是：“为什么我的查询不使用索引？”下面是一个有问题的查询：

```
select * from T where x is null;
```

这个查询无法使用我们刚才创建的索引，（NULL,NULL）行并不在索引中，因此使用索引的话实际上会返回错误的答案。只有当索引键中至少有一个列定义为 NOT NULL 时查询才会使用索引。例如，以下显示了 Oracle 会对 X IS NULL 谓词使用索引（如果索引的索引键最前面是 X 列，而且索引中其他列中至少有一列是 NOT NULL）：

```
ops$tkyte@ORA10GR1> create table t ( x int, y int NOT NULL );

Table created.



ops$tkyte@ORA10GR1> create unique index t_idx on t(x,y);

Index created.
```

```
ops$tkyte@ORA10GR1> insert into t values ( 1, 1 );

11.row created.



ops$tkyte@ORA10GR1> insert into t values ( NULL, 1 );

11.row created.



ops$tkyte@ORA10GR1> begin

  2      dbms_stats.gather_table_stats(user,'T');

  3 end;

  4 /

PL/SQL procedure successfully completed.
```

再来查询这个表，会发现：

```
ops$tkyte@ORA10GR1> set autotrace on

ops$tkyte@ORA10GR1> select * from t where x is null;

X                    Y

----------      ----------

                     1

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11.Card=11.Bytes=5)

11.0            INDEX (RANGE SCAN) OF 'T_IDX' (INDEX (UNIQUE))
(Cost=11.Card=11.Bytes=5)
```

前面我说过，B*树索引中不存储完全为 null 的条目，而且你可以充分利用这一点，以上就展示了应当如何加以利用。假设你有一个表，其中每一列只有两个可取值。这些值分布得很不均匀，例如，90%以上的行（多数行）都取某个值，而另外不到 11.%的行（少数行）

取另外一个值。可以有效地对这个列建立索引，来快速访问那些少数行。如果你想使用一个索引访问少数行，同时又想通过全面扫描来访问多数行，另外还想节省空间，这个特性就很有用。解决方案是：对多数行使用 null，而对少数行使用你希望的任何值；或者如前所示，使用一个基于函数的索引，只索引函数的非 null 返回值。

既然知道了 B*树如何处理 null 值，所以可以充分利用这一点，并预防在全都允许有 null 值的列上建立惟一约束（当心这种情况下可能有多个全 null 的行）。

11.6.3 外键是否应该加索引？

外键是否应该加索引，这个问题经常被问到。我们在第 6 章讨论死锁时谈到过这个话题。在第 6 章中，我指出，外键未加索引是我所遇到的导致死锁的最主要的原因；这是因为，无论是更新父表主键，或者删除一个父记录，都会在子表中加一个表锁（在这条语句完成前，不允许对子表做任何修改）。这就会不必要地锁定更多的行，而影响并发性。人们在使用能自动生成 SQL 来修改表的某个工具时，就经常遇到这种问题。这样的工具会生成一个更新语句，它将更新表中的每一列，而不论这个值是否被 UPDATE 语句修改。这就会导致更新主键（即使主键值其实从未改变过）。例如，Oracle Forms 就会默认地这样做，除非你告诉它只把修改过的列发送给数据库。除了可能遇到表锁问题之外，在以下情况下，外键未加索引也表现得很糟糕：

- 如果有一个 ON DELETE CASCADE，而且没有对子表建索引。例如，EMP 是 DEPT 的子表。DELETE FROM DEPT WHERE DEPTNO = 11.会级联至 EMP。如果 EMP 中的 DEPTNO 没有加索引，就会导致对 EMP 执行一个全表扫描。这种完全扫描可能是不必要的，而且如果从父表删除了多行，对于删除的每一个父行，都会把子表扫描一次。
- 从父表查询子表时。还是考虑 EMP/DEPT 的例子。在 DEPTNO 上下文查询 EMP 表相当常见。如果频繁地执行以下查询来生成一个报告或某个结果：

```
select *

from dept, emp

where emp.deptno = dept.deptno

   and dept.dname = :X;
```

你会发现，如果没有索引会使查询减慢。由于同样的原因，我在第 11.章曾建议对嵌套表中的 NESTED_COLUMN_ID 加索引。嵌套表的隐藏列 NESTED_COLUMN_ID 实际上就是一个外键。

那么，什么时候不需要对外键加索引呢？一般来说，如果满足以下条件则可如此：

- 未删除父表中的行。
- 不论是有意还是无意（如通过一个工具），总之未更新父表的惟一/主键值。
- 不论从父表联结到子表，或者更一般地讲，外键列不支持子表的一个重要的访问途径，而且你在谓词中没有使用这些外键列从子表中选择数据（如 DEPT 到 EMP）。

如果满足上述所有 3 个条件，就完全可以不加索引，也就是说，对外键加索引是不必要的，还会减慢子表上 DML 操作的速度。如果满足了其中某个条件，就要当心不加索引的后果。

另外说一句，如果你认为某个子表会由于外键为加索引而被锁住，而且希望证明这一点（或者一般来说，你想避免这种情况），可以发出以下命令：

```
ALTER TABLE <child table name> DISABLE TABLE LOCK;
```

现在，对父表的可能导致表锁的任何 UPDATE 或 DELETE 都会接收到以下错误：

```
ERROR at line 1:

ORA-00069: cannot acquire lock -- table locks disabled for <child table name>
```

这有助于跟踪到有问题的代码段，你以为它没有做某件事（比如，你认为并没有对父表的主键执行 UPDATE 或 DELETE），但实际上事与愿违，通过以上命令，最终用户就会立即向你反馈这个错误。

11.6.4 为什么没有使用我的索引？

对此有很多可能的原因。在这一节中，我们会查看其中一些最常见的原因。

1. 情况 1

我们在使用一个 B*树索引，而且谓词中没有使用索引的最前列。如果是这种情况，可以假设有一个表 T，在 T(X,Y)上有一个索引。我们要做以下查询：SELECT * FROM T WHERE Y = 5。此时，优化器就不打算使用 T(x,y)上的索引，因为谓词中不涉及 X 列。在这种情况下，倘若使用索引，可能就必须查看每一个索引条目（稍后我们会讨论一种索引跳跃式扫描，这是一种例外情况），而优化器通常更倾向于 T 对做一个全表扫描。但这并不完全排除使用索引。如果查询是 SELECT X, Y FROM T WHERE Y = 5，优化器就会注意到，它不必全面扫描表来得到 X 或 Y（X 和 Y 都在索引中），对索引本身做一个快速的全面扫描会更合适，因为这个索引一般比底层表小得多。还要注意，仅 CBO 能使用这个访问路径。

另一种情况下 CBO 也会使用 T(x,y)上的索引，这就是索引跳跃式扫描。当且仅当索引的最前列（在上一个例子中，最前列就是 Y）只有很少的几个不同值，而且优化器了解这一点，跳跃式扫描（skip scan）就能很好地发挥作用。例如，考虑（GENDER, EMPNO）上的一个索引，其中 GENDER 可取值有 M 和 F，而且 EMPNO 是惟一的。对于以下查询：

```
select * from t where empno = 5;
```

可以考虑使用 T 上的那个索引采用跳跃式扫描方法来满足这个查询，这说明从概念上讲这个查询会如下处理：

```
select * from t where GENDER='M' and empno = 5

UNION ALL

select * from t where GENDER='F' and empno = 5;
```

它会跳跃式地扫描索引，以为这是两个索引：一个对于值 M，另一个对应值 F。在查询计划中可以很容易地看出这一点。我们将建立一个表，其中有一个二值的列，并在这个列上建立索引：

```
ops$tkyte@ORA10GR1> create table t
  2 as
  3 select decode(mod(rownum,2), 0, 'M', 'F' ) gender, all_objects.*
  4 from all_objects
  5 /
Table created.


ops$tkyte@ORA10GR1> create index t_idx on t(gender,object_id)
  2 /
Index created.


ops$tkyte@ORA10GR1> begin
  2     dbms_stats.gather_table_stats
  3     ( user, 'T', cascade=>true );
  4 end;
  5 /
PL/SQL procedure successfully completed.
```

做以下查询时，可以看到结果如下：

```
ops$tkyte@ORA10GR1> set autotrace traceonly explain
ops$tkyte@ORA10GR1> select * from t t1 where object_id = 42;


Execution Plan
----------------------------------------------------------
0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=11.Bytes=95)
```

```
11.0          TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE) (Cost=4
Card=11.Bytes=95)

2 1           INDEX (SKIP SCAN) OF 'T_IDX' (INDEX) (Cost=3 Card=1)
```

INDEX SKIP SCAN 步骤告诉 Oracle 要跳跃式扫描这个索引，查找 GENDER 值有改变的地方，并从那里开始向下读树，然后在所考虑的各个虚拟索引中找到 OBJECT_ID = 42。如果大幅增加 GENDER 的可取值，如下：

```
ops$tkyte@ORA10GR1> update t

    2 set gender = chr(mod(rownum,256));

48215 rows updated.



ops$tkyte@ORA10GR1> begin

    2       dbms_stats.gather_table_stats

    3       ( user, 'T', cascade=>true );

    4 end;

    5 /

PL/SQL procedure successfully completed.
```

我们会看到，Oracle 不再认为跳跃式扫描是一个可行的计划。优化器本可以去检查 256 个小索引，但是它更倾向于执行一个全表扫描来找到所需要的行：

```
ops$tkyte@ORA10GR1> set autotrace traceonly explain

ops$tkyte@ORA10GR1> select * from t t1 where object_id = 42;



Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=158 Card=11.Bytes=95)

11.0          TABLE ACCESS (FULL) OF 'T' (TABLE) (Cost=158 Card=11.Bytes=95)
```

2. 情况 2

我们在使用一个 SELECT COUNT(*) FROM T 查询（或类似的查询），而且在表 T 上有一个 B*树索引。不过，优化器并不是统计索引条目，而是在全面扫描这个表（尽管索引比

表要小)。在这种情况下，索引可能建立在一些允许有 null 值的列上。由于对于索引键完全为 null 的行不会建立相应的索引条目，所以索引中的行数可能并不是表中的行数。这里优化器的选择是对的，如若不然，倘若它使用索引来统计行数，则可能会得到错误的答案。

3. 情况 3

对于一个有索引的列，做以下查询：

```
select * from t where f(indexed_column) = value
```

却发现没有使用 INDEX_COLUMN 上的索引。原因是这个列上使用了函数。我们是对 INDEX_COLUMN 的值建立了索引，而不是对 F(INDEXED_COLUMN)的值建索引。在此不能使用这个索引。如果愿意，可以另外对函数建立索引。

4. 情况 4

我们已经对一个字符创建了索引。这个列只包含数值数据。如果所用以下语句来查询:

```
select * from t where indexed_column = 5
```

注意查询中的数字 5 是常数 5(而不是一个字符串),此时就没有使用 INDEX_COLUMN 上的索引。这是因为，前面的查询等价于一些查询：

```
select * from t where to_number(indexed_column) = 5
```

我们对这个列隐式地应用了一个函数，如情况 3 所述，这就会禁止使用这个索引。通过一个小例子能很容易地看出这一点。在这个例子，我们将使用内置包 DBMS_XPLAN。这个包只在 Oracle9i Release 2 及以上版本中可用(在 Oracle9i Release 1 中，使用 AUTOTRACE 能很容易地查看计划，但是得不到谓词信息，这只在 Oracle9i Release 2 及以上版本中可见)：

```
ops$tkyte@ORA10GR1> create table t ( x char(1) constraint t_pk primary key,

  2 y date );

Table created.



ops$tkyte@ORA10GR1> insert into t values ( '5', sysdate );

11.row created.



ops$tkyte@ORA10GR1> delete from plan_table;

3 rows deleted.



ops$tkyte@ORA10GR1> explain plan for select * from t where x = 5;
```

```
Explained.



ops$tkyte@ORA10GR1> select * from table(dbms_xplan.display);



PLAN_TABLE_OUTPUT

----------------------------------------

Plan hash value: 749696591

-------------------------------------------------------------------------------------------------------
------

| Id  | Operation                     | Name | Rows  | Bytes | Cost  (%CPU)  |
Time          |

-------------------------------------------------------------------------------------------------------
------

| 0   | SELECT  STATEMENT    |          | 1        | 11.      | 2  (0)            |
00:00:01    |

|* 1   | TABLE  ACCESS  FULL   | T        | 1        | 11.      | 2  (0)            |
00:00:01    |

-------------------------------------------------------------------------------------------------------
------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter(TO_NUMBER("X")=5)
```

可以看到，它会全面扫描表；另外即使我们对查询给出了以下提示：

```
ops$tkyte@ORA10GR1> explain plan for select /*+ INDEX(t t_pk) */ * from t

  2 where x = 5;

Explained.
```

```
ops$tkyte@ORA10GR1> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT

------------------------------------

Plan hash value: 3473040572

-------------------------------------------------------------------------------------------------------------
--------------------------

| Id  | Operation                           | Name  | Rows            | Bytes           |
Cost (%CPU)  | Time           |

-------------------------------------------------------------------------------------------------------------
--------------------------

| 0   | SELECT STATEMENT                    |       | 1       | 11.     | 34 (0)          |
00:00:01   |

| 1   | TABLE ACCESS BY INDEX ROWID  | T          | 1       | 11.     | 34 (0)          |
00:00:01   |

|* 2   | INDEX FULL SCAN                     | T_PK     | 1       |         | 26 (0)          |
00:00:01   |

-------------------------------------------------------------------------------------------------------------
--------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter(TO_NUMBER("X")=5)
```

在此使用了索引，但是并不像我们想像中那样对索引完成惟一扫描（UNIQUE SCAN），而是完成了全面扫描（FULL SCAN）。原因从最后一行输出可以看出：filter(TO_NUMBER(“X”)=5)。这里对这个数据库列应用了一个隐式函数。X 中存储的字符串必须转换为一个数字，之后才能与值 5 进行比较。在此无法把 5 转换为一个串，因为我们的 NLS（国家语言支持）设置会控制 5 转换成串时的具体形式（而这是不确定的，不同的 NLS 设置会有不同的控制），所以应当把串转换为数字。而这样一来（由于应用了函数），就无法使用索引来快速地查找这一行了。如果只是执行串与串的比较：

```
ops$tkyte@ORA10GR1> delete from plan_table;

2 rows deleted.
```

```
ops$tkyte@ORA10GR1> explain plan for select * from t where x = '5';

Explained.


ops$tkyte@ORA10GR1> select * from table(dbms_xplan.display);


PLAN_TABLE_OUTPUT

--------------------------------------------------------------

Plan hash value: 1301177541

-------------------------------------------------------------------------------------------------------------
--------------------------

| Id  | Operation                           | Name  | Rows  | Bytes |
Cost (%CPU)| Time     |

-------------------------------------------------------------------------------------------------------------
--------------------------

| 0   | SELECT STATEMENT                    |       | 1     | 11.   | 1  (0)    |
00:00:01   |

| 1   | TABLE ACCESS BY INDEX ROWID  | T     | 1     | 11.   | 1 (0)     |
00:00:01   |

|* 2  | INDEX UNIQUE SCAN                   | T_PK  | 1     |       | 1 (0)     |
00:00:01   |

-------------------------------------------------------------------------------------------------------------
--------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("X"='5')
```

不出所料，这会得到我们期望的 INDEX UNIQUE SCAN，而且可以看到这里没有应用函数。一定要尽可能地避免隐式转换。苹果和橘子本来就是两样东西，苹果就和苹果比，而

橘子就该和橘子比。这里还经常出现一个关于日期的问题。如果做以下查询：

```
-- find all records for today

select * from t where trunc(date_col) = trunc(sysdate);
```

而且发现这个查询没有使用 DATE_COL 上的索引。为了解决这个问题。可以对 TRUNC(DATE_COL)建立索引，或者使用区间比较运算符来查询（也许这是更容易的做法）。下面来看对日期使用大于和小于运算符的一个例子。可以认识到以下条件：

```
TRUNC(DATE_COL) = TRUNC(SYSDATE)
```

与下面的条件是一样的：

```
select *

   from t

   where date_col >= trunc(sysdate)

   and date_col < trunc(sysdate+1)
```

这就把所有函数都移动等式的右边，这样我们就能使用 DATE_COL 上的索引了（而且与 WHERE TRUNC(DATE_COL)=TRUNC(SYSDATE)的效果完全一样）。

如果可能的话，倘若谓词中有函数，尽量不要对数据库列应用这些函数。这样做不仅可以使用更多的索引，还能减少处理数据库所需的工作。在上一种情况中，使用以上条件时：

```
   where date_col >= trunc(sysdate)

   and date_col < trunc(sysdate+1)
```

查询只会计算一次 TRUNC 值，然后就能使用索引来查找满足条件的值。使用 TRUNC(DATE_COL) = TRUNC(SYSDATE)时，TRUNC(DATE_COL)则必须对整个表（而不是索引）中的每一行计算一次。

5. 情况 5

此时如果用了索引，实际上反而会更慢。这种情况我见得太多了，人们想当然认为，索引总是会使查询更快。所以，他们会建立一个小表，再执行分析，却发现优化器并没有使用索引。在这种情况下，优化器的做法绝对是英明的。Oracle（对 CBO 而言）只会在合理地时候才使用索引。考虑下面的例子：

```
ops$tkyte@ORA10GR1> create table t

   2 ( x, y , primary key (x) )

   3 as

   4 select rownum x, object_name

   5 from all_objects
```

```
    6 /

Table created.



ops$tkyte@ORA10GR1> begin

    2 dbms_stats.gather_table_stats

    3 ( user, 'T', cascade=>true );

    4 end;

    5 /

PL/SQL procedure successfully completed.
```

如果运行一个查询，它只需要表中相对较少的数据，如下：

```
ops$tkyte@ORA10GR1> set autotrace on explain

ops$tkyte@ORA10GR1> select count(y) from t where x < 50;

  COUNT(Y)

      ----------

              49

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=11.Bytes=28)

11.0         SORT (AGGREGATE)

2 1             TABLE ACCESS (BY INDEX ROWID) OF 'T' (TABLE) (Cost=3 Card=41
Bytes=1148)

3 2                 INDEX (RANGE SCAN) OF 'SYS_C009167' (INDEX (UNIQUE)) (Cost=2
Card=41)
```

此时，优化器会很乐意地使用索引；不过，我们发现，如果估计通过索引获取的行数超过了一个阀值（取决于不同的优化器设计、物理统计等，这个阀值可能有所变化），就会观察到优化器将开始一个全部扫描：

```
ops$tkyte@ORA10GR1> select count(y) from t where x < 15000;

  COUNT(Y)

----------

     14999



Execution Plan

----------------------------------------------------------

0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=57 Card=11.Bytes=28)

11.0       SORT (AGGREGATE)

2  1           TABLE ACCESS (FULL) OF 'T' (TABLE) (Cost=57 Card=14994
Bytes=419832)
```

这个例子显示出优化器不一定会使用索引，而且实际上，它会做出正确的选择：采用跳跃式索引。对查询调优时，如果发现你认为本该使用的某个索引实际上并没有用到，就不要冒然强制使用这个索引，而应该先做个测试，并证明使用这个索引后确实会加快速度（通过耗用的时间和 I/O 次数来判断），然后再考虑让 CBO“就范”（强制它使用这个索引）。总得先给出理由吧。

6. 情况 6

有一段时间没有分析表了。这些表起先很小，但等到查看时，它们已经增长得非常大。现在索引就还有意义（尽管原先并非如此）。如果此时分析这个表，就会使用索引。

如果没有正确的统计信息，CBO 将无法做出正确的决定。

7. 索引情况小结

根据我的经验，这 6 种情况就是不使用索引的主要原因。归根结底，原因通常就是“不能使用索引，使用索引会返回不正确的结果“，或者”不应该使用，如果使用了索引，性能会变得很糟糕“。

11.6.5 神话：索引中从不重用空间

这是我要彻底揭穿的一个神话：在索引中确实会重用空间。这个神话是这样说的：假设有一个表 T，其中有一个列 X。在某个时间点上，你在表中放了一个值 X=5。后来把它删除了。据这个神话称：X=5 所用的空间不会被重用，除非以后你再把 X=5 放回索引中。按这个神话的说法，一旦使用了某个索引槽，它就永远只能被同一个值重用。从这个神话出发还有一个推论，认为空闲空间绝对不会返回给索引结构，而且块永远不会被重用。同样，事实并非如此。

很容易证明这个神话的第一部分是错误的。我们只需如下创建一个表：

```
ops$tkyte@ORA10GR1> create table t ( x int, constraint t_pk primary key(x) );

Table created.



ops$tkyte@ORA10GR1> insert into t values (1);

11.row created.



ops$tkyte@ORA10GR1> insert into t values (2);

11.row created.



ops$tkyte@ORA10GR1> insert into t values (9999999999);

11.row created.



ops$tkyte@ORA10GR1> analyze index t_pk validate structure;

Index analyzed.



ops$tkyte@ORA10GR1> select lf_blks, br_blks, btree_space
  2 from index_stats;



LF_BLKS      BR_BLKS     BTREE_SPACE

----------          ----------                -----------

           1                  0                    7996
```

因此，根据这个神话所述，如果我从 T 中删除了 X=2 的行，这个空间就不会得到重用，除非我再次插入数字 2。当前，这个索引使用了一个叶子块空间。如果索引键条目删除后绝对不会重用，只要我不断地插入和删除，而且从不重用任何值，那么这个索引就应该疯狂地增长。我们来看看实际是怎样的：

```
ops$tkyte@ORA10GR1> begin
  2      for i in 2 .. 999999
  3      loop
  4          delete from t where x = i;
  5          commit;
  6          insert into t values (i+1);
  7          commit;
  8      end loop;
  9 end;
  11. /
ops$tkyte@ORA10GR1> analyze index t_pk validate structure;
Index analyzed.

ops$tkyte@ORA10GR1> select lf_blks, br_blks, btree_space
  2 from index_stats;

LF_BLKS      BR_BLKS      BTREE_SPACE
----------          ----------                ----------
         1                   0                      7996
```

由此可以看出，索引中的空间确实得到了重用。不过，就像大多数神话一样，这里也有那么一点真实的地方。真实性在于，初始数字 2（介于 1～9.999.999.999 之间）所用的空间会永远保留在这个索引块上。索引不会自行“合并“。这说明，如果我用值 1～500,000 加载一个表，然后隔行删除表记录（删除所有偶数行），那么这个索引中那一列上就会有 250,000 个”洞“。只有当我重新插入数据，而且这个数据能在有洞的块中放下时，这些空间才会得到重用。Oracle 并不打算“收缩”或压缩索引，不过这可以通过 ALTER INDEX REBUILD 或 COALESCE 命令强制完成。另一方面，如果我用值 1～500,000 加载一个表，然后从表中删除值小于或等于 250,000 的每一行，就会发现从索引中清除的块将放回到索引的 freelist 中，这个空间完全可以重用。

如果你还记得，第二个神话：索引空间从不“回收”。据这个神话称：一旦使用了一个索引块，它就会一直呆在索引结构的那个位置上，而且只有当你插入数据，并放回到原来那个位置上时，这个块才会被重用。同样可以证明这是错误的。首先，需要建立一个表，其中大约有 500,000 行。为此，我们将使用 big_table 脚本。有了这个表，而且有了相应的主键索引后，我们将测量索引中有多少个叶子块，另外索引的 freelist 上有多少个块。要记住，对于一个索引，只有当块完全为空时才会放在 freelist 上，这一点与表不同。所以我们在 freelist 上看到的块都完全为空，可以重用。

```
ops$tkyte@ORA10GR1> select count(*) from big_table;

COUNT(*)

----------

500000



ops$tkyte@ORA10GR1> declare

  2      l_freelist_blocks number;

  3 begin

  4      dbms_space.free_blocks

  5          ( segment_owner => user,

  6      segment_name => 'BIG_TABLE_PK',

  7      segment_type => 'INDEX',

  8      freelist_group_id => 0,

  9      free_blks => l_freelist_blocks );

  11. dbms_output.put_line( 'blocks on freelist = ' || l_freelist_blocks );

  11. end;

  11. /

blocks on freelist = 0

PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA10GR1> select leaf_blocks from user_indexes
  2 where index_name = 'BIG_TABLE_PK';


LEAF_BLOCKS
-----------
       1043
```

执行这个批量删除之前，freelist 上没有块，而在索引的“叶子”层上有 11.043 给块，这些叶子块中包含着数据。下面，我们将执行删除，并再次测量空间的利用情况：

```
ops$tkyte@ORA10GR1> delete from big_table where id <= 250000;
250000 rows deleted.


ops$tkyte@ORA10GR1> commit;
Commit complete.


ops$tkyte@ORA10GR1> declare
  2      l_freelist_blocks number;
  3 begin
  4      dbms_space.free_blocks
  5          ( segment_owner => user,
  6          segment_name => 'BIG_TABLE_PK',
  7          segment_type => 'INDEX',
  8          freelist_group_id => 0,
  9          free_blks => l_freelist_blocks );
 11.      dbms_output.put_line( 'blocks on freelist = ' || l_freelist_blocks );
```

```
11.      dbms_stats.gather_index_stats
11.      ( user, 'BIG_TABLE_PK' );
11. end;
11. /
blocks on freelist = 520
PL/SQL procedure successfully completed.
ops$tkyte@ORA10GR1> select leaf_blocks from user_indexes
  2 where index_name = 'BIG_TABLE_PK';


LEAF_BLOCKS
-----------
       523
```

可以看到，现在，索引中一半以上的块都在 freelist 上（520 个块），而且现在只有 523 个叶子块。如果将 523 和 520 相加，又得到了原来的 11.043。这说明 freelist 上的这些块完全为空的，而且可以重用（索引 freelist 上的块必须为空，这与堆组织表的 freelist 上的块不同 。

以上例子强调了两点：

- 一旦插入了可以重用空间的行，索引块上的空间就会立即重用。
- 索引块为空时，会从索引结构中取出它，并在以后重用。这可能是最早出现这个神话的根源：与表不同，在索引结构中，不能清楚地看出一个块有没有“空闲空间”。在表中，可以看到 freelis 上的块，即使其中包含有数据。而在索引中，只能在 freelist 上看到完全为空的块；至少有一个索引条目（但其余都是空闲空间）的块就无法清楚地看到。

11.6.6 神话：最有差别的元素应该在最前面

这看上去像是一个常识。对于一个有 100,000 行的表，如果要在 C1 和 C2 列上创建一个索引，你发现 C1 有 100,000 个不同的值，而 C2 有 25,000 个不同的值，你可能想在 T(C1, C2)上创建索引。这说明，C1 应该在前面，这是“常识性”的方法。事实上，在比较数据向量时（假设 C1 和 C2 是向量），把哪一个放在前面都关系不大。考虑以下例子。我们将基于 ALL_OBJECTS 创建一个表，并基于 OWNER、OBJECT_TYPE 和 OBJECT_NAME 列创建一个索引（这些列按从最没有差别到最有差别的顺序排列，即 OWNER 列差别最小，OBJECT_TYPE 次之，OBJECT_NAME 列差别最大），另外还在 OBJECT_NAME、OBJECT_TYPE 和 OWNER 上创建了另一个索引：

```
ops$tkyte@ORA10GR1> create table t
  2 as
  3 select * from all_objects;
Table created.


ops$tkyte@ORA10GR1> create index t_idx_1 on t(owner,object_type,object_name);
Index created.


ops$tkyte@ORA10GR1> create index t_idx_2 on t(object_name,object_type,owner);
Index created.


ops$tkyte@ORA10GR1> select count(distinct owner), count(distinct object_type),
  2       count(distinct object_name ), count(*)
  3 from t;


DISTINCTOWNER                                    DISTINCTOBJECT_TYPE
DISTINCTOBJECT_NAME      COUNT(*)

           ------------                  -------------------                  -------------------
 --------

                    28                            36                         28537          4
8243
```

现在，为了显示这二者在高效使用空间方面难分伯仲，下面测量它们的空间利用情况：

```
ops$tkyte@ORA10GR1> analyze index t_idx_1 validate structure;
Index analyzed.
```

```
ops$tkyte@ORA10GR1> select btree_space, pct_used, opt_cmpr_count, opt_cmpr_pctsave
  2 from index_stats;

BTREE_SPACE       PCT  OPT_CMPR_COUNT  OPT_CMPR_PCTSAVE
-----------  ------  --------------  ----------------
    2702744    89.0               2                28

ops$tkyte@ORA10GR1> analyze index t_idx_2 validate structure;

Index analyzed.

ops$tkyte@ORA10GR1> select btree_space, pct_used, opt_cmpr_count, opt_cmpr_pctsave
  2 from index_stats;

BTREE_SPACE       PCT  OPT_CMPR_COUNT  OPT_CMPR_PCTSAVE
-----------  ------  --------------  ----------------
    2702744    89.0               1                11
.
```

它们使用的空间大小完全一样，细到字节级都一样，二者没有什么区别。不过，如果使用索引键压缩，第一个索引更可压缩，这一点由 OPT_CMP_PCTSAVE 值可知。有人提倡索引中应该按最没有差别到最有差别的顺序来安排列，这正是这种看法的一个理由。下面来看这两个索引的表现，从而确定是否有哪个索引更“优秀”，总比另一个索引更高效。要测试这一点，我们将使用一个 PL/SQL 代码块（其中包括有提示的查询，指示要使用某个索引或者另一个索引）：

```
ops$tkyte@ORA10GR1> alter session set sql_trace=true;

Session altered.

ops$tkyte@ORA10GR1> declare
```

```
2       cnt int;
3 begin
4       for x in ( select /*+FULL(t)*/ owner, object_type, object_name from t )
5       loop
6           select /*+ INDEX( t t_idx_1 ) */ count(*) into cnt
7           from t
8           where object_name = x.object_name
9               and object_type = x.object_type
11.           and owner = x.owner;
11.
11.       select /*+ INDEX( t t_idx_2 ) */ count(*) into cnt
11.       from t
11.       where object_name = x.object_name
11.           and object_type = x.object_type
11.           and owner = x.owner;
11. end loop;
11. end;
11. /
PL/SQL procedure successfully completed.
```

这些查询按索引读取表中的每一行。TKPROF 报告显示了以下结果：

```
SELECT /*+ INDEX( t t_idx_1 ) */ COUNT(*) FROM T
WHERE OBJECT_NAME = :B3 AND OBJECT_TYPE = :B2 AND OWNER = :B1

call        count       cpu    elapsed       disk      query    current        rows
```

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

Parse                1       0.00          0.00           0             0              0              0

Execute          48243      11..63         11..78          0             0              0              0

Fetch            48243       11.90          11.77          0        145133              0          48243

-------                         ------                  --------                  ----------
----------      ----------      ----------      ----------

total            96487      11..53         11..55          0        145133              0          48243

         Rows     Row Source Operation

          -------     ------------------------------------------------

        48243     SORT AGGREGATE (cr=145133 pr=0 pw=0 time=2334197 us)

        57879     INDEX RANGE SCAN T_IDX_1 (cr=145133 pr=0 pw=0 time=1440672
us)(object...

********************************************************************************
******

SELECT /*+ INDEX( t t_idx_2 ) */ COUNT(*) FROM T

WHERE OBJECT_NAME = :B3 AND OBJECT_TYPE = :B2 AND OWNER = :B1



call             count        cpu      elapsed         disk         query       current          rows

-------                         ------                  --------                  ----------
----------      ----------      ----------      ----------

Parse                1       0.00          0.00           0             0              0              0

Execute          48243      11..00         11..78          0             0              0              0

Fetch            48243       11.87         2.11. 0 145168                0          48243

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

```
total          96487      11..87      11..88          0      145168          0      48243

      Rows     Row Source Operation

       -------  ---------------------------------------------------

      48243     SORT AGGREGATE (cr=145168 pr=0 pw=0 time=2251857 us)

      57879     INDEX RANGE SCAN T_IDX_2 (cr=145168 pr=0 pw=0 time=1382547
us)(object...
```

它们处理的行数完全相同，而且块数也非常类似（之所以存在微小的差别，这是因为表中的行序有些偶然性，而且 Oracle 相应地会做一些优化），它们使用了同样的 CPU 时间，而且在大约相同的耗用时间内运行（再运行这个测试，CPU 和 ELAPSED 这两个数字会有一点差别，但是平均来讲它们是一样的）。按照各个列的差别大小来安排这些列在索引中的顺序并不会获得本质上的效率提升，另外如前所示，如果再考虑到索引键压缩，可能还更倾向于把最没有选择性的列放在最前面。如果对索引采用 COMPRESS 2，再运行前面的例子，你会发现，对于给定情况下的这个查询，第一个查询执行的 I/O 次数大约是后者的 2/3。

不过事实上，对于是把 C1 列放在 C2 列之前，这必须根据如果使用索引来决定。如果有大量如下的查询：

```
select * from t where c1 = :x and c2 = :y;

select * from t where c2 = :y;
```

那么在 T(C2,C1)上建立索引就更合理。以上这两个查询都可以使用这个索引。另外，通过使用索引键压缩（我们在介绍 IOT 时讨论过，后面还将进一步分析），如果 C2 在前，就能建立一个更小的索引。这是因为，C2 的各个值会在索引中平均重复 4 次。如果 C1 和 C2 的平均长度都是 11.字节，那么按道理这个索引的条目就是 2,000,000 字节（100,000×20）。倘若在（C2,C1）上使用索引键压缩，可以把这个索引收缩为 11.250,000（100,000×11..5）字节，因为 C2 的 4 次重复中有 3 次都可以避免。

在 Oracle 5 中（不错，确实是“古老的”Oracle 5!），曾经认为应该把最有选择性的列放在索引的最前面。其理由缘于 Oracle 5 实现索引压缩的方式（不同于索引键压缩）。这个特性在 Oracle 6 中就已经去掉了，因为 Oracle 6 中 增加了行级锁。从那以后，“把最有差别的列放在索引最前面会使索引更小或更有效率”的说法不再成立。看上去好像是这样，但实际上并非如此。如果利用索引键压缩，则恰恰相反，因为反过来才会使索引更小（即把最没有差别的列放在索引最前面）。不过如前所述，还是应该根据如何使用索引来做出决定。

11.7 小结

这一章中，我们介绍了 Oracle 必须提供的不同类型的索引。首先讨论了基本的 B*树索引，并介绍了这种索引的几种子类型，如反向键索引（为 Oracle RAC 所设计）和降序索引（来获取按升序和降序混合排序的数据）。我们还花了一些时间来讨论什么时候应当使用索引，另外解释了为什么某些情况下索引可能没有用。

然后我们介绍了位图索引，在数据仓库环境（即读密集型环境，而不是 OLTP）中，这对于为低到中基数的数据建立索引是一个绝好的方法。我们介绍了在哪些情况想适于使用位

图索引，并解释了为什么在 OLTP 环境（或多个用户必须并发地更新同一个列的任何环境）中不应该考虑使用位图索引。

接下来转向基于函数的索引，这实际上是 B*树 索引和位图索引的特例。基于函数的索引允许我们在一个列（或多个列）的函数上创建索引，这说明可以预先计算和存储复杂计算和用户编写的函数的结果，以便以 后以极快的速度完成索引获取。我们介绍了有关基于函数的索引的一些重要的实现细节，如必须有一些必要的系统级和会话级设置才能使用基于函数的索引。接下来 分别在内置 Oracle 函数和用户编写的函数上举了两个基于函数的索引例子。最后，我们谈到了关于基于函数的索引的一些警告。

然后分析了一个非常特定的索引类型，这称为应用域索引。在此没有深入地介绍如何从头构建这种形式的索引（这个过程很长，也很复杂），而是介绍了 Oracle 所实现的一个例子：文本索引。

最后我回答了一些关于索引最常问的问题，还澄清了有关索引的一些神话。这一节不仅涵盖了一些简单的问题，如“能在视图中使用索引吗？”，也涉及一些更复杂的 神话，如“索引中从不重用空间”。我们主要是通过具体的例子来回答这些问题，揭穿上述神话，并在此过程中展示有关的概念。

第 12 章 数据类型

选 择一个正确的数据类型，这看上去再容易不过了，但我屡屡见得选择不当的情况。要选择什么类型来存储你的数据，这是一个最基本的决定，而且这个决定会在以后 的数年间影响着你的应用和数据。选择适当的数据类型至关重要，而且很难事后再做改变，也就是说，一旦选择某些类型实现了应用，在相当长的时间内就只能“忍 耐”（因为你选择的类型可能不太合适）。

这一章我们将介绍 Oracle 所有可用的基本数据类型，讨论这些类型如何实现，并说明每种类型分别适用于哪些情况。在此不讨论用户定义的数据类型，因为用户定义的数据类型只是由 Oracle 内 置数据类型导出的一些复合对象。我们会分析如果不合适地使用了错误的数据类型，甚至只是对数据类型使用了不正确的参数（如长度、精度、小数位等），将会发 生什么情况。读完这一章后，你将对使用哪些类型有所了解，明白这些类型如何实现以及何时使用，还有很重要的一点，你会认识到为什么关键是要针对具体任务使 用适宜的类型。

12.1 Oracle 数据类型概述

Oracle 提供了 22 种不同的 SQL 数据类型供我们使用。简要地讲，执行数据类型如下：

- CHAR：这是一个定长字符串，会用空格填充来达到其最大长度。非 null 的 CHAR(12.)总是包含 12.字节信息（使用了默认国家语言支持 National Language Support，NLS 设置）。稍后将更详细地介绍 NLS 的作用。CHAR 字段最多可以存储 2,000 字节的信息。
- NCHAR：这是一个包含 UNICODE 格式数据的定长字符串。Unicode 是一种对字符进行编码的通用方法，而不论使用的是何种计算机系统平台。有了 NCHAR 类型，就允许数据库中包含采用两种不同字符集的数据：使用数据库字符集的 CHAR 类型和使用国家字符集的 NCHAR 类型。非 null 的 NCHAR(12.)总是包含 12.个字符的信息（注意，在这方面，它与 CHAR 类型有所不同）。NCHAR 字段最多可以存储 2,000 字节的信息。
- VARCHAR2：目前这也是 VARCHAR 的同义词。这是一个变长字符串，与 CHAR 类型不同，它不会用空格填充至最大长度。VARCHAR2(12.)可能包含 0～12.字节的信息（使用默认 NLS 设置）。VARCHAR2 最多可以存储 4,000 字节的信息。
- NVARCHAR2：这是一个包含 UNICODE 格式数据的变长字符串。

NVARCHAR2(12.)可以包含 0～12.字符的信息。NVARCHAR2 最多可以存储 4,000 字节的信息。

- RAW：这是一种变长二进制数据类型，这说明采用这种数据类型存储的数据不会发生字符集转换。可以把它看作由数据库存储的信息的二进制字节串。这种类型最多可以存储 2,000 字节的信息。

- NUMBER：这种数据类型能存储精度最多达 38 位的数字。这些数介于 $12.0\times12.^{(-130)}$——（但不包括）$12.0\times12.^{(126)}$之间。每个数存储在一个变长字段中，其长度在 0（尾部的 NULL 列就是 0 字节）～22 字节之间。Oracle 的 NUMBER 类型精度很高，远远高于许多编程语言中常规的 FLOAT 和 DOUBLE 类型。

- BINARY_FLOAT：这是 Oracle 10g Release 1 及以后版本中才有的一种新类型。它是一个 32 位单精度浮点数，可以支持至少 6 位精度，占用磁盘上 5 字节的存储空间。

- LONG：这种类型能存储最多 2G 的字符数据（2GB 是指 2 千兆字节，而不是 2 千兆字符，因为在一个多字节字符集中，每个字符可能有多个字节）。由于 LONG 类型有许多限制（后面会讨论），而且提供 LONG 类型只是为了保证向后兼容性，所以强烈建议新应用中不要使用 LONG 类型，而且在现有的应用中也要尽可能将 LONG 类型转换为 CLOB 类型。

- LONG RAW：LONG RAW 类型能存储多达 2GB 的二进制信息。由于 LONG 同样的原因，建议在将来的所有开发中都使用 CLOB 类型，另外现有的应用中也应尽可能将 LONG RAW 转换为 BLOB 类型。

- DATE：这是一个 7 字节的定宽日期/时间数据类型。其中总包含 7 个属性，包括：世纪、世纪中哪一年、月份、月中的哪一天、小时、分钟和秒。

- TIMESTAMP：这是一个 7 字节或 12.字节的定宽日期/时间数据类型。它与 DATE 数据类型不同，因为 TIMESTAMP 可以包含小数秒（fractional second）；带小数秒的 TIMESTAMP 在小数点右边最多可以保留 9 位。

- TIMESTAMP WITH TIME ZONE：与前一种类型类似，这是一个 12.字节的定宽 TIMESTAMP，不过它还提供了时区（TIME ZONE）支持。数据中会随 TIMESTAMP 存储有关时区的额外信息，所以原先插入的 TIME ZONE 会与数据一同保留。

- TIMESTAMP WITH LOCAL TIME ZONE：与 TIMESTAMP 类似，这是一种 7 字节或 12.字节的定宽日期/时间数据类型；不过，这种类型对时区敏感（time zone sensitive）。如果在数据库中有修改，会参考数据中提供的 TIME ZONE，根据数据库时区对数据中的日期/时间部分进行“规范化”。所以，如果你想使用 U.S./Pacific 时区插入一个日期/时间，而数据库时区为 U.S./Eastern，最后的日期/时间信息会转换为 Eastern 时区的日期/时间，并像 TIMESTAMP 一样存储。获取这个数据时，数据库中存储的 TIMESTAMP 将转换为会话时区的时间。

- INTERVAL YEAR TO MONTH：这是一个 5 字节的定宽数据类型，用于存储一个时间段，这个类型将时段存储为年数和月数。可以在日期运算中使用这种时间间隔使一个 DATE 或 TIMESTAMP 类型增加或减少一段时间。

- INTERVAL DAY TO SECOND：这是一个 12.字节的定宽数据类型，用于存储一

个时段，这个类型将时段存储为天/小时/分钟/秒数，还可以有最多 9 位的小数秒。

- BFILE：这种数据类型允许在数据库列中存储一个 Oracle 目录对象（操作系统目录的一个指针）和一个文件名，并读取这个文件。这实际上允许你以一种只读的方式访问数据库服务器上可用的操作系统文件，就好像它们存储在数据库表本身中一样。

- BLOB：在 Oracle9i 及以前的版本中，这种数据类型允许存储最多 4GB 的数据，在 Oracle 10g 及以后的版本中允许存储最多（4GB）×（数据库块大小）字节的数据。BLOB 包含不需要进行字符集转换的“二进制“数据，如果要存储电子表格、字处理文档、图像文件等就很适合采用这种数据类型。

- CLOB：在 Oracle9i 及以前的版本中，这种数据类型允许存储最多 4GB 的数据，在 Oracle 10g 及以后的版本中允许存储最多（4GB）×（数据库块大小）字节的数据。CLOB 包含要进行字符集转换的信息。这种数据类型很适合存储纯文本信息。

- NCLOB：在 Oracle9i 及以前的版本中，这种数据类型允许存储最多 4GB 的数据，在 Oracle 10g 及以后的版本中允许存储最多（4GB）×（数据库块大小）字节的数据。NCLOB 存储用数据库国家字符集编码的信息，而且像 CLOB 一样，这些信息要进行字符集转换。

- ROWID：ROWID 实际上是数据库中一行的 12.字节地址。ROWID 中编码有足够的信息，足以在磁盘上定位这一行，以及标识 ROWID 指向的对象（表等）。

- UROWID：UROWID 是一个通用 ROWID，用于表（如 IOT 和通过异构数据库网关访问的没有固定 ROWID 的表 。UROWID 是行主键值的一种表示，因此，取决于所指向的对象，UROWID 的大小会有所变化。

显然以上列表中还少了许多类型，如 INT、INTEGER、SMALLINT、FLOAT、REAL 等。这些类型实际上都是在上表所列的某种类型的基础上实现的，也就是说，它们只是固有 Oracle 类型的同义词。另外，诸如 XML Type、SYS.ANYTYPE 和 SDO_GEOMETRY 之类的类型在此也未列出，因为本书不打算讨论这些类型。它们是一些复杂的对象类型，包括一个属性集合以及处理这些属性的一些方法（函数）。这些复杂类型由上述基本数据类型组成，并不是传统意义上真正的数据类型，而只是你在应用中可以利用的一种实现或一组功能。

下面将更详细地讨论这些基本数据类型。

12.2 字符和二进制串类型

Oracle 中的字符数据类型包括 CHAR、VARCHAR2 以及带“N“的相应”变体“（NCHAR 和 NVARCHAR2），这些字符数据类型能存储 2,000 字节或 4,000 字节的文本。这些文本会由数据库根据需要在不同字符集之间转换。字符集（chrarcter set）是各个字符的一种二进制表示（用位和字节表示）。目前有多种不同的字符集，每种字符集能表示不同的字符，例如：

- US7ASCII 字符集是 128 字符的 ASCII 标准表示。它使用字节的低 7 位表示这 128 个字符。

- WE8ISO8859P1 字符集是一种西欧字符集，不仅能不是 128 个 ASCII 字符，还能表示 128 个扩展字符，这个字符集使用了字节的全部 8 位。

在深入分析 CHAR、VARCHAR2 及带“N “的变体（NCHAR 和 NVARCHAR2）之前，我们先来简要地了解这些不同的字符集对于我们意味着什么，搞清楚这一点会很有帮助。

12.2.1 NLS 概述

如前所述，NLS 代表国家语言支持（National Language Support）。NLS 是数据库的一个非常强大的特性，但是人们往往未能正确理解这个特性。NLS 控制着数据的许多方面。例如，它控制着数据如何存储；还有我们在数据中是会看到多个逗号和一个句号（如 12.000,000.01），还是会看到多个点号和一个逗号（如 12.000.000,01）。但是最重要的，它控制着以下两个方面：

- 文本数据持久存储在磁盘上时如何编码
- 透明地将数据从一个字符集转换到另一个字符集

正是这个透明的部分最让人困惑，它实在太透明了，以至于我们甚至不知道发生了这种转换。下面来看一个小例子：

假设你在数据库中用 WE8ISO8859P1 字符集存储 8 位的数据，但是你的某些客户使用的是一种 7 位字符集，如 US7ASCII。这些客户不想要 8 位的数据，需要从数据库将数据转换为他们能用的形式。尽管听上去不错，但是如果你不知道会发生这种转换，就会发现，过一段时间后，数据会“丢失“字符，WE8ISO8859P1 字符集中的某些字符在 US7ASCII 中并没有，这些字符会转换为 US7ASCII 中的某个字符。原因就在于这里发生了字符集转换。简而言之，如果你从数据库获取了使用字符集 1 的数据，将其转换为使用字符集 2，再把这些数据插入回数据库中（完成以上的逆过程），就极有可能大幅修改数据。字符集转换过程通常会修改数据，而你往往会把一个较大的字符集（在此例中就是 8 位字符集）映射到一个较小的字符集（此例中的 7 位字符集）。这是一种有损转换（lossy conversion），字符就会被修改，这只是因为：较小的字符集不可能表示较大字符集中的每一个字符。但是这种转换必须发生。如果数据库以一种单字节字符集存储数据，但是客户（如一个 Java 应用，因为 Java 语言使用 Unicode）希望数据采用多字节表示，就必须执行转换，只有这样客户应用才能使用这些数据。

可以很容易地看到字符集转换。例如，我有一个数据库，它的字符集设置为 WE8ISO8859P1，这是一个典型的西欧字符集：

```
ops$tkyte@ORA10G> select *
  2    from nls_database_parameters
  3    where parameter = 'NLS_CHARACTERSET';


PARAMETER                      VALUE
------------------------------ ----------------------------------------
NLS_CHARACTERSET               WE8ISO8859P1
```

现在，如果可以确保 NLS_LANG 设置为我的数据库字符集，如下（Windows 用户要在注册表中修改/验证这个设置）：

```
ops$tkyte@ORA10G> host echo $NLS_LANG

AMERICAN_AMERICA.WE8ISO8859P1
```

然后，创建一个表，并放入一些“8 位”数据，对于只希望得到 7 位 ASCII 数据的客户来说（以下将这些客户称为“7 位客户”），它们无法使用这些数据：

```
ops$tkyte@ORA10G> create table t ( data varchar2(1) );

Table created.



ops$tkyte@ORA10G> insert into t values ( chr(224) );

12.row created.



ops$tkyte@ORA10G> insert into t values ( chr(225) );

12.row created.



ops$tkyte@ORA10G> insert into t values ( chr(226) );

12.row created.



ops$tkyte@ORA10G> select data, dump(data) dump

  2 from t;



D       DUMP

-       --------------------

à       Typ=1 Len=1: 224

á       Typ=1 Len=1: 225

á       Typ=1 Len=1: 226

ops$tkyte@ORA10G> commit;
```

现在，如果打开另一个窗口，指定一个“7 位 ASCII”客户，就会看到完全不同的结果：

```
[tkyte@desktop tkyte]$ export NLS_LANG=AMERICAN_AMERICA.US7ASCII

[tkyte@desktop tkyte]$ sqlplus /

SQL*Plus: Release 12..12.0.4.0 - Production on Mon May 30 12.:58:46 2005

Copyright . 1982, 2005, Oracle. All rights reserved.

Connected to:

Oracle Database 10g Enterprise Edition Release 12..12.0.4.0 - Production

With the Partitioning, OLAP and Data Mining options

ops$tkyte@ORA10G> select data, dump(data) dump

  2      from t;



D       DUMP

-       --------------------

a       Typ=1 Len=1: 224

a       Typ=1 Len=1: 225

a       Typ=12.Len=1: 226
```

注意，在这个 7 位会话（使用 7 位字符集的会话）中，接收到 3 次字母”a”，这些 a 头上没有区分号。不过，DUMP 函数显示出数据库中实际上有 3 个不同的字符，而不都是字母“a”。数据库中数据没有变化，只是客户接收到的值有变化。而且实际上，如果这个客户要把数据库获取到宿主变量中，如下：

```
ops$tkyte@ORA10G> variable d varchar2(1)

ops$tkyte@ORA10G> variable r varchar2(20)

ops$tkyte@ORA10G> begin

  2      select data, rowid into :d, :r from t where rownum = 1;

  3 end;
```

```
  4 /

PL/SQL procedure successfully completed.
```

对它们不做任何处理，只是将其发回到数据库中：

```
ops$tkyte@ORA10G> update t set data = :d where rowid = chartorowid(:r);

12.row updated.

ops$tkyte@ORA10G> commit;

Commit complete.
```

然后，在原先的 8 位会话（使用 8 位字符集的会话）中就会观察到：丢掉了原先的一个字符。它已经代之以低 7 位表示的 a，而不是先前所示的那个奇特的á：

```
ops$tkyte@ORA10G> select data, dump(data) dump

  2 from t;

D       DUMP

-       --------------------

a       Typ=1 Len=1: 97

á       Typ=1 Len=1: 225

á       Typ=1 Len=1: 226
```

这就说明，如果环境中有不同的字符集，而且客户和数据库使用不同的 NLS 设置，这会产生直接的影响。这一点一定要注意，因为它在许多场合下都会起作用。例如，如果 DBA 使用 EXP 工具来抽取信息，他可能观察到以下警告：

```
[tkyte@desktop tkyte]$ exp userid=/ tables=t

Export: Release 12..12.0.4.0 - Production on Mon May 30 12.:12.:09 2005

Copyright . 1982, 2004, Oracle. All rights reserved.

Connected to: Oracle Database 10g Enterprise Edition Release 12..12.0.4.0 - Production

With the Partitioning, OLAP and Data Mining options

Export done in US7ASCII character set and AL16UTF16 NCHAR character set

server uses WE8ISO8859P1 character set (possible charset conversion)

...
```

要非常谨慎地处理这种警告。如果你想导出这个表，希望删除表后再使用 IMP 创建这个表，此时你会发现表中的所有数据现在都只是低 7 为数据！一定要当心这种并非有意的字符集转换。

另外还要注意，字符集转换一般都是必要的。如果客户希望数据采用某个特定的字符集，倘若向这个客户发送使用另一个字符集的信息，结果将是灾难性的。

注意 我强烈推荐所有人都应该通读 *Oracle Globalization Support Guide* 文档。其中深入地讨论了与 NLS 有关的问题，还涵盖了这里没有介绍的一些内容。如果有人要创建将全球使用的应用（设置只是跨洲使用），就很有必要阅读这个文档；或者就算是现在不需要创建这样的应用，也应该未雨绸缪，掌握这个文档中的信息。

既然我们对字符集已经有了初步的认识，并且了解了字符集可能对我们有怎样的影响，下面来介绍 Oracle 提供的各种字符串类型。

12.2.2 字符串

Oracle 中有 4 种基本的字符串类型，分别是 CHAR、VARCHAR2、NCHAR 和 NVARCHAR2。在 Oracle 中，所有串都以同样的格式存储。在数据库块上，最全面都有一个 1～3 字节的长度字段，其后才是数据，如果数据为 NULL，长度字段则表示一个但字节值 0xFF。

注意 Oracle 中尾部的 NULL 列占用 0 字节存储空间，这说明，如果表中的“最后一列”为 NULL，Oracle 不会为之存储任何内容。如果最后两列都是 NULL，那么对这两列都不会存储任何内容。但是，如果位于 NULL 列之后的某个列要求为 not null（即不允许为 null），Oracle 会使用这一节介绍的 null 标志来指示这个列缺少值。

如果串的长度小于或等于 250（0x01~0xFA），Oracle 会使用 1 个字节来表示长度。对于所有长度超过 250 的串，都会在一个标志字节 0xFE 后跟有两个字节来表示长度。因此，如果有一个包含“Hello World”的 VARCHAR2(80)，则在块中可能如图 12.-1 所示。

图 12.-1 存储在一个 VARCHAR2(80)中的 Hello World

另一方面，如果在一个 CHAR(80)中存储同样的数据，则可能如同 12.-2 所示。

图 12.-2 存储在一个 CHAR(80)中的 Hello World

CHAR/NCHAR 实际上只是伪装的 VARCHAR2/NVARCHAR2，基于这一点，所以我认为其实只需要考虑这两种字符串类型：VARCHAR 和 NVARCHAR2。我从来没有见过哪个应用适合使用 CHAR 类型。因为 CHAR 类型总是会用空格填充得到的串，使之达到一个固定宽度，所以我们很快就会发现：不论在表段还是任何索引段中，CHAR 都会占用最大的存储空间。这就够糟糕的了，避免使用 CHAR/NCHAR 类型还有另一个很重要的原因：在需要获取这些信息的应用中，CHAR/NCHAR 类型还会带来混乱（很多应用存储了信息之后却无法“找到”所存储的数据）。其原因与字符串比较的规则有关，也与执行字符串比较的严格程度有关。下面通过一个小例子来展示这个问题，这里在一个简单的表中使用了‘Hello World’串：

```
ops$tkyte@ORA10G> create table t
  2     ( char_column char(20),
  3     varchar2_column varchar2(20)
  4     )
  5 /
Table created.


ops$tkyte@ORA10G> insert into t values ( 'Hello World', 'Hello World' );
12.row created.


ops$tkyte@ORA10G> select * from t;
CHAR_COLUMN            VARCHAR2_COLUMN
--------------------   --------------------
Hello World            Hello World


ops$tkyte@ORA10G> select * from t where char_column = 'Hello World';
CHAR_COLUMN            VARCHAR2_COLUMN
--------------------   --------------------
```

```
Hello World                    Hello World



ops$tkyte@ORA10G> select * from t where varchar2_column = 'Hello World';

CHAR_COLUMN            VARCHAR2_COLUMN

--------------------    --------------------

Hello World                    Hello World
```

到目前为止，两个列看上去好像是一样的，但实际上这里发生了一些隐式转换，在与 CHAR 列比较时，CHAR(12.)直接量（’Hello World’）已经提升为一个 CHAR(20)，并在其中填充了空格。这种转换肯定已经发生了，因为 Hello World……与没有尾部空格的 Hello World 并不相同。可以确认这两个串是截然不同的：

```
ops$tkyte@ORA10G> select * from t where char_column = varchar2_column;

no rows selected
```

它们彼此并不相等。我们要么必须用空格填充 VARCHAR2_COLUMN 列，使其长度到达 20 字节，要么必须从 CHAR_COLUMN 列截去尾部的空格，如下：

```
ops$tkyte@ORA10G> select * from t where trim(char_column) = varchar2_column;

CHAR_COLUMN            VARCHAR2_COLUMN

--------------------    --------------------

Hello World                    Hello World



ops$tkyte@ORA10G> select * from t where char_column = rpad( varchar2_column, 20 );

CHAR_COLUMN            VARCHAR2_COLUMN

--------------------    --------------------

Hello World                    Hello World
```

注意　用空格填充 VARCHAR2_COLUMN 有很多方法，如使用 CAST()函数。

对于使用变长串的应用，绑定输入时会出现问题，而且肯定会得到“没有找到数据“之类的错误：

```
ops$tkyte@ORA10G> variable varchar2_bv varchar2(20)
```

```
ops$tkyte@ORA10G> exec :varchar2_bv := 'Hello World';

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select * from t where char_column = :varchar2_bv;

no rows selected

ops$tkyte@ORA10G> select * from t where varchar2_column = :varchar2_bv;

CHAR_COLUMN              VARCHAR2_COLUMN

--------------------     --------------------

Hello World              Hello World
```

在此，搜索 VARCHAR2 串成功了，但是搜索 CHAR 列未成功。VARCHAR2 绑定变量不会像字符串直接量那样提升为 CHAR(20)。在此，许多程序员会形成这样一个观点，认为“绑定变量不能工作：所以必须使用直接量“。这实在是一个极其糟糕的决定。要完成绑定，解决方案是使用 CHAR 类型：

```
ops$tkyte@ORA10G> variable char_bv char(20)

ops$tkyte@ORA10G> exec :char_bv := 'Hello World';



PL/SQL procedure successfully completed.

ops$tkyte@ORA10G>

ops$tkyte@ORA10G> select * from t where char_column = :char_bv;

CHAR_COLUMN              VARCHAR2_COLUMN

--------------------     --------------------

Hello World              Hello World

ops$tkyte@ORA10G> select * from t where varchar2_column = :char_bv;

no rows selected
```

不过，如果混合使用并匹配 VARCHAR2 和 CHAR，你就会不断地遭遇这个问题。不仅如此，开发人员现在还必须在应用中考虑字段宽度。如果开发人员喜欢使用 RPAD()技巧将绑定变量转换为某种能与 CHAR 字段比较的类型（当然，与截断（TRIM）数据库列相比，

填充绑定变量的做法更好一些，因为对列应用函数 TRIM 很容易导致无法使用该列上现有的索引），可能必须考虑到经过一段时间后列长度的变化。如果字段的大小有变化，应用就会受到影响，因为它必须修改字段宽度。

正是由于以下这些原因：定宽的存储空间可能导致表和相关索引比平常大出许多，还伴随着绑定变量问题，所以无论什么场合我都会避免使用 CHAR 类型。即便是对单字符的字段，我也想不出有什么必要使用 CHAR 类型，因为在这种情况下，这两种类型确实没有显著差异。VARCHAR2(1)和 CHAR(1)从任何方面来讲都完全相同。此时，使用 CHAR 类型并没有什么有说服力的理由，为了避免混淆，所以我“一律排斥“，即使是 CHAR(1)字段（即单字符字段）也不建议使用 CHAR 类型。

1. 字符串语法

这 4 种基本串类型的语法很简单，如表 12.-1 所示。

表 12.-1 4 种基本串类型

串类型	说明
VARCHAR2<SIZE><BYTE|CHAR>	<SIZE>是介于 1～4,000 之间的一个数，表示最多占用 4,000 字节的存储空间。在下一节中，我们将详细分析子句中 BYTE 和 CHAR 修饰符的显著区别和细微差别
CHAR(<SIZE><BYTE|CHAR>)	<SIZE>是介于 1～2,000 之间的一个数，表示最多占用 2,000 字节的存储空间
NVARCHAR2(<SIZE>)	<SIZE>是一个大于 0 的数，其上界由国家字符集指定
NCHAR(<SIZE>)	<SIZE>是一个大于 0 的数，其上界由国家字符集指定

2. 字节或字符

VARCHAR2 和 CHAR 类型支持两种指定长度的方法：

- 用字节指定：VARCHAR2(12. byte)。这能支持最多 12.字节的数据， 在一个多字节字符集中，这可能这是两个字符。
- 用字符指定：VARCHAR2(12. char)。这将支持最多 12.字符的数据，可能是多达 40 字节的信息。

使用 UTF8 之类的多字节字符集时，建议你在 VARCHAR2/CHAR 定义中使用 CHAR

修饰符，也就是说，使用 VARCHAR2(80 CHAR)，而不是 VARCHAR2(80),因为你的本意很可能是定义一个实际上能存储 80 字符数据的列。还可以使用会话参数或系统参数 NLS_LENGTH_SEMANTICS 来修改默认行为，即把默认设置 BYTE 改为 CHAR。我不建议在系统级修改这个设置，而应该只是在你的数据库模式安装脚本中把这个设置作为 ALTER SESSION 设置的一部分。只要应用需要数据库有某组特定的 NLS 设置，这必然是一个“不友好“的应用。一般来讲，这种应用无法与其他不要求这些设置（而依赖于默认设置）的应用一同安装到数据库上。

还要记住重要的一点：VARCHAR2 中存储的字节数上界是 4,000。不过，即使你指定了 VARCHAR2(4000 CHAR)，可能并不能在这个字段中放下 4,000 个字符。实际上，采用你选择的字符集时如果所有字符都要用 4 个字节来表示，那么这个字段中就只能放下 12.000 字符！

下面这个小例子展示了 BYTE 和 CHAR 之间的区别，并显示出上界的作用。我们将创建一个包括 3 列的表，前两列的长度分别是 1 字节和 1 字符，最后一列是 4,000 字符。需要说明，这个测试在一个多字节字符集数据库上完成，在此使用了字符集 AL32UTF8，这个字符集支持最新版本的 Unicode 标准，采用一种变长方式对每个字符使用 12.4 个字节进行编码：

```
ops$tkyte@O10GUTF> select *
  2    from nls_database_parameters
  3    where parameter = 'NLS_CHARACTERSET';

PARAMETER                      VALUE
------------------------------ --------------------
NLS_CHARACTERSET               AL32UTF8
ops$tkyte@O10GUTF> create table t
  2    ( a varchar2(1),
  3    b varchar2(12.char),
  4    c varchar2(4000 char)
  5 )
  6 /
Table created.
```

现在，如果想在这个表中插入一个 UTF 字符，这个字符长度为 2 个字节，可以观察到

以下结果：

```
ops$tkyte@O10GUTF> insert into t (a) values (unistr('\00d6'));

insert into t (a) values (unistr('\00d6'))

                         *

ERROR at line 1:

ORA-12899: value too large for column "OPS$TKYTE"."T"."A"

(actual: 2, maximum: 1)
```

这个例子展示了两点：

VARCHAR2(1)的单位是字节，而不是字符。这里确实只有一个 Unicode 字符，但是它在一个字节中放不下。

将应用从单字节定宽字符集移植到一个多字节字符集时，可能会发现原来在字段中能放下的文本现在却无法放下。

第二点的原因是，在一个单字节字符集中，包含 20 个字符的字符串长度就是 20 字节，完全可以在一个 VARCHAR2(20)中放下。不过，在一个多字节字符集中，20 个字符的字符串长度可以到达 80 字节（如果每个字符用 4 个字节表示），这样一来，20 个 Unicode 字符很可能无法在 20 个字节中放下。你可能会考虑将 DDL 修改为 VARCHAR2(20 CHAR)，或者在运行 DDL 创建表时使用前面提到的 NLS_LENGTH_SEMANTICS 会话参数。

如果字段原本就是要包含一个字符，在这个字段中插入一个字符时，可以观察到以下结果：

```
ops$tkyte@O10GUTF> insert into t (b) values (unistr('\00d6'));

12.row created.

ops$tkyte@O10GUTF> select length(b), lengthb(b), dump(b) dump from t;

      LENGTH(B)    LENGTHB(B)    DUMP

          ---------           ---------    --------------------

                  1                   2    Typ=12.Len=2: 195,150
```

这个 INSERT 成功了，而且可以看到，所插入数据的长度（LENGTH）就是一个字符，所有字符串函数都以字符为单位工作。这个字段的长度是一个字符，但是 LENGTHB 函数（字节长度）显示这个字段占用了 2 字节的存储空间，另外 DUMP 函数显示了这些字节到底是什么。这个例子展示了人们使用多字节字符集时遇到的一个相当常见的问题，即 VARCHAR2(N)并不一定存储 N 个字符，而只是存储 N 个字节。

人们经常遇到的另一个问题是：VARCHAR2 的最大字节长度为 4,000，而 CHAR 的最大字节长度为 2,000。

```
ops$tkyte@O10GUTF> declare
  2     l_data varchar2(4000 char);
  3     l_ch varchar2(12.char) := unistr( '\00d6' );
  4     begin
  5           l_data := rpad( l_ch, 4000, l_ch );
  6           insert into t ( c ) values ( l_data );
  7     end;
  8     /
declare
*
ERROR at line 1:
ORA-01461: can bind a LONG value only for insert into a LONG column
ORA-06512: at line 6
```

在此显示出，一个 4,000 字符的字符串实际上长度为 8,000 字节，这样一个字符串无法永久地存储在一个 VARCHAR2(4000 CHAR)字段中。这个字符串能放在 PL/SQL 变量中，因为在 PL/SQL 中 VARCHAR2 最大可以到达 32KB。不过，存储在表中时，VARCHAR2 则被硬性限制为最多只能存放 4,000 字节。我们可以成功地存储其中 2,000 个字符：

```
ops$tkyte@O10GUTF> declare
  2     l_data varchar2(4000 char);
  3     l_ch varchar2(12.char) := unistr( '\00d6' );
  4 begin
  5     l_data := rpad( l_ch, 2000, l_ch );
  6     insert into t ( c ) values ( l_data );
  7 end;
  8 /
```

```
PL/SQL procedure successfully completed.

ops$tkyte@O10GUTF> select length( c ), lengthb( c )
  2    from t
  3    where c is not null;

LENGTH(C)              LENGTHB(C)
----------             ----------
2000                   4000
```

如你所见，它占用了 4,000 字节的存储空间。

3. NVARCHAR2 和 NCHAR

为了完整地介绍字符串数据类型，下面来看 NVARCHAR2 和 NCHAR，它们有什么用呢？如果系统中需要管理和存储多种字符集，就可以使用这两个字符串类型。通常会有这样一种情况：在一个数据库中，主要字符集是一个单字节的定宽字符集（如 WE8ISO8859P1），但是还需要维护和存储一些多字节数据。许多系统都有遗留的数据，但是同时还要为一些新应用支持多字节数据；或者系统中大多数操作都需要单字节字符集的高效 率（如果一个串中每个字符可能存储不同数目的字节，与这个串相比，定宽字符串上的串操作效率更高），但某些情况下又需要多字节数据的灵活性。

NVARCHAR2 和 NCHAR 数据类型就支持这种需求。总的来讲，它们与相应的 VARCHAR2 和 CHAR 是一样的，只是有以下不同：

- 文本采用数据库的国家字符集来存储和管理，而不是默认字符集。
- 长度总是字符数，而 CHAR/VARCHAR2 可能会指定是字节还是字符。

在 Oracle9i 及以后的版本中，数据库的国家字符集有两个可取值：UTF8 或 AL16UTF16（9i 中是 UTF16，10g 中是 AL16UTF16）。这使得 NCHAR 和 NVARCHAR 类型很适于只存储多字节数据，这是对 Oracle 以前版本的一个改变（Oracle8i 及以前版本允许为国家字符集选择任何字符集）。

12.3 二进制串：RAW 类型

Oracle 除了支持文本，还支持二进制数据的存储。前面讨论了 CHAR 和 VARCHAR2 类 型需要进行字符集转换，而二进制数据不会做这种字符集转换。因此，二进制数据类型不适合存储用户提供的文本，而适于存储加密信息，加密数据不是“文本“， 而是原文本的一个二进制表示、包含二进制标记信息的字处理文档，等等。如果数据库不认为这些数据是”文本“，这些数据就应该采用一种二进制数据类型来存 储，另外不应该应用字符集转换的数据也要使用二进制数据类型存储。

Oracle 支持 3 种数据类型来存储二进制数据：

- RAW 类型，这是这一节强调的重点，它很适合存储多达 2,000 字节的 RAW 数据。

- BLOB 类型，它支持更大的二进制数据，我们将在本章“LOB 类型“一节中再做介绍。
- LONG RAW 类型，这是为支持向后兼容性提供的，新应用不应考虑使用这个类型。

二进制 RAW 类型的语法很简单：

```
RAW（<size>）
```

例如，以下代码创建了一个每行能存储 12.字节二进制信息的表：

```
ops$tkyte@ORA10GR1> create table t ( raw_data raw(12.) );

Table created.
```

从磁盘上的存储来看，RAW 类型与 VARCHAR2 类型很相似。RAW 类型是一个变长的二进制串，这说明前面创建的表 T 可以存储 1～12.字节的二进制数据。它不会像 CHAR 类型那样用空格填充。

处理 RAW 数据时，你可能会发现它被隐式地转换为一个 VARCHAR2 类型，也就是说，诸如 SQL*Plus 之类的许多工具不会直接显示 RAW 数据，而是会将其转换为一种十六进制格式来显示。在以下例子中，我们使用 SYS_GUID()在表中创建了一些二进制数据，SYS_GUID()是一个内置函数，将返回一个全局惟一的 12.字节 RAW 串（GUID 就代表全局惟一标识符，globally unique identifier）：

```
ops$tkyte@ORA10GR1> insert into t values ( sys_guid() );

12.row created.

ops$tkyte@ORA10GR1> select * from t;



RAW_D ATA

--------------------------------

FD1EB03D3718077BE030007F01002FF5
```

在此，你会马上注意到两点。首先，RAW 数据看上去就像是一个字符串。SQL*Plus 就是以字符串形式获取和打印 RAW 数据，但是 RAW 数据在磁盘上并不存储为字符串。SQL*Plus 不能在屏幕上打印任意的二进制数据，因为这可能对显示有严重的副作用。要记住，二进制数据可能包含诸如回车或换行等控制字符，还可能是一个 Ctrl+G 字符，这会导致终端发出“嘟嘟“的叫声。

其次，RAW 数据看上去远远大于 12.字节，实际上，在这个例子中，你会看到 32 个字符。这是因为，每个二进制字节都显示为两个十六进制字符。所存储的 RAW 数据其实长度就是 12.字节，可以使用 Oracle DUMP 函数确认这一点。在此，我“转储“了这个二进制串的值，并使用了一个可选参数来指定显示各个字节值时应使用哪一种进制。这里使用了基数 12.，从而能将转储的结果与前面的串进行比较：

```
ops$tkyte@ORA10GR1> select dump(raw_data,12.) from t;



DUMP(RAW_DATA,12.)

--------------------------------------------------------------------------------

Typ=23 Len=12.: fd,12.,b0,3d,37,12.,7,7b,e0,30,0,7f,12.0,2f,f5
```

DUMP 显示出，这个二进制串实际上长度为 12.字节（LEN=12.），另外还逐字节地显示了这个二进制数据。可以看到，这个转储显示与 SQL*Plus 将 RAW 数据获取为一个串时所执行的隐式转换是匹配的。另一个反向上（插入）也会执行隐式转换：

```
ops$tkyte@ORA10GR1> insert into t values ( 'abcdef' );

12.row created.
```

这不会插入串 abcdef，而会插入一个 3 字节的 RAW 数据，其字节分别是 AB、CD、EF，如果用十进制表示则为字节 171、205、239。如果试图使用一个包含非法 12.进制字符的串，就会收到一个错误消息：

```
ops$tkyte@ORA10GR1> insert into t values ( 'abcdefgh' );

insert into t values ( 'abcdefgh' )

                       *

ERROR at line 1:

ORA-01465: invalid hex number
```

RAW 类型可以加索引，还能在谓词中使用，它与其他任何数据类型有同样的功能。不过，必须当心避免不希望的隐式转换，而且必须知道确实会发生隐式转换。

在任何情况下我都喜欢使用显示转换，而且推荐这种做法，可以使用以下内置函数来执行这种操作：

- HEXTORAW：将十六进制字符串转换为 RAW 类型
- RAWTOHEX：将 RAW 串转换为十六进制串

SQL*Plus 将 RAW 类型获取为一个串时，会隐式地调用 RAWTOHEX 函数，而插入串时会隐式地调用 HEXTORAW 函数。应该避免隐式转换，而在编写代码时总是使用显示转换，这是一个很好的实践做法。所以前面的例子应该写作：

```
ops$tkyte@ORA10GR1> select rawtohex(raw_data) from t;



RAWTOHEX(RAW_DATA)
```

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

FD1EB03D3718077BE030007F01002FF5



ops$tkyte@ORA10GR1> insert into t values ( hextoraw('abcdef') );

12.row created.
```

12.4 数值类型

Oracle 10g 支持 3 种固有数据类型来存储数值。Oracle9i Release 2 及以前的版本只支持一种适合存储数值数据的固有数据类型。以下所列的数据类型中，NUMBER 类型在所有 Oracle 版本中都得到支持，后面两种类型是新的数据类型，只有 Oracle 10g 及以后的版本才支持：

- NUMBE：Oracle NUMBER 类型能以极大的精度存储数值，具体来讲，精度可达 38 位。其底层数据格式类似一种“封包小数“表示。Oracle NUMBER 类型是一种变长格式，长度为 0～22 字节。它可以存储小到 10e-130、大到（但不包括）10e126 的任何数值。这是目前最为常用的数值类型。
- BINARY_FLOAT：这是一种 IEEE 固有的单精度浮点数。它在磁盘上会占用 5 字节的存储空间：其中 4 个固定字节用于存储浮点数，另外还有一个长度字节。BINARY_FLOAT 能存储有 6 为精度、范围在～±1038.53 的数值。
- BINARY_DOUBLE：这是一种 IEEE 固有的双精度浮点数。它在磁盘上会占用 9 字节的存储空间：其中 8 个固定字节用于存储浮点数，还有一个长度字节。BINARY_DOUBLE 能存储有 12.位精度、范围在～±10308.25 的数值。

从以上简要的概述可以看到，Oracle NUMBER 类型比 BINARY_FLOAT 和 BINARY_DOUBLE 类型的精度大得多，但是取值范围却远远小于 BINARY_DOUBLE。也就是说，用 NUMBER 类型可以很精确地存储数值（有很多有效数字），但是用 BINARY_FLOAT 和 BINARY_DOUBLE 类型可以存储更小或更大的数值。下面举一个简单的例子，我们将用不同的数据类型来创建一个表，查看给定相同的输入时，各个列中会存储什么内容：

```
ops$tkyte@ORA10GR1> create table t
  2  ( num_col number,
  3  float_col binary_float,
  4  dbl_col binary_double
  5  )
  6  /
```

```
Table created.



ops$tkyte@ORA10GR1> insert into t ( num_col, float_col, dbl_col )
  2  values ( 1234567890.0987654321,
  3           1234567890.0987654321,
  4           1234567890.0987654321 );

12.row created.

ops$tkyte@ORA10GR1> set numformat 99999999999.99999999999

ops$tkyte@ORA10GR1> select * from t;

                NUM_COL                          FLOAT_COL                          DBL
_COL

 ---------------------------------- ---------------------------------- ----------------------------
---------

1234567890.09876543210       1234567940.00000000000 1234567890.09876540000
```

注意，NUM_COL 会按我们提供的输入原样返回同一个数。输入数中有效数字远远没有达到 38 位（这里提供了一个有 20 位有效数字的数），所以将完全保留原来的数。不过，使用新的 BINARY_FLOAT 类型时，FLOAT_COL 不能准确地表示这个数。实际上，它只正确保留了 7 位。DBL_COL 则要好多了，它正确地表示了这个数中的 12.位。不过，总的说来，由此可以很好地说明 BINARY_FLOAT 和 BINARY_DOUBLE 类型在金融应用中不适用！如果尝试不同的值，可能会看到不同的结果：

```
ops$tkyte@ORA10GR1> delete from t;

12.row deleted.



ops$tkyte@ORA10GR1> insert into t ( num_col, float_col, dbl_col )
  2  values ( 9999999999.9999999999,
  3  9999999999.9999999999,
  4  9999999999.9999999999 );
```

```
12.row created.



ops$tkyte@ORA10GR1> select * from t;

                NUM_COL                FLOAT_COL                      DB
L_COL

        ----------------------  ----------------------  ------------
------------

9999999999.99999999990 10000000000.00000000000     10000000000.00000000000
```

NUM_COL 又一次正确地表示了这个数，但是 FLOAT_COL 和 DBL_COL 却未能做到。这并不是说 NUMBER 类型能以“无限的“精度/准确性来存储数据，它的精度只不过相当大而已（但并不是无限的）。NUMBER 类型也有可能不正确地表示数值，这种情况很容易观察到：

```
ops$tkyte@ORA10GR1> delete from t;

12.row deleted.



ops$tkyte@ORA10GR1> insert into t ( num_col )

  2      values ( 123 * 1e20 + 123*12.-20 ) ;

12.row created.



ops$tkyte@ORA10GR1>                set                numformat
99999999999999999999999.999999999999999999999999

ops$tkyte@ORA10GR1> select num_col, 123*1e20, 123*12.-20 from t;

                                                    NUM_COL

                                  ---------------------------------------------

                                                   123*1E20

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

```
                                              123*12.-20
                              --------------------------------------------------
     1230000000000000000000000.0000000000000000000000000
     1230000000000000000000000.0000000000000000000000000
                                   .0000000000000000001230000
```

可以看到，如果把一个非常大的数（$123*12.^{20}$）和一个非常小的数（$123*12.^{-20}$）放在一起，就会丢失精度，因为这个算术运算需要的精度不止 38 位。就较大数（$123*12.^{20}$）本身而言，它能得到“忠实“的表示，较小数（$123*12.^{-20}$）也能精确地表示，但是较大数加上较小数的结果却不能如实表示。这不只是一个显示/格式化的问题，可以做以下验证：

```
ops$tkyte@ORA10GR1> select num_col from t where num_col = 123*1e20;
                                              NUM_COL
                              --------------------------------------------------
     1230000000000000000000000.0000000000000000000000000
```

NUM_COL 中的值等于 $123*12.^{20}$，而不是我们真正想插入的值。

12.4.1 NUMBER 类型的语法和用法

NUMBER 类型的语法很简单：

```
NUMBER( p,s )
```

在此 P 和 S 是可选的，用于指定：

- 精度（precision），或总位数。默认情况下，精度为 38 位，取值范围是 1～38 之间。也可以用字符*表示 38。
- 小数位置（scale），或小数点右边的位数。小数位数的合法值为-48～127，其默认值取决于是否指定了精度。如果没有知道精度，小数位数则默认有最大的取值区间。如果指定了精度，小数位数默认为 0（小数点右边一位都没有）。例如，定义为 NUMBER 的列会存储浮点数（有小数），而 NUMBER(38)只存储整数数据（没有小数），因为在第二种情况下小数位数默认为 0.

应该把精度和小数位数考虑为对数据的“编辑“，从某种程度上讲它们可以算是一种完整性工具。精度和小数位数根本不会影响数据在磁盘上如何存储，而只会影响允许有哪些值以及数值如何舍入（round）。例如，如果某个值超过了所允许的精度，Oracle 就会返回一个错误：

```
ops$tkyte@ORA10GR1> create table t ( num_col number(5,0) );

Table created.
```

```
ops$tkyte@ORA10GR1> insert into t (num_col) values ( 12345 );

12.row created.


ops$tkyte@ORA10GR1> insert into t (num_col) values ( 123456 );

insert into t (num_col) values ( 123456 )

                                *

ERROR at line 1:

ORA-01438: value larger than specified precision allows for this column
```

因此，可以使用精度来保证某些数据完整性约束。在这个例子中，NUM_COL 列不允许多于 5 位。

另一方面，小数位数可以用于控制数值的“舍入“，例如：

```
ops$tkyte@ORA10GR1> create table t ( msg varchar2(12.), num_col number(5,2) );

Table created.


ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '123.45', 123.45 );

12.row created.


ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '123.456', 123.456 );

12.row created.


ops$tkyte@ORA10GR1> select * from t;

    MSG          NUM_COL

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

```
123.45              123.45

123.456             123.46
```

可以注意到，尽管数值 123.456 超过了 5 位，但这一次插入成功了，没有报错。这是因为，这个例子中利用小数位数将 123.456“舍入“为只有两位小数，这就得到了 123.46，再根据精度来验证 123.46，发现满足精度要求，所以插入成功。不过，如果试图执行以下插入，则会失败：

```
ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '1234', 1234 );

insert into t (msg,num_col) values ( '1234', 1234 )

                                    *

ERROR at line 1:

ORA-01438: value larger than specified precision allows for this column
```

这是因为，数值 1234.00 的位数超过了 5 位。指定小数位数为 2 时，小数点左边最多只有 3 位，右边有 2 位。因此，这个数不满足精度要求。NUMBER(5,2)列可以存储介于 999.99～-999.99 之间的所有值。

允许小数位数在-84～127 之间变化，这好像很奇怪。为什么小数位数可以为负值，这有什么用意？其作用是允许对小数点左边的值舍入。就像 NUMBER(5,2)将值舍入为最接近 0.01 一样，NUMBER(5,-2)会把数值舍入为与之最接近的 100，例如：

```
ops$tkyte@ORA10GR1> create table t ( msg varchar2(12.), num_col number(5,-2) );

Table created.



ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '123.45', 123.45 );

12.row created.



ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '123.456', 123.456 );

12.row created.



ops$tkyte@ORA10GR1> select * from t;

MSG              NUM_COL
```

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

123.45                      100

123.456                     100
```

这些数舍入为与之最接近的 100.精度还是 5 位，但是现在小数点左边允许有 7 位（包括尾部的两个 0）：

```
ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '1234567', 1234567 );

12.row created.

ops$tkyte@ORA10GR1> select * from t;



MSG                 NUM_COL

----------                  ----------

123.45                        100

123.456                      100

1234567              1234600

ops$tkyte@ORA10GR1> insert into t (msg,num_col) values ( '12345678', 12345678 );

insert into t (msg,num_col) values ( '12345678', 12345678 )

                                                    *

ERROR at line 1:

ORA-01438: value larger than specified precision allows for this column
```

因此，精度指示了舍入后数值中允许有多少位，并使用小数位数来确定如何舍入。精度是一个完整性约束，而小数位数是一种“编辑“。

有一点很有意思，也很有用，NUMBER 类型实际上是磁盘上的一个变长数据类型，会占用 0～22 字节的存储空间。很多情况下，程序员会认为数值类型是一个定长类型，因为在使用 2 或 4 字节整数以及 4 或 8 字节单精度浮点数编程时，他们看到的往往就是定长类型。Oracle NUMBER 类型与变长字符串很类似。下面通过例子来看看如果数中包含不同数目的有效数字会发生什么情况。我们将创建一个包含两个 NUMBER 列的表，并用分别有 2、4、6、…、28 位有效数字的多个数填充第一列。然后再将各个值分别加 1，填充第二列：

```
ops$tkyte@ORA10GR1> create table t ( x number, y number );
```

```
Table created.

ops$tkyte@ORA10GR1> insert into t ( x )
  2     select to_number(rpad('9',rownum*2,'9'))
  3     from all_objects
  4     where rownum <= 12.;
12. rows created.

ops$tkyte@ORA10GR1> update t set y = x+1;
12. rows updated.
```

下面使用内置 VSIZE 函数，它能显示列占用多大的存储空间，从而可以看到每行中两个数的大小有怎样的差异：

```
ops$tkyte@ORA10GR1> set numformat 99999999999999999999999999999
ops$tkyte@ORA10GR1> column v1 format 99
ops$tkyte@ORA10GR1> column v2 format 99
ops$tkyte@ORA10GR1> select x, y, vsize(x) v1, vsize(y) v2
  2  from t order by x;

                             X                              Y
V1  V2
          ------------------------------ ------------------------------ ---
---
                            99                            100   2
  2
                          9999                          10000   3
 2
                        999999                        1000000   4
```

```
  2
                    99999999                      100000000     5
 2
                  9999999999                    10000000000     6
2
                999999999999                  1000000000000     7
2
              99999999999999                100000000000000     8     2
            9999999999999999              10000000000000000     9     2
          999999999999999999            1000000000000000000 12.     2
        99999999999999999999          100000000000000000000 12.     2
      9999999999999999999999        10000000000000000000000 12.     2
    999999999999999999999999      1000000000000000000000000 12.     2
  99999999999999999999999999    100000000000000000000000000 12.     2
9999999999999999999999999999  10000000000000000000000000000 12.     2

12. rows selected.
```

可以看到，随着 X 的有效数字数目的增加，需要越来越多的存储空间。每增加两位有效数字，就需要另外一个字节的存储空间。但是对各个数加 1 后得到的数总是只占 2 个字节。Oracle 存储一个数时，会存储尽可能少的内容来表示这个数。为此会存储有效数字、用于指定小数点位置的一个指数，以及有关数值符号的信息（正或负）。因此，数中包含的有效数字越多，占用的存储空间就越大。

最后一点解释了为什么有必要了解数值在变宽字段中存储方式。试图确定一个表的大小时（例如，明确一个表中的 12.000,000 行需要多少存储空间），就必须仔细考虑 NUMBER 字段。这些数会占 2 字节还是 12.字节？平均大小是多少？这样一来，如果没有代表性的测试数据，要准确地确定表的大小非常困难。你可以得到最坏情况下和最好情况下的大小，但是实际的大小往往是介于这二者之间的某个值。

12.4.2 BINARY_FLOAT/BINARY_DOUBLE 类型的语法和用法

Oracle 10g引入了两种新的数值类型来存储数据；在Oracle 10g之前的所有版本中都没有这两种类型。它们就是许多程序员过去常用的IEEE标准浮点数。要全面地了解这些数值类型是怎样的，以及它们如何实现，建议你阅读http://en.wikipedia.org/wiki/Floating-point。需要指出，在这个参考文档中对于浮点数的基本定义有以下描述（请注意我着重强调的部分）：

*浮点数是一个有理数子集中一个数的数字表示，通常用于在计算机上**近似**一个任意的实数。特别是，它表示一个整数或浮点数（有效数，或正式地说法是尾数）乘以一个底数（在计算机中通常是 2）的某个整数次幂（指数）。底数为 2 时，这就是二进制的科学计数法（通常的科学计数法底数为 12.）。*

浮点数用于近似数值；它们没有前面所述的内置 Oracle NUMBER 类型那么精确。浮点数常用在科学计算中，由于允许在硬件（CPU、芯片）上执行运算，而不是在 Oracle 子例程中运算，所以在多种不同类型的应用中都很有用。因此，如果在一个科学计算应用中执行实数处理，算术运算的速度会快得多，不过你可能不希望使用浮点数来存储金融信息。

要在表中声明这种类型的列，语法相当简单：

```
BINARY_FLOAT

BINARY_DOUBLE
```

仅此而已。这些类型没有任何选项。

12.4.3 非固有数据类型

除了 NUMBER、BINARY_FLOAT 和 BINARY_DOUBLE 类型，Oracle 在语法上还支持以下数值数据类型：

- NUMERIC(p,s)：完全映射至 NUMBER(p,s)。如果 p 未指定，则默认为 38.
- DECIMAL(p,s)或 DEC(p,s)：完全映射至 NUMBER(p,s)。如果 p 为指定，则默认为 38.
- INTEGER 或 INT：完全映射至 NUMBER(38)类型。
- SMALLINT：完全映射至 NUMBER(38)类型。
- FLOAT(b)：映射至 NUMBER 类型。
- DOUBLE PRECISION：映射至 NUMBER 类型。
- REAL：映射至 NUMBER 类型。

注意 这里我指出“在语法上支持“，这是指 CREATE 语句可以使用这些数据类型，但是在底层实际上它们都只是 NUMBER 类型。准确地将，Oracle 10g Release 1 及以后的版本中有 3 种固有数值格式，Oracle9i Release 2 及以前的版本中只有 1 种固有数值格式。使用其他的任何数值数据类型总是会映射到固有的 Oracle NUMBER 类型。

12.4.4 性能考虑

一般而言，Oracle NUMBER 类型对大多数应用来讲都是最佳的选择。不过，这个类型会带来一些性能影响。Oracle NUMBER 类型是一种软件数据类型，在 Oracle 软件本身中实现。我们不能使用固有硬件操作将两个 NUMBER 类型相加，这要在软件中模拟。不过，浮点数没有这种实现。将两个浮点数相加时，Oracle 会使用硬件来执行运算。

很容易看出这一点。如果创建一个表，其中包含大约 50,000 行，分别使用 NUMBER 和 BINARY_FLOAT/BINARY_DOUBLE 类型在其中放入同样的数据，如下：

```
ops$tkyte@ORA10G> create table t
```

```
  2     ( num_type number,
  3     float_type binary_float,
  4     double_type binary_double
  5     )
  6     /

Table created.



ops$tkyte@ORA10G> insert /*+ APPEND */ into t
  2           select rownum, rownum, rownum
  3           from all_objects
  4     /

48970 rows created.



ops$tkyte@ORA10G> commit;

Commit complete.
```

再对各种类型的列执行同样的查询，在此使用一个复杂的数学函数，如 NL（自然对数）。会观察到它们的 CPU 利用率存在显著差异：

```
select sum(ln(num_type)) from t

call     count       cpu    elapsed

-------  ------  --------  ----------

total        4      2.73        2.73

select sum(ln(float_type)) from t

call     count       cpu    elapsed

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

```
total          4       0.06          0.12.

select sum(ln(double_type)) from t

call     count       cpu    elapsed

-------  ------  --------  ----------

total          4       0.05          0.12.
```

Oracle NUMBER 类型使用的 CPU 时间是浮点数类型的 50 倍。不过，你要记住，从这 3 个查询中得到的答案并不完全相同！浮点数是数值的一个近似值，精度在 6～12.位之间。从 NUMBER 类型得到的答案比从浮点数得到的答案“精确“得多。但是如果你在对科学数据执行数据挖掘或进行复杂的数值分析，这种精度损失往往是可以接受的，另外可能会得到非常显著的性能提升。

注意 如果你对浮点数运算的具体细节以及所带来的精度损失感兴趣，可以参见http://docs.sum.com/source/806-3568/ncg_goldberg.html。

需要注意，我们可以鱼和熊掌兼得。通过使用内置的 CAST 函数，可以对 Oracle NUMBER 类型执行一种实时的转换，在对其执行复杂数学运算之前先将其转换为一种浮点数类型。这样一来，所用 CPU 时间就与使用固有浮点类型所用的 CPU 时间非常接近：

```
select sum(ln(cast( num_type as binary_double ) )) from t

call     count       cpu    elapsed

-------  ------  --------  ----------

total          4      0.12.          0.12.
```

这说明，我们可以非常精确地存储数据，如果需要提供速度，浮点类型则远远超过 Oracle NUMBER 类型，此时可以使用 CAST 函数来达到提速的目标。

12.5 LONG 类型

Oracle 中的 LONG 类型有两种：

- LONG 文本类型，能存储 2GB 的文本。与 VARCHAR2 或 CHAR 类型一样，存储在 LONG 类型中的文本要进行字符集转换。
- LONG RAW 类型，能存储 2GB 的原始二进制数据（不用进行字符集转换的数据）。

从 Oracle 6 开始就已经支持 LONG 类型，那时限制为只能存储 64KB 的数据。在 Oracle 7 中，提升为可以存储多达 2GB 的数据，但是等发布 Oracle 8 时，这种类型被 LOB 类型所取代，稍后将讨论 LOB 类型。

在此并不解释如何使用 LONG 类型，而是会解释为什么你不希望在应用中使用 LONG（或 LONG RAW）类型。首先要注意的是，Oracle 文档在如何处理 LONG 类型方面描述得很明确。Oracle SQL Reference 手册指出：

不要创建带 LONG 列的表，而应该使用 LOB 列（CLOB、NCLOB、BLOB）。支持 LONG 列只是为了保证向后兼容性。

12.5.1LONG 和 LONG RAW 类型的限制

LONG 和 LONG RAW 类型存在很多限制，如表 12.-2 所列。尽管这可能有点超前（现在还没有讲到 LOB 类型），不过在此我还是增加了一列，指出相应的 LOB 类型（用以取代 LONG/LONG RAW）类型是否也有同样的限制。

表 12.-2 LONG 类型与 LOB 类型的比较

LONG/LONG RAW 类型	CLOB/BLOB 类型
每个表中只能有一个 LONG 或 LONG RAW 列	每个表可以有最多 12.000 个 CLOB 或 BLOB 类型的列
定义用户定义的类型时，不能有 LONG/LONG RAW 类型的属性	用户定义的类型完成可以使用 CLOB 和 BLOB 类型
不能在 WHERE 子句中引用 LONG 类型	WHERE 子句中可以引用 LOB 类型，而且 DBMS_LOB 包中提供了大量函数来处理 LOB 类型
除了 NOT NULL 之外，完整性约束中不能引用 LONG 类型	完整性约束中可以引用 LOB 类型
LONG 类型不支持分布式事务	LOB 确实支持分布式事务
LONG 类型不能使用基本或高级复制技术来复制	LOB 完全支持复制
LONG 列不能在 GROUP BY、ORDER BY 或 CONNECT BY 子句中引用，也不能在使用了 DISTINCT、UNIQUE、INTERSECT、MINUS 或 UNION 的查询中使用	只要对 LOB 应用一个函数，将其转换为一个标量 SQL 类型，如 VARCHAR2、NUMBER 或 DATE，LOB 就可以出现在这些子句中
PL/SQL 函数/过程不能接受 LONG 类型的输入	PL/SQL 可以充分处理 LOB 类型
SQL 内置函数不能应用于 LONG 列（如 SUBSTR）	SQL 函数可以应用于 LOB 类型

CREATE TABLE AS SELECT 语句中不能使用	LOB 支持 CREATE TABLE AS SELECT
LONG 类型	
在包含 LONG 类型的表上不能使用 ALTER TABLE MOVE	可以移动包含 LOB 的表

可以看到，表 12.-2 很长；如果表中有一个 LONG 列，那么很多事情都不能做。对于所有新的应用，甚至根本不该考虑使用 LONG 类型。相反，应该使用适当的 LOB 类型。对于现有的应用，如果受到表 12.-2 所列的某个限制，就应该认真地考虑将 LONG 类型转换为相应的 LOB 类型。由于已经做了充分考虑来提供向后兼容性，所以编写为使用 LONG 类型的应用也能透明地使用 LOB 类型。

注意 无须多说，将生产系统从 LONG 修改为 LOB 类型之前，应当对你的应用执行一个全面的字典测试。

12.5.2 处理遗留的 LONG 类型

经常会问到这样一个问题：“那如何考虑 Oracle 中的数据字典呢？“数据字典中散布着 LONG 列，这就使得字典列的使用很成问题。例如，不能使用 SQL 搜索 ALL_VIEWS 字典视图来找出包含文本 HELLO 的所有视图：

```
ops$tkyte@ORA10G> select *
  2    from all_views
  3    where text like '%HELLO%';
where text like '%HELLO%'
      *
ERROR at line 3:
ORA-00932: inconsistent datatypes: expected NUMBER got LONG
```

这个问题并不只是 ALL_VIEWS 视图才有，许多视图都存在同样的问题：

```
ops$tkyte@ORA10G> select table_name, column_name
  2    from dba_tab_columns
  3    where data_type in ( 'LONG', 'LONG RAW' )
  4          and owner = 'SYS'
  5          and table_name like 'DBA%';
```

TABLE_NAME	COLUMN_NAME
DBA_VIEWS	TEXT
DBA_TRIGGERS	TRIGGER_BODY
DBA_TAB_SUBPARTITIONS	HIGH_VALUE
DBA_TAB_PARTITIONS	HIGH_VALUE
DBA_TAB_COLUMNS	DATA_DEFAULT
DBA_TAB_COLS	DATA_DEFAULT
DBA_SUMMARY_AGGREGATES	MEASURE
DBA_SUMMARIES	QUERY
DBA_SUBPARTITION_TEMPLATES	HIGH_BOUND
DBA_SQLTUNE_PLANS	OTHER
DBA_SNAPSHOTS	QUERY
DBA_REGISTERED_SNAPSHOTS	QUERY_TXT
DBA_REGISTERED_MVIEWS	QUERY_TXT
DBA_OUTLINES	SQL_TEXT
DBA_NESTED_TABLE_COLS	DATA_DEFAULT
DBA_MVIEW_ANALYSIS	QUERY
DBA_MVIEW_AGGREGATES	MEASURE
DBA_MVIEWS	QUERY
DBA_IND_SUBPARTITIONS	HIGH_VALUE
DBA_IND_PARTITIONS	HIGH_VALUE

```
DBA_IND_EXPRESSIONS                    COLUMN_EXPRESSION

DBA_CONSTRAINTS                        SEARCH_CONDITION

DBA_CLUSTER_HASH_EXPRESSIONS       HASH_EXPRESSION
```

那么问题到底出在哪里呢？如果你想在 SQL 中使用这些列，就需要将它们转换为一种对 SQL 友好的类型。可以使用一个用户定义的函数来做到这一点。以下例子展示了如何使用一个 LONG SUBSTR 函数来达到这个目的，这个函数允许将任何 4,000 字节的 LONG 类型转换为一个 VARCHAR2，以便用于 SQL。完成后，就能执行以下查询：

```
ops$tkyte@ORA10G> select *
  2    from (
  3           select owner, view_name,
  4                  long_help.substr_of( 'select text
  5                        from dba_views
  6                         where owner = :owner
  7                               and view_name = :view_name',
  8                                1, 4000,
  9                                'owner', owner,
 12.                               'view_name', view_name ) substr_of_view_text
 12.          from dba_views
 12.          where owner = user
 12. )
 12. where upper(substr_of_view_text) like '%INNER%'
 12. /
```

你已经将 VIEW_TEXT 列的前 4,000 字节由 LONG 转换为 VARCHAR2，现在可以对它使用谓词了。使用同样的技术，你还可以对 LONG 类型实现你自己的 INSTR、LIKE 函数。在这本书里，我只想说明如何得到一个 LONG 类型的子串。

我们要实现的包有以下规范：

```
ops$tkyte@ORA10G> create or replace package long_help
```

```
2 authid current_user
3 as
4     function substr_of
5           ( p_query in varchar2,
6           p_from in number,
7           p_for in number,
8           p_name12.in varchar2 default NULL,
9           p_bind12.in varchar2 default NULL,
12.          p_name2 in varchar2 default NULL,
12.          p_bind2 in varchar2 default NULL,
12.          p_name3 in varchar2 default NULL,
12.          p_bind3 in varchar2 default NULL,
12.          p_name4 in varchar2 default NULL,
12.          p_bind4 in varchar2 default NULL )
12. return varchar2;
12. end;
12. /
Package created.
```

注意在第 2 行上，我们指定了 AUTHID CURRENT_USER。 这使得这个包会作为调用者运行，拥有所有角色和权限。这一点很重要，原因有两个。首先，我们希望数据库安全性不要受到破坏，这个包只返回允许我们（调用 者）看到的列子串。其次，我们希望只在数据库中将这个包安装一次，就能一直使用它的功能；使用调用者权限可以保证这一点。如果我们使用 PL/SQL 的默认安全模型（定义者权限，define right），这个包会以所有者的权限来运行，这样一来，它就只能看到包所有者能看到的数据，这可能不包括允许调用者看到的数据集。

函数 SUBSTR_OF 的基本思想是取一个查询，这个查询最多只选择一行和一列：即我们感兴趣的 LONG 值。如果需要，SUBSTR_OF 会解析这个查询，为之绑定输入，并通过查询获取结果，返回 LONG 值中必要的部分。

包体（实现）最前面声明了两个全局变量。G_CURSOR 变量保证一个持久游标在会话期间一直打开。这是为了避免反复打开和关闭游标，并避免不必要地过多解析 SQL。第二个全局变量 G_QUERY 用于记住这个包中已解析的上一个 SQL 查询的文本。只要查询保持不变，就只需将其解析一次。因此，即使一个查询中查询了 5,000 行，只要我们传入这个函数的 SQL 查询不变，就只会有一个解析调用：

```
ops$tkyte@ORA10G> create or replace package body long_help

  2 as

  3

  4     g_cursor number := dbms_sql.open_cursor;

  5     g_query varchar2(32765);

  6
```

这个包中接下来是一个私有过程：BIND_VARIABLE，我们用这个过程来绑定调用者传入的输入。在此把它实现为一个单独的私有过程，这只是为了更容易一些；我们希望只在输入名不为 NULL（NOT NULL）时才绑定。并非在代码中对输入参数执行 4 次检查，而是只在这个过程中执行 1 次检查：

```
  7     procedure bind_variable( p_name in varchar2, p_value in varchar2 )

  8     is

  9     begin

 12.         if ( p_name is not null )

 12.         then

 12.                 dbms_sql.bind_variable( g_cursor, p_name, p_value );

 12.         end if;

 12. end;

 12.
```

下面是包体中 SUBSTR_OF 的具体实现。这个例程首先是包规范中指定的一个函数声明，以及一些局部变量的声明。L_BUFFER 用于返回值，L_BUFFER_LEN 用于保存长度（这是由一个 Oracle 提供的函数返回的）：

```
 12.

 12. function substr_of
```

```
12.         ( p_query in varchar2,
12.         p_from in number,
20          p_for in number,
21          p_name12.in varchar2 default NULL,
22          p_bind12.in varchar2 default NULL,
23          p_name2 in varchar2 default NULL,
24          p_bind2 in varchar2 default NULL,
25          p_name3 in varchar2 default NULL,
26          p_bind3 in varchar2 default NULL,
27          p_name4 in varchar2 default NULL,
28          p_bind4 in varchar2 default NULL )
29 return varchar2
30 as
31          l_buffer varchar2(4000);
32          l_buffer_len number;
33 begin
```

现在，代码所做的第一件事是对 P_FROM 和 P_FOR 输入执行一个合理性检查（sanity check）。P_FROM 必须是一个大于或等于 1 的数，P_FOR 必须介于 1～4,000 之间，这与内置函数 SUBSTR 的参数限制是类似的：

```
34          if ( nvl(p_from,0) <= 0 )
35          then
36                  raise_application_error
37                  (-20002, 'From must be >= 1 (positive numbers)' );
38          end if;
39          if ( nvl(p_for,0) not between 12.and 4000 )
```

```
40      then
41          raise_application_error
42          (-20003, 'For must be between 12.and 4000' );
43      end if;
44
```

接下来，我们要查看是否得到一个需要解析的新查询。如果已解析的上一个查询与当前查询相同，就可以跳过这一步。必须指出有一点很重要，在第 47 行上验证传入的 P_QUERY 必须是一个 SELECT，我们只用这个包执行 **SQL** SELECT 语句。以下检查为我们完成了这个验证：

```
45      if ( p_query <> g_query or g_query is NULL )
46      then
47          if ( upper(trim(nvl(p_query,'x'))) not like 'SELECT%')
48          then
49              raise_application_error
50              (-20001, 'This must be a select only' );
51          end if;
52          dbms_sql.parse( g_cursor, p_query, dbms_sql.native );
53          g_query := p_query;
54      end if;
```

我们已经准备好将输入绑定到这个查询。插入的所有非 NULL 名都会“绑定”到查询，所以当执行查询时，它会找到正确的行：

```
55      bind_variable( p_name1, p_bind1 );
56      bind_variable( p_name2, p_bind2 );
57      bind_variable( p_name3, p_bind3 );
58      bind_variable( p_name4, p_bind4 );
59
```

现在执行查询，并获取行。然后使用 DBMS_SQL.COLUMN_VALUE_LONG，从而抽

取出 LONG 中必要的子串，并将其返回：

```
60          dbms_sql.define_column_long(g_cursor, 1);
61          if (dbms_sql.execute_and_fetch(g_cursor)>0)
62          then
63                  dbms_sql.column_value_long
64                  (g_cursor, 1, p_for, p_from-1,
65                  l_buffer, l_buffer_len );
66          end if;
67          return l_buffer;
68 end substr_of;
69
70 end;
71 /

Package body created.
```

大功告成，你现在可以对数据库中如何遗留的 LONG 列使用这个包，这样就能执行很多以前不可能执行的 WHERE 子句操作。例如，现在你可以找出模式中 HIGH_VALUE 包含 2003 年的所有分区：

```
ops$tkyte@ORA10G> select *
  2     from (
  3             select table_owner, table_name, partition_name,
  4                     long_help.substr_of
  5                     ( 'select high_value
  6                     from all_tab_partitions
  7                     where table_owner = :o
  8                             and table_name = :n
```

```
   9                         and partition_name = :p',
  12.                 1, 4000,
  12.                 'o', table_owner,
  12.                 'n', table_name,
  12.                 'p', partition_name ) high_value
  12.           from all_tab_partitions
  12.           where table_name = 'T'
  12.                 and table_owner = user
  12.           )
  12. where high_value like '%2003%'

TABLE_OWN     TABLE       PARTIT      HIGH_VALUE
---------      -----        ------        ------------------------------
OPS$TKYTE     T             PART1      TO_DATE(' 2003-03-12. 00:00:00'
                                          , 'SYYYY-MM-DD HH24:MI:SS', 'N
                                          LS_CALENDAR=GREGORIAN')
OPS$TKYTE     T             PART2      TO_DATE(' 2003-03-12. 00:00:00'
                                          , 'SYYYY-MM-DD HH24:MI:SS', 'N
                                          LS_CALENDAR=GREGORIAN')
```

通过使用同样的技术，即取一个返回一行和一列（LONG 列）的查询，并在一个函数中处理该查询的结果，你就能根据需要实现自己的 INSTR、LIKE 等函数。

这个实现在 LONG 类型上能很好地工作，但是在 LONG RAW 类型上却不能工作。LONG RAW 不能分段地访问（DBMS_SQL 包中没有 COLUMN_VALUE_LONG_RAW 之类的函数）。幸运的是，这个限制不算太严重，因为字典中没有用 LONG RAW，而且很少需要对 LONG RAW“取子串”来完成搜索。不过，如果确实需要这么做，你可能根本不会使用 PL/SQL 来实现，除非 LONG RAW 小于或等于 32KB，因为 PL/SQL 本身中没有任何方法来处理超过 32KB 的 LONG RAW。对此，必须使用 Java、C、C++、Visual Basic 或某种其他语言。

还有一种方法，可以使用 TO_LOB 内置函数和一个全局临时表，将 LONG 或 LONG RAW 临时地转换为 CLOB 或 BLOB。为此，PL/SQL 过程可以如下：

```
Insert into global_temp_table ( blob_column )

select to_lob(long_raw_column) from t where...
```

这在偶尔需要处理单个 LONG RAW 值的应用中能很好地工作。不过，你可能不希望不断地这样做，原因是为此需要做的工作太多了。如果你发现自己需要频繁地求职于这种技术，就应该干脆将 LONG RAW 一次性转换为 BLOB，然后处理 BLOB。

12.6 DATE、TIMESTAMP 和 INTERVAL 类型

Oracle 固有数据类型 DATE、TIMESTAMP 和 INTERVAL 是紧密相关的。DATE 和 TIMESTAMP 类型存储精度可变的固定日期/时间。INTERVAL 类型可以很容易地存储一个时间量，如“8 个小时”或“30 天”。将两个日期相减，就会得到一个时间间隔（INTERVAL）；例如，将 8 小时间隔加到一个 TIMESTAMP 上，会得到 8 小时以后的一个新的 TIMESTAMP。

Oracle 的很多版本中都支持 DATE 数据类型，这甚至可以追溯到我最早使用 Oracle 的那个年代，也就是说，至少在 Oracle 5 中（可能还更早）就已经支持 DATE 数据类型。TIMESTAMP 和 INTERVAL 类型相对来讲算是后来者，因为它们在 Oracle9i Release 1 中才引入。由于这个简单的原因，你会发现 DATE 数据类型是存储日期/时间信息最为常用的类型。但是许多新应用都在使用 TIMESTAMP 类型，这有两个原因：一方面它支持小数秒，而 DATE 类型不支持；另一方面 TIMESTAMP 类型支持时区，这也是 DATE 类型力所不能及的。

我们先来讨论 DATE/TIMESTAMP 格式及其使用，然后介绍以上各个类型。

12.6.1 格式

这里我不打算全面介绍 DATE、TIMESTAMP 和 INTERVAL 格式的方方面面。这在 Oracle SQL Reference 手册中有很好的说明，任何人都能免费得到这本手册。你可以使用大量不同的格式，很好地了解这些格式至关重要。强烈推荐你先好好研究一下各种格式。

在此我想讨论这些格式会做什么，因为关于这个主题存在许多误解。这些格式用于两个目的：

- 以某种格式对数据库中的数据进行格式化，以满足你的要求。
- 告诉数据库如何将一个输入串转换为 DATE、TIMESTAMP 或 INTERVAL。

仅此而已。多年来我观察到的一个常见的误解是，使用的格式会以某种方式影响磁盘上存储的数据，并且会影响数据如何具体地存储。*格式对数据如何存储根本没有任何影响。格式只是用于将存储 DATE 所用的二进制格式转换为一个串，或者将一个串转换为用于存储 DATE 的二进制格式。对于 TIMESTAMP 和 INTERVAL 也是如此。*

关于格式，我的建议就是：应该使用格式。将一个表示 DATE、TIMESTAMP 或 INTERVAL 的 串发送到数据库时就可以使用格式。不要依赖于默认日期格式，默认格式会（而且很可能）在将来每个时刻被另外每个人所修改。如果你依赖于一个默认日期格式，而这个默认格式有了变化，你的应用可能就会受到影响。如果无法转换日期，应用可能会向最终用户返回一个错误；或者更糟糕的是，它可能会悄悄地插入错误的数 据。考虑以下 INSERT 语句，它依赖于一个默认的日期掩码：

```
Insert into t ( date_column ) values ( '01/02/03' );
```

假设应用依赖于必须有默认日期掩码 DD/MM/YY。这就表示 2003 年 2 月 1 日（假设代码在 2000 年之后执行，不过稍后还会再来讨论有关的问题）。现在，假设有人认为正确而且适当的日期格式应该是 MM/DD/YY。突然之间，原来的日期就会变成 2003 年 1 月 2 日。或者有人认为 YY/MM/DD 才对，现在的日期就会变成 2001 年 2 月 3 日。简单地说，如果日期串没有带相应的日期格式，就会有多种解释方法。这个 INSERT 语句最好写作：

```
Insert into t ( date_column ) values ( to_date( '01/02/03', 'DD/MM/YY' ) );
```

按我的观点，更好的做法是：

```
Insert into t ( date_column ) values ( to_date( '01/02/2003', 'DD/MM/YYYY' ) );
```

也就是说，它必须使用一个 4 字符的年份。并不算太久以前，整个行业都领教了捡芝麻而丢西瓜的切肤之痛，我们原本想在存储年份时“节省”2 个字节，可是这带来了重重问题，为了解决这些问题，相应地修补软件，想想看我们花了多少时间和精力。可是，随着时间的推移，我们好像又好了伤疤忘了疼。现如今，2005 年都过去了，如果还不使用 4 字符的年份实在说不过去！

从数据库取出的数据也同样存在上述问题。如果你执行 SELECT DATE_COLUMN FROM T，并在应用中把这一列获取到一个串中，就应该对其应用一个显示的日期格式。不论你的应用期望何种格式，都应该在这里显式指定。否则，如果将来某个时刻有人修改了默认日期格式，你的应用就可能会崩溃，或者有异样的表现。

接下来，我们来更详细地介绍各种日期数据类型。

12.6.2 DATE 类型

DATE 类型是一个 7 字节的定宽日期/时间数据类型。它总是包含 7 个属性，包括：世纪、世纪中哪一年、月份、月中的哪一天、小时、分钟和秒。Oracle 使用一种内部格式来表示这个信息，所以它并不是存储 20，05，06，25，12.，01，00 来表示 2005 年 6 月 25 日 12.:01:00。通过使用内置 DUMP 函数，可以看到 Oracle 实际上会存储以下内容：

```
ops$tkyte@ORA10G> create table t ( x date );

Table created.



ops$tkyte@ORA10G> insert into t (x) values

  2     ( to_date( '25-jun-2005 12.:01:00',

  3               'dd-mon-yyyy hh24:mi:ss' ) );

12.row created.



ops$tkyte@ORA10G> select x, dump(x,12.) d from t;
```

```
X                    D

---------            ----------------------------------

25-JUN-05        Typ=12. Len=7: 120,105,6,25,12.,2,1
```

世纪和年份字节（DUMP 输出中的 120,105）采用一种“加 100”（excess-100）表示法来存储。必须将其减去 100 来确定正确的世纪和年份。之所以采用加 100 表示法，这是为了支持 BC 和 AD 日期。如果从世纪字节减去 100 得到一个负数，则是一个 BC 日期，例如：

```
ops$tkyte@ORA10G> insert into t (x) values

  2     ( to_date( '01-jan-4712bc',

  3            'dd-mon-yyyybc hh24:mi:ss' ) );

12.row created.



ops$tkyte@ORA10G> select x, dump(x,12.) d from t;

X                    D

---------            ----------------------------------

25-JUN-05     Typ=12. Len=7: 120,105,6,25,12.,2,1

01-JAN-12.      Typ=12. Len=7: 53,88,12.12.12.12.1
```

因此，插入 01-JAN-4712BC 时，世纪字节是 53，而 53-100=-47，这才是我们插入的真实世纪。由于这是一个负数，所以我们知道它是一个 BC 日期。这种存储格式还允许日期已一种二进制方式自然地排序。由于 4712 BC 小于 4710 BC，我们希望能有一种支持这种顺序的二进制表示。通过转储这两个日期，可以看到 01-JAN-4710BC 比 4712 BC 中的同一天“更大”，所以它们确实能正确地排序，并很好地进行比较：

```
ops$tkyte@ORA10G> insert into t (x) values

  2     ( to_date( '01-jan-4710bc',

  3            'dd-mon-yyyybc hh24:mi:ss' ) );

12.row created.



ops$tkyte@ORA10G> select x, dump(x,12.) d from t;
```

```
X                    D

---------            ---------------------------------

25-JUN-05       Typ=12. Len=7: 120,105,6,25,12.,2,1

01-JAN-12.      Typ=12. Len=7: 53,88,12.12.12.12.1

01-JAN-12.      Typ=12. Len=7: 53,90,12.12.12.12.1
```

接下来两个字段是月份和日字节，它们会自然地存储，不做如何修改。因此，6 月 25 日的月份字节就是 6，日字节是 25.小时、分钟和秒字段采用“加 1”（excess-1）表示法存储，这说明必须将各个部分减 1，才能得到实际的时间。因此，午夜 0 点在日期字段中就表示为 12.12.1。

这种 7 字节格式能自然地排序。你已经看到了，这一个 7 字节字段，可以采用一种二进制方式按从小到大（或从大到小）的顺序非常高效地进行排序。另外，这种结构允许很容易地进行截断，而无需把日期转换为另外某种格式。例如，要截断刚才存储的日期（25-JUN-2005 12.:01:00）来得到日信息（去掉小时、分钟和秒字段），这相当简单。只需将尾部的 3 个字节设置为 12.12.1，就能很好地清除时间分量。考虑一个全新的表 T，并执行以下插入：

```
ops$tkyte@ORA10G> create table t ( what varchar2(12.), x date );

Table created.

ops$tkyte@ORA10G> insert into t (what, x) values

  2     ( 'orig',

  3     to_date( '25-jun-2005 12.:01:00',

  4     'dd-mon-yyyy hh24:mi:ss' ) );

12.row created.



ops$tkyte@ORA10G> insert into t (what, x)

  2             select 'minute', trunc(x,'mi') from t

  3     union all

  4             select 'day', trunc(x,'dd') from t

  5     union all
```

```
  6          select 'month', trunc(x,'mm') from t
  7    union all
  8          select 'year', trunc(x,'y') from t
  9 /

4 rows created.

ops$tkyte@ORA10G> select what, x, dump(x,12.) d from t;

WHAT          X                D
--------      ---------        ---------------------------------
orig          25-JUN-05 Typ=12. Len=7: 120,105,6,25,12.,2,1
minute        25-JUN-05 Typ=12. Len=7: 120,105,6,25,12.,2,1
day           25-JUN-05 Typ=12. Len=7: 120,105,6,25,12.12.1
month         01-JUN-05 Typ=12. Len=7: 120,105,6,12.12.12.1
year          01-JAN-05 Typ=12. Len=7: 120,105,12.12.12.12.1
```

要把这个日期截断，只取到年份，数据库所要做的只是在后 5 个字节上置 1，这是一个非常快速的操作。现在我们就有一个可排序、可比较的 DATE 字段，它能截断到年份级，而且我们可以尽可能高效地做到这一点。不过，许多人并不是使用 TRUNC，而是在 TO_CHAR 函数中使用一个日期格式。例如，他们会这样用：

```
Where to_char(date_column,'yyyy') = '2005'
```

而不是

```
Where trunc(date_column,'y') = to_date('01-jan-2005','dd-mon-yyyy')
```

后者不仅有更出色的表现，而且占有的资源更少。如果建立 ALL_OBJECTS 的一个副本，另存储其中的 CREATED 列：

```
ops$tkyte@ORA10G> create table t
  2 as
  3    select created from all_objects;

Table created.
```

```
ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T' );

PL/SQL procedure successfully completed.
```

然后，启用 SQL_TRACE，我们反复使用上述两种技术查询这个表，可以看到以下结果：

```
select count(*)

from

        t where to_char(created,'yyyy') = '2005'



call                count           cpu       elapsed          disk         query       current
        rows

-------             ------          --------  ----------       ----------   ----------  -
---------       ----------

Parse                   4          0.01          0.05             0            0           0
        0

Execute                 4          0.00          0.00             0            0           0
        0

Fetch                   8          0.41          0.59             0          372           0
        4

-------             ------          --------  ----------       ----------   ----------  -
---------       ----------

total                  12.         0.42          0.64             0          372           0
         4



select count(*)

from

        t where trunc(created,'y') = to_date('01-jan-2005','dd-mon-yyyy')

call                count           cpu       elapsed          disk         query       current
        rows

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

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

Parse        4      0.00       0.00          0          0          0
      0

Execute      4      0.00       0.00          0          0          0
      0

Fetch        8      0.04       0.12.         0        372          0
      4

------- ------ -------- ---------- ---------- ---------- -
--------- ----------

total       12.     0.04       0.12.         0        372          0
      4
```

可以看到存在明显的差异。与使用 TRUNC 相比，使用 TO_CHAR 所用的 CPU 时间与前者相差一个数量级（即相差 12.倍）。这是因为 TO_CHAR 必须把日期转换为一个串，这要使用一个更大的代码路径，并利用当前的所有 NLS 来完成这个工作。然后必须执行一个串与串的比较。另一方面，TRUNC 只需把后 5 个字节设置为 1.然后将两个 7 字节的二进制数进行比较，就大功告成了。因此，如果只是要截断一个 DATE 列，你将应该避免使用 TO_CHAR。

另外，甚至要尽可能完全避免对 DATE 列应用函数。把前面的例子再进一步，可以看到其目标是获取 2005 年的所有数据。那好，如果 CREATED 上有一个索引，而且表中只有很少一部分 CREATED 值是 2005 年的时间，会怎么样呢？我们可能希望能够使用这个索引，为此要使用一个简单的谓词来避免在数据库列上应用函数：

```
select count(*) from t

where created >= to_date('01-jan-2005','dd-mon-yyyy')

      and created < to_date('01-jan-2006','dd-mon-yyyy');
```

这样有两个好处：

- 这样一来就可以考虑使用 CREATED 上的索引了。
- 根本无需调用 TRUNC 函数，这就完全消除了相应的开销。

这里使用的是区间比较而不是 TRUNC 或 TO_CHAR，这种技术同样适用于稍后讨论的 TIMESTAMP 类型。如果能在查询中避免对一个数据库列应用函数，就应该全力这样做。一般来讲，避免使用函数会有更好的性能，而且允许优化器在更多的访问路径中做出选择。

1. 向 DATE 增加或减去时间

经常有人问我这样一个问题：“怎么向一个 DATE 类型增加时间，或者从 DATE 类型减去时间？”例如，如何向一个 DATE 增加 1 天，或 8 个小时，或者一年，或者一个月，等等？对此，常用的技术有 3 种：

- 向 DATE 增加一个 NUMBER。把 DATE 加 1 是增加 1 天的一种方法。因此，向 DATE 增加 12.24 就是增加 1 个小时，依此类推。
- 可以使用稍后将介绍的 INTERVAL 类型来增加时间单位。INTERVAL 类型支持两种粒度：年和月，或日/小时/分钟/秒。也就是说，可以是几年和几个月的一个时间间隔，也可以是几天、几小时、几分钟和几秒的一个时间间隔。
- 使用内置的 ADD_MONTHS 函数增加月。由于增加一个月往往不像增加 28～31 天那么简单，为了便于增加月，专门实现了这样一个函数。

表 12.-3 展示了向一个日期增加 N 个时间单位可用的技术（当然，也可以利用这些技术从一个日期减去 N 个时间单位）。

表 12.-3　向日期增加时间

时间单位	操作	描述
	DATE + n/24/60/60	一天有 86,400 秒。由于加 1 就是增加 1 天，所以加 12.86400 就是向一个日期增
N 秒	DATE + n/86400	加 1 秒。我更喜欢用 n/24/60/60 技术而不是 12.86400 技术。它们是等价的。
	DATE+NUMTODSINTERVAL	更可读的一种方法是使用 NUMTODSINTERVAL（日/秒数间隔）
	(n,'second')	函数来增加 N 秒
	DATE + n/24/60	一天有 12.440 分钟，因此加 12.1440 就是向一个 DATE 增加 1 分钟。更可读的
N 分钟	DATE + n/1440	一种方法是使用 NUMTODSINTERVAL 函数
	DATE+NUMTODSINTERVAL	
	(n,'minute')	
	DATE + n/24	
N 小时	DATE+NUMTODSINTERVAL	一天有 24 个小时，因此增加 12.24 就是向一个 DATE 增加 1 小时。更可读的
	(n,'hour')	一种方法是使用 NUMTODSINTERVAL 函数
N 天	DATE + n	向 DATE 直接加 N，就是增加或减去 N 天
N 周	DATE + 7*n	一周有 7 天，所以只需将周数乘以 7 就是要增加或减去的天数

N 月　　ADD_MONTHS(DATE,n)　　可以使用 ADD_MONTHS 内置函数或者向 DATE 增加一个 N 个月的间隔。

　　DATE+NUMTOYMINTERVAL　　请参考稍后有关 DATE 使用月间隔做出的重要警告

　　(n,'month')

N 年　　ADD_MONTHS(DATE,12.*n)　　可以使用 ADD_MONTHS 内置函数，增加或减去 N 年时将参数指定为 12.*n。

　　DATE+NUMTOYMINTERVAL　　利用年间隔可以达到类似的目标，不过请参考稍后有关日期使用年间隔做出的

　　(n,'year')　　重要警告

总的来讲，使用 Oracle DATE 类型时，我有以下建议：

- 使用 NUMTODSINTERVAL 内置函数来增加小时、分钟和秒。
- 加一个简单的数来增加天。
- 使用 ADD_MONTHS 内置函数来增加月和年。

我建议不要使用 NUMTOYMINTERVAL 函数。其原因与这个函数如何处理月末日期有关。

ADD_MONTHS 函数专门处理月末日期。它实际上会为我们完成日期的“舍入”；例如，如果向一个有 31 天的月增加 1 个月，而且下一个月不到 31 天，ADD_MONTHS 就会返回下一个月的最后一天。另外，向一个月的最后一天增加 1 个月会得到下一个月的最后一天。向有 30 天（或不到 30 天）的一个月增加 1 个月时，可以看到：

```
ops$tkyte@ORA10G> alter session set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';

Session altered.



ops$tkyte@ORA10G> select dt, add_months(dt,1)

  2    from (select to_date('29-feb-2000','dd-mon-yyyy') dt from dual )

  3    /

DT                          ADD_MONTHS(DT,1)

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

```
29-feb-2000 00:00:00 31-mar-2000 00:00:00



ops$tkyte@ORA10G> select dt, add_months(dt,1)

  2    from (select to_date('28-feb-2001','dd-mon-yyyy') dt from dual )

  3    /

DT                          ADD_MONTHS(DT,1)

--------------------        --------------------

28-feb-2001 00:00:00 31-mar-2001 00:00:00



ops$tkyte@ORA10G> select dt, add_months(dt,1)

  2    from (select to_date('30-jan-2001','dd-mon-yyyy') dt from dual )

  3    /

DT                          ADD_MONTHS(DT,1)

--------------------        --------------------

30-jan-2001 00:00:00 28-feb-2001 00:00:00



ops$tkyte@ORA10G> select dt, add_months(dt,1)

  2    from (select to_date('30-jan-2000','dd-mon-yyyy') dt from dual )

  3    /

DT                          ADD_MONTHS(DT,1)

--------------------        --------------------

30-jan-2000 00:00:00 29-feb-2000 00:00:00
```

看到了吗？向 2000 年 2 月 29 日增加 1 个月，得到的是 2000 年 3 月 31 日。2 月 29 日是该月的最后一天，所以 ADD_MONTHS 返回了下一个月的最后一天。另外，注意向 2000 年和 2001 年的 1 月 30 日增加 1 个月时，会分别得到 2000 年和 2001 年 2 月的最后一天（分

别是 2 月 29 日和 2 月 28 日）。

如果与增加一个间隔的做法相比较，会看到完全不同的结果：

```
ops$tkyte@ORA10G> select dt, dt+numtoyminterval(1,'month')

  2     from (select to_date('29-feb-2000','dd-mon-yyyy') dt from dual )

  3     /

DT                          DT+NUMTOYMINTERVAL(1

--------------------        --------------------

29-feb-2000 00:00:00 29-mar-2000 00:00:00



ops$tkyte@ORA10G> select dt, dt+numtoyminterval(1,'month')

  2     from (select to_date('28-feb-2001','dd-mon-yyyy') dt from dual )

  3     /

DT                          DT+NUMTOYMINTERVAL(1

--------------------        --------------------

28-feb-2001 00:00:00 28-mar-2001 00:00:00
```

注意，得到的日期并不是下一个月的最后一天，而只是下一个月的同一天。有人认为这种行为是可以接受的，但是请考虑一下，如果下一个月没有这么多天会怎么样：

```
ops$tkyte@ORA10G> select dt, dt+numtoyminterval(1,'month')

  2 from (select to_date('30-jan-2001','dd-mon-yyyy') dt from dual )

  3 /

select dt, dt+numtoyminterval(1,'month')

           *

ERROR at line 1:

ORA-01839: date not valid for month specified

ops$tkyte@ORA10G> select dt, dt+numtoyminterval(1,'month')
```

```
  2 from (select to_date('30-jan-2000','dd-mon-yyyy') dt from dual )

  3 /

select dt, dt+numtoyminterval(1,'month')

           *

ERROR at line 1:

ORA-01839: date not valid for month specified
```

根据我的经验，这是由于这个原因，所以一般来讲不可能在日期算术运算中使用月间隔。对于年间隔也存在一个类似的问题：如果向 2000 年 2 月 29 日增加 1 年，也会得到一个运行时错误，因为没有 2001 年 2 月 29 日。

2. 得到两个日期之差

还有一个常被问到的问题：“我怎么得到两个日期之差？”这个问题看上去似乎很简单：只需要相减就行了。这会返回表示两个日期相隔天数的一个数。另外，还可以使用内置函数 MONTHS_BETWEEN，它会返回表示两个日期相隔月数的一个数（包括月小数）。最后，利用 INTERVAL 类型，你还能用另一个方法来查看两个日期之间的逝去时间。以下 SQL 查询分别展示了将两个日期相减的结果（显示两个日期之间的天数），使用 MONTHS_BETWEEN 函数的结果，然后是使用 INTERVAL 类型的两个函数的结果：

```
ops$tkyte@ORA10G> select dt2-dt1 ,

  2           months_between(dt2,dt1) months_btwn,

  3           numtodsinterval(dt2-dt1,'day') days,

  4           numtoyminterval(months_between(dt2,dt1),'month') months

  5     from (select to_date('29-feb-2000 01:02:03','dd-mon-yyyy hh24:mi:ss') dt1,

  6           to_date('12.-mar-2001 12.:22:33','dd-mon-yyyy hh24:mi:ss') dt2

  7           from dual )



  DT2-DT1     MONTHS_BTWN             DAYS                          MONTHS

    ----------    -----------                    -------------------------------    -------------

380.430903     12..5622872  +000000380                                12.:20:30.000000000
+000000001-00
```

这些都是“正确”的值，但是对我们来说都没有大用。大多数应用都更愿意显示日期

之间相隔的年数、月数、天数、小时数、分钟数和秒数。通过使用前述函数的一个组合，就可以实现这个目标。我们将选出两个间隔：一个是年和月间隔，另一个是日/小时/分钟/秒间隔。我们使用 MONTHS_BETWEEN 内置函数来确定两个日期之间相隔的月数（包括小数），然后使用 NUMTOYMINTERVAL 内置函数将这个数转换为年数和月数。另外，使用 TRUNC 得到两个日期相隔月数中的整数部分，再使用 ADD_MONTHS 内置函数将 dt1 增加 12.个月（这会得到‘28-feb-2001 01:02:03），再从两个日期中的较大者（dt2）减去这个计算得到的日期，从而得到两个日期之间的天数和小时数：

```
ops$tkyte@ORA10G> select numtoyminterval
  2           (months_between(dt2,dt1),'month')
  3           years_months,
  4           numtodsinterval
  5                (dt2-add_months( dt1, trunc(months_between(dt2,dt1)) ),
  6                'day' )
  7           days_hours
  8     from (select to_date('29-feb-2000 01:02:03','dd-mon-yyyy hh24:mi:ss') dt1,
  9                to_date('12.-mar-2001 12.:22:33','dd-mon-yyyy hh24:mi:ss') dt2
 12.        from dual )
 12. /
YEARS_MONTHS    DAYS_HOURS
--------------              ------------------------------
+000000001-00       +000000015 12.:20:30.000000000
```

现在就很清楚了，这两个日期之间相隔 1 年、12.天、12.小时、20 分钟和 30 秒。

12.6.3 TIMESTAMP 类型

TIMESTAMP 类型与 DATE 非常类似，只不过另外还支持小数秒和时区。以下 3 小节将介绍 TIMESTAMP 类型：其中一节讨论只支持小数秒而没有时区支持的情况，另外两小节讨论存储有时区支持的 TIMESTAMP 的两种方法。

1. TIMESTAMP

基本 TIMESTAMP 数据类型的语法很简单：

```
TIMESTAMP(n)
```

这里 N 是可选的，用于指定 TIMESTAMP 中秒分量的小数位数，可以取值为 0～9.如果指定 0，TIMESTAMP 在功能上则与 DATE 等价，它们实际上会以同样的方式存储相同的值：

```
ops$tkyte@ORA10G> create table t
  2    ( dt date,
  3    ts timestamp(0)
  4    )
  5    /
Table created.


ops$tkyte@ORA10G> insert into t values ( sysdate, systimestamp );
12.row created.


ops$tkyte@ORA10G> select dump(dt,12.) dump, dump(ts,12.) dump
  2    from t;
DUMP                                DUMP
----------------------------------- -----------------------------------
Typ=12. Len=7: 120,105,6,28,12.,35,41    Typ=180 Len=7: 120,105,6,28,12.,35,41
```

这两个数据类型是不同的（由 TYP=字段可知），但是它们采用了相同的方式存储数据。如果指定要保留几位秒小数，TIMESTAMP 数据类型与 DATE 类型的长度将会不同，例如：

```
ops$tkyte@ORA10G> create table t
  2    ( dt date,
  3    ts timestamp(9)
  4    )
  5    /
```

```
Table created.



ops$tkyte@ORA10G> insert into t values ( sysdate, systimestamp );

12.row created.



ops$tkyte@ORA10G> select dump(dt,12.) dump, dump(ts,12.) dump

  2      from t;

DUMP                                          DUMP

------------------------------------          ------------------------------------

Typ=12. Len=7: 120,105,6,28,12.,46,21     Typ=180 Len=12.: 120,105,6,28,12.,46,21

                                                  ,44,101,192,208
```

现在 TIMESTAMP 占用 12.字节的存储空间，最后额外的 4 个字节包含着小数秒，通过查看所存储的时间就能看出：

```
ops$tkyte@ORA10G> alter session set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';

Session altered.



ops$tkyte@ORA10G> select * from t;

DT                          TS

--------------------        ------------------------------

28-jun-2005 12.:45:20       28-JUN-05 12..45.20.744866000 AM



ops$tkyte@ORA10G> select dump(ts,12.) dump from t;

DUMP

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

```
Typ=180 Len=12.: 78,69,6,12.,b,2e,12.,2c,65,c0,d0



ops$tkyte@ORA10G> select to_number('2c65c0d0','xxxxxxxx') from dual;

TO_NUMBER('2C65C0D0','XXXXXXXX')

--------------------------------

744866000
```

可以看到，存储的小数秒都在最后 4 个字节中。这一次我们使用了 DUMP 函数以 HEX（十六进制）来查看数据，所以能很容易地将这 4 个字节转换为十进制表示。

2. 向 TIMESTAMP 增加或减去时间

DATE 执行日期算术运算所用的技术同样适用于 TIMESTAMP，但是在使用上述技术的很多情况下，TIMESTAMP 会转换为一个 DATE，例如：

```
ops$tkyte@ORA10G> alter session set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';

Session altered.

ops$tkyte@ORA10G> select systimestamp ts, systimestamp+12.dt

  2 from dual;

TS                                        DT

----------------------------------        --------------------

28-JUN-05 12..04.49.833097 AM -04:00 29-jun-2005 12.:04:49
```

注意，这里加 1 实际上将 SYSTIMESTAMP 推进了 1 天，但是小数秒没有了，另外时区信息也没有了。这里使用 INTERVAL 更有意义：

```
ops$tkyte@ORA10G> select systimestamp ts, systimestamp +numtodsinterval(1,'day') dt

  2 from dual;

TS                                        DT

----------------------------------        ---------------------------------------

28-JUN-05 12..08.03.958866 AM -04:00 29-JUN-05 12..08.03.958866000 AM -04:00
```

使用返回一个 INTERVAL 类型的函数能保持 TIMESTAMP 的真实度。使用 TIMESTAMP 时要特别当心，以避免这种隐式转换。

但是还要记住，向 TIMESTAMP 增加月间隔或年间隔时存在相关的警告。如果所得到

的“日期”不是一个合法日期，这个操作就会失败（如果通过 INTERVAL 来增加月，向一月份的最后一天增加 1 个月总会失败，因为这会得到“2 月 31 日”，而 2 月根本没有 31 天 。

3. 得到两个 TIMESTAMP 之差

这正是 DATE 和 TIMESTAMP 类型存在显著差异的地方。尽管将 DATE 相减的结果是一个 NUMBER，但 TIMESTAMP 相减的结果却是一个 INTERVAL：

```
ops$tkyte@ORA10G> select dt2-dt1
  2    from (select to_timestamp('29-feb-2000 01:02:03.122000',
  3                 'dd-mon-yyyy hh24:mi:ss.ff') dt1,
  4                  to_timestamp('12.-mar-2001 12.:22:33.000000',
  5                  'dd-mon-yyyy hh24:mi:ss.ff') dt2
  6           from dual )
  7    /
DT2-DT1
---------------------------------------------------------------------------
+000000380 12.:20:29.878000000
```

两个 TIMESTAMP 值之差是一个 INTERVAL，而且这里显示了二者之间相隔的天数已经小时/分钟/秒数。如果想得到二者之间相差的年数和月数，可以使用以下查询（这个查询类似于先前用于日期的查询）：

```
ops$tkyte@ORA10G> select numtoyminterval
  2           (months_between(dt2,dt1),'month')
  3           years_months,
  4           dt2-add_months(dt1,trunc(months_between(dt2,dt1)))
  5           days_hours
  6    from (select to_timestamp('29-feb-2000 01:02:03.122000',
  7                 'dd-mon-yyyy hh24:mi:ss.ff') dt1,
  8                 to_timestamp('12.-mar-2001 12.:22:33.000000',
```

```
    9                  'dd-mon-yyyy hh24:mi:ss.ff') dt2

   12.         from dual )

   12. /

YEARS_MONTHS     DAYS_HOURS

-------------                 ----------------------------

+000000001-00        +000000015 12.:20:30.000000000
```

需要说明，在这种情况下，由于使用了 ADD_MONTHS，DT1 会隐式转换为一个 DATE 类型，这样就丢失了小数秒。为了保住小数秒，我们必须编写更多的代码。也许可以使用 NUMTOYMINTERVAL 来增加月，这样就能保留 TIMESTAMP；不过，这样一来，我们将遭遇运行时错误：

```
ops$tkyte@ORA10G> select numtoyminterval

   2            (months_between(dt2,dt1),'month')

   3            years_months,

   4            dt2-(dt1 + numtoyminterval( trunc(months_between(dt2,dt1)),'month' ))

   5            days_hours

   6     from (select to_timestamp('29-feb-2000 01:02:03.122000',

   7                  'dd-mon-yyyy hh24:mi:ss.ff') dt1,

   8                  to_timestamp('12.-mar-2001 12.:22:33.000000',

   9                  'dd-mon-yyyy hh24:mi:ss.ff') dt2

   12.         from dual )

   12. /

dt2-(dt1 + numtoyminterval( trunc(months_between(dt2,dt1)),'month' ))

          *

ERROR at line 4:

ORA-01839: date not valid for month specified
```

我个人认为这是不能接受的。不过，问题在于，在你显示有年和月的信息时，

TIMESTAMP 的真实性已经被破坏了。一年有多长并不固定（可能是 365 天，也可能是 366 天），同样，月的长度也不固定。如果你在显示有年和月的信息，毫秒级的损失则无关大碍；显示这种信息时细到秒级就完全足够了。

4. TIMESTAMP WITH TIME ZONE 类型

TIMESTAMP WITH TIME ZONE 类型继承了 TIMESTAMP 类型的所有特点，并增加了时区支持。TIMESTAMP WITH TIME ZONE 类型占 12.字节的存储空间，在此有额外的 2 个字节用于保留时区信息。它在结构上与 TIMESTAMP 的差别只是增加了这 2 个字节：

```
ops$tkyte@ORA10G> create table t
  2  (
  3      ts timestamp,
  4      ts_tz timestamp with time zone
  5  )
  6  /
Table created.

ops$tkyte@ORA10G> insert into t ( ts, ts_tz )
  2  values ( systimestamp, systimestamp );
12.row created.

ops$tkyte@ORA10G> select * from t;
TS                                TS_TZ
--------------------------        ---------------------------------
28-JUN-05 01.45.08.087627 PM 28-JUN-05 01.45.08.087627 PM -04:00

ops$tkyte@ORA10G> select dump(ts), dump(ts_tz) from t;
DUMP(TS)
```

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

DUMP(TS_TZ)

--------------------------------------------------------------------------------

Typ=180 Len=12.: 120,105,6,28,12.,46,9,5,57,20,248

Typ=181 Len=12.: 120,105,6,28,12.,46,9,5,57,20,248,12.,60
```

可以看到，获取数据时，默认的 TIMESTAMP WITH TIME ZONE 格式包括有时区信息（执行这个操作时，我所在的时区是 East Coast US，当时正是白天）。

存储数据时，TIMESTAMP WITH TIME ZONE 会在数据中存储指定的时区。时区成为数据本身的一部分。注意 TIMESTAMP WITH TIME ZONE 字段如何存储小时、分钟和秒（…12.,46,9…），这里采用了加 1 表示法，因此…12.,46,9…就表示 12.:45:08，而…12.,46,9…字段只存储了…12.,46,9…，这表示 12.:45:09，也就是我们插入到串中的那个时间。TIMESTAMP WITH TIME ZONE 为它增加了 4 个小时，从而存储为 GWT（也称为 UTC）时间。获取时，会使用尾部的 2 个字节适当地调整 TIMESTAMP 值。

我 并不打算在这里全面介绍时区的所有细节；这个主题在其他资料中已经做了很好的说明。我只是要指出，与此前相比，时区支持对于今天的应用更显重要。十年前， 应用不像现在这么具有全球性。在因特网普及之前，应用更多地都是分布和分散的，隐含地时区都基于服务器所在的位置。如今，由于大型的集中式系统可能由世界 各地的人使用，所以记录和使用时区非常重要。

将时区支持内建到数据类型之前，必须把 DATE 存储在一个列中，而把时区信息存储在另外一列中，然后要由应用负责使用函数将 DATE 从一个时区转换到另一个时区。现在则不然，这个任务已经交给了数据库，数据库能存储多个时区的数据：

```
ops$tkyte@ORA10G> create table t
  2     ( ts1 timestamp with time zone,
  3     ts2 timestamp with time zone
  4     )
  5     /

Table created.



ops$tkyte@ORA10G> insert into t (ts1, ts2)
  2     values ( timestamp'2005-06-05 12.:02:32.212 US/Eastern',
```

```
  3     timestamp'2005-06-05 12.:02:32.212 US/Pacific' );

12.row created.
```

并对这些数据执行正确的 TIMESTAMP 运算：

```
ops$tkyte@ORA10G> select ts1-ts2 from t;

TS1-TS2

---------------------------------------------------------------------------

-000000000 03:00:00.000000
```

因为这两个时区之间有 3 个小时的时差，尽管它们显示的是“同样的时间”——12.:02:32:212，但是从报告的时间间隔来看确实存在 3 小时的时差。在 TIMESTAMP WITH TIME ZONE 类型上执行 TIMESTAMP 运算时，Oracle 会自动地把两个类型首先转换为 UTC 时间，然后执行运算。

5. TIMESTAMP WITH LOCAL TIME ZONE 类型

这种类型与 TIMESTAMP 类型的工作是类似的。这是一个 7 字节或 12.字节的字段（取决于 TIMESTAMP 的精度），但是会进行规范化，在其中存入数据库的时区。要了解这一点，我们将再次使用 DUMP 命令。首先，创建一个包括 3 列的表，这 3 列分别是一个 DATE 列、一个 TIMESTAMP WITH TIME ZONE 列和一个 TIMESTAMP WITH LOCAL TIME ZONE 列，然后向这 3 列插入相同的值：

```
ops$tkyte@ORA10G> create table t
  2     ( dt date,
  3     ts1 timestamp with time zone,
  4     ts2 timestamp with local time zone
  5     )
  6     /

Table created.



ops$tkyte@ORA10G> insert into t (dt, ts1, ts2)
  2     values ( timestamp'2005-06-05 12.:02:32.212 US/Pacific',
  3              timestamp'2005-06-05 12.:02:32.212 US/Pacific',
```

```
  4          timestamp'2005-06-05 12.:02:32.212 US/Pacific' );

12.row created.



ops$tkyte@ORA10G> select dbtimezone from dual;

DBTIMEZONE

----------

US/Eastern
```

现在，将这些值转储如下：

```
ops$tkyte@ORA10G> select dump(dt), dump(ts1), dump(ts2) from t;

DUMP(DT)

-----------------------------------

DUMP(TS1)

-----------------------------------

DUMP(TS2)

-----------------------------------

Typ=12. Len=7: 120,105,6,5,12.,3,33

Typ=181 Len=12.: 120,105,6,6,12.3,33,12.,162,221,0,137,156

Typ=231 Len=12.: 120,105,6,5,21,3,33,12.,162,221,0
```

可以看到，在这个例子中，会存储 3 种完全不同的日期/时间表示：

- DT：这一列存储了日期/时间 5-JUN-2005 12.:02:32。时区和小数秒没有了，因为我们使用的是 DATE 类型。这里根本不会执行时区转换。我们会原样存储插入的那个日期/时间，但是会丢掉时区。
- TS1：这一列保留了 TIME ZONE 信息，并规范化为该 TIME ZONE 相应的 UTC 时间。所插入的 TIMESTAMP 值处于 US/Pacific 时区，在写这本书时这个时间与 UTC 时间相差 7 个小时。因此，存储的日期/时间是 6-JUN-2005 00:02:32.212。它把输入的时间推进了 7 个小时，使之成为 UTC 时间，并把时区 US/Pacific 保存为最后 2 个字节，这样以后就能适当地解释这个数据。
- TS2：这里认为这个列的时区就是数据库时区，即 US/Eastern。现在，12.:02:32 US/Pacific is 20:02:32 US/Eastern，所以存储为以下字节（...21,3,33...），这里采用了

加 1 表示法：取得实际时间时要记住减 1。

由于 TS1 列在最后 2 字节保留了原来的时区，获取时我们会看到以下结果：

```
ops$tkyte@ORA10G> select ts1, ts2 from t;

TS1

---------------------------------------

TS2

---------------------------------------

05-JUN-05 05.02.32.212000 PM US/PACIFIC

05-JUN-05 08.02.32.212000 PM
```

数据库应该能显示这个信息，但是有 LOCAL TIME ZONE（数据库时区）的 TS2 列只显示了数据库时区的时间，并认为这就是这一列的时区（实际上，这个数据库中有 LOCAL TIME ZONE 的所有列的时区都是数据库时区）。我的数据库处于 US/Eastern 时区，所以插入的 12.:02:32 US/Pacific 现在显示为 8:00pm East Coast 时间。

如果你不需要记住源时区，只需要这样一种数据类型，要求能对日期/时间类型提供一致的全球性处理，那么 TIMESTAMP WITH LOCAL TIME ZONE 对大多数应用来说已经能提供足够的支持。另外，TIMESTAMP(0) WITH LOCAL TIME ZONE 是与 DATE 类型等价但提供了时区支持的一种类型；它占用 7 字节存储空间，允许存储按 UTC 形式“规范化”的日期。

关于 TIMESTAMP WITH LOCAL TIME ZONE 类型有一个警告，一旦你创建有这个列的表，会发现你的数据库的时区“被冻住了”，你将不能修改数据库的时区。

```
ops$tkyte@ORA10G> alter database set time_zone = 'PST';

alter database set time_zone = 'PST'

*

ERROR at line 1:

ORA-30079: cannot alter database timezone when database has

TIMESTAMP WITH LOCAL TIME ZONE columns

ops$tkyte@ORA10G> !oerr ora 30079

30079, 00000, "cannot alter database timezone when database has
```

```
TIMESTAMP WITH LOCAL TIME ZONE columns"

// *Cause: An attempt was made to alter database timezone with

// TIMESTAMP WITH LOCAL TIME ZONE column in the database.

// *Action: Either do not alter database timezone or first drop all the

// TIMESTAMP WITH LOCAL TIME ZONE columns.
```

其原因是：倘若你能修改数据库的时区，就必须将每一个有 TIMESTAMP WITH LOCAL TIME ZONE 的表重写，否则在新时区下，它们当前的值将是不正确的！

12.6.4 INTERVAL 类型

上一节我们简要地提到了 INTERVAL 类型。这是表示一段时间或一个时间间隔的一种方法。这一节将讨论两个 INTERVAL 类型：其中一个是 YEAR TO MONTH 类型，它能存储按年和月指定的一个时段；另一个类型是 DATE TO SECOND 类型，它能存储按天、小时、分钟和秒（包括小数秒）指定的时段。

在具体介绍这两个 INTERVAL 类型之前，我想先谈谈 EXTRACT 内置函数，处理这种类型时这个函数可能非常有用。EXTRACT 内置函数可以处理 TIMESTAMP 和 INTERVAL，并从中返回各部分信息，如从 TIMESTAMP 返回时区，从 INTERVAL 返回小时/天/分钟。还是用前面的例子（其中得到了 380 天、12.小时、29.878 秒的 INTERVAL）：

```
ops$tkyte@ORA10G> select dt2-dt1
  2    from (select to_timestamp('29-feb-2000 01:02:03.122000',
  3                'dd-mon-yyyy hh24:mi:ss.ff') dt1,
  4                to_timestamp('12.-mar-2001 12.:22:33.000000',
  5                'dd-mon-yyyy hh24:mi:ss.ff') dt2
  6           from dual )
  7    /
DT2-DT1
---------------------------------------------------------------------------
+000000380 12.:20:29.878000000
```

可以使用 EXTRACT 来查看，它能很轻松地取出其中的各部分信息：

```
ops$tkyte@ORA10G> select extract( day from dt2-dt1 ) day,
```

```
  2           extract( hour from dt2-dt1 ) hour,
  3           extract( minute from dt2-dt1 ) minute,
  4           extract( second from dt2-dt1 ) second
  5     from (select to_timestamp('29-feb-2000 01:02:03.122000',
  6                 'dd-mon-yyyy hh24:mi:ss.ff') dt1,
  7                 to_timestamp('12.-mar-2001 12.:22:33.000000',
  8                 'dd-mon-yyyy hh24:mi:ss.ff') dt2
  9           from dual )
 12. /
       DAY         HOUR      MINUTE         SECOND
 ----------  ----------  ----------  ----------
       380          12.          20          29.878
```

另外，我们已经了解了创建 YEAR TO MONTH 和 DAY TO SECOND 间隔时所用的 NUMTOYMINTERVAL 和 NUMTODSINTERVAL。我发现这些函数是创建 INTERVAL 类型实例最容易的方法，远远胜于串转换函数。我不喜欢把一大堆表示天、小时、分钟和秒的数连接在一起来表示某个间隔，而是会增加 4 个 NUMTODSINTERVAL 调用来完成同样的工作。

INTERVAL 类型不只是可以用于存储时段，还可以以某种方式存储“时间”。例如，如果你希望存储一个特定的日期时间，可以使用 DATE 或 TIMESTAMP 类型。但是如果你只想存储上午 8:00 这个时间呢？INTERVAL 类型就很方便（尤其是 INTERVAL DAY TO SECOND 类型）。

1. INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH 的语法很简单：

```
INTERVAL YEAR(n) TO MONTH
```

在此 N 是一个可选的位数（用以支持年数），可取值为 0～9，默认为 2（表示年数可为 0～99）。这就允许你存储任意大小的年数（最多可达 9 位）和月数。我更喜欢用 NUMTOYMINTERVAL 函数来创建这种类型的 INTERVAL 实例。例如，要创建一个 5 年 2 个月的时间间隔，可以使用以下命令：

```
ops$tkyte@ORA10G> select numtoyminterval(5,'year')+numtoyminterval(2,'month')
  2     from dual;
```

```
NUMTOYMINTERVAL(5,'YEAR')+NUMTOYMINTERVAL(2,'MONTH')

-------------------------------------------------------------------------

+000000005-02
```

或者，利用 1 年有 12.个月这个事实，可以使用一个调用，并使用以下命令：

```
ops$tkyte@ORA10G> select numtoyminterval(5*12.+2,'month')

  2      from dual;



NUMTOYMINTERVAL(5*12.+2,'MONTH')

-------------------------------------------------------------------------

+000000005-02
```

这两种方法都能很好地工作。还可以用另一个函数 TO_YMINTERVAL 将一个串转换为一个年/月 INTERVAL 类型：

```
ops$tkyte@ORA10G> select to_yminterval( '5-2' ) from dual;



TO_YMINTERVAL('5-2')

-------------------------------------------------------------------------

+000000005-02
```

但是，由于我的应用中大多数情况下都是把年和月放在两个 NUMBER 字段中，所以我发现 NUMTOYMINTERVAL 函数更有用，而不是先从数字构建一个格式化的串。

最后，还可以直接在 SQL 中使用 INTERVAL 类型，而不用这些函数：

```
ops$tkyte@ORA10G> select interval '5-2' year to month from dual;

INTERVAL'5-2'YEARTOMONTH

-------------------------------------------------------------------------

+05-02
```

2. INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND 类型的语法很简单：

```
INTERVAL DAY(n) TO SECOND(m)
```

在此 N 是一个可选的位数，支持天数分量，取值为 0～9，默认为 2。M 是秒字段小时部分中保留的位数，其中为 0～9，默认为 6.同样，我更喜欢用 NUMTODSINTERVAL 函数来创建这种类型的 INTERVAL 实例：

```
ops$tkyte@ORA10G> select numtodsinterval( 12., 'day' )+

  2           numtodsinterval( 2, 'hour' )+

  3           numtodsinterval( 3, 'minute' )+

  4           numtodsinterval( 2.3312, 'second' )

  5    from dual;



NUMTODSINTERVAL(12.,'DAY')+NUMTODSINTERVAL(2,'HOUR')+NUMTODSINTE
RVAL(3,'MINU

---------------------------------------------------------------------------

+000000010 02:03:02.331200000
```

或者只是：

```
ops$tkyte@ORA10G> select numtodsinterval( 12.*86400+2*3600+3*60+2.3312, 'second' )

  2    from dual;



NUMTODSINTERVAL(12.*86400+2*3600+3*60+2.3312,'SECOND')

---------------------------------------------------------------------------

+000000010 02:03:02.331200000
```

这里利用了一天有 89,400 秒，一小时有 3,600 秒等事实。或者，像前面一样，可以使用 TO_DSINTERVAL 函数将一个串转换为一个 DAY TO SECOND 间隔：

```
ops$tkyte@ORA10G> select to_dsinterval( '12. 02:03:02.3312' )

  2    from dual;

TO_DSINTERVAL('1002:03:02.3312')

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

```
+000000010 02:03:02.331200000
```

或者只是在 SQL 本身中使用 INTERVAL 变量：

```
ops$tkyte@ORA10G> select interval '12. 02:03:02.3312' day to second

  2    from dual;

INTERVAL'1002:03:02.3312'DAYTOSECOND

---------------------------------------------------------------------------

+12. 02:03:02.331200
```

12.7 LOB 类型

根据我的经验，LOB 或大对象（large object）是产生许多混乱的根源。这些数据类型很容易被误解，这包括它们是如何实现的，以及如何最好地加以使用。这一节将概要介绍 LOB 如何物理地存储，并指出使用 LOB 类型是必须考虑哪些问题。这些类型有许多可选的设置，要针对你的应用做出正确的选择，这一点至关重要。

Oracle 中支持 4 种类型的 LOB：

- CLOB：字符 LOB。这种类型用于存储大量的文本信息，如 XML 或者只是纯文本。这个数据类型需要进行字符集转换，也就是说，在获取时，这个字段中的字符会从数据库的字符集转换为客户的字符集，而在修改时会从客户的字符集转换为数据库的字符集。
- NCLOB：这是另一种类型的字符 LOB。存储在这一列中的数据所采用的字符集是数据库的国家字符集，而不是数据库的默认字符集。
- BLOB：二进制 LOB。这种类型用于存储大量的二进制信息，如字处理文档，图像和你能想像到的任何其他数据。它不会执行字符集转换。应用向 BLOB 中写入什么位和字节，BLOB 就会返回什么为和字节。
- BFILE：二进制文件 LOB。这与其说是一个数据库存储实体，不如说是一个指针。带 BFILE 列的数据库中存储的只是操作系统中某个文件的一个指针。这个文件在数据库之外维护，根本不是数据库的一部分。BFILE 提供了文件内容的只读访问。

讨论 LOB 时，我会分两节讨论上述各个类型：其中一节讨论存储在数据库中的 LOB，这也称为内部 LOB，包括 CLOB、BLOB 和 NCLOB；另一节讨论存储在数据库之外的 LOB，或 BFILE 类型。我不打算分别讨论 CLOB、BLOB 或 NCLOB，因为为了从存储来看，还是从选项来看，它们都是一样的。只不过 CLOB 和 NCLOB 支持文本信息，而 BLOB 支持二进制信息。不过不论基类型是什么，所指定的选项（CHUNKSIZE、PCTVERSION 等）和要考虑的问题都是一样的。但由于 BFILE 与它们有显著不同，所以我们将单独地讨论这种类型。

12.7.1 内部 LOB

从表面看，LOB 的语法相当简单，但这只是一种假象。你可以创建有 CLOB、BLOB

或 NCLOB 数据类型列的表，就这么简单。似乎使用这些数据类型就像使用 NUMBER、DATE 或 VARCHAR2 类型一样容易：

```
ops$tkyte@ORA10G> create table t
  2     ( id int primary key,
  3     txt clob
  4     )
  5     /
Table created.
```

还能怎么样呢？但这个小例子只是显示出冰山一角，关于 LOB 能指定的选项很多，这里只是其中极少的一部分。通过使用 DBMS_METADATA 方能得到全貌：

```
ops$tkyte@ORA10G> select dbms_metadata.get_ddl( 'TABLE', 'T' )
  2     from dual;


DBMS_METADATA.GET_DDL('TABLE','T')
--------------------------------------------------------------------------------
CREATE TABLE "OPS$TKYTE"."T"
( "ID" NUMBER(*,0),
"TXT" CLOB,
PRIMARY KEY ("ID")
USING INDEX PCTFREE 12. INITRANS 2 MAXTRANS 255
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645
PCTINCREASE 0 FREELISTS 12.FREELIST GROUPS 12.BUFFER_POOL DEFAULT)
TABLESPACE "USERS" ENABLE
) PCTFREE 12. PCTUSED 40 INITRANS 12.MAXTRANS 255 NOCOMPRESS
LOGGING
```

```
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645

PCTINCREASE 0 FREELISTS 12.FREELIST GROUPS 12.BUFFER_POOL DEFAULT)

TABLESPACE "USERS"

LOB ("TXT") STORE AS (

TABLESPACE "USERS" ENABLE STORAGE IN ROW CHUNK 8192
PCTVERSION 12.

NOCACHE

STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 12.MAXEXTENTS
2147483645

PCTINCREASE 0 FREELISTS 12.FREELIST GROUPS 12.BUFFER_POOL
DEFAULT))
```

LOB 显然有以下属性：

- 一个表空间（这个例子中即为 USERS）
- ENABLE STORAGE IN ROW 作为一个默认属性
- CHUNK 8192
- PCTVERSION 12.
- NOCACHE
- 一个完整的 STORAGE 子句

由此说明，在底层 LOB 并不那么简单，而事实上确实如此。LOB 列总是会带来一种多段对象（multisegment object，这是我对它的叫法），也就是说，这个表会使用多个物理段。如果我们在一个空模式中创建这个表，就会发现以下结果：

```
ops$tkyte@ORA10G> select segment_name, segment_type

  2  from user_segments;

SEGMENT_NAME                    SEGMENT_TYPE

------------------------------  ------------------

SYS_C0011927                    INDEX

SYS_IL0000071432C00002$$        LOBINDEX
```

```
SYS_LOB0000071432C00002$$      LOBSEGMENT

T                                                TABLE
```

这里创建了一个索引来支持主键约束，这很正常，但是另外两个段呢？即 lobindex 和 lobsegment，它们做什么用？创建这些段是为了支持我们的 LOB 列。我们的实际 LOB 数据就存储在 lobsegment 中（确实，LOB 数据也有可能存储在表 T 中，不过稍后讨论 ENABLE STORAGE IN ROW 子句时还会更详细地说明这个内容）。lobindex 用于执行 LOB 的导航，来找出其中的某些部分。创建一个 LOB 列时，一般来说，存储在行中的这是一个指针（pointer），或 LOB 定位器（LOB locator）。我们的应用所获取的就是这个 LOB 定位器。当请求得到 LOB 的“12.000～2,000 字节”时，将对 lobindex 使用 LOB 定位器来找出这些字节存储在哪里，然后再访问 lobsegment。可以用 lobindex 很容易地找到 LOB 的各个部分。由此说来，可以把 LOB 想成是一种主/明细关系。LOB 按“块”（chunk）或（piece）来存储，每个片段都可以访问。例如，如果我们使用表来实现一个 LOB，可以如下做到这一点：

```
Create table parent

        ( id int primary key,

        other-data...

);



Create table lob

        ( id references parent on delete cascade,

        chunk_number int,

        data <datatype>(n),

        primary key (id,chunk_number)

);
```

从概念上讲，LOB 的存储与之非常相似，创建这两个表时，在 LOB 表的 ID.CHUNK_NUMBER 上要有一个主键（这对应于 Oracle 创建的 lobindex），而且要有一个 LOB 表来存储数据块（对应于 lobsegment）。LOB 列为我们透明地实现了这种主/明细结构。图 12.-3 可以更清楚地展示这个思想。

图 12.-3　表-lobindex-lobsegment 的对于关系

表中的 LOB 实际上只是指向 lobindex，lobindex 再指向 LOB 本身的各个部分。为了得到 LOB 中的 N~M 字节，要对表中的指针（LOB 定位器）解除引用，遍历 lobindex 结构来找到所需的数据库（chunk），然后按顺序访问。这使得随机访问 LOB 的任何部分都能同样迅速，你可以用同样快的速度得到 LOB 的最前面、中间或最后面的部分，因为无需再从头开始遍历 LOB。

既然已经从概念上了解了 LOB 如何存储，下面我将逐个介绍前面所列的各个可选设置，并解释它们的用途以及有什么具体含义。

1.　LOB 表空间

从 DBMS_METADATA 返回的 CREATE TABLE 语句包括以下内容：

```
LOB ("TXT") STORE AS ( TABLESPACE "USERS" ...
```

这里指定的 TABLESPACE 是将存储 lobsegment 和 lobindex 表空间，这可能与表本身所在的表空间不同。也就是说，保存 LOB 数据的表空间可能不同于保存实际表数据的表空间。

为什么考虑为 LOB 数据使用另外一个表空间（而不用表数据所在的表空间）呢？注意原因与管理和性能有关。从管理的角度看，LOB 数据类型表示一种规模很大的信息。如果表有数百万行，而每行有一个很大的 LOB，那么 LOB 就会极为庞大。为 LOB 数据单独使用一个表空间有利于备份和恢复以及空间管理，单从这一点考虑，将表与 LOB 数据分离就很有意义。例如，你可能希望 LOB 数据使用另外一个统一的区段大小，而不是普通表数据所用的区段大小。

另一个原因则出于 I/O 性能的考虑。默认情况下，LOB 不在缓冲区缓存中进行缓存（有关内容将在后面再做说明）。因此，默认情况下，对于每个 LOB 访问，不论是读还是写，都会带来一个物理 I/O（从磁盘直接读，或者向磁盘直接写）。

注意　LOB 可能是内联的（inline），或者存储在表中。在这种情况下，LOB 数据会被缓存，但是这只适用于小于 4,000 字节的 LOB。我们将在“IN ROW 子句”一节中进一步讨论这种情况。

由于每个访问都是一个物理 I/O，所以如果你很清楚在实际中（当用户访问时）有些对象会比大多数其他对象经历更多的物理 I/O，那么将这些对象分离到它们自己的磁盘上就很

有意义。

需要说明，lobindex 和 lobsegment 总是会在同一个表空间中。不能将 lobindex 和 lobsegment 放在不同的表空间中。在 Oralce 的更早版本中，允许为 lobindex 和 lobsegment 分别放在单独的表空间中，但是从 8i Release 3 以后，就不再允许为 lobindex 和 logsegment 指定不同的表空间。实际上，lobindex 的所有存储特征都是从 lobsegment 继承的，稍后就会看到。

2. IN ROW 子句

前面的 DBMS_METADATA 返回的 CREATE TABLE 语句还包括以下内容：

```
LOB ("TXT") STORE AS (... ENABLE STORAGE IN ROW ...
```

这控制了 LOB 数据是否总与表分开存储（存储在 lobsegment 中），或是有时可以与表一同存储，而不用单独放在 lobsegment 中。如果设置了 ENABLE STORAGE IN ROW，而不是 DISABLE STORAGE IN ROW，小 LOB（最多 4,000 字节）就会像 VARCHAR2 一样存储在表本身中。只有当 LOB 超过了 4,000 字节时，才会“移出”到 lobsegment 中。

默认行为是启用行内存储（ENABLE STORAGE IN ROW），而且一般来讲，如果你知道 LOB 总是能在表本身中放下，就应该采用这种默认行为。例如，你的应用可能有一个某种类型的 DESCRIPTION 字段。这个 DESCRIPTION 可以存储 0～32KB 的数据（或者可能更多，但大多数情况下都少于或等于 32KB）。已知很多描述都很简短，只有几百个字符。如果把它们单独存储，并在每次获取时都通过索引来访问，就会存在很大的开销，你完全可以将它们内联存储，即放在表本身中，这就能避免单独存储的开销。不仅如此，如果 LOB 还能避免获取 LOB 时所需的物理 I/O。

下面通过一个非常简单的例子来看看这种设置的作用。我们将创建包括有两个 LOB 的表，其中一个 LOB 可以在行内存储数据，而另一个 LOB 禁用了行内存储：

```
ops$tkyte@ORA10G> create table t
  2    ( id int primary key,
  3    in_row clob,
  4    out_row clob
  5    )
  6    lob (in_row) store as ( enable storage in row )
  7    lob (out_row) store as ( disable storage in row )
  8 /

Table created.
```

在这个表中，我们将插入一些串数据，所有这些串的长度都不超过 4,000 字节：

```
ops$tkyte@ORA10G> insert into t
  2          select rownum,
  3                  owner || ' ' || object_name || ' ' || object_type || ' ' || status,
  4                   owner || ' ' || object_name || ' ' || object_type || ' ' || status
  5          from all_objects
  6    /
48592 rows created.


ops$tkyte@ORA10G> commit;
Commit complete.
```

现在，如果我们想读取每一行，在此使用了 DBMS_MONITOR 包，并启用了 SQL_TRACE，执行这个工作时，可以看到这两个表获取数据时的性能：

```
ops$tkyte@ORA10G> declare
  2          l_cnt number;
  3          l_data varchar2(32765);
  4    begin
  5          select count(*)
  6                  into l_cnt
  7          from t;
  8
  9          dbms_monitor.session_trace_enable;
 12.         for i in 1 .. l_cnt
 12.         loop
 12.                 select in_row into l_data from t where id = i;
```

```
12.                 select out_row into l_data from t where id = i;
12.         end loop;
12. end;
12. /
PL/SQL procedure successfully completed.
```

查看这个小仿真的 TKPROF 报告时，结果一目了然：

```
SELECT IN_ROW FROM T WHERE ID = :B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  48592      2.99       2.78          0          0          0           0
Fetch    48592     12.84      12.80          0     145776          0       48592
------- ------  -------- ----------  ----------  ----------  ----------  ----------
total    97185      4.83       4.59          0     145776          0       48592

Rows     Row Source Operation
-------  ---------------------------------------------------
48592  TABLE ACCESS BY INDEX ROWID T (cr=145776 pr=0 pw=0 time=1770453 us)
48592   INDEX UNIQUE SCAN SYS_C0011949 (cr=97184 pr=0 pw=0 time=960814 us)

********************************************************************************

SELECT OUT_ROW FROM T WHERE ID = :B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ----------  ----------  ----------  ----------  ----------
```

```
Parse        1      0.00       0.00          0          0          0          0

Execute 48592       2.21       2.12.         0          0          0          0

Fetch    48592      7.33       8.49      48592     291554          0      48592

-------  ------  -------- ----------  ----------  ----------  ----------  ----------

total    97185      9.54      12..62      48592     291554          0      48592

Rows     Row Source Operation

-------  ---------------------------------------------------

48592    TABLE ACCESS BY INDEX ROWID T (cr=145776 pr=0 pw=0 time=1421463
us)

48592    INDEX UNIQUE SCAN SYS_C0011949 (cr=97184 pr=0 pw=0 time=737992
us)

Elapsed times include waiting on following events:

Event waited on                          Times      Max. Wait    Total Waited

----------------------------------------  Waited     ----------   ------------

direct path read                         48592          0.00          0.25
```

获取 IN_ROW 列显著地快得多，而且所占用的资源也远远少于 OUT_ROW 列。可以看到，它使用了 145,776 次逻辑 I/O（查询模式获取），而 OUT_ROW 列使用的逻辑 I/O 次数是它的两倍。初看上去，我们不太清楚这些额外的逻辑 I/O 是从哪里来的，不过，如果你还记得 LOB 是如何存储的就会明白，这是对 lobindex 段的 I/O（为了找到 LOB 的各个部分）。这些额外的逻辑 I/O 都针对这个 lobindex.

另外，可以看到，对于 OUT_ROW 列，获取 48,592 行会带来 48,592 次物理 I/O，而这会导致同样数目的“直接路径读”I/O 等待。这些都是对非缓存 LOB 数据的读取。在这种情况下，通过启用 LOB 数据的缓存，可以缓解这个问题，但是这样一来，我们又必须确保为此要有足够多的额外的缓冲区缓存。另外，如果确实有非常大的 LOB，我们可能并不希望缓存这些数据。

这种行内/行外存储设置不仅会影响读，还会影响修改。如果我们要用小串更新前 100 行，并用小串插入 100 个新行，再使用同样的技术查看性能，会观察到：

```
ops$tkyte@ORA10G> create sequence s start with 100000;

Sequence created.
```

```
ops$tkyte@ORA10G> declare
  2          l_cnt number;
  3          l_data varchar2(32765);
  4    begin
  5          dbms_monitor.session_trace_enable;
  6          for i in 1 .. 100
  7          loop
  8                  update t set in_row =
                     to_char(sysdate,'dd-mon-yyyy hh24:mi:ss') where id = i;
  9                  update t set out_row =
                     to_char(sysdate,'dd-mon-yyyy hh24:mi:ss') where id = i;
 12.                 insert into t (id, in_row) values ( s.nextval, 'Hello World' );
 12.                 insert into t (id,out_row) values ( s.nextval, 'Hello World' );
 12.         end loop;
 12. end;
 12. /
PL/SQL procedure successfully completed.
```

在得到的 TKPROF 报告中可以观察到类似的结果：

```
UPDATE T SET IN_ROW = TO_CHAR(SYSDATE,'dd-mon-yyyy hh24:mi:ss')
WHERE ID = :B1

call          count        cpu elapsed          disk         query       current        rows
-------         ------    -------- ----------      ----------      ----------      ---------- ----------
```

Parse 1 0.00 0.00 0 0 0 0

Execute 100 0.05 0.02 0 200 202 100

Fetch 0 0.00 0.00 0 0 0 0

------- ------ -------- ---------- ---------- ---------- ---------- ----------

total 101 0.05 0.02 0 200 202 100

Rows Row Source Operation

------- ---

100 UPDATE (cr=200 pr=0 pw=0 time=15338 us)

100 INDEX UNIQUE SCAN SYS_C0011949 (cr=200 pr=0 pw=0 time=2437 us)

****

UPDATE T SET OUT_ROW = TO_CHAR(SYSDATE,'dd-mon-yyyy hh24:mi:ss')

WHERE ID = :B1

call count cpu elapsed disk query current rows

------- ------ -------- ---------- ---------- ---------- ---------- ----------

Parse 1 0.00 0.00 0 0 0 0

Execute 100 0.07 0.12. 0 1100 2421 100

Fetch 0 0.00 0.00 0 0 0 0

------- ------ -------- ---------- ---------- ---------- ---------- ----------

total 101 0.07 0.12. 0 1100 2421 100

Rows Row Source Operation

------- ---

```
100                  UPDATE (cr=1100 pr=0 pw=100 time=134959 us)

100                  INDEX UNIQUE SCAN SYS_C0011949 (cr=200 pr=0 pw=0 time=2180
us)

Elapsed times include waiting on following events:

Event waited on                          Times       Max. Wait      Total Waited

----------------------------------------  Waited          ----------          ------------

direct path write                          200              0.00                0.00
```

可以看到，行外 LOB 的更新占用了更多的资源。它要花一定的时间完成直接路径写（物理 I/O），并执行更多的当前模式获取以及查询模式获取。这些都源于一点，即除了维护表本身外，还必须维护 lobindex 和 lobsegment。INSERT 操作也显示出了同样的差异：

```
INSERT INTO T (ID, IN_ROW) VALUES ( S.NEXTVAL, 'Hello World' )

call          count        cpu elapsed          disk         query       current       rows

-------       ------     -------- ----------      ----------      ----------      ---------- ----------

Parse             1       0.00       0.00              0              0              0           0

Execute         100       0.03       0.02              0              2            316         100

Fetch             0       0.00       0.00              0              0              0           0

-------       ------     -------- ----------      ----------      ----------      ---------- ----------

total           101       0.03       0.02              0              2            316         100

********************************************************************************
******

INSERT INTO T (ID,OUT_ROW) VALUES ( S.NEXTVAL, 'Hello World' )

call          count        cpu elapsed          disk         query       current       rows

-------       ------     -------- ----------      ----------      ----------      ---------- ----------

Parse             1       0.00       0.00              0              0              0           0

Execute         100       0.08       0.12.             0            605           1839         100

Fetch             0       0.00       0.00              0              0              0           0
```

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

total            101       0.08     0.12.                0          605             1839       100

Elapsed times include waiting on following events:

Event waited on                          Times      Max. Wait     Total Waited

----------------------------------------  Waited     ----------    ------------

direct path write                          200          0.00           0.00
```

注意读和写使用的 I/O 都有所增加。总之，由此显示出，如果使用一个 CLOB，而且很多串都能在“行内”放下（也就是说，小于 4,000 字节），那么使用默认的 ENABLE STORAGE IN ROW 设置就是一个不错的想法。

3. CHUNK 子句

前面的 DBMS_METADATA 返回的 CREATE TABLE 语句包括以下内容：

```
LOB ("TXT") STORE AS ( ... CHUNK 8192 ... )
```

LOB 存储在块（chunk）中；指向 LOB 数据的索引会指向各个数据块。块（chunk）是逻辑上连续的一组数据库块（block），这也是 LOB 的最小分配单元，而通常数据库的最小分配单元是数据库块。CHUNK 大小必须是 Oracle 块大小的整数倍，只有这样才是合法值。

从两个角度看，选择 CHUNK 大小时必须当心。首先，每个 LOB 实例（每个行外存储的 LOB 值）会占用至少一个 CHUNK。一个 CHUNK 有一个 LOB 值使用。如果一个表有 100 行，而每行有一个包含 7KB 数据的 LOB，你就会分配 100 个 CHUNK，如果将 CHUNK 大小设置为 32KB，就会分配 100 个 32KB 的 CHUNK。如果将 CHUNK 大小设置为 8KB，则（可能）分配 100 个 8KB 的 CHUNK。关键是，一个 CHUNK 只能有一个 LOB 使用（两个 LOB 不会使用同一个 CHUNK）。如果选择了一个 CHUNK 大小，但不符合你期望的 LOB 大小，最后就会浪费大量的空间。例如，如果表中的 LOB 平均有 7KB，而你使用的 CHUNK 大小为 32KB，对于每个 LOB 实例你都会“浪费”大约 25KB 的空间，另一方面，倘若使用 8KB 的 CHUNK，就能使浪费减至最少。

还需要注意要让每个 LOB 实例相应的 CHUNK 数减至最少。前面已经看到了，有一个 lobindex 用于指向各个块，块越多，索引就越大。如果有一个 4MB 的 LOB，并使用 8KB 的 CHUNK，你就至少需要 512 个 CHUNK 来存储这个消息。这也说明，至少需要 512 个 lobindex 条目指向这些 CHUNK。听上去好像没什么，但是你要记住，对于每个 LOB 个数的 512 倍。另外，这还会影响获取性能，因为与读取更少但更大的 CHUNK 相比，现在要花更长的数据来读取和管理许多小 CHUNK。我们最终的目标是：使用一个能使“浪费”最少，同时又能高效存储数据的 CHUNK 大小。

4. PCTVERSION 子句

前面的 DBMS_METADATA 返回的 CREATE TABLE 语句包括以下内容：

```
LOB ("TXT") STORE AS ( ... PCTVERSION 12. ... )
```

这用于控制 LOB 的读一致性。在前面的几章中，我们已经讨论了读一致性、多版本和 undo 在其中所起的作用。但 LOB 实现读一致性的方式有所不同。lobsegment 并不使用 undo

来记录其修改；而是直接在 lobsegment 本身中维护信息的版本。lobindex 会像其他段一样生成 undo，但是 lobsegment 不会。相反，修改一个 LOB 时，Oracle 会分配一个新的 CHUNK，并且仍保留原来的 CHUNK。如果回滚了事务，对 LOB 索引所做的修改会回滚，索引将再次指向原来的 CHUNK。因此，undo 维护会在 LOB 段本身中执行。修改数据时，原来的数据库保持不动，此外会创建新数据。

读 LOB 数据时这也很重要。LOB 是读一致的，这与所有其他段一样。如果你在上午 9:00 获取一个 LOB 定位器，你从中获取的 LOB 数据就是“上午 9:00 那个时刻的数据”。这就像是你在上午 9:00 打开了一个游标（一个结果集）一样，所生成的行就是那个时间点的数据行。与结果集类似，即使别人后来修改了 LOB 数据。在此，Oracle 会使用 lobsegment，并使用 logindex 的读一致视图来撤销对 LOB 的修改，从而提取获取 LOB 定位器当时的 LOB 数据。它不会使用 logsegment 的 undo 信息，因为根本不会为 logsegment 本身生成 undo 信息。

可以很容易地展示 LOB 是读一致的，考虑以下这个小表，其中有一个行外 LOB（存储在 logsegment 中）：

```
ops$tkyte@ORA10G> create table t
  2     ( id int primary key,
  3     txt clob
  4     )
  5     lob( txt) store as ( disable storage in row )
  6 /
Table created.


ops$tkyte@ORA10G> insert into t values ( 1, 'hello world' );
12.row created.


ops$tkyte@ORA10G> commit;
Commit complete.
```

如果取出 LOB 定位器，并在这个表上打开一个游标，如下：

```
ops$tkyte@ORA10G> declare
  2           l_clob clob;
```

```
3

4          cursor c is select id from t;

5          l_id number;

6    begin

7          select txt into l_clob from t;

8          open c;
```

然后修改行，并提交：

```
9

12.        update t set id = 2, txt = 'Goodbye';

12.        commit;

12.
```

可以看到，通过使用 LOB 定位器和打开的游标，会提供“获取 LOB 定位器或打开游标那个时间点”的数据：

```
12.        dbms_output.put_line( dbms_lob.substr( l_clob, 100, 1 ) );

12.        fetch c into l_id;

12.              dbms_output.put_line( 'id = ' || l_id );

12.        close c;

12. end;

12. /

hello world

id = 1



PL/SQL procedure successfully completed.
```

但是数据库中的数据很可能已经更新/修改：

```
ops$tkyte@ORA10G> select * from t;
```

```
        ID  TXT

---------  ---------------

         2  Goodbye
```

游标 C 的读一致映像来自 undo 段，而 LOB 的读一致映像则来自 LOB 段本身。

由此，我们要考虑这样一个问题：如果不用 undo 段来存储回滚 LOB 所需要的信息，而且 LOB 支持读一致性，那我们怎么避免发生可怕的 ORA-01555:snapshot too old 错误呢？还有一点同样重要，如何控制这些旧版本占用的空间呢？这正是 PCTVERSION 起作用的地方。

PCTVERSION 控制着用于实现 LOB 数据版本化的已分配 LOB 空间的百分比（这些数据库块由某个时间点的 LOB 所用，并处在 lobsegment 的 HWM 以下）。对于许多使用情况来说，默认设置 12.%就足够了，因为在很多情况下，你只是要 INSERT 和获取 LOB（通常不会执行 LOB 的更新；LOB 往往会插入一次，而获取多次）。因此，不必为 LOB 版本化预留太多的空间（甚至可以没有）。

不过，如果你的应用确实经常修改 LOB，倘若你频繁地读 LOB，与此同时另外某个会话正在修改这些 LOB，12.%可能就太小了。如果处理 LOB 时遇到一个 ORA-22924 错误，解决方案不是增加 undo 表空间的大小，也不是增加 undo 保留时间（UNDO_RETENTION），如果你在使用手动 undo 管理，那么增加更多 RBS 空间也不能解决这个问题。而是应该使用以下命令：

```
ALTER TABLE tabname MODIFY LOB (lobname) ( PCTVERSION n );
```

并增加 lobsegment 中为实现数据版本化所用的空间大小。

5. RETENTION 子句

这个子句与 PCTVERSION 子句是互斥的，如何数据库中使用自动 undo 管理，就可以使用这个子句。RETENTION 子句并非在 lobsegment 中保留某个百分比的空间来实现 LOB 的版本化，而是使用基于时间的机制来保留数据。数据库会设置参数 UNDO_RETENTION，指定要把 undo 信息保留多长时间来保证一致读。在这种情况下，这个参数也适用于 LOB 数据。

需要注意，不能使用这个子句来指定保留时间；而要从数据库的 UNDO_RETENTION 设置来继承它。

6. CACHE 子句

前面的 DBMS_METADATA 返回的 CREATE TABLE 语句包括以下内容：

```
LOB ("TXT") STORE AS (... NOCACHE ... )
```

除了 NOCACHE，这个选项还可以是 CACHE 或 CACHE READS。这个子句控制了 lobsegment 数据是否存储在缓冲区缓存中。默认的 NOCACHE 指示，每个访问都是从磁盘的一个直接读，类似地，每个写/修改都是对大盘的一个直接写。CACHE READS 允许缓存从磁盘读的 LOB 数据，但是 LOB 数据的写操作必须直接写至磁盘。CACHE 则允许读和写时都能缓存 LOB 数据。

在许多情况下，默认设置可能对我们并不合适。如果你只有小规模或中等规模的 LOB

（例如，使用 LOB 来存储只有几 KB 的描述性字段），对其缓存就很有意义。如果不缓存，当用户更新描述字段时，还必须等待 I/O 将数据写指磁盘（将执行一个 CHUNK 大小的 I/O，而且用户要等待这个 I/O 完成）。如果你在执行多个 LOB 的加载，那么加载每一行时都必须等待这个 I/O 完成。所以启用执行 LOB 缓存很合理。你可以打开和关闭缓存，来看看会有什么影响：

```
ALTER TABLE tabname MODIFY LOB (lobname) ( CACHE );

ALTER TABLE tabname MODIFY LOB (lobname) ( NOCACHE );
```

对于一个规模很多的初始加载，启用 LOB 的缓存很有意义，这允许 DBWR 在后台将 LOB 数据写至磁盘，而你的客户应用可以继续加载更多的数据。对于频繁访问或修改的小到中等规模的 LOB，缓存就很合理，可以部门让最终用户实时等待物理 I/O 完成。不过，对于一个大小为 50MB 的 LOB，把它放在缓存中就没带道理了。

要记住，此时可以充分使用 Keep 池或回收池。并非在默认缓存中将 lobsegment 数据与所有“常规”数据一同缓存，可以使用保持池或回收池将其分开缓存。采用这种方式，既能缓存 LOB 数据，而且不影响系统中现有数据的缓存。

7. LOB STORAGE 子句

最后，前面的 DBMS_METADATA 返回的 CREATE TABLE 语句还包括以下内容：

```
LOB ("TXT") STORE AS ( ... STORAGE(INITIAL 65536 NEXT 1048576

MINEXTENTS 12.MAXEXTENTS 2147483645 PCTINCREASE 0 FREELISTS 1

FREELIST GROUPS 12.BUFFER_POOL DEFAULT ) ... )
```

也就是说，它有一个完整的存储子句，可以用来控制物理存储特征。需要指出，这个存储子句同样适用于 lobsegment 和 lobindex，对一个段的设置也可以用于另一个段。假设有一个本地管理的表空间，LOB 的相关设置将是 FREELISTS、FREELIST GROUPS 和 BUFFER_POOL。我们在第 12.章讨论过 FREELISTS 和 FREELIST GROUPS 与表段的关系。这些讨论同样适用于 lobindex 段，因为 lobindex 与其他索引段的管理是一样的。如果需要高度并发地修改 LOB，可能最好在索引段上设置多个 FREELISTS。

上一节已经提到，对 LOB 段使用保持池或回收池可能是一个很有用的技术，这样就能缓存 LOB 数据，而且不会“破坏”现有的默认缓冲区缓存。并不是将 LOB 与常规表一同放在块缓冲区中，可以在 SGA 中专门为这些 LOB 对象预留一段专用的内存。BUFFER_POOL 子句可以达到这个目的。

12.7.2 BFILE

我们要讨论的最后一种 LOB 类型是 BFILE 类型。BFILE 类型只是操作系统上一个文件的指针。它用于为这些操作系统文件提供只读访问。

注意 内置包 UTL_FILE 也为操作系统文件提供了读写访问。不过它没有使用 BFILE 类型。

使用 BFILE 时，还有使用一个 Oracle DIRECTORY 对象。DIRECTORY 对象只是将一个操作系统目录映射至数据库中的一个“串”或一个名称（以提供可移植性；你可能想使用

BFILE 中的一个串，而不是操作系统特定的文件名约定）。作为一个小例子，下面创建一个带 BFILE 列的表，并创建一个 DIRECTORY 对象，再插入一行，其中引用了文件系统中的一个文件：

```
ops$tkyte@ORA10G> create table t
  2     ( id int primary key,
  3     os_file bfile
  4     )
  5     /
Table created.


ops$tkyte@ORA10G> create or replace directory my_dir as '/tmp/'
  2     /
Directory created.


ops$tkyte@ORA10G> insert into t values ( 1, bfilename( 'MY_DIR', 'test.dbf' ) );
12.row created.
```

现在，就可以把 BFILE 当成一个 LOB 来处理，因为它就是一个 LOB。例如，我们可以做下面的工作：

```
ops$tkyte@ORA10G> select dbms_lob.getlength(os_file) from t;


DBMS_LOB.GETLENGTH(OS_FILE)
---------------------------
1056768
```

可以看到所指定的文件大小为1MB。注意，这里故意在INSERT语句中使用了MY_DIR。如果使用混合大小写或小写，会得到以下错误：

```
ops$tkyte@ORA10G> update t set os_file = bfilename( 'my_dir', 'test.dbf' );
```

```
12.row updated.



ops$tkyte@ORA10G> select dbms_lob.getlength(os_file) from t;

select dbms_lob.getlength(os_file) from t

       *

ERROR at line 1:

ORA-22285: non-existent directory or file for GETLENGTH operation

ORA-06512: at "SYS.DBMS_LOB", line 566
```

这个例子只是说明：Oracle 中的 DIRECTORY 对象是标识符，而默认情况下标识符都以大写形式存储。BFILENAME 内置函数接受一个串，这个串的大小写必须与数据字典中存储的 DIRECTORY 对象的大小写完全匹配。所以，我们必须在 BFILENAME 函数中使用大写，或者在创建 DIRECTORY 对象时使用加引号的标识符：

```
ops$tkyte@ORA10G> create or replace directory "my_dir" as '/tmp/'
  2  /

Directory created.



ops$tkyte@ORA10G> select dbms_lob.getlength(os_file) from t;



DBMS_LOB.GETLENGTH(OS_FILE)

--------------------------

1056768
```

我不建议使用加引号的标识符；而倾向于在 BFILENAME 调用中使用大写。加引号的标识符属于“异类”，可能会在以后导致混淆。

BFILE 在磁盘上占用的空间不定，这取决于 DIRECTORY 对象名的文件名的长度。在前面的例子中，所得到的 BFILE 长度大约为 35 字节。一般来说，BFILE 会占用大约 20 字节的开销，再加上 DIRECTORY 对象的长度以及文件名本身的长度。

与其他 LOB 数据不同，BFILE 数据不是“读一致”的。由于 BFILE 在数据库之外管理，对 BFILE 解除引用时，不论文件上发生了什么，都会反映到你得到的结果中。所以，如果反复读同一个 BFILE，可能会产生不同的结果，这与对 CLOB、BLOB 或 NCLOB 使用 LOB

定位器不同。

12.8 ROWID/UROWID 类型

最后要讨论的数据类型是 ROWID 和 UROWID 类型。ROWID 是数据库中一行的地址。ROWID 中编入了足够多的信息，足以在磁盘上找到行，以及标识 ROWID 所指向的对象（表等）。ROWID 有一个“近亲”UROWID，它用于表，如 IOT 和通过异构数据库网关访问的没有固定 ROWID 表。UROWID 是行主键值的一个表示，因此，其大小不定，这取决于它指向的对象。

每个表中的每一行都有一个与之关联的 ROWID 或 UROWID。从表中获取时，把它们看作为伪列（pseudo column），这说明它们并不真正存储在行中，而是行的一个推导属性。ROWID 基于行的物理位置生成；它并不随行存储。UROWID 基于行的主键生成，所以从某种意义上讲，好像它是随行存储的，但是事实上并非如此，因为 UROWID 并不作为一个单独的列存在，而只是作为现有列的一个函数。

对于有 ROWID 的行（Oracle 中最常见的行“类型”；除了 IOT 中的行之外，所有行都有 ROWID），以前 ROWID 是不可变的。插入一行时，会为之关联一个 ROWID（一个地址），而且这个 ROWID 会一直与该行关联，直到这一行被删除（被物理地从数据库删除）。但是，后来情况发生了变化，因为现在有些操作可能会导致行的 ROWID 改变，例如：

- 在分区表中更新一行的分区键，使这一行必须从一个分区移至另一个分区。
- 使用 FLASHBACK TABLE 命令将一个数据库表恢复到以前的每个时间点。
- 执行 MOVE 操作以及许多分区操作，如分解或合并分区。
- 使用 ALTER TABLE SHRINK SPACE 命令执行段收缩。

如今，由于 ROWID 可能过一段时间会改变（因为它不再是不可变的），所以不建议把它们作为单独的列物理地存储在数据库表中。也就是说，使用 ROWID 作为一个数据库列的数据类型被认为是一种不好的实践做法。应当避免这种做法，而应使用行的主键（这应该是不可变的），另外引用完整性可以确保数据的完整性。对此用 ROWID 类型是做不到的，不能用 ROWID 创建从子表到一个父表的外键，而且不能保证跨表的完整性。你必须使用主键约束。

那 ROWID 类型有什么用呢？在允许最终用户与数据交互的应用中，ROWID 还是有用的。ROWID 作为行的一个物理地址，要访问任何表中的某一行，这是最快的方法。如果应用从数据库读出数据并将其提供给最终用户，它试图更新这一行时就可以使用 ROWID。应用这种方式，只需最少的工作就可以更新当前行（例如，不需要索引查找再次寻找行），并通过验证行值未被修改来确保这一行与最初读出的行是同一行。所以，在采用乐观锁定的应用中 ROWID 还是有用的。

12.9 小结

在这一章中，我们分析了 Oracle 提供的 22 种基本数据类型，并了解了这些数据类型如何物理地存储，某种类型分别有哪些选项。首先介绍的是字符串，这是最基本的一种类型，并详细讨论了有关多字节字符和原始二进制数据所要考虑的问题。接下来，我们研究了数值类型，包括非常精确的 Oracle NUMBER 类型和 Oracle 10g 以后版本提供的新的浮点类型。

我们还充分考虑了遗留的 LONG 和 LONG RAW 类型，强调了如何绕开它们另辟蹊径，

因为这些类型提供的功能远远比不上 LOB 类型提供的功能。接下来，我们讨论了能存储日期和时间的数据类型。在此介绍了日期运算的基本原理，这个问题很让人费解，如果没有实例演示，将很难搞清楚。最后，在讨论 DATE 和 TIMESTAMP 的一节中，我们讨论了 INTERVAL 类型，并说明了如何最好地加以使用。

从物理存储角度来看，这一章讲得最多、最详细的就是 LOB 一节。LOB 类型常常被开发人员和 DBA 所误解，所以这一节用了很多篇幅来解释 LOB 如何物理地实现，并分析了一些要考虑的性能问题。

我们最后介绍的数据类型是 ROWID/UROWID 类型。由于很明显的原因（你现在应该知道这些原因了），不要把这个数据类型用作数据库列的类型，因为 ROWID 不再是不可变的，而且没有完整性约束可以保证父/子关系。如果需要“指向”另一行，正确的做法可能是存储主键。

第 13 章 分区

分区（partitioning）最早在 Oracle 8.0 中引入，这个过程是将一个表或索引物理地分解为多个更小、更可管理的部分。就访问数据库的应用而言，逻辑上讲只有一个表或一个索引，但在物理上这个表或索引可能由数十个物理分区组成。每个分区都是一个独立的对象，可以独自处理，也可以作为一个更大对象的一部分进行处理。

注意 分区特性是 Oracle 数据库企业版的一个选件，不过要另行收费。标准版中没有这个特性。

在这一章中，我们将分析为什么要考虑使用分区。原因是多方面的，可能是分区能提高数据的可用性，或者是可以减少管理员（DBA）的负担，另外在某些情况下，还可能提高性能。一旦很好地了解了使用分区的原因，接下来将介绍如何对表及其相应的索引进行分区。这个讨论的目的并不是教你有关管理分区的细节，而是提供一个实用的指导，教你如何利用分区来实现应用。

我们还会讨论一个重要的事实：表和索引的分区不一定是数据库的一个“fast=true”设置。从我的经验看，许多开发人员和 DBA 都认为：只要对对象进行分区，就自然而然地会得到性能提升这样一个副作用。但是，分区只是一个工具，对索引或表进行分区时可能发生 3 种情况：使用这些分区表的应用可能运行得更慢；可能运行得更快；有可能没有任何变化。我的意见是，如果你只是一味地使用分区，而不理解它是如何工作的，也不清楚你的应用如何利用分区，那么分区极可能只会对性能产生负面影响。

最后，我们将分析当今世界使用分区的一种非常常见的用法：在 OLTP 和其他运营系统中支持大规模的在线审计跟踪。我们将讨论如何结合分区和段空间压缩来高效地在线存储很大的审计跟踪数据，并且能用最少的工作将这个审计跟踪数据中的旧记录进行归档。

13.1 分区概述

分区有利于管理非常大的表和索引，它使用了一种“分而治之”的逻辑。分区引入了一种分区键（partition key）的概念，分区键用于根据某个区间值（或范围值）、特定值列表或散列函数值执行数据的聚集。如果让我按某种顺序列出分区的好处，这些好处如下：

（1） 提高数据的可用性：这个特点对任何类型的系统都适用，而不论系统本质上是 OLTP 还是仓库系统。

（2） 由于从数据库中去除了大段，相应地减轻了管理的负担。在一个 100GB 的表上执行管理操作时（如重组来删除移植的行，或者在“净化”旧信息后回收表左边的“空白”空间），与在各个 10GB 的表分区上执行 10 次同样的操作相比，前者负担要大得多。另外，通过使用分区，可以让净化例程根本不留下空白空间，这就完全消除了重组的必要！

（3） 改善某些查询的性能：主要在大型仓库环境中有这个好处，通过使用分区，可以消除很大的数据区间，从而不必考虑它们，相应地根本不用访问这些数据。但这在事务性系统中并不适用，因为这种系统本身就只是访问少量的数据。

（4） 可以把修改分布到多个单独的分区上，从而减少大容量 OLTP 系统上的竞争：如果一个段遭遇激烈的竞争，可以把它分为多个段，这就可以得到一个副作用：能成比例地减少竞争。

下面分别讨论使用分区可能带来的这些好处。

13.1.1 提高可用性

可用性的提高源自于每个分区的独立性。对象中一个分区的可用性（或不可用）并不意味着对象本身是不可用的。优化器知道有这种分区机制，会相应地从查询计划中去除未引用的分区。在一个大对象中如果一个分区不可用，你的查询可以消除这个分区而不予考虑，这样 Oracle 就能成功地处理这个查询。

为了展示这种可用性的提高，我们将建立一个散列分区表，其中有两个分区，分别在单独的表空间中。这里将创建一个 EMP 表，它在 EMPNO 列上指定了一个分区键（EMPNO 就是我们的分区键）。在这种情况下，这个结构意味着：对于插入到这个表中的每一行，会对 EMPNO 列的值计算散列，来确定这一行将置于哪个分区（及相应的表空间）中：

```
ops$tkyte@ORA10G> CREATE TABLE emp
  2     ( empno int,
  3     ename varchar2(20)
  4     )
  5     PARTITION BY HASH (empno)
  6     ( partition part_1 tablespace p1,
  7     partition part_2 tablespace p2
  8     )
  9     /
Table created.
```

接下来，我们向表中插入一些数据，然后使用带分区的扩展表名检查各个分区的内容：

```
ops$tkyte@ORA10G> insert into emp select empno, ename from scott.emp
  2 /
14 rows created.



ops$tkyte@ORA10G> select * from emp partition(part_1);
   EMPNO      ENAME
   ---------      --------------------
      7369      SMITH
      7499      ALLEN
      7654      MARTIN
```

```
      7698 BLAKE

      7782 CLARK

      7839 KING

      7876 ADAMS

      7934 MILLER

8 rows selected.



ops$tkyte@ORA10G> select * from emp partition(part_2);

  EMPNO ENAME

 ---------- ----------

      7521 WARD

      7566 JONES

      7788 SCOTT

      7844 TURNER

      7900 JAMES

      7902 FORD

6 rows selected.
```

应该能注意到，数据的“摆放”有些随机。在此这是专门设计的。通过使用散列分区，我们让 Oracle 随机地（很可能均匀地）将数据分布到多个分区上。我们无法控制数据要分布到哪个分区上；Oracle 会根据生成的散列键值来确定。后面讨论区间分区和列表分区时，我们将了解到如何控制哪个分区接收哪些数据。

下面将其中一个表空间离线（例如，模拟一种磁盘出故障的情况），使这个分区中的数据不可用：

```
ops$tkyte@ORA10G> alter tablespace p1 offline;

Tablespace altered.
```

接下来，运行一个查询，这个查询将命中每一个分区，可以看到这个查询失败了：

```
ops$tkyte@ORA10G> select * from emp;

select * from emp

                  *

ERROR at line 1:

ORA-00376: file 12 cannot be read at this time

ORA-01110: data file 12:

'/home/ora10g/oradata/ora10g/ORA10G/datafile/p1.dbf'
```

不过，如果查询不访问离线的表空间，这个查询就能正常工作；Oracle 会消除离线分区而不予考虑。在这个特定的例子中，我使用了一个绑定变量，这只是为了展示 Oracle 肯定能消除离线分区：即使 Oracle 在查询优化时不知道会访问哪个分区，也能在运行是不考虑离线分区：

```
ops$tkyte@ORA10G> variable n number

ops$tkyte@ORA10G> exec :n := 7844;

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> select * from emp where empno = :n;

   EMPNO      ENAME

   ----------      ----------

      7844      TURNER
```

总之，只要优化器能从查询计划消除分区，它就会这么做。基于这一点，如果应用在查询中使用了分区键，就能提高这些应用的可用性。

分区还可以通过减少停机时间来提高可用性。例如，如果有一个 100GB 的表，它划分为 50 个 2GB 的分区，这样就能更快地从错误中恢复。如果某个 2GB 的分区遭到破坏，现在恢复的时间就只是恢复一个 2GB 分区所需的时间，而不是恢复一个 100GB 表的时间。所以从两个方面提高了可用性：

- 优化器能够消除分区，这意味着许多用户可能甚至从未注意到某些数据是不可用的。
- 出现错误时的停机时间会减少，因为恢复所需的工作量大幅减少。

13.1.2 减少管理负担

之所以能减少管理负担，这是因为与在一个大对象上执行操作相比，在小对象上执行同样的操作从本质上讲更为容易、速度更快，而且占用的资源也更少。

例如，假设数据库中有一个 10GB 的索引。如果需要重建这个索引，而该索引未分区，你就必须将整个 10GB 的索引作为一个工作单元来重建。尽管可以在线地重建索引，但是要完全重建完整的 10GB 索引，还是需要占用大量的资源。至少需要在某处有 10GB 的空闲存储空间来存放两个索引的副本，还需要一个临时事务日志表来记录重建索引期间对基表所做的修改。另一方面，如果将索引本身划分为 10 个 1GB 的分区，就可以一个接一个地单独重建各个索引分区。现在只需要原先所需空闲空间的 10%。另外，各个索引的重建也更快（可能是原来的 10 倍），需要向新索引合并的事务修改也更少（到此为止，在线索引重建期间发生的事务修改会更少）。

另外请考虑以下情况：10GB 索引的重建即将完成之前，如果出现系统或软件故障会发生什么。我们所做的全部努力都会付诸东流。如果把问题分解，将索引划分为 1GB 的分区，你最多只会丢掉重建工作的 10%。

或者，你可能只需要重建全部聚集索引的 10%，例如，只是“最新”的数据（活动数据）需要重组，而所有“较旧”的数据（相当静态）不受影响。

最后，请考虑这样一种情况：你发现表中 50%的行都是“移植”行（关于串链/移植行的有关详细内容请参见第 10 章），可能想进行修正。建立一个分区表将有利于这个操作。为了“修正”移植行，你往往必须重建对象，在这种情况下，就是要重建一个表。如果有一个 100GB 的表，就需要在一个非常大的“块”（chunk）上连续地使用 ALTER TABLE MOVE 来执行这个操作。另一方面，如果你有 25 个分区，每个分区的大小为 4GB，就可以一个接一个地重建各个分区。或者，如果你在空余时间做这个工作，而且有充足的资源，甚至可以在单独的会话中并行地执行 ALTER TABLE MOVE 语句，这就很可能会减少整个操作所需的时间。对于一个未分区对象所能做的工作，分区对象中的单个分区几乎都能做到。你甚至可能发现，移植行都集中在一个很小的分区子集中，因此，可以只重建一两个分区，而不是重建整个表。

这里有一个小例子，展示了如何对一个有多个移植行的表进行重建。BIG_TABLE1 和 BIG_TABLE2 都是从 BIG_TABLE 的一个 10,000,000 行的实例创建的（BIG_TABLE 创建脚本见“环境配置”一节）。BIG_TABLE1 是一个常规的未分区表，而 BIG_TABLE2 是一个散列分区表，有 8 个分区（下一节将介绍散列分区；现在只要知道它能把数据相当均匀地分布在 8 个分区上就足够了）：

```
ops$tkyte@ORA10GR1> create table big_table1
  2          ( ID, OWNER, OBJECT_NAME, SUBOBJECT_NAME,
  3          OBJECT_ID, DATA_OBJECT_ID,
  4          OBJECT_TYPE, CREATED, LAST_DDL_TIME,
  5          TIMESTAMP, STATUS, TEMPORARY,
```

```
  6           GENERATED, SECONDARY )
  7    ablespace big1
  8    as
  9    select ID, OWNER, OBJECT_NAME, SUBOBJECT_NAME,
 10           OBJECT_ID, DATA_OBJECT_ID,
 11           OBJECT_TYPE, CREATED, LAST_DDL_TIME,
 12           TIMESTAMP, STATUS, TEMPORARY,
 13           GENERATED, SECONDARY
 14 from big_table.big_table;

Table created.


ops$tkyte@ORA10GR1> create table big_table2
  2           ( ID, OWNER, OBJECT_NAME, SUBOBJECT_NAME,
  3            OBJECT_ID, DATA_OBJECT_ID,
  4            OBJECT_TYPE, CREATED, LAST_DDL_TIME,
  5            TIMESTAMP, STATUS, TEMPORARY,
  6            GENERATED, SECONDARY )
  7            partition by hash(id)
  8            (partition part_1 tablespace big2,
  9            partition part_2 tablespace big2,
 10          partition part_3 tablespace big2,
 11          partition part_4 tablespace big2,
 12          partition part_5 tablespace big2,
```

```
  13         partition part_6 tablespace big2,
  14         partition part_7 tablespace big2,
  15         partition part_8 tablespace big2
  16     )
  17 as
  18 select ID, OWNER, OBJECT_NAME, SUBOBJECT_NAME,
  19         OBJECT_ID, DATA_OBJECT_ID,
  20         OBJECT_TYPE, CREATED, LAST_DDL_TIME,
  21         TIMESTAMP, STATUS, TEMPORARY,
  22         GENERATED, SECONDARY
  23 from big_table.big_table;

Table created.
```

现在，每个表都在自己的表空间中，所以我们可以很容易地查询数据字典，来查看每个表空间中已分配的空间和空闲空间：

```
ops$tkyte@ORA10GR1> select b.tablespace_name,
  2           mbytes_alloc,
  3           mbytes_free
  4       from ( select round(sum(bytes)/1024/1024) mbytes_free,
  5                     tablespace_name
  6           from dba_free_space
  7           group by tablespace_name ) a,
  8           ( select round(sum(bytes)/1024/1024) mbytes_alloc,
  9                     tablespace_name
 10         from dba_data_files
```

```
  11         group by tablespace_name ) b
  12 where a.tablespace_name (+) = b.tablespace_name
  13         and b.tablespace_name in ('BIG1','BIG2')
  14 /

TABLESPACE MBYTES_ALLOC MBYTES_FREE
---------- ------------ -----------
BIG1             1496         344
BIG2             1496         344
```

BIG1 和 BIG2 的大小都大约是 1.5GB，每个表空间都有 344MB 的空闲空间。我们想创建第一个表 BIG_TABLE1：

```
ops$tkyte@ORA10GR1> alter table big_table1 move;
alter table big_table1 move
                *
ERROR at line 1:
ORA-01652: unable to extend temp segment by 1024 in tablespace BIG1
```

但失败了，BIG 1 表空间中要有足够的空闲空间来放下 BIG_TABLE1 的完整副本，同时它的原副本仍然保留，简单地说，我们需要一个很短的时间内有大约两倍的存储空间（可能多一点，也可能少移动，这取决于重建后表的大小）。现在试图对 BIG_TABLE2 执行同样的操作：

```
ops$tkyte@ORA10GR1> alter table big_table2 move;
alter table big_table2 move
                *
ERROR at line 1:
ORA-14511: cannot perform operation on a partitioned object
```

这说明，Oracle 在告诉我们：无法对这个“表”执行 MOVE 操作；我们必须在表的各个分区上执行这个操作。可以逐个地移动（相应地重建和重组）各个分区：

```
ops$tkyte@ORA10GR1> alter table big_table2 move partition part_1;
```

```
Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_2;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_3;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_4;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_5;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_6;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_7;

Table altered.

ops$tkyte@ORA10GR1> alter table big_table2 move partition part_8;

Table altered.
```

对于每个移动，只需要有足够的空闲空间来存放原来数据的 1/8 的副本！因此，假设有先前同样多的空闲空间，这些命令就能成功。我们需要的临时资源将显著减少。不仅如此，如果在移动到 PART_4 后但在 PART_5 完成“移动”之前系统失败了（例如，掉电），我们并不会丢失以前所做的所有工作，这与执行一个 MOVE 语句的情况不同。前 4 个分区仍是“移动”后的状态，等系统恢复时，我们可以从分区 PART_5 继续处理。

有人看到这里可能会说：“哇，8 条语句，要输入这么多语句！”不错，如果有数百个分区（或者更多），这确实有些不切实际。幸运的是，可以很容易地编写一个脚本来解决这个问题，前面的语句则变成以下脚本：

```
ops$tkyte@ORA10GR1> begin
  2          for x in ( select partition_name
  3                       from user_tab_partitions
  4                       where table_name = 'BIG_TABLE2' )
```

```
5          loop
6              execute immediate
7              'alter table big_table2 move partition ' ||
8              x.partition_name;
9          end loop;
10 end;
11 /
PL/SQL procedure successfully completed.
```

你需要的所有信息都能在 Oracle 数据字典中找到，而且大多数实现了分区的站点都有一系列存储过程，可用于简化大量分区的管理。另外，许多 GUI 工具（如 Enterprise Manager）也有一种内置的功能，可以执行这种操作而无需你键入各条命令。

关于分区和管理，还有一个因素需要考虑，这就是在维护数据仓库和归档中使用数据“滑动窗口”。在许多情况下，需要保证数据在最后 N 个时间单位内一直在线。例如，假设需要保证最后 12 个月或最后 5 年的数据在线。如果没有分区，这通常是一个大规模的 INSERT，其后是一个大规模的 DELETE。为此有相对多的 DML，并且会生成大量的 redo 和 undo。如果进行了分区，则只需做下面的工作：

（1） 用新的月（或年，或者是其他）数据加载一个单独的表。

（2） 对这个表充分建立索引（这一步甚至可以在另一个实例中完成，然后传送到这个数据库中）。

（3） 将这个新加载（并建立了索引）的表附加到分区表的最后，这里使用一个快速 DDL 命令：ALTER TABLE EXCHANGE PARTITION。

（4） 从分区表另一端将最旧的分区去掉。

这 样一来，现在就可以很容易地支持包含时间敏感信息的非常大的对象。就数据很容易地从分区表中去除，如果不再需要它，可以简单地将其删除；或者也可以归档到 某个地方。新数据可以加载到一个单独的表中，这样在加载、建索引等工作完成之前就不会影响分区表。在这一章的后面，我们还会看到关于滑动窗口的一个完整的 例子。

简单地说，利用分区，原先让人畏惧的操作（有时甚至是不可行的操作）会变得像在小数据库中一样容易。

13.1.3 改善语句性能

分区最后一个总的（潜在）好处体现在改进语句（SELECT、INSERT、UPDATE、DELETE、MERGE）的性能方面。我们来看两类语句，一种是修改信息的语句，另一种是只读取信息的语句，并讨论在这种情况下可以从分区得到哪些好处。

1. 并行 DML

修改数据库中数据的语句有可能会执行并行 DML（parallel DML，PDML）。采用 PDML 时，Oracle 使用多个线程或进程来执行 INSERT、UPDATE 或 DELETE， 而不是执行一个串行进程。在一个有充足 I/O 带宽的多 CPU 主机上，对于大规模的 DML 操作，速度的提升可能相当显著。在 Oracle9i 以前的版本中，PDML 要求必须分区。如果你的表没有分区，在先前的版本中就不能并行地执行这些操作。如果表确实已经分区，Oracle 会根据对象所有的物理分区数为对象指定一个最大并行度。从很大程度上讲，在 Oracle9i 及以后版本中这个限制已经放松，只有两个突出的例外；如果希望在一个表上执行 PDML，而且这个表的一个 LOB 列上有一个位图索引，要并行执行操作就必须对这个表分区；另外并行度就限制为分区数。不过，总的说来，使用 PDML 并不一定要求进行分区。

注意 我们会在第 14 章更详细地讨论并行操作。

2. 查询性能

在只读查询（SELECT 语句）的性能方面，分区对两类特殊操作起作用：

- 分区消除（partition elimination）：处理查询时不考虑某些数据分区。我们已经看到了一个分区消除的例子。
- 并行操作（parallel operation）：并行全表扫描和并行索引区间扫描就是这种操作的例子。

不过，由此得到的好处很多程度上取决于你使用何种类型的系统。

- OLTP 系统

在 OLTP 系统中，不应该把分区当作一种大幅改善查询性能的方法。实际上，在一个传统的 OLTP 系统中，你必须很小心地应用分区，提防着不要对运行时性能产生负面作用。在传统的 OLTP 系统中，大多数查询很可能几乎立即返回，而且大多数数据库获取可能都通过一个很小的索引区间扫描来完成。因此，以上所列分区性能方面可能 的主要优点在 OLTP 系统中表现不出来。分区消除只在大对象全面扫描时才有用，因为通过分区消除，你可以避免对对象的很大部分做全面扫描。不过，在一个 OLTP 环 境中，本来就不是对大对象全面扫描（如果真是如此，则说明肯定存在严重的设计缺陷）。即使对索引进行了分区，就算是真的能在速度上有所提高，通过扫描较小 索引所得到的性能提升也是微乎其微的。如果某些查询使用了一个索引，而且它们根本无法消除任何分区，你可能会发现，完成分区之后查询实际上运行得反而更慢 了，因为你现在要扫描 5、10 或 20 个更小的索引，而不是一个较大的索引。稍后讨论各种可用的分区索引时还会更详细地讨论这个内容。

尽管如此，有分区的 OLTP 系统确实也有可能得到效率提示。例如，可以用分区来减少竞争，从而提高并发度。可以利用分区将一个表的修改分布到多个物理分区上。并不是只有一个表段和一个索引段，而是可以有 10 个表分区和 20 个索引分区。这就像有 20 个表而不是 1 个表，相应地，修改期间就能减少对这个共享资源的竞争。

至于并行操作（将在下一章更详细地讨论），你可能不希望在一个 OLTP 系统中执行并行查询。你会慎用并行操作，而是交由 DBA 来完成重建、创建索引、收集统计信息等工作。事实上在一个 OLTP 系统中，查询已经有以下特点：即索引访问相当快，因此，分区不会让索引访问的速度有太大的提高（甚至根本没有任何提高）。这并不是说要绝对避免在 OLTP 系统中使用分区；而只是说不要指望通过分区来提供大幅的性能提升。尽管有效情况下分区

能够改善查询的性能，但是这些情况在大多数 OLTP 应用中并不成立。不过在 OLTP 系统中，你还是可以得到另外两个可能的好处：减轻管理负担以及有更高的可用性。

□ 数据仓库系统

在一个数据仓库/决策支持系统中，分区不仅是一个很强大的管理工具，还可以加快处理的速度。例如，你可能有一个大表，需要在其中执行一个即席查询。你总是按销售定额（sales quarter）执行即席查询，因为每个销售定额包含数十万条记录，而你有数百万条在线记录。因此，你想查询整个数据集中相当小的一部分，但是基于销售定额来索引不太可行。这个索引会指向数十万条记录，以这种方式执行索引区间扫描会很糟糕（有关的更多详细内容请参见第 11 章 。 处理许多查询时都要求执行一个全表扫描，但是最后却发现，一方面必须扫描数百万条记录，但另一方面其中大多数记录并不适用于我们的查询。如果使用一种明智 的分区机制，就可以按销售定额来聚集数据，这样在查询某个给定销售定额的数据时，就可以只对这个销售定额的数据进行全面扫描。这在所有可能的解决方案中是 最佳的选择。

另外，在一个数据仓库/决策支持环境中，会频繁地使用并行查询。因此，诸如并行索引区间扫描或并行快速全面索引扫描等操作不仅很有意义，而且对我们很有好处。我们希望充分地使用所有可用的资源，并行查询就提供了这样的一种途径。因此，在数据仓库环境中，分区就意味着很有可能会加快处理速度。

13.2 表分区机制

目前 Oracle 中有 4 种对表分区的方法：

□ 区间分区：可以指定应当存储在一起的数据区间。例如，时间戳在 Jan-2005 内的所有记录都存储在分区 1 中，时间戳在 Feb-2005 内的所有记录都存储在分区 2 中，依此类推。这可能是 Oracle 中最常用的分区机制。

□ 散列分区：我们在这一章一个例子中就已经看到了散列分区。这是指在一个列（或多个列）上应用一个散列函数，行会按这个散列值放在某个分区中。

□ 列表分区：指定一个离散值集，来确定应当存储在一起的数据。例如，可以指定 STATUS 列值在（’A’,’M’,’Z’）中的行放在分区 1 中，STATUS 值在 （‘D’,’P’,’Q’）中的行放在分区 2 中，依此类推。

□ 组合分区：这是区间分区和散列分区的一种组合，或者是区间分区与列表分区的组合。通过组合分区，你可以先对某些数据应用区间分区，再在区间中根据散列或列表来选择最后的分区。

在 下面几节中，我们将介绍各类分区的好处，以及它们之间有什么差别。我们还会说明什么时候应当对不同类型的应用采用何种机制。这一节并不打算对分区语法和所 有可用选项提供一个全面的介绍。相反，我们要靠例子说话，这些例子很简单，也很直观，专门设计为帮助你了解分区，使你对分区如何工作以及如何设计不同类型 的分区有一个简要认识。

注意 要想全面地了解分区语法的所有细节，建议你阅读 Oracle SQL Reference Guide 或 Oracle Administrator’s Guide。另外，Oracle Data Warehousing Guide 也是一个不错的信息源，其中对分区选项做了很好的说明，对于每个计划实现分区的人来说，这是必读的文档。

13.2.1 区间分区

我们要介绍的第一种类型是区间分区表（range partitioned table）。下面的 CREATE TABLE 语句创建了一个使用 RANGE_KEY_COLUMN 列的区间分区表。RANGE_KEY_COLUMN 值严格小于 01-JAN-2005 的所有数据要放在分区 PART_1 中，RANGE_KEY_COLUMN 值严格小于 01-JAN-2006 的所有数据则放在分区 PART_2 中。不满足这两个条件的所有数据（例如，RANGE_KEY_COLUMN 值为 01-JAN-2007 的行）将不能插入，因为它们无法映射到任何分区：

```
ops$tkyte@ORA10GR1> CREATE TABLE range_example
  2           ( range_key_column date ,
  3           data varchar2(20)
  4     )
  5     PARTITION BY RANGE (range_key_column)
  6     ( PARTITION part_1 VALUES LESS THAN
  7     (to_date('01/01/2005','dd/mm/yyyy')),
  8     PARTITION part_2 VALUES LESS THAN
  9     (to_date('01/01/2006','dd/mm/yyyy'))
 10 )
 11 /
Table created.
```

注意 我们在 CREATE TABLE 语句中使用了日期格式 DD/MM/YYYY，使之做到“国际化”。如果使用格式 DD-MON-YYYY，倘若在你的系统上一月份的缩写不是 Jan，这个 CREATE TABLE 语句就会失败，而得到一个 ORA-01843:not a valid month 错误。NLS_LANGUAGE 设置会对此产生影响。不过，我在正文和插入语句中使用了 3 字符的月份缩写，以避免月和日产生歧义，否则有时很难区分哪个分量是日，哪个分量是月。

图 13-1 显示了 Oracle 会检查 RANGE_KEY_COLUMN 的值，并根据这个值，将数据插入到两个分区之一。

图 13-1　区间分区插入示例

为了展示分区区间是严格小于某个值而不是小于或等于某个值，这里插入的行是特别选择的。我们首先插入值 15-DEC-2004，它肯定要放在分区 PART_1 中。我们还插入了日期/时间为 01-JAN-2005 之前一秒（31-dec-2004 23：59：59）的行，这一行也会放到分区 PART_1 中，因为它小于 01-JAN-2005。不过，插入的下一行日期/时间不是严格小于分区区间边界。最后一行显然应该放在分区 PART_2 中，因为它小于 PART_2 的分区区间边界。

可以从各个分区分别执行 SELECT 语句，来确认确实如此：

```
ops$tkyte@ORA10G> select to_char(range_key_column,'dd-mon-yyyy hh24:mi:ss')

  2 from range_example partition (part_1);

TO_CHAR(RANGE_KEY_CO

--------------------

15-dec-2004 00:00:00

31-dec-2004 23:59:59



ops$tkyte@ORA10G> select to_char(range_key_column,'dd-mon-yyyy hh24:mi:ss')

  2 from range_example partition (part_2);
```

```
TO_CHAR(RANGE_KEY_CO

--------------------

01-jan-2005 00:00:00

15-dec-2005 00:00:00
```

你可能想知道，如果插入的日期超出上界会怎么样呢？答案是 Oracle 会产生一个错误：

```
ops$tkyte@ORA10GR1> insert into range_example

  2           ( range_key_column, data )

  3    values

  4           ( to_date( '15/12/2007 00:00:00',

  5           'dd/mm/yyyy hh24:mi:ss' ),

  6           'application data...' );

insert into range_example

            *

ERROR at line 1:

ORA-14400: inserted partition key does not map to any partition
```

假设你想像刚才一样，将 2005 年和 2006 年的日期分别聚集到各自的分区，但是另外你还希望将所有其他日期都归入第三个分区。利用区间分区，这可以使用 MAXVALUE 子句做到这一点，如下所示：

```
ops$tkyte@ORA10GR1> CREATE TABLE range_example

  2    ( range_key_column date ,

  3    data varchar2(20)

  4    )

  5    PARTITION BY RANGE (range_key_column)

  6    ( PARTITION part_1 VALUES LESS THAN

  7    (to_date('01/01/2005','dd/mm/yyyy')),
```

```
  8     PARTITION part_2 VALUES LESS THAN

  9     (to_date('01/01/2006','dd/mm/yyyy'))

 10 PARTITION part_3 VALUES LESS THAN

 11 (MAXVALUE)

 12 )

 13 /

Table created.
```

现在，向这个表插入一个行时，这一行肯定会放入三个分区中的某一个分区中，而不会再拒绝任何行，因为分区 PART_3 可以接受不能放在 PART_1 或 PART_2 中的任何 RANG_KEY_COLUMN 值（即使 RANGE_KEY_COLUMN 值为 null，也会插入到这个新分区中）。

13.2.2 散列分区

对一个表执行散列分区（hash partitioning）时，Oracle 会对分区键应用一个散列函数，以此确定数据应当放在 N 个分区中的哪一个分区中。Oracle 建议 N 是 2 的一个幂（2、4、8、16 等），从而得到最佳的总体分布，稍后会看到这确实是一个很好的建议。

1. 散列分区如何工作

散列分区设计为能使数据很好地分布在多个不同设备（磁盘）上，或者只是将数据聚集到更可管理的块（chunk）上，为表选择的散列键应当是惟一的一个列或一组列，或者至少有足够多的相异值，以便行能在多个分区上很好地（均匀地）分布。如果你选择一个只有 4 个相异值的列，并使用两个分区，那么最后可能把所有行都散列到同一个分区上，这就有悖于分区的最初目标！

在这里，我们将创建一个有两个分区的散列表。在此使用名为 HASH_KEY_COLUMN 的列作为分区键。Oracle 会取这个列中的值，并计算它的散列值，从而确定这一行将存储在哪个分区中：

```
ops$tkyte@ORA10G> CREATE TABLE hash_example

  2     ( hash_key_column date,

  3     data varchar2(20)

  4     )

  5     PARTITION BY HASH (hash_key_column)
```

```
  6    ( partition part_1 tablespace p1,
  7    partition part_2 tablespace p2
  8    )
  9    /

Table created.
```

图 13-2 显示了 Oracle 会检查 HASH_KEY_COLUMN 中的值，计算散列，确定给定行会出现在两个分区中的哪一个分区中：

图 13-2 散列分区插入示例

前面已经提到过，如果使用散列分区，你将无从控制一行最终会放在哪个分区中。Oracle 会应用散列函数，并根据散列的结果来确定行会放在哪里。如果你由于某种原因希望将某个特定行放在分区 PART_1 中， 就不应该使用散列分区，实际上，此时也不能使用散列分区。行会按散列函数的“指示”放在某个分区中，也就是说，散列函数说这一行该放在哪个分区，它就会放 在哪个分区中。如果改变散列分区的个数，数据会在所有分区中重新分布（向一个散列分区表增加或删除一个分区时，将导致所有数据都重写，因为现在每一行可能 属于一个不同的分区）。

如果你有一个大表（如“减少管理负担”一节中所示的表），而且你想对它“分而治之”，此时散列分区最有用。你不用管理一个大表，而只是管理 8 或 16 个 较小的“表”。从某种程度上讲，散列分区对于提高可用性也很有用，这在“提高可用性”一节中已经介绍过；临

时丢掉一个散列分区，就能访问所有余下的分区。 也许有些用户会受到影响，但是很有可能很多用户根本不受影响，但是很有可能很多用户根本不受影响。另外，恢复的单位现在也更小了。你不用恢复一个完整的大 表；而只需恢复表中的一小部分。最后一点，散列分区还有利于存在高度更新竞争的环境，这在“改善语句性能”一节讨论 OLTP 系统时已经提到过。我们可以不使一个段“很热”，而是可以将一个段散列分区为 16 个“部分”，这样一来，现在每一部分都可以接收修改。

2. 散列分区数使用 2 的幂

我在前面提到过，分区数应该是 2 的幂，这很容易观察。为了便于说明，我们建立了一个存储过程，它会自动创建一个有 N 个分区的散列分区表（N 是一个参数）。这个过程会构成一个动态查询，按分区获取其中的行数，再按分区显示行数，并给出行数的一个简单直方图。最后，它会打开这个查询，以便我们看到结果。这个过程首先创建散列表。我们将使用一个名为 T 的表：

```
ops$tkyte@ORA10G> create or replace
  2     procedure hash_proc
  3             ( p_nhash in number,
  4             p_cursor out sys_refcursor )
  5             authid current_user
  6     as
  7             l_text long;
  8             l_template long :=
  9             'select $POS$ oc, "p$POS$" pname, count(*) cnt ' ||
 10         'from t partition ( $PNAME$ ) union all ';
 11 begin
 12         begin
 13                 execute immediate 'drop table t';
 14                 exception when others
 15                         then null;
 16         end;
```

```
17
18      execute immediate '
19      CREATE TABLE t ( id )
20              partition by hash(id)
21              partitions ' || p_nhash || '
22       as
23      select rownum
24      from all_objects';
```

接下来，动态构造一个查询，按分区获取行数。这里使用了前面定义的“模板”查询。对于每个分区，我们将使用分区扩展的表名来收集分区中的行数，并把所有行数合在一起：

```
25
26      for x in ( select partition_name pname,
27                              PARTITION_POSITION pos
28                  from user_tab_partitions
29                  where table_name = 'T'
30                  order by partition_position )
31      loop
32              l_text := l_text ||
33              replace(
34                      replace(l_template,
35                              '$POS$', x.pos),
36                              '$PNAME$', x.pname );
37      end loop;
```

现在，取这个查询，选出分区位置（PNAME）和该分区中的行数（CNT）。通过使用RPAD，可以构造一个相当基本但很有效的直方图：

```
   38
   39         open p_cursor for
   40                 'select pname, cnt,
   41                         substr( rpad("*",30*round( cnt/max(cnt)over(),2),"*"),1,30) hg
   42                 from (' || substr( l_text, 1, length(l_text)-11 ) || ')
   43                 order by oc';
   44
   45 end;
   46 /
Procedure created.
```

如果针对输入值 4 运行这个过程，这表示有 4 个散列分区，就会看到类似如下的输出：

```
ops$tkyte@ORA10G> variable x refcursor
ops$tkyte@ORA10G> set autoprint on
ops$tkyte@ORA10G> exec hash_proc( 4, :x );
PL/SQL procedure successfully completed.
PN    CNT         HG
--    ----------  ------------------------------
p1    12141       ******************************
p2    12178       ******************************
p3    12417       ******************************
p4    12105       ******************************
```

这个简单的直方图展示了数据很均匀地分布在这 4 个分区中。每个分区中的行数都很接近。不过，如果将 4 改成 5，要求有 5 个散列分区，就会看到以下输出：

```
ops$tkyte@ORA10G> exec hash_proc( 5, :x );
```

```
PL/SQL procedure successfully completed.

PN    CNT         HG

--    ----------  ------------------------------

p1    6102        **************

p2    12180       ****************************

p3    12419       ****************************

p4    12106       ****************************

p5    6040        **************
```

这个直方图指出，第一个和最后一个分区中的行数只是另外三个分区中行数的一半。数据根本没有得到均匀的分布。我们会看到，如果有 6 个和 7 个散列分区，这种趋势还会继续：

```
ops$tkyte@ORA10G> exec hash_proc( 6, :x );

PL/SQL procedure successfully completed.



PN    CNT         HG

--    ----------  ------------------------------

p1    6104        **************

p2    6175        ***************

p3    12420       ****************************

p4    12106       ****************************

p5    6040        **************

p6    6009        **************

6 rows selected.
```

```
ops$tkyte@ORA10G> exec hash_proc( 7, :x );

PL/SQL procedure successfully completed.



PN     CNT          HG

--       ----------     ------------------------------

p1     6105         ***************

p2     6176         ***************

p3     6161         ***************

p4     12106        ******************************

p5     6041         ***************

p6     6010         ***************

p7     6263         ***************

7 rows selected.
```

散列分区数再回到 2 的幂值（8）时，又能到达我们的目标，实现均匀分布：

```
ops$tkyte@ORA10G> exec hash_proc( 8, :x );

PL/SQL procedure successfully completed.



PN     CNT          HG

--       ----------     ------------------------------

p1     6106         ****************************

p2     6178         ****************************

p3     6163         ****************************

p4     6019         ****************************
```

```
p5    6042    **************************

p6    6010    **************************

p7    6264    ***************************

p8    6089    ***************************

8 rows selected.
```

再继续这个实验，分区最多达到 16 个，你会看到如果分区数为 9～15，也存在同样的问题，中间的分区存放的数据多，而两头的分区中数据少，数据的分布是斜的；而达到 16 个分区时，你会再次看到数据分布是直的。再达到 32 个分区和 64 个分区时也是如此。这个例子只是要指出：散列分区数要使用 2 的幂，这一点非常重要。

13.2.3 列表分区

列表分区（list partitioning）是 Oracle9i Release 1 的一个新特性。它提供了这样一种功能，可以根据离散的值列表来指定一行位于哪个分区。如果能根据某个代码来进行分区（如州代码或区代码），这通常很有用。例如，你可能想把 Maine 州（ME）、New Hampshire 州（NH）、Vermont 州（VT）和 Massachusetts 州（MA）中所有人的记录都归至一个分区中，因为这些州相互之间挨得很近，而且你的应用按地理位置来查询数据。类似地，你可能希望将 Connecticut 州（CT）、Rhode Island 州（RI）和 New York 州（NY）的数据分组在一起。

对此不能使用区间分区，因为第一个分区的区间是 ME 到 VT，第二个区间是 CT 到 RI。这两个区间有重叠。而且也不能使用散列分区，因为这样你就无法控制给定行要放到哪个分区中；而要由 Oracle 提供的内置散列函数来控制。

利用列表分区，我们可以很容易地完成这个定制分区机制：

```
ops$tkyte@ORA10G> create table list_example
  2    ( state_cd varchar2(2),
  3    data varchar2(20)
  4    )
  5    partition by list(state_cd)
  6    ( partition part_1 values ( 'ME', 'NH', 'VT', 'MA' ),
  7    partition part_2 values ( 'CT', 'RI', 'NY' )
  8    )
  9    /
```

```
Table created.
```

图 13-3 显示了 Oracle 会检查 STATE_CD 列，并根据其值将行放在正确的分区中。

就像区间分区一样，如果我们想插入列表分区中未指定的一个值，Oracle 会向客户应用返回一个合适的错误。换句话说，没有 DEFAULT 分区的列表分区表会隐含地施加一个约束（非常像表上的一个检查约束）：

```
ops$tkyte@ORA10G> insert into list_example values ( 'VA', 'data' );

insert into list_example values ( 'VA', 'data' )

                *

ERROR at line 1:

ORA-14400: inserted partition key does not map to any partition
```

图 13-3 列表分区插入示例

如果想像前面一样把这个 7 个州分别聚集到各自的分区中，另外把其余的所有州代码放在第三个分区中（或者，实际上对于所插入的任何其他行，如果 STATE_CD 列值不是以上 7 个州代码之一，就要放在第三个分区中），就可以使用 VALUES（DEFAULT）子句。在此，我们将修改表，增加这个分区（也可以在 CREATE TABLE 语句中使用这个子句）：

```
ops$tkyte@ORA10G> alter table list_example
```

```
  2     add partition

  3     part_3 values ( DEFAULT );

Table altered.



ops$tkyte@ORA10G> insert into list_example values ( 'VA', 'data' );

1 row created.
```

值列表中未显式列出的所有值都会放到这个（DEFAULT）分区中。关于 DEFAULT 的使用，有一点要注意：一旦列表分区表有一个 DEFAULT 分区，就不能再向这个表中增加更多的分区了：

```
ops$tkyte@ORA10G> alter table list_example

  2 add partition

  3 part_4 values( 'CA', 'NM' );

alter table list_example

            *

ERROR at line 1:

ORA-14323: cannot add partition when DEFAULT partition exists
```

此时必须删除 DEFAULT 分区，如何增加 PART_4，再加回 DEFAULT 分区。原因在于，原来 DEFAULT 分区可以有列表分区键值为 CA 或 NM 的行，但增加 PART_4 之后，这些行将不再属于 DEFAULT 分区。

13.2.4 组合分区

最后我们会看到组合分区（composite partitioning）的一些例子，组合分区是区间分区和散列分区的组合，或者是区间分区与列表分区的组合。

在组合分区中，顶层分区机制总是区间分区。第二级分区机制可能是列表分区或散列分区（在 Oracle9i Release 1 及以前的版本中，只支持散列子分区，而没有列表分区）。有意思的是，使用组合分区时，并没有分区段，而只有子分区段。分区本身并没有段（这就类似于分区表没有段）。数据物理的存储在子分区段上，分区成为一个逻辑容器，或者是一个指向实际子分区的容器。

在下面的例子中，我们将查看一个区间-散列组合分区。在此对区间分区使用的列集不同于散列分区使用的列集。并不是非得如此，这两层分区也可以使用同样的列集：

```
ops$tkyte@ORA10G> CREATE TABLE composite_example
  2     ( range_key_column date,
  3     hash_key_column int,
  4     data varchar2(20)
  5     )
  6     PARTITION BY RANGE (range_key_column)
  7     subpartition by hash(hash_key_column) subpartitions 2
  8     (
  9           PARTITION part_1
 10         VALUES LESS THAN(to_date('01/01/2005','dd/mm/yyyy'))
 11         (subpartition part_1_sub_1,
 12         subpartition part_1_sub_2
 13 ),
 14 PARTITION part_2
 15 VALUES LESS THAN(to_date('01/01/2006','dd/mm/yyyy'))
 16         (subpartition part_2_sub_1,
 17         subpartition part_2_sub_2
 18         )
 19 )
 20 /
Table created.
```

在区间-散列组合分区中，Oracle 首先会应用区间分区规则，得出数据属于哪个区间。然后再应用散列函数，来确定数据最后要放在哪个物理分区中。这个过程如图 13-4 所示。

图 13-4 区间-散列组合分区示例

因 此，利用组合分区，你就能把数据先按区间分解，如果认为某个给定的区间还太大，或者认为有必要做进一步的分区消除，可以再利用散列或列表将其再做分解。有 意思的是，每个区间分区不需要有相同数目的子分区；例如，假设你在对一个日期列完成区间分区，以支持数据净化（快速而且容易地删除所有就数据）。在 2004 年，CODE_KEY_COLUMN 值为“奇数”的数据量与 CODE_KEY_COLUMN 值为“偶数”的数据量是相等的。但是到了 2005 年，你发现与奇数吗相关的记录数是偶数吗相关的记录数的两倍，所以你希望对应奇数码有更多的子分区。只需定义更多的子分区，就能相当容易地做到这一点：

```
ops$tkyte@ORA10G> CREATE TABLE composite_range_list_example
  2  ( range_key_column date,
  3  code_key_column int,
  4  data varchar2(20)
  5  )
  6  PARTITION BY RANGE (range_key_column)
```

```
 7     subpartition by list(code_key_column)
 8     (
 9     PARTITION part_1
 10         VALUES LESS THAN(to_date('01/01/2005','dd/mm/yyyy'))
 11         (subpartition part_1_sub_1 values( 1, 3, 5, 7 ),
 12         subpartition part_1_sub_2 values( 2, 4, 6, 8 )
 13          ),
 14 PARTITION part_2
 15         VALUES LESS THAN(to_date('01/01/2006','dd/mm/yyyy'))
 16         (subpartition part_2_sub_1 values ( 1, 3 ),
 17         subpartition part_2_sub_2 values ( 5, 7 ),
 18         subpartition part_2_sub_3 values ( 2, 4, 6, 8 )
 19          )
 20 )
 21 /
Table created.
```

在此，最后总共有 5 个分区：分区 PART_1 有两个子分区，分区 PART_2 有 3 个子分区。

13.2.5 行移动

你可能想知道，在前面所述的各种分区机制中，如果用于确定分区的列有修改会发生什么。需要考虑两种情况：

- 修改不会导致使用一个不同的分区；行仍属于原来的分区。这在所有情况下都得到支持。
- 修改会导致行跨分区移动。只有当表启用了行移动时才支持这种情况；否则，会产生一个错误。

这些行为很容易观察。在前面的例子中，我们向 RANGE_EXAMPLE 表的 PART_1 插入了两行：

```
ops$tkyte@ORA10G> insert into range_example
  2           ( range_key_column, data )
  3     values
  4           ( to_date( '15-dec-2004 00:00:00',
  5                      'dd-mon-yyyy hh24:mi:ss' ),
  6                      'application data...' );
1 row created.


ops$tkyte@ORA10G> insert into range_example
  2           ( range_key_column, data )
  3     values
  4           ( to_date( '01-jan-2005 00:00:00',
  5                      'dd-mon-yyyy hh24:mi:ss' )-1/24/60/60,
  6                      'application data...' );
1 row created.


ops$tkyte@ORA10G> select * from range_example partition(part_1);
RANGE_KEY      DATA
---------              --------------------
15-DEC-04         application data...
31-DEC-04         application data...
```

取其中一行，并更新其 RANGE_KEY_COLUMN 值，不过更新后它还能放在 PART_1 中：

```
ops$tkyte@ORA10G> update range_example
```

```
  2           set range_key_column = trunc(range_key_column)

  3     where range_key_column =

  4           to_date( '31-dec-2004 23:59:59',

  5                    'dd-mon-yyyy hh24:mi:ss' );

1 row updated.
```

不出所料，这会成功：行仍在分区 PART_1 中。接下来，再把 RANGE_KEY_COLUMN 更新为另一个值，但这次更新后的值将导致它属于分区 PART_2：

```
ops$tkyte@ORA10G> update range_example

  2 set range_key_column = to_date('02-jan-2005','dd-mon-yyyy')

  3 where range_key_column = to_date('31-dec-2004','dd-mon-yyyy');

update range_example

       *

ERROR at line 1:

ORA-14402: updating partition key column would cause a partition change
```

这会立即产生一个错误，因为我们没有显式地启用行移动。在 Oracle8i 及以后的版本中，可以在这个表上启用行移动（row movement），以允许从一个分区移动到另一个分区。

注意　Oracle 8.0 中没有行移动功能；在这个版本中，你必须先删除行，再重新将其插入。

不过，要注意这样做有一个小小的副作用；行的 ROWID 会由于更新而改变：

```
ops$tkyte@ORA10G> select rowid

  2 from range_example

  3 where range_key_column = to_date('31-dec-2004','dd-mon-yyyy');

ROWID

------------------

AAARmfAAKAAAI+aAAB
```

```
ops$tkyte@ORA10G> alter table range_example
  2      enable row movement;
Table altered.


ops$tkyte@ORA10G> update range_example
  2             set range_key_column = to_date('02-jan-2005','dd-mon-yyyy')
  3      where range_key_column = to_date('31-dec-2004','dd-mon-yyyy');
1 row updated.


ops$tkyte@ORA10G> select rowid
  2      from range_example
  3      where range_key_column = to_date('02-jan-2005','dd-mon-yyyy');
ROWID
------------------
AAARmgAAKAAAI+iAAC
```

既然知道执行这个更新时行的 ROWID 会改变，所以要启用行移动，这样才允许更新分区键。

注意 在其他一些情况下，ROWID 也有可能因为更新而改变。更新 IOT 的主键可能导致 ROWID 改变，该行的通用 ROWID（UROWID）也会改变。Oracle 10g 的 FLASHBACK TABLE 命令可能改变行的 ROWID，此外 Oracle 10g 的 ALTER TABLE SHRINK 命令也可能使行的 ROWID 改变。

要知道，执行行移动时，实际上在内部就好像先删除了这一行，然后再将其重新插入。这会更新这个表上的索引，删除旧的索引条目，再插入一个新条目。此时会完成 DELETE 再加一个 INSERT 的相应物理工作。不过，尽管在此执行了行的物理删除和插入，在 Oracle 看来却还是一个更新，因此，不会导致 INSERT 和 DELETE 触发器触发，只有 UPDATE 触发器会触发。另外，由于外键约束可能不允许 DELETE 的子表也不会触发 DELETE 触发器。不过，还是要对将完成的额外工作有所准备；行移动的开销比正常的 UPDATE 昂贵得多。因此，如果构建的系统会频繁修改分区键，而且这种修改会导致分区移动，这实在是一个糟糕的设计决策。

13.2.6 表分区机制小结

一般来讲，如果将数据按某个（某些）值逻辑聚集，区间分区就很有用。基于时间的数据就是这方面经典的例子，如按“销售定额”、“财政年度”或“月份”分区。在许多情况下，区间分区都能利用分区消除，这包括使用完全相等性和区间（小于、大于、介于…之间等 。

如 果不能按自然的区间进行分区，散列分区就很合适。例如，如果必须加载一个表，其中装满与人口普查相关的数据，可能无法找到一个合适的属性来按这个属性完成 区间分区。不过，你可能还是想得到分区提供的管理、性能和可用性提升等诸多好处。在此，只需选择惟一的一个列或几乎惟一的一个列集，对其计算散列。这样一 来，无论有多少个分区，都能得到均匀的数据分布。使用完全相等性或 IN（value,value,…）时，散列分区对象可以利用分区消除，但是使用数据区间时，散列分区则无法利用分区消除。

如果数据中有一列有一组离散值，而且根据应用使用这一列的方式来看，按这一列进行分区很有意义（例如，这样一来，查询中可以轻松地利用分区消除），这种数据 就很适合采用列表分区。列表分区的经典例子包括按州或区域代码分区，实际上，一般来讲许多“代码”性属性都很适合应用列表分区。

如果某些数据逻辑上可以进行区间分区，但是得到的区间分区还是太小，不能有效地管理，就可以使用组合分区。可以先应用区间分区，再进一步划分各个区间，按一个散列函数或使用列表来分区。这样就能将 I/O 请求分布到任何给定大分区中的多个磁盘上。另外，现在可以得到 3 个层次的分区消除。如果在区间分区键上查询，Oracle 就能消除任何不满足条件的区间分区。如果向查询增加散列或列表键，Oracle 可以消除该区间中其他的散列或列表分区。如果只是在散列或列表键上查询（而不使用区间分区键），Oracle 就只会查询各个区间分区中的这些散列或列表子分区。

我们建议，如果可以按某个属性自然地对数据完成区间分区，就应该使用区间分区，而不是散列分区或列表分区。散列和列表分区能提供分区的许多突出优点，但是在分区消除方面都不如区间分区有用。如果所得到的区间分区太大，不能很好地管理；或者如果你想使用所有 PDML 功能或对一个区间分区使用并行索引扫描，则建议在区间分区中再使用散列或列表分区。

13.3 索引分区

索引与表类似，也可以分区。对索引进行分区有两种可能的方法：

- 随表对索引完成相应的分区：这也称为局部分区索引（locally pertitioned index）。每个表分区都有一个索引分区，而且只索引该表分区。一个给定索引分区中的所有条目都指向一个表分区，表分区中的所有行都表示在一个索引分区中。
- 按区间对索引分区：这也称为全局分区索引（globally partitioned index）。在此，索引按区间分区（或者在 Oracle 10g 中该可以按散列分区），一个索引分区可能指向任何（和所有）表分区。

图 13-5 展示了局部索引和全局索引的区别。

图 13-5 局部和全局索引分区

对于全局分区索引，要注意实际上索引分区数可能不同于表分区数。

由于全局索引只按区间或散列分区，如果希望有一个列表或组合分区索引，就必须使用局部索引。局部索引会使用底层表相同的机制分区。

注意 全局索引的散列分区是 Oracle 10g Release 1 及以后的版本中才有的新特性。在 Oracle9i 及以前的版本中，只能按区间进行全局分区。

13.3.1 局部索引

Oracle 划分了以下两类局部索引：

- 局部前缀索引（local prefixed index）：在这些索引中，分区键在索引定义的前几列上。例如，一个表在名为 LOAD_DATE 的列上进行区间分区，该表上的局部前缀索引就是 LOAD_DATE 作为其索引列列表中的第一列。
- 局部非前缀索引（local nonprefixed index）：这些索引不以分区键作为其列列表的前几列。索引可能包含分区键列，也可能不包含。

这 两类索引都可以利用分区消除，它们都支持惟一性（只有非前缀索引包含分区键）等。事实上，使用局部前缀索引的查询总允许索引分区消除，而使用局部非前缀索 引的查询可能不允许。正是由于这个原因，所以在某些人看来局部非前缀索引“更慢”，它们不能保证分区消除（但确实可以支持分区消除）。

如果查询中将索引用作访问表的初始路径，那么从本质来讲，局部前缀索引并不比局部非前缀索引更好。我的意思是说，如何查询把“扫描一个索引”作为第一步，那么前缀索引和非前缀索引之间并没有太大的差别。

1. 分区消除行为

如果查询首先访问索引，它是否能消除分区完全取决于查询中的谓词。要说明这一点，举一个小例子会很有帮助。下面的代码创建了一个表 PARTITIONED_TABLE，它在一个数字列 A 上进行区间分区，使得小于 2 的值都在分区 PART_1 中，小于 3 的值则都在分区 PART_2 中：

```
ops$tkyte@ORA10G> CREATE TABLE partitioned_table
```

```
  2     ( a int,
  3     b int,
  4     data char(20)
  5     )
  6     PARTITION BY RANGE (a)
  7     (
  8           PARTITION part_1 VALUES LESS THAN(2) tablespace p1,
  9           PARTITION part_2 VALUES LESS THAN(3) tablespace p2
 10 )
 11 /

Table created.
```

然后我们创建一个局部前缀索引 LOCAL_PREFIXED 和一个局部非前缀索引 LOCAL_NONPREFIXED。注意，非前缀索引在其定义中没有以 A 作为其最前列，这是这一点使之成为一个非前缀索引：

```
ops$tkyte@ORA10G> create index local_prefixed on partitioned_table (a,b) local;

Index created.



ops$tkyte@ORA10G> create index local_nonprefixed on partitioned_table (b) local;

Index created.
```

接下来，我们向一个分区中插入一些数据，并收集统计信息：

```
ops$tkyte@ORA10G> insert into partitioned_table
  2           select mod(rownum-1,2)+1, rownum, 'x'
  3           from all_objects;

48967 rows created.

ops$tkyte@ORA10G> begin
```

```
  2          dbms_stats.gather_table_stats
  3               ( user,
  4                'PARTITIONED_TABLE',
  5                cascade=>TRUE );
  6    end;
  7    /

PL/SQL procedure successfully completed.
```

将表空间 P2 离线，其中包含用于表和索引的 PART_2 分区：

```
ops$tkyte@ORA10G> alter tablespace p2 offline;

Tablespace altered.
```

表空间 P2 离线后，Oracle 就无法访问这些特定的索引分区。这就好像是我们遭遇了“介质故障”，导致分区不可用。现在我们查询这个表，来看看不同的查询需要哪些索引分区。第一个查询编写为允许使用局部前缀索引：

```
ops$tkyte@ORA10G> select * from partitioned_table where a = 1 and b = 1;

         A          B DATA
---------- ---------- --------------------
         1          1 x
```

这个查询成功了，通过查看解释计划，可以看到这个查询为什么能成功。我们将使用内置包 DBMS_XPLAN 来查看这个查询访问了哪些分区。输出中的 PSTART (分区开始)和 PSTOP（分区结束）这两列准确地显示出，这个查询要想成功需要哪些分区必须在线而且可用：

```
ops$tkyte@ORA10G> delete from plan_table;

4 rows deleted.



ops$tkyte@ORA10G> explain plan for

  2    select * from partitioned_table where a = 1 and b = 1;

Explained.
```

```
ops$tkyte@ORA10G> select * from table(dbms_xplan.display);



PLAN_TABLE_OUTPUT

----------------------------------------------------------------------------

| Operation                                | Name                | Rows  |
Pstart | Pstop      |

----------------------------------------------------------------------------

|  SELECT  STATEMENT                       |                     |
1      |       |       |

|  PARTITION  RANGE  SINGLE                |                     | 1     |
1      | 1     |

|  TABLE ACCESS BY LOCAL INDEX ROWID | PARTITIONED_TABLE | 1     |
1      | 1     |

|    INDEX  RANGE  SCAN                    | LOCAL_PREFIXED      | 1     |
1      | 1     |

----------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

3 - access("A"=1 AND "B"=1)
```

注意　这里对 DBMS_XPLAN 输出进行了编辑，删除了与这个列在无关的信息，从而减少了例子的篇幅，以便在一页内能放下。

因此，使用 LOCAL_PREFIXED 的查询成功了。优化器能消除 LOCAL_PREFIXED 的 PART_2 不予考虑，因为我们在查询中指定了 A=1，而且在计划中可以清楚地看到 PSTART 和 PSTOP 都等于 1.分区消除帮助了我们。不过，第二个查询却失败了：

```
ops$tkyte@ORA10G> select * from partitioned_table where b = 1;

ERROR:

ORA-00376: file 13 cannot be read at this time
```

```
ORA-01110: data file 13: '/home/ora10g/.../o1_mf_p2_1dzn8jwp_.dbf'

no rows selected
```

通过使用同样的技术，可以看到这是为什么：

```
ops$tkyte@ORA10G> delete from plan_table;

4 rows deleted.



ops$tkyte@ORA10G> explain plan for

  2  select * from partitioned_table where b = 1;

Explained.



ops$tkyte@ORA10G> select * from table(dbms_xplan.display);



PLAN_TABLE_OUTPUT

--------------------------------------------------------------------------------

| Operation                                 | Name                        | Rows      |
Pstart | Pstop            |

--------------------------------------------------------------------------------

|  SELECT  STATEMENT                                |                             |
1          |         |         |

|  PARTITION  RANGE  ALL                          |                             | 1         |
1       | 2       |

|   TABLE  ACCESS  BY  LOCAL  INDEX  ROWID  | PARTITIONED_TABLE       |
1          | 1       | 2       |

|    INDEX RANGE SCAN                         | LOCAL_NONPREFIXED     | 1         |
1       | 2       |

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

```
Predicate Information (identified by operation id):

---------------------------------------------------

3 - access("B"=1)
```

在此，优化器不能不考虑 LOCAL_NONPREFIXED 的 PART_2，为了查看是否有 B=1，索引的 PART_1 和 PART_2 都必须检查。在此，局部非前缀索引存在一个性能问题：它不能像前缀索引那样，在谓词中使用分区键。并不是说前缀索引更好，我们的意思是：要使用非前缀索引，必须使用一个允许分区消除的查询。

```
ops$tkyte@ORA10G> drop index local_prefixed;

Index dropped.



ops$tkyte@ORA10G> select * from partitioned_table where a = 1 and b = 1;

         A          B     DATA

---------- ---------- --------------------

         1          1     x
```

它会成功，但是正如我们所见，这里使用了先前失败的索引。该计划显示出，在此 Oracle 能利用分区消除，有了谓词 A=1，就有了足够的信息可以让数据库消除索引分区 PART_2 而不予考虑：

```
ops$tkyte@ORA10G> delete from plan_table;

4 rows deleted.



ops$tkyte@ORA10G> explain plan for

  2  select * from partitioned_table where a = 1 and b = 1;

Explained.



ops$tkyte@ORA10G> select * from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT

------------------------------------------------------------------------------

| Operation                                | Name                         | Rows      |
Pstart | Pstop            |

------------------------------------------------------------------------------

|   SELECT    STATEMENT                                |                              |
1          |         |         |

|  PARTITION  RANGE  SINGLE                    |                                | 1         |
1       | 1       |

|    TABLE  ACCESS  BY  LOCAL  INDEX  ROWID  | PARTITIONED_TABLE       |
1          | 1       | 1       |

|     INDEX RANGE SCAN                        | LOCAL_NONPREFIXED     | 1         |
1       | 1       |

------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter("A"=1)

3 - access("B"=1)
```

注意 PSTART 和 PSTOP 列值为 1 和 1.这就证明，优化器甚至对非前缀局部索引也能执行分区消除。

如果你频繁地用以下查询来查询先前的表：

```
select ... from partitioned_table where a = :a and b = :b;

select ... from partitioned_table where b = :b;
```

可以考虑在（b,a）上使用一个局部非前缀索引。这个索引对于前面的两个查询都是有用的。（a,b）上的局部前缀索引只对第一个查询有用。

这 里的关键是，不必对非前缀索引退避三舍，也不要认为非前缀索引是主要的性能障碍。如果你有多个如前所列的查询（可以得益于非前缀索引），就应该考虑使用一 个非前缀索引。重点是，要尽可能保证查询包含的谓词允许索引分区消除。使用前缀局部索引可以保证这一点，使用非前缀索引则不能保证。还要考虑如何使用索 引。如果将索引用作查询计划中的第一步，那么这两种类型的索引没有多少差别。

2. 局部索引和惟一约束

为了保证惟一性（这包括 UNIQUE 约束或 PRIMARY KEY 约束），如果你想使用一个局部索引来保证这个约束，那么分区键必须包括在约束本身中。在我看来，这是局部索引的最大限制。Oracle 只保证索引分区内部的惟一性，而不能跨分区。这说明什么呢？例如，这意味着不能一方面在一个 TIMESTAMP 字段上执行区间分区，而另一方面在 ID 上有一个主键（使用一个局部分区索引来保证）。Oracle 会利用全局索引来保证惟一性。

在下面的例子中，我们将创建一个区间分区表，它按一个名为 LOAD_TYPE 的列分区，却在 ID 列上有一个主键。为此，可以在一个没有任何其他对象的模式中执行以下 CREATE TABLE 语句，所以通过查看这个用户所拥有的每一个段，就能很容易地看出到底创建了哪些对象：

```
ops$tkyte@ORA10G> CREATE TABLE partitioned
  2     ( load_date date,
  3     id int,
  4     constraint partitioned_pk primary key(id)
  5     )
  6     PARTITION BY RANGE (load_date)
  7     (
  8     PARTITION part_1 VALUES LESS THAN
  9              ( to_date('01/01/2000','dd/mm/yyyy') ) ,
 10          PARTITION part_2 VALUES LESS THAN
 11          ( to_date('01/01/2001','dd/mm/yyyy') )
 12 )
 13 /
Table created.


ops$tkyte@ORA10G> select segment_name, partition_name, segment_type
  2     from user_segments;
```

```
SEGMENT_NAME    PARTITION_NAME     SEGMENT_TYPE
--------------  ---------------    -----------------
PARTITIONED     PART_1             TABLE PARTITION
PARTITIONED     PART_2             TABLE PARTITION
PARTITIONED_PK                     INDEX
```

PARTITIONED_PK 索引甚至没有分区，更不用说脚本分区了。而且我们将会看到，它根本无法进行局部分区。由于认识到非惟一索引也能像惟一索引一样保证主键，我们想以此骗过 Oracle，但是可以看到这种方法也不能奏效：

```
ops$tkyte@ORA10G> CREATE TABLE partitioned
  2     ( timestamp date,
  3     id int
  4     )
  5     PARTITION BY RANGE (timestamp)
  6     (
  7     PARTITION part_1 VALUES LESS THAN
  8             ( to_date('01-jan-2000','dd-mon-yyyy') ) ,
  9     PARTITION part_2 VALUES LESS THAN
 10         ( to_date('01-jan-2001','dd-mon-yyyy') )
 11 )
 12 /
Table created.

ops$tkyte@ORA10G> create index partitioned_idx
```

```
  2     on partitioned(id) local
  3     /
Index created.

ops$tkyte@ORA10G> select segment_name, partition_name, segment_type
  2     from user_segments;

SEGMENT_NAME         PARTITION_NAME    SEGMENT_TYPE
---------------      ---------------    ------------------
PARTITIONED          PART_1             TABLE PARTITION
PARTITIONED_IDX      PART_2             INDEX PARTITION
PARTITIONED          PART_              2 TABLE PARTITION
PARTITIONED_IDX      PART_1             INDEX PARTITION

ops$tkyte@ORA10G> alter table partitioned
  2     add constraint
  3     partitioned_pk
  4     primary key(id)
  5     /
alter table partitioned
*
ERROR at line 1:
ORA-01408: such column list already indexed
```

在此，Oracle 试图在 ID 上创建一个全局索引，却发现办不到，这是因为 ID 上已经存

在一个索引。如果已创建的索引没有分区，前面的语句就能工作，Oracle 会使用这个索引来保证约束。

为什么局部分区索引不能保证惟一性（除非分区键是约束的一部分），原因有两方面。首先，如果 Oracle 允 许如此，就会丧失分区的大多数好处。可用性和可扩缩性都会丧失殆尽，因为对于任何插入和更新，总是要求所有分区都一定可用，而且要扫描每一个分区。你的分 区越多，数据就会变得越不可用。另外，分区越多，要扫描的索引分区就越多，分区也会变得越发不可扩缩。这样做不仅不能提供可用性和可扩缩性，相反，实际上 反倒会削弱可用性和可扩缩性。

另外，倘若局部分区索引能保证惟一性，Oracle 就必须在事务级对这个表的插入和更新有效地串行化。这是因为，如果向 PART_1 增加 ID=1，Oracle 就必须以某种方式防止其他人向 PART_2 增加 ID=1。对此惟一的做法是防止别人修改索引分区 PART_2，因为无法通过对这个分区中的内容“锁定”来做到（找不出什么可以锁定）。

在一个 OLTP 系统中，惟一性约束必须由系统保证（也就是说，由 Oracle 保证），以确保数据的完整性。这意味着，应用的逻辑模型会对物理设计产生影响。惟一性约束能决定底层的表分区机制，影响分区键的选择，或者指示你应该使用全局索引。下面将更深入地介绍全局索引。

13.3.2 全局索引

全局索引使用一种有别于底层表的机制进行分区。表可以按一个 TIMESTAMP 列划分为 10 个分区，而这个表上的一个全局索引可以按 REGION 列划分为 5 个分区。与局部索引不同，全局索引只有一类，这就是前缀全局索引（prefixed global index）。如果全局索引的索引键未从该索引的分区键开始，这是不允许的。这说明，不论用什么属性对索引分区，这些属性都必须是索引键的前几列。

下面继续看前面的例子，这里给出一个使用全局索引的小例子。它显示全局分区索引可以用于保证主键的惟一性，这样一来，即使不包括表的分区键，也可以有能保证惟一性的分区索引。下面的例子创建了一个按 TIMESTAMP 分区的表，它有一个按 ID 分区的索引：

```
ops$tkyte@ORA10G> CREATE TABLE partitioned
  2    ( timestamp date,
  3    id int
  4    )
  5    PARTITION BY RANGE (timestamp)
  6    (
  7          PARTITION part_1 VALUES LESS THAN
  8                ( to_date('01-jan-2000','dd-mon-yyyy') ) ,
```

```
  9          PARTITION part_2 VALUES LESS THAN
 10              ( to_date('01-jan-2001','dd-mon-yyyy') )
 11 )
 12 /

Table created.


ops$tkyte@ORA10G> create index partitioned_index
  2  on partitioned(id)
  3  GLOBAL
  4  partition by range(id)
  5  (
  6          partition part_1 values less than(1000),
  7          partition part_2 values less than (MAXVALUE)
  8  )
  9  /

Index created.
```

注意，这个索引中使用了 MAXVALUE。MAXVALUE 可以不仅可以用于索引中，还可以用于任何区间分区表中。它表示区间的“无限上界”。在此前的所有例子中，我们都使用了区间的硬性上界（小于<某个值>的值）。不过，全局索引有一个需求，即最高分区（最后一个分区）必须有一个值为 MAXVALUE 的分区上界。这可以确保底层表中的所有行都能放在这个索引中。

下面，在这个例子的最后，我们将向表增加主键：

```
ops$tkyte@ORA10G> alter table partitioned add constraint
  2  partitioned_pk
  3  primary key(id)
  4  /
```

```
Table altered.
```

从这个代码还不能明显看出 Oracle 在使用我们创建的索引来保证主键（只有我是“明眼人”，因为我很清楚 Oracle 确实在使用这个索引），所以可以试着删除这个索引来证明这一点：

```
ops$tkyte@ORA10G> drop index partitioned_index;

drop index partitioned_index

*

ERROR at line 1:

ORA-02429: cannot drop index used for enforcement of unique/primary key
```

为了显示 Oracle 不允许创建一个非前缀全局索引，只需执行下面的语句：

```
ops$tkyte@ORA10G> create index partitioned_index2
  2  on partitioned(timestamp,id)
  3  GLOBAL
  4  partition by range(id)
  5  (
  6      partition part_1 values less than(1000),
  7      partition part_2 values less than (MAXVALUE)
  8  )
  9  /
partition by range(id)
                   *

ERROR at line 4:

ORA-14038: GLOBAL partitioned index must be prefixed
```

错误信息相当明确。全局索引必须是前缀索引。那么，要在什么时候使用全局索引呢？我们将分析两种不同类型的系统（数据仓库和 OLTP）。来看看何时可以应用全局索引。

1. 数据仓库和全局索引

原先数据仓库和全局索引是相当互斥的。数据仓库就意味着系统有某些性质，如有大量的数据出入。许多数据仓库都实现了一种滑动窗口（sliding window）方法来管理数据，也就是说，删除表中最旧的分区，并为新加载的数据增加一个新分区。在过去（Oracle8i 及以前的版本），数据仓库系统都避免使用全局索引，对此有一个很好的原因：全局索引缺乏可用性。大多数分区操作（如删除一个旧分区）都会使全局索引无效，除非重建全局索引，否则无法使用，这会严重地影响可用性，以前往往都是如此。

□ 滑动窗口和索引

下面的例子实现了一个经典的数据滑动窗口。在许多实现中，会随着时间的推移向仓库中增加数据，而最旧的数据会老化。在很多时候，这个数据会按一个日期属性进行区间分区，所以最旧的数据多存储在一个分区中，新加载的数据很可能都存储在一个新分区中。每月的加载过程涉及：

□ 去除老数据：最旧的分区要么被删除，要么与一个空表交换（将最旧的分区变为一个表），从而允许对旧数据进行归档。

□ 加载新数据并建立索引：将新数据加载到一个“工作”表中，建立索引并进行验证。

□ 关联新数据：一旦加载并处理了新数据，数据所在的表会与分区表中的一个空分区交换，将表中的这些新加载的数据变成分区表中的一个分区（分区表会变得更大）。

这个过程会没有重复，或者执行加载过程的任何周期重复；可以是每天或每周。我们将在这一节实现这个非常典型的过程，显示全局分区索引的影响，并展示分区操作期间可以用哪些选项来提高可用性，从而能实现一个数据滑动窗口，并维持数据的连续可用性。

在这个例子中，我们将处理每年的数据，并加载 2004 和 2005 财政年度的数据。这个表按 TIMESTAMP 列分区，并创建了两个索引，一个是 ID 列上的局部分区索引，另一个是 TIMESTAMP 列上的全局索引（这里为分区）：

```
ops$tkyte@ORA10G> CREATE TABLE partitioned
  2  ( timestamp date,
  3  id int
  4  )
  5  PARTITION BY RANGE (timestamp)
  6  (
  7        PARTITION fy_2004 VALUES LESS THAN
  8              ( to_date('01-jan-2005','dd-mon-yyyy') ) ,
```

```
  9          PARTITION fy_2005 VALUES LESS THAN
 10              ( to_date('01-jan-2006','dd-mon-yyyy') )
 11 )
 12 /
```

Table created.

```
ops$tkyte@ORA10G> insert into partitioned partition(fy_2004)
  2          select to_date('31-dec-2004',’dd-mon-yyyy’)-mod(rownum,360), object_id
  3          from all_objects
  4  /
```

48514 rows created.

```
ops$tkyte@ORA10G> insert into partitioned partition(fy_2005)
  2          select to_date('31-dec-2005',’dd-mon-yyyy’)-mod(rownum,360), object_id
  3          from all_objects
  4  /
```

48514 rows created.

```
ops$tkyte@ORA10G> create index partitioned_idx_local
  2  on partitioned(id)
  3  LOCAL
  4  /
```

Index created.

```
ops$tkyte@ORA10G> create index partitioned_idx_global
  2    on partitioned(timestamp)
  3    GLOBAL
  4    /
Index created.
```

这就建立了我们的“仓库”表。数据按财政年度分区，而且最后两年的数据在线。这个表有两个索引：一个是 LOCAL 索引，另一个是 GLOBAL 索引。现在正处于年末，我们想做下面的工作：

（1） 删除最旧的财政年度数据。我们不想永远地丢掉这个数据，而只是希望它老化，并将其归档。

（2） 增加最新的财政年度数据。加载、转换、建索引等工作需要一定的时间。我们想做这个工作，但是希望尽可能不影响当前数据的可用性。

第一步是为 2004 财政年度建立一个看上去就像分区表的空表。我们将使用这个表与分区表中的 FY_2004 分区交换，将这个分区转变成一个表，相应地是分区表中的分区为空。这样做的效果就是分区表中最旧的数据（实际上）会在交换之后被删除：

```
ops$tkyte@ORA10G> create table fy_2004 ( timestamp date, id int );
Table created.

ops$tkyte@ORA10G> create index fy_2004_idx on fy_2004(id)
2 /
Index created.
```

对要加载的新数据做同样的工作。我们将创建并加载一个表，其结构就像是现在的分区表（但是它本身并不是分区表）：

```
ops$tkyte@ORA10G> create table fy_2006 ( timestamp date, id int );
Table created.

ops$tkyte@ORA10G> insert into fy_2006
```

```
  2     select to_date('31-dec-2006',’dd-mon-yyyy’)-mod(rownum,360), object_id
  3     from all_objects
  4     /

48521 rows created.



ops$tkyte@ORA10G> create index fy_2006_idx on fy_2006(id) nologging
  2 /

Index created.
```

我们将当前的满分区变成一个空分区，并创建了一个包含 FY_2004 数据的“慢”表。而且，我们完成了使用 FY_2006 数据的所有必要工作，这包括验证数据、进行转换以及准备这些数据所需完成的所有复杂任务。

现在可以使用一个交换分区来更新“活动”数据：

```
ops$tkyte@ORA10G> alter table partitioned
  2     exchange partition fy_2004
  3     with table fy_2004
  4     including indexes
  5     without validation
  6     /

Table altered.



ops$tkyte@ORA10G> alter table partitioned
  2     drop partition fy_2004
  3     /

Table altered.
```

要把旧数据“老化”，所要做的仅此而已。我们将分区变成一个满表，而将空表变成一个分区。这是一个简单的数据字典更新，瞬时就会完成，而不会发生大量的 I/O。现在可以

将 FY_2004 表从数据库中导出（可能要使用一个可移植的表空间）来实现归档。如果需要，还可以很快地重新关联这些数据。

接下来，我们想“滑入”（即增加）新数据：

```
ops$tkyte@ORA10G> alter table partitioned
  2     add partition fy_2006
  3     values less than ( to_date('01-jan-2007','dd-mon-yyyy') )
  4     /
Table altered.


ops$tkyte@ORA10G> alter table partitioned
  2     exchange partition fy_2006
  3     with table fy_2006
  4     including indexes
  5     without validation
  6     /
Table altered.
```

同样，这个工作也会立即完成；这是通过简单的数据字典更新实现的。增加空分区几乎不需要多少时间来处理。然后，将新创建的空分区与满表交换（满表与空分区交换），这个操作也会很快完成。新数据是在线的。

不过，通过查看索引，可以看到下面的结果：

```
ops$tkyte@ORA10G> select index_name, status from user_indexes;
INDEX_NAME                     STATUS
------------------------------ --------
FY_2006_IDX                    VALID
FY_2004_IDX                    VALID
PARTITIONED_IDX_GLOBAL         UNUSABLE
```

当 然，在这个操作之后，全局索引是不可用的。由于每个索引分区可能指向任何表分区，而我们刚才取走了一个分区，并增加了一个分区，所以这个索引已经无效了。 其中有些条目指向我们已经生成的分区，却没有任何条目指向刚增加的分区。使用了这个索引的任何查询可能会失败而无法执行，或者如果我们跳过不可用的索引， 尽管查询能执行，但查询的性能会受到负面影响（因为无法使用这个索引）：

```
ops$tkyte@ORA10G> set autotrace on explain

ops$tkyte@ORA10G> select /*+ index( partitioned PARTITIONED_IDX_GLOBAL ) */
count(*)

  2 from partitioned

  3 where timestamp between sysdate-50 and sysdate;

select /*+ index( partitioned PARTITIONED_IDX_GLOBAL ) */ count(*)

*

ERROR at line 1:

ORA-01502: index 'OPS$TKYTE.PARTITIONED_IDX_GLOBAL' or partition

of such index is in unusable state



ops$tkyte@ORA10G> select count(*)

  2    from partitioned

  3    where timestamp between sysdate-50 and sysdate;



COUNT(*)

----------

6750



  Execution Plan

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

```
0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=59 Card=1 Bytes=9)

1    0          SORT (AGGREGATE)

2    1              FILTER

3    2                  PARTITION RANGE (ITERATOR) (Cost=59 Card=7234
Bytes=65106)

4    3                      TABLE ACCESS (FULL) OF 'PARTITIONED' (TABLE)
(Cost=59 Card=7234

ops$tkyte@ORA10G> set autotrace off
```

因此，执行这个分区操作后，对于全局索引，我们有以下选择：

- 跳过索引，可以像这个例子中一样（Oracle 10g 会透明地这样做），在 9i 中则可以通过设置会话参数 SKIP_UNUSABLE_INDEXES=TRUE 来跳过索引（Oracle 10g 将这个设置默认为 TRUE）。但是这样一来，就丢失了索引所提供的性能提升。
- 让查询接收到一个错误，就像 9i 中一样（SKIP_UNUSABLE_INDEX 设置为 FALSE），在 10g 中，显式地请求使用提示的任何查询都会接收到错误。要想让数据再次真正可用，必须重建这个索引。

到 此为止滑动窗口过程几乎不会带来任何停机时间，但是在我们重建全局索引时，需要相当长的时间才能完成。如果查询依赖于这些索引，在此期间它们的运行时查询 性能就会受到负面影响，可能根本不会运行，也可能运行时得不到索引提供的好处。所有数据都必须扫描，而且要根据数据重建整个索引。如果表的大小为数百 DB，这会占用相当多的资源。

- “活动”全局索引维护

从 Oracle9i 开始，对于分区维护又增加了另一个选项：可以在分区操作期间使用 UPDATE GLOBAL INEXES 子句来维护全局索引。这意味着，在你删除一个分区、分解一个分区以及在分区上执行任何必要的操作时，Oracle 会对全局索引执行必要的修改，保证它是最新的。由于大多数分区操作都会导致全局索引无效，这个特征对于需要提供数据连续访问的系统来说是一个大福音。你会发现，通过牺牲分区操作的速度（但是原先重建索引后会有一个可观的不可用窗口，即不可用的停机时间相当长），可以换取 100%的数据可用性（尽管分区操作的总体响应时间会更慢）。简单地说，如果数据仓库不允许有停机时间，而且必须支持数据的滑入滑出等数据仓库技术，这个特性就再合适不过了，但是你必须了解它带来的影响。

再来看前面的例子，如果分区操作在必要时使用了 UPDATE GLOBAL INDEXES 子句（在这个例子中，在 ADD PARTITION 语句上就没有必要使用这个子句，因为新增加的分区中没有任何行）：

```
ops$tkyte@ORA10G> alter table partitioned

2    exchange partition fy_2004
```

```
  3  with table fy_2004
  4  including indexes
  5  without validation
  6  UPDATE GLOBAL INDEXES
  7  /

Table altered.


ops$tkyte@ORA10G> alter table partitioned
  2  drop partition fy_2004
  3  UPDATE GLOBAL INDEXES
  4  /

Table altered.


ops$tkyte@ORA10G> alter table partitioned
  2  add partition fy_2006
  3  values less than ( to_date('01-jan-2007','dd-mon-yyyy') )
  4 /

Table altered.


ops$tkyte@ORA10G> alter table partitioned
  2  exchange partition fy_2006
  3  with table fy_2006
  4  including indexes
```

```
  5      without validation
  6    UPDATE GLOBAL INDEXES
  7    /

Table altered.
```

就会发现索引完全有效，不论在操作期间还是操作之后这个索引都是可用的：

```
ops$tkyte@ORA10G> select index_name, status from user_indexes;

INDEX_NAME                     STATUS
------------------------------ --------
FY_2006_IDX                    VALID
FY_2004_IDX                    VALID
PARTITIONED_IDX_GLOBAL         VALID
PARTITIONED_IDX_LOCAL          N/A


6 rows selected.


ops$tkyte@ORA10G> set autotrace on explain
ops$tkyte@ORA10G> select count(*)
  2    from partitioned
  3    where timestamp between sysdate-50 and sysdate;


COUNT(*)
----------
6750
```

```
Execution Plan

----------------------------------------------------------

   0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=9 Card=1 Bytes=9)

   1    0          SORT (AGGREGATE)

   2    1              FILTER

   3    2                  INDEX (RANGE SCAN) OF 'PARTITIONED_IDX_GLOBAL'
(INDEX) (Cost=9...
```

但是这里要做一个权衡：我们要在全局索引结构上执行 INSERT 和 DELETE 操作的相应逻辑操作。删除一个分区时，必须删除可能指向该分区的所有全局索引条目。执行表与分区的交换时，必须删除指向原数据的所有全局索引条目，再插入指向刚滑入的数据的新条目。所以 ALTER 命令执行的工作量会大幅增加。

实际上，通过使用 runstats，并对前面的例子稍加修改，就能测量出分区操作期间维护全局索引所执行的“额外”工作量。同前面一样，我们将滑出 FY_2004，并滑入 FY_2006，这就必须加入索引重建。由于需要重建全局索引，因此滑动窗口实现将导致数据变得不可用。然后我们再滑出 FY_2005，并滑入 FY_2007，不过这一次将使用 UPDATE GLOBAL INDEXES 子 句，来模拟提供完全数据可用性的滑动窗口实现。这样一来，即使在分区操作期间，数据也是可用的。采用这种方式，我们就能测量出使用不同技术实现相同操作的 性能，并对它们进行比较。我们期望的结果是，第一种方法占用的数据库资源更少，因此会完成得“更快”，但是会带来显著的“停机时间”。第二种方法尽管会占 用更多的资源，而且总的来说可能需要花费更长的时间才能完成，但是不会带来任何停机时间。对最终用户而言，他们的工作脚步永远不会停止。尽管可能处理起来 会稍微慢一些（因为存在资源竞争），但是他们能一直处理，而且从不停止。

因此，如果用前面的例子，不过另外创建一个类似 FY_2004 的空 FY_2005 表，并创建一个类似 FY_2006 的满 FY_2007 表，这样就可以测量索引重建方法之间有什么差别，先来看“不太可用的方法”：

```
exec runStats_pkg.rs_start;



alter table partitioned exchange partition fy_2004

with table fy_2004 including indexes without validation;
```

```
alter table partitioned drop partition fy_2004;


alter table partitioned add partition fy_2006
values less than ( to_date('01-jan-2007','dd-mon-yyyy') );


alter table partitioned exchange partition fy_2006
with table fy_2006 including indexes without validation;


alter index partitioned_idx_global rebuild;
exec runStats_pkg.rs_middle;
```

下面是可以提供高度可用性的 UPDATE GLOBAL INDEXES 方法：

```
alter table partitioned exchange partition fy_2005
with table fy_2005 including indexes without validation
update global indexes;


alter table partitioned drop partition fy_2005
update global indexes;


alter table partitioned add partition fy_2007
values less than ( to_date('01-jan-2008','dd-mon-yyyy') );


alter table partitioned exchange partition fy_2007
with table fy_2007 including indexes without validation
```

```
update global indexes;

exec runStats_pkg.rs_stop;
```

可以观察到以下结果：

```
ops$tkyte@ORA10G> exec runStats_pkg.rs_stop;

Run1 ran in 81 hsecs

Run2 ran in 94 hsecs

run 1 ran in 86.17% of the time



Name                                  Run1          Run2          Diff

...

STAT...CPU used when call star          39            59            20

...

STAT...redo entries                    938         3,340         2,402

STAT...db block gets                 1,348         5,441         4,093

STAT...session logical reads         2,178         6,455         4,277

...

LATCH.cache buffers chains           5,675        27,695        22,020

...

STAT...table scan rows gotten       97,711       131,427        33,716

STAT...undo change vector size      35,100     3,404,056     3,368,956

STAT...redo size                 2,694,172     6,197,988     3,503,816
```

索引重建方法确实运行得更快一些，从观察到的耗用时间和 CPU 时间可见一斑。正是由于这一点，许多 DBA 都会停下来说：“嘿，我不想用 UPDATE GLOBAL INDEXES，它会更慢。”不过，这就显得目光太短浅了。要记住，尽管操作总的来说会花更长的时间，但是系统上的处理不必再中断。确实，作为 DBA 可能会更长时间地盯着屏幕，但系统上要完成的真正重要的工作确实一直在进行当中。你要看看这个权衡对你来说是否有意义。如果你晚

上有 8 小时的维护窗口来加载新数据，就应该尽可能地使用索引重建方法。不过，如果必须保证连续可用，维护全局索引的能力则至关重要。

查看这种方法生成的 redo 时，可以看到 UPDATE GLOBAL INDEXES 生成的 redo 会多出许多，它是索引创建方法的 230%，而且可以想见，随着为表增加越来越多的全局索引，UPDATE GLOBAL INDEXES 生成的 redo 数量还会进一步增加。UPDATE GLOBAL INDEXES 生成的 redo 是不可避免的，不能通过 NOLOGGING 去掉，因为全局索引的维护不是其结构的完全重建，而应该算是一种增量式“维护”。另外，由于我们维护着活动索引结构，必须为之生成 undo，万一分区操作失败，必须准备好将索引置回到它原来的样子。而且要记住，undo 受 redo 本身的保护，因此你看到的所生成的 redo 中，有些来自索引更新，有些来自回滚。如果增加另一个（或两个）全局索引，可以很自然地想见这些数据量会增加。

所以，UPDATE GLOBAL INDEXES 是一种允许用资源耗费的增加来换取可用性的选项。如果需要提供连续的可用性，这就是一个必要的选择。但是，你必须理解相关的问题，并且适当地确定系统中其他组件的大小。具体地将，许多数据仓库过一段时间都会改为使用大批量的直接路径操作，而绕过 undo 生成，如果允许的话，还会绕过 redo 生成。但是倘若使用 UPDATE GLOBAL INDEXES，就不能绕过 undo 或 redo 生成。在使用这个特性之前，需要检查确定组件大小所用的规则，从而确保这种方法在你的系统上确实能正常工作。

2. OLTP 和全局索引

OLTP 系统的特点是会频繁出现许多小的读写事务，一般来讲，在 OLTP 系统中，首要的是需要快速访问所需的行，而且数据完整性很关键，另外可用性也非常重要。

在 OLTP 系统中，许多情况下全局索引很有意义。表数据可以按一个键（一个列键）分区。不过，你可能需要以多种不同的方式访问数据。例如，可能会按表中的 LOCATION 来划分 EMPLOYEE 数据，但是还需要按以下列快速访问 EMPLOYEE 数据：

- DEPARTMENT：部门的地理位置很分散。部门和位置之间没有任何关系。
- EMPLOYEE_ID：尽管员工 ID 能确定位置，但是你不希望必须按 EMPLOYEE_ID 和 LOCATION 搜索，因为这样一来索引分区上将不能发生分区消除。而且 EMPLOYEE_ID 本身必然是惟一的。
- JOB_TITLE：JOB_TITLE 和 LOCATION 之间没有任何关系。任何 LOCATION 上都可以出现所有 JOB_TITLE 值。

这里需要按多种不同的键来访问应用中不同位置的 EMPLOYEE 数据，而且速度至上。在一个数据仓库中，可以只使用这些键上的局部分区索引，并使用并行索引区间扫描来快速收集大量数据。在这些情况下不必使用索引分区消除。不过，在 OLTP 系统中则不同，确实需要使用分区消除，并发查询对这些系统不合适；我们要适当地提供索引。因此，需要利用某些字段上的全局索引。

我们要满足以下目标：

- 快速访问
- 数据完整性
- 可用性

在一个 OLTP 系统中，可以通过全局索引实现这些目标。我们可能不实现滑动窗口，而且暂时不考虑审计。我们并不分解分区（除非有一个预定的停机时间），也不会移动数据，等等。对于数据仓库中执行的操作，一般来说不会在活动的 OLTP 系统中执行它们。

以下是一个小例子，显示了如何用全局索引来达到以上所列的 3 个目标。这里使用简单的“单分区”全局索引，但是这与多个分区情况下的全局索引也没有不同（只有一点除外，增加索引分区时，可用性和可管理性会提高）。先创建一个表，它按位置 LOC 执行区间分区，根据我们的规则，这会把所有小于‘C’的 LOC 值放在分区 P1 中，小于’D‘的 LOC 值则放在分区 P2 中，依此类推：

```
ops$tkyte@ORA10G> create table emp
  2     (EMPNO NUMBER(4) NOT NULL,
  3     ENAME VARCHAR2(10),
  4     JOB VARCHAR2(9),
  5     MGR NUMBER(4),
  6     HIREDATE DATE,
  7     SAL NUMBER(7,2),
  8     COMM NUMBER(7,2),
  9     DEPTNO NUMBER(2) NOT NULL,
 10 LOC VARCHAR2(13) NOT NULL
 11 )
 12 partition by range(loc)
 13 (
 14 partition p1 values less than('C') tablespace p1,
 15 partition p2 values less than('D') tablespace p2,
 16 partition p3 values less than('N') tablespace p3,
 17 partition p4 values less than('Z') tablespace p4
 18 )
```

```
  19 /

Table created.
```

接下来修改这个表，在主键列上增加一个约束：

```
ops$tkyte@ORA10G> alter table emp add constraint emp_pk
  2     primary key(empno)
  3     /

Table altered.
```

这有一个副作用，EMPNO 列上将有一个惟一索引。由此显示出，完全可以支持和保证数据完整性，这这是我们的目标之一。最后，在 DEPTNO 和 JOB 上创建另外两个全局索引，以便通过这些属性快速地访问记录：

```
ops$tkyte@ORA10G> create index emp_job_idx on emp(job)
  2     GLOBAL
  3     /

Index created.



ops$tkyte@ORA10G> create index emp_dept_idx on emp(deptno)
  2     GLOBAL
  3     /

Index created.



ops$tkyte@ORA10G> insert into emp
  2             select e.*, d.loc
  3             from scott.emp e, scott.dept d
  4             where e.deptno = d.deptno
  5     /
```

```
14 rows created.
```

现在来看每个分区中有什么：

```
ops$tkyte@ORA10G> break on pname skip 1

ops$tkyte@ORA10G> select 'p1' pname, empno, job, loc from emp partition(p1)
  2  union all
  3       select 'p2' pname, empno, job, loc from emp partition(p2)
  4  union all
  5       select 'p3' pname, empno, job, loc from emp partition(p3)
  6  union all
  7       select 'p4' pname, empno, job, loc from emp partition(p4)
  8  /


PN     EMPNO    JOB              LOC
--         ----------    ---------           -------------
p2          7499    SALESMAN      CHICAGO
            7698    MANAGER        CHICAGO
            7654    SALESMAN      CHICAGO
            7900    CLERK           CHICAGO
            7844    SALESMAN      CHICAGO
            7521    SALESMAN      CHICAGO
p3          7369    CLERK           DALLAS
            7876    CLERK           DALLAS
            7902    ANALYST         DALLAS
```

```
             7788    ANALYST         DALLAS

             7566    MANAGER         DALLAS

p4           7782    MANAGER         NEW YORK

             7839    PRESIDENT       NEW YORK

             7934    CLERK           NEW YORK

14 rows selected.
```

这显示了数据按位置在各个分区中的分布。现在可以检查一些查询计划，来查看会有怎样的性能：

```
ops$tkyte@ORA10G> variable x varchar2(30);

ops$tkyte@ORA10G> begin

  2             dbms_stats.set_table_stats

  3             ( user, 'EMP', numrows=>100000, numblks => 10000 );

  4     end;

  5     /

PL/SQL procedure successfully completed.



ops$tkyte@ORA10G> delete from plan_table;

3 rows deleted.



ops$tkyte@ORA10G> explain plan for

  2     select empno, job, loc from emp where empno = :x;

Explained.



ops$tkyte@ORA10G> select * from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT

--------------------------------------------------------------------

|                  Operation                                         |
Name           |Rows      |Bytes  |Pstart     |Pstop     |

--------------------------------------------------------------------

|   SELECT   STATEMENT                              |                 |   1          |
27     |           |           |

|   TABLE   ACCESS   BY   GLOBAL   INDEX   ROWID  |  EMP           |  1          |
27     |ROWID    |ROWID    |

|       INDEX     UNIQUE     SCAN                          |     EMP_PK           |
1          |        |           |           |

--------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("EMPNO"=TO_NUMBER(:X))
```

注意　这里对解释计划格式进行了编辑，使之适合一页的篇幅。报告中与讨论无关的列都被忽略了。

这里的计划显示出对未分区索引 EMP_PK（为支持主键所创建）有一个 INDEX UNIQUE SCAN。然后还有一个 TABLE ACCESS GLOBAL INDEX ROWID，其 PSTART 和 PSTOP 为 ROWID/ROWID，这说明从索引得到 ROWID 时，它会准确地告诉我们读哪个索引分区来得到这一行。这个索引访问与未分区表上的访问同样有效，而且为此会执行同样数量的 I/O。这只是一个简单的单索引惟一扫描，其后是“根据 ROWID 来得到这一行”。现在，我们来看一个全局索引，即 JOB 上的全局索引：

```
ops$tkyte@ORA10G> delete from plan_table;

3 rows deleted.



ops$tkyte@ORA10G> explain plan for
```

```
  2 select empno, job, loc from emp where job = :x;

Explained.



ops$tkyte@ORA10G> select * from table(dbms_xplan.display);



PLAN_TABLE_OUTPUT

--------------------------------------------------------------------------

|
Operation                                 |Name            |Rows       |Bytes   |Pstart
 |Pstop      |

--------------------------------------------------------------------------

|  SELECT  STATEMENT                        |                |  1000      |27000
|           |           |

|  TABLE  ACCESS  BY  GLOBAL  INDEX  ROWID  |EMP             | 1000      |27000
|ROWID    |ROWID    |

|       INDEX     RANGE     SCAN                          |EMP_JOB_IDX              |
400    |           |           |     |

--------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("JOB"=:X)
```

当然，对于 INDEX RANGE SCAN，可以看到类似的结果。在此使用了我们的索引，而且可以对底层数据提供高速的 OLTP 访问。如果索引进行了分区，则必须是前缀索引，并保证索引分区消除；因此，这些索引也是可扩缩的，这说明我们可以对其分区，而且能观察到类似的行为。稍后我们将看到只使用 LOCAL 索引时会发生什么。

最后，下面来看可用性方面。Oracle 文档指出，与局部分区索引相比，全局分区索引更有利于“不那么可用”的数据。我不完全同意这种以一概全的说法。我认为，在 OLTP 系统中，全局分区索引与局部分区索引有着同样的高度可用性。考虑以下例子：

```
ops$tkyte@ORA10G> alter tablespace p1 offline;

Tablespace altered.

ops$tkyte@ORA10G> alter tablespace p2 offline;

Tablespace altered.

ops$tkyte@ORA10G> alter tablespace p3 offline;

Tablespace altered.

ops$tkyte@ORA10G> select empno, job, loc from emp where empno = 7782;

   EMPNO     JOB              LOC

   ----------     ---------          -------------

      7782     MANAGER     NEW YORK
```

在此，即使表中大多数底层数据都不可用，还是可以通过索引访问任何可用的数据。只要我们想要的 EMPNO 在可用的表空间中，而且 GLOBAL 索引可用，就可以利用 GLOBAL 索引来访问数据。另一方面，如果一直使用在前面“高度可用”的局部索引，倒有可能不允许访问数据！这是因为我们在 LOC 上分区而需要按 EMPNO 查询，所以会导致这样一个副作用。我们必须探查每一个局部索引分区，而而遭遇到不可用的索引分区时就会失败。

不过，在这种情况下，其他类型的查询不会（而且不能）工作：

```
ops$tkyte@ORA10G> select empno, job, loc from emp where job = 'CLERK';

select empno, job, loc from emp where job = 'CLERK'

                                *

ERROR at line 1:

ORA-00376: file 13 cannot be read at this time

ORA-01110: data file 13: '/home/ora10g/oradata/.../o1_mf_p2_1dzn8jwp_.dbf'
```

所有分区中都有 CLERK 数据，由于 3 个表空间离线，这一点确实会对我们带来影响。这是不可避免的，除非我们在 JOB 上分区，但是这样一来，就会像按 LOC 分区查询数据一样出现同样的问题。需要由多个不同的“键”来访问数据时就会有这个问题。Oracle 会“尽其所能”地为你提供数据。

不过，要注意，如果可以由索引来回答查询，就要避免 TABLE ACCESS BY ROWID，数据不可用的事实并不重要：

```
ops$tkyte@ORA10G> select count(*) from emp where job = 'CLERK';

  COUNT(*)

----------

         4
```

在这种情况下，由于 Oracle 并不需要表，大多数分区离线的事实也不会影响这个查询。由于 OLTP 系统中这种优化（即只使用索引来回答查询）很常见，所以很多应用都不会因为数据离线而受到影响。现在所要做的只是尽快地让离线数据可用（将其恢复）。

13.4 再论分区和性能

我经常听人说：“我对分区真是失望。我们对最大的表执行了分区，但它变得更慢了。难道分区就是这样吗？它能算一种性能提升特性吗？”

对总体查询性能来说，分区的影响无非有以下三种可能：

- ☐ 使你的查询更快
- ☐ 根本不影响查询的性能
- ☐ 是你的查询更慢，而且与未分区实现相比，会占用多出几倍的资源

在一个数据仓库中，基于数据相关问题的了解，很可能是以上第一种情况。如果查询频繁地全面扫描很大的数据表，通过消除大段的数据，分区能够对这些查询有很好的影响。假设你有一个 100 万行的表，其中有一个时间戳属性。你的查询要从这个表中获取一年的数据（其中有 10 年的数据）。查询使用了一个全表扫描来获取这个数据。如果按时间戳分区，例如每个月一个分区，就可以只对 1/10 的数据进行全面扫描（假设各年的数据是均匀分布的）。通过分区消除，90%的数据都可以不考虑。你的查询往往会运行得更快。

现在，再来看如果 OLTP 系统中有一个类似的表。在这种应用中，你肯定不会获取 100 万行表中 10%的数据，因此，尽管数据仓库中可以得到大幅的速度提升，但这种提升在事务性系统中得不到。不同系统中做的工作是不一样的，所用不可能有同样的改进。因此，一般来说，在 OLTP 系统中达不到第一种情况（不会是查询更快），你不会主要因为提供性能而应用分区。就算是要应用分区，也往往是为了提供可用性以及得到管理上的易用性。但是需要指出，在一个 OLTP 系统中，即使是要确保达到第二点（也就是说，对查询的性能没有影响，而不论是负面影响还是正面影响），也并非轻而易举，而需要付出努力。很多时候，你的目标可能只是应用分区而不影响查询响应时间。

我发现，很多情况下实现团队会看到他们有一个很大的表，例如有 1000 万行。对现在来说，1000 万听上去像是一个难以置信的大数字（在 5 年或 10 年前，这实在是一个很大的数，但是时间可以改变一切）。所以团队决定将数据分区。但是通过查看数据，却发现没有哪个属性可以用于区间分区（RANGE partitioning）。根本没有合适的属性来执行区间分区。同样，列表分区（LIST partitioning）也不可行。这个表中没有什么能作为分区的“依据”。所以，实现团队想对主键执行散列分区，而主键恰好填充为一个 Oracle 序号。看上去很完美，主键是惟一的，而且易于散列，另外很多查询都有以下形式：SELECT * FROM T WHERE PRIMARY_KEY = :X。

但问题是，对这个对象还有另外一些并非这种形式的查询，为了便于说明，假设当前表实际上是 ALL_OBJECTS 字典视图，尽管在内部许多查询的形式都是 WHERE OBJECT_ID = :X，但最终用户还会频繁地对应用发出以下请求：

- 显示 SCOTT 中 EMP 表的详细信息（WHERE OWNER=:0 AND OBJECT_TYPE=:T AND OBJECT_NAME=:N）。
- 显示 SCOTT 所拥有的所有表（WHERE OWNER=:0 AND OBJECT_TYPE=:T）。
- 显示 SCOTT 所拥有的所有对象（WHERE OWNER=:0）.

为了支持这些查询，在（OWNER.OBJECT_TYPE.OBJECT_NAME）上有一个索引。但是你听说“局部索引更可用”，而你希望系统能更可用，所以将索引实现为局部索引。最后将表重建如下，它有 16 个散列分区：

```
ops$tkyte@ORA10G> create table t

  2    ( OWNER, OBJECT_NAME, SUBOBJECT_NAME, OBJECT_ID,
DATA_OBJECT_ID,

  3    OBJECT_TYPE, CREATED, LAST_DDL_TIME, TIMESTAMP, STATUS,

  4    TEMPORARY, GENERATED, SECONDARY )

  5  partition by hash(object_id)

  6  partitions 16

  7  as

  8  select * from all_objects;

Table created.



ops$tkyte@ORA10G> create index t_idx

  2    on t(owner,object_type,object_name)

  3  LOCAL

  4  /

Index created.

ops$tkyte@ORA10G> begin
```

```
  2            dbms_stats.gather_table_stats
  3            ( user, 'T', cascade=>true);
  4      end;
  5 /
PL/SQL procedure successfully completed.
```

接下来执行经典的 OLTP 查询（你知道这些查询会频繁地运行）：

```
variable o varchar2(30)
variable t varchar2(30)
variable n varchar2(30)

exec :o := 'SCOTT'; :t := 'TABLE'; :n := 'EMP';

select *
from t
where owner = :o
        and object_type = :t
        and object_name = :n
/
select *
from t
where owner = :o
        and object_type = :t
/
```

```
select *

from t

where owner = :o

/
```

但是可以注意到，如果 SQL_TRACE=TRUE 成立，运行以上代码时，查看所得到的 TKPROF 报告会有以下性能特征：

```
select * from t where owner = :o and object_type = :t and object_name = :n

call     count       cpu    elapsed       disk      query    current        rows

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

total        4      0.00        0.00           0          34           0           1

  Rows     Row Source Operation

  -------  ---------------------------------------------------

       1   PARTITION HASH ALL PARTITION: 1 16 (cr=34 pr=0 pw=0 time=359 us)

       1   TABLE ACCESS BY LOCAL INDEX ROWID T PARTITION: 1 16 (cr=34 pr=0

       1   INDEX RANGE SCAN T_IDX PARTITION: 1 16 (cr=33 pr=0 pw=0 time=250
```

与未实现分区的同一个表相比较，会发现以下结果：

```
select * from t where owner = :o and object_type = :t and object_name = :n

call     count       cpu    elapsed       disk      query    current        rows

-------  ------  --------  ----------  ----------  ----------  ----------  ----------

total        4      0.00        0.00           0           5           0           1

  Rows     Row Source Operation

  -------  ---------------------------------------------------

       1   TABLE ACCESS BY INDEX ROWID T (cr=5 pr=0 pw=0 time=62 us)

       1   INDEX RANGE SCAN T_IDX (cr=4 pr=0 pw=0 time=63 us)
```

你可能会立即得出（错误的）结论，分区会导致 I/O 次数增加为未分区时的 7 倍：未分

区时只有 5 个查询模式获取，而有分区时却有 34 个。如果你的系统本身就存在一致获取过多的问题（以前是逻辑 I/O），现在情况就会更糟糕。就算是原来没有这个问题，现在也很可能会遭遇到它。对另外两个查询也可能观察到同样的情况。在下面的输出中，整个第一行对应分区表，第二行则对应未分区表：

```
select * from t where owner = :o and object_type = :t

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total         5      0.01        0.01           0          47           0          16
total         5      0.00        0.00           0          16           0          16


select * from t where owner = :o

call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total         5      0.00        0.00           0          51           0          25
total         5      0.00        0.00           0          23           0          25
```

各个查询返回的答案是一样的，但是分别用了 500%、300%或 200%的 I/O，这可不太好。根本原因是什么呢？就在于索引分区机制。注意在前面的计划中，最后一行所列的分区是 1～16.

```
1 PARTITION HASH ALL PARTITION: 1 16 (cr=34 pr=0 pw=0 time=359 us)

1    TABLE ACCESS BY LOCAL INDEX ROWID T PARTITION: 1 16 (cr=34 pr=0

1     INDEX RANGE SCAN T_IDX PARTITION: 1 16 (cr=33 pr=0 pw=0 time=250
```

这个查询必须查看每一个索引分区，因为对应 SCOTT 的条目可以（实际上也很可能）在每一个索引分区中。索引按 OBJECT_ID 执行逻辑散列分区，所以如果查询使用了这个索引，但在谓词中没有引用 OBJECT_ID，所有这样的查询都必须考虑每一个索引分区！

解决方案是对索引执行全局分区。例如，继续看这个 T_IDX 例子，可以选择对索引进行散列分区（这是 Oracle 10g 的新特性）：

注意　索引的散列分区是一个新的 Oracle 10g 特性，这在 Oracle9i 中是没有的。对于散列分区的索引，有关区间扫描还有一些问题需要考虑，这一节后面就会讨论。

```
ops$tkyte@ORA10G> create index t_idx
```

```
  2  on t(owner,object_type,object_name)
  3  global
  4  partition by hash(owner)
  5  partitions 16
  6  /

Index created.
```

与前面分析的散列分区表非常相似，Oracle 会取 OWNER 值，将其散列到 1～16 之间的一个分区，并把索引条目放在其中。现在，再次查看这 3 个查询的 TKPROF 信息：

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        4      0.00       0.00          0          4          0           1
total        5      0.00       0.00          0         19          0          16
total        5      0.01       0.00          0         28          0          25
```

可以看到，与先前未分区表所执行的工作很相近，也就是说，我们不会负面地影响查询执行的工作。不过，需要指出，散列分区索引无法执行区间扫描。一般来说，它最适于完全相等性比较（是否相等或是否在列表中）。如果想使用前面的索引来查询 WHERE OWNER > :X，就无法使用分区消除来执行一个简单的区间扫描，你必须退回去检查全部的 16 个散列分区。

这是不是说分区对 OLTP 性能完全没有正面影响呢？不完全如此，要换个场合来看。一般来讲，对于 OLTP 中的数据获取，分区确实没有正面的影响；相反，我们还必须小心地保证数据获取不要受到负面的影响。但是对于高度并发环境中的数据修改，分区则可能提供显著的好处。

考虑一个相当简单的例子，有一个表，而且只有一个索引，在这个表中再增加一个主键。如果没有分区，实际上这里只有一个表：所有插入都会插入到这个表中。对这个表的 freelist 可能存在竞争。另外，OBJECT_ID 列上的主键索引是一个相当“重”的右侧索引，如第 11 章所讨论的那样。假设主键列由一个序列来填充；因此，所有插入都会放到最右边的块中，这就会导致缓冲区等待。另外只有一个要竞争的索引结构 T_IDX。目前看来，“单个”的项目太多了（只有一个表，一个索引等）。

再来看分区的情况。按 OBJECT_ID 将表散列分区为 16 个分区。现在就会竞争 16 个“表”，而且只会有 1/16 个“右侧”，每个索引结构只会接收以前 1/16 的工作负载，等等。也就是说，在一个高度并发环境中可以使用分区来减少竞争，这与第 11 章使用反向键索引减少缓冲区忙等待是一样的。不过必须注意，与没有分区相比，数据的分区处理本身会占用更多的 CPU 时间。也就是说，如果没有分区，数只有一个去处，但有了分区后，则需要用

更多的 CPU 时间来查明要把数据放在哪里。

因此，与以往一样，对系统应用分区来“提供性能”之前，先要确保自己真正了解系统需要什么。如果系统目前是 CPU 密集的（占用大量 CPU 时间），但是 CPU 的使用并不是因为竞争和闩等待，那么引入分区并不能使问题好转，而只会让情况变得更糟糕！

使用 ORDER BY

这 个例子引出一个关系不大但很重要的事实。查看散列分区索引时，可以发现另一个情况：使用索引来获取数据时，并不会自动地获取有序的数据。有些人认为，如果 查询计划显示使用了一个索引来获取数据，那么获取的数据就会是有序的。并不是这样的。要以某种有序顺序来获取数据，惟一的办法就是在查询上使用 ORDER BY。如果查询不包含 ORDER BY 语句，就不能对数据的有序顺序做任何假设。

可以用一个小例子来说明。我们创建了一个小表（ALL_USERS 的一个副本），并创建一个散列分区索引，在 USER_ID 列上有 4 个分区：

```
ops$tkyte@ORA10G> create table t
  2    as
  3    select *
  4    from all_users
  5    /
  Table created.



ops$tkyte@ORA10G> create index t_idx
  2    on t(user_id)
  3    global
  4    partition by hash(user_id)
  5    partitions 4
  6    /
Index created.
```

现在，我们要查询这个表，这里使用了一个提示，要求 Oracle 使用这个索引。注意数据的顺序（实际上，可以注意到数据是无序的）：

```
ops$tkyte@ORA10G> set autotrace on explain

ops$tkyte@ORA10G> select /*+ index( t t_idx ) */ user_id

  2    from t

  3    where user_id > 0

  4    /

USER_ID

----------

11

34

...

81

157

19

22

...

139

161

5

23

...

163

167
```

```
35

37

...

75

160

38 rows selected.

Execution Plan

----------------------------------------------------------

0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=38 Bytes=494)

1    0     PARTITION HASH (ALL) (Cost=4 Card=38 Bytes=494)

2    1       INDEX (RANGE SCAN) OF 'T_IDX' (INDEX) (Cost=4 Card=38
Bytes=494)

ops$tkyte@ORA10G> set autotrace off
```

所以，即使 Oracle 在区间扫描中使用了索引，数据显然也不是有序的。实际上，可以观察到这个数据存在一个模式。这里有“4 个有序”的结果：……替代的都是按值递增的数据：在 USER_ID=34 和 81 的行之间，输出的值是递增的。然后出现了一个 USER_ID=19 行。我们观察到的结果是：Oracle 会从 4 个散列分区一个接一个地返回“有序的数据”。

在此只是一个警告：除非你的查询中有一个 ORDER BY，否则不要指望返回的数据会按某种顺序排序。（另外，GROUP BY 也不会执行排序！ORDER BY 是无可替代的）。

13.5 审计和段空间压缩

就在不久之前，还没有诸如HIPPA法案（http://www.hhs.gov/ocr/hippa）所强制的那些美国政府限制。像安然这样的一些公司仍在运营，美国政府尚不要求符合Sarbanes-Oxley法案。那时，总认为审计是“以后哪一天可能要做的事情”。不过，如今审计已经摆到了最前沿的位置，许多DBA都要为其财政、商业和卫生保健数据库保留审计跟踪信息，需要保证长达 7 年的审计跟踪信息在线。

在 数据库中可能只是插入审计跟踪信息，而这部分数据在正常操作期间从不获取。审计跟踪信息主要作为一种证据，这是一种事后证据。这些证据是必要的，但是从很 多方面来讲，这些数据只是放在磁盘上，占用着空间，而且所占的空间相当大。然后必须每个月或每年（或者每隔一段时间）对其净化或归档。如果审计从一开始就 设计不当，最后很可能

置你于“死地”。从现在算起，如果 7 年后需要第一次对旧数据进行净化或归档时你才开始考虑如何来完成这一工作，那就太迟了。除非你做了适当的设计，否则取出旧信息实在是件痛苦的事情。

下面来看两种技术：分区和段空间压缩（见第 10 章 。 利用这些技术，审计不仅是可以忍受的，而且很容易管理，并且将占用更少的空间。第二个技术可能不那么明显，因为段空间压缩只适用于诸如直接路径加载之类的 大批量操作，而审计跟踪通常一次只插入一行，也就是事件发生时才插入。这里的技巧是要将滑动窗口分区与段空间压缩结合起来。

假 设我们决定按月对审计跟踪信息分区。在第一个业务月中，我们只是向分区表中插入信息；这些插入使用的是“传统路径”，而不是直接路径，因此没有压缩。在这 个月结束之前，现在我们要向表中增加一个新的分区，以容纳下个月的审计活动。下个月开始后不久，我们会对上个月的审计跟踪信息执行一个大批量操作，具体来 讲，我们将使用 ALTER TABLE 命令来移动上个月的分区，这还有压缩数据的作用。实际上，如 果再进一步，可以将这个分区从一个可读写表空间（现在它必然在一个可读写表空间中）移动到一个通常只读的表空间中（其中包含对应这个审计跟踪信息的其他分 区）。采用这种方式，就可以一个月备份一次表空间（将分区移动到这个表空间之后才备份）；这就能确保有一个正确、干净的当前表空间只读副本；然后在这个月 不再对其备份。审计跟踪信息可以有以下表空间：

- 一个当前在线的读写表空间，它会像系统中每一个其他的正常表空间一样得到备份。这个表空间中的审计跟踪信息不会被压缩，我们只是向其中插入信息。
- 一个只读表空间，其中包含“当前这一年”的审计跟踪信息分区，在此采用一种压缩格式。在每个月的月初，置这个表空间为可读写，向这个表空间中移入上个月的审计信息，并进行压缩，再使之成为只读表空间，并完成备份。
- 用于去年、前年等的一系列表空间。这些都是只读表空间，甚至可以放在很慢的廉价存储介质上。如果出现介质故障，我们只需要从备份恢复。有时可以随机地从备份集中选择每一年的信息，确保这些信息是可恢复的（有时磁带会出故障）。

采 用这种方式，就能很容易地完成净化（即删除一个分区）。同样，归档也很轻松，只需先传送一个表空间，以后再恢复。通过实现压缩可以减少空间的占用。备份的 工作量会减少，因为在许多系统中，单个最大的数据集就是审计跟踪数据。如果可以从每天的备份中去掉某些或全部审计跟踪数据，可能会带来显著的差别。

简单地说，审计跟踪需求和分区这两个方面是紧密相关的，而不论底层系统是何种类型（数据仓库或是 OLTP 系统）。

13.6 小结

对于扩展数据库中的对象来说，分区极其有用。这种扩展体现在以下几个方面：性能扩展、可用性扩展和管理扩展。对于不同的人，所有这 3 个方面都相当重要。DBA 关心的是管理扩展。系统所有者关系可用性，因为停机时间就意味着金钱损失，只有能减少停机时间，或者能减少停机时间带来的影响，就能增加系统的回报。系统的最终用户则关心性能扩展。毕竟，没有哪个人喜欢用慢系统。

我们还了解到这样一个事实：在一个 OLTP 系统中，分区可能不能提高性能，特别是如果应用不当，甚至会使性能下降。分区可能会提高某些类型查询的性能，但是这些查询通常不在 OLTP 系统中使用。这一点很重要，一定要了解，因为许多人总是想当然地把分区和

“性能提升“联系在一起。这并不是说不应该在 OLTP 系统中应用分区；实际上，在 OLTP 环境中，分区确实能提供许多其他的优点，只是不要指望分区能带来吞吐量的大幅提升。分区能减少停机时间，可能会得到同样好的性能（如果适用适当，分区不会使速度减慢）。应用分区后，管理可能更为容易，这可能会带来性能提升，因为能由 DBA 执行的一些维护操作会更多地由他们完成。

我们分析了 Oracle 提 供的各种表分区机制，包括区间分区、散列分区、列表分区和组合分区，并讨论了何时使用这些分区机制最合适。然后用大量篇幅来介绍分区索引，并研究了前缀索 引和非前缀索引以及局部索引和全局索引之间的差别。在此分析了数据仓库中结合全局索引的分区操作，并讨论了资源消耗和可用性之间的折中。

我认为，对越来越多的人来说，随着时间的推移，这个特性将显得更重要，因为数据库应用的大小和规模都在增长。随着 Internet 的发展，由于它广泛需要数据库的支持，再加上法律方面的原因，因此需要更长时间地保留审计数据，这些都导致出现了越来越多极大的数据集合，对此，分区是一个很自然的工具，可以帮助管理这个问题。

第 14 章 并行执行

并行执行（parallel execution）是 Oracle 企业版才有的特性（标准版中没有这个特性），最早于 1994 年在 Oracle 7.1.6 中引入。所谓并行执行，是指能够将一个大型串行任务（任何 DML，或者一般的 DDL）物理地划分为多个较小的部分，这些较小的部分可以同时得到处理。Oracle 中 的并行执行正是模拟了我们在实际生活中经常见到的并发处理。你可能很少看到一个人单枪匹马地盖房子；更常见的是由多个团队并发地工作，迅速地建成房屋。采 用这种方式，某些操作可以划分为较小的任务，从而并发地执行。例如，铺设管线和电路配线就可以同时进行，以减少整个工作所需的总时间。

Oracle 中的并行执行遵循了几乎相同的逻辑。Oracle 通常可以将某个大“任务”划分为较小的部分，并且并发地执行各个部分。例如，如果需要对一个大表执行全表扫描，那么 Oracle 完全可以建立 4 个并行会话（P001~P004）来一起执行完全扫描，每个会话分别读取表中一个不同的部分（既然这样能更快地完成任务，没有理由不这样做）。如果需要存储 P001~P004 扫描的数据，这个工作可以再由另外 4 个并行会话（P005~P008）来执行，最后它们可以将结果发送给这个查询的总体协调会话。

并行执行作为一个工具，如果使用得当，可以使某些操作的响应时间大幅改善，使速度呈数量级增长。但是，如果把并行执行当作一个“fast=true”型的提速开关，结果往往适得其反。在这一章中，我们并不打算明确地解释 Oracle 中如何实现并行查询，也不会介绍由并行操作可能得到的多种计划组合，在此不会讨论诸如此类的内容。我认为，这些内容已经在各种 Oracle 手册（Oracle Administrator’s Guide、Oracle Concepts Guide，特别是 Oracle Data Warehousing Guide）中得到了很好的说明。这一章的目标只是让你对并行执行适用于哪些类型的问题（以及对哪些类型的问题不适用）有所了解。具体来讲，我们会介绍何时使用并行执行，在此之后，还将介绍下面的内容：

- 并行查询：这是指能使用多个操作系统进程或线程来执行一个查询。Oracle 会发现能并行执行的操作（如全表扫描或大规模排序），并创建一个查询计划来实现）。
- 并行 DML（PDML）：这在本质上与并行查询很相似，但是 PDML 主要是使用并行处理来执行修改（INSERT、UPDATE、DELETE 和 MERGE）。在这一章中，我们将介绍 PDML，并讨论 PDML 所固有的一些限制。
- 并行 DDL：并行 DDL 是指 Oracle 能并行地执行大规模的 DDL 操作。例如，索引重建、创建一个新索引、数据加载以及大表的重组等都可以使用并行处理。在我看来，这正是数据库中并行化的“闪光之处”，所以我们会用很多的篇幅重点讨论这方面内容。
- 并行恢复：这是指数据库能并行地执行实例（甚至介质）恢复，以减少从故障恢复所需的时间。
- 过程并行化：这是指能并行地运行所开发的代码。在这一章中，我会讨论两种过程并行化方法。第一种方法中，Oracle 以一种对开发人员透明的方式并行地运行开发的 PL/SQL 代码（开发人员并不开放代码；而是由 Oracle 透明地为其并行执行代码）。此外还有另一种方法，我把它成为“DIY 并行化”（do-it-yourself parallelism），即开发的代码本身被设计为要并行地执行。

14.1 何时使用并行执行

并行执行可能很神奇。一个过程原本需要执行数小时或者数天，通过使用并行执行，可能几分钟内就可以完成。将一个大问题分解为小部分，这在有些情况下可以显著地减少处理时间。不过，考虑并行执行时，要记住一个基本概念，这可能很有用。下面这句话选自Jonathan Lewis 所著的 Practical Oracle8i: Building Efficient Databases(Addison-Wesley, 2001)，尽管很简短，但它对这个概念做了很好的总结：

并行查询（PARALLEL QUERY）选项本质上是不可扩缩的。

并行执行本质上是一个不可扩缩的解决方案，设计为允许单个用户或每个特定 SQL 语句占用数据库的所有资源。如果某个特性允许一个人使用所有可用的资源，倘若再允许两个人使用这个特性，就会遇到明显的竞争问题。随着系统上并发用户数的增加，要应付这些并发用户，系统上的资源（内存、CPU 和 I/O）开始表现出有些勉为其难，此时能不能部署并行操作就有疑问了。例如，如果你有一台 4CPU 的主机，平均有 32 个用户同时执行查询，那你很可能不希望并行执行他们的操作。如果允许每个用户执行一个“并行度为 2”的查询，这就会在一台仅有 4 个 CPU 的主机上发生 64 个并发操作。即使并行执行之前这台机器原本不算疲于奔命，现在也很可能难逃此劫。

简单地说，并行执行也可能是一个很糟糕的想法。在许多情况下，应用并行处理后，可能只会带来资源占用的增加，因为并行执行试图使用所有可用的资源。在一个系统（如一个 OLTP 系统）中，如果资源必须由多个并发事务所共享，你就会观察到，并行执行可能导致响应时间增加。Oracle 会避开某些在串行执行计划中能高效使用的执行技术，而采用诸如全面扫描等执行路径，希望能把较大的批量操作划分为多个部分，并且并行地完成这些部分，从而能优于串行计划。但是如果并行执行应用不当，不仅不能解决性能问题，反而会成为性能问题的根源。

所以，在应用并行执行之前，需要保证以下两点成立：

- 必须有一个非常大的任务，如对 50GB 数据进行全面扫描。
- 必须有足够的可用资源。在并行全面扫描 50GB 数据之前，你要确保有足够的空闲 CPU（以容纳并行进程），还要有足够的 I/O 通道。50GB 数据可能分布在多个物理磁盘上（而不只是一个物理磁盘），以允许多个并发读请求能同时发生，从磁盘到计算机应当有足够多的 I/O 通道，以便能并行地从磁盘获取数据，等等。

如果只有一个小任务（通常 OLTP 系统中执行的查询就是这种典型的小任务），或者你的可用资源不足（这也是 OLTP 系统中很典型的情况），其中 CPU 和 I/O 资源通常已经得到最大限度的使用，那就根本不用考虑并行执行。

关于并行处理的类比

我经常使用一个类比来描述并行处理，解释为什么需要有一个大任务，还要求数据库中有足够多的空闲资源。这个类比是写一个有 10 章的详尽报告，每一章之间都要相互独立。例如，可以考虑你手上的这本书。本章与第 9 章是独立的，不必按先后顺序写这两章。

如何完成这两个任务？你认为并行处理对哪个任务有好处？

1. 一页的总结

在这个类比中，第一个任务是要完成只有一页的总结，这并不是一个大任务。你可以自己来写，也可以把这个任务安排给某个人。为什么呢？因为倘若将这个处理“并行化”，所需的工作量反而会超过你自己写这一页所需的工作。如果要实现“并行化”，你可能必须坐下来，想清楚应该有 12 段，并确定每一段不能依赖于其他段落，然后召开一个小组会议，选择 12 个 人，向他们解释问题是什么，并安排每个人完成一段，然后作为一个协调员收集所有段落，按正确的顺序摆好，验证这些段落无误，然后打印报告。与你自己串行地 写这一页相比，上述过程很可能需要花费更长的时间。对于这样一个小规模的项目，管理这么一大群人的开销远远超过了并行编写 12 段所能得到的收益。

数据库中的并行执行也是同样的道理。如果一个任务只需要几秒（或更短时间）就能串行地完成，引入并行执行后，相关的管理开销可能会让整个过程花费更长的时间。

2. 10 章的报告

现在再来分析第二个任务。你希望你尽快地写出这个 10 章的报告，对此最慢的办法就是将所有工作都安排给某一个人（相信我说的话，因为我很清楚这一点，看看这本书就知道了！有时我真是希望能有 15 个“我”在同时写）。此时，你要召开会议，审查处理步骤，分配工作，然后作为协调员收集结果，完成最后的报告并提交。这个过程可能不是在 1/10 的时间内完成，而可能在 1/8 左右的时间内完成。同样，这么说有一个附加条件，你要有足够的空闲资源。如果你的员工很多，而且目前手头都没有什么具体工作，那么划分这个工作绝对是有意义的。

不 过，假设你是经理，请考虑以下情况：你的员工可能手头有很多任务。在这种情况下，就必须谨慎对待这个大项目。你要保证不要让你的员工疲于奔命；你不希望他 们过度疲劳，超过承受极限。如果你的资源（你的员工）无力处理，你就不能再分配更多的工作，否则他们肯定会辞职。如果你的员工已经是满负荷的，增加更多的 工作只会导致所有进度都落空，所有项目都延迟。

Oracle 中的并行执行也是一样。如果一个任务要花几分钟、几小时或者几天的时间执行，引入并行执行可能会使这个任务的运行快上 8 倍。但是，重申一遍，如果资源已经被过度使用（员工已经超负荷工作），就要避免引入并行执行，因为系统可能会变得更迟钝。尽管 Oracle 服务器进程不会递交“辞职书”，但它们可能会用完所有的 RAM 并失败，或者只会长时间地等待 I/O 或 CPU，看上去就好像没有做任何工作一样。

如 果牢记这个类比（但要记住不能不合理地过分夸大类比），对于并行化是否有用，你就有了一个常识性的指导原则。如果有一个任务需要花几秒的时间完成，要想通 过使用并行执行让它更快一些，就很可能成问题，而且通常可能适得其反。如果已经在过度使用资源（也就是说，你的资源已经得到充分利用），再增加并行执行很 可能会使情况更糟，而不是更好。如果有一个相当大的任务，而且有充分的额外资源，并行执行就再合适不过了。在这一章中，我们将介绍充分利用执行资源的一些 方法。

14.2 并行查询

并行查询允许将一个 SQL SELECT 语句划分为多个较小的查询，每个部分的查询并发地运行，然后会将各个部分的结果组合起来，提供最终的答案。例如，考虑以下查询：

```
big_table@ORA10G> select count(status) from big_table;
```

通过并行查询，这个查询可以使用多个并行会话；将 BIG_TABLE 划分为较小的不重

叠的部分（片）；要求各个并行会话读取表并统计它那一部分中的行数。这个会话的并行查询协调器再从各个并行会话接收各个统计结果，进一步汇总这些结果，将最后的答案返回给客户应用。如果用图形来表示，这个过程如图 14-1 所示。

P000、P001、P002 和 P003 进程称为并行执行服务器（parallel execution server），有时也称为并行查询（parallel query，PQ）从属进程。各个并行执行服务器都是单独的会话，就像是专业服务器进程一样连接数据库。每个并行执行服务器分别负责扫描 BIG_TABLE 中一个部分（各个部分都不重叠），汇总其结果子集，将其输出发回给协调服务器（即原始会话的服务器进程），它再将这些子结果汇总为最终答案。

图 14-1　并行 select count(status)示意图

可以通过一个解释计划来查看这个查询。使用一个有 10,000,000 行数据的 BIG_TABLE，我们将逐步启用这个表的一个并行查询，来了解如何“看到”实际的并行查询。这个例子在一个 4CPU 主机上执行，所有并行参数都取默认值；也就是说，这是一个初始安装，只设置了必要的参数，包括 SGA_TARGET（设置为 1GB）、CONTROL_FILES、DB_BLOCK_SIZE（设置为 8KB）和 PGA_AGGREGATE_TARGET（设置为 512MB）。起初，我们可能会看到以下计划：

```
big_table@ORA10GR1> explain plan for

  2 select count(status) from big_table;

Explained.
```

```
big_table@ORA10GR1> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT

---------------------------------------

Plan hash value: 1287793122

-----------------------------------------------------------------------------------------------

| Id  | Operation                    | Name           | Rows  |Bytes | Cost (%CPU)   | Time       |

-----------------------------------------------------------------------------------------------------------------
--------------

|   0 | SELECT STATEMENT             |                |     1 |   17 |  32390
(2) |00:06:29    |

|   1 |  SORT AGGREGATE              |                |     1 |   17 |                |            |

|   2 |   TABLE ACCESS FULL          | BIG_TABLE      |    10M |  162M |  32390
(2) |00:06:29    |

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

这是一个典型的串行计划。这里不涉及并行化，因为我们没有请求启用并行查询，而默认情况下并不启用并行查询。

启用并行查询有多种方法，可以直接在查询中使用一个提示，或者修改表，要求考虑并行执行路径（在这里，我们将会使用后一种做法）。

可以具体指定这个表的执行路径中要考虑的并行度。例如，可以告诉 Oracle：“我们希望你在创建这个表的执行计划时使用并行度 4”：

```
big_table@ORA10GR1> alter table big_table parallel 4;

Table altered.
```

但我更喜欢这样告诉 Oracle：“请 考虑并行执行，但是你要根据当前的系统工作负载和查询本身来确定适当的并行度”。也就是说，并行度要随着系统上工作负载的增减而变化。如果有充足的空闲资 源，并行度会上升；如果可用资源有限，并行度则会下降。这样就不会为机器强加一个固定的并行度。利用这种方法，允许 Oracle 动态地增加或减少查询所需的并发资源量。

因此，我们只是通过以下 ALTER TABLE 命令来启用对这个表的并行查询：

```
big_table@ORA10GR1> alter table big_table parallel;
```

```
Table altered.
```

仅此而已。现在，对这个表的操作就会考虑并行查询。重新运行解释计划，可能看到以下结果：

```
big_table@ORA10GR1> explain plan for

  2   select count(status) from big_table;

Explained.



big_table@ORA10GR1> select * from table(dbms_xplan.display);



PLAN_TABLE_OUTPUT

----------------------------------------

Plan hash value: 1651916128

----------------------------------------------------------------------------

|Id | Operation | Name |Cost(%CPU) | TQ |IN-OUT |PQ Distrib |

----------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 4465 (1) | | | | |

| 1 | SORT AGGREGATE | | | | | | |

| 2 | PX COORDINATOR | | | | | | |

| 3 | PX SEND QC (RANDOM) | :TQ10000 | |Q1,00 | P->S |QC (RAND) |

| 4 | SORT AGGREGATE | | |Q1,00 | PCWP | |

| 5 | PX BLOCK ITERATOR | | 4465 (1) |Q1,00 |
```

```
PCWC    |                |

| 6   | TABLE ACCESS FULL      | BIG_TABLE   | 4465  (1)      |Q1,00  |
PCWP    |                |

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

注意　我们从这个计划输出中去掉了 ROWS、BYTES 和 TIME 这几列，以便能在一页篇幅内放下。不过，查询的总时间是 00:00:54，而不是先前对串行计划估计的 00:06:29。要记住，这些只是估计，不能保证肯定如此！另外，这是 Oracle 10g 的计划输出，Oracle9i 的计划输出提供的细节更少（只有 4 步，而不是 7 步），但是最终结果是一样的。

如果自下而上地读这个计划（从 ID=6 开始），它显示了图 14-1 中所示的步骤。全表扫描会分解为多个较小的扫描（第 5 步）。每个扫描汇总其 COUNT(STATUS)值（第 4 步 。这些子结果将传送给并行查询协调器（第 2 步和第 3 步），它会进一步汇总这些结果（第 1 步 ，并输出答案。

如果在一个新启动的系统上执行这个查询（这个系统还没有执行其他任何并行执行），可以观察到以下结果。在此使用 Linux ps 命令来找出并行查询进程（我们希望找不出这样的进程，因为系统开始时没有执行任何并行执行），启用并行执行后再运行这个查询，然后再次查找出并行查询进程：

```
big_table@ORA10GR1> host ps -auxww | grep '^ora10gr1.*ora_p00._ora10g'

big_table@ORA10GR1> select count(status) from big_table;



COUNT(STATUS)

-------------

10000000



big_table@ORA10GR1> host ps -auxww | grep '^ora10gr1.*ora_p00._ora10g'

ora10gr1 3411 35.5 0.5 1129068 12200 ? S 13:27 0:02 ora_p000_ora10gr1

ora10gr1 3413 28.0 0.5 1129064 12196 ? S 13:27 0:01 ora_p001_ora10gr1

ora10gr1 3415 26.0 0.5 1129064 12196 ? S 13:27 0:01 ora_p002_ora10gr1

ora10gr1 3417 23.3 0.5 1129044 12212 ? S 13:27 0:01 ora_p003_ora10gr1
```

```
ora10gr1 3419 19.5 0.5 1129040 12228 ? S 13:27 0:01 ora_p004_ora10gr1

ora10gr1 3421 19.1 0.5 1129056 12188 ? S 13:27 0:01 ora_p005_ora10gr1

ora10gr1 3423 19.0 0.5 1129056 12164 ? S 13:27 0:01 ora_p006_ora10gr1

ora10gr1 3425 21.6 0.5 1129048 12204 ? S 13:27 0:01 ora_p007_ora10gr1
```

可能看到，Oracle 现在启动了 8 个并行执行服务器。如果你很好奇，想“看看”并行查询，使用两个会话就能很容易地满足你的好奇心。在运行并行查询的会话中，我们先来确定这个会话的 SID：

```
big_table@ORA10GR1> select sid from v$mystat where rownum = 1;

       SID

----------

       162
```

在另一个会话中，准备好要运行的查询：

```
ops$tkyte@ORA10GR1> select sid, qcsid, server#, degree

  2      from v$px_session

  3      where qcsid = 162
```

在 SID=162 的会话中开始并行查询之后不久，返回到第二个会话，运行这个查询：

```
  4     /

       SID QCSID     SERVER# DEGREE

---------- ----------          ----------       ----------

       145         162                 1                 8

       150         162                 2                 8

       147         162                 3                 8

       151         162                 4                 8

       146         162                 5                 8

       152         162                 6                 8
```

```
       143        162          7          8
       144        162          8          8
       162        162

9 rows selected.
```

在此可以看到，动态性能视图中的这 9 行（对应 9 个会话）中，并行查询会话（SID=162）的 SID 就是这 9 行的查询协调器 SID（query coordinator SID, QCSID）。现在，这个会话“正在协调”或控制着这些并行查询资源。可以看到每个会话都有自己的 SID；实际上，每个会话都是一个单独的 Oracle 会话，执行并行查询时从 V$SESSION 中也可以看出这一点：

```
ops$tkyte@ORA10GR1> select sid, username, program
  2    from v$session
  3    where sid in ( select sid
  4                   from v$px_session
  5                   where qcsid = 162 )
  6    /

       SID    USERNAME                       PROGRAM
---------- ------------------------------ ------------------------------
       143    BIG_TABLE                      oracle@dellpe (P005)
       144    BIG_TABLE                      oracle@dellpe (P002)
       145    BIG_TABLE                      oracle@dellpe (P006)
       146    BIG_TABLE                      oracle@dellpe (P004)
       147    BIG_TABLE                      oracle@dellpe (P003)
       150    BIG_TABLE                      oracle@dellpe (P001)
       151    BIG_TABLE                      oracle@dellpe (P000)
       153    BIG_TABLE                      oracle@dellpe (P007)
```

```
       162 BIG_TABLE                 sqlplus@dellpe (TNS V1-V3)

9 rows selected.
```

注意 如果你的系统中没有出现并行执行，就不要指望 V$SESSION 中会显示并行查询服务器。它们会在 V$PROCESS 中，但是除非真正使用，否则不会建立会话。并行执行服务器会连接到数据库，但是不会建立一个会话。关于会话和连接的区别，有关详细内容请参见第 5 章。

简而言之，并行查询就是这样工作的（实际上，一般的并行执行就以这种方式工作）。它要求一系列并行执行服务器同心协力地工作，生成子结果，这些子结果可以传送给其他并行执行服务器做进一步的处理，也可以传送给并行查询的协调器。

在这个特定的例子中，如所示的那样，BIG_TABLE 分布在一个表空间的 4 个不同的设备上（这个表空间有 4 个数据文件）。实现并行执行时，通常“最好”将数据尽可能地分布在多个物理设备上。可以通过多种途径做到这一点：

- 跨磁盘使用 RAID 条纹（RAID striping）；
- 使用 ASM（利用其内置条纹）；
- 使用分区将 BIG_TABLE 物理地隔离到多个磁盘上；
- 使用一个表空间中的多个数据文件，第二允许 Oracle 在多个文件中为 BIG_TABLE 段分配区段。

一般而言，如果能访问尽可能多的资源（CPU、内存和 I/O），并行执行就能最好地发挥作用。但这并不是说：如果整个数据集都在一个磁盘上，就从并行查询得不到任何好处，并非如此；不过如果整个数据集都在一个磁盘上，可能确实不如使用多个磁盘那样能有更多收获。即使使用一个磁盘，在响应时间上也可能可以得到一定的速度提升，原因在于：给定的一个并行执行服务器在统计行 时并不读取这些行，反之亦然。所以，与执行串行相比，两个并行执行服务器可以在更短的时间内完成所有行的统计。

类似地，即使在一台单 CPU 主机上，甚至也能得益于并行查询。一个串行 SELECT COUNT(*)不太可能 100%地完全占用单 CPU 主机上的 CPU，其部分时间会用于执行（和等待）磁盘的物理 I/O。通过并行查询，你就能充分利用主机上的资源（在这里就是 CPU 和 I/O），而不论是什么资源。

最后一点又把我们带回到先前引用的 Practical Oracle8i: Building Efficient Databases 中的那句话：并行查询本质上是不可扩缩的。如果在这台单 CPU 主机上利用两个并行执行服务器允许四个会话同时执行查询，你可能会发现，与串行处理相比，响应时间反而更长了。需要一个稀有资源的进程越多，满足所有请求所需的时间也越长。

而且要记住，并行查询需要保证两个前提。首先，你需要执行一个大任务，例如，一个长时间运行的查询，这个查询的运行时间要分钟、小时或天为单位来度量，而不是秒或次秒。这说明，并行查询不能作为典型 OLTP 系统的解决方案，因为在 OLTP 系统中，你不会执行长时间运行的任务。在这些系统上启用执行通常是灾难性的。其次，你需要有充足的空闲资源，如 CPU、I/O 和内存。如果缺少任何一种资源，并行查询可能会过度使用资源，这就会负面地影响总体性能和运行时间。

过去，对于许多数据仓库来说，人们总认为必须应用并行查询，这只是因为过去（例如，1995 年）数据仓库很稀少，通常只有一个很小、很集中的用户群。而在 2005 年的今天，到处都有数据仓库，而且数据仓库的用户群与事务性系统的用户群同样庞大。这说明，在给定时刻，你可能没有足够的空闲资源来启用这些系统上的并行查询。但这并不是说“这种情况下并行执行通常没有用”，而是说，并行执行更应该算是一个 DBA 工具（在“并行 DDL“一节将介绍），而不是一个并行查询工具。

14.3 并行 DML

Oracle 文档将并行 DML（PDML）一词的范围限制为只包括 INSERT、UPDATE、DELETE 和 MERGE（不像平常的 DML 那样还包括 SELECT）。在 PDML 期间，Oracle 可以使用多个并行执行服务器来执行 INSERT、UPDATE、DELETE 或 MERGE，而不是只利用一个串行进程。在一个有充足 I/O 带宽的多 CPU 主机上，对于大规模的 DML 操作，可能会得到很大的速度提升。

不过，不能把 PDML 当成提高 OLTP 应用速度的一个特性。如前所述，并行操作设计为要充分、完全地利用一台机器上的所有资源。通过这种设计，一个用户可以完全使用机器上的所有磁盘、CPU 和内存。在某些数据仓库中（有大量数据，而用户很少），这可能正是你想要的。而在一个 OLTP 系统中（大量用户都在做很短、很快的事务），可能就不能希望如此了，你不想让用户能够完全占用机器资源。

这听上去好像有些矛盾：我们使用并行查询是为了扩缩，它怎么可能是不可扩缩的呢？不过要知道，应用到一个 OLTP 系统时，这种说法确实很正确。并行查询不能随着并发用户数的增加而扩展。并行查询设计为允许一个会话能像 100 个并发会话一样生成同样多的工作。但在 OLTP 系统中，我们确认不希望一个用户生成 100 个用户的工作。

PDML 在大型数据仓库环境中很有用，它有利于大量数据的批量更新。类似于 Oracle 执行的分布式查询，PDML 操作采用同样的方式执行，即每个并行执行服务器相当于一个单独数据库实例中的一个进程。表的每一部分（每一片）由一个单独的线程利用其自己的独立事务来修改（相应地，这个线程可能有自己的 undo 段）。这些事务都结束后，会执行一个相当于快速 2PC 的过程来提交这些单独的独立事务。图 14-2 展示了使用 4 个并行执行服务器的并行更新。每个并行执行服务器都有其自己独立的事务，这些事务要么都由 PDML 协调会话提交，要么无一提交。

实际上，我们可以观察到为并行执行服务器创建的各个独立事务。在此还是使用前面的两个会话。在 SID=162 的会话中，我们显式地启用了并行 DML。在这方面，PDML 有别于并行查询；除非显式地请求 PDML，否则不能执行 PDML。

```
big_table@ORA10GR1> alter session enable parallel dml;

Session altered.
```

图 14-2　并行更新（PDML）示意图

这个表是“并行的“，但与并行查询不同，这对于 PDML 还不够。之所以要显式地在会话中启用 PDML，原因是 PDML 存在一些相关的限制，我会在介绍完这个例子之后列出这些限制。

在这个会话中，下面执行一个批量 UPDATE，由于表已经“启用并行 DML“（enable parallel dml），所以实际上这会并行执行：

```
big_table@ORA10GR1> update big_table set status = 'done';
```

在另一个会话中，我们将 V$SESSION 联结到 V$TRANSACTION，来显示对应这个 PDML 操作的活动会话，并显示这些会话的独立事务信息：

```
ops$tkyte@ORA10GR1> select a.sid, a.program, b.start_time, b.used_ublk,
  2          b.xidusn ||'.'|| b.xidslot || '.' || b.xidsqn trans_id
  3    from v$session a, v$transaction b
  4    where a.taddr = b.addr
  5          and a.sid in ( select sid
  6                  from v$px_session
```

```
  7                 where qcsid = 162 )
  8      order by sid
  9      /


                                                                        SID
PROGRAM                          START_TIME          USED_UBLK       TRANS_ID
                                                                        ----
-------------------------------- ------------------- ----------- ------------
   136 oracle@dellpe (P009)      08/03/05 14:28:17          6256      18.9.37
   137 oracle@dellpe (P013)      08/03/05 14:28:17          6369     21.39.225
   138 oracle@dellpe (P015)      08/03/05 14:28:17          6799     24.16.1175
   139 oracle@dellpe (P008)      08/03/05 14:28:17          6729     15.41.68
   140 oracle@dellpe (P014)      08/03/05 14:28:17          6462     22.41.444
   141 oracle@dellpe (P012)      08/03/05 14:28:17          6436     20.46.46
   142 oracle@dellpe (P010)      08/03/05 14:28:17          6607     19.44.37
   143 oracle@dellpe (P007)      08/03/05 14:28:17             1     17.12.46
   144 oracle@dellpe (P006)      08/03/05 14:28:17             1     13.25.302
   145 oracle@dellpe (P003)      08/03/05 14:28:17             1     1.21.1249
   146 oracle@dellpe (P002)      08/03/05 14:28:17             1     14.42.49
   147 oracle@dellpe (P005)      08/03/05 14:28:17             1     12.18.323
   150 oracle@dellpe (P011)      08/03/05 14:28:17          6699     23.2.565
   151 oracle@dellpe (P004)      08/03/05 14:28:17             1     16.26.46
   152 oracle@dellpe (P001)      08/03/05 14:28:17             1     11.13.336
   153 oracle@dellpe (P000)      08/03/05 14:28:17             1     2.29.1103
```

```
 162 sqlplus@dellpe (TNS V1-V3)      08/03/05 14:25:46                2        3.13.2697

17 rows selected.
```

可以看到，与并行查询表相比，这里发生了更多事情。这个操作上有 17 个进程，而不像前面只有 9 个进程。这是因为：开发的计划中有一步是更新表，另外还有一些独立的步骤用于更新索引条目。通过查看从 DBMS_XPLAN 得到解释计划输出（我们对这个输出进行了编辑，去掉了尾部几列，以便这个输出能在一页的篇幅内放下），可以看到以下结果：

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

| Id | Operation                                | Name            |

-------------------------------------------------------------------------------

|  0 |  UPDATE STATEMENT                        |                 |

|  1 |   PX COORDINATOR                         |                 |

|  2 |    PX SEND QC (RANDOM)                   | :TQ10001        |

|  3 |     INDEX MAINTENANCE                    | BIG_TABLE       |

|  4 |      PX RECEIVE                          |                 |

|  5 |       PX SEND RANGE                      | :TQ10000        |

|  6 |        UPDATE                            | BIG_TABLE       |

|  7 |         PX BLOCK ITERATOR                |                 |

|  8 |          TABLE ACCESS FULL               | BIG_TABLE       |

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

由于 PDML 采用的一种伪分布式的实现，因此存在一些限制：

- PDML 操作期间不支持触发器。在我看来，这是一个很合理的限制，因为触发器可能会向更新增加大量开销，而你使用 PDML 的本来目的是为了更快一些，这两方面是矛盾的，不能放在一起。
- PDML 期间，不支持某些声明方式的引用完整性约束，因为表中的每一片（部

分）会在单独的会话中作为单独的事务进行修改。例如，PDML 操作不支持自引用完整性。如果真的支持自引用完整性，可能会出现死锁和其他锁定问题。

- 在提交或回滚之前，不能访问用 PDML 修改的表。
- PDML 不支持高级复制（因为复制特性的实现要基于触发器）。
- 不支持延迟约束（也就是说，采用延迟模式的约束）。
- 如果表是分区的，PDML 只可能在有位图索引或 LOB 列的表上执行，而且并行度取决于分区数。在这种情况下，无法在分区内并行执行一个操作，因为每个分区只有一个并行执行服务器来处理。
- 执行 PDML 时不支持分布式事务。
- PDML 不支持聚簇表。

如果违反了上述任何一个限制，就会发生下面的某种情况：语句会串行执行（完全不涉及并行化），或者会产生一个错误。例如，如果对表 T 执行 PDML，然后在结束事务之前试图查询表 T，就会收到一个错误。

14.4 并行 DDL

我认为，Oracle 并行就是中真正的“闪光点“就是并行 DDL。我们讨论过，并行执行通常不适用于 OLTP 系统。实际上，对于许多数据仓库来说，并行查询也变得越来越没有意义。过去，数据仓库往往是为一个很小、很集中的用户群建立的，有时只包括一两个分析人员。不过，在过去的 10 年中，我发现，数据仓库的用户群已经从很小的规模发展为包括数百甚至上千位的用户。考虑这样一个数据仓库，它以一个基于 Web 的应用作为前端，按理讲，数千（甚至更多）的用户都可以即时地访问这个数据仓库。

但是如果是 DBA 执行大规模的批操作（可能在一个维护窗口中执行）则是另一码事。DBA 只是一个人（而不是很多用户），他可能有一台能力很强的机器，有丰富的计算资源可用。DBA 只有“一件事要做“：加载这个数据，重组表，再重建索引。如果没有并行执行，DBA 将很难真正充分利用硬件的全部能力。但如果利用并行执行，则完全可以做到。以下 SQL DDL 命令允许”并行化“：

- CREATE INDEX：多个并行执行服务器可以扫描表、对数据排序，并把有序的段写出到索引结构。
- CREATE TABLE AS SELECT：执行 SELECT 的查询可以使用并行查询来执行，表加载本身可以并行完成。
- ALTER INDEX REBUILD：索引结构可以并行重建。
- ALTER TABLE MOVE：表可以并行移动。
- ALTER TABLE SPLIT|COALESCE PARTITION：单个表分区可以并行地分解或合并。
- ALTER INDEX SPLIT PARTITION：索引分区可以并行地分解。

前 4 个命令还适用于单个的表/索引分区，也就是说，可以并行地 MOVE 一个表的单个

分区。

对我来说，并行 DDL 才是 Oracle 中并行执行最突出的优点。当然，并行查询可以用来加快某些长时间运行的操作，但是从维护的观点看，以及从管理的角度来说，并行 DDL 才是影响我们（DBA 和开发人员）的并行操作。如果认为并行查询主要是为最终用户设计的，那么并行 DDL 则是为 DBA/开发人员设计的。

14.4.1 并行 DDL 和使用外部表的数据加载

Oracle9i 中我最喜欢的新特性之一是外部表（external table），在数据加载方面外部表尤其有用。我们将在下一章详细地介绍数据加载和外部表。不过，作为一个简要说明，现在先简单地了解一下有关内容，以便讨论并行 DDL 对于确定区段大小和区段截断（extent trimming）有何影响。

利用外部表，我们可以很容易地执行并行直接路径加载，而无需太多的考虑。Oracle 7.1 允许我们执行并行直接路径加载，即多个会话可以直接写至 Oracle 数据文件，而完全绕过缓冲区缓存，避开表数据的 undo 生成，可能还可以避免 redo 生成。这是通过 SQL*Loader 完成的。DBA 必须编写多个 SQL*Loader 会话的脚本，手动地分解要加载的输入数据文件，确定并行度，并协调所有 SQL*Loader 进程。简单地说，尽管这是可以做到的，但很困难。

利用并行 DDL，再加上外部表，就能通过一个简单的 CREATE TABLE AS SELECT or INSERT /*+ APPEND */来实现并行直接路径加载。不用再编写脚本，不必再分解文件，也不用协调要运行的 N 个脚本。简单地说，通过结合并行 DDL 和外部表，不仅提供了纯粹的易用性，而且全无性能损失。

下面来看这方面的一个简单的例子。稍后将看到如果创建一个外部表。下一章还会更详细地介绍如果利用外部表实现数据加载。不过对现在来说，我们只是使用一个“真实“表（而不是外部表），由此加载另一个表，这类似于许多人在数据仓库中使用暂存表（staging table）加载数据的做法。简单地说，这些技术如下：

（1） 使用某个抽取、转换、加载（extract, transform, load, ETL）工具来创建输入文件。

（2） 将执行输入文件加载到暂存表。

（3） 使用对这些暂存表的查询加载一个新表。

还是使用前面的 BIG_TABLE（启用了并行），其中包含 10,000,000 行记录。我们将把这个表联结到第二个表 USER_INFO，其中包含 ALL_USERS 字典视图中与 OWNER 相关的信息。我们的目标是将这个信息逆规范化为一个平面结构。

首先创建 USER_INFO 表，启用并行操作，然后生成这个表的统计信息：

```
big_table@ORA10GR1> create table user_info as select * from all_users;

Table created.
```

```
big_table@ORA10GR1> alter table user_info parallel;

Table altered.



big_table@ORA10GR1> exec dbms_stats.gather_table_stats( user, 'USER_INFO' );

PL/SQL procedure successfully completed.
```

现在，我们想用这个信息采用并行直接路径加载来加载一个新表。这里使用的查询很简单：

```
create table new_table parallel

as

select a.*, b.user_id, b.created user_created

from big_table a, user_info b

where a.owner = b.username
```

在 Oracle 10g 中，这个特定 CREATE TABLE AS SELECT 的计划如下所示：

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

| Id  | Operation                        | Name            | TQ      |IN-OUT   | PQ
Distrib          |

---------------------------------------------------------------------------------------------------------
---------------

|   0 |                                  | CREATE TABLE
STATEMENT      |                 |          |            |                  |

|   1 |                                  |                 PX
COORDINATOR                |                 |          |            |
  |

| 2   | PX SEND QC (RANDOM)        | :TQ10001       | Q1,01 | P->S       | QC
(RAND)       |

| 3   |    LOAD AS SELECT                |                 | Q1,01  |
PCWP     |                  |

|* 4  |     HASH JOIN                    |                 | Q1,01  |
```

```
PCWP     |                    |

|   5        |       PX    RECEIVE                    |                    |    Q1,01    |
PCWP     |                    |

|  6     |      PX   SEND   BROADCAST      |  :TQ10000        |  Q1,00  |  P->P        |
BROADCAST     |

|   7        |       PX    BLOCK    ITERATOR          |                    |    Q1,00    |
PCWC     |                    |

|   8        |         TABLE    ACCESS    FULL       |    USER_INFO    |    Q1,00    |
PCWP     |                    |

|   9        |       PX    BLOCK    ITERATOR           |                    |    Q1,01    |
PCWC     |                    |

|   10   |        TABLE    ACCESS    FULL           |    BIG_TABLE        |    Q1,01    |
PCWP     |                    |

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

如果从第 4 步向下看，这些就是查询（SELECT）部分。BIG_TABLE 的扫描和与 USER_INFO 的散列联结是并行执行的，并将各个子结果加载到表的某个部分中（第 3 步，LOAD AS SELECT）。每个并行执行服务器完成其联结和加载工作后，会把其结果发送给查询协调器。在这里，结果只是提示“成功“还是”失败“，因为工作已经执行。

仅此而已，利用并行直接路径加载，使得问题变得很容易。对于这些操作，最重要的是要考虑如何使用空间（或不使用）。有一种称为区段截断（extent trimming）的副作用相当重要，现在我要花一些时间来讨论这个内容。

14.4.2 并行 DDL 和区段截断

并行 DDL 依赖于直接路径操作。也就是说，数据不传递到缓冲区缓存以便以后写出；而是由一个操作（如 CREATE TABLE AS SELECT）来创建新的区段，并直接写入这些区段，数据直接从查询写到磁盘（放在这些新分配的区段中）。每个并行执行服务器执行自己的部分 CREATE TABLE AS SELECT 工作，并且都会写至自己的区段。INSERT /*+ APPEND */（直接路径插入）会在一个段的 HWM“之上“写，每个并行执行服务器再写至其自己的一组区段，而不会与其他并行执行服务器共享。因此，如果执行一个并行 CREATE TABLE AS SELECT，并使用 4 个并行执行服务器来创建表，就至少有 4 个分区，可能还会更多。每个并行执行服务器会分配其自己的区段，向其写入，等填满时，再分配另一个新的区段，并行执行服务器不会使用由其他并行执行服务器非品牌的区段。

图 14-3 显示了这个过程。这里由 4 个并行执行服务器执行一个 CREATE TABLE NEW_TABLE AS SELECT。在图中，每个并行执行服务器分别用一种不同的颜色表示（白色、浅灰色、深灰色或黑色）。“磁鼓“中的方框表示这个 CREATE TABLE 语句在某个数据

文件中创建的区段。每个区段分别用上述某个颜色表示，之所以能这样显示，原因很简单，任何给定区段中的所有数据都只会由这 4 个并行执行服务器中之一加载，在此显示出，P003 创建并加载了其中 4 个区段。另一方面，P000 则创建并加载了 5 个区段，等等。

图 14-3　并行 DDL 区段分配示意图

初看上去一切都很好。但是在数据仓库环境中，执行一个大规模的加载之后，这可能导致“过渡浪费“。假设你想加载 1,010MB 的数据（大约 1GB），而且正在使用一个有 100MB 区段的表空间，你决定使用 10 个并行执行服务器来加载这个数据。每个并行执行服务器先分配其自己的 100MB 区段（总共会有 10 个 100MB 的区段），并在其中填入数据。由于每个并行执行服务器都要加载 101MB 的数据，所以它会填满第一个区段，然后再继续分配另一个 100MB 的区段，但实际上只会使用这个区段中 1MB 的空间。现在就有了 20 区段，其中 10 个是满的，另外 10 个则不同，这 10 个区段中都各有 1MB 的数据，因此，总共会有 990MB 的空间是”已分配但未使用的“。下一次加载是可以使用这个空间，但是对现在来说，你就有了 990MB 的死空间。此时区段截断（extend trimming）就能派上用场了。Oracle 会试图取每个并行执行服务器的最后一个区段，并将其”截断为“可能的最小大小。

1.　区段截断和字典管理表空间

如果使用传统的字典管理表空间，Oracle 可以把只包含 1MB 数据的各个 100MB 区段转变或 1MB 的区段。遗憾的是，（在字典管理的表空间中）这会留下 10 个不连续的 99MB 空闲区段，因为你的分配机制采用的是 100MB 区段，所以这 990MB 空间就会用不上！下一次分配 100MB 时，往往无法使用现有的这些空间，因为现在的情况是：有 99MB 的空闲空间，接下来是 1MB 的已分配空间，然后又是 99MB 空闲空间，依此类推。在本书中，我们不再复习字典管理方法。

2.　区段截断和本地管理表空间

现在来看本地管理表空间。对此有两种类型：UNIFORM SIZE 和 AUTOALLOCATE，UNIFORM SIZE 是指表空间中的每个区段大小总是完全相同；AUTOALLOCATE 则表示 Oracle 会使用一种内部算法来确定每个区段应该是多大。这些方法都能很好地解决上述问题（即先有 99MB 空闲空间，其后是 1MB 已用空间，再后面又是 99MB 空闲空间，依此类推，以至于存在大量无法使用的空闲空间）。不过，这两种方法的解决策略截然不同。

UNIFORM SIZE 方法完全排除了区段截断。如果使用 UNIFORM SIZE，Oracle 就不能执行区段截断。所有区段都只能有惟一一种大小，不能有任何区段小于（或大于）这个大小。

另一方面，AUTOALLOCATE 区段支持区段截断，但是采用了一种聪明的方式。AUTOALLOCATE 方法使用一些特定大小的区段，而且能使用不同大小的空间。也就是说，利用这种算法，一段时间后将允许使用表空间中的所有空闲空间。在字典管理的表空间中，如果请求一个 100MB 区段，倘若 Oracle 只找到了 99MB 的自由区段，请求还是会失败（尽管离要求只差了一点点）。与字典管理表空间不同，有 AUTOALLOCATE 区段的本地管理表空间可以更为灵活。为了试图使用所有空闲空间，它可以减小所请求的空间大小。

下面来看这种本地管理表空间方法之间的差别。为此，我们需要处理一个实际的例子。这里将建立一个外部表，它能用于并行直接路径加载（我们会频繁地执行并行直接路径加载）。即使你还在使用 SQL*Loader 来实现并行直接路径加载数据，这一节的内容也完全适用，只不过要求你自己编写脚本来执行数据的具体加载。因此，为了讨论区段截断，我们需要建立加载示例，然后在不同的条件下执行加载，并分析结果。

□ **环境建立**

首先需要一个外部表。我多次发现，加载数据时我常常用到 SQL*Loader 的一个遗留控制文件。例如，可能如下所示：

```
LOAD DATA

INFILE '/tmp/big_table.dat'

INTO TABLE big_table

REPLACE

FIELDS TERMINATED BY '|'

(

id ,owner ,object_name ,subobject_name ,object_id

,data_object_id ,object_type ,created ,last_ddl_time

,timestamp ,status ,temporary ,generated ,secondary

)
```

使用 SQL*Loader 就可以很容易地把这个文件转换为一个外部表定义：

```
$ sqlldr big_table/big_table big_table.ctl external_table=generate_only

SQL*Loader: Release 10.1.0.3.0 - Production on Mon Jul 11 14:16:20 2005

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

注意传递给 SQL*Loader 的参数 EXTERNAL_TABLE。在这里，这个参数使得 SQL*Loader 不会加载数据，而是在日志文件中为我们生成一个 CREATE TABLE 语句。这个 CREATE TABLE 语句如下所示（在此有删减；为了让这个例子更简短，我对一些重复的元素进行了编辑）：

```
CREATE TABLE "SYS_SQLLDR_X_EXT_BIG_TABLE"

(

"ID" NUMBER,

...

"SECONDARY" VARCHAR2(1)

)

ORGANIZATION external

(

        TYPE oracle_loader

        DEFAULT DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000

        ACCESS PARAMETERS

        (

                RECORDS DELIMITED BY NEWLINE CHARACTERSET
WE8ISO8859P1

            BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'big_table.bad'

            LOGFILE 'big_table.log_xt'

            READSIZE 1048576

            FIELDS TERMINATED BY "|" LDRTRIM
```

```
                REJECT ROWS WITH ALL NULL FIELDS
                (
                    "ID" CHAR(255)
                    TERMINATED BY "|",
                    ....
                    "SECONDARY" CHAR(255)
                    TERMINATED BY "|"
                )
            )
            location
            (
                'big_table.dat'
            )
        )REJECT LIMIT UNLIMITED
```

我们所要做的只是给这个外部表起一个我们想要的名字：可能要改变目录等：

```
ops$tkyte@ORA10GR1> create or replace directory my_dir as '/tmp/'
  2  /

Directory created.
```

在此之后，只需具体创建这个表：

```
ops$tkyte@ORA10GR1> CREATE TABLE "BIG_TABLE_ET"
  2  (
  3       "ID" NUMBER,
           ...
 16      "SECONDARY" VARCHAR2(1)
```

```
 17 )
 18 ORGANIZATION external
 19 (
 20       TYPE oracle_loader
 21       DEFAULT DIRECTORY MY_DIR
 22       ACCESS PARAMETERS
 23       (
 24                RECORDS DELIMITED BY NEWLINE CHARACTERSET
WE8ISO8859P1
 25            READSIZE 1048576
 26            FIELDS TERMINATED BY "|" LDRTRIM
 27            REJECT ROWS WITH ALL NULL FIELDS
 28       )
 29       location
 30       (
 31            'big_table.dat'
 32       )
 33 )REJECT LIMIT UNLIMITED
 34 /

Table created.
```

然后使这个表启用并行（PARALLEL）。这是很神奇的一步，正是这一步使我们可以很容易地执行并行直接路径加载：

```
ops$tkyte@ORA10GR1> alter table big_table_et PARALLEL;

Table altered.
```

注意 PARALLEL 子句也可以在 CREATE TABLE 语句本身之上使用。也就是说，可以在 REJECT LIMIT UNLIMITED 后面增加关键字 PARALLEL。我之所以使用 ALTER 语

句，只是想让读者更注意这一点：外部表确实启用了并行。

□ 使用 UNIFORM 与 AUTOALLOCATE 本地管理表空间的区段截断

要建立加载组件，所要在的工作就这么多。现在，我们想分析在使用 UNIFORM 区段大小的本地管理表空间（locally-managed tablespace，LMT）中如果管理空间，以及在使用 AUTOALLOCATE 区段的 LMT 中如果管理空间，并对二者做一个比较。在这个例子中，我们将使用 100MB 的区段。首先创建 LMT_UNIFORM，它使用统一的区段大小：

```
ops$tkyte@ORA10GR1> create tablespace lmt_uniform

  2 datafile '/u03/ora10gr1/lmt_uniform.dbf' size 1048640K reuse

  3 autoextend on next 100m

  4 extent management local

  5 uniform size 100m;

Tablespace created.
```

接下来，创建 LMT_AUTO，它使用 AUTOALLOCATE 来确定区段大小：

```
ops$tkyte@ORA10GR1> create tablespace lmt_auto

  2 datafile '/u03/ora10gr1/lmt_auto.dbf' size 1048640K reuse

  3 autoextend on next 100m

  4 extent management local

  5 autoallocate;

Tablespace created.
```

每个表空间开始时有一个 1BG 的数据文件（另外 LMT 还要用额外的 64KB 来管理存储空间：如果使用 32KB 块大小，则另外需要 128KB 而不是 64KB 来管理存储空间）。我们允许这些数据文件一次自动扩展 100MB。下面要加载这个文件：

```
$ ls -lag big_table.dat

-rw-rw-r-- 1 tkyte 1067107251 Jul 11 13:46 big_table.dat
```

这是一个有 10,000,000 条记录的文件。它使用big_table.sql脚本创建（这个脚本见本书开头的“环境配置“一节），再使用 flat.sql 脚本卸载（这个脚本可以从http://asktom.oracle.com/~tkyte /flat/index.html得到）。接下来，执行一个并行直接路径加载，将这个文件加载到各个表空间中：

```
ops$tkyte@ORA10GR1> create table uniform_test
```

```
  2 parallel
  3 tablespace lmt_uniform
  4 as
  5 select * from big_table_et;
Table created.

ops$tkyte@ORA10GR1> create table autoallocate_test
  2 parallel
  3 tablespace lmt_auto
  4 as
  5 select * from big_table_et;
Table created.
```

在我的系统上（有 4 个 CPU），使用了 8 个并行执行服务器和一个协调器来执行这些 CREATE TABLE 语句。运行这些语句时，可以通过查询与并行执行有关的一个动态性能视图 V$PX_SESSION 来验证这一点：

```
sys@ORA10GR1> select sid, serial#, qcsid, qcserial#, degree
  2 from v$px_session;

       SID SERIAL# QCSID      QCSERIAL# DEGREE
---------- ---------- ----------  ----------  ----------
       137         17         154         998           8
       139         13         154         998           8
       141         17         154         998           8
       150        945         154         998           8
```

```
       161          836          154          998            8

       138            8          154          998            8

       147           15          154          998            8

       143           41          154          998            8

       154          998          154

9 rows selected.
```

注意 创建 UNIFORM_TEST 和 AUTOALLOCATE_SET 表时，我们只是在每个表上指定了“并行“（parallel），而由 Oracle 选择并行度。在这个例子中，我是这台机器的惟一用户（所有资源都可用），所以根据 CPU 数（4）和 PARALLEL_THREADS_PER_CPU 参数设置（默认为 2），Oracle 会将并行度默认为 8。

SID、SERIAL#是并行执行会话的标识符，QCSID、QCSERIAL#是并行执行查询协调器的标识符。因此，如果运行着 8 个并行执行会话，我们可能想看看空间是如果使用的。通过对 USER_SEGMENTS 的一个快速查询，就能清楚地知道：

```
ops$tkyte@ORA10GR1> select segment_name, blocks, extents

  2 from user_segments

  3 where segment_name in ( 'UNIFORM_TEST', 'AUTOALLOCATE_TEST' );



SEGMENT_NAME             BLOCKS      EXTENTS

---------------           ----------          ----------

UNIFORM_TEST              204800               16

AUTOALLOCATE_TEST       145592              714
```

由于我们使用的块大小是 8KB，这说明二者相差大约 462MB，或者从比例来看，AUTOALLOCATE_TEST 的已分配空间大约是 UNIFORM_TEST 的 70%。如果查看实际使用的空间：

```
ops$tkyte@ORA10GR1> exec show_space('UNIFORM_TEST' );

Free Blocks.....                    59,224

Total Blocks.....                   204,800
```

```
Total Bytes.....          1,677,721,600

Total MBytes.....             1,600

Unused Blocks.....                0

Unused Bytes.....                 0

Last Used Ext FileId.....         6

Last Used Ext BlockId.....        9

Last Used Block.....         12,800

PL/SQL procedure successfully completed.



ops$tkyte@ORA10GR1> exec show_space('AUTOALLOCATE_TEST' );

Free Blocks.....                 16

Total Blocks.....            145,592

Total Bytes.....          1,192,689,664

Total MBytes.....             1,137

Unused Blocks.....                0

Unused Bytes.....                 0

Last Used Ext FileId.....         8

Last Used Ext BlockId.....       41

Last Used Block.....              8

PL/SQL procedure successfully completed.
```

注意　SHOW_SPACE 过程在本身开头的“环境配置“中已经介绍过。

可以看到，如果去掉 UNIFORM_TEST 中 freelist 所占的块数（59,224 个空闲块），这两个表实际所占的空间几乎是一样的，不过，UNIFORM 表空间总共需要的空间确实大多了。这完全归因于没有区段截断。如果查看 UNIFORM_TEST，可以很清楚地看出：

```
ops$tkyte@ORA10GR1> select segment_name, extent_id, blocks
```

```
  2 from user_extents where segment_name = 'UNIFORM_TEST';


SEGMENT_NAME      EXTENT_ID        BLOCKS
--------------                  ----------         ----------
UNIFORM_TEST                 0          12800
UNIFORM_TEST                 1          12800
UNIFORM_TEST                 2          12800
UNIFORM_TEST                 3          12800
UNIFORM_TEST                 4          12800
UNIFORM_TEST                 5          12800
UNIFORM_TEST                 6          12800
UNIFORM_TEST                 7          12800
UNIFORM_TEST                 8          12800
UNIFORM_TEST                 9          12800
UNIFORM_TEST                10          12800
UNIFORM_TEST                11          12800
UNIFORM_TEST                12          12800
UNIFORM_TEST                13          12800
UNIFORM_TEST                14          12800
UNIFORM_TEST                15          12800
16 rows selected.
```

每个区段的大小都是 100MB。再来看 AUTOALLOCATE_TEST，如果把所有 714 个区段都列出来就太浪费篇幅了，所以下面汇总起来看：

```
ops$tkyte@ORA10GR1> select segment_name, blocks, count(*)
```

```
  2 from user_extents
  3 where segment_name = 'AUTOALLOCATE_TEST'
  4 group by segment_name, blocks
  5 /

SEGMENT_NAME          BLOCKS       COUNT(*)
----------------      ----------   ----------
AUTOALLOCATE_TEST          8            128
AUTOALLOCATE_TEST        128            504
AUTOALLOCATE_TEST        240              1
AUTOALLOCATE_TEST        392              1
AUTOALLOCATE_TEST        512              1
AUTOALLOCATE_TEST        656              1
AUTOALLOCATE_TEST        752              5
AUTOALLOCATE_TEST        768              1
AUTOALLOCATE_TEST       1024             72

9 rows selected.
```

对于使用 AUTOALLOCATE 的 LMT，通常都是这样来分配空间。这里的 8 块、128 块和 1,024 块的区段是“正常“区段：使用 AUTOALLOCATE 时，总能观察到这样一些区段。不过，余下的区段就不那么”正常“了，观察时不一定能看到。这是因为发生了区段截断。有些并行执行服务器完成了它们的那部分加载后，取其最后的 8MB（1,024 块）区段，并对其截断，这就留下了一些多余的空间。这样，如果另外某个并行执行会话需要空间，就可以使用这部分多余的空间。陆续地，当另外这些并行执行会话处理完自己的加载时，也会截断其最后一个区段，而留下一些多余的空间。

应该使用哪种方法呢？如果你的目标是尽可能经常并行地执行直接路径加载，我建议你采用 AUTOALLOCATE 作为区段管理策略。诸如此类的并行直接路径操作不会使用对象 HWM 以下的已用空间，即不会用 freelits 上的空间。除非你还要对表执行一些传统路径插入，否则 UNIFORM 分配方法会在这些表中永久地保留一些从不使用的额外空闲空间。除非你能把 UNIFORM LMT 的区段大小定得更小，否则，过一段时间后，就会看到我所说的

过度浪费的情况：而且要记住，这些空间与段有关，所以对表进行完全扫描时也必须扫描这些空间。

为了说明这一点，下面使用同样的输入再对这两个表执行另一个并行直接路径加载：

```
ops$tkyte@ORA10GR1> alter session enable parallel dml;

Session altered.



ops$tkyte@ORA10GR1> insert /*+ append */ into UNIFORM_TEST

  2 select * from big_table_et;

10000000 rows created.



ops$tkyte@ORA10GR1> insert /*+ append */ into AUTOALLOCATE_TEST

  2 select * from big_table_et;

10000000 rows created.



ops$tkyte@ORA10GR1> commit;

Commit complete.
```

在这个操作之后，比较两个表的空间利用情况，如下：

```
ops$tkyte@ORA10GR1> exec show_space( 'UNIFORM_TEST' );

Free Blocks.....                    118,463

Total Blocks.....                   409,600

Total Bytes.....                3,355,443,200

Total MBytes.....                     3,200

Unused Blocks.....                        0

Unused Bytes.....                         0
```

```
Last Used Ext FileId.....                      6

Last Used Ext BlockId.....                 281,609

Last Used Block.....                        12,800

PL/SQL procedure successfully completed.



ops$tkyte@ORA10GR1> exec show_space( 'AUTOALLOCATE_TEST' );

Free Blocks.....                               48

Total Blocks.....                           291,184

Total Bytes.....                      2,385,379,328

Total MBytes.....                           2,274

Unused Blocks.....                             0

Unused Bytes.....                              0

Last Used Ext FileId.....                      8

Last Used Ext BlockId.....                 140,025

Last Used Block.....                           8

PL/SQL procedure successfully completed.
```

可以看到，随着我们使用并行直接路径操作向表 UNIFORM_TEST 加载越来越多的数据，过一段时间后，空间利用情况会变得越来越糟糕。对此，我们可能希望使用一个更小的统一区段大小，或者使用 AUTOALLOCATE。一段时间后，AUTOALLOCATE 也可能生成更多的区段，但是由于会发生区段截断，所以空间利用情况要好得多。

14.5 并行恢复

Oracle 中还有另一种形式的并行执行，即能够执行并行恢复。并行恢复（parallel recovery） 可以在实例级完成，使得软件、操作系统或一般系统失败后可以更快地执行所需的恢复。还可以在介质恢复中应用并行恢复（例如，从备份恢复）。这本书并不打算 讨论与恢复有关的内容，所以在此只简单提一下，只要知道存在并行恢复就可以了。如果要进一步了解有关内容，建议阅读以下 Oracle 手册：

- Oracle Backup and Recovery Basics，其中涵盖了有关并行介质恢复的信息。
- Oracle Performance Tuning Guide，其中包括有关并行实例恢复的信息。

14.6 过程并行化

在此要讨论两种类型的过程并行化：

并行管道函数（parallel pipelined function），这是 Oracle 的一个特性。

“DIY 并行化“（DIY parallelism），这是指在你自己的应用上使用 Oracle 执行并行全表扫描所采用的技术。与其说 DIY 并行化是一种直接内建的 Oracle 中的特性，不如说是一种开发技术。

你可能经常看到，设计为串行执行的应用（一般是批处理应用）往往类似于以下过程：

```
Create procedure process_data

As

Begin

   For x in ( select * from some_table )

        Perform complex process on X

        Update some other table, or insert the record somewhere else

   End loop

end
```

在这种情况下，Oracle 的并行查询或 PDML 没有什么帮助（在这里，实际上 Oracle 的 SQL 并行执行可能只会导致数据库占用更多的资源，而且花费更长的时间）。如果 Oracle 并行地执行简单的 SELECT * FROM SOME_TABLE，可能不会提供任何显著的速度提升。如果 Oracle 在完成复杂的处理之后再并行地执行 UPDATE 或 INSERT，则没有任何好处（毕竟，这里只会 UPDATE/INSERT 一行）。

在此显然可以做这样一件事：完成复杂处理之后，对 UPDATE/INSERT 使用批量处理。不过，这样不会是运行时间减少 50%（或更多），而通常这才是你的目标。别曲解我的意思，我是说：在此你可能想对修改实现批量处理，但是这样做并不会使处理速度提高 2 倍、2 倍或更多倍。

现在，假设你晚上在一台 4CPU 的机器上运行这个过程，机器上只运行着这一个过程，而没有其他活动。你会观察到，此时只会不充分地使用这个系统上的一个 CPU，而且根本没有用多少磁盘。不仅如此，这个过程的执行要花费数小时，随着增加更多的数据，每天需要的时间会越来越长。你需要把运行时间减少几倍，它的速度应该是现在的 4 倍或 8 倍才行，所以稍稍地改善百分之几只能算杯水车薪，是远远不够的。能你该怎么做呢？

对此有两种办法。一种方法是实现一个并行管道函数，Oracle 会确定一个合适的并行度（这是推荐的做法）。Oracle 会创建会话，进行协调，并运行这些会话，这与前面使用了 CREATE TABLE AS SELECT OR INSERT /*+ APPEND */的并行 DDL 例子很相似。Oracle 会为我们完全自动地实现并行直接路径加载。另一种方法是 DIY 并行化。下面几节将分别

介绍这两种方法。

14.6.1 并行管道函数

还是用前面那个串行进程 PROCESS_DATA，不过这一次让 Oracle 为我们并行地执行这个进程。为此，需要把这个例程“倒过来“。并非从某个表中选择行，处理这些行，再插入到另一个表，而是向另一个表中插入获取某些行并对其处理的结果。我们要删除循环最下面的 INSERT，而代之以一个 PIPE ROW 子句。PIPE ROW 子句允许这个 PL/SQL 例程生成表数据作为输出，这样我们就能从这个 PL/SQL 过程 SELECT 数据。原本处理数据的过程性 PL/SQL 例程实际上变成了一个表，我们获取并处理的行就是输出。其实这种情况在这本书中已经屡屡出现，每次执行以下语句时都是如此：

```
Select * from table(dbms_xplan.display);
```

这是一个 PL/SQL 例程，它要读 PLAN_TABLE；重建输出，甚至会增加一些行；然后使用 PIPE ROW 输出这个数据，将其发回给客户。我们在这里实际上要做同样的事情，只不过允许并行地处理。

这个例子中要使用两个表：T1 和 T2。T1 是先读的表，T2 表用来移入这个信息。假设这是一种 ETL 过程，我们要运行这个过程获得每天的事务性数据，并将其转换，作为第二天的报告信息。我们要用的两个表如下：

```
ops$tkyte-ORA10G> create table t1
  2    as
  3    select object_id id, object_name text
  4    from all_objects;

Table created.



ops$tkyte-ORA10G> begin
  2            dbms_stats.set_table_stats
  3            ( user, 'T1', numrows=>10000000,numblks=>100000 );
  4    end;
  5    /

PL/SQL procedure successfully completed.
```

```
ops$tkyte-ORA10G> create table t2
  2     as
  3     select t1.*, 0 session_id
  4     from t1
  5     where 1=0;
Table created.
```

这里使用 DBMS_STATS 来“骗过“优化器，让它以为输入表中有 10,000,000 行，而且占用了 100,000 个数据库块。在此我们想模拟一个大表。第二个表 T2 是第一个表的一个副本，只是在结构中增加了一个 SESSION_ID 列。这个列很有用，可以通过它具体”看到“发生了并行化。

接下来，需要建立管道函数返回的对象类型。对于我们正在转换的这个过程，对象类型只是其“输出“的一种结构化定义。在这个例子中，对象类型类似于 T2：

```
ops$tkyte-ORA10G> CREATE OR REPLACE TYPE t2_type
  2 AS OBJECT (
  3             id number,
  4             text varchar2(30),
  5             session_id number
  6     )
  7 /
Type created.


ops$tkyte-ORA10G> create or replace type t2_tab_type
  2     as table of t2_type
  3     /
Type created.
```

现在来看管道函数，这只是重写了原来的 PROCESS_DATA 过程。现在这个过程是一个生成行的函数。它接收数据作为输入，并在一个引用游标（ref cursor）中处理。这个函数

返回一个 T2_TAB_TYPE，这就是我们刚才创建的对象类型。这是一个 PARALLEL_ENABLED（启用子并行）的管道函数。在此使用了分区（partition）子句，这就告诉 Oracle：“以任何最合适的方式划分或分解数据。我们不需要对数据的顺序做任何假设 “

还 可以在引用游标中对特定列使用散列或区间分区。这就要使用一个强类型化的引用游标，从而使编译器知道哪些列是可用的。根据所提供的散列，散列分区会向各个 并行执行服务器发送同样多的行来进行处理。区间分区则基于分区键向各个并行执行服务器发送不重叠的数据区间。例如，如果在 ID 上执行区间分区，每个并行执行服务器可能会得到区间 1…1000、1001…20000、20001…30000 等（该区间中的 ID 值 。

在此，我们只想划分数据。数据如何划分对于我们的处理并不重要，所以定义如下：

```
ops$tkyte-ORA10G> create or replace

  2 function parallel_pipelined( l_cursor in sys_refcursor )

  3 return t2_tab_type

  4 pipelined

  5 parallel_enable ( partition l_cursor by any )
```

我们想查看哪些行由哪个并行执行服务器处理，所以声明一个局部变量 L_SESSION_ID，并从 V$MYSTAT 对其进行初始化：

```
  6

  7 is

  8     l_session_id number;

  9     l_rec t1%rowtype;

 10 begin

 11 select sid into l_session_id

 12 from v$mystat

 13 where rownum =1;
```

现在可以处理数据了。在此要获取一行（多多行，因为这里当然可以使用 BULK COLLECT 来实现对引用游标的批量处理），执行复杂的处理，并输出。引用游标处理完成数时，将关闭这个游标，并返回：

```
 14 loop
```

```
15          fetch l_cursor into l_rec;
16          exit when l_cursor%notfound;
17          -- complex process here
18          pipe row(t2_type(l_rec.id,l_rec.text,l_session_id));
19 end loop;
20 close l_cursor;
21 return;
22 end;
23 /

Function created.
```

这样就创建了函数。我们准备并行地处理数据，让 Oracle 根据可用的资源来确定最合适的并行度：

```
ops$tkyte-ORA10G> alter session enable parallel dml;

Session altered.



ops$tkyte-ORA10G> insert /*+ append */
  2 into t2(id,text,session_id)
  3           select *
  4           from table(parallel_pipelined
  5           (CURSOR(select /*+ parallel(t1) */ *
  6                   from t1 )
  7           ))
  8 /

48250 rows created.
```

```
ops$tkyte-ORA10G> commit;

Commit complete.
```

为了查看这里发生了什么，可以查询新插入的数据，并按 SESSION_ID 分组，先来看使用了多少个并行执行服务器，再看每个并行执行服务器处理了多少行：

```
ops$tkyte-ORA10G> select session_id, count(*)
  2 from t2
  3 group by session_id;

SESSION_ID  COUNT(*)
----------  ----------
       241        8040
       246        8045
       253        8042
       254        8042
       258        8040
       260        8041

6 rows selected.
```

显然，对于这个并行操作的 SELECT 部分，我们使用了 6 个并行执行服务器，每个并行执行服务器处理了大约 8,040 记录。

可以看到，Oracle 对我们的过程进行了并行化，但是为此需要从头重写原先的过程。从最初的实现（串行过程）到现在（可以并行执行的过程），这实在是一条漫长的道理。所以，尽管 Oracle 可以并行地处理我们的例程，但是并非所有例程都能编写为完成并行化。如果大规模重写你的过程不太可行，你可能会对下一种实现感兴趣：DIY 并行化。

14.6.2 DIY 并行化

假设与上一节一样，也有同样的一个简单的串行过程。重写这个过程实现让我们难以承受，这实在太费劲了，但是我们又希望并行地执行这个过程。该怎么做呢？我通常采用的方法是：使用 rowid 区间将表划分为多个不重叠的区间（但要覆盖整个表）。

从概念上讲，这与 Oracle 执行并行查询的做法很相似。考虑一个全表扫描，Oracle 处理全表扫描时会提出某种方法将这个表划分为多个“小表“，每个小表分别由一个并行执行服务器处理。我们要用 rowid 区间来做同样的事情。在较早的版本中，Oracle 的并行实现实际上使用了 rowid 区间。

我们要使用一个 1,000,000 含更多 BIG_TABLE，因为这种技术最适用于有大量区段的大表，创建 rowid 区间所用的方法取决于区段边界。使用的区段越多，数据分布就越好。所以，创建 1,000,000 行的 BIG_TABLE 之后，我们将如下创建 T2：

```
big_table-ORA10G> create table t2

  2 as

  3 select object_id id, object_name text, 0 session_id

  4 from big_table

  5 where 1=0;

Table created.
```

这里将使用数据库内置的任务队列来并行处理我们的过程。我们将调度 2 数目的任务。每个认为都是对我们的过程稍加修改，只处理某个给定 rowid 区间中的行。

注意　在 Oracle 10g 中，这种简单的工作可以使用调度工具完成，但是为了使这个例子与 9i 兼容，这里使用了任务队列。

为了高效地支持任务队列，我们要使用一个参数表，向任务传递输入：

```
big_table-ORA10G> create table job_parms

  2 ( job number primary key,

  3 lo_rid rowid,

  4 hi_rid rowid

  5 )

  6 /

Table created.
```

这样一来，我们就能只向过程传入任务 ID，2 再查询表来得到所要处理的 rowid 区间。再来看我们的过程。粗体显示的代码是新增的：

```
big_table-ORA10G> create or replace

  2 procedure serial( p_job in number )
```

```
3 is
4       l_rec job_parms%rowtype;
5 begin
6       select * into l_rec
7       from job_parms
8       where job = p_job;
9
10        for x in ( select object_id id, object_name text
11                   from big_table
12                  where rowid between l_rec.lo_rid
13                    and l_rec.hi_rid )
14        loop
15              -- complex process here
16              insert into t2 (id, text, session_id )
17              values ( x.id, x.text, p_job );
18        end loop;
19
20        delete from job_parms where job = p_job;
21        commit;
22 end;
23 /
Procedure created.
```

可以看到，改动并不大。大多数新增的代码只是用于得到输入和要处理的 rowid 区间。对逻辑只有一处修改：就是在第 12 行和第 13 行增加了谓词。

下面来调度任务。在此使用一个相当复杂的查询，它使用分析函数来划分表，在这种情况下，第 19～26 行上的最内层查询将数据划分为 8 组。第 22 行上的第一个求和用于计算块数的累计总计；第 23 行上的第二个求和得出总块数。如果将累计总和整除所需的“块大小“（chunk size，在这里就是总计大小除以 8），可以创建覆盖相同数量数据的文件/块组。第 8～28 行上的查询按 GRP 找出最高和最低文件号以及块号，并返回不同的 grp。这就建立了输入，可以把这个输入发送给 DBMS_ROWID，创建 Oracle 所要的 rowid。得到该输出，并使用 DBMS_JOB 提交一个任务来处理这个 rowid 区间：

```
big_table-ORA10G> declare
  2          l_job number;
  3 begin
  4          for x in (
  5                    select dbms_rowid.rowid_create
                             ( 1, data_object_id, lo_fno, lo_block, 0 ) min_rid,
  6                          dbms_rowid.rowid_create
                             ( 1, data_object_id, hi_fno, hi_block, 10000 ) max_rid
  7                    from (
  8                          select distinct grp,
  9                                 first_value(relative_fno)
                                      over (partition by grp order by relative_fno, block_id
 10                                 rows between unbounded preceding and unbounded following)
lo_fno,
 11                                 first_value(block_id )
                                      over (partition by grp order by relative_fno, block_id
 12                                 rows between unbounded preceding and unbounded following)
lo_block,
 13                                 last_value(relative_fno)
                                      over (partition by grp order by relative_fno, block_id
 14                                 rows between unbounded preceding and unbounded following)
```

```
hi_fno,
   15                         last_value(block_id+blocks-1)
                                over (partition by grp order by relative_fno, block_id
   16                         rows between unbounded preceding and unbounded following)
hi_block,
   17                         sum(blocks) over (partition by grp) sum_blocks
   18                  from (
   19                         select relative_fno,
   20                                block_id,
   21                                blocks,
   22                                   trunc( (sum(blocks) over (order by relative_fno,
block_id)-0.01) /
   23                                (sum(blocks) over ()/8) ) grp
   24                         from dba_extents
   25                         where segment_name = upper('BIG_TABLE')
   26                                and owner = user order by block_id
   27                  )
   28           ),
   29           (select data_object_id
                       from user_objects where object_name = upper('BIG_TABLE') )
   30           )
   31           loop
   32                  dbms_job.submit( l_job, 'serial(JOB);' );
   33                  insert into job_parms(job, lo_rid, hi_rid)
```

```
  34                              values ( l_job, x.min_rid, x.max_rid );

  35                  end loop;

  36          end;

  37 /

PL/SQL procedure successfully completed.
```

这个 PL/SQL 块会为我们调度 8 个任务（如果由于区段或空间大小不足，以至于无法将表划分为 8 个部分，调度的任务可能会少一些）。如下可以看到调度了多少任务以及这些任务的输入：

```
big_table-ORA10G> select * from job_parms;



      JOB       LO_RID                          HI_RID

---------       ------------------              ------------------

      172       AAAT7tAAEAAAAkpAAA        AAAT7tAAEAAABQICcQ

      173       AAAT7tAAEAAABQJAAA        AAAT7tAAEAAABwICcQ

      174       AAAT7tAAEAAABwJAAA        AAAT7tAAEAAACUICcQ

      175       AAAT7tAAEAAACUJAAA        AAAT7tAAEAAAC0ICcQ

      176       AAAT7tAAEAAAC0JAAA       AAAT7tAAEAAADMICcQ

      177       AAAT7tAAEAAADaJAAA        AAAT7tAAEAAAD6ICcQ

      178       AAAT7tAAEAAAD6JAAA        AAAT7tAAEAAAEaICcQ

      179       AAAT7tAAEAAAEaJAAA        AAAT7tAAEAAAF4ICcQ

8 rows selected.



big_table-ORA10G> commit;

Commit complete.
```

一旦提交就会开始处理我们的任务。在参数文件在我们将 JOB_QUEUE_PROCESSES 设置为 8，所有 8 个任务都开始运行，并很快完成。结果如下：

```
big_table-ORA10G> select session_id, count(*)
  2 from t2
  3 group by session_id;


SESSION_ID  COUNT(*)
----------  ----------
       172      130055
       173      130978
       174      130925
       175      129863
       176      106154
       177      140772
       178      140778
       179       90475

8 rows selected.
```

在这里，虽然不如 Oracle 内置并行化分布那么均匀，但是已经很不错了。如果还记得，前面你曾经看到过每个并行执行服务器处理了多少行，使用内置并行化时，行数彼此之间非常接近（只相差 1 或 2）。在此，有一个任务只处理了 90,475 行，而另一个任务处理了多达 140,778 行，另外的大多数任务都处理了大约 130,000 行。

不过，假设你不想使用 rowid 处理，允许是因为查询不像 SELECT * FROM T 那么简单，而涉及联结和其他构造，这使得使用 rowid 是不切实际的。此时就可以使用某个表的主键。例如，假设你下把这个 BIG_TABLE 划分为 10 个部分，按主键并发地处理。使用内置 NTILE 分析函数就能很轻松地做到。这个过程相当简单：

```
big_table-ORA10G> select nt, min(id), max(id), count(*)
  2 from (
  3         select id, ntile(10) over (order by id) nt
  4         from big_table
```

```
    5 )
    6 group by nt;

        NT      MIN(ID)     MAX(ID) COUNT(*)
        ----------  ----------  ----------  ----------big_table-ORA10G> select nt, min(id), max(id), count(*)
2 from (
3 select id, ntile(10) over (order by id) nt
4 from big_table
5 )
6 group by nt;
NT MIN(ID) MAX(ID) COUNT(*)
---------- ---------- ---------- ----------
1 1 100000 100000
2 100001 200000 100000
3 200001 300000 100000
4 300001 400000 100000
5 400001 500000 100000
6 500001 600000 100000
7 600001 700000 100000
8 700001 800000 100000
9 800001 900000 100000
10 900001 1000000 100000
10 rows selected.
```

```
         1          1     100000     100000
         2     100001     200000     100000
         3     200001     300000     100000
         4     300001     400000     100000
         5     400001     500000     100000
         6     500001     600000     100000
         7     600001     700000     100000
         8     700001     800000     100000
         9     800001     900000     100000
        10     900001    1000000     100000

10 rows selected.
```

现在就有了 10 个互不重叠的主键区间，大小都一样，可以使用这些区间来实现如前所示的 DBMS_JOB 技术，从而并行化你的过程。

14.7 小结

在这一章中，我们分析了 Oracle 中并行执行的概念。首先我给出了一个类比，帮助你了解可以在哪里应用并行执行，以及何时应用；具体来讲，如果有长时间运行的语句或过程，而且有充足的可用资源，就可以使用并行执行。

然后我们介绍了 Oracle 如何使用并行化。首先讨论了并行查询，并说明了 Oracle 如何将大型串行操作（如全面扫描）分解为可以并发运行的小部分。接下来讨论了并行 DML（PDML），并指出了与之相关的一大堆限制。

接下来，我们介绍了并行操作的闪光点：并行 DDL。并行 DDL 是 DBA 和开发人员可用的一种工具，可以很快地执行大规模的维护操作（通常在非高峰时间而且有足够可用的资源时执行）。随后简要地谈到 Oracle 提供了并行恢复，然后开始讨论过程并行化。在此我们了解了两种实现过程并行化的技术：一种是由 Oracle 完成，另一种是我们自己来完成。

如果从头开始设计一个过程，可以考虑将其设计为允许 Oracle 为我们完成并行化，这样随着将来资源的增减，就能很容易地改变并行度。不过，如果已经有一些代码，需要很快地“修正“为能够并行执行，则可以采用 DIY 并行化，对此我们也介绍了两种技术：使用 rowid 区间和使用主键区间，它们都使用 DBMS_JOB 在后台为我们并行地执行任务。

在这一章中，我们将讨论数据的加载和卸载，换句话说，也就是如何将数据放入 Oracle 数据库以及如何从 Oracle 取出数据。这一章的重点是介绍以下批量数据加载工具：

- SQL*Loader（读作“sequel loader”）：这仍是加载数据的主流方法。
- 外部表（external table）：这是 Oracle9i 及以上版本中的一个新特性，允许访问操作系统文件，就好像它们是数据库表一样，在 Oracle 10g 及以上版本中，甚至还允许抽取表中的数据创建操作系统文件。

在数据加载方面，我们将介绍两个技术：

- 平面文件卸载（flat file unload）：平面文件卸载是定制开发的实现，但是所提供的结果却能移植到其他类型的系统（甚至电子表格）。
- 数据泵卸载（data dump unload）：数据泵是 Oracle 专业的一种二进制格式，可以通过数据泵工具和外部表访问。

15.1 SQL*Loader

SQL*Loader（SQLLDR）是 Oracle 的高速批量数据加载工具。这是一个非常有用的工具，可用于多种平面文件格式向 Oralce 数据库中加载数据。SQLLDR 可以在极短的时间内加载数量庞大的数据。它有两种操作模式：

- 传统路径：（conventional path）：SQLLDR 会利用 SQL 插入为我们加载数据。
- 直接路径(direct path)：采用这种模式，SQLLDR 不使用 SQL；而是直接格式化数据库块。

利用直接路径加载，你能从一个平面文件读数据，并将其直接写至格式化的数据库块，而绕过整个 SQL 引擎和 undo 生成，同时还可能避开 redo 生成。要在一个没有任何数据的数据库中充分加载数据，最快的方法就是采用并行直接路径加载。

我们不会介绍 SQLLDR 的方方面面。要想全面了解有关的详细内容，请参考 Oracle Utilities 手册，其中有 7 章专门介绍 Oracle 10g 的 DQLLDR。在这个手册中，居然用 7 章介绍 SQLLDR，这一点确实很引入注意，因为其他的各个实用程序（如 DBVERIFY、DBNEWID 和 LogMiner）只占了一章或不到一章的篇幅。要了解 SQLLDR 的语法和所有选项，建议你参考 Oracle Utilities 手册，因为本书这一章只是要回答参考手册中没有提到的“如何……？”等问题。

需要指出，在 Oracle 8.1.6 Release 1 及以上版本中，Oracle 调用接口（Oracle Call Interface，OCI）允许使用 C 编写你自己的直接路径加载工具。如果你要执行的操作在 SQLLDR 中做不到，或者如果需要 SQLLDR 与你的应用无缝集成，Oracle OCI 就很有用。SQLLDR 是一个命令行工具（也就是说，这是一个单独的程序）。它并非一个 API，例如，不能“从 PL/SQL 调用”。

如果不带任何输入地从命令行执行 SQLLDR，它会提供以下帮助：

```
[tkyte@desktop tkyte]$ sqlldr

SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:32:28 2005
```

Usage: SQLLDR keyword=value [,keyword=value,...]

Valid Keywords:

userid -- ORACLE username/password
control -- control file name
log -- log file name
bad -- bad file name
data -- data file name
discard -- discard file name
discardmax -- number of discards to allow (Default all)
skip -- number of logical records to skip (Default 0)
load -- number of logical records to load (Default all)
errors -- number of errors to allow (Default 50)
rows -- number of rows in conventional path bind array or
between direct path data saves
(Default: Conventional path 64, Direct path all)
bindsize -- size of conventional path bind array in bytes (Default 256000)
silent -- suppress messages during run
(header,feedback,errors,discards,partitions)
direct -- use direct path (Default FALSE)
parfile -- parameter file: name of file that contains parameter specifications

```
        parallel --            do parallel load (Default FALSE)

              file --            file to allocate extents from

skip_unusable_indexes -- disallow/allow unusable indexes or index partitions

                (Default FALSE)

skip_index_maintenance -- do not maintain indexes, mark affected indexes as unusable

                (Default FALSE)

commit_discontinued -- commit loaded rows when load is discontinued (Default FALSE)

readsize -- size of read buffer (Default 1048576)

external_table -- use external table for load; NOT_USED, GENERATE_ONLY, EXECUTE

                (Default NOT_USED)

columnarrayrows -- number of rows for direct path column array (Default 5000)

streamsize -- size of direct path stream buffer in bytes (Default 256000)

multithreading -- use multithreading in direct path

resumable -- enable or disable resumable for current session (Default FALSE)

resumable_name -- text string to help identify resumable statement

resumable_timeout -- wait time (in seconds) for RESUMABLE (Default 7200)

date_cache -- size (in entries) of date conversion cache (Default 1000)

...
```

我并不打算解释每个参数技术上的含义，而只是建议你阅读 Oracle Utilities 手册，特别是 Oracle 10g Utilities Guide 中的第 7 章和 Oracle9i Utilities Guide 中的第 4 章。本书这一章会展示其中为数不多的一些参数的用法。

要使用 SQLLDR，需要有一个控制文件（control file）。 控制文件中包含描述输入数据的信息（如输入数据的布局、数据类型等），另外还包含有关目标表的信息。控制文件甚至还可以包含要加载的数据。在下面的例子 中，我们将一步一步地建立一个简单的控制文件，并对这些命令提供必须的解释（注意，代码左边加括号的数并不是控制文件中的一部分，显示这些数只是为了便于 引用）。

```
(1) LOAD DATA
```

```
(2) INFILE *

(3) INTO TABLE DEPT

(4) FIELDS TERMINATED BY ','

(5) (DEPTNO, DNAME, LOC )

(6) BEGINDATA

(7) 10,Sales,Virginia

(8) 20,Accounting,Virginia

(9) 30,Consulting,Virginia

(10) 40,Finance,Virginia
```

- LOAD DATA （1）：这会告诉 SQLLDR 要做什么（在这个例子中，则指示要加载数据）。SQLLDR 还可以执行 CONTINUE_LOAD，也就是继续加载。只有在继续一个多表直接路径加载时才能使用后面这个选项。

- INFILE * （2）：这会告诉 SQLLDR 所要加载的数据实际上包含在控制文件本身上，如第 6～10 行所示。也可以指定包含数据的另一个文件的文件名。如果愿意，可以使用一个命令行参数覆盖这个 INFILE 语句。要当心，命令行选项总会涵盖控制文件设置。

- INTO TABLE DEPT （3）：这会告诉 SQLLDR 要把数据加载到哪个表中（在这个例子中，数据要加载到 DEPT 表中）。

- FIELDS TERMINATED BY ‘,’（4）：这会告诉 SQLLDR 数据的形式应该是用逗号分隔的值。为 SQLLDR 描述输入数据的方式有数十种；这只是其中较为常用的方法之一。

- (DEPTNO, DNAME, LOC) （5）：这会告诉 SQLLDR 所要加载的列、这些列在输入数据中的顺序以及数据类型。这是指输入流中数据的数据类型，而不是数据库中的数据类型。在这个例子中，列的数据类型默认为 CHAR(255)，这已经足够了。

- BEGINDATA （6）：这会告诉 SQLLDR 你已经完成对输入数据的描述，后面的行（第 7～10 行）是要加载到 DEPT 表的具体数据。

这个控制文件采用了最简单、最常用的格式之一：将定界数据加载到一个表。这一章还会看一些复杂的例子，不过可以从这个简单的控制文件入手，这是一个不错的起点。要使用这个控制文件（名为 demo1.ctl），只需创建一个空的 DEPT 表：

```
ops$tkyte@ORA10G> create table dept

  2 ( deptno number(2) constraint dept_pk primary key,
```

```
  3 dname varchar2(14),

  4 loc varchar2(13)

  5 )

  6 /

Table created.
```

并运行以下命令：

```
[tkyte@desktop tkyte]$ sqlldr userid=/ control=demo1.ctl

SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:59:06 2005

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

Commit point reached - logical record count 4
```

如果表非空，就会收到一个错误消息：

```
SQLLDR-601: For INSERT option, table must be empty. Error on table DEPT
```

这是因为，这个控制文件中几乎所有选项都取默认值，而默认的加载选项是 INSERT（而不是 APPEND、TRUNCATE 或 REPLACE）。要执行 INSERT， SQLLDR 就认为表为空。如果想向 DEPT 表中增加记录，可以指定加载选项为 APPEND；或者，为了替换 DEPT 表中的数据，可以使用 REPLACE 或 TRUNCATE。REPLACE 使用一种传统 DELETE 语句；因此，如果要加载的表中已经包含许多记录，这个操作可能执行得很慢。TRUNCATE 则不同，它使用 TRUNCATE SQL 命令，通常会更快地执行，因为它不必物理地删除每一行。

每个加载都会生成一个日志文件。以上这个简单加载的日志文件如下：

```
SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:59:06 2005

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



Control File: demo1.ctl

Data File: demo1.ctl

Bad File: demo1.bad

Discard File: none specified

(Allow all discards)
```

Number to load: ALL

Number to skip: 0

Errors allowed: 50

Bind array: 64 rows, maximum of 256000 bytes

Continuation: none specified

Path used: Conventional

Table DEPT, loaded from every logical record.

Insert option in effect for this table: INSERT

Column Name	Position	Len	Term	Encl	Datatype
DEPTNO	FIRST	*	,		CHARACTER
DNAME	NEXT	*	,		CHARACTER
LOC	NEXT	*	,		CHARACTER

Table DEPT:

4 Rows successfully loaded.

0 Rows not loaded due to data errors.

0 Rows not loaded because all WHEN clauses were failed.

0 Rows not loaded because all fields were null.

Space allocated for bind array: 49536 bytes(64 rows)

Read buffer bytes: 1048576

```
Total logical records skipped: 0

Total logical records read: 4

Total logical records rejected: 0

Total logical records discarded: 0



Run began on Sat Jul 16 10:59:06 2005

Run ended on Sat Jul 16 10:59:06 2005



Elapsed time was: 00:00:00.15

CPU time was: 00:00:00.03
```

日志文件会告诉我们关于加载的很多方面，从中可以看到我们所用的选项（默认或默认选项）；可以看到读取了多少记录，加载了多少记录等。日志文件指定了所有 BAD 文件和 DISCARD 文件的位置，甚至还会告诉我们加载用了多长时间。每个日志文件对于验证加载是否成功至关重要，另外对于诊断错误也很有意义。如果所加载的数据导致 SQL 错误（也就是说，输入数据是“坏的“，并在 BAD 文件中建立了记录），这些错误就会记录在这个日志文件中。日志文件中的信息很大程度上不言自明，所以这里不再花时间做过多的解释。

15.1.1 用 SQLLDR 加载数据的 FAQ

现在来回答 Oracle 数据库中关于用 SQLLDR 加载数据最常问到的一些问题。

1. 如何加载定界数据？

定价数据（delimited data）即用某个特定字符分隔的数据，可以用引号括起，这是当前平面文件最常见的数据格式。在大型机上，定长、固定格式的文件可能是最可识别的文件格式，但是在 UNIX 和 NT 上，定界文件才是“标准“。在这一节中，我们将分析用于加载定界数据的常用选项。

对于定界数据，最常用的格式是逗号分隔值（comma-separated values，CSV） 格式。采用这种文件格式，数据中的每个字段与下一个字段用一个逗号分隔。文本串可以用引号括起，这样就允许串本身包含逗号。如果串还必须包含引号，一般约 定是使用两个引号（在下面的代码 中，我们将使用 ““而不是 ‘’）。要加载定界数据，相应的典型控制文件与前面第一个例子很相似，但是 FIELDS TERMINATED BY 子句通常如下指定：

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

它指定用逗号分隔数据字段，每个字段可以用双引号括起。如果我们把这个控制文件的最后部分修改如下：

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'

(DEPTNO, DNAME, LOC )

BEGINDATA

10,Sales,"Virginia,USA"

20,Accounting,"Va, ""USA"""

30,Consulting,Virginia

40,Finance,Virginia
```

使用这个控制文件运行 SQLLDR 时，结果如下：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC

---------- --------------  -------------

        10 Sales           Virginia,USA

        20 Accounting      Va, "USA"

        30 Consulting      Virginia

        40 Finance         Virginia
```

要特别注意以下几点：

- 部门 10 中的 Virginia.USA：这是因为输入数据是“Virginia.USA”。输入数据字段必须包括在引号里才能保留数据中的逗号。否则，数据中的这个逗号会被认为是字段结束标记，这样就会只加载 Virginia，而没有 USA 文本。
- Va,”USA”：这是因为输入数据是“Va,””USA”””。对于引号括起的串，SQLLDR 会把其中“的两次出现计为一次出现。要加载一个包含可选包围字符（enclosure character）的串，必须保证这个包围字符出现两次。

另一种常用的格式是制表符定界数据（tag-delimited data），这是用制表符分隔而不是逗号分割的数据。有两种方法使用 TERMINATED BY 子句来加载这种数据：

- TERMINATED BY X’09’（使用十六进制格式的制表符；采用 ASCII 时，制表符为 9）
- TERMINATED BY WHITESPACE

这两种方法在实现上有很大差异，下面将会说明。还是用前面的 DEPT 表，我们将使用以下控制文件加载这个表：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY WHITESPACE

(DEPTNO, DNAME, LOC)

BEGINDATA

10 Sales Virginia
```

从字面上不太容易看得出来，不过要知道，在这里各部分数据之间都有两个制表符。这里的数据行实际上是：

```
10\t\tSales\t\tVirginia
```

在此\t 是普通可识别的制表符转义字符。使用这个控制文件时（包含如前所示的 TERMINATED BY WHITESPACE），表 DEPT 中的数据将是：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC

---------- --------------      -------------

        10 Sales               Virginia
```

TERMINATED BY WHITESPACE 会解析这个串，查找空白符（制表符、空格和换行符）的第一次出现，然后继续查找，直至找到下一个非空白符。因此，解析数据时，DEPTNO 会赋给 10，后面的两个制表符被认为是空白符，Sales 会赋给 DNAME 等。

另一方面，如果要使用 FIELDS TERMINATED BY X’09’，如以下控制文件所示，这里稍做修改：

```
...

FIELDS TERMINATED BY X'09'

(DEPTNO, DNAME, LOC )

...
```

可以看到 DEPT 中加载了以下数据：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC

---------- --------------     -------------

        10                    Sales
```

在此，一旦 SQLLDR 遇到一个制表符，就会输出一个值。因此，将 10 赋给 DEPTNO，DNAME 得到了 NULL，因为在第一个制表符和制表符的下一次出现之间没有数据。Sales 赋给了 LOC。

这是 TERMINATED BY WHITESPACE 和 TERMINATED BY <character>的有意行为。至于使用哪一种方法更合适，这取决于输入数据以及你要如何解释输入数据。

最后，加载这样的定界数据时，很可能想逃过输入记录中的某些列。例如，你可能加载字段 1、3 和 5，而跳过第 2 列和第 4 列。为此，SQLLDR 提供了 FILLER 关键字。这允许你映射一个输入记录中的一列，但不把它放在数据库中。例如，给定 DEPT 表以及先前的最高一个控制文件，可以修改这个控制文件，使用 FILLER 关键字正确地加载数据（跳过制表符）：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY x'09'

(DEPTNO, dummy1 filler, DNAME, dummy2 filler, LOC)

BEGINDATA

10    Sales        Virginia
```

所得到的表 DEPT 现在如下所示：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC

---------- --------------     -------------

        10 Sales              Virginia
```

2. 如何加载固定格式数据？

通常会有一个有某个外部系统生成的平面文件，而且这是一个定长文件，其中包含着

固定位置的数据（positional data）。例如，NAME 字段位于第 1～10 字节，ADDRESS 字段位于地 11～35 字节等。我们将介绍 SQLLDR 如何为我们导入这种数据。

这种定宽的固定位置数据是最适合 SQLLDR 加载的数据格式。要加载这种数据，使用 SQLLDR 是最快的处理方法，因为解析输入数据流相当容易。SQLLDR 会在数据记录中存储固定字节的偏移量和长度，因此抽取某个给定字段相当简单。如果要加载大量数据，将其转换为一种固定位置格式通常是最好的办法。当然，定宽文件也有一个缺点，它比简单的定界文件格式可能要大得多。

要加载定宽的固定位置数据，将会在控制文件中使用 POSITION 关键字，例如：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

( DEPTNO position(1:2),

DNAME position(3:16),

LOC position(17:29)

)

BEGINDATA

10Accounting Virginia,USA
```

这个控制文件没有使用 FIELDS TERMINATED BY 子句；而是使用了 POSITION 来告诉 SQLLDR 字段从哪里开始，到哪里结束。关于 POSITION 子句有意思的是，我们可以使用重叠的位置，可以在记录中来回反复。例如，如果如下修改 DEPT 表：

```
ops$tkyte@ORA10G> alter table dept add entire_line varchar(29);

Table altered.
```

并使用以下控制文件：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE
```

```
( DEPTNO position(1:2),

DNAME position(3:16),

LOC position(17:29),

ENTIRE_LINE position(1:29)

)

BEGINDATA

10Accounting Virginia,USA
```

字段 ENTIRE_LINE 定义的 POSITION(1:29)。这会从所有 29 字节的输入数据中抽取出这个字段的数据，而其他字段都是输入数据的子串。这个控制文件的输出如下：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC             ENTIRE_LINE

---------- -------------- -------------   ----------------------------

        10 Accounting     Virginia,USA    10Accounting Virginia,USA
```

使用 POSITION 时，可以使用相对偏移量，也可以使用绝对偏移量。在前面的例子中使用了绝对偏移量，我们明确地指示了字段从哪里开始，到哪里结束。也可以把前面的控制文件写作：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

( DEPTNO position(1:2),

DNAME position(*:16),

LOC position(*:29),

ENTIRE_LINE position(1:29)

)
```

```
BEGINDATA

10Accounting Virginia,USA
```

指示控制文件得出上一个字段在哪里结束。因此，在这种情况下，（:16）与(3:16)是一样的。注意，控制文件中可以混合使用相对位置和绝对位置。另外。使用*表示法时，可以把它与偏移量相加。例如，如果 DNAME 从 DEPTNO 结束之后的 2 个字节处开始，可以使用(*+2:16)。在这个例子中，其作用就相当于使用(5:16)。

POSITION 子句中的结束位置必须是数据结束的绝对列位置。有时，可能指定每个字段的长度更为容易，特别是如果这些字段是连续的（就像前面的例子一样）。采用这种方式，只需告诉 SQLLDR：记录从第 1 个字节开始，然后指定每个字段的长度就行了。这样我们就可以免于计算记录中的开始和结束偏移量，这个计算有时可能很困难。为此，可以不指定结束位置，而是指定定长记录中各个字段的长度，如下：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

( DEPTNO position(1) char(2),

DNAME position(*) char(14),

LOC position(*) char(13),

ENTIRE_LINE position(1) char(29)

)

BEGINDATA

10Accounting Virginia,USA
```

在此只需告诉 SQLLDR 第一个字段从哪里开始及其长度。后面的每个字段都从上一个字段结束处开始，并具有指定的长度。直至最后一个字段才需要再次指定位置，因为这个字段又要从记录起始处开始。

3. 如何加载日期？

使用 SQLLDR 加载日期相当简单，但是看起来这个方面经常导致混淆。你只需在控制文件中使用DATE 数据类型，并指定要使用的日期掩码。这个日期掩码与数据库中 TO_CHAR 和 TO_DATE 中使用的日期掩码是一样的。SQLLDR 会向数据应用这个日期掩码，并为你完成加载。

例如，如果再把 DEPT 表修改如下：

```
ops$tkyte@ORA10G> alter table dept add last_updated date;

Table altered.
```

可以用以下控制文件加载它：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

(DEPTNO,

DNAME,

LOC,

LAST_UPDATED date 'dd/mm/yyyy'

)

BEGINDATA

10,Sales,Virginia,1/5/2000

20,Accounting,Virginia,21/6/1999

30,Consulting,Virginia,5/1/2000

40,Finance,Virginia,15/3/2001
```

所得到的 DEPT 表如下所示：

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC            LAST_UPDA

---------- -------------- -------------- ---------

        10 Sales          Virginia       01-MAY-00

        20 Accounting     Virginia       21-JUN-99
```

```
30 Consulting    Virginia        05-JAN-00

40 Finance       Virginia        15-MAR-01
```

就这么简单。只需在控制文件中应用格式，SQLLDR 就会为我们完成日期转换。在某些情况想，可能使用一个更强大的 SQL 函数更为合适。例如，如果你的输入文件包含多种不同格式的日期：有些有时间分量，有些没有；有些采用 DD-MON-YYYY 格式；有些格式为 DD/MM/YYYY；等等。

在下一节中你会了解到如何在 SQLLDR 中使用函数来解决这些问题。

4. 如果使用函数加载数据？

在这一节中，我们将介绍加载数据时如何使用函数。

一旦你了解了 SQLLDR 如何构建其 INSERT 语句，在 SQLLDR 中使用函数就很容易了。要在 SQLLDR 脚本中向某个字段应用一个函数，只需块这个函数增加到控制文件中（用两个引号括起）。例如，假设有前面的 DEPT 表，你想确保所加载的数据都是大写的。可以使用以下控制文件来加载：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

LAST_UPDATED date 'dd/mm/yyyy'

)

BEGINDATA

10,Sales,Virginia,1/5/2000

20,Accounting,Virginia,21/6/1999

30,Consulting,Virginia,5/1/2000

40,Finance,Virginia,15/3/2001
```

数据库中得到的数据如下：

```
ops$tkyte@ORA10G> select * from dept;

DEPTNO     DNAME              LOC                ENTIRE_LINE                LAST_UPD
A

        ------     --------------          -------------          ----------------------------     ---------

            10     SALES                VIRGINIA                                    01-MAY-0
0

            20     ACCOUNTING      VIRGINIA                                    21-JUN-99

            30     CONSULTING      VIRGINIA                                    05-JAN-00

            40     FINANCE             VIRGINIA                                    15-MAR-0
1
```

可以注意到，只需向一个绑定变量应用 UPPER 函数就可以很容易地将数据变为大写。要注意，SQL 函数可以引用任何列，而不论将函数实际上应用于哪个列。这说明，一个列可以是对两个或更多其他列应用一个函数的结果。例如，如果你想加载 ENTIRE_LINE 列，可以使用 SQL 连接运算符。不过，这种情况下这样做稍有些麻烦。现在，输入数据集中有 4 个数据元素。如果只是向控制文件中如下字符 ENTIRE_LINE：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

LAST_UPDATED date 'dd/mm/yyyy',

ENTIRE_LINE ":deptno||:dname||:loc||:last_updated"

)
```

```
BEGINDATA

10,Sales,Virginia,1/5/2000

20,Accounting,Virginia,21/6/1999

30,Consulting,Virginia,5/1/2000

40,Finance,Virginia,15/3/2001
```

就会看到，日志文件中对于每个输入记录出现以下错误：

```
Record 1: Rejected - Error on table DEPT, column ENTIRE_LINE.

Column not found before end of logical record (use TRAILING NULLCOLS)
```

在此，SQLLDR 告诉你：没等处理完所有列，记录中就没有数据了。这种情况下，解决方案很简单。实际上，SQLLDR 甚至已经告诉了我们该怎么做：这就是使用 TRAILING NULLCOLS。这样一来，如果输入记录中不存在某一列的数据，SQLLDR 就会为该列绑定一个 NULL 值。在这种情况下，增加 TRAILING NULLCOLS 会导致绑定变量：ENTIRE_LINE 成为 NULL。所以再尝试这个控制文件：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

LAST_UPDATED date 'dd/mm/yyyy',

ENTIRE_LINE ":deptno||:dname||:loc||:last_updated"

)

BEGINDATA
```

```
10,Sales,Virginia,1/5/2000

20,Accounting,Virginia,21/6/1999

30,Consulting,Virginia,5/1/2000

40,Finance,Virginia,15/3/2001
```

现在表中的数据如下：

```
ops$tkyte@ORA10G> select * from dept;

DEPTNO DNAME          LOC          ENTIRE_LINE                 LAST_UP
DA

------ -------------- ------------ --------------------------- ---------

    10 SALES          VIRGINIA     10SalesVirginia1            /5/2000
01-MAY-00

    20 ACCOUNTING     VIRGINIA     20AccountingVirginia21      /6/1999
21-JUN-99

    30 CONSULTING     VIRGINIA     30ConsultingVirginia5       /1/2000
05-JAN-00

    40 FINANCE        VIRGINIA     40FinanceVirginia15         /3/2001
15-MAR-01
```

之所以可以这样做，原因在于 SQLLDR 构建其 INSERT 语句的做法。SQLLDR 会查看前面的控制文件，并看到控制文件中的 DEPTNO、DNAME、LOC、LAST_UPDATED 和 ENTIRE_LINE 这几列。它会根据这些列建立 5 个绑定变量。通常，如果没有任何函数，所建立的 INSERT 语句就是：

```
INSERT INTO DEPT ( DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE )

VALUES ( :DEPTNO, :DNAME, :LOC, :LAST_UPDATED, :ENTIRE_LINE );
```

然后再解析输入流，将值赋给相应的绑定变量，然后执行语句。如果使用函数，SQLLDR 会把这些函数结合到 INSERT 语句中。在上一个例子中，SQLLDR 建立的 INSERT 语句如下所示：

```
INSERT INTO T (DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE)

VALUES ( :DEPTNO, upper(:dname), upper(:loc), :last_updated,

              :deptno||:dname||:loc||:last_updated );
```

然后再做好准备，把输入绑定到这个语句，再执行语句。所以，SQL 中能做的事情都

可以结合到 SQLLDR 脚本中。由于 SQL 中增加了 CASE 语句，所以这样做不仅功能极为强大，而且相当容易。例如，假设你的输入文件有以下格式的日期：

- HH24:MI:SS：只有一个时间；日期默认为 SYSDATE。
- DD/MM/YYYY：只有一个日期；时间默认为午夜 0 点。
- HH24:MI:SS DD/MM/YYYY：日期和时间都要显式提供。

可以使用如下的一个控制文件：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

LAST_UPDATED

"case

when length(:last_updated) > 9

then to_date(:last_updated,'hh24:mi:ss dd/mm/yyyy')

when instr(:last_updated,':') > 0

then to_date(:last_updated,'hh24:mi:ss')

else to_date(:last_updated,'dd/mm/yyyy')

end"

)

BEGINDATA
```

```
10,Sales,Virginia,12:03:03 17/10/2005

20,Accounting,Virginia,02:23:54

30,Consulting,Virginia,01:24:00 21/10/2005

40,Finance,Virginia,17/8/2005
```

可以得到以下结果：

```
ops$tkyte@ORA10G> alter session set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';

Session altered.



ops$tkyte@ORA10G> select deptno, dname, loc, last_updated

2 from dept;



DEPTNO     DNAME              LOC              LAST_UPDATED

    ----------     --------------          -------------          --------------------

          10     SALES              VIRGINIA         17-oct-2005 12:03:03

          20     ACCOUNTING     VIRGINIA         01-jul-2005 02:23:54

          30     CONSULTING     VIRGINIA         21-oct-2005 01:24:00

          40     FINANCE            VIRGINIA         17-aug-2005 00:00:00
```

现在会向输入字符串应用 3 个日期格式中的一个（注意，这里不再加载一个 DATE；而只是加载一个串）。CASE 函数会查看串的长度和内容，从而确定应该使用哪一个掩码。

有意思的是，你可以编写自己的函数来由 SQLLDR 调用。这直接应验了可以从 SQL 调用 PL/SQL。

5. 如何加载有内嵌换行符的数据？

过去，如果要加载可能包含换行符的自由格式的数据，这对于 SQLLDR 来说很成问题。换行符是 SQLLDR 的默认行结束符，过去对此也提出了一些解决方法，但是灵活性都不够。幸运的是，在 Oracle 8.1.6 及以后版本中，我们有了一些新的选择。要加载内嵌有换行符的数据，现在的选择如下：

- 加载数据，其中用非换行符的其他字符来表示换行符（例如，在文本中应该出现换行符的位置上放上串\n），并在加载时使用一个 SQL 函数用一个 CHR(10)替换

该文本。

- 在 INFILE 指令上使用 FIX 属性，加载一个定长平面文件。
- 在 INFILE 指令上使用 VAR 属性，加载一个定宽文件，在该文件使用的格式中，每一行的前几个字节指定了这一行的长度（字节数）。
- 在 INFILE 指令上使用 STR 属性，加载一个变宽文件，其中用某个字符序列来表示行结束符，而不是用换行符来表示。

后面的几个小节将分别介绍这些方法。

- **使用一个非换行符的字符**

如果你能对如何生成输入数据加以控制，这就是一种很容易的方法。如果创建数据文件时能很容易地转换数据，这种方法就能奏效。其思想是，就数据加载到数据库时对数据应用一个 SQL 函数，用某个字符串来替换换行符。下面向 DEPT 表再增加另一个列：

```
ops$tkyte@ORA10G> alter table dept add comments varchar2(4000);

Table altered.
```

我们将使用这一列来加载文本。下面是一个有内联数据的示例控制文件：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

COMMENTS "replace(:comments,'\\n',chr(10))"

)

BEGINDATA

10,Sales,Virginia,This is the Sales\nOffice in Virginia
```

```
20,Accounting,Virginia,This is the Accounting\nOffice in Virginia

30,Consulting,Virginia,This is the Consulting\nOffice in Virginia

40,Finance,Virginia,This is the Finance\nOffice in Virginia
```

注意，调用中必须使用\\n 来替换换行符，而不只是\n。这是因为\n 会被 SQLLDR 识别为一个换行符，而且 SQLLDR 会把它转换为一个换行符，而不是一个两字符的串。利用以上控制文件执行 SQLLDR 时，DEPT 表中将加载以下数据：

```
ops$tkyte@ORA10G> select deptno, dname, comments from dept;

DEPTNO     DNAME              COMMENTS

   ----------     --------------          -------------------------

         10     SALES              This is the Sales

                                    Office in Virginia

         20     ACCOUNTING     This is the Accounting

                                    Office in Virginia

         30     CONSULTING     This is the Consulting

                                    Office in Virginia

         40     FINANCE            This is the Finance

                                    Office in Virginia
```

□ **使用 IFX 属性**

另一种可用的方法是使用 FIX 属性。如果使用这种方法，输入数据必须出现在定长记录中。每个记录与输入数据集中所有其他记录的长度都相同，即有相同的字节数。对于固定位置的数据，使用 FIX 属性就特别适合。这些文件通常是定长输入文件。使用自由格式的定界数据时，则不太可能是一个定长文件，因为这些文件通常是变长的（这正是定界文件的关键：每一行不会不必要地过长）。

使用 FIX 属性时，必须使用一个 INFILE 子句，因为 FIX 属性是 INFILE 的一个选项。另外，如果使用这个选项，数据必须在外部存储，而并非存储在控制文件本身。因此，假设有定长的输入记录，可以使用如下的一个控制文件：

```
LOAD DATA

INFILE demo.dat "fix 80"
```

```
INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

COMMENTS

)
```

这个文件指定了一个输入数据文件（domo.dat），这个文件中每个记录有 80 字 节，这包括尾部的换行符（每个记录最后可能有换行符，也可能没有）。在这种情况下，输入数据文件中的换行符并不是特殊字符。这只是要加载（或不加载）的另 一个字符而已。要知道：记录的最后如果有换行符，它会成为这个记录的一部分。为了充分理解这一点，我们需要一个实用程序将文件的内容转储在屏幕上，以便我 们看到文件中到底有什么。使用 UNIX（或任何 Linux 版本），利用 od 就很容易做到，这个程序可以将文件以八进制（和其他格式）转储到屏幕上。我们将使用下面的 demo.dat 文件。注意以下输入中的第一列实际上是八进制，所以第 2 行上的数字 0000012 是一个八进制数，不是十进制数 10.由此我们可以知道所查看的文件中有哪些字节。我对这个输出进行了格式化，使得每行显示 10 个字符（使用-w10），所以 0、12、24 和 36 实际上就是 0、10、20 和 30。

```
[tkyte@desktop tkyte]$ od -c -w10 -v demo.dat

0000000       1   0   ,   S   a   l   e   s   ,   V

0000012       i   r   g   i   n   i   a   ,   T   h

0000024       i   s   i   s   t   h   e

0000036       S   a   l   e   s  \n   O   f   f   i

0000050       c   e   i   n   V   i   r   g

0000062       i   n   i   a

0000074

0000106
```

0000120 2 0 , A c c o u n t

0000132 i n g , V i r g i n

0000144 i a , T h i s i s

0000156 t h e A c c o u

0000170 n t i n g \n O f f i

0000202 c e i n V i r g

0000214 i n i a

0000226

0000240 3 0 , C o n s u l t

0000252 i n g , V i r g i n

0000264 i a , T h i s i s

0000276 t h e C o n s u

0000310 l t i n g \n O f f i

0000322 c e i n V i r g

0000334 i n i a

0000346

0000360 4 0 , F i n a n c e

0000372 , V i r g i n i a ,

0000404 T h i s i s t h

0000416 e F i n a n c e \n

0000430 O f f i c e i n

0000442 V i r g i n i a

0000454

```
0000466

0000500

[tkyte@desktop tkyte]$
```

注意，在这个输入文件中，并没有用换行符（\n）来指示 SQLLDRE 记录在哪里结束；这里的换行符只是要加载的数据而已。SQLLDR 使用 FIX 宽度（80 字节）来得出要读取多少数据。实际上，如果查看输入数据，可以看到，输入文件中提供给 SQLLDR 的记录甚至并非以\n 结束。部门 20 的记录之前的字符是一个空格，而不是换行符。

既然我们知道了每个记录的长度为 80 字节，现在就可以用前面有 FIX80 子句的控制文件来加载这些数据了。完成加载后，可以看到以下结果：

```
ops$tkyte@ORA10G> select "" || comments || "" comments from dept;

COMMENTS

--------------------------------------------------------------------------------

"This is the Sales

Office in Virginia "

"This is the Accounting

Office in Virginia "

"This is the Consulting

Office in Virginia "

"This is the Finance

Office in Virginia "
```

你可能需要“截断“这个数据，因为尾部的空白符会保留。可以在控制文件中使用 TRIM 内置 SQL 函数来完成截断。

如果你恰好同时在使用 Windows 和 UNIX，能你很“幸运“，在此需要提醒一句：这两个平台上的行结束标记是不同的。在 UNIX 上，行结束标记就是\n（SQL 中的 CHR(10)）。在 Windows NT 上，行结束标记却是\r\n（SQL 中的 CHR(13)||CHR(10)）。一般来讲，如果使用 FIX 方法，就要确保是在同构平台上创建和加载文件（UNIX 上创建，UNIX 上加载；或者 Windows 上创建，Windows 上加载）。

- **使用 VAR 属性**

要加载有内嵌换行符的数据，另一种方法是使用 VAR 属性。使用这种格式时，每个记录必须以某个固定的字节数开始，这表示这个记录的总长度。通过使用这种格式，可以加载

包含内嵌换行符的变长记录，但是每个记录的开始处必须有一个记录长度字段。因此，如果使用如下的一个控制文件：

```
LOAD DATA

INFILE demo.dat "var 3"

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS

(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

COMMENTS

)
```

VAR 3 指出每个输入记录的前 3 个字节是输入记录的长度。如果取以下数据文件：

```
[tkyte@desktop tkyte]$ cat demo.dat

05510,Sales,Virginia,This is the Sales

Office in Virginia

06520,Accounting,Virginia,This is the Accounting

Office in Virginia

06530,Consulting,Virginia,This is the Consulting

Office in Virginia

05940,Finance,Virginia,This is the Finance

Office in Virginia

[tkyte@desktop tkyte]$
```

可以使用该控制文件来加载。在我们的输入数据文件中有 4 行数据。第一行从 055 开始，这

说明接下来 55 字节是第一个输入记录。这 55 字节包括单词 Virginia 后的结束换行符。下一行从 065 开始。这一行有 65 字节的文本，依此类推。使用这种格式数据文件，可以很容易地加载有内嵌换行符的数据。

同样，如果你在使用 UNIX 和 Windows（前面的例子都在 UNIX 上完成，其中换行符只是一个字符长），就必须调整每个记录的长度字段。在 Windows 上，前例.dat 文件中的长度字段应该是 56、66、66 和 60.

□ **使用 STR 属性**

要加载有内嵌换行符的数据，这可能是最灵活的一种方法。通过使用 STR 属性，可以指定一个新的行结束符（或字符序列）。 就能创建一个输入数据文件，其中每一行的最后有某个特殊字符、换行符不再有“特殊“含义。

我更喜欢使用字符序列，通常会使用某个特殊标记，然后再加一个换行符。这样，在一个文本编辑器或某个实用程序中查看输入数据时，就能很容易地看到行结束符，因为每个记录的最后仍然有一个换行符。STR 属性以十六进制指定，要得到所需的具体十六进制串，最容易的方法是使用 SQL 和 UTL_RAW 来生成十六进制串。例如，假设使用的是 UNIX 平台，行结束标记是 CHR(10)（换行），我们的特殊标记字符是一个管道符号（|），则可以写为：

```
ops$tkyte@ORA10G> select utl_raw.cast_to_raw( '|'||chr(10) ) from dual;



UTL_RAW.CAST_TO_RAW('|'||CHR(10))

--------------------------------------------------------------------------------

7C0A
```

由此可知，在 UNIX 上我们需要使用的 STR 是 X’7C0A’。

注意 在 Windows 上，要使用 UTL_RAW.CAST_TO_RAW(‘|”||chr(13)||chr(10))。

为了使用这个方法，要有以下控制文件：

```
LOAD DATA

INFILE demo.dat "str X'7C0A'"

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ','

TRAILING NULLCOLS
```

```
(DEPTNO,

DNAME "upper(:dname)",

LOC "upper(:loc)",

COMMENTS

)
```

因此，如果输入数据如下：

```
[tkyte@desktop tkyte]$ cat demo.dat

10,Sales,Virginia,This is the Sales

Office in Virginia|

20,Accounting,Virginia,This is the Accounting

Office in Virginia|

30,Consulting,Virginia,This is the Consulting

Office in Virginia|

40,Finance,Virginia,This is the Finance

Office in Virginia|

[tkyte@desktop tkyte]$
```

其中，数据文件中的每个记录都以|\n 结束，前面的控制文件就会正确地加载这些数据。

□ **内嵌换行符小结**

关于加载有内嵌换行符的数据，这一节讨论了至少 4 种方法。在后面的“平面文件卸载“一节中，我们还将看到会使用这里的一种技术，可以在一个通用卸载实用程序使用 STR 属性来避免与文本中换行符有关的问题。

另外要注意一个问题，我先前已经多次提到，Windows（包括各种版本）上的文本文件可能以\r\n（ASCII 13+ASCII 10，回车/换行）结束。\r 是记录的一部分，控制文件必须适应这一点。具体地将，FIX 和 VAR 中的字节数已经 STR 使用的串必须有所调整。例如，如果取先前的某个.dat 文件（目前其中只包含\n），并使用一个 ASCII 传输工具（默认）将其通过 FTP 传输到 Windows，将各个\n 将转换为\r\n。原来 UNIX 中能工作的控制文件现在却不能加载数据了。这一点你必须当心，建立控制文件时一定要有所考虑。

6. 如果加载 LOB？

现在来考虑在 LOB 的一些方法。这不是一个 LONG 或 LONG RAW 字段，而是更可取

的数据类型 BLOB 和 CLOB。这些数据类型是 Oracle 8.0 及以后版本中引入的，如第 12 章所述，与遗留的 LONG 和 LONG RAW 类型相比，它们支持更丰富的接口/功能集。

我们将分析两种加载这些字段的方法：SQLLDR 和 PL/SQL。除此之外，还可以采用另外一些方法，如 Java 流、Pro*C 和 OCI。我们将首先介绍使用 PL/SQL 加载 LOB 的方法，然后介绍如何使用 SQLLDR 加载 LOB。

□ **通过 PL/SQL 加载 LOB**

DBMS_LOB 包的入口点为 LoadFromFile、LoadBLOBFromFile 和 LoadCLOBFromFile。通过这些过程，我们可以使用一个 BFILE（用于读取操作系统文件）来填充数据库中的 BLOB 或 CLOB。LoadFromFile 和 LoadBLOBFromFile 例程之间没有显著的差别，只不过后者会返回一些 OUT 参数，指示已经向 BLOB 列中加载了多少数据。不过，LoadCLOBFromFile 例程还提供了一个突出的特性：字符集转换。如果你还记得，第 12 章中曾讨论过 Oracle 数据库的某些国家语言支持（NLS）特性，还介绍过字符集的重要性。使用 LoadCLOBFromFile 时，我们可以告诉数据库：这个文件将以另外某种字符集（不同于数据库正在使用的字符集）来加载，而且要执行必要的字符集转换。例如，可能有一个 UTF8 兼容的数据库，但是收到的要加载的文件却以 WE8ISO8859P1 字符集编码，或反之。利用这个函数就能成功地加载这些文件。

注意 DBMS_LOB 包中可以过程的全部细节及其完整的输入和输出集，请参考 Oraccle9i Oracle Supplied Packages Guide 和 Oracle 10g Oracle PL/SQL Packages and Types Reference。

要使用这些过程，需要在数据库中创建一个 DIRECTORY 对象。这个对象允许我们创建并打开 BFILE（BFILE 指向文件系统上数据库服务器能访问的一个现有文件）。最后一句话中提到：“数据库服务器能访问的……“，这是使用 PL/SQL 加载 LOB 时一个要点。DBMS_LOB 包完全在服务器中执行。它只能看到服务器能看到的文件系统。特别是，如果你通过网络访问 Oracle，DBMS_LOB 包将无法看到你的本地文件系统。

所以，我们需要先在数据库中创建一个 DIRECTORY 对象。这是一个很简单的过程。我们将为这个例子创建两个目录（注意，这些例子都在 UNIX 环境中执行；你要针对你的操作系统，使用适合的语法来引用目录）：

```
ops$tkyte@ORA10G> create or replace directory dir1 as '/tmp/';

Directory created.

ops$tkyte@ORA10G> create or replace directory "dir2" as '/tmp/';

Directory created.
```

注意 Oracle DIRECTORY 对象是逻辑目录，这说明，它们是指向操作系统中现有物理目录的指针。CREATE DIRECTORY 命令并不是具体在文件系统中创建一个目录，这个操作（物理创建目录）必须单独执行。

执行这个操作的用户要有 CREATE ANY DIRECTORY 权限。我们之所以要创建两个目录，这是为了展示一个与 DIRECTORY 对象有关的常见问题，即大小写问题（大写字符还是小写字符）。Oracle 创建第一个目录 DIR1 时，它会以大写存储对象名，因为这是默认设

置。在使用 dir2 的第二个例子中，它创建的 DIRECTORY 对象保留了名字中原来使用的大小写。稍后使用 BFILE 对象时将说明这一点的重要性。

下面，我们希望将一些数据加载到 BLOB 或 CLOB 中。对此，方法非常简单，例如：

```
ops$tkyte@ORA10G> create table demo
2 ( id int primary key,
3 theClob clob
4 )
5 /
Table created.


ops$tkyte@ORA10G> host echo 'Hello World!' > /tmp/test.txt
ops$tkyte@ORA10G> declare
  2              l_clob clob;
  3              l_bfile bfile;
  4          begin
  5              insert into demo values ( 1, empty_clob() )
  6              returning theclob into l_clob;
  7
  8              l_bfile := bfilename( 'DIR1', 'test.txt' );
  9              dbms_lob.fileopen( l_bfile );
 10
 11              dbms_lob.loadfromfile( l_clob, l_bfile,
 12              dbms_lob.getlength( l_bfile ) );
 13
```

```
  14                dbms_lob.fileclose( l_bfile );

  15        end;

  16        /

PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select dbms_lob.getlength(theClob), theClob from demo

  2 /


DBMS_LOB.GETLENGTH(THECLOB)     THECLOB

                          ---------------------------        ---------------

                                                  13        Hello World!
```

通过分析前面的代码，可见：

- 在第 5 行和第 6 行上，我们在表中创建了一行，将 CLOB 设置为一个 EMPTY_CLOB()，并从一个调用获取其值。除了临时 LOB 外，其余的 LOB 都“住“在数据库中，如果没有指向一个临时 LOB 的指针，或者指向一个已经在数据库中的 LOB，将无法写至 LOB 变量。EMPTY_CLOB()不是一个 NULL CLOB；而是指向一个空结构的合法指针（非 NULL）。它还有一个作用，可以得到一个 LOB 定位器，指向已锁定行中的数据。如果要选择这个值，而没有锁定底层的行，写数据就会失败，因为 LOB 在写之前必须锁定（不同于其他结构化数据）。通过插入一行，当然我们也就锁定了这一行。如果我们要修改一个现有的行而不是插入新行，则可以使用 SELECT FOR UPDATE 来获取和锁定这一行。
- 在第 8 行上，我们创建了一个 BFILE 对象。注意，这里 DIR1 用的是大写，稍后就会看到，这是一个键。这是因为我们向 BFILENAME()传入了一个对象的名称，而不是对象本身。因此，必须确保这个名称与 Oracle 所存储的对象名称大小写匹配。
- 第 9 行打开了 LOB。以便读取。
- 在第 11 行和第 12 行上，我们将操作系统文件/tmp/test.txt 的完整内容加载到刚插入的 LOB 定位器。这里使用 DBMS_LOB.GETLENGTH()告诉 LOADFROMFILE()例程要加载多少字节的 BFILE（这里就是要加载全部字节）。
- 最后，在第 14 行我们关闭了所打开的 BFILE，CLOB 已加载。

如果前例中试图使用 dir1 而不是 DIR1，可能会遇到以下错误：

```
ops$tkyte@ORA10G> declare
```

```
...
6           returning theclob into l_clob;
7
8       l_bfile := bfilename( 'dir1', 'test.txt' );
9           dbms_lob.fileopen( l_bfile );
...
15 end;
16 /
declare
*
ERROR at line 1:
ORA-22285: non-existent directory or file for FILEOPEN operation
ORA-06512: at "SYS.DBMS_LOB", line 523
ORA-06512: at line 9
```

这是因为目录 dir1 并不存在，只有目录 DIR1。如果想使用混合有大小写的目录名，在创建这样的目录四时应该使用带引号的标识符，就像我们创建 dir2 时一样。这样你就能编写如下所示的代码：

```
ops$tkyte@ORA10G> declare
2           l_clob clob;
3           l_bfile bfile;
4 begin
5           insert into demo values ( 1, empty_clob() )
6           returning theclob into l_clob;
7
```

```
  8          l_bfile := bfilename( 'dir2', 'test.txt' );
  9          dbms_lob.fileopen( l_bfile );
 10
 11          dbms_lob.loadfromfile( l_clob, l_bfile,
 12          dbms_lob.getlength( l_bfile ) );
 13
 14          dbms_lob.fileclose( l_bfile );
 15 end;
 16 /

PL/SQL procedure successfully completed.
```

除了从文件例程加载外，还有其他一些方法，利用这些方法也可以使用 PL/SQL 填充 LOB。如果你想加载整个文件，就可以使用 DBMS_LOB 及其提供的例程，这是到目前为止最容易的方法。如果需要在加载文件的同时处理文件的内容，还可以在 BFILE 上使用 DBMS_LOB.READ 来读取数据。如果读取的数据实际上是文本，而不是 RAW，那么使用 UTL_RAW.CAST_TO_VARCHAR2 会很方便。然后你可以使用 DBMS_LOB.WRITE 或 WRITEAPPEND 将数据放入一个 CLOB 或 BLOB。

□ **通过 SQLLDR 加载 LOB 数据**

现在我们来分析如何通过 SQLLDR 向 LOB 加载数据。对此方法不止一种，但是我们主要讨论两种最常用的方法：

□ 数据“内联“在其他数据中。

□ 数据外联存储（在外部存储），输入数据包含一个文件名，指示该行要加载的数据在哪个文件中。在 SQLLDR 术语中，这也称为二级数据文件（secondary data file，SDF）。

先从内联数据谈起。

加载内联的 LOB 数据。这些 LOB 通常内嵌有换行符和其他特殊字符。因此，往往会使用“如何加载有内嵌换行符的数据？“一节中详细讨论的 4 种方法之一来加载这种数据。下面先来修改 DEPT 表，使 COMMENTS 列是一个 CLOB 而不是一个大的 VARCHAR2 字段：

```
ops$tkyte@ORA10G> truncate table dept;

Table truncated.
```

```
ops$tkyte@ORA10G> alter table dept drop column comments;

Table altered.



ops$tkyte@ORA10G> alter table dept add comments clob;

Table altered.
```

例如，假设有一个数据文件（demo.dat），它有以下内容：

```
10, Sales,Virginia,This is the Sales

Office in Virginia|

20,Accounting,Virginia,This is the Accounting

Office in Virginia|

30,Consulting,Virginia,This is the Consulting

Office in Virginia|

40,Finance,Virginia,"This is the Finance

Office in Virginia, it has embedded commas and is

much longer than the other comments field. If you

feel the need to add double quoted text in here like

this: ""You will need to double up those quotes!"" to

preserve them in the string. This field keeps going for up to

1000000 bytes (because of the control file definition I used)

or until we hit the magic end of record marker,

the | followed by an end of line - it is right here ->"|
```

每个记录最后都是一个管道符号（|），后面是行结束标记。部门 40 的文本比其他部门的文本长得多，有多个换行符、内嵌的引号以及逗号。给定这个数据文件，可以创建一个如下的控制文件：

```
LOAD DATA

INFILE demo.dat "str X'7C0A'"

INTO TABLE DEPT

REPLACE

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'

TRAILING NULLCOLS

   (DEPTNO,

   DNAME "upper(:dname)",

   LOC "upper(:loc)",

   COMMENTS char(1000000)

   )
```

注意　这个例子在 UNIX 上执行，UNIX 平台上行结束标记长度为 1 字节，因此可以使用以上控制文件中的 STR 设置。在 Windows 上，STR 设置则必须是’7C0D0A’。

要加载这个数据文件，我们在 COMMENTS 列上指定了 CHAR(1000000)，因为 SQLLDR 默认所有人们字段都为 CHAR(255)。CHAR(1000000)则允许 SQLLDR 处理多达 1,000,000 字节的输入文本。可以把这个长度值设置为大于输入文件中任何可能文本块的大小。通过查看所加载的数据，可以看到以下结果：

```
ops$tkyte@ORA10G> select comments from dept;

COMMENTS

--------------------------------------------------------------------------------

This is the Consulting

Office in Virginia



This is the Finance

Office in Virginia, it has embedded commas and is

much longer than the other comments field. If you
```

```
feel the need to add double quoted text in here like

this: "You will need to double up those quotes!" to

preserve them in the string. This field keeps going for up to

1000000 bytes or until we hit the magic end of record marker,

the | followed by an end of line - it is right here ->



This is the Sales

Office in Virginia



This is the Accounting

Office in Virginia
```

这里可以观察到：原来重复两次的引号不再重复。SQLLDR 去除了在此放置的额外的引号。

加载外联的 LOB 数据。可能要把包含有一些文件名的数据文件加载在 LOB 中，而不是让 LOB 数据与结构化数据混在一起，这种情况很常见。这提供了更大程度的灵活性，因为提供给 SQLLDR 的数据文件不必使用上述的 4 种方法之一来避开输入数据中的内嵌换行符问题，而这种情况在大量的文本或二进制数据中会频繁出现。SQLLDR 称这种额外的数据文件为 LOBFILE。

SQLLDR 还可以支持加载结构化数据文件（指向另外单独一个数据文件）。我们可能告诉 SQLLDR 如何从另外这个文件分析 LOB 数据，这样就可以加载其中的一部分作为结构化数据中的每一行。我认为这种模式的用途很有限（到目前为止，我自己还从来没有见过哪里用到这种方法），在此也不做过多的讨论。SQLLDR 把这种外部引用的文件称为复杂二级数据文件（complex secondary data file）。

LOBFILE 是一种相对简单的数据文件，旨在简化 LOB 加载。在 LOBFILE 中，没有记录的概念，因此换行符不会成为问题，正是这个性质使得 LOBFILE 与主要数据文件有所区别。在 LOBFILE 中，数据总是采用以下某种格式：

- 定长字段（例如，从 LOBFILE 加载字节 100 到 1,000）
- 定界字段（以某个字符结束，或者用某个字符括起）
- 长度/值对，这是一个变长字段

其中最常见的类型是定界字段，实际上就是以一个文件结束符（EOF）结束。一般来讲，可能有这样一个目录，其中包含你想加载到 LOB 列中的文件，每个文件都要完整地放

在一个 BLOB 中。此时，就可以使用带 TERMINATED BY EOF 子句的 LOBFILE 语句。

假设我们有一个目录，其中包含想要加载到数据库中的文件。我们想加载文件的 OWNER、文件的 TIME_STAMP、文件的 NAME 以及文件本身。要加载数据的表如下所示：

```
ops$tkyte@ORA10G> create table lob_demo

    2 ( owner varchar2(255),

    3 time_stamp date,

    4 filename varchar2(255),

    5 data blob

    6 )

    7 /

Table created.
```

在 UNIX 上使用一个简单的 ls －l 来捕获输出（或者在 Windows 上使用 dir/q/n），我们就能生成输入文件，并使用如下的一个控制文件加载（这里使用 UNIX 平台）：

```
LOAD DATA

INFILE *

REPLACE

INTO TABLE LOB_DEMO

    ( owner position(17:25),

    time_stamp position(44:55) date "Mon DD HH24:MI",

    filename position(57:100),

    data LOBFILE(filename) TERMINATED BY EOF

    )

BEGINDATA

-rw-r--r--     1 tkyte        tkyte          1220342 Jun 17 15:26 classes12.zip

-rw-rw-r--           1 tkyte        tkyte          10 Jul 16 16:38 foo.sql
```

```
-rw-rw-r--  1 tkyte  tkyte  751 Jul 16 16:36 t.ctl
-rw-rw-r--  1 tkyte  tkyte  491 Jul 16 16:38 testa.sql
-rw-rw-r--  1 tkyte  tkyte  283 Jul 16 16:38 testb.sql
-rw-rw-r--  1 tkyte  tkyte  231 Jul 16 16:38 test.sh
-rw-rw-r--  1 tkyte  tkyte  235 Apr 28 18:03 test.sql
-rw-rw-r--  1 tkyte  tkyte  1649 Jul 16 16:36 t.log
-rw-rw-r--  1 tkyte  tkyte  1292 Jul 16 16:38 uselast.sql
-rw-rw-r--  1 tkyte  tkyte  909 Jul 16 16:38 userbs.sql
```

现在，运行 SQLLDR 之后检查 LOB_DEMO 表的内容，会发现以下结果：

```
ops$tkyte@ORA10G> select owner, time_stamp, filename, dbms_lob.getlength(data)
  2 from lob_demo
  3 /

OWNER    TIME_STAM    FILENAME       DBMS_LOB.GETLENGTH(DATA)
--------  ---------  --------------  ------------------------
tkyte    17-JUN-05  classes12.zip 1220342
tkyte    16-JUL-05  foo.sql              10
tkyte    16-JUL-05  t.ctl                875
tkyte    16-JUL-05  testa.sql            491
tkyte    16-JUL-05  testb.sql            283
tkyte    16-JUL-05  test.sh              231
tkyte    28-APR-05  test.sql            235
tkyte    16-JUL-05  t.log                0
tkyte    16-JUL-05  uselast.sql         1292
```

```
tkyte          16-JUL-05       userbs.sql          909

10 rows selected.
```

这不光适用于 BLOB，也适用于 CLOB。以这种方式使用 SQLLDR 来加载文本文件的目录会很容易。

将 LOB 数据加载到对象列。既然知道了如何将数据加载到我们自己创建的一个简单表中，可能会发现，有时需要将数据加载到一个复杂的表中，其中可能有一个包含 LOB 的复杂对象类型（列）。使用图像功能时这种情况最为常见。图像功能使用一个复杂的对象类型 ORDSYS.ORDIMAGE 来实现。我们需要告诉 SQLLDR 如何向其中加载数据。

要把一个 LOB 加载到一个 ORDIMAGE 类型的列中，首先必须对 ORDIMAGE 类型的结构有所了解。在 SQL*Plus 中使用要加载的一个目标表以及该表上的 DESCRIBE，可以发现表中有一个名为 IMAGE 的 ORDSYS.ORDIMAGE 列，最终我们想在这一列中加载 IMAGE.SOURCE.LOCALDATA。只有安装并配置好 interMedia，项目的例子才能正常工作；否则，数据类型 ORDSYS.ORDIMAGE 将是一个未知类型：

```
ops$tkyte@ORA10G> create table image_load(
  2 id number,
  3 name varchar2(255),
  4 image ordsys.ordimage
  5 )
  6 /
Table created.


ops$tkyte@ORA10G> desc image_load
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
ID                                                 NUMBER
NAME                                               VARCHAR2(255)
IMAGE                                              ORDSYS.ORDIMAGE
```

```
ops$tkyte@ORA10G> desc ordsys.ordimage

Name                                    Null?    Type

----------------------------------------- --------  ----------------------------

SOURCE                                            ORDSYS.ORDSOURCE

HEIGHT                                             NUMBER(38)

WIDTH                                              NUMBER(38)

CONTENTLENGTH                                     NUMBER(38)

...

ops$tkyte@ORA10G> desc ordsys.ordsource

Name                                    Null?    Type

----------------------------------------- --------  ----------------------------

LOCALDATA                                          BLOB

SRCTYPE                                           VARCHAR2(4000)

SRCLOCATION                                       VARCHAR2(4000)

...
```

注意 可以在 SQL*Plus 中执行 SET DESC DEPTH ALL 或 SET DESC DEPTH <n>一次显示整个层次结构。由于 ORDSYS.ORDIMAGE 的输出可能有几项的篇幅，所以我打算逐部分地介绍。

加载这种数据的控制文件可能如下所示：

```
LOAD DATA

INFILE *

INTO TABLE image_load

REPLACE

FIELDS TERMINATED BY ','

( ID,
```

```
    NAME,
    file_name FILLER,
    IMAGE column object
    (
            SOURCE column object
            (
                LOCALDATA LOBFILE (file_name) TERMINATED BY EOF
              NULLIF file_name = 'NONE'
            )
    )
)
BEGINDATA
1,icons,icons.gif
```

这里我引入了两个新构造：

- COLUMN OBJECT：这会告诉 SQLLDR 这不是一个列名；而是列名的一部分。它不会映射到输入文件中的一个字段，只是用来构建正确的对象列引用，从而在加载中使用。在前面的文件中有两个列对象标记，其中一个（SOURCE）嵌入在另一个（SOURCE）嵌入在另一个（IMAGE）中。因此，根据我们的需要，要使用的列名是 IMAGE.SOURCE.LOCALDATA。注意，我们没有加载这两个对象类型的任何其他属性（例如，IMAGE.HEIGHT、IMAGE.CONTENTLENGTH 和 IMAGE.SOURCE.SRCTYPE）。稍后，我们将介绍如何填充这些属性。
- NULL IF FILE_NAME = ‘NONE’：这会告诉 SQLLDR，如果字段 FILE_NAME 包含单词 NONE，则向对象列中加载一个 NULL。

一旦已经加载了一个 interMedia 类型，通常需要使用 PL/SQL 对已经加载的数据进行后处理，以便 interMedia 能够处理该数据。例如，对于前面的数据，可能想运行以下代码来正确地为图像设置属性：

```
begin
    for c in ( select * from image_load ) loop
```

```
            c.image.setproperties;

        end loop;

end;

/
```

SETPROPERTIES 是 ORDSYS.ORDIMAGE 类型提供的对象方法，它处理图像本身，并用适当的值更新对象的其余属性。

7. 如何从存储过程调用 SQLLDR？

这个问题的答案很简单：这是办不到的。SQLLDR 不是一个 API：它不能调用。SQLLDR 是一个命令行程序。你完全可以用 Java 或 C 编写一个运行 SQLLDR 的外部过程，但是这与“调用”SQLLDR 是两码事。加载会在另一个会话中发生，它不受你的事务控制。另外，你必须解析所得到的日志文件来确定加载是否成功，以及成功的程度如何（也就是说，由于程序一个错误而导致加载终止之前已经加载了多少行）。我不建议从存储过程调用 SQLLDR。

过去，在 Oracle9i 之前你可以实现你自己的类 SQLLDR 过程。例如，可以有以下选择：

- 用 PL/SQL 编写一个微型 SQLLDR。它可以使用 BFILE 来读取二进制数据，或使用 UTL_FILE 读取文本数据来解析和加载。
- 用 Java 编写一个微型 SQLLDR。与基于 PL/SQL 的加载工具相比，这可能稍有点复杂，这样能利用许多可用的 Java 例程。
- 用 C 编写一个 SQLLDR，并作为一个外部过程来调用。

幸运的是，在 Oracle9i 及以后的版本中，我们可以使用外部表，这不仅能提供 SQLLDR 的几乎所有功能，另外，还可以做 SQLLDR 做不到的一些事情。这一章将介绍一个外部表的简单例子，其中将使用外部表来自动执行一个并行直接路径加载。稍后会用更多的篇幅来介绍这个内容。不过，在以上讨论的最后，下面对 SQLLDR 给出几个警告。

15.1.2 SQLLDR 警告

在这一节中，我们将讨论使用 SQLLDR 时要注意的几个问题。

1. TRUNCATE 的工作好像不太一样

SQLLDR 的 TRUNCATE 选项看上去好像与 SQL*Plus（或其他如何工具）中的 TRUNCATE 有所不同。SQLLDR 有一个假设，认为你会向表中重新加载同样数目的数据，因此会使用一种扩展形式的 TRUNCATE。具体地将，它会执行以下命令：

```
truncate table t reuse storage
```

REUSE STORAGE 选项并不释放已分配的区段，它只是将这些区段标记为“空闲空间”。如果这不是你想要的结果，就应当在执行 SQLLDR 之前先对表完成截除（truncate）。

2. SQLLDR 默认地使用 CHAR(255)

默认的输入字段长度为 255 字符。如果你的字段比这要长，就会将收到一个错误消息：

```
Record N: Rejected - Error on table T, column C.

Field in data file exceeds maximum length
```

这并不是说这个数据无法放在数据库列中；而是说，它指示 SQLLDR 希望有不少或等于 255 字节的输入数据，不过稍多一些也会接收。对此解决方案很简单，只需在控制文件中使用 CHAR(N)，在此 N 要足够大，能容纳输入文件中最长的字段长度。

3. 命令行会覆盖控制文件

SQLLDR 的许多选项既可以放在控制文件中，也可以在命令行上使用。例如，可以使用 INFILE FILENAME，也可以使用 SQLLDR…DATA=FILENAME。命令行会覆盖控制文件中的任何选项。不能指望一定会使用控制文件中的选项，因为执行 SQLLDR 的人可能会通过命令行覆盖这些选项。

15.1.3 SQLLDR 小结

在 这一节中，我们分析了加载数据的许多方面。在此介绍了每天可能遇到的一些典型问题：加载定界文件、加载定长文件、加载包含图像文件的一个目录，以及在输入 数据上使用函数来转换输入等。我们没有详细介绍如何使用直接路径加载工具来加载大量数据；而只是简单地提了一下。我们的目标是回答使用 SQLLDR 时最常出现而且影响面最广的问题。

15.2 外部表

外部表在 Oracle9i Release 1 中首次引入。简单地说，利用外部表，我们可以把一个操作系统文件当成是一个只读的数据库表。它们不是“实际”表的替代品，也并非用来取代实际表；而只是用作一种简化加载的工具（在 Oracle 10g 中还可以简化数据卸载）。

外部表特性首次出现时，我常常称之为“SQLLDR 的替代品”。大多数情况下，这种想法还是对的。既然如此，你可能会奇怪，为什么我们还要在前面花那么多时间来介绍 SQLLDR。原因是，SQLLDR 的使用历史已久，有相当多的遗留控制文件都以它为基础。SQLLDR 仍是一个常用的工具；这也是许多人很了解和一直在使用的工具。我们还处在从使用 SQLLDR 向外部表转变的中间阶段，所以 SQLLDR 仍很重要。

许多 DBA 可能不知道：他们对 SQLLDR 控制文件的了解在使用外部表时也完全可以用上。学习本章这部分的例子时，你会发现，外部表结合了许多 SQLLDR 语法和许多 SQLLDR 技术。

在以下 3 种情况下，应该选择 SQLLDR 而不是外部表：

- 必须通过网络加载数据，也就是说，输入文件不在数据库服务器上。外部表要求必须能在数据库服务器上访问输入文件，这是外部表的限制之一。
- 多个用户必须并发地使用相同的外部表来处理不同的输入文件。
- 必须使用 LOB 类型。外部表不支持 LOB。

在考虑这些情况的前提下，通常我强烈建议使用外部表来得到其扩展功能。SQLLDR 是一个相对简单的工具，能生成一个 INSERT 并加载数据。其使用 SQL 的能力仅限于逐行地调用 SQL 函数。外部表则不受此限制，可以充分利用所有 SQL 功能集来加载数据。在我

看来，外部表超越 SQLLDR 的关键功能特性如下：

- 可以使用复杂的 WHERE 条件有选择地加载数据。尽管 SQLLDR 有一个 WHEN 子句用来选择要加载的行，但是你只能使用 AND 表达式和执行相等性比较的表达式 在 WHEN 子句中不能使用区间（大于、小于），没有 OR 表达式，也没有 IS NULL 等。
- 能够合并（MERGE）数据。可以取一个填满数据的操作系统文件，并由它更新现有的数据库记录。
- 能执行高效的代码查找。可以将一个外部表联结到另一个数据库表作为加载过程的一部分。
- 使用 INSERT 更容易地执行多表插入。从 Oracle9i 开始，通过使用复杂的 WHEN 条件，可以用一个 INSERT 语句插入一个或多个表。尽管 SQLLDR 也可以加载到多个表中，但是相应的语法相当复杂。
- 对于开发新手来说，学习曲线更短。SQLLDR 作为“另外一种工具”，除了学习编程语言、开发工具、SQL 语句等之外，还要另外学习。但另一方面，只要开发人员懂 SQL，就可以直接将这些知识应用到批量数据加载，而不必学习一个新工具（SQLLDR）。

记住以上几点，下面来看如何使用外部表。

15.2.1 建立外部表

作为外部表的首次简单展示，首先再来运行前面的 SQLLDR 例子，向 DEPT 表中加载批量数据。你可能已经想不起来了，所以下面再次列出我们所用的简单控制文件，如下：

```
LOAD DATA

INFILE *

INTO TABLE DEPT

FIELDS TERMINATED BY ','

(DEPTNO, DNAME, LOC )

BEGINDATA

10,Sales,Virginia

20,Accounting,Virginia

30,Consulting,Virginia

40,Finance,Virginia
```

目前，作为起步，最容易的方法是使用这个现有的遗留控制文件来提供外部表的定义。

以下 SQLLDR 命令会为我们的外部表生成 CREATE TABLE 语句：

```
[tkyte@desktop tkyte]$ sqlldr / demo1.ctl external_table=generate_only

SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 17:34:51 2005

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

[tkyte@desktop tkyte]$
```

EXTERNAL_TABLE 参数有以下 3 个值：

- NOT_USED：其含义不言自明，这是默认值。
- EXECUTE：这个值说明 SQLLDR 不会生成并执行一个 SQL INSERT 语句；而是会创建一个外部表，并使用一个批量 SQL 语句来加载。
- GENERATE_ONLY：这个值使得 SQLLDR 并不具体加载任何数据，而只是会生成所执行的 SQL DDL 和 DML 语句，并放到它创建的日志文件中。

警告 DIRECT=TRUE 覆盖 EXTENAL_TABLE=GENERATE_ONLY。如果指定了 DIRECT=TRUE，则会加载数据，而不会生成外部表。

使用 GENERATE_ONLY 时，可以在 demo.log 文件中看到以下内容：

```
CREATE DIRECTORY statements needed for files

------------------------------------------------------------------------

CREATE DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000 AS '/home/tkyte'
```

我们可能会看到日志文件中有一个 CREATE DIRECTORY 语句（也可能看不到）。在生成外部表脚本期间，SQLLDR 连接到数据库，并查询数据字典来查看是否已经存在合适的目录。在这个例子中，由于没有合适的目录，所以 SQLLDR 为我们生成一个 CREATE DIRECTORY 语句。接下来，它为外部表生成 CREATE TABLE 语句：

```
CREATE TABLE statement for external table:

------------------------------------------------------------------------

CREATE TABLE "SYS_SQLLDR_X_EXT_DEPT"

(

        "DEPTNO" NUMBER(2),

        "DNAME" VARCHAR2(14),

        "LOC" VARCHAR2(13)
```

```
)
```

SQLLDR 已经登录到数据库；只有这样它才知道这个外部表定义中要用的具体数据类型（例如，DEPTNO 是一个 NUMBER(2)）。SQLLDR 根据数据字典来确定这些数据类型。接下来，我们来看这个外部表定义开始处的内容：

```
ORGANIZATION external

(

    TYPE oracle_loader

    DEFAULT DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
```

ORGANIZATION EXTERNAL 子句告诉 Oracle：这不是一个“正常”表。我们以前在第 10 章讨论 IOT 时曾经见过这个子句。目前有 3 种组织类型：HEAP 对应“正常”表，INDEX 对应 IOT，EXTERNAL 对应外部表。余下的文本则告诉 Oracle 有关这个外部表的更多信息。ORACEL_LOADER 类型是目前支持的两种类型之一（Oracle9i 中只支持这一种类型）。另一种类型是 ORACLE_DATAPUMP，这是 Oracle 10g 及以上版本中 Oracle 的专用数据泵格式。我们将在后面介绍数据卸载一节中讨论这个类型，这种格式不仅可以用于加载数据，也可以卸载数据。外部表可以用于创建一个数据泵格式文件，再读取这个文件。

下一部分是外部表的 ACCESS PARAMETERS 部分。在此我们告诉数据库如何处理这个输入文件。看到这个描述时，应该注意到，这与 SQLLDR 控制文件非常相似；这绝非偶然。在大多数情况下，SQLLDR 和外部表使用的语法都很相似。

```
ACCESS PARAMETERS

(

    RECORDS DELIMITED BY NEWLINE CHARACTERSET WE8ISO8859P1

    BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'demo1.bad'

    LOGFILE 'demo1.log_xt'

    READSIZE 1048576

    SKIP 7

    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY "" LDRTRIM

    REJECT ROWS WITH ALL NULL FIELDS

    (

        "DEPTNO" CHAR(255)
```

```
            TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',

            "DNAME" CHAR(255)

            TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',

            "LOC" CHAR(255)

            TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'

        )

)
```

这些访问参数显示了如何建立一个外部表，使之能像 SQLLDR 一样几乎以同样的方式处理文件：

- RECORDS：记录默认以换行符结束，SQLLDR 中的记录就是如此。
- BADFILE：在刚创建的目录中建立了一个坏文件（无法处理的记录都记录到这个文件中）。
- LOGFILE：在当前的工作目录中记录了一个等价于 SQLLDR 日志文件的日志文件。
- READSIZE：这是 Oracle 读取输入数据文件所用的默认缓冲区。在这里是 1MB。如果采用占用服务器模式，这个内存来自 PGA，如果采用共享服务器模式，则来自 SGA。它用于缓存输入数据文件中对应一个会话的信息（参见第 4 章，其中讨论了 PAG 和 SGA 内存）。如果你在使用共享服务器，要记住一点：内存从 SGA 分配。
- SKIP 7：在确定了应该跳过输入文件中的多少记录。你可能会问：“为什么有’skip 7’？“是这样，在这个例子中我们使用了 INFILE *；使用 SKIP 7 就是跳过控制文件本身来得到内嵌的数据。如果没有使用 INFILE *，就根本不会有 SKIP 子句。
- FIELDS TERMINATED BY：这与控制文件中的用法一样。不过，外部表没有增加 LDRTRIM，这代表 LoaDeR TRIM。这是一种截断模式，模拟了 SQLLDR 截断数据的默认做法。还有另外一些选项，包括 LRTRIM、LTRIM 和 RTRIM，表示左截断/右截断空白符；NOTRIM 表示保留所有前导/尾随的空白符。
- REJECT ROWS WITH ALL NULL FIELDS：这导致外部表会在坏文件中记录所有全空的行，而且不加载这些行。
- 列定义本身：这是有关所期望输入数据值的元数据。它们是所加载数据文件中的字符串，长度最多可达 255 个字符（SQLLDR 的默认大小），以逗号（,）结束，（还可以选择用引号括起来）。

注意 要全面了解使用外部表时可用的所有选项，请参考 Oracle Utilities Guide 手册。这本手册中有一节专门讨论外部表。Oracle SQL Reference Guide 手册提供了基本语法，但是不包括 ACCESS PARAMETERS 部分的细节。

最后，我们来看外部表定义中的 LOCATION 部分：

```
        location

        (

                'demo1.ctl'

        )

) REJECT LIMIT UNLIMITED
```

这告诉 Oracle 所加载文件的文件名，在这里就是 demo1.ctl，因为我们在原控制文件中使用了 INFILE *。控制文件中的下一条语句是默认的 INSERT，可以用于从外部表本身加载表：

```
INSERT statements used to load internal tables:

------------------------------------------------------------------------

INSERT /*+ append */ INTO DEPT

(

        DEPTNO,

        DNAME,

        LOC

)

SELECT

        "DEPTNO",

        "DNAME",

        "LOC"

FROM "SYS_SQLLDR_X_EXT_DEPT"
```

如果可能，这会执行一个逻辑上与直接路径加载等价的操作（假设可以遵循 APPEND 提示；如果存在触发器或外键约束，可能不允许发生直接路径操作）。

最后，在日志文件中，我们会看到一些语句，这些语句可以用于删除加载完成之后 SQLLDR 我我们创建的对象：

```
statements to cleanup objects created by previous statements:

-----------------------------------------------------------------------

DROP TABLE "SYS_SQLLDR_X_EXT_DEPT"

DROP DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
```

这就可以了。如果取这个日志文件，并在适当的地方插入/，使之成为一个合法的 SQL*Plus 脚本，就万事俱备了；也可能还不行，这取决于当前的权限。例如，假设我登录的模式有 CREATE ANY DIRECTORY 权限，或者能 READ 访问一个现有的目录，则可能观察到一个错误：

```
ops$tkyte@ORA10G> INSERT /*+ append */ INTO DEPT
  2 (
  3         DEPTNO,
  4         DNAME,
  5         LOC
  6 )
  7 SELECT
  8         "DEPTNO",
  9         "DNAME",
 10         "LOC"
 11 FROM "SYS_SQLLDR_X_EXT_DEPT"
 12 /
INSERT /*+ append */ INTO DEPT
*
ERROR at line 1:
ORA-29913: error in executing ODCIEXTTABLEOPEN callout
ORA-29400: data cartridge error
```

```
KUP-04063: unable to open log file demo1.log_xt

OS error Permission denied

ORA-06512: at "SYS.ORACLE_LOADER", line 19

ORA-06512: at line 1
```

初看上去好像不对劲。我作为 TKYTE 登录到操作系统，我登录的目录是/home/tkyte，而且我拥有这个目录，所以我当然能写这个目录（毕竟，我在那里创建了 SQLLDR 日志文件）。为什么会发生错误呢？发生了什么？现在的情况是，外部表代码在 Oracle 服务器软件中运行（在我的专业或共享服务器上）。试图读取输入数据文件的进程是 Oracle 软件所有者，而不是我的账户。试图创建日志文件的进程也是 Oracle 软件所有者，而不是我的账户。显然，Oracle 没有必要的权限来写我的目录，因此，对外部表的访问会失败。这一点很重要。要读一个表，运行数据库的账户（Oracle 软件所有者）必须能做下面的事情：

- 读我们指向的文件。在 UNIX 中，这说明 Oracle 软件所有者必须对指向这个文件的所有目录路径有读和执行权限。在 Windows 中，Oracle 软件所有者必须能读该文件。
- 写日志文件目录（即要写日志文件的目录；或者绕过日志文件生成，但是一般来讲，不建议这样做）。实际上，如果已经存在日志文件，Oracle 软件所有者必须能写现有的文件。
- 写所指定的如何坏文件，就像日志文件一样。

再回到前面的例子，以下命令使得 Oracle 能写我的目录：

```
ops$tkyte@ORA10G> host chmod a+rw .
```

警告 这个命令实际上会使所有人都能写我的目录!这只是一个演示;通常我会使用 Oracle 软件所有者自己拥有的一个特殊目录来做这个工作。

接下来，再运行 INSERT 语句：

```
ops$tkyte@ORA10G> l
  1 INSERT /*+ append */ INTO DEPT
  2 (
  3         DEPTNO,
  4         DNAME,
  5         LOC
  6 )
```

```
  7 SELECT
  8         "DEPTNO",
  9         "DNAME",
  10        "LOC"
  11* FROM "SYS_SQLLDR_X_EXT_DEPT"


ops$tkyte@ORA10G> /

4 rows created.



ops$tkyte@ORA10G> host ls -l demo1.log_xt

-rw-r--r--  1  ora10g  ora10g  578  Jul 17 10:45  demo1.log_xt
```

可以看到，这一次访问文件时，我成功地加载了 4 行，并创建了日志文件，另外实际上这个日志文件有“Oracle“拥有，而不属于我的操作系统账户。

15.2.2 处理错

理想世界中是没有错误的。输入文件中的数据如果是理想的，就会全部正确地加载。这几乎是不可能的。那么，如何跟踪加载过程的错误呢？

最常用的方法是使用 BADFILE 选项。Oracle 会在这个坏文件中记录所有未能处理的记录。例如，如果我们的控制文件包含一个 DEPTNO ‘ABC’的记录，这个记录会失败，最后出现在坏文件中，因为’ABC’无法转换为一个数字。我们将在下面的例子中展示这一点。

首先，将下面一行添加到 demo1.ctl 中，作为最后一行（这会向输入添加一行无法加载的数据）：

```
ABC,XYZ,Hello
```

接下来，运行以下命令，来证明 demo1.bad 文件尚不存在：

```
ops$tkyte@ORA10G> host ls -l demo1.bad

ls: demo1.bad: No such file or directory
```

然后查询外部表来显示内容：

```
ops$tkyte@ORA10G> select * from SYS_SQLLDR_X_EXT_DEPT;
```

```
DEPTNO     DNAME              LOC

----------     --------------          -------------

        10     Sales              Virginia

        20     Accounting         Virginia

        30     Consulting         Virginia

        40     Finance            Virginia
```

现在，我们发现存在这个坏文件，而且可以获取其内容：

```
ops$tkyte@ORA10G> host ls -l demo1.bad

-rw-r--r--    1       ora10g       ora10g       14 Jul 17 10:53         demo1.bad

ops$tkyte@ORA10G> host cat demo1.bad

ABC,XYZ,Hello
```

但是，如何通过程序来检查这些坏记录以及生成的日志呢？幸运的是，利用另一个外部表就能很容易地做到。假设我们建立了这个外部表：

```
ops$tkyte@ORA10G> create table et_bad

  2 ( text1 varchar2(4000) ,

  3           text2 varchar2(4000) ,

  4           text3 varchar2(4000)

  5 )

  6 organization external

  7 (type oracle_loader

  8           default directory SYS_SQLLDR_XT_TMPDIR_00000

  9           access parameters

 10           (

 11                records delimited by newline
```

```
12              fields
13              missing field values are null
14              ( text1 position(1:4000),
15              text2 position(4001:8000),
16              text3 position(8001:12000)
17          )
18 )
19          location ('demo1.bad')
20 )
21 /
Table created.
```

这是一个可以读取任何文件而不会遭遇数据类型错误的表（只要文件中的行少于12,000 个字符）。如果行多于 12,000 个字符，可以再增加更多的文本列来容纳它们。

可以通过一个简单的查询看到被拒绝的记录：

```
ops$tkyte@ORA10G> select * from et_bad;
TEXT1           TEXT2           TEXT3
--------------      --------------      --------------
ABC,XYZ,Hello
```

COUNT(*)会 告诉我们有多少记录被拒绝。根据与这个外部表相关的日志文件，可以创建另一个外部表，它能告诉我们为什么这些记录被拒绝。不过，我们想更进一步，使得这成 为一个可重复的过程。原因是：如果使用外部表时没有出现错误，坏文件不会“置空“。所以，如果有一些现有的坏文件，其中包含一些数据，尽管这一次使用外部 表时没有生成任何错误，但是看到坏文件中原有的数据，我们可能会被误导，误以为外部表中存在错误。

我过去采用过 3 种方法来解决这个问题：

- 使用 UTL_FILE，并重置坏文件，实际上，就是以写模式打开坏文件，再将其关闭，这样就能清除坏文件中原有的数据。
- 使用 UTL_FILE 重命名现有的坏文件，保留其内容，同时允许创建一个新的坏文件。
- 在坏文件（和日志文件）名中加入 PID。我们将在后面的“多用户问题“一节

中介绍这个方法。

采用这些方法，我们就能区别坏文件中的坏记录是我们刚才生成的，还是这个文件本身以前版本中留下来的（这对我们来说没有意义）。

ALTER TABLE T PROJECT COLUMN REFERENCED|ALL

这一节前面的 COUNT(*)使我想到 Oracle 10g 中的一个新特性：只访问外部文件中在查询中引用的字段来优化外部表访问。也就是说，如果外部表定义为有 100 个数字字段，但是你只选择了其中一个字段，可以指示 Oracle 绕过其他 99 个串转换为数字的过程。听上去很不错，但是可能导致从每个查询返回不同数目的行。假设外部表中有 100 行数据。C1 列的所有行都是“合法“的，都可以转换为一个数字。而 C2 列中的所有数据都不是”合法“的，它们都不能转换为一个数字。如果从这个外部表选择 C1，会返回 100 行。但是如果从这个外部表选择 C2，则会得到 0 行。

必须显式地启用这个优化，而且你要考虑这样使用是否“安全“（只有对你的应用及其处理有足够的了解，才能回答这个”是否安全？“的问题）。还是前面的例子，增加一行坏数据，可以想见，查询外部表时会看到以下输出：

```
ops$tkyte@ORA10G> select dname

2 from SYS_SQLLDR_X_EXT_DEPT

3 /

DNAME

--------------

Sales

Accounting

Consulting

Finance



ops$tkyte@ORA10G> select deptno

2 from SYS_SQLLDR_X_EXT_DEPT

3 /

DEPTNO
```

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

10

20

30

40
```

我们知道，“坏”记录已经记入 BADFILE。但是，如果只是 ALTER 外部表，并告诉 Oracle 只“投影”（处理）所引用的列，如下：

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT

2 project column referenced

3 /

Table altered.



ops$tkyte@ORA10G> select dname

2 from SYS_SQLLDR_X_EXT_DEPT

3 /

DNAME

--------------

Sales

Accounting

Consulting

Finance

XYZ



ops$tkyte@ORA10G> select deptno
```

```
  2 from SYS_SQLLDR_X_EXT_DEPT

  3 /

  DEPTNO

  ----------

  10

  20

  30

  40
```

从各个查询会得到不同数目的行。输入文件中每一行记录的 DNAME 字段都是合法的，但是 DEPTNO 列则不然。如果没有获取 DEPTNO 列，就不会拒绝 DEPTNO 列非法的这个坏记录，所以结果集有显著改变。

15.2.3 使用外部表加载不同的文件

通常需要在一段时期内使用一个外部表从不同名的文件加载数据。也就是说，我们现在必须加载 file1.dat，下一周再加载 file2.dat，等等。到目前为止，我们一直只是从一个固定的文件名（demo1.dat）加载。如果随后需要从另一个文件（demo2.dat）加载会怎么样呢？

幸运的是，这很容易满足。可以用 ALTER TABLE 命令重新指示外部表的位置设置：

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT

        2 location( 'demo2.dat' );

Table altered.
```

仅此而已。对该外部表的下一个查询就会访问文件 demo2.dat。

15.2.4 多用户问题

在这一节的引言中，我指出过去 3 种情况下外部表可能没有 SQLLDR 那么有用。其中之一就是一种特定的多用户问题。我们刚才看到了如果改变一个外部表的位置，如果使之读取文件 2 而不是文件 1，等等。如果多个用户都试图并发地使用这个外部表，而在各个会话中将其指向不同的文件，就会出现这个多用户问题。

这是不允许的。在任何给定的时刻，外部表会指向一个文件（或一组文件）。如果我登录后，将表修改为指向文件 1，而你几乎同时做了同样的事情，如何我们都查询这个表，就会访问同一个文件。

一般都不会遇到这个问题。外部表不是用来取代“数据库表“；它们只是一种加载数

据的方法，你不能过分倚重外部表，把它们作为你的应用的一部分频繁使用。外部表通常只是 DBA 或开发人员用来加载信息的一个工具，可以一次性地加载，或者按周期复发加载（类似于数据仓库加载）。如果 DBA 要使用同一个外部表向数据库中加载 10 个文件，就不应该顺序地加载它们，也就是说，将外部文件指向文件 1，并处理，再指向文件 2，并处理，如此继续。而应该将外部表指向这两个文件，让数据库来处理：

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT

  2 location( 'file1.dat', 'file2.dat')

  3 /

Table altered.
```

如果需要“并行处理“，这是完全可以做到的，数据库已经有相应的内置功能，这在上一章已经介绍过。

所以，多用户问题只可能是：两个会话试图几乎同时修改文件位置。不过这只是一种需要当心的可能情况，而我不认为你真的会经常遇到这种情况。

另一个多用户问题与坏文件名和日志文件名有关。如果有多个会话在并发地查看同一个外部表，或者在使用并行处理（从某种程度上讲是一个多用户情况），会怎么样呢？如果能按会话聚集这些文件，那该多好，幸运的是，确实可以这样做。可以在文件名中结合以下特殊串：

- %P：PID。
- %a：并行执行服务器代理 ID。为并行执行服务器指定了数字 001、002、003 等。

采用这种方式，每个会话都会生成自己的坏文件和日志文件。例如，如果你在先前的 CREATE TABLE 命令中使用以下 BADFILE 语法：

```
RECORDS DELIMITED BY NEWLINE CHARACTERSET WE8ISO8859P1

BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'demo1_%p.bad'

LOGFILE 'demo1.log_xt'
```

就会看到如下命名的一个文件：

```
$ ls *.bad

demo1_7108.bad
```

不过，如果时间长了，还是可能遇到问题。大多数操作系统上都会重用 PID。所以前面所列的处理错误的技术还是很有用，你要重置你的坏文件；或者如果已经存在坏文件，而且你认为这会带来问题，则要对它重命名。

15.2.5 外部表小结

在这一节中，我们分析了外部表。这是 Oracle9i 及以后版本中的一个新特性，从很大程度上讲，它可以取代 SQLLDR。我们分析了使用外部表的最快捷的方法：使用 SQLLDR 来转换以前使用的控制文件。这里还展示了通过坏文件检测和处理错误的一些技术，最后，我们讨论了有关外部表的一些多用户问题。

下面进入这一章最后一节，我们将讨论如何从数据库卸载数据。

15.3 平面文件卸载

有一件事 SQLLDR 做不了，而且 Oracle 对此没有提供任何命令行工具，这就是以 SQLLDR 或者其他程序可以理解的格式卸载数据。要把数据从一个系统移动到另一个系统，如果没有使用 EXP/IMP 或 EXPDP/IMPDP（用于取代 EXP 和 IMP 的新数据泵），平面卸载就很有用。尽管使用 EXP(DP)/IMP(DP)可以很好地将数据从一个系统移到另一个系统，但要求两个系统都是 Oracle。

注意 HTML DB 提供了一个数据导出特性，作为 SQL Workshop 的一部分。你可以很容易地以一种 CSV 格式导出信息。这对于几 MB 的信息可以很好地工作，但是不适用于数十 MB（或更多）的数据。

我们将开发一个很小的PL/SQL实用程序，用来以一种SQLLDR友好的格式在服务器上卸载数据。另外，Ask Tom网站（http://asktom.oracle.com/~tkyte/flat/index.html）上还提供了用Pro*C和SQL*Plus编写的等价工具。这个PL/SQL实用程序在大多数小规模情况下可以很好地工作，但是使用Pro*C可以得到更好的性能。如果你需要在客户机上生成文件，而不是在服务器上（PL/SQL就会在服务器上创建文件），则Pro*C和SQL*Plus也很有用。

我们创建的包的规范如下：

```
ops$tkyte@ORA10G> create or replace package unloader
  2 AUTHID CURRENT_USER
  3 as
  4 /* Function run -- unloads data from any query into a file
  5                  and creates a control file to reload that
  6                  data into another table
  7
  8          p_query = SQL query to "unload". May be virtually any query.
  9          p_tname = Table to load into. Will be put into control file.
```

```
10          p_mode = REPLACE|APPEND|TRUNCATE -- how to reload the data
11          p_dir = directory we will write the ctl and dat file to.
12          p_filename = name of file to write to. I will add .ctl and .dat
13                  to this name
14          p_separator = field delimiter. I default this to a comma.
15          p_enclosure = what each field will be wrapped in
16          p_terminator = end of line character. We use this so we can unload
17                  and reload data with newlines in it. I default to
18                  "|\n" (a pipe and a newline together) and "|\r\n" on NT.
19                  You need only to override this if you believe your
20                  data will have that sequence in it. I ALWAYS add the
21                  OS "end of line" marker to this sequence, you should not
22          */
23          function run( p_query in varchar2,
24                  p_tname in varchar2,
25                  p_mode in varchar2 default 'REPLACE',
26                  p_dir in varchar2,
27                  p_filename in varchar2,
28                  p_separator in varchar2 default ',',
29                  p_enclosure in varchar2 default '"',
30                  p_terminator in varchar2 default '|' )
31          return number;
32  end;
```

```
33 /

Package created.
```

注意这里使用了 AUTHID CURRENT_USER。这样一来，这个包就可以在数据库上只安装一次，可由任何人用来卸载数据。要卸载数据，只要求一点：对所卸载的表要有 SELECT 权限，另外对这个包有 EXECUTE 权限。如果这里没有使用 AUTHID CURRENT_USER，则需要这个包的所有者在要卸载的所有表上都有直接的 SELECT 权限。

注意 SQL 会以这个例程的调用者的权限执行。不过，所有 PL/SQL 调用都会以所调用例程定义者的权限运行；因此，对于具有这个包执行权限的所有人，都隐含地允许他使用 UTL_FILE 写至一个目录。

包体如下。我们使用 UTL_FILE 来写一个控制文件和一个数据文件。DBMS_SQL 用于动态地处理所有查询。我们在查询中使用了一个数据类型：VARCHAR2(4000)。这说明，如果 LOB 大于 4,000 字节，就不能使用这个方法来卸载 LOB。不过，只需使用 DBMS_LOB.SUBSTR，我们就可以使用这个方法卸载任何最多 4,000 字节的 LOB。另外，由于我们用一个 VARCHAR2 作为惟一的输出数据类型，所以可以处理长度最多 2,000 字节的 RAW（4,000 个十六进制字符），除了 LONG RAW 和 LOB 外，这对其他类型都足够了。另外，如果查询引用了一个非标量属性（一个复杂的对象类型，嵌套表等），则不能使用这个简单的实现。以下是一个 90%可用的解决方案，这说明 90%的情况下它都能解决问题：

```
ops$tkyte@ORA10G> create or replace package body unloader

  2 as

  3

  4

  5         g_theCursor integer default dbms_sql.open_cursor;

  6         g_descTbl dbms_sql.desc_tab;

  7         g_nl varchar2(2) default chr(10);

  8
```

以上是这个包体中使用的一些全局变量。全局游标打开一次，即第一次引用这个包时打开，它会一起打开，直到我们注销。这就不用每次调用这个包时都要得到一个新游标，从而避免相应的开销。G_DESCTBL 是一个 PL/SQL 表，将保存 DBMS_SQL.DESCRIBE 调用的输出。G_NL 是一个换行符。在需要内嵌有换行符的串中会使用这个变量。我们无需针对 Windows 调整这个变量，UTL_FILE 会看到字符串中的 CHR(10)，并自动为我们将其转换为一个回车/换行符。

接下来，我们使用了一个很小的便利函数，它能将字符转换为一个十六进制数。为此使用了内置函数：

```
9

10 function to_hex( p_str in varchar2 ) return varchar2

11 is

12 begin

13          return to_char( ascii(p_str), 'fm0x' );

14 end;

15
```

最后，我们又创建了另一个便利函数 IS_WINDOWS，它会返回 TRUE 或 FALSE，这取决于我们所用的是否是 Windows 平台，如果在 Windows 平台上，行结束符就是一个两字符的串，而大多数其他平台上的行结束符只是单字符。我们使用了内置 DBMS_UTILITY 函数：GET_PARAMETER_VALUE，可以用这个函数读取几乎所有参数。我们获取了 CONTROL_FILES 参数，并查找其中是否存在\，如果有，则说明在 Windows 平台上：

```
16 function is_windows return boolean

17 is

18          l_cfiles varchar2(4000);

19          l_dummy number;

20 begin

21          if (dbms_utility.get_parameter_value( 'control_files', l_dummy, l_cfiles )>0)

22          then

23                    return instr( l_cfiles, '\' ) > 0;

24          else

25                    return FALSE;

26          end if;
```

注意　IS_WINDOWS 函数依赖于 CONTROL_FILES 参数中使用了\。要记住，其中也有可能使用/，但这极为少见。

下面的过程会创建一个控制文件来重新加载卸载的数据，这里使用了 DBMS_SQL.DESCRIBE_COLUMN 生成的 DESCRIBE 表。它会为我们处理有关操作系统的细节，如操作系统是否使用回车/换行符（用于 STR 属性）：

```
28
29 procedure dump_ctl( p_dir in varchar2,
30         p_filename in varchar2,
31         p_tname in varchar2,
32         p_mode in varchar2,
33         p_separator in varchar2,
34         p_enclosure in varchar2,
35         p_terminator in varchar2 )
36 is
37         l_output utl_file.file_type;
38         l_sep varchar2(5);
39         l_str varchar2(5) := chr(10);
40
41 begin
42         if ( is_windows )
43         then
44                 l_str := chr(13) || chr(10);
45         end if;
46
47         l_output := utl_file.fopen( p_dir, p_filename || '.ctl', 'w' );
48
49         utl_file.put_line( l_output, 'load data' );
50                 utl_file.put_line( l_output, 'infile ''' ||
```

```
51              p_filename || '.dat" "str x'" ||
52               utl_raw.cast_to_raw( p_terminator ||
53               l_str ) || """ );
54          utl_file.put_line( l_output, 'into table ' || p_tname );
55          utl_file.put_line( l_output, p_mode );
56          utl_file.put_line( l_output, 'fields terminated by X'" ||
57               to_hex(p_separator) ||
58               '" enclosed by X'" ||
59               to_hex(p_enclosure) || '" ' );
60          utl_file.put_line( l_output, '(' );
61
62          for i in 1 .. g_descTbl.count
63          loop
64               if ( g_descTbl(i).col_type = 12 )
65               then
66                    utl_file.put( l_output, l_sep || g_descTbl(i).col_name ||
67                         ' date "ddmmyyyyhh24miss" ');
68               else
69                    utl_file.put( l_output, l_sep || g_descTbl(i).col_name ||
70                         ' char(' ||
71                         to_char(g_descTbl(i).col_max_len*2) ||' )' );
72               end if;
73               l_sep := ','||g_nl ;
```

```
74          end loop;
75          utl_file.put_line( l_output, g_nl || ')' );
76          utl_file.fclose( l_output );
77 end;
78
```

这是一个简单的函数，会返回一个加引号的串（使用所选择的包围字符作为引号）。注意，串不只是包含字符，倘若串中还存在包围字符，还会把包围字符重复两次，从而保留这些包围字符：

```
79 function quote(p_str in varchar2, p_enclosure in varchar2)
80          return varchar2
81 is
82 begin
83          return p_enclosure ||
84          replace( p_str, p_enclosure, p_enclosure||p_enclosure ) ||
85          p_enclosure;
86 end;
87
```

下面是主函数 RUN。因为这个函数相当大，我会一边列出函数一边解释：

```
88 function run( p_query in varchar2,
89          p_tname in varchar2,
90          p_mode in varchar2 default 'REPLACE',
91          p_dir in varchar2,
92          p_filename in varchar2,
93          p_separator in varchar2 default ',',
94          p_enclosure in varchar2 default '"',
```

```
95          p_terminator in varchar2 default '|' ) return number
96 is
97          l_output utl_file.file_type;
98          l_columnValue varchar2(4000);
99          l_colCnt number default 0;
100         l_separator varchar2(10) default '';
101         l_cnt number default 0;
102         l_line long;
103         l_datefmt varchar2(255);
104         l_descTbl dbms_sql.desc_tab;
105 begin
```

我们将 NLS_DATE_FORMAT 保存到一个变量中，从而在将数据转储到磁盘上时可以把它改为一种保留日期和时间的格式。采用这种方式，我们会保留日期的时间分量。然后建立一个异常块，从而在接收到错误时重置 NLS_DATE_FORMAT：

```
106         select value
107                 into l_datefmt
108         from nls_session_parameters
109         where parameter = 'NLS_DATE_FORMAT';
110
111         /*
112         Set the date format to a big numeric string. Avoids
113         all NLS issues and saves both the time and date.
114         */
115         execute immediate
```

```
116             'alter session set nls_date_format="ddmmyyyyhh24miss" ';
117
118     /*
119     Set up an exception block so that in the event of any
120     error, we can at least reset the date format.
121     */
122     begin
```

接下来，解析并描述这个查询。将 G_DESCTBL 设置为 L_DESCTBL 来“重置“全局表；否则，其中会包含前一个 DESCRIBE 生成的数据，而不只是当前查询生成的数据。一旦完成，再调用 DUMP_CTL 具体创建控制文件：

```
123             /*
124             Parse and describe the query. We reset the
125             descTbl to an empty table so .count on it
126             will be reliable.
127             */
128             dbms_sql.parse( g_theCursor, p_query, dbms_sql.native );
129             g_descTbl := l_descTbl;
130             dbms_sql.describe_columns( g_theCursor, l_colCnt, g_descTbl );
131
132             /*
133             Create a control file to reload this data
134             into the desired table.
135             */
136             dump_ctl( p_dir, p_filename, p_tname, p_mode, p_separator,
```

```
137         p_enclosure, p_terminator );
138
139         /*
140         Bind every single column to a varchar2(4000). We don't care
141         if we are fetching a number or a date or whatever.
142         Everything can be a string.
143         */
```

现在可以将具体数据转储到磁盘上了。首先将每个列定义为 VARCHAR2(4000)来获取数据。所有类型（NUMBER、DATE、RAW）都要转换为 VARCHAR2。在此之后，执行查询来准备获取：

```
144         for i in 1 .. l_colCnt loop
145             dbms_sql.define_column( g_theCursor, i, l_columnValue, 4000);
146         end loop;
147
148         /*
149         Run the query - ignore the output of execute. It is only
150         valid when the DML is an insert/update or delete.
151         */
```

现在打开数据文件准备写，从查询获取所有行，并将其打印到数据文件：

```
152         l_cnt := dbms_sql.execute(g_theCursor);
153
154         /*
155         Open the file to write output to and then write the
156         delimited data to it.
157         */
```

```
158         l_output := utl_file.fopen( p_dir, p_filename || '.dat', 'w',
159                 32760 );
160         loop
161                 exit when ( dbms_sql.fetch_rows(g_theCursor) <= 0 );
162                 l_separator := '';
163                 l_line := null;
164                 for i in 1 .. l_colCnt loop
165                         dbms_sql.column_value( g_theCursor, i,
166                           l_columnValue );
167                         l_line := l_line || l_separator ||
168                           quote( l_columnValue, p_enclosure );
169                         l_separator := p_separator;
170                 end loop;
171                 l_line := l_line || p_terminator;
172                 utl_file.put_line( l_output, l_line );
173                 l_cnt := l_cnt+1;
174         end loop;
175         utl_file.fclose( l_output );
176
```

最后，将日期格式设置回原来的样子（如果先前的代码由于某种原因失败了，异常块也会做这个工作），并返回：

```
177         /*
178         Now reset the date format and return the number of rows
179         written to the output file.
```

```
180             */
181             execute immediate
182                     'alter session set nls_date_format='' || l_datefmt || '';
183             return l_cnt;
184             exception
185             /*
186             In the event of ANY error, reset the data format and
187             re-raise the error.
188             */
189                     when others then
190                             execute immediate
191                               'alter session set nls_date_format='' || l_datefmt || '';
192                             RAISE;
193             end;
194     end run;
195
196
197 end unloader;
198 /
Package body created.
```

要运行这个代码，可以使用以下命令（要注意，当然以下代码需要你将 SCOTT.EMP 的 SELECT 权限授予某个角色，或者直接授予你自己）：

```
ops$tkyte@ORA10G> set serveroutput on
```

```
ops$tkyte@ORA10G> create or replace directory my_dir as '/tmp';

Directory created.



ops$tkyte@ORA10G> declare
  2          l_rows number;
  3 begin
  4          l_rows := unloader.run
  5                  ( p_query => 'select * from scott.emp order by empno',
  6                    p_tname => 'emp',
  7                    p_mode => 'replace',
  8                    p_dir => 'MY_DIR',
  9                    p_filename => 'emp',
 10                    p_separator => ',',
 11                    p_enclosure => '"',
 12                    p_terminator => '~' );
 13
 14          dbms_output.put_line( to_char(l_rows) ||
 15                    ' rows extracted to ascii file' );
 16 end;
 17 /
14 rows extracted to ascii file

PL/SQL procedure successfully completed.
```

由此生成的控制文件显示如下（注意，括号里粗体显示的数字并不是真的包括在文件中；加上这些数字只是为了便于引用）：

```
load data (1)

infile 'emp.dat' "str x'7E0A'" (2)

into table emp (3)

replace (4)

fields terminated by X'2c' enclosed by X'22' (5)

( (6)

        EMPNO char(44 ), (7)

        ENAME char(20 ), (8)

        JOB char(18 ), (9)

        MGR char(44 ), (10)

        HIREDATE date 'ddmmyyyyhh24miss' , (11)

        SAL char(44 ), (12)

        COMM char(44 ), (13)

        DEPTNO char(44 ), (14)

)
```

关于这个控制文件，要注意以下几点：

- （2）行：使用了 SQLLDR 的 STR 特性。可以指定用什么字符或串来结束一个记录。这样就能很容易地加载有内嵌换行符的数据。串 x’7E0A’只是换行符后面跟一个波浪号“～“。
- （5）行：使用了我们的分隔符和包围符。这里没有使用 OPTIONALLY ENCLOSED BY，因为我们将把原数据中包围字符的所有出现都重复两次，再把每个字段括起来。
- （11）行：使用了一个很大的“数值“日期格式。这有两个作用：可以避免与日期有关的所有 NLS 问题，还可以保留日期字段的时间分量。

从前面的代码生成的原始数据（.dat）文件如下：

```
"7369","SMITH","CLERK","7902","17121980000000","800","","20"~

"7499","ALLEN","SALESMAN","7698","20021981000000","1600","300","30"~
```

```
"7521","WARD","SALESMAN","7698","22021981000000","1250","500","30"~
"7566","JONES","MANAGER","7839","02041981000000","2975","","20"~
"7654","MARTIN","SALESMAN","7698","28091981000000","1250","1400","30"~
"7698","BLAKE","MANAGER","7839","01051981000000","2850","","30"~
"7782","CLARK","MANAGER","7839","09061981000000","2450","","10"~
"7788","SCOTT","ANALYST","7566","19041987000000","3000","","20"~
"7839","KING","PRESIDENT","","17111981000000","5000","","10"~
"7844","TURNER","SALESMAN","7698","08091981000000","1500","0","30"~
"7876","ADAMS","CLERK","7788","23051987000000","1100","","20"~
"7900","JAMES","CLERK","7698","03121981000000","950","","30"~
"7902","FORD","ANALYST","7566","03121981000000","3000","","20"~
"7934","MILLER","CLERK","7782","23011982000000","1300","","10"~
```

.dat 文件中要注意的问题如下：

- ☐ 每个字段都用包围字符括起来。
- ☐ DATE 卸载为很大的数字。
- ☐ 这个文件中的数据行按要求以一个～结束。

现在可以使用 SQLLDR 很容易地重新加载这个数据。你可以向 SQLLDR 命令行增加你认为合适的选项。

如前所述，卸载包的逻辑可以用多种语言和工具来实现。在 Ask Tom 网站上，你会看到这个实例不仅用 PL/SQL 实现（如在此所示），还使用了 Pro*C 和 SQL*Plus 脚本来实现。Pro*C 是最快的实现，总会写至客户工作站文件系统。PL/SQL 是一种很好的通用实现（没有必要在客户工作站上编译和安装），但是总是写至服务器文件系统。SQL*Plus 是一个很好的折衷，可以提供不错的性能，而且可以写至客户文件系统。

15.4 数据泵卸载

Oracle9i 引入了外部表，作为向数据库中读取数据的一种方法。Oracle 10g 则从另一个方向引入了这个特性，可以使用 CREATE TABLE 语句创建外部数据，从而由数据库卸载数据。从 Oracle 10g 起，这个数据从一种专用二进制格式抽取，这种格式称为数据泵格式（Data Pump format），Oracle 提供的 EXPDP 和 IMPDP 工具将数据从一个数据库移动另一个数据库所用的就是这种格式。

使用外部表卸载确实相当容易，就像使用 CREATE TABLE AS SELECT 语句一样简单。首先，需要一个 DIRECTORY 对象：

```
ops$tkyte@ORA10GR1> create or replace directory tmp as '/tmp'
  2 /

Directory created.
```

现在，准备使用一个简单的 SELECT 语句向这个目录中卸载数据，例如：

```
ops$tkyte@ORA10GR1> create table all_objects_unload
  2 organization external
  3 ( type oracle_datapump
  4       default directory TMP
  5       location( 'allobjects.dat' )
  6 )
  7 as
  8       select
  9       *
 10       from all_objects
 11 /

Table created.
```

我有意关闭了 ALL_OBJECTS 视图，因为这是一个相当复杂的视图，有大量的联结和谓词。这个例子显示出，可以使用这个数据泵卸载技术从数据库中抽取任意的数据。我们可以使用谓词或所需的任何技术来抽取一部分数据。

注意　这个例子显示出，我们可以使用这个数据泵卸载技术从数据库中抽取任意的数据。不错，我把这句话又“啰唆“了一遍。从安全的角度看，这样一来，能访问这个信息的人可以很容易地从任何地方”取得“这些信息。对于能够创建 DIRECTORY 对象并写至这些目录的人，你需要控制这些人的访问，另外对物理服务器有必要的访问权限来得到卸载数据的人的访问也 uyao 有所控制。

最后一步是把 allobjects.dat 复制到另一个服务器（可能是一台执行测试的开发主机），并在这个服务器上抽取 DDL 重建这个表：

```
ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE',
```

```
'ALL_OBJECTS_UNLOAD' )
    2 from dual;
DBMS_METADATA.GET_DDL('TABLE','ALL_OBJECTS_UNLOAD')
--------------------------------------------------------------------------------
CREATE TABLE "OPS$TKYTE"."ALL_OBJECTS_UNLOAD"
(                "OWNER" VARCHAR2(30),
                 "OBJECT_NAME" VARCHAR2(30),
                 "SUBOBJECT_NAME" VARCHAR2(30),
                 "OBJECT_ID" NUMBER,
                 "DATA_OBJECT_ID" NUMBER,
                 "OBJECT_TYPE" VARCHAR2(19),
                 "CREATED" DATE,
                 "LAST_DDL_TIME" DATE,
                 "TIMESTAMP" VARCHAR2(19),
                 "STATUS" VARCHAR2(7),
                 "TEMPORARY" VARCHAR2(1),
                 "GENERATED" VARCHAR2(1),
                 "SECONDARY" VARCHAR2(1)
)
ORGANIZATION EXTERNAL
 ( TYPE ORACLE_DATAPUMP
 DEFAULT DIRECTORY "TMP"
                 LOCATION
```

```
        ( 'allobjects.dat'

        )

)
```

这样就能很容易地在另一个数据库上加载这个片段信息，因为只需：

```
SQL> insert /*+ append */ into some_table select * from all_objects_unload;
```

就万事大吉，数据则已经加载。

15.5 小结

在这一章中，我们介绍了数据加载和卸载的许多细节。首先介绍了 SQL*Loader（SQLLDR），并分析了加载定界数据、定宽数据、LOB 等多种基本技术。我们讨论了将这些知识应用于外部表，这是 Oracle9i 及以后版本中引入的一个新特性，可以用来取代 SQLLDR，不过它还是利用了 SQLLDR 的一些技巧。

接下来我们讨论了加载的逆过程，即数据卸载，说明了如何以其他工具（如电子表格等）可以使用的某种格式从数据库中取出数据。在讨论中，我们开发了一个 PL/SQL 实用程序来展示这个过程，它会以 SQLLDR 友好的格式卸载数据，不过可以很容易地修改这个程序来满足我们的需要。

最后，我们介绍了一个新的 Oracle 10g 特性——外部表卸载，利用外部表卸载，可以很容易地从数据库向另一个数据库创建和移动数据片断。